**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Ensure that all input received by a system is checked and sanitized before processing. This prevents malicious data from exploiting vulnerabilities, such as SQL injection or buffer overflows, which could compromise the system's integrity. |
| 1. Heed Compiler Warnings | Pay attention to warnings issued by compilers during code compilation. These warnings often highlight potential security issues or coding errors that could lead to vulnerabilities if left unaddressed. |
| 1. Architect and Design for Security Policies | Integrate security principles into the system’s architecture and design phase. This involves planning for security from the outset, including defining and implementing security policies to protect against threats and vulnerabilities. |
| 1. Keep It Simple | Design systems and applications to be as simple as possible. Complexity often leads to hidden vulnerabilities and makes systems harder to maintain and secure. Simple designs are easier to understand, audit, and protect. |
| 1. Default Deny | By default, deny access to resources or operations unless explicitly allowed. This principle ensures that only authorized actions are permitted, reducing the risk of unauthorized access and potential exploitation. |
| 1. Adhere to the Principle of Least Privilege | Grant users and processes only the minimum level of access necessary to perform their functions. Limiting privileges helps reduce the potential impact of security breaches and limits the extent of damage if an account or process is compromised. |
| 1. Sanitize Data Sent to Other Systems | Before transmitting data to other systems, ensure it is properly sanitized to avoid introducing vulnerabilities or malware. This includes validating and cleaning data to prevent issues such as cross-site scripting (XSS) or injection attacks. |
| 1. Practice Defense in Depth | Employ multiple layers of security measures to protect systems. This strategy ensures that if one layer is compromised, other layers will still provide protection, reducing the overall risk and improving resilience against attacks. |
| 1. Use Effective Quality Assurance Techniques | Implement robust quality assurance practices, including regular testing, code reviews, and static analysis, to identify and fix security vulnerabilities. Effective QA helps ensure that security issues are caught early in the development process. |
| 1. Adopt a Secure Coding Standard | Follow established secure coding practices and guidelines to prevent common vulnerabilities and security issues. Adopting a secure coding standard helps maintain consistency and promotes best practices in developing secure software. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Using the correct data type is essential for ensuring that variables hold appropriate values, that operations are performed accurately, and that the system behaves predictably. Correct data types help prevent issues like overflow, loss of precision, and undefined behavior, leading to more reliable and maintainable code. |

| **Noncompliant Code** |
| --- |
| This code attempts to compare an unsigned int with a negative value, which is logically incorrect and can lead to misleading results or compiler warnings. |
| void displayValue(unsigned int value) {  if (value < 0) { // Comparison with a negative value  std::cout << "Negative value: " << value << std::endl;  }  } |

| **Compliant Code** |
| --- |
| This code uses int for the variable value and performs a valid comparison with a negative value, avoiding the issues associated with comparing unsigned int with negative numbers. |
| void displayValue(unsigned int value) {  if (value < 0) { // Comparison with a negative value  std::cout << "Negative value: " << value << std::endl;  }  } |

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

| **Principles(s):** Validate input Data(1)  This principle maps to the standard because using the correct data type ensures that input data is correctly processed and interpreted by the system. For example, comparing an unsigned int with a negative value can lead to incorrect results and potential vulnerabilities. Ensuring that data types align with expected input values prevents logical errors and enhances system reliability. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| high | likely | low | high | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 LTS | Cpp (Null pointers should not be dereferenced) | SonarQube can be configured to check for data type issues, such as inappropriate comparisons involving unsigned int. It flags potential issues like logic errors in type usage, helping developers quickly identify and correct them before they reach production. |
| Coverity | 2024.5 | CWE-190 (Integer Overflow or Wraparound) | Coverity’s analysis can detect potential integer overflow or wraparound issues related to incorrect data type usage, providing actionable insights to prevent vulnerabilities in the codebase. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Initializing data values is crucial to prevent undefined behavior and potential security vulnerabilities. Uninitialized variables can contain garbage values, leading to unpredictable behavior, crashes, or security breaches. Proper initialization ensures that variables start with a known and valid state, improving code reliability and security. |

