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



# **Green Pace Secure Development Policy**

## 

## 

## 

## Contents

Overview 2

Purpose 2

Scope 2

Module Three Milestone 2

Ten Core Security Principles 2

C/C++ Ten Coding Standards 3

Coding Standard 1 4

Coding Standard 2 5

Coding Standard 3 6

Coding Standard 4 7

Coding Standard 5 8

Coding Standard 6 9

Coding Standard 7 10

Coding Standard 8 11

Coding Standard 9 13

Coding Standard 10 14

Defense-in-Depth Illustration 15

Project One 15

1. Revise the C/C++ Standards 15

2. Risk Assessment 15

3. Automated Detection 15

4. Automation 15

5. Summary of Risk Assessments 16

6. Create Policies for Encryption and Triple A 16

7. Map the Principles 17

Audit Controls and Management 18

Enforcement 18

Exceptions Process 18

Distribution 19

Policy Change Control 19

Policy Version History 19

Appendix A Lookups 19

Approved C/C++ Language Acronyms 19

## **Overview**

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

## **Purpose**

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

## **Scope**

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

## 

## 

## 

## 

## 

## 

## **Module Three Milestone**

### **Ten Core Security Principles**

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validating input data is essential to ensure that the incoming data meets the expected criteria and does not contain any malicious or unexpected content. Potential vulnerabilities such as buffer overflows or injection attacks can be prevented by validating input data. |
| 2. Heed Compiler Warnings | Compiler warnings often indicate potential issues in the code that could lead to security vulnerabilities. Ignoring these warnings can result in exploitable weaknesses. Attention to and addressing all compiler warnings to enhance code security is essential. |
| 3. Architect and Design for Security Policies | Security policies should be incorporated into software development's architectural and design phases. By considering security requirements from the beginning, potential vulnerabilities can be mitigated, and the overall security posture of the system can be improved. |
| 4. Keep It Simple | Simplicity in design and implementation can contribute to better security. Complex systems are more prone to vulnerabilities and are more challenging to analyze and secure. The attack surface can be reduced by keeping things simple, and the system can be more easily understood and validated. |
| 5. Default Deny | The default deny principle means that access should be denied by default unless explicitly granted. This approach helps to minimize the risk of unauthorized access and ensures that only necessary privileges are granted to users or processes. |
| 6. Adhere to the Principle of Least Privilege | The principle of least privilege dictates that each user or process should only have the minimum privileges necessary to perform their required tasks. This principle reduces the potential impact of a compromised user or process and limits the scope of potential attacks. |
| 7. Sanitize Data Sent to Other Systems | When sending data to other systems, it is crucial to sanitize it to prevent injection attacks or other data manipulation. Sanitization involves validating, cleaning, and encoding data to ensure that it does not contain malicious content that could compromise the integrity or security of the receiving system. |
| 8. Practice Defense in Depth | Defense in depth refers to implementing multiple layers of security controls to protect against various attacks. Employing multiple defensive measures such as firewalls, intrusion detection systems, and encryption reduces the likelihood of a successful attack, as a single control failure does not compromise the entire system. |
| 9. Use Effective Quality Assurance Techniques | Quality assurance techniques such as code reviews, testing, and static analysis are crucial in identifying and resolving security vulnerabilities. By employing effective quality assurance practices, potential weaknesses can be identified and addressed early in the development process, enhancing the overall security of the software. |
| 10. Adopt a Secure Coding Standard | Following a secure coding standard, such as the ***SEI CERT C++ Coding Standard***, provides guidelines and best practices for developing secure software. By adhering to a standard, developers can avoid common security pitfalls and ensure that their code meets recognized security requirements. |

### 

### **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** | DT-001-CPP | Data Type Standard   * Data types must be chosen carefully based on the range of values they must represent, and signed and unsigned types must not be mixed in the same expression. It also recommends using the <cstdint> header to ensure portable and standardized integer types. |

| **Noncompliant Code** |
| --- |
| This example defines an int variable *x* and an unsigned int variable *y*. We then compare *x* and *y* in an if statement, which violates the standard because it mixes signed and unsigned types in the same expression. |
| #include <iostream>  int main() {  int x = 10;  unsigned int y = 20;  if (x < y) {  std::cout << "x is less than y" << std::endl;  }  return 0;  } |

| **Compliant Code** |
| --- |
| In this example, we use the <cstdint> header to define the integer types int32\_t and uint32\_t, which represent signed and unsigned 32-bit integers, respectively. We use these types appropriately to ensure that the variables a and b are represented correctly.  The code block also demonstrates the compliant use of integer types in expressions. The sum variable is defined as the sum of two signed integers compliant with the standard. |
| #include <cstdint>  int32\_t a = 10; // Use int32\_t to represent a 32-bit signed integer  uint32\_t b = 50; // Use uint32\_t to represent a 32-bit unsigned integer  int32\_t sum = a + static\_cast<int32\_t>(b); // OK: Sum two signed integers |

