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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles for 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 | Validating user input is the process of checking any data entered by the user. This is extremely important because the wrong input could jeopardize your system, or data within. It could also lead to errors or buffer overflows that could leave your program open to potential attacks. |
| 1. Heed Compiler Warnings | Heeding compiler warnings is an important step in development. These warnings are put in place by other developers to make development on the IDE more secure. If there is a warning, it should be investigated and remediated as soon as possible. |
| 1. Architect and Design for Security Policies | Security is never something that should be left to the end of a project. Designing while keeping security as a priority through development will ensure more secure code. If it was left to the end of the development cycle, those changes are liable to break your program and end up costing you more time and money. Lastly, designing with security in mind allows for swifter mitigation of potential issues in the future. |
| 1. Keep It Simple | A more complicated design can lead to more vulnerabilities that are overlooked. A simple design will lead to cleaner, more efficient code, as well as making future maintenance on the system easier as well. |
| 1. Default Deny | This is the principle that states if permission is not explicitly granted, it is denied. Leaving a grey area could lead to potential attacks in the future or lead to data leaks. |
| 1. Adhere to the Principle of Least Privilege | Like default deny, the principle of least privilege states that a user should only have access to enough information, resources, or applications to complete the desired task. Any more privilege could lead to potential issues or attacks. |
| 1. Sanitize Data Sent to Other Systems | The process of ensuring that data conforms to the requirements of subsystems, or other systems before it is sent. This can involve deleting storage data so it may not be recovered for the sending systems, cryptographically sealing the data, or the physical destruction of servers. |
| 1. Practice Defense in Depth | Defense in depth is a strategy in which a series of defensive mechanisms are put in place to mitigate or prevent potential threats. Think of the firewall on your computer, or a 2-factor authentication system. Multiple tools are used in “layers” to ensure the security of an application. |
| 1. Use Effective Quality Assurance Techniques | Effective quality assurance is the backbone of security in an application. This is the process of vigorously testing applications before and after release. Principles such as test-driven development, code review, and integration testing are some examples of quality assurance techniques. |
| 1. Adopt a Secure Coding Standard | Establishing guidelines can ensure that developers adhere to correct security principles throughout development. Secure coding principles aren’t a 100% fix, but they aim to increase the security of your code, which in turn increases the security for your users and their data. Coding standards also ensure that future work on the program can be picked up easily without threatening to break any existing code in place. |

### 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-CLG | Implement abstract data types using opaque types.  Abstract data types are not restricted to object-oriented programming languages like Java and C++. This should be used in C programming languages as well. Abstract data types are most effective when used with private data types and information hiding. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example is based on the managed string library developed by CERT. 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 within the structure, violating the software engineering principles of information hiding and data encapsulation 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** |
| --- |
| Here we redefine the ‘string\_mx’ type to private, which in turn hide the implementation from users of the library. Also, in the header file, the ‘struct\_mx’ is there but hidden from users that are interacting with the data via 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):**  **Adopt a Secure Coding Standard (10):** The secure coding principle of always encapsulating data should always be implemented.  **Architect and Design for Security Policies (3):** As you can see in the noncompliant example, after making the struct insecure it can be accessed by the user. Since there is no security, malicious programmers could easily manipulate the data. It is important to implement software security early, and in every process. |
| --- |

**Threat Level**

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

**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 | R2023b | 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 | 2023.1 | CERT\_C-DCL12-a | If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Do not cast an out-of-range enumeration value.  The arithmetic value being cast must be within the range of values the enumeration can represent. By doing so, we ensure that we follow the rules of the intended design of enum constructs. We also promote easy-to-read code and reduce the risk of undefined behavior. |

| **Noncompliant Code** |
| --- |
| Here we attempt to check if a given value is within the range of acceptable enumeration values provided. The problem is that we are doing so after casting the enumeration type. It may not 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** |
| --- |
| This solution checks the enumeration value before performing any conversions to ensure that it does not result in unwanted values. We do this by restricting the value to a value in which we know an enumeration value exist (First). |
| **enum** EnumType {    First,    Second,    Third  };    **void** f(**int** intVar) {  **if** (intVar < First || intVar > Third) {      // Handle error    }    EnumType enumVar = **static\_cast**<EnumType>(intVar);  } |

| **Principles(s):**  **Validate Input Data (1):** Always validate that the input for the values is in range before casting. If done incorrectly, it can result in unspecified behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Cast-integer-to-enum | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-INT50 | N/A |
| CodeSonar | 8.0p0 | LANG.CAST.COERCE  LANG.CAST.VALUE | Coercion Alters Value  Cast Alters Value |
| Helix QAC | 2023.3 | C++3013 | N/A |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Guarantee that storage for strings has sufficient space for character data and the null terminator.  When you copy or insert a string to another string with insufficient space, it can lead to issues such as buffer overflow. This can leave vulnerabilities in your code, allowing for unwanted hackers to gain access to sensitive data. |

