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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 6](#_Toc52464060)

[Coding Standard 3 8](#_Toc52464061)

[Coding Standard 4 10](#_Toc52464062)

[Coding Standard 5 14](#_Toc52464063)

[Coding Standard 6 16](#_Toc52464064)

[Coding Standard 7 18](#_Toc52464065)

[Coding Standard 8 20](#_Toc52464066)

[Coding Standard 9 22](#_Toc52464067)

[Coding Standard 10 24](#_Toc52464068)

[Defense-in-Depth Illustration 26](#_Toc52464069)

[Project One 26](#_Toc52464070)

[1. Revise the C/C++ Standards 26](#_Toc52464071)

[2. Risk Assessment 26](#_Toc52464072)

[3. Automated Detection 26](#_Toc52464073)

[4. Automation 26](#_Toc52464074)

[5. Summary of Risk Assessments 27](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 28](#_Toc52464076)

[7. Map the Principles 29](#_Toc52464077)

[Audit Controls and Management 30](#_Toc52464078)

[Enforcement 30](#_Toc52464079)

[Exceptions Process 30](#_Toc52464080)

[Distribution 31](#_Toc52464081)

[Policy Change Control 31](#_Toc52464082)

[Policy Version History 31](#_Toc52464083)

[Appendix A Lookups 31](#_Toc52464084)

[Approved C/C++ Language Acronyms 31](#_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 | Input data is any data that comes from an outside source. Input data should always be assumed to be malicious or corrupt until it has gone through a validation process. The validation process should either deny or correct any mal input before it is allowed to proceed through the system. |
| 1. Heed Compiler Warnings | During the compilation process, the compiler will search the code for any security vulnerabilities and errors. In addition, the compiler will also provide warnings to bits of code that, while not critical, could lead to errors. It’s important to remember that the compiler will only be able to catch a limited scope of errors. Compiler warnings should NOT be the only safeguard. Instead, it should be used in conjunction with other mitigation tools and security techniques . |
| 1. Architect and Design for Security Policies | Security should never be an afterthought. Instead, security should be at the forefront of the solution design. Considering security in the initial design phase will save time and money as opposed to trying to implement security into an existing application. When security is built into the application form the start, it will be better integrated and thus better protect the system. |
| 1. Keep It Simple | By default, complexity adds attack surface. It’s important to keep the code coherent and optimized in an effort to be maintainable and serviceable. It’s important to note that security features should not be removed in favor of a simpler design. |
| 1. Default Deny | Default deny means that the default setting is to deny access. If the default setting is to deny, then the entities that are allowed access have been clearly authenticated. In the opposite case, default allow, the organization would have to keep updating a blacklist. This is not practical because of the case where if the blacklist is not up to date, a threat actor could easily slip by. Default deny is best practice. |
| 1. Adhere to the Principle of Least Privilege | The Principle of Least Privilege means that an authenticated user is only given the minimum amount of privileges to perform whatever task they need to. Giving a user extraneous privileges could allow that user account to abuse those privileges if that user account were to be used by a threat actor. |
| 1. Sanitize Data Sent to Other Systems | Similar to input validation, a system should never fully trust data from an outside source, even if that outside source is another system within the umbrella of the operation. Outside data should always be assumed to be corrupt and malicious and should always be sanitized. This will prevent infected systems from infecting other systems via data transmission. |
| 1. Practice Defense in Depth | Security at just one layer is not enough. Defense in Depth is the practice of adding multiple layers of security across the application stack. Just because a component is not forward facing, does not mean it is not vulnerable to an attack. All components should be secured against malicious interaction and also interactions that could crash the component. |
| 1. Use Effective Quality Assurance Techniques | Creating security measures is just the start. Security measures need to be tested to ensure that they are in fact protecting the system against malicious use. When testing is implemented from design to production, it should be tested along with other components of the system. Checking for errors, bugs, and security vulnerabilities is quintessential to a secure application. It’s important to note that testing cases should be extensive and cover both expected and unexpected scenarios. |
| 1. Adopt a Secure Coding Standard | A secure coding standard should be applied throughout the entire development process. An emphasis should always be placed on security and using mitigation techniques. A part of this standard should be a set of application specific rules that all developers must follow to ensure consistent application of secure coding practices across the development effort. |