**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** | DV-002-CPP | Data Value Standard   * Requires that data values be chosen carefully based on their intended purpose and range of valid values. |

| **Noncompliant Code** |
| --- |
| In this example, the variable value is declared but not initialized. This violates the standard, as using an uninitialized variable can lead to undefined behavior and potential security vulnerabilities. To make this code compliant, we should initialize the value to a valid value before using it in the if statement. |
| #include <iostream>  int main() {  int value; // Non-compliant: Variable is not initialized  if (value > 0) {  std::cout << "Value is positive" << std::endl;  }  return 0;  } |

| **Compliant Code** |
| --- |
| In this example, we define a constant MAX\_VALUE to represent the maximum allowed value. Using a constant ensures that the value is not modified and reflects the intended limit. The variable value is initialized with a valid value of 5. The code then checks if the value exceeds MAX\_VALUE and prints a message if it does. |
| #include <iostream>  int main() {  const int MAX\_VALUE = 10; // Compliant: Define a constant for the maximum value  int value = 5; // Compliant: Initialize the variable with a valid value  if (value > MAX\_VALUE) {  std::cout << "Value exceeds maximum" << 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 3**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STC-003-CPP | String Correctness Coding Standard   * Emphasizes the importance of properly null-terminating strings, avoiding buffer overflows, and correctly handling string length and size. * Prevents security vulnerabilities and undefined behavior related to string manipulation. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the char pointer str is initialized to the address of a string literal. Attempting to modify the string literal is undefined behavior: |
| **char** \*str = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| As an array initializer, a string literal specifies the initial values of characters in an array as well as the size of the array. This code creates a copy of the string literal in the space allocated to the character array str. The string stored in str can be modified safely. |
| **Char** srt[] = “string literal”;  str[0] = ‘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 4**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | SQL-004-CPP | The SQL Injection Standard   * Prevents SQL injection vulnerabilities in C++ code. * Emphasizes the importance of using prepared statements or parameterized queries when interacting with databases to avoid the risk of maliciously crafted input being executed as part of a SQL query. |

| **Noncompliant Code** |
| --- |
| In this example, a SQL query is constructed by concatenating user-provided input directly into the query string. This approach is vulnerable to SQL injection, as an attacker can manipulate the input to execute arbitrary SQL commands. This violates the ERR58-CPP standard, exposing the application to security risks. |
| #include <iostream>  #include <mysql/mysql.h>  int main() {  MYSQL mysql;  mysql\_init(&mysql);  const char\* query = "SELECT \* FROM users WHERE username='" + getUsername() + "' AND password='" + getPassword() + "'";  mysql\_real\_query(&mysql, query, std::strlen(query)); // Non-compliant: Vulnerable to SQL injection  MYSQL\_RES\* result = mysql\_store\_result(&mysql);  // Process the result...  mysql\_close(&mysql);  return 0;  } |