| **Noncompliant Code** |
| --- |
| In this example, we do not ensure that the input is bounded. Instead, we leave it unbounded and do not have checks in place to prevent a buffer overflow. |
| #include <iostream>    **void** f() {  **char** buf[12];    std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| In this code, we used std::string instead of a bounded array. This ensures that there is enough space because the std::string class has built-in functionality to prevent buffer overflows. |
| #include <iostream>  #include <string>    **void** f() {    std::string input;    std::string stringOne, stringTwo;    std::cin >> stringOne >> stringTwo;  } |

| **Principles(s):**  **Architect and Design for Security Policies (3):** Always verify your strings have sufficient space before transferring or copying them to avoid common issues like buffer overflow. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | STRING\_OVERFLOW  BUFER\_SIZE  OVERRUN  STRING\_SIZE | Fully implemented |
| Astree | 23.04 |  | Supported  Astrée reports all buffer overflows resulting from copying data to a buffer that is not large enough to hold that data. |
| CodeSonar | 8.0p0 | **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 |
| Klockwork | 2023.4 | SV.FMT\_STR.BAD\_SCAN\_FORMAT SV.UNBOUND\_STRING\_INPUT.FUNC |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-JAV | Prevent SQL injection.  When accepting input from outside sources, this can lead to SQL injection. Thus, leading to the possibility of data being stolen. It can be prevented with proper input validation to ensure that incoming data is clean. |

| **Noncompliant Code** |
| --- |
| In this example, we use Java database code to authenticate a user. We pass the password to a char array and then send it to the database for authentication. The problem is that there is no checks in place to ensure that the inputted string is not malicious. |
| **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** |
| --- |
| In the solution, we use the “?” character to act as a placeholder for the incoming argument. We also have checks in place ensuring the length is not too long. We also incorporate a prepared statement to ensure no injections are taking place. |
| **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        }      }    }  } |

| **Principles(s):**  **Validate Input Data (1):** Always validate all input to detect any malicious content.  **Heed Compiler Warnings (2**): Always listen to the analysis tools available to us through IDEs. They can catch vulnerabilities early on.  **Architect and Design for Security Policies (3):** Injections can be easily prevented if proper security design is implemented.  **Default Deny (5):** Always restrict access if the user is not properly authenticated.  **Sanitize Data Sent to Other Systems (7):** Checking the data to and from sources is a necessary practice to prevent harm from outside sources.  **Practice Defense in Depth (8):** Frequent testing and audits should catch common issues like injection vulnerabilities.  **Use Effective Quality Assurance Techniques (9):** Unit testing and stress testing can be used to detect injection vulnerabilities.  **Adopt a Secure Coding Standard (10):** SQL injections are very common and should be adopted into any security standard. |
| --- |

**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 | 8.0p0 | JAVA.IO.INJ.SQL | SQL Injection (Java) |
| Findbugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Parasoft Jtest | 2023.1 | CERT.IDS00.TDSQL | Protect against SQL injection |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Detect and handle memory allocation errors.  Memory allocation errors can lead to multiple issues with loading and retrieving data, or even crash the program. Anytime we initialize or modify a string, vector, or map, we are allocating memory on the heap. Using the “new” operator in the C standard library may or may not throw an error, so we must have checks in place to prevent these issues. |

| **Noncompliant Code** |
| --- |
| In this example, we initialize the array using the new[] operator and do not check the results. This could lead to the program crashing unexpectedly. |
| #include <cstring>    **void** f(**const** **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy = **new** **int**[size];    std::**memcpy**(copy, array, size \* **sizeof**(\*copy));    // ...  **delete** [] copy;  } |

| **Compliant Code** |
| --- |
| Here we use the nothrow option of the new operator. While no exception is thrown, instead we will receive a null pointer value if the allocation fails. This way, we can easily check if the allocation was successful or not. |
| #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;  } |

| **Principles(s):**  **Heed Compiler Warnings (2):** Compilers tend to catch memory errors, making them easy to prevent.  **Architect and Design for Security Policies (3):** Memory errors are something that need to be addressed during the development lifecycle. Ensuring proper design should prevent any.  **Adopt a Secure Coding Standard (10):** Memory errors are common and should be incorporated into every security policy. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled |
| LDRA tool suite | 9.7.1 | 45 D | Partially implemented |
| Parasoft Insure++ |  |  | Runtime detection |
| Polyspace Bug Finder | R2023b | CERT C++: MEM52-CPP | Checks for unprotected dynamic memory allocation (rule partially covered) |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CLG | Use static assertions to test the value of a constant expression.  Assertions are used to test the value of expressions in multiple programming languages. They ensure that the values are consistent and improve code reliability. |

