**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 | Validation is the process of ensuring that input data falls within the acceptable range for a program. This involves making sure that inputs meet the class or subsystem's input criteria and adhere to requirements regarding type and numerical range. |
| 1. Heed Compiler Warnings | When encountering compiler warnings, it is best to address the underlying issues in the code rather than simply adding comments to explain them away. Understanding the cause of the warning and finding solutions such as using consistent types and minimizing type casts is more effective than superficially masking the warnings with formatting changes. |
| 1. Architect and Design for Security Policies | To enforce security policies effectively, design your program with a software architecture that divides the system into distinct, communicating subsystems. Each subsystem should have appropriate privileges assigned to it, especially if different privileges are required at various stages of operation. |
| 1. Keep It Simple | Simplify designs to reduce implementation errors and ensure correct use. Complex designs increase the likelihood of errors in configuration and usage. Additionally, increased complexity in security measures requires more effort to achieve the desired level of assurance. |
| 1. Default Deny | Implement an authorization-based access control system where access is denied by default. The protection system defines the terms and conditions for granting access, rather than excluding individuals outright. |
| 1. Adhere to the Principle of Least Privilege | Follow the principle of least privilege: Processes should operate with minimal privileges necessary for the task. Higher privileges should only be utilized for the duration needed to complete the specific privileged task. This approach limits attackers' ability to execute arbitrary actions. |
| 1. Sanitize Data Sent to Other Systems | Data sent to sophisticated subsystems must be sanitized to prevent exploitation through SQL, command, or other injection techniques. The calling process, aware of the context, should handle data cleansing before invoking the subsystem, as the subsystems lack context understanding and may inadvertently expose vulnerabilities. |
| 1. Practice Defense in Depth | Employ multiple layers of defense to mitigate risk: if one layer fails, another can prevent security flaws from becoming exploitable vulnerabilities or minimize the impact of successful exploits. Secure programming practices combined with secure runtime environments can reduce the likelihood of code vulnerabilities being exploited in the operational environment. |
| 1. Use Effective Quality Assurance Techniques | High-quality assurance procedures like fuzz testing, penetration testing, and source code audits can help find and address vulnerabilities. Systems benefiting from independent security examinations tend to be more secure as external reviewers offer fresh insights, including correcting false assumptions. |
| 1. Adopt a Secure Coding Standard | Develop and implement a secure coding standard tailored to your chosen programming language and platform. |

### 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** | **Avoid casting to an out-of-range enumeration value.** |
| --- | --- | --- |
| **Data Type** | [STD-001-  CPP] | In C++, there are two types of enumerations: scoped enumerations with a fixed underlying type and unscoped enumerations with a potentially unfixed underlying type. Both forms of enumeration can represent values beyond those explicitly listed. The range of valid enumeration values is determined by the C++ Standard in paragraph 8 of [dcl.enum]. |

| **Noncompliant Code** |
| --- |
| The code example attempts to verify if a value falls within the acceptable range of an enumeration. However, it casts the value to the enumeration type before this check, risking representation of values outside the valid range. This may lead to unspecified behavior due to potential integer values that are not valid in the EnumType on a two's complement system. |
| **enum** EnumType {  First,  Second,  Third  };  **void** f(**int** intVar) {  EnumType enumVar = **static\_cast**<EnumType>(intVar);  **if** (enumVar < First || enumVar > Third) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| The solution, in compliance, verifies if the value aligns with the enumeration type before conversion to prevent any potential unspecified outcome. This is achieved by constraining the converted value to a specific enumerator within the type. |
| **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):** [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** | **Employ valid references, pointers, and iterators to access elements of a `basic\_string`.** |
| --- | --- | --- |
| **Data Value** | [STD-002-  CPP] | Given that std::basic\_string functions as a character container, this guideline aligns with CTR51-CPP, advising the use of valid references, pointers, and iterators for container element referencing. Similar to other STL containers, std::basic\_string supports iterators; nonetheless, it possesses distinctive invalidation semantics. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code snippet, input is copied into a std::string, where semicolon (;) characters are substituted with spaces. The code is noncompliant due to the invalidation of the iterator loc after the initial insert() call. Subsequent insert() invocations exhibit undefined behavior. |
| #include <string>  **void** f(**const** std::string &input) {  std::string email;  // Copy input into email converting ";" to " "  std::string::iterator loc = email.begin();  **for** (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {  email.insert(loc, \*i != ';' ? \*i : ' ');  }  } |

