**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 | Any data received from external sources should be validated and sanitized to prevent malicious input from harming the system. This can be done by ensuring that input data meets the expected format, range, and length. |
| 1. Heed Compiler Warnings | Developers should pay attention to warnings that C++ compilers often provide for code that may be insecure and take appropriate action to address any potential security issues. Ignore compiler warnings can lead to vulnerabilities that can be exploited by attackers. |
| 1. Architect and Design for Security Policies | Developers should design and implement security policies into their C++ applications from the beginning. They should consider security from the early stages of the software development lifecycle (SDLC) and ensuring that the architecture of the application is secure. |
| 1. Keep It Simple | Simple code is less likely to contain vulnerabilities. Developers should keep their C++ codes as simple as possible to reduce the likelihood of introducing security flaws. |
| 1. Default Deny | The access to resources should be denied by default and only granted to authorized users. This can be done by using authentication and access control mechanisms. |
| 1. Adhere to the Principle of Least Privilege | In the principle of least privilege, users should be given the minimum level of access required to perform their tasks. This can reduce the impact of a security breach by limiting the resources available to an attacker. |
| 1. Sanitize Data Sent to Other Systems | Before sending to other systems, data should be sanitized to prevent malicious code from being transmitted. This can be done by coding and decoding data using secure protocols. |
| 1. Practice Defense in Depth | Every organization should apply Defense in Depth (DiD) that implements multiple layers of security controls to protect against different types of attacks. DiD can be deployed by using firewalls, detection systems, and access control mechanisms. |
| 1. Use Effective Quality Assurance Techniques | Developers should use effective quality assurance techniques to test their C++ code for security vulnerabilities, such as static code analysis, dynamic testing, and manual code reviews. |
| 1. Adopt a Secure Coding Standard | Developers should adopt a secure coding standard that provide guidelines and best practices for developing secure C++ code. Developers can look for coding standards in CERT C++ for example, which provide guidelines for writing secure C++ code. |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-030-CPP | Declare objects with appropriate types. This guideline recommends declaring objects with appropriate types that they are using correctly and safely. Developers should use the smallest possible type that can represent the range of values needed by the project. Ref.: DCL30-C. |

| **Noncompliant Code** |
| --- |
| int p = 0 //Declare p as an int |
| In this code, p is declared as an int using ‘int’ type, even though it is only used for non-negative values. It potentially causes integer overflow or other issues if ‘p’ is used to represent a larger value than an ‘int’ can hold. |

| **Compliant Code** |
| --- |
| sign\_t p = 0; //Declare p as an unsigned integer type in C++ |
| This code uses “uint32\_t” data type to store an unassigned 32-bit integer value. This code is a compliant code with Data Type Coding Standard as it used a fixed-sized integer type and avoids the use of platform-specific integer types. |

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

| **Principles(s):** 1. Validate Input Data – The standard “Input Validation” (Section DCL30-C) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Moderate-High | High | Implementation |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Checkmarx | 9.5.0 | Input Validation | Checkmarx is a static analysis tool that can be used to identify potential security vulnerabilities in C and C++ code. It includes a checker that can detect issues related to input validation like command injection and SQL injection |
| Coverity | 2022.12.2 | Input Validation | Coverity is a static analysis tool that can be used to identify potential security vulnerabilities in C and C++ code. It includes a checker that can detect issues related to input validation, such as buffer overflows, and format string vulnerabilities |
| SonarQube | 10.0 | Input Validation | SonarQube is a static analysis tool that can be used to identify potential security vulnerabilities in C and C++ code. It includes a checker that can detect issues related to input validation, such as cross-site scripting (XSS) and SQL injection. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-033-CPP | Ensure that unsigned integer operations do not wrap.  When performing operations on unsigned integers, it is important to ensure the operation does not exceed the maximum value of the data type. If it does, the value will wrap around to zero, leading to unexpected behavior and potentially causing errors in the programs. The coding standard recommends checking for potential overflow before performing any unsigned integer operation. Ref.: INT30-C. |

