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



# Green Pace Secure Development Policy

# Evelyn Pinarello

# April 13, 2023

## 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 | Assess the validity of all input data, regardless of source. This can help to mitigate security vulnerabilities. Treating all data sources as a potential vulnerability can help protect the software. |
| 1. Heed Compiler Warnings | Paying attention to messages from the compiling program regarding potential issues in the code will help to mitigate errors that may not affect the functionality of the code but could affect the security of the code. |
| 1. Architect and Design for Security Policies | Design the code with functions and systems designed with different required security policies in mind to implement the highest security available per software or user type. |
| 1. Keep It Simple | Writing the program or code as clearly and simply as possible decreases the possibility or errors and vulnerabilities within the program. |
| 1. Default Deny | Allowing access based on granted permission rather than until denied strengthens the overall security of the program by only granting access to predetermined portions of the code. |
| 1. Adhere to the Principle of Least Privilege | Functions and processes of the program should only be granted the minimum level of permissions to properly run. This limits the security vulnerabilities from potential clearances and permissions that have higher access to data than necessary. |
| 1. Sanitize Data Sent to Other Systems | Ensure the data being transmitted meets the functionality and security requirements of the system it is being sent to to avoid injection attacks. |
| 1. Practice Defense in Depth | Implementing the Defense in Depth security strategy will implement multiple complimentary layers of security to the system and program will which help to mitigate security vulnerabilities and potential attacks from multiple sources. |
| 1. Use Effective Quality Assurance Techniques | Assuring the quality of the program or code produced through various techniques will be beneficial in detecting and mitigating potential security vulnerabilities. It is also beneficial to perform reviews and audits which can identify potential risks that may have been missed during other testing methods. |
| 1. Adopt a Secure Coding Standard | Creating, implementing, reviewing, and adapting a secure standard for each code, coding languages, and platform can help to mitigate security vulnerabilities while strengthening the creation of functional and secure programs and codes. |

### 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** | Never qualify a reference type with const or volatile |
| --- | --- | --- |
| **Data Type** | [STD-001-C] | C++ does not allow you to change the value of a reference type, effectively treating all references as being const qualified. The C++ Standard, [dcl.ref], paragraph 1 [[ISO/IEC 14882-2014](https://wiki.sei.cmu.edu/confluence/display/cplusplus/AA.+Bibliography#AA.Bibliography-ISO/IEC14882-2014)], states the following:  Cv-qualified references are ill-formed except when the cv-qualifiers are introduced through the use of a typedef-name (7.1.3, 14.1) or decltype-specifier (7.1.6.2), in which case the cv-qualifiers are ignored.  Thus, C++ prohibits or ignores the [cv-qualification](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-cvqualify) of a reference type. Only a value of non-reference type may be cv-qualified. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a const-qualified reference to a char is formed instead of a reference to a const-qualified char. This results in undefined behavior. |
| #include <iostream>    void f(char c) {    char &const p = c;    p = 'p';    std::cout << c << std::endl;  } |

| **Compliant Code** |
| --- |
| This compliant solution removes the const qualifier. |
| #include <iostream>    void f(char c) {    char &p = c;    p = 'p';    std::cout << c << std::endl;  } |

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

| **Principles(s):** Using these reference types could result in undefined behavior which could cause the system to store unexpected data and values which is a security violation |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 |  | Checks for violations of this rule; produces error if found |
| Polyspace Bug Finder | R2023a | CERT C++: DCL52-CPP | Checks for const-qualified and modification of const-qualified reference types |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-DCL52-a | Reference type should never be qualified using const or volatile |

#### Coding Standard 2

| **Coding Standard** | **Label** | Implement abstract data types using opaque types |
| --- | --- | --- |
| **Data Value** | [STD-002-C] | Abstract data types are not restricted to object-oriented languages such as C++ and Java. They should be created and used in C language programs as well. Abstract data types are most effective when used with private (opaque) data types and information hiding. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example is based on the managed string library developed by CERT [[Burch 2006](https://wiki.sei.cmu.edu/confluence/display/c/AA.+Bibliography#AA.Bibliography-Burch06)]. In this example, the managed string type and the functions that operate on this type are defined in the string\_m.h header file as follows: |
| struct string\_mx {    size\_t size;    size\_t maxsize;    unsigned char strtype;    char \*cstr;  };    typedef struct string\_mx string\_mx;    /\* Function declarations \*/  extern errno\_t strcpy\_m(string\_mx \*s1, const string\_mx \*s2);  extern errno\_t strcat\_m(string\_mx \*s1, const string\_mx \*s2);  /\* ... \*/ |