| **Noncompliant Code** |
| --- |
| The variable count is declared but not initialized, leading to undefined behavior when its value is used. |
| void process() {  int count; // Uninitialized variable  // Use count for further processing  std::cout << "Count is: " << count << std::endl; // Output could be unpredictable  } |

| **Compliant Code** |
| --- |
| The variable count is explicitly initialized to 0, ensuring it has a known value before use. |
| void process() {  int count = 0; // Initialized variable  // Use count for further processing  std::cout << "Count is: " << count << std::endl; // Output is predictable  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques (9)  This principle maps to the standard because initializing data values is a fundamental aspect of maintaining code quality and reliability. Uninitialized variables can introduce unpredictable behavior and security vulnerabilities, which effective quality assurance practices aim to prevent. By initializing variables, developers ensure that the code operates in a predictable manner, reducing the risk of defects. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 LTS | Cpp(Variables should be initialized before use) | SonarQube can automatically detect uninitialized variables and flag them for correction. This helps ensure that all variables are initialized before use, preventing undefined behavior and potential security risks. |
| Coverity | 2024.5 | CWE-457 (Use of Uninitialized Variable) | Coverity’s analysis can identify variables that are used without being initialized, providing detailed insights into where and how these issues occur in the codebase, facilitating quick remediation. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Correctly handling strings is essential to prevent vulnerabilities such as buffer overflows, null termination issues, and other security risks. Ensuring string correctness helps maintain the integrity of data, prevents unexpected behavior, and protects against attacks that exploit improper string manipulation. |

| **Noncompliant Code** |
| --- |
| This code concatenates two strings without checking the bounds of the destination buffer, risking a buffer overflow if dest is not large enough to hold both the original string and src. |
| void unsafeConcat(char\* dest, const char\* src) {  // No bounds checking, potential buffer overflow  while (\*dest) dest++;  while ((\*dest++ = \*src++) != '\0');  } |

| **Compliant Code** |
| --- |
| This code uses std::strncat to safely concatenate the source string to the destination buffer, ensuring that the destination buffer is not overrun and remains null-terminated. |
| #include <cstring>  void safeConcat(char\* dest, const char\* src, std::size\_t destSize) {  std::strncat(dest, src, destSize - std::strlen(dest) - 1); // Ensure no buffer overflow  } |

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

| **Principles(s):** Validate Input Data (1)  This principle maps to the standard as correctly handling strings involves ensuring that input data, particularly strings, is properly managed to prevent security vulnerabilities. Buffer overflows and other issues arising from improper string manipulation can be mitigated by validating and managing strings correctly(10), thus maintaining data integrity and preventing unexpected behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeQL | 2.11.5 | cpp/unsafe-string-concatenation | CodeQL can analyze code to detect unsafe string concatenation operations, flagging areas where buffer overflows or other string-related issues might occur, allowing developers to rectify these before they cause problems. |
| Fortify Static Code Analyzer | 21.2.0 | Buffer Overflow: Out-of-bounds Write | Fortify’s Static Code Analyzer detects potential buffer overflow vulnerabilities by analyzing string operations that may write beyond the bounds of a buffer, providing insights into areas where corrective actions are necessary. |
| Clang Static Analyzer | 14.0.1 | core.CStringMisuse | The Clang Static Analyzer includes checks for common string manipulation errors, such as unsafe concatenation and improper null termination, which can lead to security vulnerabilities. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Preventing SQL injection is critical for protecting databases from unauthorized access and manipulation. SQL injection can lead to data breaches, loss of data integrity, and other security issues. By using prepared statements and parameterized queries, the risk of SQL injection can be significantly reduced, ensuring that user input is safely handled. |

| **Noncompliant Code** |
| --- |
| This code constructs an SQL query by directly concatenating user input, which can be manipulated to inject malicious SQL code. |
| #include <iostream>  #include <string>  #include <mysql/mysql.h>  void executeQuery(MYSQL\* conn, const std::string& userInput) {  std::string query = "SELECT \* FROM users WHERE username = '" + userInput + "';";  if (mysql\_query(conn, query.c\_str())) {  std::cerr << "Query failed: " << mysql\_error(conn) << std::endl;  }  } |

