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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | This principle emphasizes the importance of thoroughly validating and sanitizing any data received from external sources, such as user inputs or data from the internet, to prevent potential vulnerabilities like SQL injection or cross-site scripting attacks. |
| 1. Heed Compiler Warnings | Pay attention to warnings and alerts issued by your code compiler or development environment. These warnings often point out potential security vulnerabilities or coding errors that could be exploited by attackers. |
| 1. Architect and Design for Security Policies | When designing software or systems, prioritize security considerations from the beginning. Develop a security architecture that aligns with organizational security policies and standards to ensure robust protection. |
| 1. Keep It Simple | Simplicity is key to security. Avoid unnecessary complexity in your code and system design, as complexity often introduces vulnerabilities. A simpler design is easier to analyze and secure. |
| 1. Default Deny | Implement a default-deny approach, which means that unless explicitly allowed, all access and interactions are denied. This minimizes the attack surface and reduces the risk of unauthorized access. |
| 1. Adhere to the Principle of Least Privilege | Assign the minimum level of access or permissions necessary for users, processes, and systems to perform their tasks. Limiting access helps prevent misuse or abuse of privileges. |
| 1. Sanitize Data Sent to Other Systems | Before transmitting data to external systems, ensure it is properly sanitized and validated. This helps prevent the unintentional exposure of sensitive information or the exploitation of vulnerabilities in other systems. |
| 1. Practice Defense in Depth | Implement multiple layers of security defenses to protect against various threats. This principle emphasizes the importance of redundancy and diversity in security measures to minimize the impact of potential breaches. |
| 1. Use Effective Quality Assurance Techniques | Employ rigorous testing and quality assurance processes to identify and rectify security vulnerabilities and bugs in your code. Thorough testing helps ensure that your software is resilient to attacks. |
| 1. Adopt a Secure Coding Standard | Establish and adhere to a secure coding standard or framework that defines best practices for writing secure code. This standard should cover topics like input validation, authentication, and encryption to ensure consistency in security measures throughout your development process. |

### 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] | Maintaining data type consistency is crucial to prevent type-related errors and vulnerabilities. When data types are inconsistent, it can lead to unexpected behavior, data corruption, and security issues. |

| **Noncompliant Code** |
| --- |
| Mixing data types without proper conversion can lead to errors: |
| int x = 5; // Compliant, x is an integer  std::string y = "2"; // Noncompliant, mixing data types |

| **Compliant Code** |
| --- |
| Ensure data type consistency by using appropriate data types consistently: |
| int x = 5; // Compliant, x is an integer  int y = 2; // Compliant, y is an integer |

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

| **Principles(s):**  Conversions and Data Types (C INT) - This principle emphasizes the importance of proper data type usage and conversions. Compliance with this principle ensures that data types are used correctly, which directly supports the Data Type standard. By adhering to this principle, developers reduce the risk of data type-related vulnerabilities and inconsistencies. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | `TypeMismatch` | This checker flags situations where there is a mismatch between the types of operands in expressions or assignments. |
| PC-Lint | 2.0 | Visual Lint | Visual Lint is a commercial Lint tool that integrates with Visual Studio IDE. It can check C/C++ code for various issues, including data type-related problems. |
| Clang Static Analyzer | 12 | `-analyze` | Clang's Static Analyzer is primarily a tool for finding a wide range of code issues, including data type-related problems, during the compilation process. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Data validation prevents security risks associated with malicious or malformed input, such as SQL injection, buffer overflows, or cross-site scripting (XSS). |

| **Noncompliant Code** |
| --- |
| Failing to validate data can lead to security vulnerabilities: |
| #include <iostream>  #include <regex>  std::string email = "user@example.com; DROP TABLE users;";  std::regex pattern("[^@]+@[^@]+\\.[^@]+");  if (std::regex\_match(email, pattern)) {  std::cout << "Email is valid." << std::endl; // Noncompliant, doesn't reject malicious input  } |

| **Compliant Code** |
| --- |
| Validate user inputs to ensure they conform to expected data formats and ranges: |
| #include <iostream>  #include <regex>  std::string email = "user@example.com";  std::regex pattern("[^@]+@[^@]+\\.[^@]+");  if (std::regex\_match(email, pattern)) {  std::cout << "Email is valid." << std::endl; // Compliant  } |

