

# 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 6](#_Toc52464060)

[Coding Standard 3 9](#_Toc52464061)

[Coding Standard 4 12](#_Toc52464062)

[Coding Standard 5 17](#_Toc52464063)

[Coding Standard 6 21](#_Toc52464064)

[Coding Standard 7 24](#_Toc52464065)

[Coding Standard 8 27](#_Toc52464066)

[Coding Standard 9 38](#_Toc52464067)

[Coding Standard 10 41](#_Toc52464068)

[Defense-in-Depth Illustration 47](#_Toc52464069)

[Project One 47](#_Toc52464070)

[1. Revise the C/C++ Standards 47](#_Toc52464071)

[2. Risk Assessment 47](#_Toc52464072)

[3. Automated Detection 47](#_Toc52464073)

[4. Automation 47](#_Toc52464074)

[5. Summary of Risk Assessments 48](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 48](#_Toc52464076)

[7. Map the Principles 49](#_Toc52464077)

[Audit Controls and Management 50](#_Toc52464078)

[Enforcement 50](#_Toc52464079)

[Exceptions Process 50](#_Toc52464080)

[Distribution 51](#_Toc52464081)

[Policy Change Control 51](#_Toc52464082)

[Policy Version History 51](#_Toc52464083)

[Appendix A Lookups 51](#_Toc52464084)

[Approved C/C++ Language Acronyms 51](#_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 | Systems must validate all input that comes from untrusted data sources. These external data sources include command line arguments, network interfaces, environmental variables, and user-controlled files. Validating all input correctly eliminates a great majority of vulnerabilities within the software. |
| 1. Heed Compiler Warnings | The code should be compiled with the highest warning level available for the compiler being used. Warnings should also be eliminated by modifying the code. Furthermore, use analysis tools (static and dynamic) to find and fix any additional security flaws. |
| 1. Architect and Design for Security Policies | When creating a software, build an architecture and design that will implement and enforce security policies. An example of this principle is allowing different system privileges at different times. To build this system, it should get divided into subsystems that have their own privilege set. |
| 1. Keep It Simple | The system design should be as simple as possible. If the design is too complex, it increases the likelihood of errors being made during implementation, configuration, and use. Also, achieving an appropriate level of assurance would require a lot of effort for complex security mechanisms. |
| 1. Default Deny | Access decisions should be based on permission, not exclusion. When based on permission, access is denied and would identify conditions that would grant access. |
| 1. Adhere to the Principle of Least Privilege | The least set of privileges necessary to complete a task should get executed during every process. Furthermore, any permission needed and accessed should get utilized for the minimal time required for that task. Following this principle reduces an attacker’s opportunities to execute arbitrary code with elevated privileges. |
| 1. Sanitize Data Sent to Other Systems | All data passed to complex subsystems should get sanitized. These subsystems include command shells, relational databases, and commercial off-the-shelf components. Attackers could run an injection attack through the unused functionality in these components, such as SQL, command, etc. |
| 1. Practice Defense in Depth | Risk should be managed using multiple defense strategies. DiD provides another layer of defense to prevent a security flaw if the first layer proves inadequate. Also, it can prevent any exploitable vulnerabilities and/or limit the consequences of an attack. |
| 1. Use Effective Quality Assurance Techniques | Practicing good quality assurance techniques can help identify and eliminate any system vulnerabilities. Some QA practices to include in the quality assurance program are fuzz testing, penetration testing, and source code audits. |
| 1. Adopt a Secure Coding Standard | It is essential to apply and develop using the secure coding standards for the language and platform being used to build the system. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-GLC  (DCL12- C) | Implement abstract data types using opaque types  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** |
| --- |
| In this example, the managed string type and the functions that operate on this type are defined in the string\_m.h header file (example below). The implementation of the string\_mx type is fully visible to the user of the data type after including the string\_m.h file. Programmers are consequently more likely to directly manipulate the fields withing the structure, violating the software engineering principles of information hiding and data encapsulating and increasing the probability of developing incorrect or nonportable code. |
| **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 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. Also, In the internal header file, struct string\_mx is fully defined but not visible to a user of the data abstraction. |
| **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);  /\* ... \*/  **struct** string\_mx {  **size\_t** size;  **size\_t** maxsize;    unsigned **char** strtype;  **char** \*cstr;  }; |

| **Principles(s):**  Architect and Design for Security Policies (3): In the noncompliant example, the struct is fully visible to the user making it insecure. It allows programmers to directly manipulate the data because it is not encapsulated. So, it is important to design the code with compliant and secure information hiding.  Adopt a Secure Coding Standard (10): Adopting a secure coding standard, especially always encapsulating sensitive data, is very important. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | Low (P1) | Low (3) |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL12 | N/A |
| LDRA tool suite | 9.7.1 | 104 D | Partially implemented |
| Polyspace Bug Finder | R2022a | CERT C: Rec. DCL12-C | Checks for structure or union object implementation visible in file where pointer to this object is not dereferenced (rule partially covered) |
| Parasoft C/C++ test | 2022.1 | CERT\_C-DCL12-a | If a pointer to a structure or union is never dereferenced withing a translation unit, then the implementation of the object should be hidden |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP  (INT50-CPP) | Do not cast to an out-of-range enumeration value  Enumerations in C++ come in two forms: scoped enumerations in which the underlying type is fixed and unscoped enumerations in which the underlying type may or may not be fixed. The range of values that can be represented by either form of enumeration may include enumerator values not specified by the enumeration itself. To avoid operating on unspecified values, the arithmetic value being cast must be within the range of values the enumeration can represent. When dynamically checking for out-of-range values, checking must be performed before the cast expression. |

| **Noncompliant Code** |
| --- |
| Bounds Checking – This 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. |
| **enum** EnumType {    First,    Second,    Third  };    **void** f(**int** intVar) {    EnumType enumVar = **static\_cast**<EnumType>(intVar);    **if** (enumVar < First || enumVar > Third) {      // Handle error    }  } |

