**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. Organizations can mitigate the risk of various security threats by ensuring that the incoming data adheres to the expected criteria. Potential vulnerabilities such as buffer overflows (when a program attempts to write more data into a buffer than it can handle) or injection attacks (malicious code or commands being injected into an application's input fields) can be prevented by validating input data. Organizations can strengthen their defenses by implementing robust input validation mechanisms and reducing the risk of exploitation or compromise. |
| 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. By paying attention to these warnings and taking appropriate action to resolve them, developers can reduce the risk of vulnerabilities and improve the overall security of their code. Paying attention to and addressing compiler warnings is essential to maintaining code security. By taking these warnings seriously and resolving potential issues, developers can reduce the risk of vulnerabilities and protect against potential exploitation by malicious actors. |
| 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 outset, organizations can proactively identify potential vulnerabilities and implement appropriate mitigations, improving overall security posture. Security considerations should be integrated into the system's design principles and decision-making process during the architectural phase. In the design phase, security requirements should be translated into specific security controls and mechanisms. By integrating security policies into the early stages of software development, organizations can minimize the risk of introducing security flaws later in the development lifecycle. This proactive approach helps establish a strong security foundation and reduces the likelihood of costly security incidents or breaches. |
| 4. Keep It Simple | Simplicity in design and implementation can significantly enhance security. Complex systems introduce more potential vulnerabilities and can be challenging to analyze and secure. Keeping things simple reduces the attack surface, making it easier to understand and validate the system's security. Complexity often leads to unintended interactions and dependencies, creating hidden security risks. In contrast, a more straightforward design promotes transparency, and maintainability and allows for a more focused security analysis. Embracing simplicity as a guiding principle can enhance the overall security stance of a system. |
| 5. Default Deny | The default deny principle means that access should be denied by default unless explicitly granted. Instead of granting broad access permissions to users or processes, this principle ensures that only the necessary privileges are explicitly provided. This reduces the risk of accidental or intentional misuse of resources and helps prevent unauthorized access to sensitive information. The default deny principle also aligns with the principle of defense in depth, where multiple security controls are implemented to protect assets. By adopting the default deny principle, denying access by default, and only granting necessary privileges, organizations minimize the risk of unauthorized access and reduce the potential for security breaches. |
| 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. By adhering to the principle of least privilege, unnecessary privileges are avoided, which helps to minimize the potential damage that can be caused by an attacker who gains unauthorized access. Each user or process is granted only the specific permissions necessary to carry out their designated functions, preventing them from accessing sensitive resources or performing actions beyond their intended scope. By reducing unnecessary privileges, organizations can minimize the impact of compromised accounts and limit the potential for unauthorized access and malicious activities. |
| 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. Validation is the first step in the sanitization process. It involves checking the data against expected criteria or patterns to ensure that it meets the required format and does not contain any unexpected or malicious content. Cleaning is the next step, where any unnecessary or potentially dangerous characters or elements are removed from the data. Encoding is the final step, where the data is transformed into a safe and standardized format for transmission. By following these steps and implementing robust sanitization practices, organizations can prevent injection attacks, data manipulation, and other forms of data manipulation that could compromise the receiving system's security. |
| 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 the Defense in Depth approach recognizes that no single security control can provide complete protection against all potential threats. Therefore, multiple layers of security controls are implemented to provide a more comprehensive and resilient security posture. Even if one layer of defense is breached, other layers can still provide protection and prevent the attacker from reaching their ultimate goal. By employing a range of defensive measures, organizations can reduce the likelihood of a successful attack and create comprehensive and resilient security. |
| 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. Code reviews are an essential quality assurance technique that involves reviewing code to identify errors, inconsistencies, and potential security vulnerabilities. Testing is another critical quality assurance technique that verifies that the software functions as intended and does not contain any errors or vulnerabilities. Static analysis is a technique that involves analyzing the source code of the software to identify potential security vulnerabilities. Organizations can significantly enhance the overall security of their software by employing effective quality assurance techniques such as code reviews, testing, and static analysis. |
| 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** | STD-001-CPP | Data Type   * 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 |

| **Principles(s):** Data Integrity  This standard aligns with the principle of data integrity by emphasizing the careful selection of data types to represent the range of values they need to hold accurately. Mixing signed and unsigned types in the same expression can lead to unexpected behavior and potential loss of data integrity. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| LOW | MEDIUM | LOW | MEDIUM | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.0 | Data Type | CppCheck is a static analysis tool for C/C++ code that can help identify issues related to data type usage. |
| Clang-Tidy | 12.0 | Data Type | Clang-Tidy is a static analysis tool for C/C++ code that can provide warnings and suggestions for data types. |