| **Compliant Code** |
| --- |
| This compliant solution reimplements the string\_mx type as a private type, hiding the implementation of the data type from the user of the managed string library. To accomplish this, the developer of the private data type creates two header files: an external string\_m.h header file that is included by the user of the data type and an internal file that is included only in files that implement the managed string abstract data type.  In the external string\_m.h file, the string\_mx type is defined to be an instance of struct string\_mx, which in turn is declared as an [incomplete type](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-incompletetype): |
| struct string\_mx;  typedef struct string\_mx string\_mx;    /\* Function declarations \*/  extern errno\_t strcpy\_m(string\_mx \*s1, const string\_mx \*s2);  extern errno\_t strcat\_m(string\_mx \*s1, const string\_mx \*s2);  /\* ... \*/ |

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

| **Principles(s):** Using opaque abstract data types can reduce the number of defects within the code |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA Tool Suite | 9.7.1 | 104D | Partially implemented |
| Polyspace Bug Finder | R2023a | CERT C: Rec. DCL12-C | Checks for structure or union object implementation visible in file where pointer to this object is not dereferenced |
| Parasoft C/C++ test | 2022.2 | CERT\_C-DCL12-a | If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden |

#### Coding Standard 3

| **Coding Standard** | **Label** | Do not specify the bound of a character array initialized with a string literal |
| --- | --- | --- |
| **String Correctness** | [STD-003-C] | The C Standard allows an array variable to be declared both with a bound index and with an initialization literal. The initialization literal also implies an array size in the number of elements specified. For strings, the size specified by a string literal is the number of characters in the literal plus one for the terminating null character. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example initializes an array of characters using a string literal that defines one character more (counting the terminating '\0') than the array can hold: |
| const char s[3] = "abc"; |

| **Compliant Code** |
| --- |
| This compliant solution does not specify the bound of a character array in the array declaration. If the array bound is omitted, the compiler allocates sufficient storage to store the entire string literal, including the terminating null character. |
| const char s[] = "abc"; |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.04 |  | Can detect code defects this rule tries to prevent |
| ÉCLAIR | 1.2 | CC2.STR36 | Fully implemented |
| Parasoft C/C++ test | 2022.2 | CERT-C-STR11-a | Do not specify the bound of a character array initialized with a string literal |
| PC-lint Plus | 1.4 | 784 | Partially supported |

#### Coding Standard 4

| **Coding Standard** | **Label** | Prevent SQL injection |
| --- | --- | --- |
| **SQL Injection** | [STD-004-JAV] | SQL injection vulnerabilities arise in applications where elements of a SQL query originate from an untrusted source. Without precautions, the [untrusted data](https://wiki.sei.cmu.edu/confluence/display/java/Rule+BB.+Glossary#RuleBB.Glossary-untrusteda) may maliciously alter the query, resulting in information leaks or data modification. The primary means of preventing SQL injection are [sanitization](https://wiki.sei.cmu.edu/confluence/display/java/Rule+BB.+Glossary#RuleBB.Glossary-sa) and validation, which are typically implemented as parameterized queries and stored procedures. |

| **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](https://wiki.sei.cmu.edu/confluence/display/java/Rule+BB.+Glossary#RuleBB.Glossary-sanitize) 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):** Failure to sanitize user input before processing can lead to SQL injection vulnerabilities and attacks |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and Security Errors |
| CodeSonar | 7.3p0 | JAVA.IO.INJ.SQL | SQL injection |
| Parasoft JTest | 2022.2 | CERT.IDS00.TDSQL | Protect against SQL injection |
| SonarQube | 6.7 | S2077  S3649 | SQL queries should not be vulnerable to injection attacks |

