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



Green Pace Secure Development Policy \_ Zane Brown

Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

# Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

# Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

# Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

# Module Three Milestone

## Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | When programs allow users to input their own data, we have to make sure to validate the data that is being input into the system. This is because data from outside sources should always be considered untrusted. Outside input data can cause specific vulnerabilities in the system itself. For example, a malicious user can input unrecognized data such as 1=1 in a password field and cause a SQL Injection into the program and cause it to become confused and thus allow the user to enter the system. By validating the input data, developers can avoid such vulnerabilities (Seacord, R. 2018) |
| 1. Heed Compiler Warnings | Heed compiler warnings are a way for developers to eliminate any bugs or errors in their code. There are various levels of these compiler warnings and developers should make sure to address all of them and eliminate any warnings that are present in the program. Developers can do this by modifying their code until the warnings stop. In many cases, a developer can run the program will no issues even if these warnings are there. (Seacord, R. 2018). However, by addressing these warnings at a high level, a developer can avoid and stop any unnecessary vulnerabilities or bugs even before the testing phase begins. This can save an individual, company or user a lot of time and money. |
| 1. Architect and Design for Security Policies | The architect and design for security policies principle is a valuable principle to follow for any developer who is developing a program or system. This principle states that developers should develop software with security policy architecture and design in mind. Software should be set up with interconnected systems that can enforce specific security policies such as specific privileges (Seacord, R. 2018). |
| 1. Keep It Simple | In previous modules, we learned about the difference between the complex secure coding methodology and the simple secure coding methodology. The principle ‘keep it simple’ gives more credit to the simple secure coding methodology. This principle states that a security design for a system should be simple and effective. By creating a simple and small security design, a developer can reduce the amount of possible errors or vulnerabilities contained within the security system. As a system becomes more complex, the possibility of errors and vulnerabilities also increases. (Seacord, R. 2018). |
| 1. Default Deny | This is a simple, yet important principle to follow for developers developing protection schemes involving granting access to a program. This principle states that by default access to a program should be set to deny. A user should not have access to system until the conditions set by the protection scheme are met. (Seacord, R. 2018). By doing this, a developer may avoid any malicious access to a system. |
| 1. Adhere to the Principle of Least Privilege | This is one of the first principles that I learned in computer science. The principle of least privilege should always be considered for programs that deal with different privileges such as read, write, update and delete. This principle states that systems should always execute with the least available privileges for the user. If a user only needs to read in a system to complete their tasks, the read privilege should only be provided. (Seacord, R. 2018). Additional privileges can be accessed if necessary and only for the allotted time to complete the task. This security principle makes sure that malicious users cannot update, delete, etc. anything within the system. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data is similar to validating input data but differs in some ways. What this is saying is that developers need to make sure that data being input through subsystems such as the command shell or relational databases should be checked and sanitized to make sure that no illegal data is being entered. (Seacord, R. 2018). |
| 1. Practice Defense in Depth | The principle of practice defense in depth and the previous ‘keep it simple’ principle seem like they conflict, but they actually work very closely together. The practice defense in depth principle comes from the “castle defense” idea. The goal is to implement several layers of defense in order to make sure that if a layer of security is breached, there will be another layer to stop the attack. (Seacord, R. 2018). Developers should implement several layers of security in their systems in order to have a better chance of preventing any hackers from entering their system. A good example is a system using a firewall, then an email scanner and then a virus protection software. Each of these layers provide their own version of defense and each subsequent layer is implemented for additional defense. Of course, too many layers or too complex layers can provide vulnerabilities and therefore should use the ‘keep it simple’ principle when designing these layers. |
| 1. Use Effective Quality Assurance Techniques | Identifying as many vulnerabilities as possible before finishing a program is ideal for any development team. The ‘use effective quality assurance techniques’ principle aims to do just that. This principle states that developers should implement quality assurance techniques in order to find and eliminate vulnerabilities. These techniques include fuzz testing and penetration testing as well as external reviewers to provide unique input in regards to the implemented security in the system. (Seacord, R. 2018). |
| 1. Adopt a Secure Coding Standard | The final secure coding principle, adopt a secure coding standard, is extremely important for any development team. Developing a secure coding standard is extremely important because it defines the rules, standards, and techniques that the entire team must adhere to. (Seacord, R. 2018). Without these standards, secure code might look very different from individual to individual and thus create problems in the system itself. If any of the secure code conflicts with other secure code, it could be used as a leverage into 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.

**\*\* NOTE: I will be inserting a citation page for all resources used at the end of this document. I was not sure if this was necessary and therefore I am inserting this note here. \*\***

### Coding Standard 1

| **Coding Standard** | **Label** | **Prevent SQL Injection** |
| --- | --- | --- |
| **SQL Injection** | IDS00-J | SQL injections caused when a program accepts outside input into the program. This outside input is usually considered untrusted data. This is because this input data can expose vulnerabilities in the program. If a malicious user enters any type of illegal data into a system (e.g. in a password input object), it could allow the hacker to enter the system, leak information or modify/delete data. An example of this type of illegal data would be ‘1=1’. Developers usually implement ways to sanitize or validate data in order to prevent these SQL Injections. (Mohindra, D. (2021). By adding data validation/sanitation to a system, SQL Injections can be avoided and thus reducing the risk of data manipulation or loss for the company and/or user. |