| **Compliant Code** |
| --- |
| The iterator `loc` is updated after each `insert()` call in this compliant solution to avoid accessing an invalidated iterator. After the loop, the updated iterator is incremented. |
| #include <string>  **void** f(**const** std::string &input) {  std::string email;  // Copy input into email converting ";" to " "  std::string::iterator loc = email.begin();  **for** (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {  loc = email.insert(loc, \*i != ';' ? \*i : ' ');  }  } |

**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** | **Performing range checks when accessing elements.** |
| --- | --- | --- |
| **String Correctness** | [STD-003-  CPP] | The `operator[]` methods of `std::string`, `const\_reference operator[](size\_type) const`, and `reference operator[](size\_type)` return the character at position `pos`. If `pos >= size()`, they return a reference to a `charT` object with value `charT()`. These operators are unchecked for range errors, hence no exceptions are thrown. Modifying an out-of-range object leads to undefined behavior. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example may lead to undefined behavior because the value returned by `get\_index()` could exceed the number of elements stored in the string. |
| #include <string>  **extern** std::**size\_t** get\_index();  **void** f() {  std::string s("01234567");  s[get\_index()] = '1';  } |

| **Compliant Code** |
| --- |
| This compliant solution utilizes the `std::basic\_string::at()` function, similar to the index `operator[]`, but it throws a `std::out\_of\_range` exception if `pos >= size()`. |
| #include <stdexcept>  #include <string>  **extern** std::**size\_t** get\_index();  **void** f() {  std::string s("01234567");  **try** {  s.at(get\_index()) = '1';  } **catch** (std::out\_of\_range &) {  // Handle error  }  } |

**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** | **Avoid storing an already-owned pointer value in an unrelated smart pointer.** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-  CPP] | Smart pointers like `std::unique\_ptr` and `std::shared\_ptr` integrate pointer ownership semantics into the type system. They encapsulate a pointer value, offer pointer-like behavior via `operator\*()` and `operator->()` member functions, and manage the pointer's lifespan. The pointer passed into a smart pointer during construction is considered owned by that smart pointer. |

| **Noncompliant Code** |
| --- |
| Two unassociated smart pointers are generated using the identical underlying pointer value in this noncompliant code instance. Upon the deletion of the local automatic variable p2, it also erases the managed pointer value. Consequently, when the local automatic variable p1 is also deleted, it removes the same pointer value, leading to a double-free vulnerability. |
| #include <memory>  **void** f() {  **int** \*i = **new int**;  std::shared\_ptr<**int**> p1(i);  std::shared\_ptr<**int**> p2(i);  } |

| **Compliant Code** |
| --- |
| In this compliant approach, std::shared\_ptr objects are interconnected via copy construction. Upon the deletion of the local automatic variable p2, the shared pointer's use count decreases but remains above zero. Subsequently, when the local automatic variable p1 is destroyed, the use count drops to zero, leading to the destruction of the managed pointer. This compliance strategy also employs std::make\_shared() instead of directly allocating a raw pointer and storing it in a local variable. |
| #include <memory>  **void** f() {  std::shared\_ptr<**int**> p1 = std::make\_shared<**int**>();  std::shared\_ptr<**int**> p2(p1);  } |