| **Noncompliant Code** |
| --- |
| unsigned int a = 4294967295;  unsigned int b = 1;  unsigned int c = a + b; // This causes integer wraparound |
| In this code, ‘a’ is the max. value that can be stored in unsigned integer of 32 bits and adding 1 will cause integer wraparound that resulting in ‘c’ having a value of 0 instead of 4294967296. |

| **Compliant Code** |
| --- |
| #include <cstdint> // For uint32\_t  uint32\_t a = 4294967295U;  uint32\_t b = 1U;  uint32\_t c;  if (b <= (a - c)) { // Check for possible wraparound  c = a + b;  } else {  // Handle the error condition appropriately  } |
| In this code, ‘uint32\_t’ is used instead of ‘unsigned int’ that ensures the variable is a 32-bits unsigned integer on all platforms. The ‘if’ statement checks whether adding ‘b’ to ‘c’ would cause integer wraparound. |

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

| **Principles(s):** 2. Heed Compiler Warnings |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 17.0 | Compiler Warnings | Detect issues related to ignoring or dismissing compiler warnings, such as unused variables and potential null pointer dereferences. |
| SonarQube | 10.0 | Compiler Warnings | Detect issues related to ignoring or dismissing compiler warnings, such as unused variables and potential null pointer dereferences. |
| PVS-Studio | 7.24 | Compiler Warnings | Detect issues related to ignoring or dismissing compiler warnings, such as unused variables and unused functions. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-050-CPP | Guarantee that storage for the strings has sufficient space for character data and the null terminator. The null terminator is a special character that indicates the end of the string, and is represented by the value 0. Developers should allocate one extra byte of memory beyond the size of the string for the null terminator. Ref.: STR31-C. |
|  |  |  |

| **Noncompliant Code** |
| --- |
| char name[10];  strcpy(name, ”Hai Nguyen”); //Error not enough space allocated for null terminator. |
| This code has allocated an array of 10 characters for the name string, but has not allocated space for the null terminator. When the strcpy() function tries to add the null terminator to the end of the string, it will write past the end of the allocated space, causing undefined behavior. |

| **Compliant Code** |
| --- |
| char name[12];  strcpy(name, ”Hai Nguyen”); //Enough space allocated for null terminator. |
| This code is a compliant example that has enough space to store 10 characters of the name plus the null terminator. |

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

| **Principles(s):** 3. Architect and Design for Security Policies |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | High | High | Design |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP ASVS | 5.0 | Security Policies | The ASVS includes requirements for authentication and session management, access control, cryptography, error handling, etc. |
| Microsoft Threat Modeling | 7.3.21108.2 | TMT | Microsoft Threat Modeling Tool can be used to identify and mitigate potential security threats that provides a graphical interface for creating threat models. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-054-CPP | Sanitize input to prevent SQL Injection attacks. Ref.: CWE-89, MEM10-C.  Using prepared statements, limit the user input, validate the input, parameterized queries instead of building SQL queries dynamically from user input are among the most effective way to prevent SQL injection attacks. |

| **Noncompliant Code** |
| --- |
| std::string query = “SELECT \* FROM users WHERE username = '" + username + "' AND password = '" + password + "'"; |
| This code constructs a SQL query by using concatenating user input directly into the string. This can lead SQL Injection vulnerabilities if a user’s input contains malicious SQL code. |

| **Compliant Code** |
| --- |
| std::string username = “Nguyen”;  std::string password = “Hai2023”;  std::string query = “SELECT \* FROM users WHERE username = '" + username + "' AND password = '" + password + "'"; |
| This code is compliant with the SQL Injection Coding Standard because it uses prepared statements or parameterized queries to prevent SQL Injection attacks. |