| **Noncompliant Code** |
| --- |
| The aim of this block of code is to authenticate users into a system. The code is made up of several functions such as the getConnection() function which connects the user to a specific server. The hasPassword() function that takes in the password and hashes it and then the doPriviledgeAction() function that checks for the connection and then checks the users password. However, this program has not been protected against illegal data. This means that a malicious user can enter illegal data such as 1=1 in order to cause a SQL Injection and thus attack the system.  \*\*Note: This noncompliant code and information was provided from the SEI Cert Oracle Coding Standard for Java section of the SEI CERT secure coding link \*\* |
| 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 block of code, which does the same thing as the noncompliant code, is set up much better in order to mitigate the chances of a SQL Injection. This block of code mitigates these chances by first, setting a length limit on the characters that should be entered into the password field. If the number of characters exceeds the limit, the program will handle the error. Additionally, the program uses set() methods to perform strong checking.  \*\*Note: This noncompliant code and information was provided from the SEI Cert Oracle Coding Standard for Java section of the SEI CERT secure coding link \*\* |
| 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):** The principle that maps to SQL Injections would be #7.) **Sanitize data sent to other systems**. This principle states that all data passed into a system should be sanitized. (Seacord, R. 2018). As we know, SQL Injections are caused by malicious users entering unsafe data into a system and thus accessing the internal system. Therefore this principle applies. |

**Threat Level**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

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

**Automation**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and Security errors |
| 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 |
| Parasoft Jtest | 2020.2 | BD-SECURITY-TDSQL | Protect against SQL Injection |

### Coding Standard 2

| **Coding Standard** | **Label** | **Data Type: Implement abstract data types using opaque types** |
| --- | --- | --- |
| **Data Type** | DCL12-C | This coding standard states that developers should implement abstract data types using opaque types. Abstract data types are data types that are hidden from the user via data structure. Opaque data types are private data that allows the program to use the data without having to change anything. This standard is very important because if a specific type of data is not hidden from the user, then they may be able to manipulate any of the fields within the structure (Seacord, R. 2019). |

| **Noncompliant Code** |
| --- |
| The program sets up some functions and string types. According to the SEI CERT C Coding Standard site, these functions and string types are defined using the string\_m.h header file. When looking specifically at the string\_mx type, this is a cause for concern. This is because this data type is not set as a private type. This means that any user will be able to manipulate this data type and cause issues for the program itself.  \*\*Note: This noncompliant code and information was provided from the SEI CERT C Coding Standard section of the SEI CERT secure coding link \*\* |
| 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** |
| --- |
| The solution to the above noncompliant code is fairly simple. The developer will need to take the string\_mx type and make it private. According to the SEI CERT site, this is best done by creating two different header files. One external and one internal. The external header instantiates the string\_mx type from the internal header where the string\_mx type is defined. This keeps the data type private and therefore cannot be manipulated via the user of the program. This will mitigate any problems that involves this specific data type. |
| **External header file:**  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);  /\* ... \*/  **Internal header file:**  struct string\_mx {    size\_t size;    size\_t maxsize;    unsigned char strtype;    char \*cstr;  }; |

|  |
| --- |
| **Principles(s):** The principle that maps to this standard is #9.) **use effective quality assurance techniques** and to a certain extent #6.) **Adhere to the principle of least privilege**. Using effective quality assurance techniques states that quality techniques can be effective in identifying and eliminating vulnerabilities. Additionally, the principle of least privilege states that you should only give a user permission to access functions that will allow them to complete their job and nothing more. In this standard, if a user and/or malicious attacker has access to certain data that should be hidden, it could cause a vulnerability in the system. Therefore these two principles apply. |

**Threat Level**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

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

**Automation**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus | 6.9.0 | CertC-DCL12 | **SEI CERT site provides to description**.  Axivion Bauhaus allows users to check the compliance of the software architecture (Axivion, 2021). |
| LDRA tool suite | 9.7.1 | 104 D | Partially implemented |
| Polyspace Bug Finder | R2020a | CERT C: Rec. DCL12-C | Checks for structure or union object implementation visible in file where pointer to this object is not dereferenced. |
| Parasoft C/C++ test | 2020.2 | CERT\_C-DCL12-a | IF a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden. |

### Coding Standard 3

| **Coding Standard** | **Label** | **String Correctness: Guarantee that storage for strings has sufficient space for character data and the null terminator** |
| --- | --- | --- |
| **String Correctness** | STR50-CPP | This standard is not too complex, but is extremely important. This standard states that developers must ensure that storage allocated for strings must have enough space for character data and null terminator. Strings are of course made up of characters and the end null terminator. If a storage holding the string is not large enough, the string will cause the program to have a buffer overflow. There are a number of ways to fix this issue. Such as, using operators to limit number of acceptable characters or truncation. If this is done, developers can avoid buffer overflow issues. (Pincar, J. 2021) |

| **Noncompliant Code** |
| --- |
| Here, we can see that two different arrays are being created. They are both set to 12. If a user enters more characters than is allowed, this could cause some issues. In the SEI CERT C++ Coding Standard, it states that the first read would fill with a truncated string due to the width() call and the second read could cause a buffer overflow. It is also important to note that the block of code is using char types to take in input. These types are notorious for causing buffer overflows. String types should be used instead. Truncated input could cause users information to be lost or allow malicious users to find a vulnerability. (Pincar, J. 2021).  \*\*Note: This noncompliant code and information was provided from the SEI CERT C++ Coding Standard section of the SEI CERT secure coding link \*\* |
| #include <iostream>    void f() {    char bufOne[12];    char bufTwo[12];    std::cin.width(12);    std::cin >> bufOne;    std::cin >> bufTwo;  } |