**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** | **Correctly release dynamically allocated resources.** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-  CPP] | C offers multiple methods like std::malloc(), std::calloc(), and std::realloc() for memory allocation, usable in C++ code. Nonetheless, when it comes to memory deallocation, C only has one function, std::free(). It's crucial to follow guidelines like MEM31-C and MEM34-C, which emphasize freeing dynamically allocated memory when no longer required, ensuring proper handling as per C's allocation and deallocation standards. |

| **Noncompliant Code** |
| --- |
| The noncompliant code exhibits passing the local variable space as the argument to the placement new operator, followed by passing the resulting pointer to ::operator delete(). This action triggers undefined behavior since ::operator delete() tries to release memory not allocated by ::operator new(). |
| #include <iostream>  **struct** S {  S() { std::cout << "S::S()" << std::endl; }  ~S() { std::cout << "S::~S()" << std::endl; }  };  **void** f() {  alignas(**struct** S) **char** space[**sizeof**(**struct** S)];  S \*s1 = **new** (&space) S;  // ...  **delete** s1;  } |

| **Compliant Code** |
| --- |
| In this compliant resolution, the use of ::operator delete() is eliminated, with the destructor of s1 being explicitly invoked instead. This scenario justifies the explicit invocation of a destructor, deviating from typical practices. |
| #include <iostream>  **struct** S {  S() { std::cout << "S::S()" << std::endl; }  ~S() { std::cout << "S::~S()" << std::endl; }  };  **void** f() {  alignas(**struct** S) **char** space[**sizeof**(**struct** S)];  S \*s1 = **new** (&space) S;  // ...  s1->~S();  } |

**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** | **Employ a static assertion to examine the value of a constant expression.** |
| --- | --- | --- |
| **Assertions** | [STD-006-  CPP] | Runtime assertions are a crucial diagnostic mechanism for detecting and eradicating software flaws that could potentially lead to security vulnerabilities. Despite being beneficial, the runtime assert() macro has drawbacks such as increasing runtime overhead and invoking abort(), restricting its utility to pinpointing erroneous assumptions rather than performing real-time error verification. Given these limitations, runtime assertions are typically deemed unsuitable for server applications or embedded systems. |

| **Noncompliant Code** |
| --- |
| This code that does not comply utilizes the assert() macro to affirm a crucial property related to a memory-mapped structure required for the proper functioning of the code. |
| #include <assert.h>  **struct** timer {  unsigned **char** MODE;  unsigned **int** DATA;  unsigned **int** COUNT;  };  **int** func(**void**) {  **assert**(**sizeof**(**struct** timer) == **sizeof**(unsigned **char**)  + **sizeof**(unsigned **int**) + **sizeof**(unsigned **int**));  } |

| **Compliant Code** |
| --- |
| When dealing with assertions that exclusively involve constant expressions, a compliant approach involves utilizing a preprocessor conditional statement, exemplified in this solution. |
| **struct** timer {  unsigned **char** MODE;  unsigned **int** DATA;  unsigned **int** COUNT;  };  #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned  int) + sizeof(unsigned int)))  #error "Structure must not have any padding"  #endif |

**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** | **Ensure proper resource management during exception handling to prevent resource leaks.** |
| --- | --- | --- |
| **Exceptions** | [STD-007-  CPP] | It is crucial to reclaim resources in the event of exceptions. When an exception occurs, there is a risk that cleanup code might be skipped, or an object could remain partially initialized. This incomplete object would breach basic exception safety principles, as outlined in ERR56-CPP. To ensure exceptional safety, it is best to have resources automatically reclaimed by employing the RAII design pattern [Stroustrup 2001] as objects leave scope. This approach eliminates the necessity for intricate cleanup code during resource allocation. |

| **Noncompliant Code** |
| --- |
| In this code example that does not comply with standards, pst is not correctly released if process\_item throws an exception, leading to a resource leak. |
| #include <new>  **struct** SomeType {  SomeType() noexcept; // Performs nontrivial initialization.  ~SomeType(); // Performs nontrivial finalization.  **void** process\_item() noexcept(**false**);  };  **void** f() {  SomeType \*pst = **new** (std::**nothrow**) SomeType();  **if** (!pst) {  // Handle error  **return**;  }  **try** {  pst->process\_item();  } **catch** (...) {  // Process error, but do not recover from it; rethrow.  **throw**;  }  **delete** pst;  } |

| **Compliant Code** |
| --- |
| In this solution that follows the guidelines, the exception handler releases pst by invoking the delete operation. |
| #include <new>  **struct** SomeType {  SomeType() noexcept; // Performs nontrivial initialization.  ~SomeType(); // Performs nontrivial finalization.  **void** process\_item() noexcept(**false**);  };  **void** f() {  SomeType \*pst = **new** (std::**nothrow**) SomeType();  **if** (!pst) {  // Handle error  **return**;  }  **try** {  pst->process\_item();  } **catch** (...) {  // Process error, but do not recover from it; rethrow.  **delete** pst;  **throw**;  }  **delete** pst;  } |