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

| **Principles(s):** 4. Keep It Simple |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Low | High | All levels |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.0 | Keep It Simple | Indicate areas of code that are overly complex or difficult to maintain. It also provides a dashboard with visualizations and detailed reports that can help developers prioritize areas for improvement. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-055-CPP | Do not access free memory. Ref.: MEM30-CPP.  One common problem that can happen when working properly with dynamically allocated memory is accessing memory after it has been freed. This can lead to undefined behavior and can cause crashes, data corruption, and many other security problems. There are some effective ways to prevent this issue by following the coding standard, including avoiding manual memory management, initializing null pointer, and avoiding pointer arithmetic. |

| **Noncompliant Code** |
| --- |
| int\* ptr = new int;  delete ptr;  int x = \*ptr; //accessing freed memory |
| This code allocates an integer on the heap using the ‘new’ operator and stores the address of allocated memory in ‘new’ operator and stores the address of the allocated memory in ‘ptr’. The programmer then deletes the memory pointed to by ‘ptr’, and tries to access the freed memory by dereferencing ‘ptr’ and assigning its value to ‘x’. This is undefined behavior and may lead to a crash. |

| **Compliant Code** |
| --- |
| int\* ptr = new int;  int x = \*ptr; //reading the value of allocated memory  delete ptr; //freeing the memory. |
| By accessing the allocated memory before deleting it, the programmer avoids accessing freed memory and ensures that the behavior of the program is well-defined. |

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

| **Principles(s):** 5. Default Deny |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Low | Low | High | 2 (out of 5) |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Security Content Automation Protocol (SCAP) | 1.3 | XCCDF (eXtensible Configuration Checklist Description Format) | SCAP is a suite of standards and specifications that can be used to automate vulnerability management, measurement, and policy compliance evaluation. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-056-CPP | Avoid information leakage when passing a structure across a trust boundary. Ref.: DCL39-C.  Information leakage can occur when a structure contains sensitive information such as passwords, keys, or personal data, and that information is not protected when the structure is passed across a trust boundary. Use encryption and authentication mechanisms to protect sensitive information when transmitting a structure over a network or writing it to a file. |

| **Noncompliant Code** |
| --- |
| #include <cstddef>  struct test {  int a;  char b;  int c;  };  /\* Safely copy bytes to user space \*/  extern int copy\_to\_user(void\* dest, void\* src, size\_t size);  void do\_stuff(void\* usr\_buf) {  test arg = {.a = 1, .b = 2, .c = 3};  copy\_to\_user(usr\_buf, &arg, sizeof(arg));  } |
| In this code, the padding bytes may contain sensitive information, so there is a risk of that information being leaked when the structure is copied to user space using the ‘copy\_to\_user’ function. |

| **Compliant Code** |
| --- |
| #include <cassert>  #include <cstddef>  struct test {  int a;  char b;  int c;  };  /\* Safely copy bytes to user space \*/  extern int copy\_to\_user(void\* dest, void\* src, size\_t size);  void do\_stuff(void\* usr\_buf) {  test arg = {.a = 1, .b = 2, .c = 3};    // Check for padding bytes that may contain sensitive information  assert(sizeof(arg) == sizeof(arg.a) + sizeof(arg.b) + sizeof(arg.c));    copy\_to\_user(usr\_buf, &arg, sizeof(arg));  } |
| By including the assertion code in this code, I can ensure that any padding bytes in the ‘arg’ structure are detected and handled properly before the structure is copied to user space. |

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

| **Principles(s):** 6. Adhere to the Principle of Least Privilege. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | High | High | 2 (out of 5) |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify Real-Time Management | 22.2.0 | Privilege management | Detects privilege escalation vulnerabilities and provides |
| SonarQube | 10.0 | S2068 | Detects code that uses elevated privileges and provides guidance on how to implement the principle of least privilege. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-057-CPP | Handle all exceptions thrown before main() begin executing.  Ref.: ERR58-CPP  This is a good practice in C++ for several reasons to ensure proper initialization, prevent unexpected termination, and improve code reliability. |

| **Noncompliant Code** |
| --- |
| **struct** S {    S() noexcept(**false**);  };    **static** S globalS; |
| In this code, the constructor for S may throw an exception that is not caught when globalS is constructed during the program startup. |

| **Compliant Code** |
| --- |
| **struct** S {    S() noexcept(**false**);  };    S &globalS() {  **try** {  **static** S s;  **return** s;    } **catch** (...) {      // Handle error, perhaps by logging it and gracefully terminating the application.    }    // Unreachable.  } |
| This code uses try and catch block to catch any exception thrown during the initialization of the static value. By handling exceptions explicitly in this way, the code is more robust and less likely to encounter unexpected behavior. |