| **Compliant Code** |
| --- |
| The solution to the above noncompliant code is to first change the *char* types to *String* types. This is because char types are known to be the source of buffer overflows. Second, the width() statement was taken out in order to stop truncating the input data.   \*\*Note: This noncompliant code and information was provided from the SEI CERT C++ Coding Standard section of the SEI CERT secure coding link \*\* |
| #include <iostream>  #include <string>    void f() {    std::string input;    std::string stringOne, stringTwo;    std::cin >> stringOne >> stringTwo;  } |

|  |
| --- |
| **Principles(s):** The principle that can be mapped to this standard is #1.) **Validate Input** and #2.) **Keep it simple**. Validating input states that all untrusted sources should be validated. Untrusted sources being any external data being input. Keeping it simple states that a simple design is best is most situations because complex designs can easily lead to vulnerabilities in the system. (Seacord, R. 2018). For this standard, we are worried about space and making sure that the program properly checks for this space and acts accordingly if a string is too large. Validating a user’s input and making sure it’s a correct size as well as keeping this checking system simple is why these two principles apply. |

**Threat Level**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

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

**Automation**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | MISC.MEM.NTERM | No space for null terminator  Buffer overrun  Type overrun |
| Klocwork | 2018 | NNTS.MIGHT  NNTS.TAINTED | SEI CERT Site Provides no Description  Integrates with CI/CD tools to find and fix security issues (Perforce, 2021). |
| LDRA tool suite | 9.7.1 | 489 S, 66 X, 70 X, 71 X | Partially implemented |
| Parasoft C/C++ | 2020.2 | CERT\_CPP-STR50-b  CERT\_CPP-STR50-c  CERT\_CPP-STR50-e  CERT\_CPP-STR50-f  CERT\_CPP-STR50-g | Avoid overflow due to reading a not zero terminated string  Avoid overflow when writing to a buffer  Prevent buffer overflow from tainted data  Avoid buffer write overflow from tainted data  Do not use the ‘char’ buffer to store input from ‘std::cin’ |

### Coding Standard 4

| **Coding Standard** | **Label** | **Memory Protection: Properly deallocate dynamically allocated resources** |
| --- | --- | --- |
| **Memory Protection** | MEM51-CPP | This standard is much more difficult than others that we have looked at thus far. This standard states that developers should properly deallocate dynamically allocated resources. Developers should not deallocate anything except for a nullptr or returned value using such as std::free(), global operator delete(), etc. (Ballman, A. 2020). If an incorrect deallocation occurs, it could result in undefined behavior. Therefore, developers should strive to be careful and properly deallocate/allocate resources to avoid any undefined behavior within the system. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, we can see that the program is attempting to free memory. However, undefined behavior occurs because there was nothing to be done here. This is because the variable space that was created in the void f() function was also placed in the *new* operator. Since there was nothing returned via the new() operator, freeing memory using the delete() operator is impossible. (Ballman, A. 2020).  \*\*Note: This noncompliant code and information was provided from the SEI CERT C++ Coding Standard section of the SEI CERT secure coding link \*\* |
| #include <iostream>    struct S {    S() { std::cout << "S::S()" << std::endl; }    ~S() { std::cout << "S::~S()" << std::endl; }  };    void f() {    alignas(struct S) char space[sizeof(struct S)];    S \*s1 = new (&space) S;      // ...      delete s1;  } |

| **Compliant Code** |
| --- |
| As this type of secure coding standard is very complex in its understanding, I had to do a lot more information on this subject. I am still trying to understand how this works, but according to the SEI CERT C++ secure coding standard site, the best way to solve such a memory issue is to completely get rid of the delete operator form the noncompliant code. (Ballman, A. 2020).  \*\*Note: This noncompliant code and information was provided from the SEI CERT C++ Coding Standard section of the SEI CERT secure coding link \*\* |
| #include <iostream>    struct S {    S() { std::cout << "S::S()" << std::endl; }    ~S() { std::cout << "S::~S()" << std::endl; }  };    void f() {    alignas(struct S) char space[sizeof(struct S)];    S \*s1 = new (&space) S;      // ...      s1->~S();  } |

|  |
| --- |
| **Principles(s):** The principle that best applies to this standard is #10.) **Adopt a secure coding standard**. This principle states that a team should develop a secure coding standard. This can help keep all team members on the same page and code similarly to everyone on the team. This is important here because the team who are implementing memory protection should properly deallocate dynamically allocated resources. If a member of the team develops a function that interacts with this memory and doesn’t properly deallocate it, could cause a lot of issues for the entire system. Therefore, this principle applies. |

**Threat Level**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

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

**Automation**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 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 | 6.0p0 | ALLOC.FNH  ALLOC.DF  ALLOC.TM | Free non-heap variable  Double free  Type mismatch |
| LDRA tool suite | 9.7.1 | 232 S, 236 S, 239 S, 407 S, 469 S, 470 S, 483 S, 484 S, 485 S, 64 D, 112 D | Use the same form in corresponding calls to new/malloc and delete/free  Always provide empty brackets ([]) for delete when deallocating arrays  Both copy constructor and copy assignment operator should be declared for classes with nontrivial destructor  Properly deallocate dynamically allocated resources |
| Parasoft Insure++ | N/A | N/A | Runtime detection |

