**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 | It is vital to validate and sanitize all user input to safeguard against common vulnerabilities such as SQL injection and cross-site scripting (XSS). One can prevent malicious code or data from infiltrating one’s application, thereby maintaining the confidentiality, integrity, and availability of one’s system. |
| 1. Heed Compiler Warnings | Compiler warnings often indicate potential security issues in one's code. By addressing these warnings promptly and taking them seriously, one can identify and rectify vulnerabilities before they are exploited. Failure to heed compiler warnings can result in security flaws that may prove difficult to detect and resolve later. |
| 1. Architect and Design for Security Policies | To maintain a secure system, it is crucial to design one's application with security policies in mind from the outset. This involves considering access controls, authentication mechanisms, and other security-related requirements during the architecture and design phases. One can effectively protect a system by incorporating security from the ground up. |
| 1. Keep It Simple | Simplicity is key to maintaining security. The more complex a system becomes, the more challenging it is to understand, maintain, and secure. It is possible to minimize an attack surface by identifying and mitigating vulnerabilities by keeping one's application design and implementation as straightforward as possible. |
| 1. Default Deny | The default deny principle dictates that all access should be denied by default, and only explicitly authorized actions should be permitted. This approach helps prevent unintended access and reduces the risk of unauthorized activities. Starting with a "deny-all stance" and selectively granting permissions can enhance the control and security of a system. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege stipulates that users, processes, and systems should only be granted the minimum permissions necessary to perform intended functions. This security approach helps to limit the potential damage caused by a compromised account or system, as the attacker's access and capabilities will be restricted. |
| 1. Sanitize Data Sent to Other Systems | When an application transmits data to other systems, it is necessary to sanitize that data to prevent the introduction of vulnerabilities. This process includes escaping or encoding special characters, validating the format and content of the data, and ensuring that it does not contain any malicious code or payloads. Proper data sanitization helps maintain the integrity and security of the overall system. |
| 1. Practice Defense in Depth | Defense in depth is a security strategy that involves implementing multiple layers of security controls to protect a system. This includes firewalls, intrusion detection and prevention systems, encryption, and physical security to name a few security measures. Having multiple lines of defense allows one to enhance the overall resilience of a system and make it more challenging for attackers to succeed. |
| 1. Use Effective Quality Assurance Techniques | Comprehensive quality assurance (QA) testing is necessary for identifying and addressing security vulnerabilities. This includes techniques such as static code analysis, dynamic testing, and penetration testing for an application or system. Incorporating security-focused QA practices into one's development lifecycle makes detecting and fixing issues before they are deployed to production easier. |
| 1. Adopt a Secure Coding Standard | Establishing and adhering to a secure coding standard helps ensure that one's application is developed with security in mind. This includes guidelines for input validation, error handling, authentication, authorization, and other security-related aspects of a codebase. A consistent set of secure coding practices can minimize the likelihood of introducing vulnerabilities. |

### 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** | **Guarantee that storage for strings has sufficient space for character data and the null terminator** |
| --- | --- | --- |
| **Data Type** | [STR-050-CPP] | This standard prevents buffer overflow vulnerabilities and ensures proper string handling and compatibility with string manipulation functions. |