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

| **Principles(s):** 7. Sanitize Data Sent to Other Systems. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.0 | S2083 | Detects potential vulnerabilities arising from injection of untrusted data into XML or JSON response. |
| Fortify | 22.2.0 | XSS | Check for cross-site scripting vulnerabilities by detecting instances where input is not sanitized. |
| Checkmarx | 9.5.0 | Taint | Detects cross-site scripting, command injection, and SQL injection vulnerabilities due to data that is not sanitized. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Initialization | STD-058-CPP | Avoiding cycles during initialization of static objects.  Ref.: DCL56-CPP  One potential that can arise during initialization is the creation of cycles or circular dependencies between static objects. To avoid cycles during initialization of static objects, it’s significant to carefully manage object dependencies and ensure that they are initialized in a well-defined order. |

| **Noncompliant Code** |
| --- |
| #include <iostream>  class A {  public:  A() { std::cout << "A::A()" << std::endl; }  };  class B {  public:  static A a;  B() { std::cout << "B::B()" << std::endl; }  };  A B::a;  B b;  int main() {  std::cout << "main()" << std::endl;  return 0;  } |
| In this code, the initialization order of static objects is unspecified, it’s possible that the constructor for ‘a’ will be called before ‘b’, leading to an undefined order of initialization and potentially causing issue due to cyclic dependencies. |

| **Compliant Code** |
| --- |
| #include <iostream>  class A {  public:  A() { std::cout << "A::A()" << std::endl; }  };  class B {  public:  static A& getA() {  static A a;  return a;  }  B() { std::cout << "B::B()" << std::endl; }  };  B b;  int main() {  std::cout << "main()" << std::endl;  B::getA();  return 0;  } |
| In this code, the ‘B’ class has been modified to use a static member function ‘getA’ that initializes the ‘A’ object on its first call and returns a reference to it on subsequent calls. |

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

| **Principles(s):** 8. Practice Defense in Depth. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP ZAP | 5.0 | Multiple | Provides multiple built-in checks and plugins that can help implement defense-in-depth techniques, such as performing automated scans, intercepting, and modifying requests, and identifying vulnerabilities in web application. By using ZAP, developers can add an extra layer of security to their application and practice DiD. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Concurrency | STD-059-CPP | Do not destroy a mutex while it is locked.  Ref.: CON50-CPP.  A mutex is used to protect shared resources from being accessed by multiple threads at the same time. If a mutex is destroyed while it is locked, it can lead to a situation where other threads that were blocked on that mutex will be left waiting infinitely. Therefore, it is important to follow the “Do not destroy a mutex while it is locked” rule to ensure correct and reliable behavior of multi-threaded programs. |

| **Noncompliant Code** |
| --- |
| #include <mutex>  #include <thread>    **const** **size\_t** maxThreads = 10;    **void** do\_work(**size\_t** i, std::mutex \*pm) {    std::lock\_guard<std::mutex> lk(\*pm);      // Access data protected by the lock.  }    **void** start\_threads() {    std::**thread** threads[maxThreads];    std::mutex m;    **for** (**size\_t** i = 0; i < maxThreads; ++i) {      threads[i] = std::**thread**(do\_work, i, &m);    }  } |
| This code contains a race condition, allowing the mutex to be destroyed while it is still owned, because start\_threads() may invoke the mutex's destructor before all of the threads have exited. |

| **Compliant Code** |
| --- |
| #include <mutex>  #include <thread>    **const** **size\_t** maxThreads = 10;    **void** do\_work(**size\_t** i, std::mutex \*pm) {    std::lock\_guard<std::mutex> lk(\*pm);      // Access data protected by the lock.  }    std::mutex m;    **void** start\_threads() {    std::**thread** threads[maxThreads];    **for** (**size\_t** i = 0; i < maxThreads; ++i) {      threads[i] = std::**thread**(do\_work, i, &m);    }  } |
| This code eliminates the race condition by extending the lifetime of the mutex. |

