**Green Pace Developer: Security Policy Guide**



# 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 | Data validation is key to preventing unsafe data from entering an application. All input data should be properly sanitized and validated before being utilized. This process involves checking user inputs against a set of rules (e.g. format, type, length, allowable characters) to make sure that it adheres to what is permitted. This principle is especially crucial for preventing common vulnerabilities like SQL injection, cross-site scripting (XSS), and command injection. |
| 1. Heed Compiler Warnings | Compiler warnings are generated as a way to catch potential vulnerabilities and coding mistakes at a very early stage. Compiler warnings should always be reviewed and responded to as a best practice. Using deprecated code and unsafe practices should be avoided as these often compromise the security of an application. |
| 1. Architect and Design for Security Policies | Ensuring that security is a forethought rather than an afterthought is crucial to creating systems that are fundamentally secure. Security should be embedded into the architecture and design phase of software development, ensuring that security policies are enforced throughout the system’s lifecycle. Adding security to a system after its initial design only complicates the process and can lead to security issues down the road. |
| 1. Keep It Simple | Designing code that is simple and effective leaves less room for potential errors and security flaws. When code is overly complex, it is more difficult to understand, maintain, and audit. Code should be created with the least amount of complexity possible. |
| 1. Default Deny | This principle dictates that access should be denied by default, and explicit permissions should be required to grant access to resources. By establishing a “deny-first” approach, systems can prevent unauthorized access from the outset. This is particularly important for safeguarding sensitive data and critical system functions. For instance, when configuring a firewall, rules should be set to block all incoming traffic by default, only allowing specific, trusted sources through. |
| 1. Adhere to the Principle of Least Privilege | Similar to the default deny principle, access to systems and resources should only be provided when it is necessary for a function to be performed. This applies to users, processes, and systems. This is important for reducing the potential impact that a breach may have, if one is unable to be prevented. The less access that an attacker has, the less damage they are able to do. |
| 1. Sanitize Data Sent to Other Systems | Before data is transmitted to another system, potentially harmful content should be removed or neutralized. This principle is crucial to preventing injection attacks and other exploits where malicious data could compromise external systems. For instance, when integrating with a database, ensuring that data has been properly escaped or using parameterized queries prevents SQL injection attacks that could lead to unauthorized data access or manipulation. |
| 1. Practice Defense in Depth | Defense in depth (DiD) is a cybersecurity defense strategy that employs a layered approach to security by deploying multiple defense mechanisms across a system to protect against a wide range of potential attack vectors. The underlying concept is that no single security control can be entirely effective on its own, so DiD integrates various measures at different levels to create overlapping defenses. These layers can include physical security, network protection (e.g., firewalls and intrusion detection systems) endpoint security (e.g., antivirus and anti-malware software), application security measures, and user authentication protocols. Each layer acts as a barrier, providing backup should one layer be breached. |
| 1. Use Effective Quality Assurance Techniques | Implementing thorough testing and quality assurance processes helps detect and fix security vulnerabilities early in the development cycle. Techniques such as code reviews, static analysis, and automated testing can identify weaknesses and ensure that the code meets secure coding standards. |
| 1. Adopt a Secure Coding Standard | Following a set of secure coding practices ensures consistency and mitigates common vulnerabilities. Secure coding standards help developers write code that avoids known pitfalls and aligns with best practices for security, making software more resilient to attacks. |

### 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** | **Do not cast to an out-of-range enumeration value** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Casting an integer to an enumeration type should only be done if the integer value falls within the valid range of the enumeration. Casting out-of-range values to an enumeration can result in undefined behavior and potentially causes errors in a program. It is important to ensure that the cast value is within the valid set of enumeration values. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the function f uses a color choice to determine whether it is within the range of available colors. However, it does this by casting colorChoice to the enumeration type Color which may not be able to represent the given colorChoice. If colorChoice is outside of the representable values of the Color enum, the cast results in an unspecified value, and using that value within the if statement results in unspecified behavior. |
| enum Color {    Red,    Green,    Blue  };    void f(int colorChoice) {    Color color = static\_cast<Color>(colorChoice);      if (color < Red || color > Blue) {      // Handle error    }  } |

| **Compliant Code** |
| --- |
| In this compliant code example, the value of colorChoice is checked to ensure that it is within the valid range of the enumeration before casting. |
| enum Color {    Red,    Green,    Blue  };    void f(int colorChoice) {    if (colorChoice < Red || colorChoice > Blue) {      // Handle error    }    Color color = static\_cast<Color>(colorChoice);  } |

