**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 | Validating input data is essentially ensuring that data received by a program is both correct and safe. This involves checking that inputs conform to expected formats, types, and ranges before processing them. Proper validation helps prevent common vulnerabilities such as SQL injection, buffer overflows, and cross-site scripting (XSS). |
| 1. Heed Compiler Warnings | Heed Compiler warnings are indicators of potential issues in the code that may not necessarily prevent compilation but could lead to unexpected behaviors or vulnerabilities. Ignoring these warnings can result in bugs, security flaws, and unstable software. |
| 1. Architect and Design for Security Policies | Architect and Design for Security Policies are phases where security considerations should be made to ensure integrity to the system. This involves defining clear security requirements, access controls, and data protection mechanisms that align with organizational policies and regulatory standards. |
| 1. Keep It Simple | Keep It Simple is useful in design and implementation to reduce the likelihood of introducing errors and vulnerabilities. Complex systems are harder to understand, maintain, and secure, increasing the chances of security flaws slipping through. |
| 1. Default Deny | The Default Deny principle ensures that access to resources is restricted by default, only allowing actions that have been explicitly permitted. This should minimize attacks by preventing unauthorized access and reducing the potential for exploitation. |
| 1. Adhere to the Principle of Least Privilege | The Principle of Least Privilege dictates that users and system components should operate with the minimal level of access necessary to perform their functions. By limiting these privileges, the potential impact of attacks on users or components is significantly reduced. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing Data Sent to Other Systems is a great principle to prevent the transmission or sharing of malicious data. This process involves removing or encoding potentially harmful data, like executable code or special characters, to mitigate risks like injection attacks and data corruption. |
| 1. Practice Defense in Depth | Practice Defense in Depth involves implementing multiple layers of security controls to protect systems from various types of threats. This is done by combining physical, technical, and administrative security features, allowing organizations to create a comprehensive security posture that addresses different attacks. |
| 1. Use Effective Quality Assurance Techniques | Effective Quality Assurance (QA) techniques, like code reviews, automated testing, and vulnerability assessments, are crucial for identifying and addressing security issues before deployment. By systematically testing and validating code, developers can catch and remediate vulnerabilities early, leading to more secure and reliable applications. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard involves following best practices and guidelines that promote writing safe and resilient code. Standards such as OWASP, CERT, and company-specific guidelines provide frameworks for avoiding common security pitfalls and enforcing consistent security measures across codebases. |

### 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-ASR] | Assertions are a fundamental tool in software development used to enforce program invariants and detect programming errors during development and testing phases. In C++, assertions are typically implemented using the assert() macro provided by the <cassert> header. The logical rationale behind this standard is to ensure that assertions are used appropriately to validate conditions that should logically hold true if the program is functioning correctly. |

| **Noncompliant Code** |
| --- |
| The noncompliant code example demonstrates improper use of assertions that violate the ASR50-CPP standard. An assertion is used to handle a runtime error, which is inappropriate as assertions should not replace proper error handling mechanisms. |
| #include <cassert>  #include <iostream>    void openFile(const std::string &filename) {  FILE \*file = fopen(filename.c\_str(), "r");  assert(file != nullptr); // Improper use: Should handle the error instead  // Proceed with file operations...  fclose(file);  }    int main() {  openFile("nonexistent.txt");  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant code example, the assertion checks that the index is within the valid range before accessing the vector element. There are no side effects within the assertion, ensuring consistent behavior even if assertions are disabled. |
| #include <cassert>  #include <vector>    int getElement(const std::vector<int> &v, size\_t index) {  assert(index < v.size() && "Index out of bounds"); // Valid use: invariant check  return v[index]; // No side effects  }    int main() {  std::vector<int> v = {1, 2, 3};  int value = getElement(v, 2); // Safe: index is within bounds  return 0;  } |