| **Compliant Code** |
| --- |
| Bounds Checking - 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);  } |
| Scoped Enumeration - This compliant solution uses a scoped enumeration, which has a fixed underlying int type by default, allowing any value from the parameter to be converted into a valid enumeration value. It does not restrict the converted value to one for which there is a specific enumerator value, but it could do so as shown in the previous compliant solution. |
| **enum** **class** EnumType {    First,    Second,    Third  };    **void** f(**int** intVar) {    EnumType enumVar = **static\_cast**<EnumType>(intVar);  } |
| Fixed Unscoped Enumeration - Like the Scoped Enumeration solution, this solution uses an unscoped enumeration but provides a fixed underlying int type allowing any value from the parameter to be converted to a valid enumeration value. |
| **enum** EnumType : **int** {    First,    Second,    Third  };    **void** f(**int** intVar) {    EnumType enumVar = **static\_cast**<EnumType>(intVar);  } |

| **Principles(s):**  ValidateInput Data (1): Validate that the data input for the values are in-range before casting the enumeration type. Failure to do so could result in unspecified behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | Low (P4) | Low (3) |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-INT50 | N/A |
| CodeSonar | 7.0p0 | LANG.CAST.COERCE  LANG.CAST.VALUE | Coercion Alters Value  Cast Alters Value |
| Helix QAC | 2022.2 | C++3013 | N/A |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration. |
| PRQA QA-C++ | 4.4 | 3013 | N/A |
| PVS-Studio | 7.19 | V1016 | N/A |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP  (STR52-CPP) | Use valid references, pointers, and iterators to reference elements of a basic\_string  std::basic\_string is a container of characters. As a container, it supports iterators just like other containers in the Standard Template Library. However, the std::basic\_string template class has unusual invalidation semantics.  References, pointers, and iterators referring to the elements of a basic\_string sequence may be invalidated by the following uses of that basic\_string object:   * As an argument to any standard library function taking a reference to non-const basic\_string as an argument. * Calling non-const member functions, except operator[], at, front, back, begin, rbegin, end, and rend.   Examples of standard library functions taking a reference to non-const std::basic\_string are std::swap(), ::operator>>(basic\_istream &, string &), and std::getline().  Do not use an invalidated reference, pointer, or iterator because doing so results in unidentified behavior. |

| **Noncompliant Code** |
| --- |
| This code example copies input into a std::string, replacing semicolon characters with spaces. This example is noncompliant because the iterator loc is invalidated after the first call to insert(). The behavior of subsequent calls to insert() is undefined. |
| #include <string>    **void** f(**const** std::string &input) {    std::string email;      // Copy input into email converting ";" to " "    std::string::iterator loc = email.begin();  **for** (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {      email.insert(loc, \*i != ';' ? \*i : ' ');    }  } |
| In this example, data is invalidated after the call to replace(), and so its use in g() is undefined behavior. |
| #include <iostream>  #include <string>    **extern** **void** g(**const** **char** \*);    **void** f(std::string &exampleString) {  **const** **char** \*data = exampleString.data();    // ...    exampleString.replace(0, 2, "bb");    // ...    g(data);  } |

| **Compliant Code** |
| --- |
| std::string::insert() - In this compliant solution, the value of the iterator loc is updated as a result of each call to insert() so that the invalidated iterator is never accessed. The updated iterator is then incremented at the end of the loop. |
| #include <string>    **void** f(**const** std::string &input) {    std::string email;      // Copy input into email converting ";" to " "    std::string::iterator loc = email.begin();  **for** (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {      loc = email.insert(loc, \*i != ';' ? \*i : ' ');    }  } |
| std::replace() - This compliant solution uses a standard algorithm to perform the replacement. When possible, using a generic algorithm is preferable to inventing your own solution. |
| #include <algorithm>  #include <string>    **void** f(**const** std::string &input) {    std::string email{input};    std::replace(email.begin(), email.end(), ';', ' ');  }  In this compliant solution, the pointer to exampleString's internal buffer is not generated until after the modification from replace() has completed. |
| #include <iostream>  #include <string>    **extern** **void** g(**const** **char** \*);    **void** f(std::string &exampleString) {    // ...    exampleString.replace(0, 2, "bb");    // ...    g(exampleString.data());  } |

| **Principles(s):**  Architect and Design for Security Policies (3): Make sure that string iterators and pointers are validated. Don’t create them if they won’t get used properly because it can lead to undefined behavior and possible data leakage.  Heed Compiler Warnings (2): When using a high-level compiler and analysis tool, this vulnerability should show. Do not ignore it.  Adopt a Secure Coding Standard (10): Adopting a coding standard to uniformly use insert() or replace() when referencing and changing elements of a string will help with running into this vulnerability. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | Medium (P6) | Medium (2) |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.0p0 | ALLOC.UAF | Use After Free |
| Helix QAC | 2022.2 | C++4746, C++4747, C++4748, C++4749 | N/A |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-STR52-a | Use valid references, pointers, and iterators to reference elements of a basic\_string. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-JAV  (IDS00-J) | Prevent SQL injection  SQL injection vulnerabilities arise in applications where elements of a SQL query originate from an untrusted source. Without precautions, the untrusted data may maliciously alter the query, resulting in information leaks or data modification. The primary means of preventing SQL injection are sanitization 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. |
| **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        }      }    }  } |
| PreparedStatement - The JDBC library provides an API for building SQL commands that sanitize untrusted data. The java.sql.PreparedStatement class properly escapes input strings, preventing SQL injection when used correctly. This code example modifies the doPrivilegedAction() method to use a PreparedStatement instead of java.sql.Statement. However, the prepared statement still permits a SQL injection attack by incorporating the unsanitized input argument username into the prepared statement. |
| **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;        PreparedStatement stmt = connection.prepareStatement(sqlString);          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        }      }    }  } |

| **Compliant Code** |
| --- |
| PreparedStatement - 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 inputting a long username. |
| **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      }    }  } |