| **Noncompliant Code** |
| --- |
| Here we use the assert() method to assert to test a block that is essential for the code to run. The problem is that an assert method should not stop a program, but rather inform the developer of a wrong value. |
| #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** |
| --- |
| Here we check using the static\_assert method instead to check and print a statement to the console. |
| #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"); |

| **Principles(s):**  **Architect and Design for Security Policies (3):** Implementing tests for testing constant values is a policy that should be maintained throughout development to ensure proper values are maintained. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| CodeSonar | 8.0p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| ÉCLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Handle all exceptions thrown before main() begins executing.  Exceptions are a way of telling us of issues in our program without crashing it entirely. However, these exceptions must be handled early to ensure potential issues are resolved. |

| **Noncompliant Code** |
| --- |
| Here we can see that neither f() nor main() are catching exceptions thrown by throwing\_func(). This will lead to std::terminate() being called. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| In this example, we incorporate a try-catch block to handle ant exceptions thrown. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {  **try** {      f();    } **catch** (...) {      // Handle error    }  } |

| **Principles(s):**  **Architect and Design for Security Policies (3):** Exception handling should be incorporated into the design to prevent any errors that may occur.  **Adopt a Secure Coding Standard (10):** Exceptions should be a common practice for developers and managing them should absolutely be in any security policy. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Potentially-throwing-static-initialization | Partially checked |
| Clang | 3.9 | LANG.STRUCT.EXCP.THROW | Checked by clang-tidy |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-ERR58-a | Exceptions shall be raised only after start-up and before termination of the program |
| RuleChecker | 22.10 | potentially-throwing-static-initialization | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Data Protection | STD-008-JAV | Do not expose private members of an outer class from within a nested class.  Ensuring that the private member stays secure within nested classes will reduce any unintended access and prevent unwanted code from running on your program. |

| **Noncompliant Code** |
| --- |
| In this code, we expose the private (x,y) values in the getPoint() function. The other class (AnotherClass) can also access this data as well. |
| **class** Coordinates {  **private** **int** x;  **private** **int** y;    **public** **class** Point {  **public** **void** getPoint() {        System.out.println("(" + x + "," + y + ")");      }    }  }    **class** AnotherClass {  **public** **static** **void** main(String[] args) {      Coordinates c = **new** Coordinates();      Coordinates.Point p = c.**new** Point();      p.getPoint();    }  } |

| **Compliant Code** |
| --- |
| Using the private access specifier will prevent the outer classes from accessing the methods within. |
| **class** Coordinates {  **private** **int** x;  **private** **int** y;    **private** **class** Point {  **private** **void** getPoint() {        System.out.println("(" + x + "," + y + ")");      }    }  }    **class** AnotherClass {  **public** **static** **void** main(String[] args) {      Coordinates c = **new** Coordinates();      Coordinates.Point p = c.**new** Point();    // Fails to compile      p.getPoint();    }  } |

| **Principles(s):**  **Heed Compiler Warnings (2):** Sometimes, compilers can catch when private members are exposed, allowing for easy remediation of the error.  **Architect and Design for Security Policies (3):** When designing the system, keep in mind how other classes are exposed throughout the program to prevent any issues.  **Default Deny (5):** Default deny relates to the exposure of classes by keeping private members private.  **Adopt a Secure Coding Standard (10):** This is a common oversight and basic principle of programming; it should be implemented in any security policy. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | P8 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.0p0 | JAVA.CLASS.ICSBS | Inner Class Should Be Static (Java) |
| ParaSoft Jtest | 2023.1 | CERT.OBJ08.INNER | Make all member classes “private” |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Data Value | STD-009-CPP | Value-returning functions must return a value from all exit paths.  A function that is supposed to return a value but does not can result in undefined behavior. It also leaves your code vulnerable to attacks from malicious users. |

| **Noncompliant Code** |
| --- |
| Here we did not return the input value for positive input. |
| **int** absolute\_value(**int** a) {  **if** (a < 0) {  **return** -a;    }  } |

| **Compliant Code** |
| --- |
| All paths return a value. |
| **int** absolute\_value(**int** a) {  **if** (a < 0) {  **return** -a;    }  **return** a;  } |