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

| **Principles(s):** Validate Input Data | Adopt a Secure Coding Standard | Use Effective Quality Assurance Techniques:  The principle “Adopting a Secure Coding Standard” ensures that assertions are used correctly to validate program invariants, preventing misuse for runtime error handling. While “Validating Input Data” is also applied by verifying conditions before executing code, contributing to the stability and security of the software. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| **HIGH** | **Unlikely** | **Low** | **HIGH** | **2** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.7 | assertUse | Cppcheck can utilize the assertUse checker specifically to identify instances where assertions are misused, such as handling runtime errors or containing side effects, ensuring that assertions are used solely for enforcing program invariants. |
| Visual Studio | 2022 | Code Analysis Rules | Visual Studio built-in code analysis tool includes rules for proper assertion usage. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-002-EXP] | Expressions focus on the correct usage of pointers, especially when deleting dynamically allocated arrays. The logical rationale behind this standard is to prevent undefined behavior and potential security vulnerabilities that arise from deleting an array through a pointer type that does not match the array’s dynamic type. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an array of Derived objects is created and the pointer is stored in a Base \*.Despite Base::~Base() being declared virtual, it still results in undefined behavior. Further, attempting to perform pointer arithmetic on the static type Base \* violates CTR56-CPP. Do not use pointer arithmetic on polymorphic objects. |
| **struct** Base {  **virtual** ~Base() = **default**;  };    **struct** Derived final : Base {};    **void** f() {  Base \*b = **new** Derived[10];  // ...  **delete** [] b;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the static type of b is Derived \*, which removes the undefined behavior when indexing into the array as well as when deleting the pointer. |
| **struct** Base {  **virtual** ~Base() = **default**;  };    **struct** Derived final : Base {};    **void** f() {  Derived \*b = **new** Derived[10];  // ...  **delete** [] b;  } |

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

| **Principles(s):** Adopting a Secure Coding Standard | Validating Input Data:  By applying the principles of “Adopting a Secure Coding Standard” and “Validating Input Data”, this standard ensures that memory management is handled securely, preventing undefined behavior from improper pointer usage. Proper design and validation reduce the likelihood of vulnerabilities caused by dynamic memory allocation issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.7 | useAfterDelete | Identifies scenarios where memory is deleted through a pointer type that does not match the allocated type, preventing undefined behavior associated with mismatched pointer deletions. |
| Visual Studio | 2022 | Code Analysis Rule (CXX\_PTR\_CAST) | VS code built-in code analysis tool includes the CXX\_PTR\_CAST rule, which detects pointer casts that could lead to improper memory deletion. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard (Data Value Coding Standard)** |
| --- | --- | --- |
| **Data Value** | [STD-003-INT] | Declarations and Initialization focus on the proper declaration and initialization of variables in programming. The logical rationale behind this standard is to prevent a range of security vulnerabilities that can arise from improper handling of type casting, particularly when casting integers to enumeration types. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example attempts to check whether a given value is within the range of acceptable enumeration values. However, it is doing so after casting to the enumeration type, which may not be able to represent the given integer value. On a two’s complement system, the valid range of values that can be represented by EnumType are [0..3], so if a value outside of that range were passed to f(), the cast to EnumType would result in an unspecified value, and using that value within the if statement results in unspecified behavior. |
| **enum** EnumType {  First,  Second,  Third  };    **void** f(**int** intVar) {  EnumType enumVar = **static\_cast**<EnumType>(intVar);    **if** (enumVar < First || enumVar > Third) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| This compliant solution checks that the value can be represented by the enumeration type before performing the conversion to guarantee the conversion does not result in an unspecified value. It does this by restricting the converted value to one for which there is a specific enumerator value. |
| **enum** EnumType {  First,  Second,  Third  };    **void** f(**int** intVar) {  **if** (intVar < First || intVar > Third) {  // Handle error  }  EnumType enumVar = **static\_cast**<EnumType>(intVar);  } |