| **Principles(s):**  ValidateInput Data (1): Validate all inputs to make sure they do not contain any malicious characters.  Heed Compiler Warnings (2): Use an analysis tool to make sure there are no query warnings that could cause injection.  Architect and Design for Security Policies (3): Design the program to prevent and stop SQL injection attacks.  Default Deny (5): Deny access to data if password is not correct and/or other conditions are not met.  Sanitize Data Sent to Other Systems (7): Always sanitize and clear user input, especially passwords.  Practice Defense in Depth (8): Using DiD will keep sensitive data safe in case of a successful SQL injection attack.  Use Effective Quality Assurance Techniques (9): Test the code early and through out development to make sure the code is properly secured. Then, make sure developers are knowledgeable in preventing attacks.  Adopt a Secure Coding Standard (10): Make it a standard to always program the code to scan for malicious user input, denying the attacker access to data. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | High (P12) | High (1) |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and security errors |
| CodeSonar | 7.0p0 | JAVA.IO.INJ.SQL | SQL Injection (Java) |
| Coverity | 7.5 | SQLI  FB.SQL\_PREPARED\_STATEMENT\_GENERATED\_  FB.SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Findbugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Fortify | 1.0 | HTTP\_Response\_Splitting  SQL\_Injection\_\_Persistence  SQL\_Injection | Implemented |
| Klocwork | N/A | SV.DATA.BOUND , SV.DATA.DB , SV.HTTP\_SPLIT  SV.PATH , SV.PATH.INJ , SV.SQL | Implemented |
| Parasoft Jtest | 2022.1 | CERT.IDS00.TDSQL | Protect against SQL injection |
| SonarQube | 6.7 | S2077  S3649 | Executing SQL queries is security-sensitive  SQL queries should not be vulnerable to injection attacks |
| SpotBugs | 4.6.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE  SQL\_PREPARED\_STATEMENT\_GENERATED\_FROM\_NONCONSTANT\_STRING | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP  (MEM52-CPP) | Detect and handle memory allocation errors  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. Furthermore, operator new[] can throw an error of type std::bad\_array\_new\_length, a subclass of std::bad\_alloc, if the size argument passed to new is negative or excessively large.  When using the nonthrowing form, it is imperative to check that the return value is not nullptr before accessing the resulting pointer. |

| **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;  } |
| In this noncompliant code example, two memory allocations are performed within the same expression. Because the memory allocations are passed as arguments to a function call, an exception thrown as a result of one of the calls to new could result in a memory leak. |
| **struct** A { /\* ... \*/ };  **struct** B { /\* ... \*/ };    **void** g(A \*, B \*);  **void** f() {    g(**new** A, **new** B);  } |

| **Compliant Code** |
| --- |
| std::nothrow - When using std::nothrow, the new operator returns either a null pointer or a pointer to the allocated space. |
| #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;  } |
| std::bad\_alloc - can use ::operator new[] without std::nothrow and instead catch a std::bad\_alloc exception if sufficient memory cannot be allocated. |
| #include <cstring>  #include <new>    **void** f(**const** **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy;  **try** {      copy = **new** **int**[size];    } **catch**(std::bad\_alloc) {      // Handle error  **return**;    }    // At this point, copy has been initialized to allocated memory    std::**memcpy**(copy, array, size \* **sizeof**(\*copy));    // ...  **delete** [] copy;  } |
| noexcept(false)- If the design of the function is such that the caller is expected to handle exceptional situations, it is permissible to mark the function explicitly as one that may throw, as in this compliant solution. Marking the function is not strictly required, as any function without a noexcept specifier is presumed to allow throwing. |
| #include <cstring>    **void** f(**const** **int** \*array, std::**size\_t** size) noexcept(**false**) {  **int** \*copy = **new** **int**[size];    // If the allocation fails, it will throw an exception which the caller    // will have to handle.    std::**memcpy**(copy, array, size \* **sizeof**(\*copy));    // ...  **delete** [] copy;  } |
| std::unique\_ptr - In this compliant solution, a std::unique\_ptr is used to manage the resources for the A and B objects with **Resource Acquisition Is Initialization**. |
| #include <memory>    **struct** A { /\* ... \*/ };  **struct** B { /\* ... \*/ };    **void** g(std::unique\_ptr<A> a, std::unique\_ptr<B> b);  **void** f() {    g(std::make\_unique<A>(), std::make\_unique<B>());  } |
| References - When possible, the more resilient compliant solution is to remove the memory allocation entirely and pass the objects by reference instead. |
| **struct** A { /\* ... \*/ };  **struct** B { /\* ... \*/ };    **void** g(A &a, B &b);  **void** f() {    A a;    B b;    g(a, b);  } |

| **Principles(s):**  Heed Compiler Warnings (2): Utilizing a good analysis tool, memory allocation error handling issues can be caught early on.  Architect and Design for Security Policies (3): Always design a program to detect and handle memory allocation errors.  Adopt a Secure Coding Standard (10): Developers should have adequate knowledge on error handling and when to use it. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High (P18) | High (1) |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.19 | V522 , V668 | N/A |
| Klocwork | 2022.2 | NPD.CHECK.CALL.MIGHT , NPD.CHECK.CALL.MUST  NPD.CHECK.MIGHT , NPD.CHECK.MUST  NPD.CONST.CALL , NPD.CONST.DEREF  NPD.FUNC.CALL.MIGHT , NPD.FUNC.CALL.MUST  NPD.FUNC.MIGHT , PND.FUNC.MUST  NPD.GEN.CALL.MIGHT , NPD.GEN.CALL.MUST  NPD.GEN.MIGHT , NPD.GEN.MUST  RNPD.CALL , RNPD.DEREF | N/A |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-MEM52-a  CERT\_CPP-MEM52-b | Check the return value of new  Do not allocate resources in function argument list because the order of evaluation of a function’s parameters is undefined |
| Polyspace Bug Finder | R2022a | CERT C++: MEM52-CPP | Checks for unprotected dynamic memory allocation (rule partially covered) |
| Helix QAC | 2022.2 | C++ 3225 , C++ 3226 , C++ 3227 , C++ 3228  C++ 3229 , C++ 4632 | N/A |
| PRQA QA-C++ | 4.4 | 3225 , 3226 , 3227 , 3228 , 3229 , 4632 | N/A |
| LDRA tool suite | 9.7.1 | 45 D | Partially implemented |
| Compass/ROSE | N/A | N/A | N/A |
| Parasoft Insure++ | N/A | N/A | Runtime detection |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CLG  (DCL03-C) | Use a static assertion to test the value of a constant expression  Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities. 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. Using #error directives allows for clear diagnostic messages. Since this approach evaluates assertions at compile time, there is no runtime penalty. |
| **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 |
| static\_assert - Static assertions allow incorrect assumptions to be diagnosed at compile time instead of resulting in a silent malfunction or runtime error. Because the assertion is performed at compile time, no runtime cost in space or time is incurred. |
| #include <assert.h>    **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    static\_assert(**sizeof**(**struct** timer) == **sizeof**(unsigned **char**) + **sizeof**(unsigned **int**) + **sizeof**(unsigned **int**),                "Structure must not have any padding") |