### Coding Standard 5

| **Coding Standard** | **Label** | **Assertions: Use a static assertion to test the value of a constant expression** |
| --- | --- | --- |
| **Assertions** | DCL03-C | This standard uses assertions to test values of constant expressions. This means that if a developer uses the runtime assert() or static\_assert(), they can detect and eliminate certain defects or vulnerabilities in a system. For example, the static\_assert(constant-expression, string-literal) statement can be used to check when the constant-expression is false. If so, the program will output an error message. (Seacord, R. 2018). This is very useful for developers when they need a certain function to always be true. If the function for some reason becomes false, it could cause undefined behavior. The runtime assert() macros can help alleviate this problem. |

| **Noncompliant Code** |
| --- |
| In this block of code, the assert() macro has no specific issues in the code itself. Its job here in the code is to focus on a memory mapping structure. The problem here is that this will only be run once during the launch of the program. Better practice would be to initialize the assert() macro in a function. (Seacord, R. 2018).  \*\*Note: This noncompliant code and information was provided from the SEI CERT C Coding Standard section of the SEI CERT secure coding link \*\* |
| #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 this complaint code solution, a conditional statement is used to check if the constant expression is false. If it is false, then the #error diagnostic message will be output.  \*\*Note: This noncompliant code and information was provided from the SEI CERT C Coding Standard section of the SEI CERT secure coding link \*\* |
| 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 |

|  |
| --- |
| **Principles(s):** The principles that map to this standard are #4.) **Keep it simple**, #9.) **Use effective quality assurance techniques** and #2.) **Heed compiler warnings**. These two principles have been defined in previous standards. For this specific standard, it states that developers should use a static assertion to test the value of a constant expression because it can be used to check when the constant-expression is false. (Seacord, R. 2018). This is a great way to easily check for defined behavior and therefore the keep it simple and effective quality assurance principles apply. |

**Threat Level**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

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

**Automation**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | Misc-static-assert | Checked by clang-tidy |
| CodeSonar | 6.0p0 | (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 (due to all values being known at compile time), then the code should use static-assert instead; this assumes ROSE can recognize macro invocation |
| ÉCLAIR | 1.2 | CC2.DCL03 | Fully implemented |

### Coding Standard 6

| **Coding Standard** | **Label** | **Exceptions: Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | ERR51-CPP | This standard is quite important for programs that deal with thrown exceptions. When a program has an exception that is thrown, it searches for a handler that matches the specific exception type. If not handler is found, the program runs the terminate() call. (Ballman, A. 2020). This can be a problem because the terminate() call causes an abnormal termination. This can cause the program and its resources to stop in an indeterminate state. Therefore, a developer must ensure that all exceptions are caught properly in order to guarantee the program properly unwinds its resources before termination. |

| **Noncompliant Code** |
| --- |
| This noncompliant code is fairly simple about why it is not working properly. We can see here that the functions f() and main() are trying to use the throwing\_func() function. The thowing\_func() function is designed to, as the name suggests, throw an exception. However, there is no handler that matches the specific exceptions that are thrown and therefore the program abnormally terminates.  \*\*Note: This noncompliant code and information was provided from the SEI CERT C++ Coding Standard section of the SEI CERT secure coding link \*\* |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    f();  } |

| **Compliant Code** |
| --- |
| In this compliant code solution example, the same functions are used. However, in the main() function both a *try* and *catch* are used to properly handle the exceptions. This makes sure that the program does not abnormally terminates and ensures that all resources are protected before the program stops.  \*\*Note: This noncompliant code and information was provided from the SEI CERT C++ Coding Standard section of the SEI CERT secure coding link \*\* |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    try {      f();    } catch (...) {      // Handle error    }  } |

|  |
| --- |
| **Principles(s):** The principle that maps to this standard is #2.) **Heed compiler warnings**. This principle states that developers should compile code using the highest warning level available and eliminate warnings by modifying the code. (Seacord, R. 2018). I believe that this is principle maps to this standard very well because these compiler warnings/errors can easily occur when exceptions are not handled well. The program can either crash or create a compiler message that lets the developer know something is wrong. Therefore, this principle applies. |

**Threat Level**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

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

**Automation**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Main-function-catch-all  Early-catch-all | Partially checked |
| RuleChecker | 20.10 | Main-function-catch-all  Early-catch-all | Partially Checked |
| Polyspace Bug Finder | R2020a | CERT C++:ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |

### Coding Standard 7

| **Coding Standard** | **Label** | **Data Value: Ensure that integer conversions do not result in lost or misinterpreted data** |
| --- | --- | --- |
| **Data Value** | INT31-C | This standard dealing with data values states that developers should ensure that integer conversions do not result in lost or misinterpreted data. Regardless of the programming language being used, this is quite important. When an integer is being converted, if done incorrectly, could result in lost data. Developers must pay particular concern to integer operands of pointer arithmetic, specific functions with type *size\_t*, etc. (Seacord, R. 2021). |

| **Noncompliant Code** |
| --- |
| In this code example, we can see that the func() function has an unsigned integer variable u\_a that contains ULONG\_MAX value. There is another variable that is a signed char named sc. The function then attempts to convert the unsigned integer to a signed integer. This is very bad practice because this will result in the unsigned variable to be truncated when converted into the signed variable and thus losing data.  \*\*Note: This noncompliant code and information was provided from the SEI CERT C Coding Standard section of the SEI CERT secure coding link \*\* |
| #include <limits.h>    void func(void) {    unsigned long int u\_a = ULONG\_MAX;    signed char sc;    sc = (signed char)u\_a; /\* Cast eliminates warning \*/    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| The easiest way to solve the above noncompliant code is to validate the ranges before converting the unsigned integer variable into the signed char variable. This will ensure that no truncation or lose of data occurs. (Seacord, R. 2021).  \*\*Note: This noncompliant code and information was provided from the SEI CERT C Coding Standard section of the SEI CERT secure coding link \*\* |
| void func(void) {    unsigned long int u\_a = ULONG\_MAX;    signed char sc;    if (u\_a <= SCHAR\_MAX) {      sc = (signed char)u\_a;  /\* Cast eliminates warning \*/    } else {      /\* Handle error \*/    }  } |

|  |
| --- |
| **Principles(s):** The principle that maps to this standard is #4.) **Keep it simple** and #8.) **practice defense in depth**. Keep it simple has been discussed above. Practice in depth simply states that applying multiple layers of secure code can keep a system protected better (Seacord, R. 2018). Note, all standards that we see in this report can be mapped to the DiD principle. |

