**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 | An important security principle is validating input data, which means confirming that information entered into a system satisfies predetermined standards before processing. This lowers the possibility of malicious exploits and helps avoid a variety of attacks, including injection attacks, by guaranteeing that only expected and safe data is received. |
| 1. Heed Compiler Warnings | Compiler warnings are alerts that the compiler produces while the code is being compiled that point out possible problems. For security purposes, it is imperative that these warnings be heeded and addressed. Disregarding alerts may result in weaknesses and unexpected actions. Developers can improve the overall resilience of the code and identify any security problems by swiftly addressing compiler warnings. |
| 1. Architect and Design for Security Policies | Security policies outline an organization's approach to security. Designing systems with security policies in mind ensures that the design supports the organization's security objectives. This includes things like access controls, authentication mechanisms, and data security rules. |
| 1. Keep It Simple | One important security principle is simplicity. Complex systems frequently present more potential points of vulnerability. Developers can lessen attack surface and make code easier to understand and secure by making designs and implementations as simple as possible. In general, easy-to-manage solutions with minimal complexity are less likely to introduce unintentional security risks. |
| 1. Default Deny | This principle by default deny all access, only resources that are specifically authorized should be permitted. This method makes sure that access is prohibited unless a specific permission is given. It helps lower the possibility of forgetting security setups and lessens the danger of unwanted access. |
| 1. Adhere to the Principle of Least Privilege | Giving users, processes, and systems the minimal amount of access or rights required to carry out their jobs is the focus of the least privilege principle. This lowers the possibility of unforeseen effects arising from granting users more privileges and decreases the possible harm in the event of a security breach. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data before sending it is crucial when connecting with other systems to avoid contaminating the recipient system with malicious content or code. Validating, purifying, and guaranteeing data integrity are all part of data sanitization, which guards against security flaws such data manipulation and injection attacks. |
| 1. Practice Defense in Depth | Using several tiers of security controls across a system or network is known as defense in depth. This tactic makes sure that additional protection is provided by other levels even if one is compromised. Organizations can improve their capacity to identify, stop, and mitigate different kinds of security risks by expanding their security measures. |
| 1. Use Effective Quality Assurance Techniques | Throughout the development process, quality assurance is essential for spotting and fixing security vulnerabilities. Prior to vulnerabilities being used in production, they can be found and fixed with the aid of comprehensive testing, code reviews, and other quality assurance procedures. This proactive approach helps to create software that is more dependable and secure. |
| 1. Adopt a Secure Coding Standard | Best practices and recommendations for writing secure code are provided by a secure coding standard. Following such guidelines lessens the possibility of introducing common security vulnerabilities and helps to assure consistency. It offers developers a guide on safe coding techniques, addressing things like input validation, handling errors, and encryption. |

### 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] | To ensure code is clear, consistent, and easy to read, avoid using unnecessary qualifiers like const or volatile on reference types. These qualifiers are redundant and can cause confusion without offering real benefits, making the code more difficult to understand. |

| **Noncompliant Code** |
| --- |
| A constant reference (const int& refX) is used to refer to an integer (x). Using const with a reference to an int is redundant because it doesn't provide any additional safety or benefits. It adds complexity to the code without a clear reason, potentially confusing readers. The const-qualified reference is unnecessary in this context and violates the guideline. |
| int main(){  int x = 42;  const int& refX = x; //redundant use of a constant reference  cout<< “Value of x:” <<refX <<endl;  return 0;  } |
|  |

| **Compliant Code** |
| --- |
| The compliant code follows the guideline by not using an unnecessary const qualifier on a reference. The reference (int& refX) is non-const, which improves code clarity and removes redundancy. There's no need for the reference to be constant since it doesn't stop changes to the variable x. |
| int main(){  int x = 42;  int& refX = x; //reference without use of a constant qualifier  cout<< “Value of x:” <<refX <<endl  return 0;  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Converting a value to an enumeration type outside its valid range can cause unpredictable behavior and affect data integrity. Always verify that the value being converted is within the enumeration's valid range to ensure consistency and avoid unexpected problems. |