| **Compliant Code** |
| --- |
| This code uses a prepared statement with parameterized queries to safely handle user input, preventing SQL injection by not directly concatenating user input into the SQL query. |
| #include <iostream>  #include <string>  #include <mysql/mysql.h>  void executeQuery(MYSQL\* conn, const std::string& userInput) {  const char\* query = "SELECT \* FROM users WHERE username = ?";  MYSQL\_STMT\* stmt = mysql\_stmt\_init(conn);  if (!stmt) {  std::cerr << "Statement initialization failed: " << mysql\_error(conn) << std::endl;  return;  }  if (mysql\_stmt\_prepare(stmt, query, strlen(query))) {  std::cerr << "Statement preparation failed: " << mysql\_stmt\_error(stmt) << std::endl;  return;  }  MYSQL\_BIND bind[1];  memset(bind, 0, sizeof(bind));  bind[0].buffer\_type = MYSQL\_TYPE\_STRING;  bind[0].buffer = (char\*)userInput.c\_str();  bind[0].buffer\_length = userInput.length();  if (mysql\_stmt\_bind\_param(stmt, bind)) {  std::cerr << "Parameter binding failed: " << mysql\_stmt\_error(stmt) << std::endl;  return;  }  if (mysql\_stmt\_execute(stmt)) {  std::cerr << "Statement execution failed: " << mysql\_stmt\_error(stmt) << std::endl;  return;  }  // Fetch and process results as needed  mysql\_stmt\_close(stmt);  } |

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

| **Principles(s):** Validate Input Data (1)  The principle of validating input data directly maps to this standard by emphasizing the importance of handling user input securely. SQL injection occurs when user input is not properly validated or sanitized before being used in SQL queries. By employing prepared statements and parameterized queries, this principle is upheld, preventing malicious input from altering SQL commands and accessing or corrupting the database. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.3.9 | Sql-injection | SonarQube identifies potential SQL injection vulnerabilities in code by analyzing how SQL queries are constructed and ensuring that user inputs are handled safely. It provides actionable insights to help developers secure their code against such attacks. |
| Fortify Static Code Analyzer | 21.2.0 | SQL Injection | Fortify’s SCA provides detailed analysis of SQL injection risks by checking how queries are built and executed in the code. It ensures that all user inputs used in SQL statements are properly sanitized or parameterized, reducing the risk of injection. |
| CodeQL | 2.11.5 | cpp/sql-injection | CodeQL examines the code for SQL injection vulnerabilities by tracing the flow of user inputs into SQL queries. It flags instances where inputs are not properly parameterized, helping developers mitigate these risks. |
| OWASP ZAP | 2.11.1 | SQL Injection | OWASP ZAP is a dynamic security testing tool that can be used to detect SQL injection vulnerabilities in web applications by simulating attack patterns and identifying weak points where user inputs might exploit SQL queries. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Ensuring memory protection is essential for preventing vulnerabilities like buffer overflows, memory leaks, and dangling pointers, which can lead to undefined behavior, system crashes, and security breaches. Proper memory management, including bounds checking and using safe allocation/deallocation practices, enhances the reliability and security of the code. |

| **Noncompliant Code** |
| --- |
| This code lacks bounds checking, leading to potential buffer overflows if src is larger than dest. |
| void copyData(char\* dest, const char\* src, std::size\_t size) {  // No bounds checking  for (std::size\_t i = 0; i <= size; ++i) {  dest[i] = src[i];  }  } |

| **Compliant Code** |
| --- |
| This code uses std::strncpy to safely copy data into the destination buffer, ensuring that it does not overflow and is properly null-terminated. |
| #include <cstring>  void safeCopyData(char\* dest, const char\* src, std::size\_t destSize) {  std::strncpy(dest, src, destSize - 1);  dest[destSize - 1] = '\0'; // Ensure null-termination  } |