**Threat Level**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

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

**Automation**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | N/A | Supported via MISRA C:2012 Rules 10.1, 10.3, 10.4, 10.6 and 10.7 |
| Coverity\* | 2017.07 | NEGATIVE\_RETURNS  REVERSE\_NEGATIVE  MISRA\_CAST | Coverity\* can find array accesses, loop bounds, and other expressions which contain dangerous int conversions.  Can also find when negativity check occurs  Can find instances where int expressions is implicitly converted to a narrower int type |
| Klocwork | 2018 | PRECISION.LOSS  PRECISION.LOSS.CALL | This tool works with C, C++ and C#. It identifies software security and reliability issues (Perforce, 2021). |
| Polyspace Bug Finder | R2020a | CERT C: Rule INT31-C | Polyspace Bug Finder checks for the following:  Integer conversion overflow  Call to memset with unintended value  Sign change integer conversion overflow  Tainted sign change conversion  Unsigned integer conversion overflow |

### Coding Standard 8

| **Coding Standard** | **Label** | **Input Output: Close files when they are no longer needed** |
| --- | --- | --- |
| **Input Output** | FIO51-CPP | This coding standard states that developers should ensure that a file, after being opened, should be closed when it is no longer used. This specific standard uses the C++ example and states that the call std::basic\_filebuf<T>::open(), when used, should also include the std::basic\_filebuf<T>::close(). By using the close() call, the developer can ensure that the file is properly closed. Failure to properly close a file after the program terminates could result in a malicious user exhausting the system resources and potentially causing data to be written into in-memory file buffers. (Pincar, J. 2019). |

| **Noncompliant Code** |
| --- |
| Here, we can see that the function f() is opening a file. However, from the description above, we also know that when the std::fstream opens a file, it must also be closed before terminating the program. This block of code fails to do this. Therefore, when the program terminates, it could create vulnerabilities in the system.  \*\*Note: This noncompliant code and information was provided from the SEI CERT C++ Coding Standard section of the SEI CERT secure coding link \*\* |
| #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** |
| --- |
| To fix the above noncompliant code, all that is needed is to insert the file.close() block of code after the file is no longer needed. This will close the file and thus ensure that there are no issues when the program terminates.  \*\*Note: This noncompliant code and information was provided from the SEI CERT C++ Coding Standard section of the SEI CERT secure coding link \*\* |
| #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();  } |

|  |
| --- |
| **Principles(s):** The principle that best applies to this standard is #10.) **Adopt a secure coding standard** and #1.) **Validate Input**. This principle states that a team should develop a secure coding standard and validate external data coming into the system. (Seacord, R. 2018). This can help keep all team members on the same page and code similarly to everyone on the team. This is important here because the team who are an input/output system will need to properly close files when no longer needed. If a member of the team develops a function that interacts with this input/output system and doesn’t properly close files, it could cause data to be written into memory buffers. Thus, these principles apply. |

**Threat Level**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

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

**Automation**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | ALLOC.LEAK | Leak |
| Klocwork | 2018 | RH.LEAK | **SEI CERT Site Provides no Description**  This tool works with C, C++ and C#. It identifies software security and reliability issues (Perforce, 2021). |
| Parasoft C/C++ test | 2020.2 | CERT\_CPP-FIO51-a | Ensure resource are freed |
| Polyspace Bug Finder | R2020a | CERT C++:FI051-CPP | Checks for resource leak (rule partially covered) |

### Coding Standard 9

| **Coding Standard** | **Label** | **Expressions: Use offsetof() on valid types and members** |
| --- | --- | --- |
| **Expressions** | EXP59-CPP | This standard states that developers should use offsetof() on valid types and members. Offsetof() is a feature that takes two different parameters. Its goal is to evaluate offsets of a specific struct or union type. If a developer passes in the incorrect type or member into offsetof(), it could cause undefined behavior such as data integrity violations or incorrect values and this could be used by a malicious users. (Ballman, A. 2019). |

| **Noncompliant Code** |
| --- |
| According to the SEI CERT C++ Secure Coding site, the noncompliant code below is passing in a illegal type into the offsetof() call. This illegal value is the D() function. This function does not contain the correct standard for offsetof() call. This will eventually cause undefined behavior and possibly cause incorrect values to be output.  \*\*Note: This noncompliant code and information was provided from the SEI CERT C++ Coding Standard section of the SEI CERT secure coding link \*\* |
| #include <cstddef>    struct D {    virtual void f() {}    int i;  };    void f() {    size\_t off = offsetof(D, i);    // ...  } |