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

| **Principles(s):** **Validate Input Data**  The integer being cast to an enumeration must be validated to fall within the valid range, preventing undefined behavior and ensuring safe input handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **cast-integer-to-enum** | Partially checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **LANG.CAST.COERCE**  **LANG.CAST.VALUE** | Coercion Alters Value  Cast Alters Value |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-INT50-a** | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **cast-integer-to-enum** | Partially checked |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not read uninitialized memory** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Memory must be initialized before it is read to prevent undefined behavior. Reading uninitialized memory can lead to unpredictable program behavior and hard-to-diagnose bugs. Properly initializing variable before they are accessed ensures consistent and safe behavior in C++ programs. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a variable is read before it is initialized, leading to undefined behavior. |
| #include <iostream>    void f() {    int uninitializedVar;    std::cout << “Value: “ << uninitializedVar << std::endl;  } |

| **Compliant Code** |
| --- |
| In this compliant code example, the variable has been initialized before it is read. This ensures the program will have in a predictable fashion. |
| #include <iostream>    void f() {    int initializedVar = 0;    std::cout << “Value: “ << initializedVar << std::endl;  } | |

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

| **Principles(s):** **Heed Compiler Warnings**  Uninitialized memory can lead to undefined behavior and bugs that may be caught by compiler warnings. Proper initialization addresses these issues, ensuring that potential vulnerabilities are detected early, promoting safer code and more predictable program behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **uninitialized-read** | Partially checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 3.9 | **LANG.STRUCT.RPL LANG.MEM.UVAR** | Return pointer to local Uninitialized variable |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-EXP53-a** | Avoid use before initialization |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | |  |  | | --- | --- | | **uninitialized-read** |  | | Partially checked |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Detect errors when converting a string to a number** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Appropriate error checking should be performed when converting strings to numeric types to handle potential issues such as invalid input or out-of-range values. Proper error handling prevents undefined behavior and ensures that the program can gracefully handle erroneous input without crashing or producing incorrect results. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, user input is taken through the standard input stream and converted into numbers without error checking. However, if user input cannot be converted into a number representable by a double, undefined behavior will result. |
| #include <iostream>    void processInput() {    double x, y;    std::cin >> x >> y;    std:: cout << “Values entered: ” << x << “ and “ << y << std::endl;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, exceptions are enabled so that a conversion failure will result in an exception being thrown. This approach is unable to distinguish between which values are valid and which values are not. |
| #include <iostream>  #include <limits>  void processInput() {  double x, y;  std::cin.exceptions(std::istream::failbit | std::istream::badbit);  try {  std::cin >> x >> y;  std::cout << "Values entered: " << x << " and " << y << std::endl;  } catch (const std::istream::failure& e) {  std::cerr << "Input error: Failed to read values. Please ensure valid input." << std::endl;  std::cin.clear(); // Clear the error state  std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');  }  } |

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

| **Principles(s):** **Validate Input Data**  Error checking during string-to-numeric conversions ensures input is valid and prevents undefined behavior, allowing the program to handle invalid or out-of-range values gracefully. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | |  |  | | --- | --- | |  | 2024.3 | | **C++3161** | N/A |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-ERR62-a** | The library functions atof, atoi and atol from library stdlib.h shall not be used |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2024.3 | **CERT.ERR.CONV.STR\_TO\_NUM** | N/A |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: ERR62-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr62cpp.html) | Checks for unvalidated string-to-number conversion (rule fully covered) |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent SQL injection** |
| --- | --- | --- |
| **SQL Injection** | [STD-001-JAV] | User inputs should never be directly included in SQL queries without proper validation or parameterization. Failing to do so can lead to SQL injection attacks, where attackers may execute arbitrary SQL commands, leading to data breaches or manipulation. The use of parameterized queries or prepared statements is essential to avoid these vulnerabilities. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, user input is concatenated directly into a SQL query, which opens the system up to SQL injection vulnerabilities. |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.ResultSet;  import java.sql.SQLException;  import java.sql.Statement;  class UserAuth {  public Connection establishConnection() throws SQLException {  DriverManager.registerDriver(new com.mysql.cj.jdbc.Driver());  String dbUrl = ConfigManager.getConfig("database.url");  // Example URL format: "jdbc:mysql://<HOST>:3306/dbname"  return DriverManager.getConnection(dbUrl);  }  String createPasswordHash(char[] password) {  // Logic to hash the password  }  public void authenticateUser(String username, char[] password)  throws SQLException {  Connection connection = establishConnection();  if (connection == null) {  // Handle error  }  try {  String hashedPassword = createPasswordHash(password);  // SQL injection vulnerability here: direct user input is used in the query  String query = "SELECT \* FROM users WHERE username = '" + username +  "' AND password = '" + hashedPassword + "'";  Statement stmt = connection.createStatement();  ResultSet rs = stmt.executeQuery(query);  if (!rs.next()) {  throw new SecurityException("Invalid username or password");  }  // Authenticated; continue processing  } finally {  try {  connection.close();  } catch (SQLException ex) {  // Handle exception  }  }  }  } |