| **Noncompliant Code** |
| --- |
| The code attempts to convert an integer (value) to an enumeration type (Color) without checking if the integer is within the valid range of values defined by the Color enumeration. If the integer is outside this range (not Red, Green, or Blue), it may cause undefined behavior. This lack of range validation makes the code prone to unexpected behavior and potential errors. |
| enum Color { Red, Green, Blue };  void setColor(int value) {  Color color = static\_cast<Color>(value);// Casting without range  //validation  // Process color...  } |

| **Compliant Code** |
| --- |
| The compliant code verifies if the integer value is within the valid range of the Color enumeration before performing the cast. If the value is out of range, an error-handling mechanism is triggered. This ensures the cast is only executed when the integer matches a valid Color enumerator, preventing undefined behavior and improving code robustness. |
| enum Color { Red, Green, Blue };  void setColor(int value) {  if (value >= Red && value <= Blue) {// Validation check for the range  Color color = static\_cast<Color>(value); // Casting with range  //validation  // Process color...  }  else {  // Handle error for out-of-range value  }  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Make sure that string storage has enough room for both character data and the null terminator. This approach helps prevent buffer overflows and ensures correct string handling in C++. |

| **Noncompliant Code** |
| --- |
| The function processString tries to copy the string "Hello" into a buffer (str) with a given length. However, the calculation for copyLength doesn't include the null terminator. If the buffer's length matches the length of "Hello" exactly, there won't be room for the null terminator, which could cause buffer overflows and undefined behavior. |
| #include <cstring>  void processString(char\* str, size\_t length) {  // Insufficient space for the null terminator  size\_t copyLength = std::min(length, strlen("Hello"));  std::strncpy(str, "Hello", copyLength);  // Process string...  } |

| **Compliant Code** |
| --- |
| The compliant code resolves the issue by calculating copyLength to allow room for the null terminator. It also explicitly adds the null terminator to ensure the string is correctly terminated. This guarantees the buffer is sufficiently large to hold both the character data and the null terminator, preventing buffer overflows and ensuring proper string handling. |
| #include <cstring>  void processString(char\* str, size\_t length) {  // Guarantee sufficient space for the null terminator  size\_t copyLength = std::min(length - 1, strlen("Hello"));  std::strncpy(str, "Hello", copyLength);  str[copyLength] = '\0'; // Ensure null terminator  // Process string...  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | When developers use file streams for both reading and writing, it's important to adjust the file stream cursor with positioning functions (like seekg, seekp) before switching operations. This practice avoids unexpected issues and maintains the integrity of file operations. Failing to properly position the cursor when alternating between input and output can lead to SQL injection risks, as it may cause untrusted data to be mishandled in SQL queries. |

| **Noncompliant Code** |
| --- |
| The noncompliant code switches between reading and writing on a file stream without using a positioning call in between. This can cause unexpected issues, like overwriting existing data, reading incorrect information, or corrupting data. Not repositioning the file pointer before changing operations can lead to unintended behavior. |
| #include <fstream>  #include <iostream>  void processFile(const std::string& filename, const std::string&  userInput) {  std::fstream file(filename, std::ios::in | std::ios::out);    // Noncompliant: Alternating input and output without positioning  file << "User input: " << userInput << std::endl;  std::string dataFromUser;  file >> dataFromUser;    // Process dataFromUser...  } |