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

| **Principles(s):** Validating Input Data | Adopt a Secure Coding Standard:  The “Validating Input Data” and “Adopt a Secure Coding Standard” principles ensure that casting integers to enumeration types is done safely, mitigating security risks related to type casting errors. Input validation avoids potential issues arising from improper casting or data handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.7 | enumConversion | Identifies instances where integers are cast to enumeration types without proper range validation. |
| Visual Studio | 2022 | CXX\_ENUM\_CAST | Detects instances where integers are cast to enumeration types without proper range checks. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard (Memory Protection Coding Standard)** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-SQL] | SQL Injection is a prevalent and severe security vulnerability that allows attackers to manipulate SQL queries by injecting malicious input. In C++ applications that interact with databases, improper handling of user inputs can lead to unauthorized access, data manipulation, or even complete system compromise. The logical rationale behind this standard is to enforce practices that eliminate the possibility of SQL Injection by ensuring that all database queries are constructed safely. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example demonstrates insecure practices that make the application susceptible to SQL Injection Attacks. User input is directly concatenated into an SQL query string without any validation or sanitization, allowing attackers to manipulate the query structure. |
| #include <iostream>  #include <string>  #include <mysql/mysql.h> // Example using MySQL Connector/C++    void getUserData(MYSQL \*conn, const std::string &username) {  std::string query = "SELECT \* FROM users WHERE username = '" + username + "';";  if (mysql\_query(conn, query.c\_str())) {  std::cerr << "Query Failed: " << mysql\_error(conn) << std::endl;  return;  }  MYSQL\_RES \*result = mysql\_store\_result(conn);  // Process result...  mysql\_free\_result(result);  } |

| **Compliant Code** |
| --- |
| This compliant example uses parameterized queries provided by the MySQL Connector/C++ library. This approach prevents any malicious input from altering the structure or intent of the SQL query. |
| #include <iostream>  #include <string>  #include <sql.h>  #include <sqlext.h>    void getUserData(SQLHDBC dbc, const std::string &username) {  SQLHSTMT stmt;  SQLAllocHandle(SQL\_HANDLE\_STMT, dbc, &stmt);    const char \*query = "SELECT \* FROM users WHERE username = ?";  SQLPrepare(stmt, (SQLCHAR\*)query, SQL\_NTS);    // Bind the username parameter  SQLBindParameter(stmt, 1, SQL\_PARAM\_INPUT, SQL\_C\_CHAR, SQL\_VARCHAR, 255, 0,  (SQLPOINTER)username.c\_str(), 0, NULL);    if (SQLExecute(stmt) != SQL\_SUCCESS && SQLExecute(stmt) != SQL\_SUCCESS\_WITH\_INFO) {  std::cerr << "SQL Execution Failed\n";  SQLFreeHandle(SQL\_HANDLE\_STMT, stmt);  return;  }    // Process results...    SQLFreeHandle(SQL\_HANDLE\_STMT, stmt);  } |

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

| **Principles(s):** Sanitizing Data Sent to Other Systems | Validating Input Data:  Following the principles of “Sanitizing Data Sent to Other Systems” and “Validating Input Data”, this standard enforces secure database interaction by preventing malicious input from altering SQL queries. Proper input handling and query sanitization protect against injection attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.7 | sqlInjection | Identifies instances where user inputs are directly concatendated into SQL queries without proper sanitization or parameterization. |
| Visual Studio | 2022 | CA2100: SQL queries should not be constructed from concatenated steings | Detects instances where SQL queries are constructed using concatenated strings. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard (String Correctness Coding Standard)** |
| --- | --- | --- |
| **String Correctness** | [STD-005-STR] | Characters and strings are fundamental data types in C++ programming. However, improper handling of string storage can lead to buffer overflows, memory corruption, and security vulnerabilities. The logical rationale behind this standard is to ensure that all string operations are performed safely by guaranteeing that buffers have adequate space to store both the character data and the null terminator. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the unformatted input function std::basic\_istream<T>::read() is used to read an unformatted character array of 32 characters from the given file. However, the read() function does not guarantee that the string will be null terminated, so the subsequent call of the std::string constructor results in undefined behavior if the character array does not contain a null terminator. |
| #include <fstream>  #include <string>    **void** f(std::istream &in) {  **char** buffer[32];  **try** {  in.read(buffer, **sizeof**(buffer));  } **catch** (std::ios\_base::failure &e) {  // Handle error  }    std::string str(buffer);  // ...  } |