| **Noncompliant Code** |
| --- |
| Because the input is unbounded, the following code could lead to a buffer overflow. |
| #include <iostream>  void f() {  char buf[12];  std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| The best solution for ensuring that data is not truncated and for guarding against buffer overflows is to use `std::string` instead of a bounded array, as in this compliant solution. |
| #include <iostream>  #include <string>  void f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):** Validate Input Data, Keep It Simple, Sanitize Data Sent to Other Systems, Adopt a Secure Coding Standard.  Validating input data prevents buffer overflow vulnerabilities caused by malicious actors. Keeping string handling straightforward reduces complexity and minimizes the chances of introducing vulnerabilities. Sanitizing data before sending it to other systems prevents potential vulnerabilities or compatibility issues. Adopting this coding standard promotes secure string handling and helps prevent buffer overflow vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | **P18** | **L1** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **stream-input-char-array** | Partially checked + soundly supported |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **MISC.MEM.NTERM**  **LANG.MEM.BO LANG.MEM.TO** | No space for null terminator  Buffer overrun Type overrun |
| [SonarQube C/C++ Plugin](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046388) | 4.10 | **[S3519](https://www.sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp/rules-cpp.html" \l "RSPEC-3519)** |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not cast to an out-of-range enumeration value** |
| --- | --- | --- |
| **Data Value** | [INT-050-CPP] | Casting to an out-of-range enumeration value leads to undefined behavior, as the resulting value may not correspond to any valid enumerator, making the code unpredictable and potentially introducing bugs. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example attempts to check whether a given value is within the range of acceptable enumeration values. However, it is doing so after casting to the enumeration type, which may not be able to represent the given integer value. On a two's complement system, the valid range of values that can be represented by `EnumType` are [0..3], so if a value outside of that range were passed to `f()`, the cast to `EnumType` would result in an unspecified value, and using that value within the `if` statement results in unspecified behavior. |
| enum EnumType {  First,  Second,  Third  };  void f(int intVar) {  EnumType enumVar = static\_cast<EnumType>(intVar);  if (enumVar < First || enumVar > Third) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| This compliant solution checks that the value can be represented by the enumeration type before performing the conversion to guarantee the conversion does not result in an unspecified value. It does this by restricting the converted value to one for which there is a specific enumerator value. |
| enum EnumType {  First,  Second,  Third  };  void f(int intVar) {  if (intVar < First || intVar > Third) {  // Handle error  }  EnumType enumVar = static\_cast<EnumType>(intVar);  } |

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

| **Principles(s):** Default Deny, Validate Input Data, Sanitize Data Sent to Other Systems.  Prohibiting casting to out-of-range enumeration values aligns with the default deny principle by disallowing potentially unsafe operations. Validating input data prevents the use of out-of-range values that could lead to undefined behavior. Sanitizing data sent to other systems makes sure that only valid enumeration values are passed, maintaining predictable behavior and avoiding the introduction of bugs. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | **P4** | **L3** |

**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 |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-INT50** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | **LANG.CAST.COERCE**  **LANG.CAST.VALUE** | Coercion Alters Value  Cast Alters Value |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do not attempt to create a std::string from a null pointer** |
| --- | --- | --- |
| **String Correctness** | [STR-051-CPP] | This standard prevents undefined behavior, as this process of string creation typically causes a runtime crash or exception, as the constructor expects a valid C-style string. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a `std::string object` is created from the results of a call to `std::getenv()`. However, because `std::getenv()` returns a null pointer on failure, this code can lead to undefined behavior when the environment variable does not exist (or some other error occurs). |
| #include <cstdlib>  #include <string>  void f() {  std::string tmp(std::getenv("TMP"));  if (!tmp.empty()) {  // ...  }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the results from the call to `std::getenv()` are checked for null before the `std::string` object is constructed. |
| #include <cstdlib>  #include <string>  void f() {  const char \*tmpPtrVal = std::getenv("TMP");  std::string tmp(tmpPtrVal ? tmpPtrVal : "");  if (!tmp.empty()) {  // ...  }  } |