#### **Coding Standard 2**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Data Value   * 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;  } |

| **Principles(s):** Data Validation  This standard aligns with the principle of data validation by emphasizing the careful selection of data values based on their intended purpose and range of valid values. It promotes the practice of validating data to ensure it meets the expected criteria, reducing the risk of data integrity issues or incorrect behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| MEDIUM | MEDIUM | LOW | MEDIUM | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.0 | Data Value | SonarQube is a static analysis tool that can help identify issues related to data value selection and validation. |
| ESLint | 8.0 | Data Value | ESLint is a popular static analysis tool for JavaScript code that can provide warnings and suggestions for data values. |

#### 

#### **Coding Standard 3**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | String Correctness   * 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’; |

| **Principles(s):** Input Validation and Sanitization  This standard aligns with the principle of input validation and sanitization by emphasizing the proper handling of strings to prevent security vulnerabilities and undefined behavior. It promotes practices such as null-terminating strings, avoiding buffer overflows, and correctly handling string length and size, which are crucial for ensuring the integrity and security of string manipulation operations. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | HIGH | MEDIUM | HIGH | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.0 | StringSafety | PVS-Studio is a static analysis tool that can help identify issues related to string safety, including null-termination errors. |
| Coverity | 8.0 | BufferOverflow | Coverity is a static analysis tool that can detect buffer overflow vulnerabilities, helping to ensure string correctness. |

#### 

#### 

#### 

#### 

#### **Coding Standard 4**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | SQL Injection   * 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;  } |

| **Principles(s):** Preventing SQL injection vulnerabilities in C++ code.  It emphasizes the importance of using prepared statements or parameterized queries to mitigate the risk of executing maliciously crafted input as part of a SQL query. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | HIGH | MODERATE | HIGH | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify | 20.1 | SQLInjection | Fortify is a static analysis tool that can detect and help prevent SQL injection vulnerabilities by analyzing C++ code for unsafe database queries. |
| Checkmarx | 9.0 | SQLInjection | Checkmarx is a static analysis tool that can identify potential SQL injection vulnerabilities in C++ code, providing guidance for secure coding practices. |

#### 

#### **Coding Standard 5**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Memory Protection   * 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 array's bounds. |
| #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):** Ensuring Memory Protection  This involves adequately allocating and deallocating memory, avoiding buffer overflows and underflows, and preventing access to freed memory. This principle emphasizes the importance of proper memory management to prevent memory-related vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | HIGH | HIGH | HIGH | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AddressSanitizer | 12.0 | MemoryProtection | AddressSanitizer is a compiler-based tool that can detect memory-related vulnerabilities, including buffer overflows and use-after-free errors. |
| Valgrind | 3.16 | Memcheck | Valgrind is a dynamic analysis tool that can detect memory-related issues, including buffer overflows, memory leaks, and use-after-free errors. |

#### 

#### 

#### **Coding Standard 6**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions   * 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:** Assertions   * This principle emphasizes the importance of using assertions to validate assumptions and detect programming errors during development and testing. By incorporating assertions into the code, developers can improve code reliability and maintainability by catching errors early and providing meaningful diagnostics. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| MEDIUM | MEDIUM | LOW | MEDIUM | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.6 | Assertion | Cppcheck is a static analysis tool that can detect potential issues in C++ code, including the presence or absence of assertions. |
| Clang-Tidy | 12.0 | Assertion | Clang-Tidy is a static analysis tool that can analyze C++ code and suggest improvements, including assertions usage. |

#### 

#### **Coding Standard 7**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Exceptions   * 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 retrieves 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:** Exceptions   * This principle emphasizes the proper handling of exceptions in C++ code. It highlights the use of exception objects to transfer control to the nearest catch handler when an exception is thrown. By following this principle, developers can ensure that exceptions are handled appropriately, improving code reliability and maintainability. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| MEDIUM | MODERATE | MODERATE | MEDIUM | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | Latest | Exception handling | PVS-Studio is a commercial static code analysis tool. It includes a checker specifically designed to detect issues related to exception handling. It can analyze the codebase and identify potential problems such as missing catch blocks, improper exception handling, and incorrect use of exception types. |
| FindBugs | Latest | Exception | FindBugs is an open-source static code analysis tool for Java. It includes a checker for exception-related issues. It can detect potential problems such as unused or uncaught exceptions, incorrect exception handling, and improper use of exception types. |
| ReSharper | Latest | Exception | ReSharper is a commercial extension for Visual Studio. It includes a checker for exception-related issues in C# code. It can detect potential problems such as unused or uncaught exceptions, incorrect exception handling, and improper use of exception types. |