| **Compliant Code** |
| --- |
| This compliant solution assumes that the input from the file is at most 32 characters. Instead of inserting a null terminator, it constructs the std::string object based on the number of characters read from the input stream. If the size of the input is uncertain, it is better to use std::basic\_istream<T>::readsome() or a formatted input function, depending on need. |
| #include <fstream>  #include <string>    **void** f(std::istream &in) {  **char** buffer[32];  **try** {  in.read(buffer, **sizeof**(buffer));  } **catch** (std::ios\_base::failure &e) {  // Handle error  }  std::string str(buffer, in.gcount());  // ...  } |

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

| **Principles(s):** Validating Input Data | Adopting a Secure Coding Standard:  The principles of “Validating Input Data” and “Adopting a Secure Coding Standard” apply here by ensuring that string operations, particularly those related to buffers, are handled safely. This prevents buffer overflows and guarantees proper memory usage, protecting data integrity. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.7 | bufferOverrun | Identifies scenarios where string operations may exceed buffer limits. |
| Visual Studio | 2022 | CA2263: Review SQL queries for security vul | Detects unsafe string operations that may lead to buffer overflows. |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard (Memory Protection Coding Standard)** |
| --- | --- | --- |
| **Memory Protection (Assertions)** | [STD-006-MEM] | Memory management in C++ requires careful handling of dynamically allocated memory to prevent undefined behavior and security vulnerabilities. Accessing memory that has already been deallocated results in dangling pointers, which can lead to arbitrary code execution, data corruption, and program instability. The logical rationale behind this standard is to ensure that once memory is freed, all pointers referencing that memory are either reset or redirected to valid memory regions. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. Typically, dynamic memory allocations and deallocations are far removed, making it difficult to recognize and diagnose such problems. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {  S \*s = **new** S;  // ...  **delete** s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the dynamically allocated memory is not deallocated until it is no longer required. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {  S \*s = **new** S;  // ...  s->f();  **delete** s;  } |

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

| **Principles(s):** Adopting a Secure Coding Standard | Defense in Depth:  By adhering to “Adopting a Secure Coding Standard” and “Defense in Depth”, this standard ensures proper memory deallocation to avoid security risks like dangling pointers. Correct memory management practices contribute to reducing vulnerabilities in code execution. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.7 | danglingPointer | Identifies instances where pointers are used after the memory they reference has been deallocated. |
| Visual Studio | 2022 | CXX\_PTR\_USE\_AFTER\_FREE | Detects instances where pointers are used after the memory they reference has been deallocated. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-FIO] | Proper management of file stream operations in C++ is essential to prevent undefined behavior and ensure data integrity. The C++ Standard specifies that when performing both input and output operations on a file stream opened in update mode, an intervening call to a flush or a file positioning function is required between alternating input and output operations. Failing to adhere to this requirement can result in undefined behavior, including data corruption, inconsistent file states, and potential security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example appends data to the end of a file and then reads from the same file. However, because there is no intervening positioning call between the formatted output and input calls, the behavior is undefined. |
| #include <fstream>  #include <string>    **void** f(**const** std::string &fileName) {  std::fstream file(fileName);  **if** (!file.is\_open()) {  // Handle error  **return**;  }    file << "Output some data";  std::string str;  file >> str;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the std::basic\_istream<T>::seekg() function is called between the output and input, eliminating the undefined behavior. |
| #include <fstream>  #include <string>    **void** f(**const** std::string &fileName) {  std::fstream file(fileName);  **if** (!file.is\_open()) {  // Handle error  **return**;  }    file << "Output some data";    std::string str;  file.seekg(0, std::ios::beg);  file >> str;  } |