#### Coding Standard 5

| **Coding Standard** | **Label** | Allocate sufficient memory for an object |
| --- | --- | --- |
| **Memory Protection** | [STD-005-C] | The types of integer expressions used as size arguments to malloc(), calloc(), realloc(), or aligned\_alloc() must have sufficient range to represent the size of the objects to be stored. If size arguments are incorrect or can be manipulated by an attacker, then a buffer overflow may occur. Incorrect size arguments, inadequate range checking, integer overflow, or truncation can result in the allocation of an inadequately sized buffer.  Typically, the amount of memory to allocate will be the size of the type of object to allocate. When allocating space for an array, the size of the object will be multiplied by the bounds of the array. When allocating space for a structure containing a flexible array member, the size of the array member must be added to the size of the structure. (See [MEM33-C. Allocate and copy structures containing a flexible array member dynamically](https://wiki.sei.cmu.edu/confluence/display/c/MEM33-C.++Allocate+and+copy+structures+containing+a+flexible+array+member+dynamically).) Use the correct type of the object when computing the size of memory to allocate.  [STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator](https://wiki.sei.cmu.edu/confluence/display/c/STR31-C.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator) is a specific instance of this rule. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, inadequate space is allocated for a struct tm object because the size of the pointer is being used to determine the size of the pointed-to object: |
| #include <stdlib.h>  #include <time.h>    struct tm \*make\_tm(int year, int mon, int day, int hour,                     int min, int sec) {    struct tm \*tmb;    tmb = (struct tm \*)malloc(sizeof(tmb));    if (tmb == NULL) {      return NULL;    }    \*tmb = (struct tm) {      .tm\_sec = sec, .tm\_min = min, .tm\_hour = hour,      .tm\_mday = day, .tm\_mon = mon, .tm\_year = year    };    return tmb;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the correct amount of memory is allocated for the struct tm object. When allocating  space for a single object, passing the (dereferenced) pointer type to the sizeof operator is a simple way to allocate sufficient memory. Because the sizeof operator does not evaluate its operand, dereferencing an uninitialized or null pointer in this context is well-defined behavior. |
| #include <stdlib.h>  #include <time.h>    struct tm \*make\_tm(int year, int mon, int day, int hour,                     int min, int sec) {    struct tm \*tmb;    tmb = (struct tm \*)malloc(sizeof(\*tmb));    if (tmb == NULL) {      return NULL;    }    \*tmb = (struct tm) {      .tm\_sec = sec, .tm\_min = min, .tm\_hour = hour,      .tm\_mday = day, .tm\_mon = mon, .tm\_year = year    };    return tmb;  } |

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

| **Principles(s):** Providing invalid size arguments to memory allocation functions can lead to buffer overflows and security vulnerabilities |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.04 | Malloc-size-insufficient | Besides direct violations, all defined behaviour resulting from invalid memory accesses is reported by Astree |
| Coverity | 2017.07 | SIZECHECK | Can find instances where string length is miscalculated for memory allocation purposes |
| Parasoft C/C++ test | 2022.2 | CERT\_C-MEM35-a | Do not use sizeof operator on pointer type to specify the size of the memory to be allocated via ‘malloc’, ‘calloc’, or ‘realloc’ function |
| Polyspace Bug Finder | R2023a | CERT C: Rule MEM35C | |  |  | | --- | --- | |  | Checks for: Pointer access out of bounds and memory allocation with tainted size  Rule partially covered. | |

#### Coding Standard 6

| **Coding Standard** | **Label** | Use a static assertion to test the value of a constant expression |
| --- | --- | --- |
| **Assertions** | [STD-006-C] | Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in [vulnerabilities](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-vulnerability) (see [MSC11-C. Incorporate diagnostic tests using assertions](https://wiki.sei.cmu.edu/confluence/display/c/MSC11-C.+Incorporate+diagnostic+tests+using+assertions)). The runtime assert() macro has some limitations, however, in that it incurs a runtime overhead and because it calls abort(). Consequently, the runtime assert() macro is useful only for identifying incorrect assumptions and not for runtime error checking. As a result, runtime assertions are generally unsuitable for server programs or embedded systems. |

| **Noncompliant Code** |
| --- |
| This noncompliant code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly: |
| #include <assert.h>    struct timer {    unsigned char MODE;    unsigned int DATA;    unsigned int COUNT;  };    int func(void) {    assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| For assertions involving only constant expressions, a preprocessor conditional statement may be used, as in this compliant solution: |
| struct timer {    unsigned char MODE;    unsigned int DATA;    unsigned int COUNT;  };    #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))    #error "Structure must not have any padding"  #endif |

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

| **Principles(s):** Static assertion is a valuable diagnostic tool for finding and eliminating software defects that may result in [vulnerabilities](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-vulnerability) at compile time. The absence of static assertions, however, does not mean that code is incorrect. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| CodeSonar | 7.3p0 | Customization | Users can implement a custom check that reports uses of assert() macro |
| ÉCLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | Handle all exceptions |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | When an exception is thrown, control is transferred to the nearest handler with a type that matches the type of the exception thrown. If no matching handler is directly found within the handlers for a try block in which the exception is thrown, the search for a matching handler continues to dynamically search for handlers in the surrounding try blocks of the same thread. The C++ Standard, [except.handle], paragraph 9 [[ISO/IEC 14882-2014](https://wiki.sei.cmu.edu/confluence/display/cplusplus/AA.+Bibliography#AA.Bibliography-ISO/IEC14882-2014)], states the following:  If no matching handler is found, the function std::terminate() is called; whether or not the stack is unwound before this call to std::terminate() is implementation-defined. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    try {      f();    } catch (...) {      // Handle error    }  } |