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

| **Principles(s):** Architect and Design for Security Policies (3)  The principle of architecting and designing for security policies maps to this standard by emphasizing the importance of considering memory safety from the outset of the design process. Ensuring that memory is properly protected through bounds checking, safe memory allocation, and deallocation practices prevents vulnerabilities like buffer overflows, memory leaks, and dangling pointers, which can lead to severe security risks, including system crashes and undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17.0 | Memcheck | Valgrind’s Memcheck tool detects memory-related issues such as buffer overflows, memory leaks, and use-after-free errors in programs. It helps developers identify and correct memory protection issues, ensuring safer and more secure code. |
| AddressSanitizer (ASan) | Integrated with GCC/Clang | AddressSanitizer | ASan is a fast memory error detector that identifies memory bugs such as out-of-bounds accesses and use-after-free errors. It provides detailed reports to help developers fix memory issues early in the development process. |
| Coverity Static Analysis | 2023.12 | Memory-Overruns | Coverity’s static analysis tool identifies memory-related vulnerabilities, including buffer overruns and memory leaks, in code. It offers actionable insights to ensure memory is managed securely, preventing potential security risks. |
| PVS-Studio | 7.16 | V512 (buffer overflow) | PVS-Studio is a static code analyzer that detects potential vulnerabilities, including buffer overflows and other memory-related issues. It helps developers adhere to secure coding standards by flagging risky memory operations. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions are used to document and enforce assumptions made by the programmer about the state of the program at specific points. Proper use of assertions can catch logic errors and invalid program states during development, leading to more robust and reliable software. However, overuse or misuse of assertions can lead to performance overhead and maintenance challenges. |

| **Noncompliant Code** |
| --- |
| This code does not use assertions to check if the pointer arr is null before dereferencing it, leading to potential undefined behavior if a null pointer is passed. |
| void processArray(int\* arr, std::size\_t size) {  // Missing assertion for null pointer check  for (std::size\_t i = 0; i < size; ++i) {  arr[i] = i \* 2;  }  } |

| **Compliant Code** |
| --- |
| This code uses assert to ensure that the pointer arr is not null before it is dereferenced, preventing potential null pointer dereference errors. |
| #include <cassert>  void processArray(int\* arr, std::size\_t size) {  assert(arr != nullptr); // Ensure arr is not null  for (std::size\_t i = 0; i < size; ++i) {  arr[i] = i \* 2;  }  } |

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

| **Principles(s):** Validate Input and Output(1) to Prevent Security Vulnerabilities  The principle of validating input and output aligns with this standard by ensuring that assumptions about the program's state are explicitly checked using assertions. This helps in catching logic errors and invalid program states early during development, leading to more robust and reliable software. Assertions ensure that input and output conditions are as expected, which reduces the likelihood of security vulnerabilities arising from incorrect assumptions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0 | Null Dereference | Clang Static Analyzer detects potential null pointer dereferences and other logic errors. It helps developers ensure that assumptions about program states are validated using assertions or other checks. |
| GCC - Wall | 11.2 | Warning for missing null checks | GCC's -Wall option enables warnings for common issues, including missing null checks, which can help catch potential errors that could be mitigated using assertions. |
| PVS-Studio | 7.16 | V601 | PVS-Studio detects potential logic errors and vulnerabilities, including unvalidated assumptions. It recommends using assertions where appropriate to enforce assumptions about program states. |
| CodeSonar | 6.2 | Logic errors - Invalid states | CodeSonar performs a deep static analysis to identify logic errors and potential invalid program states. It suggests the use of assertions to catch such errors early in the development process. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Using exceptions for error handling helps separate normal code flow from error handling code, making the program easier to read and maintain. Proper use of exceptions ensures that errors are handled appropriately, preventing undefined behavior, data corruption, and program crashes. Misusing exceptions can lead to performance overhead, unhandled exceptions, and resource leaks. |

| **Noncompliant Code** |
| --- |
| This code fails silently if the file cannot be opened, providing no feedback or error handling mechanism. |
| void readFile(const std::string& filename) {  std::ifstream file(filename);  if (!file.is\_open()) {  // Silent failure  return;  }  // Process file  file.close();  } |

| **Compliant Code** |
| --- |
| This code throws a std::runtime\_error exception if the file cannot be opened, providing a clear error handling mechanism that can be caught and managed by the caller. |
| #include <stdexcept>  #include <fstream>  void readFile(const std::string& filename) {  std::ifstream file(filename);  if (!file.is\_open()) {  throw std::runtime\_error("Failed to open file: " + filename);  }  // Process file  file.close();  } |