### 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-CPP | 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):** The non-compliant code for this example results in undefined behavior. Preventing issues like this one falls under the principle of adopting a secure coding standard. There may be tools like static code checkers that can identify this type of issue. Using a code checker is an aspect of using effective Quality Assurance. Adopting a secure coding standard entails the effort of preventing these kinds of errors from happening. Further, if an error such as this is present, the secure coding efforts will attempt to discover and remedy it. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL52 |  |
| Helix QAC | 2023.1 | C++0014 |  |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-DCL52-a | Never qualify a reference type with 'const' or 'volatile' |
| Polyspace Bug Finder | [Insert text.] | CERT C++: DCL52-CPP | Checks for:   * const-qualified reference types * Modification of const-qualified reference types   Rule fully covered. |
| Clang | 3.9 |  | Clang checks for violations of this rule and produces an error without the need to specify any special flags or options. |
| SonarQube C/C++ Plugin | 4.10 | S3708 |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not use floating-point variables as loop counters** |
| --- | --- | --- |
| **Data Value** | STD-002-C | Because floating-point numbers represent real numbers, it is often mistakenly assumed that they can represent any simple fraction exactly. Floating-point numbers are subject to representational limitations just as integers are, and binary floating-point numbers cannot represent all real numbers exactly, even if they can be represented in a small number of decimal digits. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a floating-point variable is used as a loop counter. The decimal number 0.1 is a repeating fraction in binary and cannot be exactly represented as a binary floating-point number. Depending on the implementation, the loop may iterate 9 or 10 times. |
| **void** func(**void**) {  **for** (**float** x = 0.1f; x <= 1.0f; x += 0.1f) {      /\* Loop may iterate 9 or 10 times \*/    }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the loop counter is an integer from which the floating-point value is derived: |
| #include <stddef.h>    **void** func(**void**) {  **for** (**size\_t** count = 1; count <= 10; ++count) {  **float** x = count / 10.0f;      /\* Loop iterates exactly 10 times \*/    }  } |

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

| **Principles(s):** This is another example of the Adopting a Secure Coding Standard principle. This type of error can be easily committed by a rushed developer. However, using effective Quality Assurance tools like code checkers can help to spot these errors in the development phase. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | For-loop-float | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-FLP30 | Fully implemented |
| Clang | 3.9 | Cert-fpl30-c | Checked by clang-tidy |
| CodeSonar | 7.3p0 | LANG.STRUCT.LOOP.FPC | Float-typed loop counter |
| Coverity | 2017.07 | MISRA C 2004 Rule 13.4 | Implemented |
| ÉCLAIR | 1.2 | CC2.FPL30 | Fully implemented |
| Helix QAC | 2023.1 | C3339, C3340, C3342 |  |
| SonarQube C/C++ Plugin | 3.11 | S2193 | Fully implemented |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do not attempt to modify string literals** |
| --- | --- | --- |
| **String Correctness** | STD-003-C | According to the C Standard, 6.4.5, paragraph 3 [ISO/IEC 9899:2011]:  A character string literal is a sequence of zero or more multibyte characters enclosed in double-quotes, as in "xyz". A UTF−8 string literal is the same, except prefixed by u8. A wide string literal is the same, except prefixed by the letter L, u, or U.  At compile time, string literals are used to create an array of static storage duration of sufficient length to contain the character sequence and a terminating null character. String literals are usually referred to by a pointer to (or array of) characters. Ideally, they should be assigned only to pointers to (or arrays of) const char or const wchar\_t. It is unspecified whether these arrays of string literals are distinct from each other. The behavior is undefined if a program attempts to modify any portion of a string literal. Modifying a string literal frequently results in an access violation because string literals are typically stored in read-only memory. (See undefined behavior 33.) |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the char pointer str is initialized to the address of a string literal. Attempting to modify the string literal is undefined behavior: |
| **char** \*str  = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| As an array initializer, a string literal specifies the initial values of characters in an array as well as the size of the array. (See STR11-C. Do not specify the bound of a character array initialized with a string literal.) This code creates a copy of the string literal in the space allocated to the character array str. The string stored in str can be modified safely. |
| **char** str[] = "string literal";  str[0] = 'S'; |

