**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 2-3](#_Toc52464057)

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

[Coding Standard 1 4-5](#_Toc52464059)

[Coding Standard 2 6-7](#_Toc52464060)

[Coding Standard 3](#_Toc52464061) 8-9

[Coding Standard 4 10-11](#_Toc52464062)

[Coding Standard 5 12-13](#_Toc52464063)

[Coding Standard 6 14-15](#_Toc52464064)

[Coding Standard 7 16-17](#_Toc52464065)

[Coding Standard 8 18-19](#_Toc52464066)

[Coding Standard 9 20-21](#_Toc52464067)

[Coding Standard 10 22-23](#_Toc52464068)

[Defense-in-Depth Illustration 24](#_Toc52464069)

[Project One 24](#_Toc52464070)

[1. Revise the C/C++ Standards 24](#_Toc52464071)

[2. Risk Assessment 24](#_Toc52464072)

[3. Automated Detection 24](#_Toc52464073)

[4. Automation 25](#_Toc52464074)

[5. Summary of Risk Assessments 26](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 26-27](#_Toc52464076)

[7. Map the Principles 28-29](#_Toc52464077)

[Audit Controls and Management 30](#_Toc52464078)

[Enforcement 30](#_Toc52464079)

[Exceptions Process 30](#_Toc52464080)

[Distribution 31](#_Toc52464081)

[Policy Change Control 31](#_Toc52464082)

[Policy Version History 31](#_Toc52464083)

[Appendix A Lookups 31](#_Toc52464084)

[Approved C/C++ Language Acronyms 31](#_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 | Always validate any data entered by users before being processed into the system. This prevents harmful data, like malicious code, from slipping through and causing issues such as SQL injection or buffer overflows. Input validation helps maintain the integrity of the system and promises only correctly formatted data is accepted. Without validation, simple mistakes or attacks could disrupt operations or compromise security. |
| 1. Heed Compiler Warnings | Compiler warnings should never be ignored since they often highlight potential vulnerabilities in the code. These warnings might indicate insecure practices, logic errors, or inefficiencies that could lead to problems down the road. By addressing these warning early on, developers can prevent many security issues from escalating into more serious vulnerabilities which lead to safer, more reliable software. |
| 1. Architect and Design for Security Policies | Security should be a core consideration during the design phase of any software project. By incorporating security into the architecture, developers can anticipate potential threats and build safeguards from the beginning. This proactive approach makes sure that the system is less vulnerable to attacks and reduces the need for costly fixes later. Building security in at the design phase helps keep the program compliant with best practice and industry standards. |
| 1. Keep It Simple | Complex code is more difficult to maintain, test, and secure. By keeping your code simple and straightforward, you reduce the chances of bugs or vulnerabilities creeping in. Simplicity also makes it easier for other developers to understand and maintain the code which makes future updates and security patches more effective. A simpler system has fewer points of failure and is easier to audit for potential risks. |
| 1. Default Deny | In secure systems, the default setting should be to deny access to all users or services unless explicitly granted. This minimizes the risk of unauthorized access and ensures that only approved actions can be performed. Locking down access by default creates a more secure environment, where permissions are carefully managed and restricted to necessary functions only. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege dictates that users, programs, and processes should have only the minimum access necessary to complete their tasks. This reduces the attack surface and limits the potential damage if an account or system is compromised. Security risks are minimized, and systems become more resilient to attacks when permissions are tightly controlled. |
| 1. Sanitize Data Sent to Other Systems | Any data sent from one system to another should be thoroughly cleaned or “sanitized” to make sure it doesn’t contain harmful code or unexpected inputs. This process prevents security vulnerabilities like cross-site scripting or injection attacks from spreading between systems. Sanitizing data helps maintain trust in the communication between systems and keeps sensitive information secure. |
| 1. Practice Defense in Depth | Rather than relying on a single security measure, defense in depth uses multiple layers of security to protect the systems. If one layer is breached, others are in place to prevent further damage. Even if an attacker gains access through one vulnerability, they are met with additional defenses which make it harder to cause widespread harm. Note: Too many layers create complications... see Rule 4. |
| 1. Use Effective Quality Assurance Techniques | Continuous testing and review of your code are critical for maintaining security. Regular quality assurance checks, such as static analysis, penetration testing, and code reviews, help to identify vulnerabilities early. By making security testing part of your development cycle, you can catch issues before they become serious threats. |
| 1. Adopt a Secure Coding Standard | Adhering to established secure coding standards like the ones from SEI CERT, align the coding process with best practices for security. These standards provide clear guidelines for writing secure, maintainable, and reliable software. By following these guidelines, developers can avoid common security pitfalls and reduce the risk of vulnerabilities in their code. |