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

| **Principles(s):** Adopting a Secure Coding Standard | Effective Quality Assurance Techniques:  The “Adopting a Secure Coding Standard” and “Effective Quality Assurance Techniques” principles emphasize proper file stream handling by ensuring correct sequencing of input and output operations. This avoids undefined behavior, ensuring file integrity and preventing potential data corruption. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.7 | fileStreamManagement | Identifies instances where file operations are improperly sequenced, such as missing flushes or file positioning calls between input and output operations. |
| Visual Studio | 2022 | CA2100: SQL queries should not be constructed from | Again, CA2100 can detect instances where file operations are improperly sequenced, such as missing flushes or file positioning calls between input and output operations. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard (Exceptions Coding Standard)** |
| --- | --- | --- |
| Exceptions | [STD-008-ERR] | Proper exception handling and program termination are critical for maintaining program stability, resource integrity, and security. The use of functions like std::abort(), std::quick\_exit(), and std::Exit() leads to immediate program termination without invoking destructors for objects with automatic, thread, or static storage duration, and without calling exit handlers registered with std::atexit(). |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the call to f(), which was registered as an exit handler with std::at\_exit(), may result in a call to std::terminate() because throwing\_func() may throw an exception. |
| #include <cstdlib>    **void** throwing\_func() noexcept(**false**);    **void** f() { // Not invoked by the program except as an exit handler.  throwing\_func();  }    **int** main() {  **if** (0 != std::**atexit**(f)) {  // Handle error  }  // ...  } |

| **Compliant Code** |
| --- |
| In this compliant solution, f() handles all exceptions thrown by throwing\_func() and does not rethrow. |
| #include <cstdlib>    **void** throwing\_func() noexcept(**false**);    **void** f() { // Not invoked by the program except as an exit handler.  **try** {  throwing\_func();  } **catch** (...) {  // Handle error  }  }    **int** main() {  **if** (0 != std::**atexit**(f)) {  // Handle error  }  // ...  } |

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

| **Principles(s):** Adopting a Secure Coding Standard | Defense in Depth:  The principles “Adopting a Secure Coding Standard” and “Defense in Depth” ensure that exceptions are managed properly to maintain system stability and security. Handling exceptions carefully prevents program crashes and secures resource management. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.7 | exceptionSafety | Identifies instances where exceptions might not be properly handled, ensuring that termination functions like std::abort(), std::quick\_exit, and std::exit() are used appropriately. |
| Visual Studio | 2022 | CA1064: Exceptions should be public, and not throw objects | Detects scenarios where exceptions are not handled gracefully or where termination functions are used inappropriately. |

#### Coding Standard 9

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Data Type** | [STD-009-DCL] | Declarations and Initialization focuses on the proper declaration and initialization of variables in programming. The logical rationale behind this standard is in preventing an array of security vulnerabilities that can arise from improper handling of variable declarations and initializations. |

|  |
| --- |
| **Noncompliant Code** |
| This noncompliant code example uses a C-style variadic function to add a series of integers together. The function reads arguments until the value 0 is found. Calling this function without passing the value 0 as an argument (after the first two arguments) results in undefined behavior. Furthermore, passing any type other than an int also results in undefined behavior. |
| #include <cstdarg>    **int** add(**int** first, **int** second, ...) {  **int** r = first + second;  **va\_list** va;  **va\_start**(va, second);  **while** (**int** v = **va\_arg**(va, **int**)) {  r += v;  }  **va\_end**(va);  **return** r;  } |

|  |
| --- |
| **Compliant Code** |
| In this compliant solution, a variadic function using a function parameter pack is used to implement the add() function, allowing identical behavior for call sites. Unlike the C-style variadic function used in the noncompliant code example, this compliant solution does not result in undefined behavior if the list of parameters is not terminated with 0. Additionally, if any of the values passed to the function are not integers, the code is ill-formed rather than producing undefined behavior. |
| #include <type\_traits>    **template** <**typename** Arg, **typename** std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  **int** add(Arg f, Arg s) { **return** f + s; }    **template** <**typename** Arg, **typename**... Ts, **typename** std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  **int** add(Arg f, Ts... rest) {  **return** f + add(rest...);  } |