| **Compliant Code** |
| --- |
| This compliant example uses a PreparedStatement with parameterized queries to safely handle user input and prevent SQL injection. |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.PreparedStatement;  import java.sql.ResultSet;  import java.sql.SQLException;  public class UserAuth {  public Connection establishConnection() throws SQLException {  DriverManager.registerDriver(new com.mysql.cj.jdbc.Driver());  String dbUrl = ConfigManager.getConfig("database.url");  // Example URL format: "jdbc:mysql://<HOST>:3306/dbname"  return DriverManager.getConnection(dbUrl);  }  String createPasswordHash(char[] password) {  // Logic to hash the password  }  public void authenticateUser(String username, char[] password)  throws SQLException {  Connection connection = establishConnection();  if (connection == null) {  // Handle error  }  try {  String hashedPassword = createPasswordHash(password);  // Use a parameterized query to prevent SQL injection  String query = "SELECT \* FROM users WHERE username = ? AND password = ?";  PreparedStatement stmt = connection.prepareStatement(query);  stmt.setString(1, username);  stmt.setString(2, hashedPassword);  ResultSet rs = stmt.executeQuery();  if (!rs.next()) {  throw new SecurityException("Invalid username or password");  }  // Authenticated; proceed with further actions  } finally {  try {  connection.close();  } catch (SQLException ex) {  // Handle exception  }  }  }  } |

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

| **Principles(s):** **Sanitize Data Sent to Other Systems**  Validating and parameterizing user inputs ensures potentially harmful content is neutralized before being included in SQL queries, preventing SQL injection attacks and protecting data integrity. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [The Checker Framework](https://wiki.sei.cmu.edu/confluence/display/java/The+Checker+Framework) | 2.1.3 | **Tainting Checker** | Trust and security errors  (see Chapter 8) |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **JAVA.IO.INJ.SQL** | SQL Injection (Java) |
| [Parasoft Jtest](https://wiki.sei.cmu.edu/confluence/display/java/Parasoft) | 2024.1 | **CERT.IDS00.TDSQL** | Protect against SQL injection |
| [Findbugs](https://wiki.sei.cmu.edu/confluence/display/java/Findbugs) | 1.0 | **SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE** | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Detect and handle memory allocation errors** |
| --- | --- | --- |
| **Memory Protection** | [STD-004-CPP] | Programs should properly detect and handle memory allocation failures to avoid undefined behavior, program crashes, or security vulnerabilities. Failure to check for successful memory allocation can lead to dereferencing null pointers and subsequent program instability. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, raw pointers are used for memory allocation without proper error handling or automatic memory management, which can lead to memory leaks if an exception is thrown. |
| #include <iostream>  struct X { /\* ... \*/ };  struct Y { /\* ... \*/ };  void process(X\*, Y\*);  void execute() {  process(new X, new Y); // Potential for memory leak if an exception occurs  } |

| **Compliant Code** |
| --- |
| In this compliant version, std::unique\_ptr is used to ensure that memory is automatically managed and released, even if an exception is thrown. |
| #include <iostream>  #include <memory>  struct X { /\* ... \*/ };  struct Y { /\* ... \*/ };  void process(std::unique\_ptr<X> x, std::unique\_ptr<Y> y);  void execute() {  process(std::make\_unique<X>(), std::make\_unique<Y>()); // Memory is managed safely  } |