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

| **Principles(s):**  Architect and Design for Security Policies (3): It’s a good habit to design the code to test constant values. Implementing this within the program saves time to catch errors at compile time. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | Low (P1) | Low (3) |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL03 | N/A |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| CodeSonar | 7.0p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| Compass/ROSE | N/A | N/A | Could detect violations of this rule merely by looking for calls to assert(), and if it can evaluate the assertion, then the code should use static-assert instead. |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP  (ERR51-CPP) | Handle all exceptions  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. All exceptions thrown by an application must be caught by a matching exception handler. Even if the exception cannot be gracefully recovered from, using the matching exception handler ensures that the stack will be properly unwound and provides an opportunity to gracefully manage external resources before terminating the process. |

| **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();  } |
| In this noncompliant code example, the thread entry point function thread\_start() does not catch exceptions thrown by throwing\_func(). If the initial thread function exits because an exception is thrown, std::terminate() is called. |
| #include <thread>    **void** throwing\_func() noexcept(**false**);    **void** thread\_start() {    throwing\_func();  }    **void** f() {    std::**thread** t(thread\_start);    t.join();  } |

| **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    }  } |
| In this compliant solution, the thread\_start() handles all exceptions and does not rethrow, allowing the thread to terminate normally. |
| #include <thread>    **void** throwing\_func() noexcept(**false**);    **void** thread\_start(**void**) {  **try** {      throwing\_func();    } **catch** (...) {      // Handle error    }  }    **void** f() {    std::**thread** t(thread\_start);    t.join();  } |

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

| **Principles(s):**  Architect and Design for Security Policies (3): It is always important and a basic secure design policy to handle exceptions to avoid unwanted behavior and data leakage.  Adopt a Secure Coding Standard (10): Developers should learn to always handle exceptions |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Medium | Low (P4) | Low (3) |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | main-function-catch-all  early-catch-all | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR51 | N/A |
| CodeSonar | 7.0p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| Helix QAC | 2022.2 | C++4035 , C++4036 , C++4037 | N/A |
| Klocwork | 2022.2 | MISRA.CATCH.ALL | N/A |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions  Each exception explicitly thrown in the code shall have a dandler of a compatible type in all call paths that could lead to that point |
| Polyspace Bug Finder | R2022a | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| PRQA QA-C++ | 4.4 | 4035 , 4036 , 4037 | N/A |
| RuleChecker | 20.10 | main-function-catch-all  early-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Appropriate Storage/String Correctness** | STD-008-CLG  (STR31-C) | Guarantee that storage for strings has sufficient space for character data and the null terminator  Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. Buffer overflows occur frequently when manipulating strings. To prevent such errors, either limit copies through truncation or, preferably, ensure that the destination is of sufficient size to hold the character data to be copied and the null-termination character. |

| **Noncompliant Code** |
| --- |
| Off-by-One Error - This noncompliant code example demonstrates an off-by-one error. The loop copies data from src to dest. However, because the loop does not account for the null-termination character, it may be incorrectly written 1 byte past the end of dest. |
| #include <stddef.h>    **void** copy(**size\_t** n, **char** src[n], **char** dest[n]) {  **size\_t** i;    **for** (i = 0; src[i] && (i < n); ++i) {       dest[i] = src[i];     }     dest[i] = '\0';  } |
| gets() - The gets() function is inherently unsafe and should never be used because it provides no way to control how much data is read into a buffer from stdin. This noncompliant code example assumes that gets() will not read more than BUFFER\_SIZE - 1 characters from stdin, which can result in a buffer overflow. |
| #include <stdio.h>  #define BUFFER\_SIZE 1024    **void** func(**void**) {  **char** buf[BUFFER\_SIZE];  **if** (**gets**(buf) == NULL) {      /\* Handle error \*/    }  } |
| getchar() - This noncompliant code example uses the getchar() function to read one character at a time from stdin instead of reading the entire line at once. The stdin stream is read until end-of-file is encountered or a newline character is read. Any newline character is discarded, and a null character is written immediately after the last character read into the array, which can result in a buffer overflow. |
| #include <stdio.h>    **enum** { BUFFERSIZE = 32 };    **void** func(**void**) {  **char** buf[BUFFERSIZE];  **char** \*p;  **int** ch;    p = buf;  **while** ((ch = **getchar**()) != '\n' && ch != EOF) {      \*p++ = (**char**)ch;    }    \*p++ = 0;  **if** (ch == EOF) {        /\* Handle EOF or error \*/    }  } |
| fscanf() - In this noncompliant example, the call to fscanf() can result in a write outside the character array buf. |
| #include <stdio.h>    **enum** { BUF\_LENGTH = 1024 };    **void** get\_data(**void**) {  **char** buf[BUF\_LENGTH];  **if** (1 != **fscanf**(stdin, "%s", buf)) {      /\* Handle error \*/    }      /\* Rest of function \*/  } |
| argv - Command-line arguments are passed to main() as pointers to strings in the array members argv[0] through argv[argc - 1]. Vulnerabilities can occur when inadequate space is allocated to copy a command-line argument or other program input. In this noncompliant code example, an attacker can manipulate the contents of argv[0] to cause a buffer overflow. |
| #include <string.h>    **int** main(**int** argc, **char** \*argv[]) {    /\* Ensure argv[0] is not null \*/  **const** **char** \***const** name = (argc && argv[0]) ? argv[0] : "";  **char** prog\_name[128];  **strcpy**(prog\_name, name);    **return** 0;  } |
| getenv() - Environment variables can be arbitrarily large, and copying them into fixed-length arrays without first determining the size and allocating adequate storage can result in a buffer overflow. |
| #include <stdlib.h>  #include <string.h>    **void** func(**void**) {  **char** buff[256];  **char** \*editor = **getenv**("EDITOR");  **if** (editor == NULL) {      /\* EDITOR environment variable not set \*/    } **else** {  **strcpy**(buff, editor);    }  } |
| sprintf() - In this noncompliant code example, name refers to an external string; it could have originated from user input, the file system, or the network. The program constructs a file name from the string in preparation for opening the file. |
| #include <stdio.h>    **void** func(**const** **char** \*name) {  **char** filename[128];  **sprintf**(filename, "%s.txt", name);  } |