#### 

#### **Coding Standard 8**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Don’t Repeat Yourself(DRY) | STD-008-CPP | DRY(Don’t Repeat Yourself)   * 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;  } |

| **Principles:** DRY (Don't Repeat Yourself)   * This principle 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. By following this principle, developers can write more efficient, maintainable, and reliable code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| LOW | MODERATE | LOW | MEDIUM | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | Latest | Duplicated Code | SonarQube is a widely used open-source tool for static code analysis. It includes a checker specifically designed to detect duplicated code. It can analyze the codebase and identify instances of duplicated code, providing recommendations for refactoring and eliminating redundancy. |
| PMD | Latest | Duplicated Code | PMD is an open-source static code analysis tool. It includes a checker for duplicate code detection. It can analyze the source code and identify duplicated code blocks, methods, or classes. |
| ReSharper | Latest | Duplicated Code | ReSharper is a commercial extension for Visual Studio. It includes a checker for duplicate code detection in C# code. It can analyze the codebase and identify instances of duplicated code. |

#### **Coding Standard 9**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Principle of Least Privilege | STD-009-CPP | Principle of Least Privilege:  Ensures that users and processes are granted only the minimum privileges necessary to perform their tasks, reducing the potential impact of security breaches. |

| **Noncompliant Code** |
| --- |
| This example does not adhere to the principle of least privilege. The `performSensitiveOperation` function allows all users, regardless of their privileges, to perform the sensitive operation. This increases the risk of unauthorized access and potential security breaches. |
| // Non-compliant: Lack of least privilege principle  void performSensitiveOperation(const std::string& username) {  // Code to perform sensitive operation  // Allow all users to perform the sensitive operation, regardless of their privileges  // ...  }  void processRequest(const std::string& username) {  // Code to process the request  performSensitiveOperation(username);  // Code to continue processing the request  // ...  } |

| **Compliant Code** |
| --- |
| The principle of least privilege is followed by implementing appropriate access controls. The `performSensitiveOperation` function only allows the admin user to perform the sensitive operation, while other users are denied access. This ensures that only users with the necessary privileges can execute the sensitive operation. |
| // Compliant: Implementing least privilege principle  void performSensitiveOperation(const std::string& username) {  // Code to perform sensitive operation  if (username == "admin") {  // Only allow the admin user to perform the sensitive operation  // ...  } else {  // Handle unauthorized access  // ...  }  }  void processRequest(const std::string& username) {  // Code to process the request  performSensitiveOperation(username);  // Code to continue processing the request  // ...  } |

| **Principles(s):** Principle of Least Privilege   * This principle ensures that users and processes are granted only the minimum privileges necessary to perform their tasks, reducing the potential impact of security breaches. By following this principle, developers can limit access rights and permissions to prevent unauthorized actions and minimize the potential damage caused by security incidents. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | HIGH | HIGH | HIGH | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| No specific automation tools are required for this principle as it involves defining and enforcing access rights and permissions based on the principle of least privilege. However, there are tools and technologies that can assist in implementing and enforcing least privilege principles, such as:  - Identity and Access Management (IAM) systems: These systems provide centralized control over user access rights and permissions, allowing administrators to define and manage privileges based on roles and responsibilities.  - Privileged Access Management (PAM) solutions: PAM solutions help manage and secure privileged accounts, ensuring that only authorized users have access to critical systems and resources.  - Security Information and Event Management (SIEM) systems: SIEM systems can monitor and analyze user activity, providing insights into potential privilege misuse or unauthorized access attempts.  Using these tools and technologies can help automate the enforcement of the principle of least privilege, reducing the risk of security breaches and unauthorized access. | | | |
|
|
|