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

| **Principles(s):** **Validate Input Data**  Checking for successful memory allocation ensures that inputs (in this case, system resources) are valid before use, preventing undefined behavior and maintaining program stability. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Coverity) | 7.5 | **CHECKED\_RETURN** | Finds inconsistencies in how function call return values are handled |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 2023.1 | **CERT\_CPP-MEM52-a** **CERT\_CPP-MEM52-b** | Check the return value of new Do not allocate resources in function argument list because the order of evaluation of a function's parameters is undefined |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: MEM52-CPP](https://www.mathworks.com/help/bugfinder/ref/certcmem52cpp.html) | Checks for unprotected dynamic memory allocation (rule partially covered) |
| |  |  | | --- | --- | | [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) |  | | |  |  | | --- | --- | |  | 7.33 | | [**V522**](https://pvs-studio.com/en/docs/warnings/v522/)**,** [**V668**](https://pvs-studio.com/en/docs/warnings/v668/) | N/A |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use a static assertion to test the value of a constant expression** |
| --- | --- | --- |
| **Assertions** | [STD-005-CPP] | Constant expressions should be checked at compile time using static\_assert in place of assert(). This practice helps catch logic errors early in the development process and prevents issues that could arise from incorrect assumptions about constant values. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the code relies on assert() to validate the structure's size, which only performs checks at runtime. |
| #include <assert.h>  struct dataBlock {  char HEADER;  int ID;  double VALUE;  };  void checkSize() {  // Runtime check only  assert(sizeof(struct dataBlock) == sizeof(char) + sizeof(int) + sizeof(double));  } |

| **Compliant Code** |
| --- |
| In this compliant example, static\_assert is used to ensure the size of the structure is correct at compile time. |
| #include <assert.h>  struct dataBlock {  char HEADER;  int ID;  double VALUE;  };  // Compile-time check to ensure no padding exists  static\_assert(sizeof(struct dataBlock) == sizeof(char) + sizeof(int) + sizeof(double),  "Structure must not have any padding");  } |

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

| **Principles(s):** **Use Effective Quality Assurance Techniques**  Using static\_assert for compile-time checks identifies logic errors early in development, reducing runtime issues and improving code reliability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-DCL03** | [Insert text.] |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **(customization)** | Users can implement a custom check that reports uses of the assert() macro |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | **CC2.DCL03** | Fully implemented |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **44 S** | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-006-CPP] | All exceptions must be handled appropriately in C++ programs. Unhandled exceptions can cause program termination, lead to resource leaks, and potentially expose the program to undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, a function can throw an exception that is not caught, leading to program termination. |
| #include <iostream>  #include <stdexcept>  void riskyFunction() {  throw std::runtime\_error("An error occurred");  }  void callFunction() {  riskyFunction(); // No exception handling provided  std::cout << "Function completed successfully" << std::endl;  } |

| **Compliant Code** |
| --- |
| In this compliant version, exceptions are handled using a try-catch block to ensure the program can gracefully handle errors. |
| #include <iostream>  #include <stdexcept>  void riskyFunction() {  throw std::runtime\_error("An error occurred");  }  void callFunction() {  try {  riskyFunction();  std::cout << "Function completed successfully" << std::endl;  } catch (const std::runtime\_error& e) {  std::cerr << "Caught runtime error: " << e.what() << std::endl;  // Handle error appropriately, such as cleaning up resources or logging  } catch (const std::exception& e) {  std::cerr << "Caught general exception: " << e.what() << std::endl;  // Handle any other standard exception  } catch (...) {  std::cerr << "Caught an unknown exception" << std::endl;  // Handle non-standard exceptions or unknown errors  }  } |

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

| **Principles(s):** **Architect and Design for Security Policies**  Properly handling all exceptions ensures that error scenarios are accounted for during design, preventing crashes, resource leaks, and undefined behavior while supporting a secure and stable architecture. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Medium | Low | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **main-function-catch-all early-catch-all** | Partially checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-ERR51** | N/A |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **LANG.STRUCT.UCTCH** | Unreachable Catch |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **527 S** | Partially implemented |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Ensure your random number generator is properly seeded** |
| --- | --- | --- |
| **Random Numbers** | [STD-007-CPP] | Pseudorandom number generators (PRNGs) should be correctly seeded to produce non-deterministic and varied sequences of numbers. Failing to seed an PRNG properly can result in the same sequence of numbers being generated each time the program runs, which can undermine the purpose of randomization and contribute to vulnerabilities, especially in security protocols. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the RNG is not seeded, which causes the generated numbers to follow the same pattern each time the program is executed. |
| #include <random>  #include <iostream>  void generateSequence() {  std::mt19937 rng; // Not seeded properly  for (int i = 0; i < 5; ++i) {  std::cout << rng() << ", ";  }  } |

| **Compliant Code** |
| --- |
| In this compliant example, std::random\_device is used to seed std::mt19937, ensuring a varied and less predictable sequence. |
| #include <random>  #include <iostream>  void generateSequence() {  std::random\_device seedGen;  std::mt19937 rng(seedGen()); // Properly seeded with a non-deterministic source  for (int i = 0; i < 5; ++i) {  std::cout << rng() << ", ";  }  } |