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

| **Principles(s):**  Conversions and Data Types (C INT) and Integer Operations (INT) - These principles highlight the need for proper handling of data values, including integer operations and type conversions. Compliance ensures that data values are used safely, aligning with the Data Value standard. Following these principles helps prevent common issues related to data value manipulation. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Likely | Low | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12 | core.DivideZero  core.NonNullParamChecker  alpha.unix.Stream | These checkers can help identify a wide range of issues related to data handling, value consistency, and potential vulnerabilities in your C and C++ code. |
| Flawfinder | 2.0.19 | Security-Relevant Variable Names | It identifies variable names that might be related to security-sensitive operations, such as passwords or encryption keys, and flags them for review. |
| PVS-Studio | 7.26 | V547 (Mismatched Read/Write/Update of the 'Foo' Variable)  V560 (A part of conditional expression is always true/false) | This checker detects situations where a variable is read or updated in a way that doesn't match its intended usage. It can help identify cases where data values are not being validated correctly.  Flags conditional expressions that are always evaluated as true or false. These conditions may impact data validation logic. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Sanitizing strings is essential to prevent security vulnerabilities, like SQL injection or XSS attacks, by removing or neutralizing harmful characters and sequences. |

| **Noncompliant Code** |
| --- |
| Using unsanitized strings can expose your application to injection attacks: |
| #include <iostream>  std::string user\_input = "<script>alert('XSS');</script>";  std::cout << user\_input << std::endl; // Noncompliant, vulnerable to XSS |

| **Compliant Code** |
| --- |
| Sanitize user-generated or external data before using it in string operations to prevent security vulnerabilities: |
| #include <iostream>  #include <string>  std::string user\_input = "<script>alert('XSS');</script>";  std::string sanitized\_input = sanitize\_html(user\_input);  std::cout << sanitized\_input << std::endl; // Compliant, XSS is prevented |

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

| **Principles(s):**  Conversions and Data Types (C INT) – This principle addresses the proper use of data types and conversions, which includes string handling. Compliance with this principle ensures that strings are used correctly and consistently, supporting the String Correctness standard. It helps prevent string-related vulnerabilities and ensures robust string manipulation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | Null Pointer Checks | Cppcheck can detect potential null pointer dereferences, which can help ensure that string pointers are valid before accessing them. |
| Flawfinder | 2.0.19 | Unsafe String Handling | May detect issues where strings are not properly sanitized or validated before use, which could lead to security vulnerabilities. |
| Coverity | 2022.12 | String Manipulation Checkers | These checkers focus on string manipulation functions, identifying potential issues like improper use of string functions, missing null-termination, and incorrect length calculations. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Parameterized queries ensure that user inputs are treated as data, not executable code, preventing SQL injection vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Constructing SQL queries using user inputs directly can introduce SQL injection vulnerabilities: |
| #include <iostream>  #include <sqlite3.h>  sqlite3\* db;  sqlite3\_stmt\* stmt;  const char\* username = "admin'; DROP TABLE users;--";  const char\* query = "SELECT \* FROM users WHERE username = '" + std::string(username) + "'";  int result = sqlite3\_prepare\_v2(db, query, -1, &stmt, nullptr);  // Noncompliant, vulnerable to SQL injection |

| **Compliant Code** |
| --- |
| Use prepared statements to prevent SQL injection: |
| #include <iostream>  #include <sqlite3.h>  sqlite3\* db;  sqlite3\_stmt\* stmt;  const char\* username = "admin";  const char\* query = "SELECT \* FROM users WHERE username = ?";  int result = sqlite3\_prepare\_v2(db, query, -1, &stmt, nullptr);  if (result == SQLITE\_OK) {  sqlite3\_bind\_text(stmt, 1, username, -1, SQLITE\_STATIC);  // Execute the prepared statement  } |