| **Compliant Code** |
| --- |
| The compliant code achieves correct positioning by using seekp before transitioning from output to input operations on the file stream. This moves the file pointer to the start (std::ios::beg) before input, preventing unintended outcomes and potential file content corruption. Proper repositioning ensures that file operations are clearly defined and do not interfere with each other. |
| #include <fstream>  #include <iostream>  void processFile(const std::string& filename, const std::string&userInput) {  std::fstream file(filename, std::ios::in | std::ios::out);  // Compliant: Repositioning before alternating input and output  file << "User input: " << userInput << std::endl;  file.seekp(0, std::ios::beg); // Reposition to the beginning before  input  std::string dataFromUser;  file >> dataFromUser;  // Process dataFromUser...  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | In C++, dynamic memory allocation is often used to allocate memory for objects or arrays with sizes that aren't known at compile time. It's crucial to properly deallocate this memory to return resources to the system and avoid memory leaks. This rule applies to all dynamically allocated resources, such as memory, file handles, network connections, and any other resources obtained through dynamic allocation. Not deallocating memory can result in memory leaks, where unreleased memory builds up over time, potentially causing the program to run out of memory. |

| **Noncompliant Code** |
| --- |
| Dynamically allocated memory is not deallocated using delete[], leading to a memory leak. Failing to release  dynamically allocated memory can result in inefficient memory usage and potential memory depletion. |
| #include <iostream>  void allocateMemory() {  int\* dynamicArray = new int[10];  // Process dynamicArray...  // Missing deallocation  } |

| **Compliant Code** |
| --- |
| The compliant code properly deallocates the dynamically allocated memory using delete[]. This ensures that  the memory is released back to the system, preventing memory leaks. |
| #include <iostream>  void allocateMemory() {  int\* dynamicArray = new int[10];  // Process dynamicArray...  delete[] dynamicArray; // Proper deallocation  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [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** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | To evaluate a constant expression, employ a static assertion (static\_assert) at compile time. This approach validates the condition during compilation, allowing for early error detection and removing the runtime overhead linked with dynamic assertions. |

| **Noncompliant Code** |
| --- |
| A runtime assertion (assert) is employed to verify a constant expression (MAX\_VALUE). Typically, assertions are meant for runtime checks, but in this scenario, the condition is determined at compile time. Utilizing a runtime assertion adds unnecessary overhead during execution, and any issues with the constant expression won't be identified until runtime. |
| #include <cassert>  constexpr int MAX\_VALUE = 100;  void processData(int value) {  assert(value <= MAX\_VALUE); // Noncompliant: Runtime assertion for a  //constant expression  // Process data...  } |

| **Compliant Code** |
| --- |
| A static assertion (static\_assert) is utilized to check the constant expression (MAX\_VALUE) during compilation. The compiler evaluates the condition, and if it fails, a compilation error with a specific message is generated. This approach allows for early detection of issues related to the constant expression without adding runtime overhead. |
| #include <cassert>  constexpr int MAX\_VALUE = 100;  void processData(int value) {  static\_assert(value <= MAX\_VALUE, "Value exceeds maximum allowed");  // Compliant: Static assertion for a constant expression  // Process data...  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Proper exception handling is crucial for developing stable and dependable software. Failing to address exceptions correctly can cause erratic behavior, complicate debugging, and lead to runtime problems. |

| **Noncompliant Code** |
| --- |
| The riskyOperation() function might throw an exception, but the catch block within the function captures it without handling or rethrowing it. Consequently, the main function continues without resolving the potential issues caused by the exception. Overlooking exceptions in this way can result in unpredictable behavior and makes it difficult to identify and fix problems. |
| #include <iostream>  void riskyOperation() {  try {  // Risky operation that may throw an exception  throw std::runtime\_error("An exception occurred");  } catch (std::exception& e) {  // Catching the exception, but not handling or rethrowing  }  }  int main() {  riskyOperation();  // Code continues after riskyOperation, potential issues not  addressed  return 0;  } |