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

| **Principles(s):** **Adopt a Secure Coding Standard** Correctly seeding PRNGs aligns with secure coding practices, ensuring non-deterministic outputs that prevent predictable behavior, particularly in security-critical applications. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **default-construction** | Partially checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-MSC51** | N/A |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.3 | **C++5041** | N/A |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-MSC51-a** | Properly seed pseudorandom number generators |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Never hard code sensitive information** |
| --- | --- | --- |
| **Sensitive Information** | [STD-008-CPP] | Sensitive data, such as passwords, API keys, or encryption keys, should never be hard-coded into source code. Hard-coded secrets can be exposed through version control, code leaks, or reverse engineering, leading to security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, sensitive information is hard-coded directly in the source code, exposing it to potential security risks. |
| #include <iostream>  #include <string>  void authenticateUser() {  std::string password = "SuperSecret123"; // Hard-coded sensitive information  std::cout << "Authenticating with password: " << password << std::endl;  } |

| **Compliant Code** |
| --- |
| In this compliant example, the password is provided securely through a safer method, such as an environment variable or secure user input. |
| #include <iostream>  #include <cstdlib> // For getenv()  void authenticateUser() {  const char\* password = std::getenv("APP\_PASSWORD"); // Get password from environment variable  if (password == nullptr) {  std::cerr << "Error: Password not set in environment variables." << std::endl;  return;  }    std::cout << "Authenticating with provided password." << std::endl;  // Proceed with using the password securely  } |

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

| **Principles(s):** **Architect and Design for Security Policies** Avoiding hard-coded sensitive data in source code ensures that security is embedded in the system's architecture, preventing potential vulnerabilities from being introduced during the design phase. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/c/Helix+QAC) | 2024.3 | **C3122**  **C++3842** | N/A |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-MSC41-a** | Do not hard code string literals |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024a | [CERT C: Rule MSC41-C](https://www.mathworks.com/help/bugfinder/ref/certcrulemsc41c.html) | Checks for hard coded sensitive data (rule partially covered) |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | **2460** | Assistance provided: reports when a literal is provided as an argument to a function parameter with the ‘noliteral’ argument Semantic; several Windows API functions are marked as such and the ‘-sem’ option can apply it to other functions as appropriate |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Close files when they are no longer needed** |
| --- | --- | --- |
| Input  Output | [STD-009-CPP] | Programs should properly close files when they are no longer needed to prevent resource leaks, which can lead to memory and file descriptor exhaustion. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the file is not explicitly closed before calling std::terminate(), which can result in a resource leak. |
| #include <exception>  #include <fstream>  #include <string>  void processFile(const std::string &fileName) {  std::ifstream file(fileName);  if (!file.is\_open()) {  // Handle error (e.g., file not found)  return;  }  // Perform file operations  // ...  std::terminate(); // File is not explicitly closed before termination  } |

| **Compliant Code** |
| --- |
| In this compliant example, the file is properly closed before calling std::terminate(), ensuring that resources are released. |
| #include <exception>  #include <fstream>  #include <string>  void processFile(const std::string &fileName) {  std::ifstream file(fileName);  if (!file.is\_open()) {  // Handle error (e.g., file not found)  return;  }  // Perform file operations  // ...  file.close(); // Explicitly close the file to release resources  if (file.fail()) {  // Handle any errors that occurred during closing  }  std::terminate(); // Safe to terminate as the file is closed  } |

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

| **Principles(s):** **Use Effective Quality Assurance Techniques** Properly closing files ensures that resources are managed correctly, preventing resource leaks and system instability, which is a key aspect of maintaining high-quality, reliable code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **ALLOC.LEAK** | Leak |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-FIO51-a** | Ensure resources are freed |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2024.3 | **RH.LEAK** | N/A |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: FIO51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcfio51cpp.html) | Checks for resource leak (rule partially covered) |

### 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.

Automation can be seamlessly integrated into Green Pace’s DevOps process to enforce the coding standards defined in this policy. During the **Assess and Plan** phase, automation tools like static code analyzers (e.g., SonarQube) and security frameworks (e.g., OWASP) can identify potential issues such as uninitialized memory reads, improper random number generator seeding, or SQL injection risks before coding begins. In the **Build** phase, pre-commit hooks and automated dependency scans can ensure compliance by preventing hard-coded sensitive data and verifying memory management. The **Verify and Test** stage is critical for running automated unit tests, functional tests, and security scans to confirm adherence to standards, such as ensuring files are properly closed and exceptions are handled correctly.