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

| **Principles(s):**  Input Output (FIO) - Focuses on input and output operations. Compliance with this principle includes secure handling of user input, which directly aligns with the SQL Injection standard. Adhering to this principle helps prevent SQL injection vulnerabilities by properly validating and sanitizing input data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP ZAP | 2.13 | SQL Injection (Detection) | This is a fundamental checker in OWASP ZAP designed to detect SQL injection vulnerabilities in web applications. It analyzes input fields and queries for potential SQL injection attacks. |
| SQLMap | 1.7 | Error-based Detection | SQLMap sends malicious payloads to web application parameters and analyzes error messages returned by the database. If the error messages contain SQL-related syntax, it considers it a potential SQL injection point. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Buffer overflows can lead to memory corruption and unauthorized code execution. Prevention mechanisms ensure code behaves as intended. |

| **Noncompliant Code** |
| --- |
| Failing to prevent buffer overflows can lead to memory corruption: |
| #include <iostream>  #include <cstring>  char dest[20];  const char\* source = "This is a long string that can cause a buffer overflow";  strncpy(dest, source, sizeof(dest)); // Noncompliant, potential buffer overflow |

| **Compliant Code** |
| --- |
| Use safe string functions and bounds checking to prevent buffer overflows: |
| #include <iostream>  #include <cstring>  char dest[20];  const char\* source = "This is a long string";  strncpy(dest, source, sizeof(dest) - 1);  dest[sizeof(dest) - 1] = '\0'; // Compliant, prevents buffer overflow |

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

| **Principles(s):**  Memory Management (MEM) - Emphasizes proper memory management practices, including memory protection. Compliance with this principle supports the Memory Protection standard by ensuring that memory is allocated, accessed, and freed securely. Following this principle helps prevent memory-related vulnerabilities and buffer overflows. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AddressSanitizer | 3.1 | Buffer Overflows | Can detect buffer overflows by checking memory accesses that go beyond the bounds of an allocated buffer. |
| Valgrind | 3.21.0 | Memcheck | Memcheck is the default memory checker in Valgrind, and it is extremely powerful for detecting memory-related issues such as memory leaks, invalid reads, and writes. While it primarily focuses on memory leaks, it can also catch various memory protection problems. |
| Clang Static Analyzer | 12 | unix.Malloc | The checker analyzes memory allocation functions like `**malloc`**, `**calloc`**, and `**realloc`** to identify potential issues such as memory leaks and improper handling of allocation failures. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions help identify and handle errors early in development, preventing unexpected behavior and enhancing code reliability. |

| **Noncompliant Code** |
| --- |
| Omitting assertions may result in uncaught errors: |
| #include <iostream>  bool file\_exists(const std::string& file\_path) {  // Check if the file exists  return false; // For demonstration purposes  }  int main() {  std::string file\_path = "myfile.txt";  if (!file\_exists(file\_path)) {  std::cout << "File does not exist" << std::endl; // Noncompliant, missing assertion  }  return 0;  } |

| **Compliant Code** |
| --- |
| Use assertions to validate critical conditions and assumptions: |
| #include <cassert>  bool file\_exists(const std::string& file\_path) {  // Check if the file exists  return true; // For demonstration purposes  }  int main() {  std::string file\_path = "myfile.txt";  assert(file\_exists(file\_path) && "File does not exist"); // Compliant  return 0;  } |

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

| **Principles(s):**  Conversions and Data Types (C INT) and Memory Management (MEM) - These principles emphasize correctness and memory management. Compliance with these principles ensures that assertions are used correctly and consistently, supporting the Assertions standard. Adhering to these principles helps developers identify and handle errors effectively. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Likely | Medium | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | Asserts with Constant Conditions | Cppcheck can identify assert statements where the condition is a constant that will always evaluate to true or false. |
| Coverity | 2022.12 | ASSERT\_SIDE\_EFFECT  MISSING\_ASSERT  UNUSED\_ASSERT  ASSERT\_ON\_EXIT | These Coverity checkers for assertions can assist in ensuring that assertions in your codebase are used correctly, effectively, and do not introduce issues or dead code. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Proper exception handling prevents application crashes and helps avoid revealing sensitive information in error messages. |