#### **Coding Standard 10**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure Authentication and Authorization | STD-010-CPP | Secure Authentication and Authorization:   * Implement strong authentication mechanisms (e.g., multi-factor authentication) and enforce appropriate authorization controls to protect against unauthorized access. |

| **Noncompliant Code** |
| --- |
| This example does not adhere to the principle of least privilege. The `performSensitiveOperation` function allows all users, regardless of their privileges, to perform the sensitive operation. This increases the risk of unauthorized access and potential security breaches. |
| // Non-compliant: Using weak authentication mechanism  bool authenticateUser(const std::string& username, const std::string& password) {  if (username == "admin" && password == "123456") {  return true; // Authentication successful  } else {  return false; // Authentication failed  }  }  // Non-compliant: Lack of appropriate authorization controls  bool hasPermission(const std::string& username, const std::string& resource) {  return true; // No proper check for authorization, granting permission to all users  }  void processSensitiveData(const std::string& username, const std::string& resource) {  if (authenticateUser(username, "password123")) {  if (hasPermission(username, resource)) {  // Code to process and access sensitive data  // ...  } else {  // Handle unauthorized access - this block will never be executed in the non-compliant example  // ...  }  } else {  // Handle authentication failure - this block will never be executed in the non-compliant example  // ...  }  } |

| **Compliant Code** |
| --- |
| Here, a secure authentication mechanism is used (`authenticateUser`) which can involve techniques like hashing and salting the password. Additionally, appropriate authorization controls (`hasPermission`) are implemented to check if the user has the necessary permissions to access a specific resource. The `processSensitiveData` function uses these secure authentication and authorization mechanisms to process and access sensitive data. |
| // Compliant: Using a secure authentication mechanism  bool authenticateUser(const std::string& username, const std::string& password) {  // Code for authenticating the user using a secure mechanism (e.g., hashing and salting the password)  // ...  return true; // Authentication successful  }  // Compliant: Implementing appropriate authorization controls  bool hasPermission(const std::string& username, const std::string& resource) {  // Code for checking if the user has appropriate permissions to access the specified resource  // ...  return true; // User has permission  }  void processSensitiveData(const std::string& username, const std::string& resource) {  if (authenticateUser(username, "password123")) {  if (hasPermission(username, resource)) {  // Code to process and access sensitive data  // ...  } else {  // Handle unauthorized access  // ...  }  } else {  // Handle authentication failure  // ...  }  } |

| **Principles(s):** Secure Authentication and Authorization   * This principle emphasizes the implementation of strong authentication mechanisms, such as multi-factor authentication, and the enforcement of appropriate authorization controls to protect against unauthorized access. By following this principle, developers can ensure that only authenticated and authorized users can access sensitive resources, reducing the risk of unauthorized access and data breaches. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | HIGH | HIGH | HIGH | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| No specific automation tools are required for these principles as they involve implementing and enforcing secure authentication mechanisms and appropriate authorization controls. However, there are tools and technologies that can assist in achieving these principles, such as:  - Identity and Access Management (IAM) systems: These systems provide centralized control over user authentication and authorization, allowing administrators to implement robust authentication mechanisms (e.g., multi-factor authentication) and define granular access controls based on roles and responsibilities.  - Security Assertion Markup Language (SAML) or OpenID Connect (OIDC): These protocols enable secure authentication and single sign-on (SSO) capabilities, allowing users to authenticate once and access multiple applications or services securely.  - Privileged Access Management (PAM) solutions: PAM solutions help manage and secure privileged accounts, ensuring that only authorized users can access critical systems and resources.  These tools and technologies can help automate the implementation of secure authentication mechanisms and appropriate authorization controls. | | | |
|
|
|

### **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 | MEDIUM | LOW | MEDIUM | 2 |
| STD-002-CPP | MEDIUM | HIGH | MEDIUM | HIGH | 3 |
| STD-003-CPP | HIGH | MEDIUM | HIGH | HIGH | 4 |
| STD-004-CPP | HIGH | HIGH | MOD to HIGH | HIGH | 5 |
| STD-005-CPP | HIGH | HIGH | HIGH | HIGH | 5 |
| STD-006-CPP | MEDIUM | LOW | LOW | MEDIUM | 3 |
| STD-007-CPP | MEDIUM | MEDIUM | MODERATE | MEDIUM | 2 |
| STD-008-CPP | LOW | MEDIUM | LOW | LOW | 2 |
| STD-009-CPP | HIGH | HIGH | MEDIUM | HIGH | 4 |
| STD-010-CPP | HIGH | HIGH | HIGH | HIGH | 5 |