| **Compliant Code** |
| --- |
| Off-by-One Error - In this compliant solution, the loop termination condition is modified to account for the null-termination character that is appended to dest. |
| #include <stddef.h>    **void** copy(**size\_t** n, **char** src[n], **char** dest[n]) {  **size\_t** i;    **for** (i = 0; src[i] && (i < n - 1); ++i) {       dest[i] = src[i];     }     dest[i] = '\0';  } |
| fgets() - The fgets() function reads, at most, one less than the specified number of characters from a stream into an array. This solution is compliant because the number of characters copied from stdin to buf cannot exceed the allocated memory. |
| #include <stdio.h>  #include <string.h>    **enum** { BUFFERSIZE = 32 };    **void** func(**void**) {  **char** buf[BUFFERSIZE];  **int** ch;    **if** (**fgets**(buf, **sizeof**(buf), stdin)) {      /\* fgets() succeeded; scan for newline character \*/  **char** \*p = **strchr**(buf, '\n');  **if** (p) {        \*p = '\0';      } **else** {        /\* Newline not found; flush stdin to end of line \*/  **while** ((ch = **getchar**()) != '\n' && ch != EOF)          ;  **if** (ch == EOF && !**feof**(stdin) && !**ferror**(stdin)) {            /\* Character resembles EOF; handle error \*/        }      }    } **else** {      /\* fgets() failed; handle error \*/    }  } |
| gets\_s() - The gets\_s() function reads, at most, one less than the number of characters specified from the stream pointed to by stdin into an array. If end-of-file is encountered and no characters have been read into the destination array, or if a read error occurs during the operation, then the first character in the destination array is set to the null character and the other elements of the array take unspecified values. |
| #define \_\_STDC\_WANT\_LIB\_EXT1\_\_ 1  #include <stdio.h>    **enum** { BUFFERSIZE = 32 };    **void** func(**void**) {  **char** buf[BUFFERSIZE];    **if** (gets\_s(buf, **sizeof**(buf)) == NULL) {      /\* Handle error \*/    }  } |
| getline() - If passed a null pointer, getline() dynamically allocates a buffer of sufficient size to hold the input. If passed a pointer to dynamically allocated storage that is too small to hold the contents of the string, the getline() function resizes the buffer, using realloc(), rather than truncating the input. Allocated memory must be explicitly deallocated by the caller to avoid memory leaks. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    **void** func(**void**) {  **int** ch;  **size\_t** buffer\_size = 32;  **char** \*buffer = **malloc**(buffer\_size);    **if** (!buffer) {      /\* Handle error \*/  **return**;    }    **if** ((ssize\_t size = getline(&buffer, &buffer\_size, stdin))          == -1) {      /\* Handle error \*/    } **else** {  **char** \*p = **strchr**(buffer, '\n');  **if** (p) {        \*p = '\0';      } **else** {        /\* Newline not found; flush stdin to end of line \*/  **while** ((ch = **getchar**()) != '\n' && ch != EOF)          ;  **if** (ch == EOF && !**feof**(stdin) && !**ferror**(stdin)) {           /\* Character resembles EOF; handle error \*/        }      }    }  **free** (buffer);  } |
| getchar() - In this compliant solution, characters are no longer copied to buf once index == BUFFERSIZE - 1, leaving room to null-terminate the string. The loop continues to read characters until the end of the line, the end of the file, or an error is encountered. When chars\_read > index, the input string has been truncated. |
| #include <stdio.h>    **enum** { BUFFERSIZE = 32 };    **void** func(**void**) {  **char** buf[BUFFERSIZE];  **int** ch;  **size\_t** index = 0;  **size\_t** chars\_read = 0;    **while** ((ch = **getchar**()) != '\n' && ch != EOF) {  **if** (index < **sizeof**(buf) - 1) {        buf[index++] = (**char**)ch;      }      chars\_read++;    }    buf[index] = '\0';  /\* Terminate string \*/  **if** (ch == EOF) {      /\* Handle EOF or error \*/    }  **if** (chars\_read > index) {      /\* Handle truncation \*/    }  } |
| fscanf() - In this compliant solution, the call to fscanf() is constrained not to overflow buf. |
| #include <stdio.h>    **enum** { BUF\_LENGTH = 1024 };    **void** get\_data(**void**) {  **char** buf[BUF\_LENGTH];  **if** (1 != **fscanf**(stdin, "%1023s", buf)) {      /\* Handle error \*/    }      /\* Rest of function \*/  } |
| argv - The strlen() function can be used to determine the length of the strings referenced by argv[0] through argv[argc - 1] so that adequate memory can be dynamically allocated. |
| #include <stdlib.h>  #include <string.h>    **int** main(**int** argc, **char** \*argv[]) {    /\* Ensure argv[0] is not null \*/  **const** **char** \***const** name = (argc && argv[0]) ? argv[0] : "";  **char** \*prog\_name = (**char** \*)**malloc**(**strlen**(name) + 1);  **if** (prog\_name != NULL) {  **strcpy**(prog\_name, name);    } **else** {      /\* Handle error \*/    }  **free**(prog\_name);  **return** 0;  } |
| argv - The strcpy\_s() function provides additional safeguards, including accepting the size of the destination buffer as an additional argument. The strcpy\_s() function can be used to copy data to or from dynamically allocated memory or a statically allocated array. If insufficient space is available, strcpy\_s() returns an error. |
| #define \_\_STDC\_WANT\_LIB\_EXT1\_\_ 1  #include <stdlib.h>  #include <string.h>    **int** main(**int** argc, **char** \*argv[]) {    /\* Ensure argv[0] is not null \*/  **const** **char** \***const** name = (argc && argv[0]) ? argv[0] : "";  **char** \*prog\_name;  **size\_t** prog\_size;      prog\_size = **strlen**(name) + 1;    prog\_name = (**char** \*)**malloc**(prog\_size);    **if** (prog\_name != NULL) {  **if** (strcpy\_s(prog\_name, prog\_size, name)) {        /\* Handle  error \*/      }    } **else** {      /\* Handle error \*/    }    /\* ... \*/  **free**(prog\_name);  **return** 0;  } |
| argv - If an argument will not be modified or concatenated, there is no reason to make a copy of the string. Not copying a string is the best way to prevent a buffer overflow and is also the most efficient solution. |
| **int** main(**int** argc, **char** \*argv[]) {    /\* Ensure argv[0] is not null \*/  **const** **char** \* **const** prog\_name = (argc && argv[0]) ? argv[0] : "";    /\* ... \*/  **return** 0;  } |
| getenv() - Environmental variables are loaded into process memory when the program is loaded. As a result, the length of these strings can be determined by calling the strlen() function, and the resulting length can be used to allocate adequate dynamic memory. |
| #include <stdlib.h>  #include <string.h>    **void** func(**void**) {  **char** \*buff;  **char** \*editor = **getenv**("EDITOR");  **if** (editor == NULL) {      /\* EDITOR environment variable not set \*/    } **else** {  **size\_t** len = **strlen**(editor) + 1;      buff = (**char** \*)**malloc**(len);  **if** (buff == NULL) {        /\* Handle error \*/      }  **memcpy**(buff, editor, len);  **free**(buff);    }  } |
| sprintf() - Buffer overflow can be prevented by adding a precision to the %s conversion specification. If the precision is specified, no more than that many bytes are written. |
| #include <stdio.h>    **void** func(**const** **char** \*name) {  **char** filename[128];  **sprintf**(filename, "%.123s.txt", name);  } |
| snprintf() - A more general solution is to use the snprintf() function. |
| #include <stdio.h>    **void** func(**const** **char** \*name) {  **char** filename[128];    snprintf(filename, **sizeof**(filename), "%s.txt", name);  } |