| **Noncompliant Code** |
| --- |
| Inadequate exception handling can lead to unhandled exceptions: |
| #include <iostream>  void divide(int a, int b) {  if (b == 0) {  throw std::runtime\_error("Division by zero is not allowed");  }  int result = a / b;  // Process the result  }  int main() {  try {  divide(10, 0); // Noncompliant, no error handling  } catch (const std::exception& e) {  std::cerr << e.what() << std::endl;  }  return 0;  } |

| **Compliant Code** |
| --- |
| Implement robust exception handling mechanisms: |
| #include <iostream>  void divide(int a, int b) {  try {  if (b == 0) {  throw std::runtime\_error("Division by zero is not allowed");  }  int result = a / b;  // Process the result  } catch (const std::exception& e) {  std::cerr << e.what() << std::endl; // Compliant, handles exception  }  }  int main() {  divide(10, 0);  return 0;  } |

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

| **Principles(s):**  Conversions and Data Types (C INT) and Memory Management (MEM) - Similar to the Assertions standard, Principles 3 and 5 guide developers in using exceptions correctly. Compliance with these principles ensures that exceptions are handled safely, supporting the Exceptions standard. This alignment helps prevent issues related to exception handling. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Likely | Medium | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12 | core.CallAndMessage  cplusplus.NewDelete | This checker analyzes calls and messages, including those related to exception handling. It can detect issues like null pointer dereferences that could lead to exceptions.  This checker focuses on the proper use of `**new`** and `**delete`** operators in C++. It can help identify issues related to resource management during exception handling. |
| PC-Lint | 1.2 | MISRA-C Compliance | PC-Lint offers MISRA-C compliance checking, which includes rules related to exception handling. You can enable MISRA-C checks to ensure your code follows best practices in exception handling. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Validation | [STD-008-CPP] | Apply strict input validation by defining expected data patterns and rejecting any inputs that do not conform. This helps prevent various attacks like command injection and cross-site scripting (XSS). |

| **Noncompliant Code** |
| --- |
| Insufficient input validation can open avenues for attacks: |
| #include <iostream>  #include <string>  int main() {  std::string user\_input = "malicious\_input; rm -rf /";  // Process the input without validation  } // Noncompliant, vulnerable to command injection |

| **Compliant Code** |
| --- |
| Validate user input to reject potentially harmful characters: |
| #include <iostream>  #include <string>  bool is\_input\_valid(const std::string& user\_input) {  // Check if user\_input contains harmful characters  return user\_input.find\_first\_of(";|&") == std::string::npos;  }  int main() {  std::string user\_input = "safe\_input";  if (is\_input\_valid(user\_input)) {  // Process the input  } // Compliant, rejects potentially harmful characters  } |

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

| **Principles(s):**  Conversions and Data Types (C INT), Integer Operations (INT), Input Output (FIO), and Errors (ERR) - These principles collectively address input validation, error handling, and data type conversions. Compliance with these principles supports the Input Validation standard by guiding developers to validate and sanitize input data properly. It helps prevent a wide range of security vulnerabilities related to user input. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Checkmarx | 2.94.10 | Cross-Site Scripting (XSS) Checkers | **Reflected XSS**: Detects instances where user inputs are directly echoed into web responses without proper validation or encoding.  **Stored XSS**: Identifies stored XSS vulnerabilities, where malicious inputs are stored in the application and later rendered to users without proper validation. |
| Fortify | 22.2.2 | Unvalidated Input | This checker identifies instances where user input is not properly validated before being used in code execution. It helps find potential security vulnerabilities like injection attacks (e.g., SQL injection, XSS) that can occur when unvalidated input is processed. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Authentication and Authorization | [STD-009-CPP] | Implement strong authentication methods, such as multi-factor authentication (MFA), for user access to sensitive systems or data. Strong authentication helps protect against unauthorized access. |