### 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** | **Data Type Safety** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | The correct use of data types is critical for preventing errors like buffer overflows, data corruption, and unexpected behavior. Data type mismatches or unsafe conversions can lead to vulnerabilities in memory management, especially in languages like C++ where strict type safety isn’t enforced automatically. Following strict data type rules improves code reliability and security by handling data as expected. |

| **Noncompliant Code** |
| --- |
| In this example, unsafe type conversion is used which leads to potential data loss and undefined behavior.  int largeValue = 300;  char smallType = largeValue; // Potential data loss due to type mismatch |
| This noncompliant code uses an inappropriate data type which can lead to logical errors and inefficient memory usage.  Unsigned int largeNumber = -5; // Logical error since unsigned types cannot store negative values |

| **Compliant Code** |
| --- |
| In this compliant example, the code safely checks the value before converting types.  int largeValue = 300;  if (largeValue <= CHAR\_MAX) {  char smallType = static\_cast<char>(largeValue); // Type cast with boundary check  } |
| In this example, the correct data type is used and handles the negative values properly.  int largeNumber = -5; // Use signed int to properly handle negative numbers |

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

| **Principles(s):**  **Validate Input Data –** This standard ensures that all data types are correctly validated before being used, preventing unintended behavior or vulnerabilities. **Heed Compiler Warnings –** Proper data type usage can eliminate compiler warnings related to type safety and potential memory corruption. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | P2 | L4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [cppcheck](https://sourceforge.net/projects/cppcheck/files/cppcheck/2.6/) | 2.6.4 | Type Safety Checks | Cppcheck identifies unsafe type conversion, uninitialized variables, and other issues that can lead to data type vulnerabilities. |
| [Clang-Tidy](https://releases.llvm.org/12.0.0/tools/clang/tools/extra/docs/clang-tidy/index.html) | 12.0 | Cppcoreguidelines-pro-type-cstyle-cast | This checker detects unsafe or incorrect type conversions, promising that data type safety is maintained throughout the code. |
| SonarQube | 8.9 | Cpp:S5527 | SonarQube’s rule cpp:S5527 focuses on type safety, flagging type mismatches, and potential loss of data due to improper type handling. |
| [Visual Studio Code Analysis](https://visualstudio.microsoft.com/vs/older-downloads/) | 2019 | C26439 | This tool checks for type safety issues, including incorrect type usage and pointer casting which improve the robustness of the code. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Data Value Safety** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Handling data values correctly is essential for your program to operate securely and efficiently. Incorrect data values, such as those that are out of range, can cause crashes, memory corruption, or other undefined behaviors. Data values should be validated to prevent vulnerabilities and promise the program handles unexpected values. |

| **Noncompliant Code** |
| --- |
| This example shows how failing to check the value of an array index leads to out-of-bounds access causing potential memory corruption or crashes.  int arr[10];  int index = 15; // Index is out of bounds  arr[index] = 100; // Potential memory corruption or crash |
| In this noncompliant code, an integer overflow occurs that leads to incorrect calculations and potential vulnerabilities.  int largeValue = INT\_MAX;  int result = largeValue + 10; // Causes overflow and wraps around |

| **Compliant Code** |
| --- |
| In the compliant example, the array index is checked to make sure it is within bounds and prevents out-of-bounds access.  int arr[10];  int index = 15;  if (index >= 0 && index < 10) {  arr[index] = 100; // Safe access after bounds check  } |
| The compliant code checks for overflow before performing the calculation keeping the result valid and avoiding unexpected behavior.  int largeValue = INT\_MAX;  int result;  if (largeValue <= INT\_MAX – 10) {  result = largeValue + 10; // Safe addition with overflow check  } |