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

| **Principles(s):** Validate Input Data, Sanitize Data Sent to Other Systems, Adopt a Secure Coding Standard.  Validating input data by checking for null pointers before creating a `std::string` prevents runtime crashes or exceptions caused by invalid input. Sanitizing data sent to other systems, such as the `std::string` constructor, allows only valid C-style strings to be passed, maintaining proper functionality. Adopting this coding standard promotes secure string handling and helps prevent undefined behavior caused by null pointers. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | **P18** | **L1** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **assert\_failure** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **LANG.MEM.NPD** | Null Pointer Dereference |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.1 | **DF4770, DF4771, DF4772, DF4773, DF4774** |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [IDS-000-J] | This standard is for preventing SQL injection, which is crucial to maintain the security and integrity of an application and database by validating and sanitizing user input before incorporating it into SQL queries, using techniques like parameterized queries, input validation, and least privilege principles. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example shows JDBC code to authenticate a user to a system. The password is passed as a `char` array, the database connection is created, and then the passwords are hashed.  Unfortunately, this code example permits a SQL injection attack by incorporating the unsanitized input argument `username` into the SQL command, allowing an attacker to inject `validuser' OR '1'='1`. The `password` argument cannot be used to attack this program because it is passed to the `hashPassword()` function, which also sanitizes the input. |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.ResultSet;  import java.sql.SQLException;  import java.sql.Statement;  class Login {  public Connection getConnection() throws SQLException {  DriverManager.registerDriver(new  com.microsoft.sqlserver.jdbc.SQLServerDriver());  String dbConnection =  PropertyManager.getProperty("db.connection");  // Can hold some value like  // "jdbc:microsoft:sqlserver://<HOST>:1433,<UID>,<PWD>"  return DriverManager.getConnection(dbConnection);  }  String hashPassword(char[] password) {  // Create hash of password  }  public void doPrivilegedAction(String username, char[] password)  throws SQLException {  Connection connection = getConnection();  if (connection == null) {  // Handle error  }  try {  String pwd = hashPassword(password);  String sqlString = "SELECT \* FROM db\_user WHERE username = '"  + username +  "' AND password = '" + pwd + "'";  Statement stmt = connection.createStatement();  ResultSet rs = stmt.executeQuery(sqlString);  if (!rs.next()) {  throw new SecurityException(  "User name or password incorrect"  );  }  // Authenticated; proceed  } finally {  try {  connection.close();  } catch (SQLException x) {  // Forward to handler  }  }  }  } |

| **Compliant Code** |
| --- |
| This compliant solution uses a parametric query with a `?` character as a placeholder for the argument. This code also validates the length of the `username` argument, preventing an attacker from submitting an arbitrarily long user name. |
| public void doPrivilegedAction(  String username, char[] password  ) throws SQLException {  Connection connection = getConnection();  if (connection == null) {  // Handle error  }  try {  String pwd = hashPassword(password);  // Validate username length  if (username.length() > 8) {  // Handle error  }  String sqlString =  "select \* from db\_user where username=? and password=?";  PreparedStatement stmt = connection.prepareStatement(sqlString);  stmt.setString(1, username);  stmt.setString(2, pwd);  ResultSet rs = stmt.executeQuery();  if (!rs.next()) {  throw new SecurityException("User name or password incorrect");  }  // Authenticated; proceed  } finally {  try {  connection.close();  } catch (SQLException x) {  // Forward to handler  }  }  } |

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

| **Principles(s):** Validate Input Data, Adhere to the Principle of Least Privilege, Sanitize Data Sent to Other Systems.  Validating and sanitizing user input before incorporating it into SQL queries is essential to prevent SQL injection vulnerabilities. Adhering to the principle of least privilege ensures that users and processes have only the necessary permissions, limiting the potential damage caused by a successful SQL injection attack. Sanitizing data sent to the database system prevents the introduction of malicious SQL code that could compromise the security and integrity of the application and database. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | **P18** | **L1** |

**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) |
| [SonarQube](https://wiki.sei.cmu.edu/confluence/display/java/SonarQube) | 9.9 | [**S2077**](https://rules.sonarsource.com/java/RSPEC-2077)  [**S3649**](https://rules.sonarsource.com/java/RSPEC-3649) | [Executing SQL queries is security-sensitive](https://rules.sonarsource.com/java/RSPEC-2077)  [SQL queries should not be vulnerable to injection attacks](https://rules.sonarsource.com/java/RSPEC-3649) |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Detect and handle memory allocation errors** |
| --- | --- | --- |
| **Memory Protection** | [MEM-052-CPP] | This standard is crucial to prevent program crashes, data corruption, and security vulnerabilities that can arise when memory allocation fails due to insufficient system resources. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an array of `int` is created using `::operator new[](std::size\_t)` and the results of the allocation are not checked. The function is marked as `noexcept`, so the caller assumes this function does not throw any exceptions. Because `::operator new[](std::size\_t)` can throw an exception if the allocation fails, it could lead to abnormal termination of the program. |
| #include <cstring>  void f(const int \*array, std::size\_t size) noexcept {  int \*copy = new int[size];  std::memcpy(copy, array, size \* sizeof(\*copy));  // ...  delete [] copy;  } |