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

| **Principles(s):** Allowing the application to abnormally terminate can lead to resources not being freed, closed, and so on. It is frequently a vector for denial-of-service attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Main-function-catch-all-early-catch-all | Partially checked |
| CodeSonar | 7.3p0 | LANG.STRUCT.UCTCH | Unreachable catch |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-ERR51-a | Always catch exceptions |
| PolySpace Bug Finder | R2023a | CERT C++: ERR51-CPP | Checks for unhandled exceptions (partially covered) |

#### Coding Standard 8

| **Coding Standard** | **Label** | Close files when they are no longer needed |
| --- | --- | --- |
| Files | [STD-008-CPP] | A call to the std::basic\_filebuf<T>::open() function must be matched with a call to std::basic\_filebuf<T>::close() before the lifetime of the last pointer that stores the return value of the call has ended or before normal program termination, whichever occurs first.  Note that std::basic\_ifstream<T>, std::basic\_ofstream<T>, and std::basic\_fstream<T> all maintain an internal reference to a std::basic\_filebuf<T> object on which open() and close() are called as needed. Properly managing an object of one of these types (by not leaking the object) is sufficient to ensure compliance with this rule. Often, the best solution is to use the stream object by value semantics instead of via dynamic memory allocation, ensuring compliance with [MEM51-CPP. Properly deallocate dynamically allocated resources](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM51-CPP.+Properly+deallocate+dynamically+allocated+resources). However, that is still insufficient for situations in which destructors are not automatically called. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a std::fstream object file is constructed. The constructor for std::fstream calls std::basic\_filebuf<T>::open(), and the default std::terminate\_handler called by std::terminate() is std::abort(), which does not call destructors. Consequently, the underlying std::basic\_filebuf<T> object maintained by the object is not properly closed. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {    std::fstream file(fileName);    if (!file.is\_open()) {      // Handle error      return;    }    // ...    std::terminate();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, std::fstream::close() is called before std::terminate() is called, ensuring that the file resources are properly closed. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {    std::fstream file(fileName);    if (!file.is\_open()) {      // Handle error      return;    }    // ...    file.close();    if (file.fail()) {      // Handle error    }    std::terminate();  } |

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

| **Principles(s):** Failing to properly close files may allow an attacker to exhaust system resources and can increase the risk that data written into in-memory file buffers will not be flushed in the event of abnormal program termination. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | ALLOC.LEAK | Leak |
| Coverity | 2017.07 | RESOURCE\_LEAK | Partially implemented |
| Parasoft C/C++ test | 2022.2 | CERT\_C-FIO42-a | Ensure resources are freed |
| Polyspace Bug Finder | R2023a | CERT C: Ruke FIO42-C | Checks for resource leak (partially covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | Detect and handle memory allocation errors |
| --- | --- | --- |
| Memory Allocation | [STD-009-CPP] | The default memory allocation operator, ::operator new(std::size\_t), throws a std::bad\_alloc exception if the allocation fails. Therefore, you need not check whether calling ::operator new(std::size\_t) results in nullptr. The nonthrowing form, ::operator new(std::size\_t, const std::nothrow\_t &), does not throw an exception if the allocation fails but instead returns nullptr. The same behaviors apply for the operator new[] versions of both allocation functions. Additionally, the default allocator object (std::allocator) uses ::operator new(std::size\_t) to perform allocations and should be treated similarly. |