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

| **Principles(s):**  **Validate Input Data –** This principle applies as data values safety involves verifying that values are within acceptable ranges to prevent logic errors or unexpected behavior.  **Architect and Design for Security Policies –** Planning for value safety at the design stage helps prevent vulnerabilities like integer overflows and underflows. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Medium | Medium | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.6.4 | Value Range Analysis | This tool identifies when variables values exceed their expected ranges or lead to potential overflows, flagging data value issues early in the development process. |
| [Clang Static Analyzer](https://releases.llvm.org/) | 12.0 | Core.CallAndMessage | Checks for incorrect value handling and flagging risky operations that can lead to value corruption. |
| [Coverity](https://sig-docs.synopsys.com/polaris/topics/r_coverity-compatible-platforms_2021.03.html) | 2021.03 | SV.UNEXPECTED\_VALUE | Coverity’s checker focuses on identifying values that deviate from expected patterns, catching errors like out-of-range values and logic flaws. |
| [Polyspace](https://www.mathworks.com/products/polyspace.html) | R2021a | Value Boundary Checks | Polyspace uses value boundary analysis to identify cases where values may go beyond defined limits, reducing the risk of data corruption and undefined behavior. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **String Handling Safety** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Handling strings in C++ is a common source of vulnerabilities, especially with functions that do not check the length of input. Problems like buffer overflows, memory corruption, or unexpected crashes can occur when strings are not properly managed. To prevent these issues, it is important that string operations are safe and well-bounded. |

| **Noncompliant Code** |
| --- |
| This noncompliant code uses the unsafe strcpy function, which does not check the length of the source string before copying it to the destination that leads to potential buffer overflow.  char dest[10];  const char\* src = “This is a long string”;  strcpy(dest, src); // Potential buffer overflow |
| This noncompliant example manually changes a dynamic string without proper bounds checking, potentially causing a buffer overflow if the string grows too large.  char\* dynamicStr = new char[10];  strcat(dynamicStr, “Adding a long string here”); // No check for buffer overflow |

| **Compliant Code** |
| --- |
| The compliant version uses strncpy which limits the number or characters copied to prevent buffer overflow.  char dest[10];  const char\* src = “This is a long string”;  strncpy(dest, src, sizeof(dest) – 1); // Safely copy with buffer size limit  dest[sizeof(dest) = 1] = ‘\0’; // Validate null termination |
| In the compliant version, std::string is used which handles memory management automatically and prevents buffer overflows by resizing as needed.  std::string dynamicStr = “Initial string”;  dynamicStr += “ Adding a long string here”; // Safe concatenation without risk of buffer overflow |

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

| **Principles(s):** **Sanitize Data Sent to Other Systems –** String correctness is essential to avoid passing corrupted or malicious data, especially when transferring strings between systems.  **Keep It Simple –** Proper sting management simplifies code, making it less prone to errors like buffer overflows and memory corruption. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | P2 | L4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.6.4 | Buffer Overrun Analysis | Detects buffer overflows and string handling errors by analyzing the size and boundaries of string operations. |
| Clang-Tidy | 12.0 | Cert-str34-c | Focuses on identifying unsafe string operations, such as improper use of strcpy and strncpy functions, to prevent buffer overflow. |
| [Valgrind](https://valgrind.org/index.html) | 3.17.0 | Memcheck | Detects invalid memory accesses, use of initialized string variables, and improper memory allocation related to strings. |
| [Flawfinder](https://dwheeler.com/flawfinder/) | 2.0.10 | String-safety | Scans C/C++ code for common string vulnerabilities, such as buffer overflows and unsafe function calls, highlighting dangerous string manipulation patterns. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **SQL Injection Prevention** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | SQL injection prevention is a serious vulnerability where attackers can execute arbitrary SQL code on a database by exploiting improperly sanitized user inputs. This can lead to unauthorized data access, data corruption, or even full system compromise. Preventing SQL injection by using parameterized queries and input sanitization is critical for guaranteeing database security. |

| **Noncompliant Code** |
| --- |
| This example uses string concatenation to build an SQL query which is highly vulnerable to SQL injection when user input is directly inserted into the query.  std::string userInput = “1 OR 1=1”; // Attacker input  std::string query = “SELECT \* FROM users WHERE id =’” + userInput + “’”; // Unsafe concatenation  executeQuery(query); // Vulnerable SQL injection |
| This noncompliant example directly includes user input in an SQL INSERT statement without validating or sanitizing it which exposes the system to SQL injection attacks.  std::string username = “admin’--“; // Attacker input to comment out the rest of the query  std::string query = “INSERT INTO users (name) VALUES (‘” + username +”’); // Vulnerable to SQL injection  executeQuery(query); |

| **Compliant Code** |
| --- |
| The compliant version uses a parameterized query where user input is treated as a parameter rather than a part of the SQL command which prevents the SQL injection.  std::string userInput = “1 OR 1=1”;  std::string query = “SELECT \* FROM users WHERE id = ?”;  prepareStatement(query);  bindParameter(1, userInput); // User input is safely bound as a parameter  executeQuery(); // Safe from SQL injection |
| In the compliant example, the code uses prepared statements with bind parameters to make sure the user input is treated as data and not executable SQL code.  std::string username = “admin’--";  std::string query = “INSERT INTO users (name) VALUES (?)”;  prepareStatement(query);  bindParameter(1, username); // User name is bound safely  executeQuery(); // Safe from SQL injection |