| **Compliant Code** |
| --- |
| When using `std::nothrow`, the new operator returns either a null pointer or a pointer to the allocated space. Always test the returned pointer to ensure it is not `nullptr` before referencing the pointer. This compliant solution handles the error condition appropriately when the returned pointer is `nullptr`. |
| #include <cstring>  #include <new>  void f(const int \*array, std::size\_t size) noexcept {  int \*copy = new (std::nothrow) int[size];  if (!copy) {  // Handle error  return;  }  std::memcpy(copy, array, size \* sizeof(\*copy));  // ...  delete [] copy;  } |

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

| **Principles(s):** Validate Input Data, Practice Defense in Depth, Use Effective Quality Assurance Techniques.  Validating input data, such as checking for extreme values or sizes that could cause memory allocation issues, helps prevent crashes and data corruption. Practicing defense in depth by implementing multiple layers of memory protection mechanisms, such as error handling and resource monitoring, enhances the overall resilience of the system against memory allocation failures. Using effective quality assurance techniques, including testing for memory allocation errors and boundary conditions, helps identify and address potential issues before they manifest in production environments. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | **P18** | **L1** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Compass/ROSE](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Rose) |  |  |  |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Coverity) | 7.5 | **CHECKED\_RETURN** | Finds inconsistencies in how function call return values are handled |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.1 | **C++3225, C++3226, C++3227, C++3228, C++3229, C++4632** |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Avoid information leakage when passing a class object across a trust boundary** |
| --- | --- | --- |
| **Assertions** | [DCL-055-CPP] | This standard prevents the passing a class object across a trust boundary (e.g., between modules or systems with different security levels) in a way that can potentially leak sensitive information if the object's internal state is not properly secured, leading to unintended data exposure and compromising the system's security. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, `arg` is value-initialized through direct initialization. Because `test` does not have a user-provided default constructor, the value-initialization is preceded by a zero-initialization that guarantees the padding bits are initialized to `0` before any further initialization occurs. It is akin to using `std::memset()` to initialize all of the bits in the object to `0`. However, compilers are free to implement `arg.b = 2` by setting the low byte of a 32-bit register to `2`, leaving the high bytes unchanged, and storing all 32 bits of the register into memory. This could leak the high-order bytes resident in the register to a user. |
| #include <cstddef>  struct test {  int a;  char b;  int c;  };  // Safely copy bytes to user space  extern int copy\_to\_user(void \*dest, void \*src, std::size\_t size);  void do\_stuff(void \*usr\_buf) {  test arg{};  arg.a = 1;  arg.b = 2;  arg.c = 3;  copy\_to\_user(usr\_buf, &arg, sizeof(arg));  } |

| **Compliant Code** |
| --- |
| This compliant solution serializes the structure data before copying it to an untrusted context. This code ensures that no uninitialized padding bits are copied to unprivileged users. The structure copied to user space is now a packed structure and the `copy\_to\_user()` function would need to unpack it to recreate the original, padded structure. |
| #include <cstddef>  #include <cstring>  struct test {  int a;  char b;  int c;  };  // Safely copy bytes to user space.  extern int copy\_to\_user(void \*dest, void \*src, std::size\_t size);  void do\_stuff(void \*usr\_buf) {  test arg{1, 2, 3};  // May be larger than strictly needed.  unsigned char buf[sizeof(arg)];  std::size\_t offset = 0;  std::memcpy(buf + offset, &arg.a, sizeof(arg.a));  offset += sizeof(arg.a);  std::memcpy(buf + offset, &arg.b, sizeof(arg.b));  offset += sizeof(arg.b);  std::memcpy(buf + offset, &arg.c, sizeof(arg.c));  offset += sizeof(arg.c);  copy\_to\_user(usr\_buf, buf, offset /\* size of info copied \*/);  } |