| **Compliant Code** |
| --- |
| The riskyOperation() function is updated to handle exceptions correctly. The catch block within the function outputs an error message to std::cerr and may rethrow the exception. In the main() function, an extra try-catch block is included to manage exceptions at a higher level. This ensures proper logging or handling of exceptions and prevents the program from continuing with unresolved issues. |
| #include <iostream>  void riskyOperation() {  try {  // Risky operation that may throw an exception  throw std::runtime\_error("An exception occurred");  } catch (std::exception& e) {  // Proper handling or rethrowing of the exception  std::cerr << "Exception caught: " << e.what() << std::endl;  // Optionally rethrow the exception  throw;  }  }  int main() {  try {  riskyOperation();  // Code continues after riskyOperation, potential issues  addressed  } catch (std::exception& e) {  // Handle or log the exception at a higher level  std::cerr << "Main function caught exception: " << e.what() <<  std::endl;  }  return 0;  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| OOP | [STD-008-CPP] | Object-oriented programming techniques and efficient memory management serve as the foundation for this standard. It emphasizes how crucial it is to use virtual destructors in conjunction with polymorphic objects to guarantee that resources are properly and thoroughly cleaned away. This method aids in the prevention of errors, undefined behaviors, memory leaks, and resource leaks. |

| **Noncompliant Code** |
| --- |
| A pointer of Base type is assigned to a polymorphic object of the Derived type. The destructor of the Base class is the only one invoked when the delete operation is applied to this polymorphic object; the destructor of the Derived class is not called. This results in unclear behavior and insufficient cleanup. Without a virtual destructor, deleting a polymorphic object might result in resource leaks and improper program behavior. |
| #include <iostream>  class Base {  public:  Base() {  std::cout << "Base constructor" << std::endl;  }  ~Base() {  std::cout << "Base destructor" << std::endl;  }  };  class Derived : public Base {  public:  Derived() {  std::cout << "Derived constructor" << std::endl;  }  ~Derived() {  std::cout << "Derived destructor" << std::endl;  }  };  int main() {  Base\* polymorphicObject = new Derived();  delete polymorphicObject; // Noncompliant: Deleting a polymorphic  object without a virtual destructor  return 0;  } |

| **Compliant Code** |
| --- |
| Because the base class (Base) contains a virtual destructor, the derived class (Derived)'s correct destructor is also called when a pointer to the base class is removed. It is crucial to correctly clean up resources in the derived class when deleting polymorphic objects, and this is ensured by the base class's usage of a virtual destructor. |
| #include <iostream>  class Base {  public:  Base() {  std::cout << "Base constructor" << std::endl;  }  virtual ~Base() {  std::cout << "Base destructor" << std::endl;  }  };  class Derived : public Base {  public:  Derived() {  std::cout << "Derived constructor" << std::endl;  }    ~Derived() override {  std::cout << "Derived destructor" << std::endl;  }  };  int main() {  Base\* polymorphicObject = new Derived();  delete polymorphicObject; // Compliant: Deleting a polymorphic  //object with a virtual destructor  return 0;  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | [STD-009-CPP] | To prevent iterator invalidation, perform bounds checks, preserve memory safety, and prevent undefined behavior, references, pointers, and iterators must all be valid when accessing elements in a container. This ensures that these components are correctly accessed and stay inside the boundaries of the container. |

| **Noncompliant Code** |
| --- |
| Undefined behavior occurs when an iterator (invalidIterator) is used to attempt to retrieve an element that is outside of the valid range of the container (numbers). This happens because it makes memory reads outside of the boundaries of the vector, which can lead to unexpected program states, crashes, and corrupted data. |
| #include <iostream>  #include <vector>  int main() {  std::vector<int> numbers = {1, 2, 3, 4, 5};  // Noncompliant: Using an invalid iterator  std::vector<int>::iterator invalidIterator = numbers.begin() + 10;  \*invalidIterator = 42; // Undefined behavior, accessing beyond  //container bounds  return 0;  } |