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

| **Principles(s):**  ValidateInput Data (1): Check to make sure the input is a sufficient size to prevent buffer overflow.  Heed Compiler Warnings (2): Typically, a high-quality compiler and analyzer will catch an overflow error.  Use Effective Quality Assurance Techniques (9): Test inputs to make sure the code reacts appropriately when trying to overflow a buffer. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High (P18) | High (1) |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 | N/A | Supported  Astrée reports all buffer overflows resulting from copying data to a buffer that is not large enough to hold that data. |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR31 | Detects calls to unsafe string function that may cause buffer overflow Detects potential buffer overruns, including those caused by unsafe usage of fscanf() |
| CodeSonar | 7.0p0 | LANG.MEM.BO  LANG.MEM.TO  MISC.MEM.NTERM  BADFUNC.BO.\* | Buffer overrun Type overrun No space for null terminator A collection of warning classes that report uses of library functions prone to internal buffer overflows |
| Compass/ROSE | N/A | N/A | Can detect violations of the rule. However, it is unable to handle cases involving strcpy\_s() or manual string copies such as the one in the first example |
| Coverity | 2017.07 | STRING \_OVERFLOW , BUFFER\_SIZE  OVERRUN , STRING\_SIZE | Fully implemented |
| Fortify SCA | 5.0 | N/A | N/A |
| Helix QAC | 2022.2 | **C2840 , C2841 , C2842 , C2843 , C2845 C2846 , C2847 , C2848 , C2930 , C2931 C2932 , C2933 , C2935 , C2936 , C2937 C2938**  **C++0145 , C++2840 , C++2841 , C++2842 C++2843 , C++2845 , C++2846 , C++2847 C++2848 , C++2930 , C++2931 , C++2932 C++2933 , C++2935 , C++2936 , C++2937 C++2938** | N/A |
| Klocwork | 2022.2 | SV.FMT\_STR.BAD\_SCAN\_FORMAT  SV.UNBOUND\_STRING\_INPUT.FUNC | N/A |
| LDRA tool suit | 9.7.1 | 489 S , 109 D , 66 X , 70 X , 71 X | Partially implemented |
| Parasoft C/C++ test | 2022.1 | **CERT\_C-STR31-a CERT\_C-STR31-b CERT\_C-STR31-c CERT\_C-STR31-d CERT\_C-STR31-e** | Avoid accessing arrays out of bounds Avoid overflow when writing to a buffer Prevent buffer overflows from tainted data Avoid buffer write overflow from tainted data Avoid using unsafe string functions which may cause buffer overflows |
| PC-lint Plus | 1.4 | 421 , 498 | Partially supported |
| Polyspace Bug Finder | R2022a | CERT C: Rule STR31-C | Checks for:   * Use of dangerous standard function * Missing null in string array * Buffer overflow from incorrect string format specifier * Destination buffer overflow in string manipulation * Tainted NULL or non-null-terminated string   Rule partially covered. |
| PRQA QA-C | 9.7 | **5009 , 5038 , 2840 , 2841 , 2842 , 2843 2845 , 2846 , 2847 , 2848 , 2930 , 2931 2932 , 2933 , 2935 , 2936 , 2937 , 2938** | Partially implemented |
| PRQA QA-C++ | 4.4 | **0145 , 2840 , 2841 , 2842 , 2843 , 2845 2846 , 2847 , 2848 , 2930 , 2931 , 2932 2933 , 2935 , 2936 , 2937 , 2938 , 5006 5038** | N/A |
| PVS-Studio | 7.19 | V518 , V645 , V727 , V755 | N/A |
| Splint | 3.1.1 | N/A | N/A |
| TrustInSoft Analyzer | 1.38 | mem\_access | Exhaustively verified |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-009-CPP  (MSC52-CPP) | Value-returning functions must return a value from all exit paths  A value-returning function must return a value from all code paths; otherwise, it will result in undefined behavior. This includes returning through less-common code paths, such as from a function-try-block. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the programmer forgot to return the input value for positive input, so not all code paths return a value. |
| **int** absolute\_value(**int** a) {  **if** (a < 0) {  **return** -a;    }  } |
| In this noncompliant code example, the function-try-block handler does not return a value, resulting in undefined behavior when an exception is thrown. |
| #include <vector>    std::**size\_t** f(std::vector<**int**> &v, std::**size\_t** s) **try** {    v.resize(s);  **return** s;  } **catch** (...) {  } |

| **Compliant Code** |
| --- |
| In this compliant solution, all code paths now return a value. |
| **int** absolute\_value(**int** a) {  **if** (a < 0) {  **return** -a;    }  **return** a;  } |
| In this compliant solution, the exception handler of the *function-try-block* also returns a value. |
| #include <vector>    std::**size\_t** f(std::vector<**int**> &v, std::**size\_t** s) **try** {    v.resize(s);  **return** s;  } **catch** (...) {  **return** 0;  } |