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

| **Principles(s):** **Sanitize Data Sent to Other Systems –** Preventing SQL injection relies on sanitizing user input before passing it into SQL queries. **Adhere to the Principle of Least Privilege –** Minimizing the scope of SQL commands and restricting database permissions reduces the impact of successful injection attempts.  **Default Deny –** Setting access control defaults to deny ensures that only authorized SQL command and queries are executed. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | High | High | P1 | L5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Spp:S3649 | Identifies SQL injection vulnerabilities by analyzing SQL query construction and highlighting unsafe use of concatenated strings. |
| OWASP ZAP | 2.10.0 | Active and Passive SQL Injection Checks | This tool performs both active and passive scans to detect SQL injection vulnerabilities in applications, highlighting unsafe query construction. |
| [Fortify Static Code Analyzer](https://www.microfocus.com/documentation/fortify-static-code-analyzer-and-tools/2020/) | 20.2.2 | SQL Injection: Unsafe Query Construction | Fortify SCA identifies SQL injection risks by tracking the flow of untrusted input into SQL commands, ensuring the only sanitized inputs are used. |
| Visual Studio Code Analysis | 2019 | C6385 | Detects potential SQL injection vulnerabilities by identifying unsafe user input handling in SQL commands, flagging risky patterns. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Memory Management and Protection** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Managing memory correctly is critical in C++ to avoid vulnerabilities like memory leaks, buffer overflows, and use-after-free errors. These problems can cause crashes, security breaches, or even complete system compromises. Correct handling of memory allocation and deallocation helps to prevent unintended behavior or exploitation by malicious actors. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, memory is dynamically allocated but not released which results in a memory leak and can degrade system performance over time.  int\* ptr = new int[100];  // Code that uses ptr…  // Memory is never released |
| This example demonstrates a use-after-free vulnerability where the program tries to access memory that has already been deallocated which can lead to undefined behavior or security risks.  int\* ptr = new int[10];  delete[] ptr; // Memory is freed  ptr[0] = 42; // Accessing memory after it has been deallocated |

| **Compliant Code** |
| --- |
| In the compliant version, memory is correctly allocated and released when no longer needed, preventing memory leaks.  int\* ptr = new int[100];  // Code that uses ptr…  delete[] ptr; // Memory is released after use |
| In the compliant code, the pointer is set to nullptr after the memory is deallocated which prevents accidental access to freed memory.  int\* ptr = new int[10];  delete[] ptr; // Memory is freed  ptr = nullptr; // Pointer is set to nullptr to avoid further access |

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

| **Principles(s):** **Architect and Design for Security Policies –** Proper planning during design is crucial for preventing memory management issues such as buffer overflows, use-after-free, and memory leaks.  **Practice Defense in Depth –** Using multiple safeguards, such as bounds checking and automated memory management tools, helps reduce the risk of memory vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | High | P2 | L4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17.0 | Memcheck | Detects memory management issues such as invalid memory access, memory leaks, and use-after-free errors. |
| AddressSanitizer (ASan) | 12.0 | Address Bounds Checker | Identifies buffer overflows, invalid pointer uses, and memory corruption, providing detailed reports on memory safety violations. |
| Clang Static Analyzer | 12.0 | Unix.Malloc | Check for common memory management mistakes such as double-free, memory leaks, and invalid deallocations. |
| Coverity | 2021.03 | USE\_AFTER\_FREE | Detects use-after-free vulnerabilities, null pointer dereferences, and other memory safety issues. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Proper Use of Assertions** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions are useful during development for verifying assumptions and catching errors early; however, improper use of assertions can lead to undesired behavior and security vulnerabilities. Leaving the assertion in production code or using them to handle runtime errors are a couple of examples. Assertions should only be used for conditions that should never fail under normal circumstances and not for handling external input or critical runtime checks. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, an assertion is used to validate user input which should be handled with proper error handling mechanisms instead. If assertions are disabled in production, this input validation would be skipped.  int getUserInput(int input) {  assert(input >0); // Improper use of assertion for user input validation  return input;  } |
| This noncompliant example places an assertion in production code to check for a condition that could realistically fail due to external factors which could crash the program if triggered.  void processData(int\* data) {  assert(data != nullptr); // This check should be handled by proper error checking not an assertion  // Process data…  } |

| **Compliant Code** |
| --- |
| The compliant version uses proper error handling instead of an assertion which allows the code to handle invalid inputs even when assertions are disabled.  int getUserInput(int input) {  if(input <= 0) {  throw std::invalid\_argument(“Input must be greater than 0”); // Proper error handling  }  return input;  } |
| The compliant version checks for null pointers using proper error handling instead of an assertion which allows the code to handle potential errors without relying on assertions in production.  void processData(int\* data) {  if(data == nullptr) {  throw std::runtime\_error(“Null pointer encountered”); // Proper error handling for null pointers  }  // Process data…  } |