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

| **Principles(s):** This is yet another error in which the developer makes a mistake. Adopting a Secure Coding standard in conjunction with Quality Assurance are the principles that will catch and fix this type of error. This may overlap with the principle of sanitizing data from other systems. In which case, it is important that the team checks all the interactions using Quality Assurance tools such as code checkers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | String-literal-modification-write-to-string-literal | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR30 | Fully implemented |
| Coverity | 2017.07 | PW | Deprecates conversion from string literal to “char \*” |
| Helix QAC | 2023.1 | C0556, C0752, C0753, C)754 |  |
| Parasoft C/C++ test | 2022.2 | CERT\_C-STR30-a CERT\_C-STR30-b | A string literal shall not be modified |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent SQL injection** |
| --- | --- | --- |
| **SQL Injection** | STD-001-JAV | 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. The password argument cannot be used to attack this program because it is passed to the hashPassword() function, which also sanitizes the input. |
| **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.ResultSet;  **import** java.sql.SQLException;  **import** java.sql.Statement;    **class** Login {  **public** Connection getConnection() **throws** SQLException {      DriverManager.registerDriver(**new**              com.microsoft.sqlserver.jdbc.SQLServerDriver());      String dbConnection =        PropertyManager.getProperty("db.connection");      // Can hold some value like      // "jdbc:microsoft:sqlserver://<HOST>:1433,<UID>,<PWD>"  **return** DriverManager.getConnection(dbConnection);    }      String hashPassword(**char**[] password) {      // Create hash of password    }    **public** **void** doPrivilegedAction(String username, **char**[] password)  **throws** SQLException {      Connection connection = getConnection();  **if** (connection == **null**) {        // Handle error      }  **try** {        String pwd = hashPassword(password);          String sqlString = "SELECT \* FROM db\_user WHERE username = '"                           + username +                           "' AND password = '" + pwd + "'";        Statement stmt = connection.createStatement();        ResultSet rs = stmt.executeQuery(sqlString);    **if** (!rs.next()) {  **throw** **new** SecurityException(            "User name or password incorrect"          );        }          // Authenticated; proceed      } **finally** {  **try** {          connection.close();        } **catch** (SQLException x) {          // Forward to handler        }      }    }  } |

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

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

| **Principles(s):** This coding standard falls under the validate input data principle. The query for an SQL database (any database really) is likely going to include user input in some fashion. It is important to correctly validate and handle user input before feeding it to the SQL database to prevent injection attacks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | 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 |
| Coverity | 7.5 | SQLI | Implemented |
| Findbugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Parasoft Jtest | 2022.2 | CERT.IDS00.TDSQL | Protect against SQL injection |
| SonarQube | 9.9 | S2077 | Executing SQL queries is security-sensitive |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Properly deallocate dynamically allocated resources** |
| --- | --- | --- |
| **Memory Protection** | STD-004-CPP | The C programming language provides several ways to allocate memory, such as std::malloc(), std::calloc(), and std::realloc(), which can be used by a C++ program. However, the C programming language defines only a single way to free the allocated memory: std::free(). See MEM31-C. Free dynamically allocated memory when no longer needed and MEM34-C. Only free memory allocated dynamically for rules specifically regarding C allocation and deallocation requirements. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, two allocations are attempted within the same try block, and if either fails, the catch handler attempts to free resources that have been allocated. However, because the pointer variables have not been initialized to a known value, a failure to allocate memory for i1 may result in passing ::operator delete() a value (in i2) that was not previously returned by a call to ::operator new(), resulting in undefined behavior. |
| #include <new>    **void** f() {  **int** \*i1, \*i2;  **try** {      i1 = **new** **int**;      i2 = **new** **int**;    } **catch** (std::bad\_alloc &) {  **delete** i1;  **delete** i2;    }  } |

| **Compliant Code** |
| --- |
| This compliant solution initializes both pointer values to nullptr, which is a valid value to pass to ::operator delete(). |
| #include <new>    **void** f() {  **int** \*i1 = nullptr, \*i2 = nullptr;  **try** {      i1 = **new** **int**;      i2 = **new** **int**;    } **catch** (std::bad\_alloc &) {  **delete** i1;  **delete** i2;    }  } |

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

| **Principles(s):** This is once again a developer created flaw which can result in undefined behavior. When it comes to memory management, the compiler provides a fair amount of memory-based information. However, not all memory issues are directly caught by the compiler. The Heed Compiler Warnings principle should be applied here as well as the principle of Quality Assurance. Tools like static checkers will allow developers to better find issues such as these and remedy them before they get released into production. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Invalid\_dynamic\_memory\_allocation |  |
| Axivion Bauhaus Suite | 3.9 | clang-analyzer-cplusplus.NewDeleteLeaks  -Wmismatched-new-delete  clang-analyzer-unix.MismatchedDeallocator | Checked by clang-tidy, but does not catch all violations of this rule |
| CodeSonar | 7.3p0 | ALLOC.FNH | Free non-heap variable Double free Type mismatch Leak |
| Klocwork | 2023.1 | CL.FFM.ASSIGN |  |
| LDRA tool suite | 9.7.1 | 232 S | Partially Implemented |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use a static assertion to test the value of a constant expression** |
| --- | --- | --- |
| **Assertions** | STD-005-CPP | Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities (see 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** |
| --- |
| This portable compliant solution uses static\_assert: |
| #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):** Using assert statements is part of the principle of Quality Assurance. Assert statements verify that the code is behaving as expected and it should be used in both development and testing. Using these types of Quality Assurance measures should be a part of a larger effort of the principle of Adopting Secure Coding practices. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bahaus Site | 7.2.0 | CertD-DCL03 |  |
| Clang | 3.9 | Misc-static-assert | Checked by clang-tidy |
| CodeSonar | 7.3p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| Éclair | 1.2 | CC2.DL03 | Fully Implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | STD-006-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. |