| **Compliant Code** |
| --- |
| This example uses a prepared statement instead of directly concatenating user input into the query string. The query uses placeholders (?) for the username and password values. The user-provided input is then securely bound to the prepared statement using mysql\_stmt\_bind\_param(), ensuring that it is treated as data rather than executable code. This approach prevents SQL injection vulnerabilities. |
| #include <iostream>  #include <mysql/mysql.h>  int main() {  MYSQL mysql;  mysql\_init(&mysql);  const char\* query = "SELECT \* FROM users WHERE username=? AND password=?";  MYSQL\_STMT\* stmt = mysql\_stmt\_init(&mysql);  mysql\_stmt\_prepare(stmt, query, std::strlen(query)); // Compliant: Prepared statement  const char\* username = getUsername();  const char\* password = getPassword();  MYSQL\_BIND bindParams[2];  bindParams[0].buffer\_type = MYSQL\_TYPE\_STRING;  bindParams[0].buffer = (void\*)username;  bindParams[0].buffer\_length = std::strlen(username);  bindParams[0].is\_null = 0;  bindParams[1].buffer\_type = MYSQL\_TYPE\_STRING;  bindParams[1].buffer = (void\*)password;  bindParams[1].buffer\_length = std::strlen(password);  bindParams[1].is\_null = 0;  mysql\_stmt\_bind\_param(stmt, bindParams); // Compliant: Parameterized query  mysql\_stmt\_execute(stmt);  MYSQL\_RES\* result = mysql\_stmt\_result\_metadata(stmt);  // Process the result...  mysql\_stmt\_close(stmt);  mysql\_close(&mysql);  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 5**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM-005-CPP | Memory Protection Coding Standard   * Emphasizes the importance of adequately allocating and deallocating memory, avoiding buffer overflows and underflows, and ensuring that memory is not accessed after it has been freed. |

| **Noncompliant Code** |
| --- |
| In this example, an array of int is dynamically allocated with space for 10 integers using new. However, the loop that initializes the array uses <= instead of < for the loop condition, resulting in a buffer overflow when the loop tries to access ptr[10]. |
| #include <iostream>  int main() {  int\* ptr = new int[10];  for (int i = 0; i <= 10; i++) { // Non-compliant: Buffer overflow  ptr[i] = i;  }  std::cout << ptr[5] << std::endl;  delete[] ptr;  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant example, an array of int is dynamically allocated using a std::unique\_ptr, which ensures that the memory is automatically deallocated when it goes out of scope. The loop that initializes the array uses < for the loop condition, ensuring that it does not exceed the bounds of the array. |
| #include <iostream>  #include <memory>  int main() {  std::unique\_ptr<int[]> ptr(new int[10]); // Compliant: Unique pointer used  for (int i = 0; i < 10; i++) {  ptr[i] = i;  }  std::cout << ptr[5] << 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 6**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | AST-006-CPP | * Emphasizes the importance of using assertions to validate assumptions and detect programming errors during development and testing. * Aims to improve code reliability and maintainability by catching errors early and providing meaningful diagnostics. |

| **Noncompliant Code** |
| --- |
| An assertion is used to check if the divisor is not zero before performing division. However, the assertion does not provide a diagnostic message, making it difficult to understand the cause of the assertion failure. |
| #include <iostream>  #include <cassert>  int divide(int dividend, int divisor) {  assert(divisor != 0); // Non-compliant: Assertion without diagnostic message  return dividend / divisor;  }  int main() {  int result = divide(10, 0);  std::cout << "Result: " << result << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant example, the assertion is modified to include a diagnostic message using a string literal. This provides a meaningful explanation when the assertion fails, helping developers identify and fix the issue. |
| #include <iostream>  #include <cassert>  int divide(int dividend, int divisor) {  assert(divisor != 0 && "Divisor must be nonzero"); // Compliant: Assertion with diagnostic message  return dividend / divisor;  }  int main() {  int result = divide(10, 0);  std::cout << "Result: " << result << 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 7**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | EXCP-007-CPP | * When an exception is thrown, the value of the object in the throw expression is used to initialize an anonymous temporary object called the *exception object*. * The type of this exception object is used to transfer control to the nearest catch handler, which contains an exception declaration with a matching type. |

| **Noncompliant Code** |
| --- |
| In this example, an exception is thrown using a string literal when the divisor is zero. However, throwing string literals is not recommended, as they can be difficult to manage and may not provide enough information about the nature of the exception. |
| #include <iostream>  int divide(int dividend, int divisor) {  if (divisor == 0) {  throw "Divide by zero"; // Non-compliant: Throwing string literal  }  return dividend / divisor;  }  int main() {  try {  int result = divide(10, 0);  std::cout << "Result: " << result << std::endl;  }  catch (const char\* message) {  std::cerr << "Exception caught: " << message << std::endl;  }  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant example, an exception is thrown using std::runtime\_error, which provides a more structured approach to exception handling and allows for more informative diagnostic messages. The what() method is used to retrieve the diagnostic message from the exception object. |
| #include <iostream>  #include <stdexcept>  int divide(int dividend, int divisor) {  if (divisor == 0) {  throw std::runtime\_error("Divide by zero"); // Compliant: Throwing std::runtime\_error  }  return dividend / divisor;  }  int main() {  try {  int result = divide(10, 0);  std::cout << "Result: " << result << std::endl;  }  catch (const std::exception& e) {  std::cerr << "Exception caught: " << 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** |
| --- | --- | --- |
| Don’t Repeat Yourself(DRY) | DRY-008-CPP | * Standard focuses on avoiding code duplication. * It emphasizes the importance of identifying and eliminating redundant code to improve code readability and reduce the risk of errors. * It Aims to help developers write more efficient, maintainable, and reliable code. |