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

| **Principles(s):**  **Heed Compiler Warnings –** Improper use of assertions often triggers compiler warnings, signaling potential logic errors or misuse of assertions in production code.  **Architect and Design for Security Policies –** Proper use of assertions should be part of the design to validate assumptions during development without impacting production code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Medium | Low | P5 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0 | Cert-err34-c | Flags improper use of assertions, such as assertions used for error handling or runtime validation in production code. |
| Cppcheck | 2.6.4 | Assert Usage Analysis | Detects improper assertion usage and helps identify cases where assertions should be replaced with proper error handling. |
| [PVS-Studio](https://pvs-studio.com/en/pvs-studio/download/) | 7.13 | V590 | Identifies problematic assertions that can lead to unintended program behavior, ensuring they are used correctly in development. |
| Visual Studio Code Analysis | 2019 | C26472 | Highlights misuse of assertions, such as using assertions in production builds, which could lead to undefined behavior or system crashes. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Exception Handling** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Exceptions are a powerful tool for managing runtime errors in C++, but they need to be used carefully to avoid creating unpredictable or insecure behavior. Properly structured exception handling helps programs recover from errors, while unhandled or misused exceptions can lead to program crashes, security vulnerabilities, or resource leaks. Effective exception handling makes code more robust and secure. |

| **Noncompliant Code** |
| --- |
| In this example, the code fails to catch and handle exceptions properly which can result in unhandled exceptions that crash the program.  void openFile(const std::string& filename) {  std::ifstream file(filename);  file.open(); // If the file doesn’t exist, an exception will be thrown, but it is not handled  // Perform file operations…  } |
| This example shows a function that throws exceptions but does not clean up allocated resources properly that could lead to memory leaks if an exception is thrown.  void processData() {  int\* data = new int[100]; // Dynamically allocated memory  if (someConditionFails()) {  throw std::runtime\_error(“Error occurred”); // Exception throw but memory is not cleaned up  }  delete[] data; // Never reached if an exception is thrown  } |

| **Compliant Code** |
| --- |
| In this compliant example, the code includes a try-catch block to handle exceptions when opening a file that allows the program to recover or report errors without crashing.  void openFile(const std::string& filename) {  try {  std::ifstream file(filename);  file.open();  // Perform file operations…  }  catch( const std::exception& e) {  std::cerr << “Error: Unable to open file: “ << e.what() << std::endl; // Exception handling  }  } |
| The compliant version uses RAII (Resource Acquisition Is Initialization) to manage resources so the memory is freed even if an exception occurs.  void processData() {  std::vector<int> data(100); // Automatically managed memory via RAII  if (someConditionFails()) {  throw std::runtime\_error(“Error occurred”); // Memory is still cleaned up when exception is thrown  }  // No need for manual cleanup  } |

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

| **Principles(s):**  **Heed Compiler Warnings –** Proper exception handling reduces compiler warnings and signals correct handling of errors and unexpected situations.  **Use Effective Quality Assurance Techniques –** Testing exception handling paths ensures that exceptions are managed correctly, preventing crashes and data loss. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Low | Medium | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0 | Alpha.unix.cstring.InvalidArg | Detects incorrect handling of exceptions, such as catching exceptions by value instead of reference, leading to potential performance or logic issues. |
| Cppcheck | 2.6.4 | Exception Safety Analysis | Identifies missing or incorrect try-catch blocks, guaranteeing that exceptions are properly handled and not ignored. |
| PVS-Studio | 7.13 | V601 | Flags catch blocks that are too broad or vague, which can cause logic errors or mask real problems. |
| Visual Studio Code Analysis | 2019 | C26440 | Highlights common exception handling issues, such as throwing exceptions from destructors or using exceptions to manage program flow. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Pointer Safety** |
| --- | --- | --- |
| Pointer Safety | [STD-008-CPP] | Mismanagement of pointers can lead to critical issues such as dangling pointers, memory leaks, and crashes. C++ allows for direct memory manipulation, which makes it powerful but also prone to errors. Proper pointer management can prevent serious vulnerabilities and improve program stability. |