| **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):** Catching all exceptions is a part of the Architect and Design for Security Policies principle. Handling all exceptions entails that the developers and design team have created a plan for every eventuality the application may face. Using a static code checker to catch these issues is a part of the Secure Coding standard and Quality Assurance principles. |
| --- |

**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 |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR51 |  |
| CodeSonar | 7.3p0 | LANG>STRUCT>UCTCH | Unreachable Catch |
| Helic QAC | 2023.1 | C++4035 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Do not use std::rand() for generating pseudorandom numbers** |
| --- | --- | --- |
| Random Number Generation | STD-007-CPP | Pseudorandom number generators use mathematical algorithms to produce a sequence of numbers with good statistical properties, but the numbers produced are not genuinely random.  The C Standard rand() function, exposed through the C++ standard library through <cstdlib> as std::rand(), makes no guarantees as to the quality of the random sequence produced. The numbers generated by some implementations of std::rand() have a comparatively short cycle, and the numbers can be predictable. Applications that have strong pseudorandom number requirements must use a generator that is known to be sufficient for their needs. |

| **Noncompliant Code** |
| --- |
| The following noncompliant code generates an ID with a numeric part produced by calling the rand() function. The IDs produced are predictable and have limited randomness. Further, depending on the value of RAND\_MAX, the resulting value can have modulo bias. |
| #include <cstdlib>  #include <string>    **void** f() {    std::string id("ID"); // Holds the ID, starting with the characters "ID" followed                          // by a random integer in the range [0-10000].    id += std::to\_string(std::**rand**() % 10000);    // ...  } |

| **Compliant Code** |
| --- |
| The C++ standard library provides mechanisms for fine-grained control over pseudorandom number generation. It breaks random number generation into two parts: one is the algorithm responsible for providing random values (the engine), and the other is responsible for distribution of the random values via a density function (the distribution). The distribution object is not strictly required, but it works to ensure that values are properly distributed within a given range instead of improperly distributed due to bias issues. This compliant solution uses the Mersenne Twister algorithm as the engine for generating random values and a uniform distribution to negate the modulo bias from the noncompliant code example. |
| #include <random>  #include <string>    **void** f() {    std::string id("ID"); // Holds the ID, starting with the characters "ID" followed                          // by a random integer in the range [0-10000].    std::uniform\_int\_distribution<**int**> distribution(0, 10000);    std::random\_device rd;    std::mt19937 engine(rd());    id += std::to\_string(distribution(engine));    // ...  } |

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

| **Principles(s):** Avoiding the use of the std::rand() function is an example of the Defense in Depth principle. Just because the perimeter of the application is secure does not mean the interior is as well. The use of rand() and have a predictability vulnerability. Using the principle of Establishing a Secure Coding Standard will help catch and prevent this error along with Quality Assurance tools like code checkers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Bad-function(AUTOSAR.26.5.1A) | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MSC50 |  |
| Clang | 4.0 (prerelease) | Cert-msc50-cpp | Checked by clang-tidy |
| CodeSonar | 7.3p0 | BADFUNC.RANDOM.RAND | Use of rand |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Close files when they are no longer needed** |
| --- | --- | --- |
| File Usage | 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. 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):** Failure to adhere to this rule may result in objects not being closed properly which can lead to memory and storage issues. Implementing checks for this issue is a part of deploying a Secure Coding standard. Checking the code for these kinds of issues with a static code checker are part of the Quality Assurance principle. |
| --- |

**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 |
| Helix QAC | 2023.1 | DF4786 |  |
| Klocwork | 2023.1 | RH.EAK |  |
| Parasoft Insure++ | 2022.2 | CERT\_CPP-FIO51-a | Ensure resources are freed |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Do not access freed memory** |
| --- | --- | --- |
| Memory Management | STD-009-CPP | 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** |
| --- |
| [Noncompliant description] |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...  **delete** s;    // ...    s->f();  } |

| **Compliant Code** |
| --- |
| [Compliant description] |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...    s->f();  **delete** s;  } |