| **Compliant Code** |
| --- |
| In order to fix the above noncompliant code, a correct standard is created within the D() function. The struct InnerStandardLayout{} sets up a correct standard for offsetof() call to use. It is then used by inserting both names into the offsetof() call in addition to the i value.  \*\*Note: This noncompliant code and information was provided from the SEI CERT C++ Coding Standard section of the SEI CERT secure coding link \*\* |
| #include <cstddef>    struct D {    virtual void f() {}    struct InnerStandardLayout {      int i;    } inner;  };    void f() {    size\_t off = offsetof(D::InnerStandardLayout, i);    // ...  } |

|  |
| --- |
| **Principles(s):** The principles that map to this standard are #1.) **Validate Input** and #2.) **Heed compiler warnings**. Both principles have already been discussed above. This standard states that developers need to make sure to use valid types and members when using offsetof(). If used incorrectly, it could cause undefined behavior. Therefore, developers should validate data coming into the system as well as heed any compiler warnings that may come up when incorrect types are used in order to avoid these issues. Therefore, these principles apply. |

**Threat Level**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

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

**Automation**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -Winvalid-offsetof | Emits an error diagnostic on invalid member designators, and emits a warning diagnostic on invalid types |
| GCC | 4.9 | -Winvalid-offsetof | Emits an error diagnostic on invalid member designators, and emits a warning diagnostic on invalid types. |
| Parasoft C/C++ Test | 2020.2 | CERT-CPP-EXP59-a | Use offsetof() on valid types and members |
| Polyspace Bug Finder | R2020a | CERT C++:EXP59-CPP | Checks use of offsetof macro with nonstandard layout class (rule fully covered) |

### Coding Standard 10

| **Coding Standard** | **Label** | **Integers: Ensure that unsigned integer operations do not wrap** |
| --- | --- | --- |
| **Integers** | INT30-C | This is one of the first standards that we learned in this class. This standard states that developers should make sure that an unsigned integer operation does not wrap. This means that when an integer value reaches a certain point (usually above the available MAX or MIN), the operation will wrap. This is dangerous because wrapping can cause buffer overflows and thus can be used as a vulnerability via a malicious user. |

| **Noncompliant Code** |
| --- |
| The func() function is taking in two different arguments that will be used in the int type usum variable. Within this variable, the two arguments are added together. For the most part, as long as the values are not too high or low, the program will run normally. However, assuming that all users will use the program correctly is very irresponsible. If a malicious user enters values that exceed a certain limit a wrap will occur. This may cause unexpected behavior such as buffer overflows, allocate insufficient memory for an operation, etc. (Seacord, R. 2020).  \*\*Note: This noncompliant code and information was provided from the SEI CERT C Coding Standard section of the SEI CERT secure coding link \*\* |
| void func(unsigned int ui\_a, unsigned int ui\_b) {    unsigned int usum = ui\_a + ui\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| To fix the above noncompliant code, we can simply insert IF/ELSE statements that will check for certain conditions. These conditional statements will check to see if the entered values exceed certain parameters. If it does, it will handle the error. Else it will add the values together and run normally.  \*\*Note: This noncompliant code and information was provided from the SEI CERT C++ Coding Standard section of the SEI CERT secure coding link \*\* |
| #include <limits.h>    void func(unsigned int ui\_a, unsigned int ui\_b) {    unsigned int usum;    if (UINT\_MAX - ui\_a < ui\_b) {      /\* Handle error \*/    } else {      usum = ui\_a + ui\_b;    }    /\* ... \*/  } |

|  |
| --- |
| **Principles(s):** The principle that best applies to this standard is #10.) **Adopt a secure coding standard**, #1.) **Validate Input** and #4.) **Keep it simple** These principles can help mitigate the issue of wrapping unsigned integers. These principles can help by making sure the entire team follows a secure coding standard, validating any data that is input into the system (In this case integers above MAX or MIN) and keeping it simple in order to avoid any mistakes when dealing with integer wrapping. |

**Threat Level**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

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

**Automation**

**\*\* Information Sited from same source as above description of standard via SEI CERT Coding Site \*\***

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Integer-overflow | Fully checked |
| CodeSonar | 6.0p0 | ALLOC.SIZE.IOFLOW  ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  MISC.MEM.SIZE.ADDOFLOW  MISC.MEM.SIZE.BAD  MISC.MEM.SIZE.MULOFLOW  MISC.MEM.SIZE.SUBUFLOW | Addition overflow of allocation size  Integer overflow of allocation size  Multiplication overflow of allocation size  Subtraction underflow of allocation size  Addition overflow of size  Unreasonable size argument  Multiplication overflow of size  Subtraction underflow of size. |
| Coverity | 2017.07 | INTEGER\_OVERFLOW | Implemented |
| LDRA tool suite | 9.7.1 | 493 S, 494 S | Partially Implemented |

