Green Pace

Green Pace Secure Development Policy

Contents

Green Pace

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.

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. Validate Input Data	In the realm of secure coding standards, input validation remains paramount. This imperative underscores the necessity that any data received and utilized by a program across a trust boundary requires validation. Robust validation mechanisms, encompassing a variety of techniques, should be utilized to prevent vulnerabilities and exploits, such as buffer overflow, SQL injection, and XSS attacks. Deploying adequate validation protocols is pivotal to safeguarding systems against malicious attack and vulnerability.
2. Heed Compiler Warnings	In the pursuit of secure coding practices an active acknowledgement of compile warnings assumes a pivotal role. To best fortify code against vulnerability, it is important to compile code using the highest warning level available, and to eliminate warnings via code refactoring. Additionally, it is heavily advised to employ dynamic and static analysis tools to further detect and eliminate vulnerabilities.
3. Architect and Design for Security Policies	For the sake of heightened security, software architecture and design should enforce security principles. As an example, a web application that handles sensitive user data should be built from the ground-up with consideration of how to handle authentication, encryption, etc. This better ensures the adoption of Defense in Depth (DiD), and that security is not an afterthought or 'patchwork' solution. By considering security from inception, a developer reduces the risk of vulnerability and will ultimately create a more robust system.
4. Keep It Simple	For the sake of security and efficiency, keeping a design as simple and small as possible remains an important tenet in the principles of security. Complex designs and implementations increase the likelihood that errors and vulnerabilities will be introduced in implementation, configuration, and use. In addition to the increased likelihood of vulnerabilities being introduced, as a program becomes more complex, it subsequently becomes increasingly difficult to attain adequate assurance and testing coverage.

Principles	Write a short paragraph explaining each of the 10 principles of security.
5. Default Deny	In the realm of security, the principle of default denial stands as a fundamental strategy. It is imperative to base access decisions on permission rather than exclusion. By default, access is denied until such conditions are met in which access is permitted. By adopting default deny, a developer can establish a proactive defense, reducing attack surface and minimizing vulnerabilities.
6. Adhere to the Principle of Least Privilege	Echoing the philosophy behind the principle of default deny, the principle of least privilege asserts that processes should execute with the least level of privileges necessary for the task. Additionally, any elevated permissions should be held for only the required duration. This principle serves to further reduce the attack surface of the program and help prevent malicious attacks.
7. Sanitize Data Sent to and Received by Other Systems	In order to adhere to secure coding practices, it is imperative to sanitize data before transmitting to external systems. This includes but is not limited to command shells, databases (dbs), and OTC components. By ensuring the information shared with other systems is sanitized, developers help to mitigate the risk of injection attack and unauthorized manipulations. Sanitizing data is a crucial link in the chain of providing DiD.
8. Practice Defense in Depth	In the world of security, no single solution provides a 'cover-all', it remains imperative to practice DiD. Managing risk with multiple defensive strategies is essential to providing a robust and effective security atmosphere. DiD ensures that if one layer of security fails, there exists one or more subsequent layers of security to mitigate or prevent the attack or vulnerability.
9. Use Effective Quality Assurance Techniques	In the pursuit of robust security, effective quality assurance techniques are an indispensable means of verifying the absence of vulnerabilities beyond the development phase. Throughout the course of development, individual test cases as well as static and dynamic testing can be utilized to ensure component functionality. Techniques like penetration testing and code auditing can be performed upon completion to prevent the propagation and continuance of vulnerabilities. A comprehensive and well-thought-out quality assurance strategy ensures the identification and mitigation of vulnerabilities, contributing to system resilience.
10. Adopt a Secure Coding Standard	The development or adoption of a secure coding standard is a fundamental practice in shielding software from potential threats. A robust coding standard addresses aspects like input validation, secure data storage, and error handling. This practice both contributes to the security and resilience of an application, and remains fundamental in secure coding.

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-CPP]	This coding standard aims to establish consistent and secure practices for handling data types. Adhering to this standard will reduce the risk of vulnerabilities and exploits as well as enhance code quality. 1. **Choose Variables Appropriately**: Choose the correct data type (int, double, string…) based on the requirements, including correct sizing (short, long, signed, unsigned). This will help ensure efficient memory usage and prevent unnecessary over/underflow. 2. **Implement Consistent Naming Conventions**: Implement and abide by a consistent naming convention. This will help ensure code readability making it easier for subsequent developers to analyze and refactor the code. This will ultimately boost efficiency. 3. **Ensure Conversions do not Result in Loss of Data:** It is important to remain mindful of data type conversions to avoid loss of precision or data. Ensure that the destination data type can accommodate the values being converted. This will help prevent vulnerability and unexpected behavior. **4.** **Initialize Variables Before Use:** Always initialize variables before use to avoid unpredictable behavior and potential vulnerability. **5.** **Validate User Input:** Validate and sanitize user input to ensure conformity with expected data types. This will help prevent vulnerability and unexpected behavior. **6.** **Avoid Hardcoding Magic Numbers:** Refrain from using unexplained numeric literals in code. Use named constants or variables instead. This will help increase code clarity and readability, ultimately boosting efficiency.