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

| **Principles(s):** Adopting a Secure Coding Standard | Validating Input Data:  “Adopting a Secure Coding Standard” and “Validating Input Data” ensure safe use of variadic functions, preventing undefined behavior from improper type handling. Proper declaration and initialization of variables safeguard against security vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.7 | typeSafety | Identifies instances where C-style variadic functions are used without proper type checking, ensuring that only the correct types are passed and processed. |
| Visual Studio | 2022 | CXX\_VARIADIC\_TEMPLATES | Detects the use of C-style variadic functions and recommends the use of C++ variadic templates for type-safe handling of variable arguments. |

#### Coding Standard 10

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard (Memory Protection Coding Standard)** |
| **Memory Protection** | [STD-010-CTR] | Containers in C++ require meticulous management of indices and iterators to ensure that all references and accesses remain within the valid bounds of the container. The logical rationale behind this standard is to prevent security vulnerabilities such as buffer overflows, memory corruption, and undefined behavior that can arise from out-of-bounds access. |

|  |
| --- |
| **Noncompliant Code** |
| This noncompliant code example shows a function, insert\_in\_table(), that has two int parameters, pos and value, both of which can be influenced by data originating from untrusted sources. The function performs a range check to ensure that pos does not exceed the upper bound of the array, specified by tableSize, but fails to check the lower bound. Because pos i declared as a (signed) int, this parameter can assume a negative value, resulting in a write outside the bounds of the memory referenced by table. |
| #include <cstddef>    **void** insert\_in\_table(**int** \*table, std::**size\_t** tableSize, **int** pos, **int** value) {  **if** (pos >= tableSize) {  // Handle error  **return**;  }  table[pos] = value;  } |

|  |
| --- |
| **Compliant Code** |
| In this compliant solution, the parameter pos is declared as size\_t, which prevents the passing of negative arugments. |
| #include <cstddef>    **void** insert\_in\_table(**int** \*table, std::**size\_t** tableSize, std::**size\_t** pos, **int** value) {  **if** (pos >= tableSize) {  // Handle error  **return**;  }  table[pos] = value;  } |

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

| **Principles(s):** Adopting a Secure Coding Standard | Defense in Depth:  “Adopting a Secure Coding Standard” and “Defense in Depth” ensure that container boundaries are respected to prevent out-of-bounds access. Proper management of indices and iterators protects against buffer overflows, ensuring that memory is handled securely. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.7 | boundsCheck | Identifies instances where container accesses may exceed their valid ranges, ensuring that all indices and iterators are properly managed to prevent buffer overflows and memory corruption. |
| Visual Studio | 2022 | CXX\_BOUNDS\_CHECK | Detects instances where containers are accessed without proper bounds checks, recommending the use of safe access methods to prevent out-of-bounds accesses. |

### 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



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.

To incorporate automation into Green Pace’s DevSecOps pipeline, tools should be integrated at multiple stages to ensure compliance with secure coding standards. Automation can begin as early as the “Build” phase, where secure builds are ensured by running static analysis tools such as SonarQube and Cppcheck to enforce coding standards and detect vulnerabilities early. These tools can be integrated into the continuous integration (CI) process to automatically scan code with every commit. In the “Verify and Test” phase, automation will perform dynamic analysis, vulnerability scanning, and compliance checks to ensure the application’s security posture before deployment. This includes running security tests on builds and ensuring trusted repositories and secure open-source libraries are used. As the application transitions into the “Production” phase, tools such as security configuration validators and penetration testing frameworks can be automated to ensure the deployment meets all security configurations. Post-deployment, monitoring and detecting tools like SIEM systems can be automated to alert on any security breaches, ensuring continuous security vigilance and rapid response.