## 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.

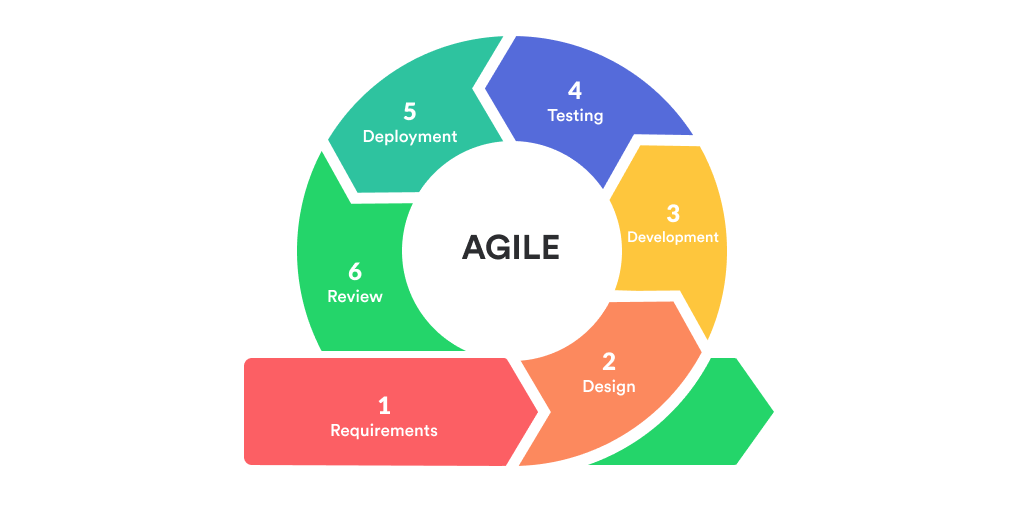


Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

The above image is a representation of Green Pace’s DevOps processes and infrastructure. This established cycle provides a model for the enforcement of and compliance to secure coding standards. In the first phase we can see pre-production. Here, the development team must Assess and plan, design, build and verify and test. The assessing and planning stage revolves around regulatory changes, impact analysis, prioritize backlog, etc. Design aims to do test-driven design and implement app best practices. Building refers to secure building, using trusted repositories and source usage. Finally, verifying and testing’s goals are to scan for vulnerabilities, do security testing and more.

In the next phase, we can see the production. Here, the development team does transition and health checks, monitors and detects, responds and maintains/stabilizes. In the transition and health check phase, configuration and deployment and security settings are established. Monitoring and detection aim to log collections, focus on analytics, event alerting and intrusion detection. The respond section of this phase blocks attacks and turns off services. Finally maintaining and stabilizing strives to assess against security baseline, returns to a baseline and returns to stable state after attack.

The above DevSecOps diagram establishes a cycle like development model for implementing and automating secure coding standards. This lifecycle model can also be compared with the agile development model that has become increasingly popular throughout the years. We can see the similarities in the below diagram.



(Anurina, O. 2021)

The agile development model is usually done through a series of sprints until the product is finished. Similarly, the above DevSecOps secure coding development model can also be implemented within the agile model. The beginning of a sprint usually starts with the requirements and design of the product. During this time, Green Pace’s pre-production phase of their model can begin as well. Here, possible threats and the coding standards that we discussed earlier in this paper can start to be established and designed. Once the standards have been established, development may also start. During the development phase, the product will be built. In conjunction with the product production, development and implementation of the established coding standards can also begin. Once the specific function or goal has been met for both the product sprint and DevSecOps sprint, test can begin. Testing the product with the secure code should be done at the same time. Testing both the product and implemented security can help developers find any bugs or vulnerabilities in the system when it is run. This can help mitigate any vulnerabilities in the system that may be present if testing was left to the end.

Once a product has been shipped, the development team is far from over. A review phase begins, and the team monitors the product to make sure than users are not finding any unexpected bugs, errors, or vulnerabilities. If found, the team can implement a patch to fix these issues. The DevSecOps cycle also has steps for this within the ‘production’ phase. We can see that measures are taken once an attack happens or a system is compromised. Therefore, these two phases in the models can be implemented at the same time.

The agile development model and DevSecOps model were used in conjunction with each other because they both provide similar steps to achieve their goals. Because the agile development model works though multiple sprints in order to finish the product, it would be best to also implement these secure coding practices during the agile development model. DevSecOps model’s steps almost match up perfectly with the steps in agile, therefore, the implementation of these two models should not prove to be much of an issue.

It must be noted that these steps may need to take more time in certain phases than others. This also depends on the type of product being made. For example, the above 10 secure coding standards were:

* Prevent SQL Injection
* Implement abstract data types using opaque types
* Guarantee that storage for strings has sufficient space for character data and the null terminator
* Properly deallocate dynamically allocated resources
* Use a static assertion to test the value of a constant expression
* Handle all exceptions
* Ensure that integer conversions do not result in lost or misinterpreted data
* Close files when they are no longer needed
* Use offsetof() on valid types and members
* Ensure that unsigned integer operations do not wrap

Many of these standards vary in their uses and multiple standards may be used in one program. Some of these standards may not be used in a program at all. This is where ‘Assess and Plan’ and ‘Design’ phases of the DevSecOps model play a major role. Once the planning and design is finished and developers know which standards to apply, the building of the app and implementation of the standards can take place. Testing is then done for the implemented security. Once these pre-production steps have taken place, production will then take care of the monitoring, responding and maintaining of any future security measures. Additional security coding practices may need to be implemented to the same program if a vulnerability is found.

## Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| IDS00-J | High | Probable | Medium | P12 | L1 |
| DCL12-C | Low | Unlikely | High | P1 | L3 |
| STR50-CPP | High | Likely | Medium | P18 | L1 |
| MEM51-CPP | High | Likely | Medium | P18 | L1 |
| DCL03-C | Low | Unlikely | High | P1 | L3 |
| ERR51-CPP | Low | Probable | Medium | P4 | L3 |
| INT31-C | High | Probable | High | P6 | L2 |
| FIO51-CPP | Medium | Unlikely | Medium | P4 | L3 |
| EXP59-CPP | Medium | Unlikely | Medium | P4 | L3 |
| INT30-C | High | Likely | High | P9 | L2 |