| **Noncompliant Code** |
| --- |
| Weak or nonexistent authentication methods can lead to unauthorized access: |
| #include <iostream>  #include <string>  bool validate\_password(const std::string& username, const std::string& password) {  // Validate the password  return true; // For demonstration purposes  }  int main() {  std::string username = "user123";  std::string password = "insecure\_password";  if (validate\_password(username, password)) {  // Grant access (noncompliant, lacks strong authentication)  }  } |

| **Compliant Code** |
| --- |
| Implement strong authentication methods: |
| #include <iostream>  #include <string>  bool validate\_password(const std::string& username, const std::string& password) {  // Validate the password  return true; // For demonstration purposes  }  bool validate\_otp(const std::string& username, const std::string& otp) {  // Validate the OTP  return true; // For demonstration purposes  }  int main() {  std::string username = "user123";  std::string password = "secure\_password";  std::string otp = "123456";  if (validate\_password(username, password) && validate\_otp(username, otp)) {  // Grant access (compliant, strong authentication)  }  } |

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

| **Principles(s):**  Conversions and Data Types (C INT), Input Output (FIO), Authentication and Password Management (AUT), and Authorization (FDP) - These principles encompass various aspects of authentication and authorization. Compliance with these principles aligns with the Authentication and Authorization standard by ensuring that these critical security components are implemented correctly and securely. It helps prevent unauthorized access and data breaches. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Nessus | 10.6.1 | Insecure Authentication Protocols | Nessus can flag the use of insecure authentication protocols or configurations, such as clear text passwords or outdated authentication methods. |
| OpenVAS | 22.7.5 | Password Policy Checks  User Enumeration | OpenVAS can assess the strength and compliance of password policies, such as weak passwords or password complexity requirements.  It can detect whether an application or service allows attackers to enumerate valid usernames, which could be used in brute-force attacks. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Encryption | [STD-010-CPP] | Encrypt sensitive data both in transit and at rest using strong encryption algorithms and secure key management practices. Encryption safeguards data from unauthorized access. |

| **Noncompliant Code** |
| --- |
| Storing or transmitting sensitive information without encryption exposes it to potential theft: |
| #include <iostream>  #include <string>  int main() {  std::string data = "My sensitive data";  // Store or transmit data without encryption (noncompliant)  return 0;  } |

| **Compliant Code** |
| --- |
| Encrypt sensitive data before storing it: |
| #include <iostream>  #include <string>  #include <cryptopp/aes.h>  #include <cryptopp/filters.h>  #include <cryptopp/modes.h>  #include <cryptopp/osrng.h>  int main() {  std::string plaintext = "My sensitive data";  byte key[CryptoPP::AES::MAX\_KEYLENGTH];  byte iv[CryptoPP::AES::BLOCKSIZE];  // Initialize key and iv securely (not shown here)    CryptoPP::AES::Encryption aesEncryption(key, CryptoPP::AES::MAX\_KEYLENGTH);  CryptoPP::CBC\_Mode\_ExternalCipher::Encryption cbcEncryption(aesEncryption, iv);  std::string ciphertext;  CryptoPP::StreamTransformationFilter stfEncryptor(cbcEncryption, new CryptoPP::StringSink(ciphertext));  stfEncryptor.Put(reinterpret\_cast<const unsigned char\*>(plaintext.c\_str()), plaintext.length());  stfEncryptor.MessageEnd();  // Store ciphertext securely (not shown here)  return 0;  } |

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

| **Principles(s):**  Conversions and Data Types (C INT) – These lay the foundation for secure data handling, including encryption. Compliance with these principles supports the Encryption standard by ensuring that encryption is applied correctly to protect sensitive data. It helps prevent data exposure and security breaches. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Nessus | 10.6.1 | Certificate Chain Validation | Nessus checks the validity of certificate chains and whether they are correctly configured. |
| OpenSSL Analyzer | 3.1 | Vulnerability Scanning | The tool scans for known vulnerabilities in the OpenSSL libraries themselves. It ensures that you are using an up-to-date and patched version of OpenSSL. |