### **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.

| **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | All sensitive data stored in databases, file systems, or any other storage medium must be encrypted using strong encryption algorithms. At-rest encryption ensures that data remains protected even when it is stored on physical or digital storage devices. It prevents unauthorized access to sensitive information if the storage media is lost, stolen, or improperly accessed. This policy should be applied whenever sensitive data is stored in databases, file systems, cloud storage, or any other storage medium. At-rest encryption adds an extra layer of security to sensitive data, mitigating the risk of unauthorized access in case of physical theft, unauthorized access to storage devices, or data breaches. |
| Encryption at flight | All sensitive data transmitted over networks must be encrypted using secure protocols such as TLS (Transport Layer Security) or SSL (Secure Sockets Layer). In-flight encryption ensures that data transmitted between systems or over networks is protected from unauthorized access. This policy should be applied whenever sensitive data is being transmitted over networks, such as when accessing web applications, sending emails, or transferring files. In-flight encryption safeguards sensitive information from being compromised while it is in transit, reducing the risk of data breaches and unauthorized access. |
| Encryption in use | Sensitive data being processed or used by applications or systems must be encrypted during runtime or while being accessed by authorized users. In-use encryption protects sensitive data while it is being processed or accessed by authorized users or applications. It ensures that even if an attacker gains access to the system, they cannot view the plaintext data. This policy should be applied whenever sensitive data is being processed or accessed by applications, databases, or users. In-use encryption provides an additional layer of protection to sensitive data, reducing the risk of unauthorized access or leakage during runtime or when accessed by authorized users. |

| **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | All users accessing the system must authenticate themselves using unique credentials, such as usernames and passwords, or through multi-factor authentication (MFA). Authentication verifies the identity of users before granting them access to the system. It ensures that only authorized individuals can access sensitive information or perform specific actions. This policy should be applied to all systems and applications that require user authentication, including login portals, remote access systems, and administrative interfaces. Authentication prevents unauthorized access and protects sensitive information from being accessed by malicious actors. |
| Authorization | Users must be granted appropriate access privileges based on their organizational roles and responsibilities. Access controls should be implemented to enforce least privilege and need-to-know principles. Authorization controls what actions users can perform and what resources they can access within the system. This policy should be applied to all systems and applications that require user authorization, including file servers, databases, and application interfaces. Authorization helps prevent unauthorized actions and protects sensitive information from unauthorized users' access or modification. It reduces the risk of data breaches and insider threats. |
| Accounting | System activities must be logged and audited regularly, including user actions, access attempts, and security events. Audit logs should be protected from tampering and retained for a specified period. Auditing records system activities and events, allowing for monitoring, incident investigation, and compliance purposes. It helps detect security incidents, identify anomalies, and track user actions. This policy should be applied to all systems and applications that require auditing capabilities, including network devices, servers, and security systems. Auditing enables organizations to track and review system activities for security monitoring and compliance. |

**\***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**

| **Standard** | **Principle(s)** | **Explanation** |
| --- | --- | --- |
| Data Type | 1,7 | Validating input data ensures that the data type is correct and meets the expected criteria, while sanitizing data sent to other systems prevents injection attacks and manipulation of data types. |
| Data Value | 1 | Validating input data ensures that the data values are within the expected range and do not contain any unexpected or malicious content. |
| String Correctness | 1 | Validating input data ensures that strings are correctly formatted, preventing issues such as buffer overflows or injection attacks. |
| SQL Injection | 1 | Validating input data helps prevent SQL injection attacks by ensuring that user input is properly sanitized before being used in SQL queries. |
| Memory Protection | 2,9 | Paying attention to compiler warnings helps identify potential memory-related vulnerabilities, and effective quality assurance techniques can help identify and address memory-related issues. |
| Assertions | 9 | Using assertions in code helps identify and catch potential issues during testing and debugging, improving overall code quality and security. |
| Exceptions | 9, 10 | Effective quality assurance techniques help identify and handle exceptions properly, while following a secure coding standard ensures that exception handling is done securely. |
| Don't Repeat Yourself (DRY) | 4 | Keeping code simple reduces duplication and promotes reusable code, improving maintainability and reducing the risk of introducing errors or vulnerabilities through repetition. |
| Principle of Least Privilege | 5, 6 | Default deny ensures that access is denied by default unless explicitly granted, reducing the risk of unauthorized access. The principle of least privilege limits user or process privileges to the minimum necessary, reducing the potential impact of compromised accounts or processes. |
| Secure Authentication and Authorization | 3, 6 | Incorporating security policies into the system's architecture and design helps ensure secure authentication and authorization mechanisms are implemented. Adhering to the principle of least privilege ensures that users or processes only have the necessary privileges for authentication and authorization. |

**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 |  |
| 2.0 | 12/01/2023 | Completed Document | Geoffrey Nix | [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 |