| **Noncompliant Code** |
| --- |
| This example demonstrates a dangling pointer issue where memory is deallocated, but the pointer still attempts to access it which leads to undefined behavior.  int\* ptr – new int(5);  delete ptr; // Memory is freed  std::cout << \*ptr; // Dangling pointer dereference |
| This noncompliant example shows the use of raw pointers without proper memory management, leading to a potential memory leak if the memory is not manually freed.  int\* ptr = new int(10);  // Forgetting to call delete leads to a memory leak |

| **Compliant Code** |
| --- |
| In this compliant example, the pointer is set to nullptr after deallocation to prevent further dereferencing of invalid memory.  int\* ptr = new int(5);  delete ptr; // Memory is freed  ptr = nullptr; // Pointer is set to nullptr to avoid dangling pointer dereference |
| The compliant version replaces raw pointers with std::unique\_ptr preventing memory leaks. The smart pointer automatically handles memory allocation and deallocation.  std::unique\_ptr<int> ptr = std::make\_unique<int>(10); // Automatically deallocated when out of scope |

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

| **Principles(s):** **Architect and Design for Security Policies –** Proper pointer management should be a priority in the design phase to prevent vulnerabilities like null pointer dereferencing and memory corruption.  **Practice Defense in Depth –** Using multiple safety mechanisms such as smart pointers and automated analysis tools reduces the risk of pointer-related vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | P2 | L4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AddressSanitizer (ASan) | 12.0 | Pointer Safety Checker | Detects invalid pointer usage, out-of-bounds pointer access, and use-after-free errors, providing detailed diagnostics. |
| Valgrind | 3.17.0 | Memcheck | Identifies pointer-related issues such as dangling pointers, null pointer dereferencing, and memory corruption. |
| Clang-Tidy | 12.0 | Cppcoreguidelines-pro-bounds-pointer-arithmetic | Highlights unsafe pointer arithmetic and access patterns that can lead to vulnerabilities in pointer management. |
| PVS-Studio | 7.13 | VS12 | Flags dangerous pointer manipulations and dereferences that could result in undefined behavior or program crashes. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Dead Code Elimination** |
| --- | --- | --- |
| Dead Code | [STD-009-CPP] | Dead code refers to parts of the program that are never executed or no longer needed. Leaving dead code in a project increases the maintenance burden, makes the code more difficult to read, and can introduce hidden security risks. Removing dead code improves performance, reduces the size of the executable, and simplifies future maintenance while ensuring the codebase is efficient and free of unnecessary complexity. |

| **Noncompliant Code** |
| --- |
| This example includes a function that is never called which leads to dead code increasing complexity and maintenance overhead.  void unusedFunction() {  // This function is never called within the program  std::cout << “This is dead code”;  } |
| This noncompliant example demonstrates a conditional block that is never executed due to a constant condition that results in dead code in the program.  int main() {  if (false) { // This block is never executed  std::cout << “This will never be printed”; // Dead code  }  } |

| **Compliant Code** |
| --- |
| In the compliant example, the unused function is removed from the codebase, simplifying the program and reducing unnecessary clutter.  // The unusedFunction() has been removed since it is not needed |
| The compliant version removes the dead code which makes a cleaner, more maintainable codebase.  int main() {  // The unreachable block has been removed  } |

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

| **Principles(s):**  **Keep It Simple –** Removing dead code simplifies the codebase which makes it easier to read, maintain, and less error-prone.  **Use Effective Quality Assurance Techniques –** Dead code can hide vulnerabilities or introduce unexpected behavior. Regular analysis and testing help identify and remove such code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Low | Low | P6 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.6.4 | Dead Code Analysis | Identifies unused functions, variables, and unreachable code blocks that can be safely removed. |
| Clang-Tidy | 12.0 | Misc-unused-variables | Detects variables and functions that are declared but never used, helping to eliminate dead code. |
| SonarQube | 8.9 | Cpp:S1068 | Identifies dead code such as unused local variables and unused function parameters, simplifying the codebase. |
| Visual Studio Code Analysis | 2019 | C26451 | Detects dead code segments and redundant computations, helping reduce code bloat and complexity. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Global Variable Minimization** |
| --- | --- | --- |
| Global Variables | [STD-010-CPP] | Global variables can be assessed and modified from anywhere in a program, which increases the risk of unintended side effects, bugs, and security vulnerabilities. Excessive use of global variables also makes code harder to maintain and test. Minimizing or eliminating global variables promotes better encapsulation, reduces the chance of accidental data modification, and improves the overall maintainability and security of the program. |