### 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-ASR | **HIGH** | **Unlikely** | **Low** | **HIGH** | **2** |
| STD-002-EXP | **High** | **Unlikely** | **Medium** | **High** | **2** |
| STD-003-INT | **High** | **Unlikely** | **Medium** | **High** | **2** |
| STD-004-SQL | **High** | **Likely** | **Medium** | **High** | **1** |
| STD-005-STR | **High** | **Likely** | **Low** | **High** | **2** |
| STD-006-MEM | **High** | **Likely** | **Medium** | **High** | **2** |
| STD-007-FIO | **High** | **Likely** | **Medium** | **High** | **1** |
| STD-008-ERR | **High** | **Likely** | **Low** | **High** | **2** |
| STD-009-DCL | **High** | **Likely** | **Medium** | **High** | **2** |
| STD-010-CTR | **High** | **Likely** | **Low** | **High** | **2** |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest refers to the protection of data stored on a database, file system, or storage device to prevent unauthorized access when the data is not actively being used. All sensitive data must be encrypted at rest using AES-256 or a stronger encryption standard, and it must be encrypted before being written to a storage device and decrypted upon access by authorized systems. Encryption at rest protects data from unauthorized access and breaches in case of physical theft or unauthorized system access. The policy applies whenever sensitive information is stored, ensuring that data remains secure even when it is not actively being used. |
| Encryption in flight | Encryption in flight refers to protecting data while it is transmitted across networks. This includes data sent between client devices and servers or between internal systems within a secure network. All sensitive data transmitted across internal or external networks must use TLS 1.2 (or higher) or secure VPN connections to ensure encryption in flight. Encryption in flight protects against data breaches during transit by ensuring the data cannot be intercepted or read by attackers. The policy applies whenever data is transmitted over networks, particularly over the internet, where it is most vulnerable to interception. |
| Encryption in use | Encryption in use refers to the protection of data being actively processed or used in memory, typically within applications or databases. This ensures that sensitive information remains protected while it is being accessed by authorized applications. Sensitive data being processed in memory must be protected through secure enclaves or hardware-based encryption mechanisms such as Intel SGX or AMD SEV. Encryption in use is critical to protect sensitive data while it is being processed, especially in cloud systems or systems where memory access could be exploited. The policy applies during any processing of critical data to prevent unauthorized access from malicious insiders or external threats. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication refers to the process of verifying the identity of a user or system attempting to access resources or services. All users and systems must authenticate using multi-factor authentication (MFA) before gaining access to critical systems or sensitive data. Password policies must enforce complexity requirements and mandatory periodic changes, and OAuth or SAML protocols should be used for federated authentication where applicable. Authentication ensures that only authorized users and systems can access sensitive information and services. This policy applies to all user logins and system access attempts to minimize the risk of unauthorized access due to stolen credentials or weak authentication. |
| Authorization | Authorization determines what an authenticated user or system is allowed to do within a system, defining their permissions and access rights. Users and systems must be granted the least privilege necessary to perform their tasks. Access control lists (ACLs) and role-based access control (RBAC) must be implemented to define and enforce user access levels. All changes to access levels must be logged and periodically reviewed for compliance. Authorization ensures that even authenticated users can only access and modify resources that are necessary for their role. This policy applies to access to databases, applications, and files to ensure that no user has more privileges than required, reducing the risk of insider threats and privilege escalation. |
| Accounting | Accounting refers to tracking and recording user actions and changes made to systems or databases, enabling auditing and accountability. All user actions, including logins, access to files, changes to databases, and additions of new users, must be logged with timestamps, user IDs, and the nature of the action. Logs must be stored securely and maintained for a minimum of one year, with periodic reviews for suspicious activity. Accounting provides an audit trail of user actions, allowing for forensic analysis in case of a security incident. This policy applies to monitoring all critical systems, databases, and files to ensure accountability and detect any unauthorized or suspicious activity. |

### 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 | 09/29/2024 | Initial Template | Ty O’Neill |  |
| 1.1 | 10/13/2024 | Completed Template | Ty O’Neill |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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