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

| **Principles(s):** Separate Normal Code Flow from Error Handling (3) (4)  The principle of separating normal code flow from error handling aligns with this standard by encouraging the use of exceptions to manage errors in a controlled manner. This separation ensures that the main logic of the program remains clean and readable, while error handling is managed in a structured way. Properly using exceptions prevents undefined behavior, data corruption, and program crashes by ensuring that errors are appropriately handled and communicated to the caller. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0 | modernize-use-std-exception | Clang-Tidy detects code that could benefit from modern exception handling practices, such as using std::exception and its derivatives. It encourages using exceptions to handle errors properly and ensures that the code adheres to modern C++ standards. |
| GCC -Wall -Wextra | 11.2 | Warning for unhandled exceptions | GCC's -Wall and -Wextra options enable warnings for common issues, including unhandled exceptions and potential resource leaks due to exceptions. It helps developers ensure that exceptions are caught and managed appropriately. |
| PVS-Studio | 7.16 | V562 | PVS-Studio detects potential exceptions that are not handled or that could cause resource leaks. It provides recommendations for improving exception safety and ensuring that resources are properly managed even when exceptions occur. |
| SonarQube | 8.9 | cpp | SonarQube analyzes C++ code for potential issues related to exception handling, such as uncaught exceptions and resource leaks. It helps developers ensure that exceptions are used appropriately and that error handling is robust. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Avoid Unnecessary Casting | STD-008-CPP | Avoiding unnecessary casting enhances code readability and maintainability while reducing the risk of errors. Unnecessary casts can obscure the programmer's intent, lead to subtle bugs, and introduce undefined behavior, especially when converting between incompatible types. |

| **Noncompliant Code** |
| --- |
| This code uses an unnecessary cast from float to int, which can be avoided by designing the function to work directly with the correct data types. |
| int calculate(float value) {  int intValue = (int)value; // Unnecessary cast from float to int  return intValue \* 2;  } |

| **Compliant Code** |
| --- |
| This code eliminates the need for casting by directly using the appropriate data type, ensuring better readability and reducing potential errors. |
| int calculate(int value) {  return value \* 2;  } |

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

| **Principles(s):** Maintain Code Readability and Reduce Risk (4)  This principle maps to the standard of avoiding unnecessary casting by emphasizing the importance of code clarity and reducing the likelihood of introducing subtle bugs or undefined behavior. Unnecessary casting can obscure the programmer's intent, making the code harder to understand and maintain. By avoiding these casts, developers can ensure that their code is more straightforward, less error-prone, and easier to maintain over time. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0 | readability-avoid-casting | Clang-Tidy includes a checker that detects unnecessary type casting and provides recommendations for eliminating them. It helps improve code readability and ensures that casting is only used when necessary. |
| GCC -Wall -Wextra | 11.2 | Warning for unnecessary type casting | GCC's -Wall and -Wextra options can generate warnings for unnecessary type casting, helping developers identify and remove redundant casts in their code. |
| PVS-Studio | 7.16 | V200 | PVS-Studio detects unnecessary casts that may obscure code intent or introduce potential bugs. It provides insights into where casting can be avoided and suggests improvements. |
| SonarQube | 8.9 | cpp | SonarQube analyzes C++ code for unnecessary casting and suggests refactoring opportunities to improve code readability and maintainability. It helps ensure that type conversions are handled appropriately and without introducing risks. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Resource Management | STD-09-CPP | Proper resource management is critical for preventing resource leaks, such as memory leaks, file descriptor leaks, and other system resource leaks. This ensures that resources are properly allocated, used, and released, maintaining system stability and performance. |

| **Noncompliant Code** |
| --- |
| This code opens a file but does not close it, leading to a file descriptor leak. |
| void process() {  FILE\* file = fopen("data.txt", "r");  if (!file) {  return;  }  // Process file  // Missing fclose  } |

| **Compliant Code** |
| --- |
| This code ensures that the file is properly closed after processing, preventing a file descriptor leak. |
| void process() {  FILE\* file = fopen("data.txt", "r");  if (!file) {  return;  }  // Process file  fclose(file);  } |