| **Noncompliant Code** |
| --- |
| This example uses a global variable that can be modified from any part of the program which might lead to unintended side effects making the code harder to maintain.  int globalCounter = 0;  void incrementCounter() {  globalCounter++; // Global variable modified  } |
| This noncompliant example uses a global variable to store configuration data, making the program more prone to unintended modifications and side effects from different parts of the code.  std::string config = “default”;  void updateConfig(const std::string& newConfig) {  config = newConfig; // Global variable modified from anywhere in the program  } |

| **Compliant Code** |
| --- |
| In this compliant version, the global variable is removed, and the variable is passed to functions as a parameter which promotes better encapsulation and control over the state.  void incrementCounter(int& counter) {  counter++; // Counter passed by reference avoiding global variables  } |
| The compliant code encapsulates the configuration data within a class that provides controlled access through methods and reduces the risk of unintended modifications.  class Config {  std::string config;  public:  void setConfig(const std::string& newConfig) {  config = newConfig; // Controlled access to the config variable  }  std::string getConfig() const {  return config;  }  }; |

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

| **Principles(s):**  **Adhere to the Principle of Least Privilege –** Using local variables instead of global variables limits the scope of data access, reducing unintended interactions and minimizing the risk of unintended data modification.  **Architect and Design for Security Policies –** Proper design minimizes reliance on global variables, encouraging encapsulation and better modularity. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Medium | Medium | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.6.4 | Global Variable Usage Analysis | Detects instances of global variable usage, helping identify areas where refactoring is needed to localize variables. |
| Clang-Tidy | 12.0 | Misc-no-recursion | Detects recursive functions that use global variables, highlighting risky patterns that can lead to hard-to-track bugs. |
| PVS-Studio | 7.13 | V1001 | Flags uses of global variables, especially in multi-threaded contexts, to prevent race conditions and unexpected data modifications. |
| SonarQube | 8.9 | Cpp:S1039 | Identifies global variable declarations and highlights areas where encapsulation can be improved to limit scope and access. |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

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

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### 

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

The DevSecOps diagram is a valuable tool for understanding the flow of security integration across the software development lifecycle. Its structure provides a solid foundation, allowing for minimal changes to be made while maintaining effective security. The focus is on enhancing existing phases rather than overhauling the entire process.

After reviewing each stage, modifications were deemed necessary in the Code & Build, Test, and Release & Deploy phases to automate enforcement of the coding standards defined in this policy. During the Code & Build phase, static analysis tools such as Cppcheck and Clang-Tidy will be integrated into pre-commit hooks to catch coding standard violations early. In the Test phase, tools like Valgrind and Polyspace will be used to perform in-depth memory and pointer safety analysis, identifying potential vulnerabilities before the code progresses. Before deployment, a final quality gate using Fortify and SonarQube will analyze the entire codebase to ensure compliance with critical standards.

Other phases, such as Plan & Design and Monitor, were reviewed and found to have adequate security measures already in place. As a result, no modifications were deemed necessary in these areas, which keeps the automation process focused and efficient. By strategically enhancing key stages, we can strengthen security enforcement without disrupting the overall flow.

### 

### 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 | Medium | Low | P2 | L4 |
| STD-002-CPP | Medium | Medium | Medium | P3 | L3 |
| STD-003-CPP | High | High | Medium | P2 | L4 |
| STD-004-CPP | Critical | High | Medium | P1 | L5 |
| STD-005-CPP | High | Medium | High | P2 | L4 |
| STD-006-CPP | Low | Medium | Low | P5 | L2 |
| STD-007-CPP | Medium | Low | Medium | P4 | L3 |
| STD-008-CPP | High | Medium | Medium | P2 | L4 |
| STD-009-CPP | Low | Low | Low | P6 | L1 |
| STD-010-CPP | Medium | Medium | Medium | P3 | L3 |

### 

### 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 all sensitive data stored in databases, files, or backups using strong encryption algorithms like AES-256. It automatically encrypts data when saved to disk and decrypts when read. It applies to any stored data not being actively processed to protect against unauthorized access. |
| Encryption in flight | Secures data during transmission between systems using TLS or similar encryption protocols. It encrypts data traveling over networks to prevent interception. It’s used whenever data moves between applications, servers, or over public networks. |
| Encryption in use | Protects sensitive data while being processed in memory using secure enclaves like Intel SGX. It encrypts data temporarily stored in memory, especially in untrusted or multi-tenant environments. It’s applied when data is being actively used or processed. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Verifies the identity of users and systems before granting access. It uses credentials like passwords and MFA for secure access. It’s applied to all systems that require identity verification. |
| Authorization | Grants access to resources based on roles and permissions. It implements role-based access control to limit access rights. It’s enforced when users request access to sensitive resources. |
| Accounting | Logs all access and actions for auditing and monitoring. It’s used to secure logs and monitoring tools to track activity. It is applied to all critical systems for maintaining accountability. |