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

| **Principles(s):**  Heed Compiler Warnings (2): If there is an issue with a return value, the compiler or analysis will catch it.  Keep It Simple (4): Don’t make the functions overly complicated with various exit paths. Otherwise, a return value maybe missed.  Use Effective Quality Assurance Techniques (9): When testing the code, make sure all exit paths have return values that output properly. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | Medium (P8) | Medium (2) |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | return-implicit | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MSC52 | N/A |
| Clang | 3.9 | -Wreturn-type | Does not catch all instances of this rule, such as function-try-blocks |
| CodeSonar | 7.0p0 | LANG.STRUCT.MRS | Missing return statement |
| Helix QAC | 2022.2 | C++ 2888 | N/A |
| Klocwork | 2022.2 | FUNCRET.GEN  FUNCRET.IMPLICIT | N/A |
| LDRA tool suite | 9.7.1 | 2 D , 36 S | Fully implemented |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-MSC52-a | All exit paths from a function, except main(), with non-void return type shall have an explicit return statement with an expression |
| Polyspace Bug Finder | R2022a | CERT C++: MSC52-CPP | Checks for missing return statements (rule partially covered) |
| SonarQube C/C++ Plugin | 4.10 | S935 | N/A |
| PRQA QA-C++ | 4.4 | 1510 | N/A |
| PVS-Studio | 7.19 | V591 | N/A |
| RuleChecker | 20.10 | return-implicit | Fully checked |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-010-CPP  (MEM50-CPP) | Do not access freed memory  Evaluating a pointer—including dereferencing the pointer, using it as an operand of an arithmetic operation, type casting it, and using it as the right-hand side of an assignment—into memory that has been deallocated by a memory management function is undefined behavior. Pointers to memory that has been deallocated are called dangling pointers. Accessing a dangling pointer can result in exploitable vulnerabilities. |

| **Noncompliant Code** |
| --- |
| new and delete - In this noncompliant code example, s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...  **delete** s;    // ...    s->f();  } |
| std::unique\_ptr - In the following noncompliant code example, the dynamically allocated memory managed by the buff object is accessed after it has been implicitly deallocated by the object's destructor. |
| #include <iostream>  #include <memory>  #include <cstring>    **int** main(**int** argc, **const** **char** \*argv[]) {  **const** **char** \*s = "";  **if** (argc > 1) {  **enum** { BufferSize = 32 };  **try** {        std::unique\_ptr<**char**[]> buff(**new** **char**[BufferSize]);        std::**memset**(buff.get(), 0, BufferSize);        // ...        s = std::**strncpy**(buff.get(), argv[1], BufferSize - 1);      } **catch** (std::bad\_alloc &) {        // Handle error      }    }      std::cout << s << std::endl;  } |
| std::string::c\_str() - In this noncompliant code example, std::string::c\_str() is being called on a temporary std::string object. The resulting pointer will point to released memory once the std::string object is destroyed at the end of the assignment expression, resulting in undefined behavior when accessing elements of that pointer. |
| #include <string>    std::string str\_func();  **void** display\_string(**const** **char** \*);    **void** f() {  **const** **char** \*str = str\_func().c\_str();    display\_string(str);  /\* Undefined behavior \*/  } |
| In this noncompliant code example, an attempt is made to allocate zero bytes of memory through a call to operator new(). If this request succeeds, operator new() is required to return a non-null pointer value, resulting in undefined behavior. |
| #include <new>    **void** f() noexcept(**false**) {    unsigned **char** \*ptr = **static\_cast**<unsigned **char** \*>(::operator **new**(0));    \*ptr = 0;    // ...    ::operator **delete**(ptr);  } |

| **Compliant Code** |
| --- |
| new and delete - In this compliant solution, the dynamically allocated memory is not deallocated until it is no longer required. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...    s->f();  **delete** s;  } |
| Automatic Storage Duration - When possible, use automatic storage duration instead of dynamic storage duration. Since s is not required to live beyond the scope of g(), this compliant solution uses automatic storage duration to limit the lifetime of s to the scope of g(). |
| **struct** S {  **void** f();  };    **void** g() {    S s;    // ...    s.f();  } |
| std::unique\_ptr - In this compliant solution, the lifetime of the buff object extends past the point at which the memory managed by the object is accessed. |
| #include <iostream>  #include <memory>  #include <cstring>    **int** main(**int** argc, **const** **char** \*argv[]) {    std::unique\_ptr<**char**[]> buff;  **const** **char** \*s = "";    **if** (argc > 1) {  **enum** { BufferSize = 32 };  **try** {        buff.reset(**new** **char**[BufferSize]);        std::**memset**(buff.get(), 0, BufferSize);        // ...        s = std::**strncpy**(buff.get(), argv[1], BufferSize - 1);      } **catch** (std::bad\_alloc &) {        // Handle error      }    }      std::cout << s << std::endl;  } |
| In this compliant solution, a variable with automatic storage duration of type std::string is used in place of the std::unique\_ptr<char[]>, which reduces the complexity and improves the security of the solution. |
| #include <iostream>  #include <string>    **int** main(**int** argc, **const** **char** \*argv[]) {    std::string str;    **if** (argc > 1) {      str = argv[1];    }      std::cout << str << std::endl;  } |
| std::string::c\_str() - In this compliant solution, a local copy of the string returned by str\_func() is made to ensure that string str will be valid when the call to display\_string() is made. |
| #include <string>    std::string str\_func();  **void** display\_string(**const** **char** \*s);    **void** f() {    std::string str = str\_func();  **const** **char** \*cstr = str.c\_str();    display\_string(cstr);  /\* ok \*/  } |
| If the programmer intends to allocate a single unsigned char object, the compliant solution is to use new instead of a direct call to operator new(), as this compliant solution demonstrates. |
| **void** f() noexcept(**false**) {    unsigned **char** \*ptr = **new** unsigned **char**;    \*ptr = 0;    // ...  **delete** ptr;  } |
| If the programmer intends to allocate zero bytes of memory, the recommended solution is to declare ptr as a void \*, which cannot be dereferenced by a conforming implementation. |
| #include <new>    **void** f() noexcept(**false**) {  **void** \*ptr = ::operator **new**(0);    // ...    ::operator **delete**(ptr);  } |