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

| **Principles(s):** Ensure Proper Resource Allocation and Deallocation  This principle maps to the standard of resource management by emphasizing the importance of managing resources efficiently and responsibly. Proper resource management ensures that system resources like memory, file handles, and network connections are allocated and released appropriately. Failure to manage these resources can lead to leaks, which degrade system performance, cause crashes, and potentially introduce security vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Moderate | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17.0 | Memcheck | Valgrind's Memcheck tool detects memory leaks and other resource management issues by monitoring memory allocation, usage, and deallocation. It helps identify resource leaks and provides detailed reports for fixing them. |
| AddressSanitizer (ASan) | GCC 11.2, Clang 12.0 | LeakSanitizer | AddressSanitizer is a runtime memory error detector that includes a LeakSanitizer tool for detecting memory leaks. It tracks allocations and ensures that resources are correctly deallocated, preventing memory leaks. |
| PVS-Studio | 7.16 | V730 | PVS-Studio provides static analysis for detecting resource management issues, such as forgetting to close file handles or deallocate memory. It helps ensure that all resources are properly managed. |
| Coverity | 2022.03 | RESOURCE\_LEAK | Coverity's static analysis tool detects resource leaks in code, such as unclosed file handles or unreleased memory. It helps maintain proper resource management by identifying potential leaks before they become critical. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| RAII for Resource Management | STD-010-CPP | Resource Acquisition Is Initialization (RAII) is a programming idiom that ensures resources are properly released when they are no longer needed. By tying resource management to object lifetime, RAII simplifies code and prevents resource leaks, ensuring robust and maintainable resource management. |

| **Noncompliant Code** |
| --- |
| This code manually manages memory allocation and deallocation, which can lead to memory leaks if the code path is not straightforward. |
| void process() {  int\* data = new int[100];  // Process data  delete[] data; // Manual resource management  } |

| **Compliant Code** |
| --- |
| This code uses std::vector, which manages memory automatically, ensuring that resources are properly released when the vector goes out of scope. |
| #include <vector>  void process() {  std::vector<int> data(100);  // Process data  } |

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

| **Principles(s):** Leverage Object Lifetimes for Resource Management  This principle is central to the RAII (Resource Acquisition Is Initialization) idiom, which ties the allocation and deallocation of resources directly to the lifespan of objects. By associating resource management with object lifetimes, RAII helps ensure that resources are automatically cleaned up when objects go out of scope, preventing leaks and simplifying code maintenance. This approach enhances reliability and maintainability by reducing the likelihood of errors related to manual resource management. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Low to Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0 | modernize-use-auto, modernize-use-smart-ptr | Clang-Tidy provides checks that encourage modern C++ practices, including the use of smart pointers for automatic resource management. These checks help enforce RAII by recommending the use of std::unique\_ptr, std::shared\_ptr, and std::vector, reducing manual memory management. |
| PVS-Studio | 7.16 | V595 | PVS-Studio's V595 checker identifies potential resource management issues and suggests the use of RAII-compliant constructs. This helps developers refactor code to ensure that resources are managed automatically and efficiently. |
| Cppcheck | 2.7 | memleak | Cppcheck includes checks for memory leaks and recommends RAII as a solution for preventing them. By analyzing the code for potential memory leaks, it helps enforce proper resource management practices. |
| Coverity | 2022.03 | MISSING\_RAII | Coverity's static analysis tool includes a checker that identifies opportunities to apply RAII. It helps ensure that resources are automatically managed by analyzing object lifetimes and suggesting RAII-compliant alternatives to manual resource management. |

### 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:** To make sure Green Pace’s DevSecOps pipeline runs smoothly and securely, we’ll be putting together a solid testing plan. Our goal is to integrate security checks into every stage of the software development process. For example, we'll run automated tests during the build process to catch any vulnerabilities in the code, and we'll check security configurations before anything moves to production. By doing this, we can be confident that all code and configurations meet our security standards before they move forward.

We’ll also put our system through some stress tests to see how well the automated security measures hold up under pressure, like during high traffic or simulated attacks. This helps ensure that the system remains secure, even when things get tough. On top of that, we’ll have continuous monitoring in place to keep an eye on everything in real time, allowing us to quickly make any necessary adjustments. This way, we can be sure that our security standards are being consistently applied throughout the entire development process.