**\***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.

* **Principle 1: Validate Input Data**
  + Data Type Safety (STD-001-CPP): Validating input data prevents unexpected behaviors due to type mismatches that could lead to crashes or undefined behavior.
  + Data Value Safety (STD-002-CPP): Verifying data values reduces the risk of overflow, underflow, or incorrect calculations so only appropriate values are processed.
  + String Correctness (STD-003-CPP): Input validation is crucial for strings to avoid buffer overflows and injection vulnerabilities.
  + SQL Injection Prevention (STD-004-CPP): Proper input validation prevents SQL injection attacks by ensuring that untrusted data is not directly used in query construction.
* **Principle 2: Heed Compiler Warnings**
  + Data Type Safety (STD-001-CPP): Compiler warnings often indicate type mismatches or unsafe operations that need to be addressed early.
  + Assertions (STD-006-CPP): Addressing warning related to assertions make sure that they are used correctly for development and testing purposes.
  + Exceptions (STD-007-CPP): Compiler warnings for missing or improperly handled exceptions must be resolved to ensure stability and predictable exception handling.
* **Principle 3: Architect and Design for Security Policies**
  + Memory Protection (STD-005-CPP): Planning for safe memory management, like avoiding buffer overflows and double frees, begins during the design phase.
  + Pointer Safety (STD-008-CPP): Designing for pointer safety includes using smart pointers and avoiding risky pointer arithmetic.
  + Global Variable Minimization (STD-010-CPP): Avoiding global variables promotes encapsulation and secure design which reduces unintended data access.
* **Principle 4: Keep It Simple**
  + Assertions (STD-006-CPP): Keeping assertions straightforward means they are not misused in production code and are easy to test and maintain.
  + Dead Code Elimination (STD-009-CPP): Removing unnecessary codes reduces complexity, making the codebase easier to review and less error-prone.
* **Principle 5: Default Deny**
  + SQL Injection Prevention (STD-004-CPP): Setting database permissions to deny access by default reduces the impact of a successful SQL injection, as attackers will have limited permissions.
* **Principle 6: Adhere to the Principle of Least Privilege**
  + SQL Injection Prevention (STD-004-CPP): Applying least privilege guarantees that database users will only have access to the data that they need, which minimizes the risk if an account is compromised.
  + Global Variable Minimization (STD-010-CPP): Reducing the use of global variables restricts data access and enforces encapsulation, minimizing unintended interactions between components.
* **Principle 7: Sanitize Data Sent to Other Systems**
  + String Correctness (STD-003-CPP): Sanitizing strings before using them in other systems prevents malformed inputs and injection attacks.
  + SQL Injection Prevention (STD-004-CPP): Input sanitization before construction SQL queries helps prevent injection attacks that could manipulate database contents.
* **Principle 8: Practice Defense in Depth**
  + String Correctness (STD-003-CPP): Using multiple safeguards, like input validation and buffer size checks, provides layered protection against string manipulation vulnerabilities.
  + Memory Protection (STD-005-CPP): Combining memory management tools and runtime checks adds layers of security to catch memory issues at various stages.
  + Pointer Safety (STD-008-CPP): Implementing multiple safeguards like static analysis, bounds checking, and smart pointers reduces the risk of pointer-related vulnerabilities.
* **Principle 9: Use Effective Quality Assurance Techniques**
  + Data Safety Value (STD-002-CPP): Boundary testing and QA methods make sure that data values conform to expectations, reducing the risk of incorrect processing.
  + Exceptions (STD-007-CPP): QA techniques help test exception paths to verify correct exception handling and prevent crashes.
  + Dead Code Elimination (STD-009-CPP): Code Reviews and static analysis detect dead code that can be safely removed to improve maintainability.
* **Principle 10: Adopt a Secure Coding Standard**
  + All standards incorporate secure coding guidelines to prevent common vulnerabilities. This principle applies universally across the coding standards. Adopting a secure coding standard ensures that the guidelines followed in each standard align with SEI CERT C++ rules, promoting consistent and secure code development.

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 09/18/2024 | Module Three Requirements | Justin Hancock |  |
| 1.2 | 10/10/2024 | Project One Requirements | Justin Hancock |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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