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

| **Principles(s):** Validate Input Data, Sanitize Data Sent to Other Systems, Practice Defense in Depth, Adhere to the Principle of Least Privilege.  Validating input data verifies the class object passed across a trust boundary does not contain any malicious or unexpected data that could leak sensitive information. Sanitizing data sent to other systems, such as removing or encrypting sensitive fields before passing the object, helps prevent unintended data exposure. Practicing defense in depth by implementing multiple layers of security controls, such as access control mechanisms and data encryption, improves the protection of sensitive information within the class object. Adhering to the principle of least privilege assures that the receiving module or system has only the necessary access rights to the class object, limiting the potential impact of information leakage. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | **P1** | **L3** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-DCL55** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **MISC.PADDING.POTB** | Padding Passed Across a Trust Boundary |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.1 | **DF4941, DF4942, DF4943** |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions thrown before main() begins executing** |
| --- | --- | --- |
| **Exceptions** | [ERR-058-CPP] | This standard ensures proper program startup and prevents unexpected termination. |

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

| **Compliant Code** |
| --- |
| This compliant solution makes `globalS` into a local variable with static storage duration, allowing any exceptions thrown during object construction to be caught because the constructor for `S` will be executed the first time the function `globalS()` is called rather than at program startup. This solution does require the programmer to modify source code so that previous uses of `globalS` are replaced by a function call to `globalS()`. |
| struct S {  S() noexcept(false);  };  S &globalS() {  try {  static S s;  return s;  } catch (...) {  // Handle error, perhaps by logging it and gracefully terminating the application.  }  // Unreachable.  } |

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

| **Principles(s):** Heed Compiler Warnings, Use Effective Quality Assurance Techniques, Adopt a Secure Coding Standard.  Heeding compiler warnings related to unhandled exceptions thrown before main() begins executing helps identify potential issues early in the development process. Using effective quality assurance techniques, such as comprehensive exception testing and static code analysis, helps detect and address exceptions that may occur before main() starts. Adopting a secure coding standard that mandates handling all exceptions thrown before main() begins executing promotes consistent and reliable program startup, preventing unexpected termination. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | **P9** | **L2** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **potentially-throwing-static-initialization** | Partially checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-ERR58** |  |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | cert-err58-cpp | Checked by clang-tidy |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Ensure your random number generator is properly seeded** |
| --- | --- | --- |
| Miscellaneous | [MSC-051-CPP] | With a unique and unpredictable value, such as the current time or system entropy, this standard guarentees that the generated random numbers are not predictable and repeatable across program runs, which is essential for security-sensitive applications and simulations. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example generates a sequence of 10 pseudorandom numbers using the Mersenne Twister engine. No matter how many times this code is executed, it always produces the same sequence because the default seed is used for the engine. |
| #include <random>  #include <iostream>  void f() {  std::mt19937 engine;  for (int i = 0; i < 10; ++i) {  std::cout << engine() << ", ";  }  } |

| **Compliant Code** |
| --- |
| This compliant solution uses `std::random\_device` to generate a random value for seeding the Mersenne Twister engine object. The values generated by `std::random\_device` are nondeterministic random numbers when possible, relying on random number generation devices, such as `/dev/random`. When such a device is not available, `std::random\_device` may employ a random number engine; however, the initial value generated should have sufficient randomness to serve as a seed value. |
| #include <random>  #include <iostream>  void f() {  std::random\_device dev;  std::mt19937 engine(dev());  for (int i = 0; i < 10; ++i) {  std::cout << engine() << ", ";  }  } |