**Validate Input Data Explanation:** Incorrect data types can lead to severe issues such as overflows, underflows, and logic errors, which may compromise the security and functionality of the system. The likelihood of such issues occurring is significant, but the cost to remediate is generally low since it primarily involves correcting the data type. Given the potential impact, this standard should be prioritized highly.

**Initializing Data Values Explanation:** Uninitialized variables can lead to undefined behavior, which may result in security vulnerabilities, crashes, or other unpredictable issues in the system. The severity of these issues is high because they can potentially compromise system integrity. The likelihood of occurrence is also high, especially in complex systems where oversight may occur. However, the cost of remediation is low, as it simply involves ensuring proper initialization during development. Given the potential impact, addressing this standard should be a high priority.

**String Correctness Explanation:** Buffer overflow vulnerabilities, particularly with string handling, can lead to critical security breaches, including arbitrary code execution and system crashes. The likelihood of such vulnerabilities is high in systems where proper bounds checking is not enforced. Remediation costs can be moderate as it may involve refactoring code to ensure all string operations are safe. Due to the potential for severe consequences, addressing this standard is of the highest priority.

**SQL Injection Explanation:** SQL injection is one of the most severe security vulnerabilities as it can lead to unauthorized access to the database, data breaches, and potentially catastrophic loss of data integrity. The likelihood of such an attack is high, particularly in applications that handle user input without proper sanitization. Although implementing secure coding practices such as parameterized queries can be complex and costly, the critical nature of this vulnerability means that remediation should be a top priority.

**Memory Protection Explanation:** Memory protection issues like buffer overflows and memory leaks can result in serious security vulnerabilities, such as allowing an attacker to execute arbitrary code or causing the system to crash. The likelihood of such vulnerabilities occurring is moderate, especially in systems with complex memory management. Although the remediation cost may vary depending on the complexity of the system, the high severity of the consequences makes it a priority to address these issues.

**Assertions Explanation:** The lack of assertions can lead to undetected logic errors and invalid program states, which may result in undefined behavior or crashes, especially in cases where critical assumptions are violated. The likelihood of encountering such issues is high in complex systems, but the cost of remediation is typically low because adding assertions is straightforward. Given the potential for significant problems if such issues go unchecked, the priority is medium, but attention should be given to ensuring assertions are used appropriately.

**Exceptions Explanation:** The misuse of exceptions or the failure to handle them can lead to significant issues, such as unhandled exceptions, resource leaks, and program crashes, which are critical in production environments. The severity of these issues is high, especially if they affect critical systems. The likelihood of encountering such problems varies based on the complexity of the code and the handling of edge cases. The remediation cost can be medium, as it involves refactoring code to properly handle exceptions. Given the potential impact on system stability and reliability, the priority is high.

**Avoid Unnecessary Casting Explanation:** While unnecessary casting may not always lead to critical issues, it can introduce subtle bugs that are difficult to detect and debug. The severity is medium, as these bugs can cause incorrect behavior, especially when dealing with type conversions that may result in loss of precision or undefined behavior. The likelihood is medium, depending on how frequently casting is used in the codebase. The remediation cost is low, as removing unnecessary casts is usually straightforward. The priority is medium, as improving code readability and reducing potential errors is important for long-term maintainability.

**Resource Management Explanation:** Resource leaks, such as memory leaks or file descriptor leaks, can lead to significant performance degradation, system instability, and even system crashes. The severity is high because these issues can impact the entire system, especially in long-running applications. The likelihood is medium, as resource leaks are common in complex systems where resources are frequently allocated and deallocated. The remediation cost is medium, as detecting and fixing leaks can be time-consuming but is critical to maintaining system reliability. The priority is high, as resource management is essential for the stability and performance of the software.