| **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](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-abnormaltermination) 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):** Failing to detect allocation failures can lead to abnormal program termination and denial-of-service attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled |
| LDRA tool suite | 9.7.1 | 45 D | Partially implemented |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-MEM52-a | Check the return value of new |
| Polyspace Bug Finder | R2023a | CERT C++L MEM52-CPP | Checks for unprotected dynamic memory allocation (partially covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | Do not write syntactically ambiguous declarations |
| --- | --- | --- |
| Declarations | [STD-010-CPP] | It is possible to devise syntax that can ambiguously be interpreted as either an expression statement or a declaration. Syntax of this sort is called a vexing parse because the compiler must use disambiguation rules to determine the semantic results. The C++ Standard, [stmt.ambig], paragraph 1 [[ISO/IEC 14882-2014](https://wiki.sei.cmu.edu/confluence/display/cplusplus/AA.+Bibliography#AA.Bibliography-ISO/IEC14882-2014)], in part, states the following:  There is an ambiguity in the grammar involving expression-statements and declarations: An expression-statement with a function-style explicit type conversion as its leftmost subexpression can be indistinguishable from a declaration where the first declarator starts with a (. In those cases the statement is a declaration. [Note: To disambiguate, the whole statement might have to be examined to determine if it is an expression-statement or a declaration. ...  A similarly vexing parse exists within the context of a declaration where syntax can be ambiguously interpreted as either a function declaration or a declaration with a function-style cast as the initializer. The C++ Standard, [dcl.ambig.res], paragraph 1, in part, states the following:  The ambiguity arising from the similarity between a function-style cast and a declaration mentioned in 6.8 can also occur in the context of a declaration. In that context, the choice is between a function declaration with a redundant set of parentheses around a parameter name and an object declaration with a function-style cast as the initializer. Just as for the ambiguities mentioned in 6.8, the resolution is to consider any construct that could possibly be a declaration a declaration.  Do not write a syntactically ambiguous declaration. With the advent of uniform initialization syntax using a braced-init-list, there is now syntax that unambiguously specifies a declaration instead of an expression statement. Declarations can also be disambiguated by using nonfunction-style casts, by initializing using =, or by removing extraneous parenthesis around the parameter name. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an anonymous local variable of type std::unique\_lock is expected to lock and unlock the mutex m by virtue of [RAII.](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-RAII) However, the declaration is syntactically ambiguous as it can be interpreted as declaring an anonymous object and calling its single-argument converting constructor or interpreted as declaring an object named m and default constructing it. The syntax used in this example defines the latter instead of the former, and so the mutex object is never locked. |
| #include <mutex>    static std::mutex m;  static int shared\_resource;    void increment\_by\_42() {    std::unique\_lock<std::mutex>(m);    shared\_resource += 42;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the lock object is given an identifier (other than m) and the proper converting constructor is called. |
| #include <mutex>    static std::mutex m;  static int shared\_resource;    void increment\_by\_42() {    std::unique\_lock<std::mutex> lock(m);    shared\_resource += 42;  } |

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

| **Principles(s):** Syntactically ambiguous declarations can lead to unexpected program execution. However, it is likely that rudimentary testing would uncover violations of this rule. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | LANG.STRUCT.DECL.FNEST | Nested Function Declaration |
| LDRA tool suite | 9.7.1 | 296 S | Partially implemented |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-DCL53- | Parameter names in function declarations should not be enclosed in parentheses |
| Polyspace Bug Finder | R2023a | CERT C++: DCL53-CPP | Checks for declarations that can be confused between function and object declaration |

### 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.

Implementing automation tools in the Design and Build steps of pre-production will help ensure any bugs or vulnerabilities will be flagged throughout the entire writing and testing process as opposed to trying to insert security measures upon completion of the program. Using automation tools from the beginning helps to mitigate risk while also adhering to the secure coding standards. Automation tools should also be used throughout the entire production process in DevSecOps to ensure the program is implementing secure code methods, mitigating vulnerabilities, and acknowledging any vulnerabilities and threats as they appear.

### 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 | Low | Unlikely | Low | P3 | L3 |
| STD-002-C | Low | Unlikely | High | P1 | L3 |
| STD-003-C | Low | Probable | Low | P6 | L2 |
| STD-004-JAV | High | Probable | Medium | P12 | L1 |
| STD-005-C | High | Probable | High | P6 | L2 |
| STD-006-C | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-009-CPP | High | Likely | Medium | P18 | L1 |
| STD-010-CPP | Low | Unlikely | Medium | P2 | L3 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | This refers to encrypting data that is not currently in use but is still stored somewhere within the system, such as database or server. This helps to mitigate risks such as security breaches and the malicious acquisition of data. |
| Encryption at flight | This refers to encrypting data that is in the process or moving in to or out of the system or program. This data must be encrypted because it mitigates the potential threat of malicious data being input into the program, system, or database. |
| Encryption in use | This refers to encrypting data that is currently in use by the program, system, or user. Implementing the Triple A framework will help to encrypt the data in use. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | This is the portion of the framework that verifies the identity of the user trying to access the data. This can include usernames, ID numbers, password, or secondary authentication such as secure codes sent to an associated email or phone number. |
| Authorization | This is the portion of the framework that verifies the user has the specified level of clearance to access certain data levels. This helps mitigate users having access to viewing and modifying data they do not require. This limits the number of users who have access to sensitive data. |
| Accounting | This is the portion of the framework that tracks the activity of the users which provides a level of security by holding users accountable for changes made from their activity. This also provides a map of activities to be reviewed and used in the event of security breaches. |

**\***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 |  |
| 2.0 | 03/21/2023 | Milestone One Updates | Evelyn Pinarello |  |
| 3.0 | 04/13/2023 | Project One Updates | Evelyn Pinarello | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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