| **Compliant Code** |
| --- |
| The conforming code demonstrates the usage of a valid iterator (validIterator) that remains inside the valid range of the container (numbers). Before dereferencing and altering the value, it contains a check to make sure the iterator is not at or past the end() point. By using this technique, elements outside of the container's bounds are kept inaccessible to undefined behavior. |
| #include <iostream>  #include <vector>  int main() {  std::vector<int> numbers = {1, 2, 3, 4, 5};  // Compliant: Using a valid iterator  std::vector<int>::iterator validIterator = numbers.begin() + 2;  if (validIterator != numbers.end()) {  \*validIterator = 42; // Valid operation within container bounds  } else {  // Handle case where iterator is at or beyond the end  std::cerr << "Invalid iterator position." << std::endl;  }  return 0;  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | [STD-010-  CPP] | Maintaining memory integrity, avoiding undefined behaviors, preventing buffer overflows, and improving code quality are all facilitated by making sure library functions don't overrun. For safety reasons, it is imperative to secure library functions because buffer overflows can lead to unpredictable behavior as well as memory corruption. |

| **Noncompliant Code** |
| --- |
| An element at index 5, which is beyond the vector's valid range, can be accessed using the std::vector method at(). If the index is invalid, the at() function throws std::out\_of\_range after checking the boundaries. But in this case, the exception is not handled, which could lead to program shutdown and unknown behavior. It is not guaranteed that library functions will not overrun by the noncompliant code. |
| #include <iostream>  #include <vector>  int main() {  const size\_t size = 3;  std::vector<int> numbers(size);  // Noncompliant: Using library function without bounds checking  int value = numbers.at(5); // Throws std::out\_of\_range, but not  //caught  return 0;  } |

| **Compliant Code** |
| --- |
| When an index is out of bounds, the at() method raises a std::out\_of\_range exception, which is handled by the complying code using a try-catch block. The code prevents undefined behavior and enables error handling by controlling the exception. This method incorporates adequate bounds checking and error handling to prevent library functions from overflowing. |
| #include <iostream>  #include <vector>  int main() {  const size\_t size = 3;  std::vector<int> numbers(size);  // Compliant: Using library function with bounds checking  try {  int value = numbers.at(2); // Valid index within the vector's  //range  // Process value...  } catch (const std::out\_of\_range& e) {  // Handle the out-of-range exception  std::cerr << "Out of range exception: " << e.what() << std::endl;  }  return 0;  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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.

[Insert your written explanations here.]

### 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 | Medium | Likely | Low | Medium | 2 |
| STD-002-CPP | High | Likely | Medium | High | 3 |
| STD-003-CPP | Moderate | Likely | Low | Medium | 2 |
| STD-004-CPP | Medium | Likely | Low | Medium | 3 |
| STD-005-CPP | High | Likely | Low | High | 2 |
| STD-006-CPP | Medium | Unlikely | Low | Medium | 2 |
| STD-007-CPP | Moderate | Medium | Low | High | 2 |
| STD-008-CPP | High | Medium | Medium | High | 4 |
| STD-009-CPP | High | Medium | Medium | High | 4 |
| STD-010-CPP | Medium | Medium | Medium | Medium | 3 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

### 

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest is the process of safeguarding data while it is stored in databases, file systems, or any other storage device. This policy is applicable to all sensitive and confidential data that is stored within the organization's systems. |
| Encryption in flight | Encryption in flight ensures the security of data during transmission over networks. This policy is crucial for safeguarding data from interception and unauthorized access during transmission. |
| Encryption in use | Data is safeguarded during the processing or utilization of applications by encryption. This policy is essential for the protection of sensitive information during active processing. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of confirming the identity of users and systems that are able to access the resources of the organization. This policy guarantees that sensitive information is accessible only to authorized entities. |
| Authorization | The level of access that authenticated users are permitted is determined by authorization. This policy guarantees that users are permitted to access only the resources that are essential for their respective responsibilities. |
| Accounting | Accounting entails the surveillance and recording of user activities within the organization's systems. This policy is essential for the preservation of an audit trace of user actions. |

**\***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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [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 |

Reference

SEI CERT C++ Coding Standard

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>