**RAII for Resource Management Explanation:** Mismanagement of resources, particularly memory, can lead to severe consequences such as memory leaks, crashes, and undefined behavior, making the severity high. The likelihood is medium because manual resource management is prone to errors, especially in complex code paths. However, by implementing RAII, the remediation cost is relatively low to medium, as it often involves refactoring code to use RAII-compliant structures like smart pointers or containers. The priority is high because adopting RAII ensures robust and automatic resource management, significantly reducing the risk of resource-related errors.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Likely | Low | High | 2 |
| STD-002-CPP | High | Likely | Low | High | 2 |
| STD-003-CPP | Critical | Likely | Medium | High | 1 |
| STD-004-CPP | Critical | Likely | High | Very High | 1 |
| STD-005-CPP | High | Unlikely | Medium | High | 2 |
| STD-006-CPP | Moderate | Likely | Low | Medium | 3 |
| STD-007-CPP | High | Unlikely | Medium | High | 4 |
| STD-008-CPP | Moderate | Unlikely | Low | Medium | 3 |
| STD-009-CPP | High | Unlikely | Medium | High | 1 |
| STD-010-CPP | High | Unlikely | Low to Medium | High | 1 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest refers to the encryption of data that is stored on a physical medium (such as a hard drive, database, or cloud storage) to protect it from unauthorized access. This type of encryption ensures that even if someone gains physical access to the storage medium, they cannot read or manipulate the data without the decryption key.  The policy applies to all data stored on company servers, employee devices, and cloud storage services. This is crucial for protecting sensitive information such as customer data, intellectual property, and financial records from data breaches. Encryption at rest should be enforced through the use of strong encryption algorithms such as AES-256. The policy mandates that all stored data must be encrypted by default, and access to encryption keys must be strictly controlled. |
| Encryption in flight | Encryption in flight refers to the encryption of data as it is transmitted over networks, such as the internet or private networks. This type of encryption protects data from being intercepted by unauthorized parties during transmission.  The policy applies to all data transmitted over public or private networks, including emails, file transfers, and API communications. Encryption in flight is critical for protecting data from eavesdropping, man-in-the-middle attacks, and other network-based threats. The policy requires that all network communications use secure protocols such as TLS (Transport Layer Security) to encrypt data in transit. This ensures that sensitive information, such as login credentials, payment information, and personal data, remains confidential during transmission. |
| Encryption in use | Encryption in use refers to the encryption of data while it is being processed or used by applications, typically within the memory of a device. This type of encryption helps protect data from being accessed or tampered with during its active use.  The policy applies to data being processed by applications, particularly those handling sensitive information. Encryption in use is essential for preventing unauthorized access to data during its processing, especially in environments where data is at risk of being exposed, such as in multi-tenant cloud environments or during sensitive operations like encryption key management. The policy mandates that sensitive data must remain encrypted while in use, and technologies such as homomorphic encryption or secure enclaves should be employed where applicable. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identity of a user or system before granting access to resources. This ensures that only authorized individuals or systems can access sensitive data or perform critical operations.  The policy applies to all systems, applications, and databases within the organization. Strong authentication mechanisms, such as multi-factor authentication (MFA), are required for user logins, especially for users with elevated privileges or access to sensitive data. This policy is crucial for preventing unauthorized access, protecting against account takeovers, and ensuring that only legitimate users can perform actions such as logging into systems, accessing confidential files, or making changes to the database. |
| Authorization | Authorization is the process of determining what actions a user or system is allowed to perform once authenticated. This includes granting or denying access to specific resources, files, or services based on the user's role or permissions.  The policy applies to all users and systems interacting with the organization's resources. It ensures that users have the minimum level of access necessary to perform their duties (principle of least privilege). This policy is essential for controlling access to sensitive information, ensuring that users can only perform actions appropriate to their role, and preventing unauthorized changes to critical systems. The policy mandates regular reviews of user permissions, especially after changes to user roles or job functions. |
| Accounting | Accounting (also known as auditing) involves tracking user activities, such as logins, file access, and changes to the database. This ensures that all actions are recorded and can be reviewed for compliance, security, and troubleshooting purposes.  The policy applies to all user activities within the organization's systems. Comprehensive logging of user actions, including successful and failed login attempts, file access, database changes, and the addition of new users, is required. This policy is vital for maintaining a record of activities, detecting unauthorized actions, and ensuring compliance with regulatory requirements. Logs must be securely stored and regularly reviewed to detect suspicious activity and ensure 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.

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 07/17/2024 | Module Three Milestone | Irina Borozan |  |
| 2.1 | 08/09/2024 | Project One | Irina Borozan |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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