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

| **Principles(s):** Validate Input Data, Keep It Simple, Practice Defense in Depth.  Validating the input used for seeding the random number generator guarantees that the seed value is unique, unpredictable, and obtained from a reliable source, such as the current time or system entropy. Keeping the seeding process simple and straightforward reduces the chances of introducing vulnerabilities or weaknesses in the random number generation. Practicing defense in depth by combining multiple sources of entropy for seeding and using well-established cryptographic random number generators improves the overall security and unpredictability of the generated random numbers. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Likely | Low | **P18** | **L1** |

**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** | [Insert text.] |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **HARDCODED.SEED** **MISC.CRYPTO.TIMESEED** | Hardcoded Seed in PRNG Predictable Seed in PRNG |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do not use pointer-to-member operators to access nonexistent members** |
| --- | --- | --- |
| Object Oriented Programming | [OOP-055-CPP] | This standard prevents one from using pointer-to-member operators to access nonexistent members resulting in undefined behavior, leading to potential program crashes, data corruption, or unexpected results, as the compiler cannot detect the error at compile-time. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a null pointer-to-member value is passed as the second operand to a pointer-to-member expression, resulting in undefined behavior. |
| struct B {  virtual ~B() = default;  };  struct D : B {  virtual ~D() = default;  virtual void g() { /\* ... \*/ }  };  static void (D::\*gptr)(); // Not explicitly initialized, defaults to nullptr.  void call\_memptr(D \*ptr) {  (ptr->\*gptr)();  }  void f() {  D \*d = new D;  call\_memptr(d);  delete d;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, `gptr` is properly initialized to a valid pointer-to-member value instead of to the default value of `nullptr`. |
| struct B {  virtual ~B() = default;  };  struct D : B {  virtual ~D() = default;  virtual void g() { /\* ... \*/ }  };  static void (D::\*gptr)() = &D::g; // Explicitly initialized.  void call\_memptr(D \*ptr) {  (ptr->\*gptr)();  }  void f() {  D \*d = new D;  call\_memptr(d);  delete d;  } |

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

| **Principles(s):** Heed Compiler Warnings, Use Effective Quality Assurance Techniques, Adopt a Secure Coding Standard.  Heeding compiler warnings related to invalid member access can help identify potential issues with pointer-to-member operators during the development process. Using effective quality assurance techniques, such as thorough code reviews and dynamic testing, can detect attempts to access nonexistent members and prevent undefined behavior. Adopting a secure coding standard that prohibits the use of pointer-to-member operators to access nonexistent members promotes safer and more reliable object-oriented programming practices. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | **P6** | **L2** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **overflow\_upon\_dereference invalid\_function\_pointer** |  |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-OOP55** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **LANG.MEM.UVAR** | Uninitialized Variable |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Copy operations must not mutate the source object** |
| --- | --- | --- |
| Object Oriented Programming | [OOP-058-CPP] | This standard makes sure that that copy operations do not mutate the source object maintaining the expected behavior and integrity of the copied object, preventing unintended side effects and making the code more predictable and easier to reason about. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the copy operations for `A` mutate the source operand by resetting its member variable `m` to `0`. When `std::fill()` is called, the first element copied will have the original value of `obj.m, 12`, at which point `obj.m` is set to `0`. The subsequent nine copies will all retain the value `0`. |
| #include <algorithm>  #include <vector>  class A {  mutable int m;  public:  A() : m(0) {}  explicit A(int m) : m(m) {}  A(const A &other) : m(other.m) {  other.m = 0;  }  A& operator=(const A &other) {  if (&other != this) {  m = other.m;  other.m = 0;  }  return \*this;  }  int get\_m() const { return m; }  };  void f() {  std::vector<A> v{10};  A obj(12);  std::fill(v.begin(), v.end(), obj);  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the copy operations for `A `no longer mutate the source operand, ensuring that the vector contains equivalent copies of `obj`. Instead, `A` has been given move operations that perform the mutation when it is safe to do so. |
| #include <algorithm>  #include <vector>  class A {  int m;  public:  A() : m(0) {}  explicit A(int m) : m(m) {}  A(const A &other) : m(other.m) {}  A(A &&other) : m(other.m) { other.m = 0; }  A& operator=(const A &other) {  if (&other != this) {  m = other.m;  }  return \*this;  }  A& operator=(A &&other) {  m = other.m;  other.m = 0;  return \*this;  }  int get\_m() const { return m; }  };  void f() {  std::vector<A> v{10};  A obj(12);  std::fill(v.begin(), v.end(), obj);  } |