## Create Policies for Encryption and Triple A

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | The main goal for encryption in rest is to protect data that is stored in a system. If a malicious user enters a system, they may be able to easily steal data from the user and/or company. Encryption in rest mitigates this issue by encrypting data on disk. If a malicious user is able to steal a physical hard drive, they will not be able to access the data due to the encryption. (Azure, 2020).  Encryption in rest is a very important secure coding practice that should be common in most workplaces. This provides a good way to deal with any malicious users within the company itself. If someone aims to steal the data, they will need encryption keys to do so. These keys would more than likely be spread out between different staff members and thus causing the malicious user to be unable to do anything. |
| Encryption at flight | Encryption at flight is another practice in which a company and/or user implements a system in which data that is being transmitted is encrypted in order to prevent data theft. Just like encryption at rest, encryption at flight should be a common practice for anyone who wishes to protect their data. (Quantum, 2020). Malicious users may be able to intercept data that is being transmitted and therefore steal it and use it for their own gain. Encrypting this data will mean that these malicious users have to beat the encryption in order to get to the data. For the most part, hackers will not waste their time trying to do this. |
| Encryption in use | Encryption in use is essentially the ability to run encrypted application or transform data for use. (StackOverflow, 2018). A good example of encryption in use would be protecting a user’s password through hashing. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying who is the individual trying to access the server or internal network. Authentication can be done through a number of ways. For example, authentication can be achieved through username/password logins, SSO systems, biometrics, digital certificates and key infrastructures. (O’Carroll, B. 2018). The authentication process can be seen as the first layer of defense for the system. This layer of defense is extremely important and will need to become increasingly more complex the more advanced technology becomes. Authentication can have some vulnerabilities, however. For example, username/password authentication systems can be ‘tricked’ by SQL injection attacks. Therefore, authentication systems also need to be pared with secure coding to help alleviate any possible vulnerabilities. |
| Authorization | Authorization is the second layer of defense in this triple A defense model. Once a user enters their credentials and is successful authenticated, the system will then assign that user a specific level of authorization. This authorization is given to a user through a number of ways. Such as location restrictions, time restrictions, frequency of logins, IP address filtering, etc. (O’Carroll, B. 2018). These levels of authorization allow users to do only a certain number of things in the system. For example, an employee may only have read permissions and therefore cannot write to programs in the internal network. Whereas administrators may have read-write permissions that allow them to do significantly more than the lower-level employee. This is extremely important because a good authorization system can block malicious users from obtaining or manipulating data found in the system. |
| Accounting | Accounting is the final layer of defense in the Triple A secure coding model. Accounting is the process that records what users are doing in the network. Accounting records resources used, session statistics, session duration and data, trend analysis, etc. (O’Carroll, B. 2018). This final layer of defense allows administrators to see what each individual who gained access to the network is doing or has done during their time logged into the network. This is extremely important because having a log of user activity can help administrators see any suspicious activity or help them discover if an attack had occurred. |

## 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.

**\*\* These answers can be found below the coding standards under the *principles* tab \*\***

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

# Audit Controls and Management

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

Evidence will include the following:

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

# Enforcement

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

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

# Exceptions Process

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

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

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

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

# Distribution

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

# Policy Change Control

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

# Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 03/17/2021 | Module 3 | Zane Brown | Andujar Dewin |
| 3.0 | 4/10/2021 | Module 6 | Zane Brown | Andujar Dewin |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

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

Resources:

Anurina, O. (2019, May 10). Agile sdlc: How your project can benefit from this model. Retrieved April 11, 2021, from <https://mlsdev.com/blog/agile-sdlc>

Ballman, A. (Sep 25, 2020). Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions>

Ballman, A. (Oct 21, 2019). Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP59-CPP.+Use+offsetof%28%29+on+valid+types+and+members>

Ballman, A. (Sep 22, 2020). Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM51-CPP.+Properly+deallocate+dynamically+allocated+resources>

O'Carroll, B. (2018, November 27). What is AAA Security? An introduction to authentication, authorisation and accounting. Retrieved April 10, 2021, from <https://codebots.com/application-security/aaa-security-an-introduction-to-authentication-authorisation-accounting>

Pincar, J. (Oct 21, 2019). Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO51-CPP.+Close+files+when+they+are+no+longer+needed>

Pincar, J. (Feb 23, 2021). Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator>

Mike BMike (2018, March 27). What's a practical example of encryption "in use" or "in process"? Retrieved April 11, 2021, from <https://security.stackexchange.com/questions/182398/whats-a-practical-example-of-encryption-in-use-or-in-process>

Msmbaldwin. (n.d.). Azure data encryption-at-rest - azure security. Retrieved April 11, 2021, from <https://docs.microsoft.com/en-us/azure/security/fundamentals/encryption-atrest>

Mohindra, D. (Feb 26, 2021). Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection>

Seacord, R. (Dec 11, 2018). Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/c/DCL03-C.+Use+a+static+assertion+to+test+the+value+of+a+constant+expression>

Seacord, R. (Aug 30, 2019). Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/c/DCL12-C.+Implement+abstract+data+types+using+opaque+types>

Seacord, R. (Aug 30, 2019). Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/c/INT30-C.+Ensure+that+unsigned+integer+operations+do+not+wrap>

Static code analysis for C, C++, C#, and Java. (n.d.). Retrieved April 11, 2021, from <https://www.perforce.com/products/klocwork>

Technology leader for static code analysis. (2021, March 26). Retrieved April 11, 2021, from <https://www.axivion.com/en/>

Quantum. (n.d.). Encryption-in-Flight. Retrieved April 11, 2021, from <https://www.quantum.com/resources/glossary/encryption-in-flight>