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

| **Principles(s):** 9. Use Effective Quality Assurance Techniques |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Medium | High | Medium | process |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Checkmarx | 9.5.0 | CWE-772 | Detects when resources are not properly released after usage in C and C++. Leaving resources like memory and file handles open can lead to resource leaks, service downtime, and security vulnerabilities. |
| Fortify | 22.2.0 | Resource cleanup | Detects when a resource is not properly closed after being used. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Integer | STD-060-CPP | Do not cast to an out-of-range enumeration value.  Ref.: INT50-CPP.  When casting an integer to an enumeration type, the value must be within the range of valid enumeration constants, otherwise, it leads to undefined behavior. |

| **Noncompliant Code** |
| --- |
| enum Color { RED, GREEN, BLUE };  void foo(int value)  {  Color c = static\_cast<Color>(value); // noncompliant  // ...  } |
| In this code, if the ‘value’ is outside the range of valid enumeration constants, the resulting behavior is undefined. |

| **Compliant Code** |
| --- |
| enum Color { RED, GREEN, BLUE };  void foo(int value)  {  if (value >= 0 && value <= 2) {  Color c = static\_cast<Color>(value); // compliant  // ...  }  } |
| This code uses ‘if’ statement to check that the value being cast to ‘Color’ is within the range valid enumeration constants. If the condition is met, a static\_cast is used to convert the value to the ‘Color’ enumeration type. This ensures the resulting behavior is well-defined and compliant with the integer coding standard. |

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

| **Principles(s):** 10. Adopt a Secure Coding Standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.0 | S2070 | Checks if the code follows secure coding standards like OWASP, CERT, and CWE. |
| PMD | 7.0.0-rc1 | Category/JVM/  SecurityCodeMaturityRule | Checks for the usage of deprecated or risky functions and the observance to security coding standards. |
| Checkmarx | 9.5.0 | Query language | Provides a query language to check if the code follows secure coding standards. |

### 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.

DevSecOps pipeline for Green Pace would involve focusing on integrating security and compliance into every step of the software development lifecycle, using a combination of automated tools and manual processes to ensure that security is a consistent priority throughout the organization.

### 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 |
| STD-030-CPP Data Type | High | High | Moderate-High | High | Implementation |
| STD-033-CPP Data Value | Medium | High | Low | Medium | Implementation |
| STD-050-CPP String Correctness | High | Medium | High | High | Design |
| STD-054-CPP SQL Injection | High | High | Low | High | All levels |
| STD-055-CPP Memory Protection | High | Low | Low | High | 2 |
| STD-056-CPP Assertions | High | Medium | High | High | 2 |
| STD-057-CPP Exceptions | High | High | High | High | 2 |
| STD-058-CPP Initialization | High | High | High | High | 1 |
| STD-059-CPP Concurrency | Medium | Medium | High | Medium | Process |
| STD-060-CPP Integer | Medium | High | Low | High | Implementation |