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

| **Principles(s):** Heed Compiler Warnings, Keep It Simple, Adopt a Secure Coding Standard.  Heeding compiler warnings related to potential mutations of the source object during copy operations can help identify and prevent unintended side effects. Keeping the copy operations simple and ensuring that they do not modify the source object makes the code more predictable, maintainable, and less prone to errors. Adopting a secure coding standard that mandates copy operations to not mutate the source object promotes consistent and reliable object-oriented programming practices, reducing the risk of introducing vulnerabilities or unexpected behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | **P9** | **L2** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | **LANG.FUNCS.COPINC** | Copy Operation Parameter Is Not const |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.1 | **C++4075** |  |
| [Klocwork](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Klocwork) | 2024.1 | **CERT.OOP.COPY\_MUTATES** |  |

### 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.

To implement changes for transitioning a DevOps process into a DevSecOps process, one would typically need to configure one’s CI/CD pipelines with the relevant security tools. This might involve writing custom scripts or using plugins for one’s given CI/CD tooling (e.g., Jenkins, GitLab CI, CircleCI). For each stage of the DevSecOps cycle, one would select tools that are relevant to the kind of security assurance that one is seeking - SAST, DAST, IAST, RASP, dependency checking, configuration management, etc.

Enforcing these checks as part of the DevOps process will require that the tools for static analysis, like SonarQube, and for runtime analysis, like a SIEM tool, are well-integrated into one’s build pipelines in a CI/CD system. Code repositories must reject commits that do not pass the security checks, and one’s deployment processes must be capable of rolling back automatically if a security issue is detected post-deployment.

### 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 |
| --- | --- | --- | --- | --- | --- |
| STR-050-CPP | High | Likely | Medium | **P18** | **L1** |
| INT-050-CPP | Medium | Unlikely | Medium | **P4** | **L3** |
| STR-051-CPP | High | Likely | Medium | **P18** | **L1** |
| IDS-000-J | High | Likely | Medium | **P18** | **L1** |
| MEM-052-CPP | High | Likely | Medium | **P18** | **L1** |
| DCL-055-CPP | Low | Unlikely | High | **P1** | **L3** |
| ERR-058-CPP | Low | Likely | Low | **P9** | **L2** |
| MSC-051-CPP | Medium | Likely | Low | **P18** | **L1** |
| OOP-055-CPP | High | Probable | High | **P6** | **L2** |
| OOP-058-CPP | Low | Likely | Low | **P9** | **L2** |