| **Noncompliant Code** |
| --- |
| In this example, the function printMessageTwice calls printMessage twice, resulting in redundant code. This violates the DRY04-CPP standard, as it makes the code harder to read, maintain, and increases the risk of errors. |
| #include <iostream>  void printMessage() {  std::cout << "Hello, world!" << std::endl;  }  void printMessageTwice() {  printMessage();  printMessage(); // Non-compliant: Redundant code  }  int main() {  printMessageTwice();  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant example, the function printMessageNTimes is introduced to avoid code duplication. This function takes an integer parameter *n* and calls printMessage *n* times using a loop. This follows the DRY standard by eliminating redundant code and making the code more readable and maintainable. |
| #include <iostream>  void printMessage() {  std::cout << "Hello, world!" << std::endl;  }  void printMessageNTimes(int n) {  for (int i = 0; i < n; i++) {  printMessage();  }  }  int main() {  printMessageNTimes(2); // Compliant: DRY principle applied  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** |
| --- | --- | --- |
| Single Responsibility Principle | SRP-009-CPP | * The SRP states that a class should have only one reason to change, meaning it should only have one responsibility or concern. * This standard emphasizes the importance of designing focused and cohesive classes, making them easier to understand, maintain, and reuse. |

| **Noncompliant Code** |
| --- |
| In this example, the Rectangle class has two responsibilities: calculating the area of the rectangle and printing its dimensions. This violates the SRP standard, making the class less focused and cohesive, complicating its understanding, maintenance, and reuse. |
| #include <iostream>  class Rectangle {  public:  Rectangle(int width, int height) : width\_(width), height\_(height) {}  int getArea() const {  return width\_ \* height\_;  }  void print() const {  std::cout << "Width: " << width\_ << ", Height: " << height\_ << std::endl;  }  private:  int width\_;  int height\_;  };  int main() {  Rectangle rectangle(5, 10);  rectangle.print();  std::cout << "Area: " << rectangle.getArea() << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant example, the Rectangle class has only one responsibility: calculating the area of the rectangle. The printing functionality has been moved to a separate function printRectangle, which takes a const reference to a Rectangle object and prints its dimensions. This follows the SRP standard by making the Rectangle class focused and cohesive, improving its readability, maintainability, and reusability. |
| #include <iostream>  class Rectangle {  public:  Rectangle(int width, int height) : width\_(width), height\_(height) {}  int getArea() const {  return width\_ \* height\_;  }  private:  int width\_;  int height\_;  };  void printRectangle(const Rectangle& rectangle) {  std::cout << "Width: " << rectangle.width\_ << ", Height: " << rectangle.height\_ << std::endl;  }  int main() {  Rectangle rectangle(5, 10);  printRectangle(rectangle);  std::cout << "Area: " << rectangle.getArea() << 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** |
| --- | --- | --- |
| Proper Commenting | COM-010-CPP | * It emphasizes the importance of using comments to improve code readability and provide context and rationale for code decisions. * This standard aims to help developers write more understandable and maintainable code. |

| **Noncompliant Code** |
| --- |
| In this example, there are no comments to provide context or rationale for the code. |
| #include <iostream>  int main() {  int x = 5;  int y = 10;  int sum = x + y;  std::cout << "Sum: " << sum << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant example, comments have been added to provide context and rationale for the code. |
| #include <iostream>  int main() {  int x = 5; // Initialize variable x  int y = 10; // Initialize variable y  int sum = x + y; // Calculate the sum of x and y  std::cout << "Sum: " << sum << std::endl; // Print the sum  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 | 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***.***

a. Explain each type of encryption, how it is used, and why and when the policy applies.

b. 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 in rest | [Insert text.] |
| Encryption at 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 |