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

| **Principles(s):** This kind of error can lead to abnormal program termination. Preventing these kinds of issues is a part of the Defense in Depth principle. The inner workings of the application need to be hardened along with the perimeter to apply Defense in Depth. Using a static code checker is a part of the Secure Coding Standard and Quality assurance principles. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Dangling\_pointer\_use |  |
| Axivious Bauhaus Suite | 7.2.0 | CertC++-MEM50 |  |
| Clang | 3.9 | Clang-analyzer-cplusplus.NewDelete | Checked by clang-tidy, but does not catch all violations of this rule |
| CodeSonar | 7.3p0 | ALLOC.UAF | User after free |

### 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.

Many of the coding standards outlined in this document were a part of the Quality Assurance principle. The automation tools defined for these standards were quite similar in that they were static code analysis tools. Static code analysis tools would be used in the “Verify and test” section of the DevSecOps process. Once the code has been built, the static analysis can be run on it. If the static analysis is clean, the process can move on to “Transition and health check”. However, if the static analysis finds flaws, the process should be redirected back to the design and build phase. The warnings addressed by the static analysis should be fixed or redesigned on an individual basis. When the necessary refactors are made, the code can be run with the static analyzer again. This cycle should be repeated until the static analysis is clean. Then the code can move onto the “Transition and health check” phase.

Additionally, any compiler warnings should be addressed at the “Verify and test” phase. Once the code has been statically analyzed, compiled languages will require the code to be compiled. The compiler is equipped with its own set of flaw detections, and it will generate a report similar to the static analyzer. All compiler warnings should prevent the code from moving to the next stage. Compiler warnings should have the same trajectory as the static analysis warnings in that they move the code back to the design and build phase for refactoring.

### 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-CPP | Low | Probable | Low | P6 | L2 |
| STD-003-CPP | Low | Likely | Low | P9 | L2 |
| STD-001-JAV | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | Low | Unlikely | High | P1 | L3 |
| STD-006-CPP | Low | Probable | Medium | P4 | L3 |
| STD-007-CPP | Medium | Unlikely | Low | P6 | L2 |
| STD-008-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-009-CPP | High | Likely | Medium | P18 | L1 |

### 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 | Examples of data at rest include backups, archives, and other data stores that are infrequently used. Encryption of these data stores prevents a threat actor from being able to read the data if they were able to gain access to it. Symmetric encryption algorithms are best suited for this kind of data. There is only one encryption key and the algorithms currently used are very strong. AES-256 is considered military grade and has never been cracked |
| Encryption at flight | Encryption in flight is any data that is transferred from one machine to another. In order to stop a threat actor from being able to read any intercepted communications, encryption in flight should be used. Asymmetric encryption algorithms are best used here because the machines communicating need to agree on the encryption via public and private key. The SSL encryption method currently used is RSA. However, with the rise of quantum computing, questions have arisen surrounding the plausibility of cracking the RSA algorithm. Despite RSA being the most commonly used, a more secure asymmetric encryption algorithm such as ECC-224 should be used. |
| Encryption in use | Encryption in use corresponds to data that is currently being used. An example of this is data sitting in memory that has been pulled from a file or a database. The data sitting in memory is unencrypted and thus vulnerable. Homomorphic encryption algorithms are best used here because they allow data in use to remain encrypted, but still be able to be used. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the practice of verifying that the party in question is who they say they are. An example of this is the common username and password scheme that is used in the industry. Only the person with the password is allowed access. The password is intended to be a secret and used to verify the identity of the party. Authentication is important because it prevents only parties that have been granted access the ability to use the application. |
| Authorization | Authorization is essentially a word for privileges. Once the user has made it through the authentication or login stage, the user has a set of privileges that they are allowed to use. An example might be that they have the privilege to read from the database but cannot write to it. This is an example of level of access and each user type should be given the least amount of privileges needed to complete whatever tasks they need to complete. |
| Accounting | Accounting is simply the recording of events. An example of this in security practice is using a log to keep track of all users who have edited the database. Accounting is used to verify that security measures are currently working and also identify potential holes in security. Accounting is also used to keep track of other resources (files, databases, API’s) for security auditing. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

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

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

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 05/22/2023 | Principles and Coding Standards | Jason O’Connell |  |
| 1.2 | 06/10/2023 | Threat Assessment and Automation | Jason O’Connell |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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