| **Principles(s):**  Heed Compiler Warnings (2): Trying to access freed memory will cause a warning to show on an analyzer.  Sanitize Data Sent to Other Systems (7): When a program is coded to properly free and delete memory after it is used and sent to another function or system, it should not be accessed again.  Practice Defense in Depth (8): If a mistake were to happen, and memory is accessed after being freed, the data should be encrypted. Encryption would prevent sensitive data to be taken if it were accessed through memory.  Adopt a Secure Coding Standard (10): It is essential to have developers know how to securely work with memory. So, it’s beneficial to go over how to properly use and free memory. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High (P18) | High (1) |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | dangling\_pointer\_use | N/A |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM50 | N/A |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete  clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule |
| CodeSonar | 7.0p0 | ALLOC.UAF | Use after free |
| Compass/ROSE | N/A | N/A | N/A |
| Coverity | v7.5.0 | USE\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| Helix QAC | 2022.2 | C++4303 , C++4304 | N/A |
| Klocwork | 2022.2 | UFM.DEREF.MIGHT , UFM.DEREF.MUST  UFM.FFM.MIGHT , UFM.FFM.MUST  UFM.RETURN.MIGHT , UFM.RETURN.MUST  UFM.USE.MIGHT , UFM.USE.MUST | N/A |
| LDRA tool suite | 9.7.1 | 483 S , 484 S | Partially implemented |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-MEM50-a | Do not use resources that have been freed |
| Parasoft Insure++ | N/A | N/A | Runtime detection |
| Polyspace Bug Finder | R2022a | CERT C++: MEM50-CPP | Checks for:   * Pointer access out of bounds * Deallocation of previously deallocated pointer * Use of previously freed pointer   Rule partially covered. |
| PRQA QA-C++ | 4.4 | 4303, 4304 | N/A |
| PVS-Studio | 7.19 | V586 , V774 | N/A |
| Splint | 5.0 | N/A | N/A |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

### Automation



Automation helps keeps the program secure and compliant with the coding standards. During the design process, utilizing the coding standards within the policy would help minimize any vulnerabilities found after and during development. Next, during building, typically compilers will warn the developer of any crucial errors withing the code. Then, after the program is built, using an automated static analyzer during the verifying phase will find any other defects early in the development process. Finding and mitigating these vulnerabilities early saves time and money. Furthermore, after deployment, using automation while maintaining the program helps it stay up-to-date and secure as the coding standards change.

### 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-GLC | Low | Unlikely | High | Low | 3 |
| STD-002-CPP | Medium | Unlikely | Medium | Low | 3 |
| STD-003-CPP | High | Probable | High | Medium | 2 |
| STD-004-JAV | High | Probable | Medium | High | 1 |
| STD-005-CPP | High | Likely | Medium | High | 1 |
| STD-006-CLG | Low | Unlikely | High | Low | 3 |
| STD-007-CPP | Low | Probable | Medium | Low | 3 |
| STD-008-CLG | High | Likely | Medium | High | 1 |
| STD-009-CPP | Medium | Probable | Medium | Medium | 2 |
| STD-010-CPP | High | Likely | Medium | High | 1 |

### Create Policies for Encryption and Triple A

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption in rest is stored data that sits in the system, such as credit card information, address, and phone number, is encrypted when it is not used by an authorized user. Only authorized users with the key to decode it can see the data. This policy applies because very sensitive data may get stored within systems we build. So, it is important to keep that data secure and safe from any attack. |
| Encryption at flight | Encryption in flight is where data that is being transported is encrypted while in route. This policy applies because sensitive data will get sent and requested between our servers and clients. Encrypting this data prevents it from being viewed if it were to get intercepted by an attacker during transportation. |
| Encryption in use | Encryption in use is where active sensitive data, stored in memory, is encrypted within the system, no matter if it is at rest, in transit, or being accessed. For example, if a system user inputs their password, that input is encrypted through hashing to verify it. Also, it is encrypted when it is being sent to the server. Encryption in use applies because most systems require user authentication with login information. Furthermore, protecting data no matter its lifecycle stage greatly reduces the risk of the data being taken during a breach. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is where the system, typically using login information, identifies the user. Authentication can be done by username, email, password, phone number, keys, and biometrics. By using authentication, it keeps data secure from being seen by just anyone. The user can only view and access data or files they are allowed to see. Authentication applies because all new users added should register accounts to gain access to our systems. |
| Authorization | Authorization is giving data access to users based on their role and privileges. For example, during authentication, the system will verify the level of access that specific user has, such as admin, tech support, or user. The data the user can access depends on what their system role is listed as. This allows for important system data to be safe from unwanted eyes. Authorization applies because all our systems should utilize it. A typical user should not have access to and/or be able to change databases within our systems. |
| Accounting | Accounting is where the system tracks and logs the system user activity. For example, if an admin user makes a change to the database, this should be tracked incase of any malicious intent. Accounting prevents attacks because it catches suspicious user activity quickly, preventing any possible system breaches. Since we will be using user logins and databases that could get targeted, it is important we log any activity associated with them. |

**\***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

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

## 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 | 07/12/2022 | Security Principles and Coding Standards | Elizabeth Hodgman | Jane Doe |
| 2.1 | 8/3/2022 | Finished Security Principles, Mapping the principles and Risk Assessment Summary | Elizabeth Hodgman | Jane Doe |
| 2.2 | 8/4/2022 | Automation and Encryption/Triple A policies | Elizabeth Hodgman | Jane Doe |

## Apendix A Lookups

### Approved C/C++ Language Acronyms

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

**Reference Used**

Carnegie Mellon University Software Engineering Institute. [SEI External Wiki Home - Homepage - Confluence (cmu.edu)](https://wiki.sei.cmu.edu/confluence/). Provides information about security policies for programming systems and languages.

Das, P. (n.d.). In-Use Encryption – What it is and How Companies Benefit. *Soterosoft*. Retrieved from [Data in Use Encryption | What It Is and How Companies Benefit (soterosoft.com)](https://www.soterosoft.com/blog/data-in-use-encryption-data-in-motion-encryption/)