### 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 |
| STD-002-CPP] | Medium | Likely | Low | Medium | 3 |
| STD-003-CPP] | High | Likely | Medium | High | 2 |
| STD-004-CPP] | High | Likely | High | High | 1 |
| STD-005-CPP] | High | Likely | High | High | 1 |
| STD-006-CPP] | Medium | Likely | Medium | Medium | 3 |
| STD-007-CPP] | Medium | Likely | Medium | Medium | 3 |
| STD-008-CPP] | High | Likely | High | High | 1 |
| STD-009-CPP] | High | Likely | High | High | 1 |
| STD-010-CPP] | High | Likely | High | High | 1 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption at rest is the practice of securing data when it is stored on physical or digital storage devices. All sensitive data, whether on physical devices like servers and storage drives or in cloud-based storage solutions, must be encrypted at rest using strong encryption algorithms and secure key management practices. Encryption at rest protects data from unauthorized access in case of physical theft, data breaches, or unauthorized access to storage devices. It ensures that even if data is compromised, it remains unreadable without the proper decryption keys. This policy applies to all storage devices and data repositories used within the organization, including databases, file servers, and cloud storage solutions. |
| Encryption at flight | Encryption in flight, also known as data in transit encryption, is the practice of securing data while it is being transmitted over a network. All communication channels and protocols used within our organization must implement encryption in flight. This includes using secure transport protocols like TLS/SSL for web applications and secure communication methods for email, file transfers, and any data exchanged over networks. Encryption in flight ensures that data remains confidential and secure during transmission. It prevents eavesdropping and man-in-the-middle attacks, safeguarding sensitive information as it moves between systems and endpoints. This policy applies to all network communications within the organization, both internal and external, regardless of the data's sensitivity level. |
| Encryption in use | Encryption in use refers to the practice of securing data while it is being actively processed or used by applications or systems. All applications and systems that process sensitive data must implement encryption in use. This includes encrypting data during computations, while it is being manipulated in memory, and ensuring that encryption keys are securely managed. Encryption in use protects sensitive data from unauthorized access and exposure during processing. It ensures that data remains confidential and secure throughout its lifecycle, including when it is actively used by applications.This policy applies to all applications, systems, and processes that involve the processing of sensitive data, including data in memory and during computations. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identity of users, systems, or entities attempting to access our organization's resources. All users and systems must undergo a robust authentication process before gaining access to our organization's resources. This includes strong password policies, multi-factor authentication (MFA), and secure identity management. Authentication ensures that only authorized individuals or entities can access sensitive data and systems. It helps prevent unauthorized access, data breaches, and insider threats. This policy applies to all access points and systems within the organization, including network access, application access, and physical access to facilities. |
| Authorization | Authorization is the process of granting or denying access to specific resources and functionalities based on the authenticated user's or system's permissions and privileges. Our organization must implement role-based access control (RBAC) and fine-grained access control mechanisms to ensure that users and systems have access only to the resources and functions they are explicitly authorized to access. Authorization prevents unauthorized users or systems from accessing sensitive data or performing actions that could compromise security. It enforces the principle of least privilege (PoLP) and reduces the attack surface. This policy applies to all resources, applications, and systems within the organization, defining who can access what and under what conditions. |
| Accounting | Accounting, also known as auditing, involves the tracking and logging of activities and events related to user and system interactions with organization resources. Our organization must implement comprehensive logging and auditing mechanisms to record and monitor user and system activities, access attempts, and security events. Log data should be securely stored and regularly reviewed. Accounting provides visibility into user and system behavior, helping detect and respond to security incidents, policy violations, and anomalies. It supports compliance efforts and forensic investigations. This policy applies to all systems, applications, and resources within the organization, ensuring that logging and auditing are consistently applied and reviewed. |

**\***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 |  |
| 1.1 | 09/17/2023 | Revision 1 | Kentrell Edwards |  |
| 1.2 | 10/07/2023 | Revision 2 | Kentrell Edwards |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

| Language | Acronym |
| --- | --- |
| C++ | CPP |
| C | CLG |
| Java | JAV |