| **Principles(s):**  **Heed Compiler Warnings (2):** Compilers tend to catch issues relating to return values.  **Keep it Simple (4):** Keep syntax similar and don’t make functions unnecessarily complicated or muddied.  **Use Effective Quality Assurance Techniques (9):** When unit testing, make sure that all functions or classes returning data have the proper exit paths.  **Adopt a Secure Coding Standard (10):** This is a common problem and can lead to vulnerabilities from outside sources. Implementing it in your security policy is a good idea. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | P8 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Return-implicit | Fully checked |
| Clang | 3.9 | -Wreturn-type | Does not catch all instances of this rule, such as *function-try-blocks* |
| CodeSonar | 8.0p0 | LANG.STRUCT.MRS | Missing return statement |
| LDRA tool suite | 9.7.1 | 2 D, 36 S | Fully implemented |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory Management. | STD-010-CPP | Do not access freed memory.  Evaluating a pointer into a memory address that has been deallocated by a memory management function can lead to undefined behavior. Dangling pointers are memory that has been deallocated and can result in exploitable vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Here, s is dereferenced after it has been deallocated. This can be exploited to run arbitrary code with granted permissions. |
| #include <stdlib.h>    **struct** node {  **int** value;  **struct** node \*next;  };    **void** free\_list(**struct** node \*head) {  **for** (**struct** node \*p = head; p != NULL; p = p->next) {  **free**(p);    }  } |

| **Compliant Code** |
| --- |
| Here we correct the mistake by storing the reference to p in q before freeing it. |
| #include <stdlib.h>    **struct** node {  **int** value;  **struct** node \*next;  };    **void** free\_list(**struct** node \*head) {  **struct** node \*q;  **for** (**struct** node \*p = head; p != NULL; p = q) {      q = p->next;  **free**(p);    }  } |

| **Principles(s):**  **Heed Compiler Warnings (2):** Accessing freed memory will trigger most compilers to warn you of potential issues.  **Sanitize Data Sent to Other Systems (7)**: Programs should implement proper deletion of data when it is done running.  **Practice Defense in Depth (8):** Encrypting data is a principle of most programming languages. Any data left in the system should be either deleted, stored, or encrypted.  **Adopt a Secure Coding Standard (10):** Programming languages (especially C++) can be prone to memory allocation errors. Programmers should be skilled enough to manage these issues, and it should be included in all policies. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | 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 |
| CodeSonar | 8.0p0 | ALLOC.UAF | Use After Free |
| PC-lint Plus | 1.4 | 449, 2434 | Fully supported |
| TrustInSoft Analyzer | 1.38 | Dangling\_pointer | Exhaustively verified (see [one compliant and one non-compliant example](https://taas.trust-in-soft.com/tsnippet/t/0d556bb8)). |

### 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.

[Insert your written explanations here.]

### 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-CLG | Low | Unlikely | High | P1 | L3 |
| STD-002-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-JAV | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CLG | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Likely | Low | P9 | L2 |
| STD-008-JAV | Medium | Probable | Medium | P8 | L2 |
| STD-009-CPP | Medium | Probable | Medium | P8 | L2 |
| STD-010-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 | Encryption at rest is protecting data while it is in databases or on devices. This kind on encryption should always be implemented wherever data is being stored to safeguard sensitive data and prevent exposure. |
| Encryption at flight | In flight, data can be vulnerable to attack and possibly be altered in the process. To protect data in transit, we must ensure the proper tools are utilized so that data keeps its integrity and no harm from altered or eavesdropped data. Encrypting files before and after transit should always be practiced. |
| Encryption in use | Ensuring that data is valid and secure during processing is encryption in use. This allows data to be utilized with confidence and security to be maintained while in use, preventing unauthorized access. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is one of the first defenses in software security. Validating credentials or user input is a necessity in systems to allow and maintain access to information and content. It also ensures that the right person maintains important systems like making changes to the database rules or structure. |
| Authorization | Controlling access and specifying levels of content users can access is authorization at work. Authorization should be implemented to control user permissions and create role systems like system techs and admins. Using the principle of least privilege as a guideline for implementation, all systems that need to limit access should use authorization. |
| Accounting | Keeping track of user activities and system status to maintain a safe environment and help mediate any errors that arise in accounting. When we audit or implement unit or stress testing on a system to regulate security, we ensure that we have a solid defense against potential threats. It also ensures that we keep with regulation and play our part in the protection of our users and employees. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 01/19/2024 | Added coding principles and examples | Phillip Cabaniss | Professor North |
| 3.0 | 02/08/2024 | Completed document | Phillip Cabaniss | Professor North |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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