### 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 refers to the encryption of data that is stored on a device or system, such as files on a hard drive or data in a database. This type of encryption protects data from unauthorized access if the storage device is lost, stolen, or compromised.  Adhere to the Principle of Least Privilege, Practice Defense in Depth, Adopt a Secure Coding Standard.  Adhering to the principle of least privilege ensures that only authorized users and processes can access the encrypted data at rest. Practicing defense in depth by implementing multiple layers of encryption and access controls enhances the overall security of the data at rest. Adopting a secure coding standard that mandates the use of strong encryption algorithms and proper key management practices helps protect data at rest from unauthorized access or breaches. |
| Encryption in flight | Encryption in flight, also known as encryption in transit, refers to the encryption of data as it travels between systems or devices over a network. This type of encryption protects data from being intercepted, read, or modified by unauthorized parties during transmission.  Validate Input Data, Sanitize Data Sent to Other Systems, Practice Defense in Depth.  Validating input data ensures that the data being encrypted and transmitted is free from malicious content or unauthorized modifications. Sanitizing data sent to other systems by applying appropriate encryption techniques helps protect the confidentiality and integrity of the data in transit. Practicing defense in depth by using secure communication protocols, such as TLS or SSL, and implementing additional security measures, like message authentication codes, enhances the overall security of the data in flight. |
| Encryption in use | Encryption in use, also known as encryption in memory, refers to the encryption of data while it is being processed or used by an application. This type of encryption protects data from being accessed or manipulated by unauthorized parties while it is in a system's memory.  Adhere to the Principle of Least Privilege, Practice Defense in Depth, Adopt a Secure Coding Standard.  Adhering to the principle of least privilege ensures that only authorized processes and users can access and manipulate the unencrypted data in use. Practicing defense in depth by implementing secure memory management techniques, such as secure enclaves or hardware-based encryption, helps protect the data in use from unauthorized access or tampering. Adopting a secure coding standard that emphasizes secure memory handling, side-channel attack mitigation, and proper key management practices helps safeguard the data in use and maintain its confidentiality and integrity. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identity of a user or entity before granting access to a system or resource. It ensures that the user is who they claim to be, often through user logins and credentials.  Validate Input Data, Practice Defense in Depth, Use Effective Quality Assurance Techniques.  Validating input data during the authentication process, such as user logins, helps prevent unauthorized access and ensures the integrity of the authentication mechanism. Practicing defense in depth by implementing multiple factors of authentication, such as passwords and biometric data, enhances the overall security of the authentication process. Using effective quality assurance techniques, like penetration testing and code reviews, helps identify and address vulnerabilities in the authentication system, including the addition of new users. |
| Authorization | Authorization is the process of granting or restricting access to specific resources, functionalities, or data based on the authenticated user's permissions and privileges. It determines what actions a user is allowed to perform within a system, such as accessing files or making changes to the database.  Adhere to the Principle of Least Privilege, Validate Input Data, Sanitize Data Sent to Other Systems.  Adhering to the principle of least privilege ensures that users are granted only the necessary permissions to perform their tasks, limiting the potential impact of unauthorized access or misuse. Validating input data, such as user-supplied information for accessing files or modifying the database, helps prevent unauthorized changes and maintains the integrity of the system. Sanitizing data sent to other systems, like databases or file servers, ensures that only authorized and safe operations are performed, mitigating the risk of data corruption or unauthorized modifications. |
| Accounting | Accounting, also known as auditing, is the process of recording and monitoring user activities and system events. It provides a trail of user actions, including file access, changes to the database, and the addition of new users, enabling administrators to detect and investigate potential security breaches or misuse of resources.  Practice Defense in Depth, Use Effective Quality Assurance Techniques, Adopt a Secure Coding Standard.  Practicing defense in depth by implementing multiple layers of logging and monitoring, such as application-level logging and system-level auditing, enhances the overall visibility and accountability of user activities. Using effective quality assurance techniques, like log analysis and anomaly detection, helps identify suspicious or unauthorized activities, such as unusual file access patterns or unauthorized changes to the database. Adopting a secure coding standard that emphasizes proper logging, error handling, and secure data management practices ensures that the accounting mechanism captures relevant events and maintains the integrity of the audit trail, facilitating incident response and forensic analysis.  By applying these principles to the authentication, authorization, and accounting components of the Triple-A Framework, organizations can effectively manage user access, protect sensitive resources, and maintain a secure environment. Regularly reviewing and updating user level of access, monitoring file access patterns, and auditing changes to the database are essential practices to ensure the ongoing effectiveness of the security controls in place. |

**\***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 |  |
| 1.2.5 | 03/31/2024 | Milestone 3-2 | Malik Alnakhaleh |  |
| 1.5 | 04/15/2024 | Project One | Malik Alnakhaleh |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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