### Create Policies for Encryption and Triple A

Include all three types of encryptions (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 | It refers to the practice of encrypting data while it is stored on disk or other physical storage media. This is done to protect the data in case the storage media is stolen, lost, or accessed by unauthorized users. The encryption at rest policy requires sensitive data to be encrypted using secure encryption algorithm and the encryption keys be properly managed and protected. |
| Encryption in flight | It refers to the process of encrypting data as it is transmitted between two systems. It is also referred to as transport-level encryption or secure socket layer (SSL) encryption. During encryption in flight, data is encrypted using an encryption algorithm and a unique key and sent over the network where it can only be decrypted by the recipient who has the key to decipher the data. This encryption should be used whenever sensitive information is transmitted over an untrusted network like email communications, and file transfers over the internet. |
| Encryption in use | It refers to the protection of sensitive data when it is being processed or used by an application. With the encryption in use, data is temporarily stored in memory during being transited between two different systems. The policy of encryption in use applies when an application is handling sensitive data that needs to be used during its lifecycle. This policy is important in scenarios when sensitive data is being processed in a shared environment, such as cloud computing, where multiple users may have access to the same computing resources. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identity of a user, device, or system. Authentication is used to ensure that only authorized entities can access sensitive data or systems. The policy of authentication applies whenever a user, device or system tries to access a resource or system. It is very important when dealing with confidential information, such as financial transactions, medical records, or government systems. |
| Authorization | Authorization is the process of verifying whether an authenticated user has the necessary permissions to access a particular resource or perform a specific action. Authorization involves the use of access control mechanisms to ensure that only authorized users are granted access to specific resources. The authorization is essential to maintain the security and confidentiality of sensitive information and prevent unauthorized access or data breaches. This policy applies whenever there is a need to control access to sensitive data or resources, such as financial data, customer information, and other types of confidential information. |
| Accounting | Accounting is a security mechanism that involves the tracking and logging of user activities and events within an information system, such as monitoring and recording of login attempts, access to sensitive data, changes, and other critical activities. Accounting is used to provide an audit trail of system activities to detect and investigate security incidents, ensure compliance with policies and regulations. The policy for accounting applies whenever there is a need to monitor and record user activities within an information system, such as handling sensitive data, key infrastructure, or those that are subject to regulatory requirements. |

**\***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 best practice.

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Principle** | **Justification** |
| 1. Data Type | Validate Data Input | Input validation is an important security practice to ensure the data type of the input is correct. Validating input data types can prevent attackers from exploiting vulnerabilities caused by incorrect data type processing. |
| 2. Data Value | Heed Compiler Warnings | Heeding compiler warnings is essential in ensuring the integrity of the code, and it can help identify and prevent various security vulnerabilities. Data value checks can be used to prevent buffer overflows, which can be identified by a warning from the compiler. |
| 3. String Correctness | Architect and Design for Security Policies | Implementing policies such as password complexity requirements, input validation, and output encoding can help ensure that string data is handled securely. |
| 4. SQL Injection | Keep It Simple | Simple code is easier to review and analyze for potential security vulnerabilities and can be more easily audited for compliance with secure coding standards. By keeping code simple and avoiding unnecessary complexity to prevent SQL injection vulnerabilities. |
| 5. Memory Protection | Default Deny | By denying access to memory until it is explicitly granted, the risk of an attacker exploiting a memory related vulnerability can be reduced. |
| 6. Assertions | Adhere to the Principle of Least Privilege | Adhering to the principle of least privilege can help ensure that assertions do not unintentionally expose sensitive information or functionally. |
| 7. Exceptions | Sanitize Data Sent to Other Systems | Exceptions can provide valuable information to attackers if not handled properly, making it important to sanitize any data sent in exceptions. |
| 8. Initialization | Practice Defense in Depth | Practicing defense in depth provides additional layers of protection in the event of a vulnerability, helping to mitigate the risk of exploitation. Applying proper initialization of variables and resources is significant for ensuring code executes as intended and preventing vulnerabilities such as uninitialized memory access. |
| 9. Concurrency | Use Effective Quality Assurance Techniques | Techniques like code reviews, testing, and analysis can help ensure that code is free of concurrency-related vulnerabilities. |
| 10. Integer | Adopt a Secure Coding Standard | Adopting a secure coding standard that includes best practices for integer handling to ensure that code is written in a way that prevents these types of vulnerabilities. |

**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 |