As the code transitions to production, automated tools in the **Transition and Health Check** phase, such as CI/CD pipelines, can validate that the application meets all defined standards through pre-deployment checks and configurations. Once in the **Monitor and Detect** stage, runtime monitoring tools and SIEM solutions can detect anomalies like unhandled exceptions, resource leaks, or unexpected behaviors. Finally, in the **Maintain and Stabilize** phase, automated compliance scans and periodic audits can ensure continued adherence to security baselines, while rollback mechanisms provide a swift return to a compliant state if issues arise.

### 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 | Medium | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Probable | Medium | High | 4 |
| STD-003-CPP | Medium | Unlikely | Medium | Low | 2 |
| STD-004-CPP | High | Likely | Medium | High | 5 |
| STD-005-CPP | Low | Unlikely | High | Low | 1 |
| STD-006-CPP | Low | Probable | Medium | Low | 2 |
| STD-007-CPP | Medium | Likely | Low | Medium | 3 |
| STD-008-CPP | High | Probable | Medium | High | 4 |
| STD-009-CPP | Medium | Unlikely | Medium | Low | 2 |
| STD-001-JAV | High | Likely | Medium | 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***.***

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest protects data stored on disk or other storage media by converting it into a secure format, ensuring that if unauthorized parties gain access to physical storage devices, the data remains unreadable without the encryption key. It is commonly used to safeguard databases, file systems, backups, and archives by encrypting files, folders, or entire storage drives, typically using techniques like AES (Advanced Encryption Standard). This policy applies whenever sensitive data is stored on devices such as hard drives, SSDs, or cloud storage, ensuring compliance with data security regulations and protecting against breaches caused by physical theft or unauthorized access to stored data. |
| Encryption in flight | Encryption at rest protects data stored on disk or other storage media by converting it into a secure format, ensuring that if unauthorized parties gain access to physical storage devices, the data remains unreadable without the encryption key. It is commonly used to safeguard databases, file systems, backups, and archives by encrypting files, folders, or entire storage drives, typically using techniques like AES (Advanced Encryption Standard). This policy applies whenever sensitive data is stored on devices such as hard drives, SSDs, or cloud storage, ensuring compliance with data security regulations and protecting against breaches caused by physical theft or unauthorized access to stored data. |
| Encryption in use | Encryption in use protects data while it is being processed or actively used in applications, memory, or CPU operations, ensuring that even during computations, the data remains protected. Techniques such as homomorphic encryption, trusted execution environments, or secure multiparty computation are commonly employed to keep data encrypted while it is in use. The policy applies when sensitive data must be securely processed without exposing it in plaintext, particularly in cases involving highly confidential information such as financial transactions or healthcare data. This mitigates risks from memory scraping, insider threats, and other vulnerabilities during active data usage. |

| 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, devices, or systems attempting to access a resource. It ensures that only authorized parties can interact with sensitive data or services by using methods such as passwords, biometrics, security tokens, or multi-factor authentication. This policy applies by ensuring that every access request is tied to a validated identity, helping prevent unauthorized access and potential security breaches. |
| Authorization | Authorization is the process that determines what authenticated users are allowed to do within a system or network, defining their access levels and permissions. After authentication, the system checks the user's roles, attributes, or policies to grant or deny access to resources. The policy applies by ensuring that only users with the appropriate permissions can access or perform specific actions on sensitive data, systems, or applications. |
| Accounting | Accounting refers to the tracking of user activities within a system. It records what actions users perform, when they occur, and where they happen, which helps in monitoring for suspicious activity and ensures compliance with security policies. This policy applies by enforcing logging mechanisms to provide a traceable record of all access attempts and actions performed on sensitive data or systems. This record should include items such as the addition of new users, file access, database changes, and logs of user actions to ensure accountability and traceability. |

**\***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 | 11/16/2024 | Security Principles and Coding Standards. | Jeremy Snow |  |
| 1.2 | 12/06/2024 | Principle mapping, risk assessment summary, automation, and encryption and Triple A policies. | Jeremy Snow |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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