**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** | **Ensure that library functions do not encounter overflow situations.** |
| --- | --- | --- |
| [Student Choice] | [STD-008-  CPP] | When data is copied into a container that lacks sufficient space, a buffer overflow occurs. To avert these mistakes, the data copied to the target container should be limited based on the container's size, or ideally, the target container should be ensured to be adequately sized to accommodate the data being copied. |

| **Noncompliant Code** |
| --- |
| STL containers are susceptible to similar vulnerabilities as array data types. The std::copy() algorithm lacks inherent bounds checking, which can result in a buffer overflow. In this code example that does not comply with standards, a vector of integers is copied from src to dest using std::copy(). Since std::copy() does not expand the dest vector, the program will encounter a buffer overflow when copying the first element. |
| #include <algorithm>  #include <vector>  **void** f(**const** std::vector<**int**> &src) {  std::vector<**int**> dest;  std::copy(src.begin(), src.end(), dest.begin());  // ...  } |

| **Compliant Code** |
| --- |
| To utilize std::copy() correctly, it's essential to guarantee that the destination container has enough space to accommodate all elements being copied. In this compliant approach, the vector's capacity is increased before the copy operation takes place. |
| #include <algorithm>  #include <vector>  **void** f(**const** std::vector<**int**> &src) {  // Initialize dest with src.size() default-inserted elements  std::vector<**int**> dest(src.size());  std::copy(src.begin(), src.end(), dest.begin());  // ...  } |

**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** | **Utilize valid iterator ranges.** |
| --- | --- | --- |
| [Student Choice] | [STD-009-  CPP] | While iterating through the elements of a container, the iterators employed should traverse a valid range. An iterator range consists of a pair of iterators that point to the first element and one past the end of the range, respectively. |

| **Noncompliant Code** |
| --- |
| In the provided noncompliant example, two iterators delineate a range within the same container, with the first iterator not preceding the second. The loop in std::for\_each() compares the iterators for equality, incrementing the first one until they match. However, increasing the iterator representing the end of the range causes undefined behavior. |
| #include <algorithm>  #include <iostream>  #include <vector>  **void** f(**const** std::vector<**int**> &c) {  std::for\_each(c.end(), c.begin(), [](**int** i) { std::cout << i; });  } |

| **Compliant Code** |
| --- |
| In this compliant resolution, the iterator values supplied to std::for\_each() are passed in the correct sequence. |
| #include <algorithm>  #include <iostream>  #include <vector>  **void** f(**const** std::vector<**int**> &c) {  std::for\_each(c.begin(), c.end(), [](**int** i) { std::cout << i; });  } |

**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** | **Ensure that the storage allocated for strings has ample room for character data along with the null terminator.** |
| --- | --- | --- |
| [Student Choice] | [STD-010-  CPP] | Copying data to a buffer size insufficient to contain it leads to a buffer overflow, a common issue when handling strings. To avert these errors, restrict copies via truncation or, better yet, confirm that the destination can accommodate the data. C-style strings necessitate a null character to denote the end, whereas the C++ std::basic\_string template does not need such a character. |

| **Noncompliant Code** |
| --- |
| Due to the unbounded nature of the input, the code below has the potential to trigger a buffer overflow. |
| #include <iostream>    **void** f() {  **char** buf[12];  std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| The optimal approach to prevent data truncation and buffer overflows is to utilize std::string instead of a limited array, as demonstrated in this compliant solution. |
| #include <iostream>  #include <string>  **void** f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

**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 | High | Unlikely | Medium | High | 2 |
| [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.] | [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.] | [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.] |
| [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.] | [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.] |

### 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 | [Insert text.] |
| Encryption in flight | [Insert text.] |
| Encryption in use | [Insert text.] |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | [Insert text.] |
| Authorization | [Insert text.] |
| Accounting | [Insert text.] |

**\***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 |