Noncompliant Code

The following code demonstrates improper adherence to the standard. The declared short int does not properly contain the number 50,000 and will cause a buffer overflow.
// Rule 1: Choose Variables Appropriately unsigned short int incorrectSize = 50000; // Violates rule by using an invalid size
The following code demonstrates improper adherence to the standard. The example below shows a naming convention that would be in violation of this standard.
// Rule 2: Implement Consistent Naming Conventions int DIFFERENT_NaMiNg_convention = 10; // Violates rule by using inconsistent naming conventions
The following code demonstrates improper adherence to the standard. The conversion utilized in the example causes potential loss of data thus violating the principle.
// Rule 3: Ensure Conversions do not Result in Loss of Data

Noncompliant Code

```
int intValue = 1000;
char charValue = static_cast<char>(intValue);  // Violates rule by potentially losing data during conversion
```

The following code demonstrates improper adherence to the standard. The example below attempts to utilize an uninitiated variable.

```
// Rule 4: Initialize Variables Before Use
cout << "Uninitialized Variable: " << uninitializedVariable << endl;  // Violates rule by using uninitialized variable
```

The following code demonstrates improper adherence to the standard. The following example does not include any input validation and thus violates the principle.

```
// Rule 5: Validate User Input
int userInput;
cout << "Enter a number: ";
cin >> userInput;  // Violates rule by not validating user input
```

The following code demonstrates improper adherence to the standard. Rather than declaring a variable for pi, magic numbers are utilized directly in the code, violating the principle.

```
// Rule 6: Avoid Hardcoding Magic Numbers
double area = 3.1415 * 5 * 5;  // Violates rule by using magic number 3.1415 directly in the code
```

Compliant Code

The following code demonstrates proper adherence to the standard. In the example below, the variable is resized to accommodate the number.

```
    // Rule 1: Choose Variables Appropriately
    unsigned int appropriateSize = 50000;  // Uses an appropriate size
```

The following code demonstrates proper adherence to the standard. The naming convention in the example below has been normalized and now adheres to the principle.

```
    // Rule 2: Implement Consistent Naming Conventions
    int differentNamingConvention = 10;  // Fixed to use a consistent naming convention
```

The following code demonstrates proper adherence to the standard. This example utilizes two types that will guarantee no loss of data. Correct implementation will depend on specific instances.

```
    // Rule 3: Ensure Conversions do not Result in Loss of Data
    int intValue = 42;
    double doubleValue = static_cast<double>(intValue);  // Ensures no data loss during conversion
    // Please note the responsibility to avoid data loss in specific scenarios falls upon the developer
```

The following code demonstrates proper adherence to the standard. This example properly initializes the variable before usage.

Compliant Code

```
// Rule 4: Initialize Variables Before Use
int initializedVariable = 0;  // Initializing variable before use
cout << "Initialized Variable: " << initializedVariable << endl;
```

The following code demonstrates proper adherence to the standard. The example below includes a form of input validation, specific implementation will vary based on needs.

```
// Rule 5: Validate User Input
int userInput;
cout << "Enter a number: ";
cin >> userInput;
if (cin.fail()) { // Validate user input (example)
    cout << "Invalid input." << endl;
    return 1;
}
```

The following code demonstrates proper adherence to the standard. The example below utilizes a named constant rather than a magic number.

```
// Rule 6: Avoid Hardcoding Magic Numbers
const double PI = 3.1415;
double radius = 5;
double area = PI * radius * radius;  // Using a named constant instead of a magic number
cout << "Area: " << area << endl;
```

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

Standard: Data Type
Principles(s):

1. **Validate Input Data:** Validating input is an important principle to be aware of and adhere to when concerning the standard Data Type. Ensuring to practice input validation will help to align the developer and their work with best security practices, mitigate unexpected behavior and vulnerability, as well as bolster a robust DiD system. Specifically concerning Data Type, this will help the developer avoid loss of data, and mitigate potential attacks.
2. **Heed Compiler Warnings:** Paying close attention to compiler warnings is important for most every standard, including Data Type. Addressing compiler warnings is critical to addressing potential errors and vulnerabilities present in code. Improper handling or neglecting of warnings can lead to unexpected behavior, vulnerability, and improper alignment with the standard and secure principles. As illustrated in the examples above, there are many improper handlings of Data Type, and developers should be mindful to heed compiler warnings.
3. **Architect for Security:** Developers should remain cognisant of the principle of architect and design for security policies when handling Data Type. Building a robust codebase from the ground-up with common vulnerabilities in mind and mitigation strategies implemented throughout development will result in a more layered and secure application. Developers should be mindful to adhere to this principle regardless of the standard.
4. **Keep it Simple:** Developers should be mindful of the principle of keeping it simple. This mundane statement, if considered adequately, can help mitigate vulnerability and error. By practicing the principle of simplicity, a developer can ensure alignment with the principle, secure standard set in this document, as well as help enforce their understanding of the principles of least privilege and default deny. Applying this to Data Type will help a developer ensure mitigation of incompliant and vulnerable code.
5. **Utilize Quality Assurance:** Utilizing robust quality assurance methods is critical throughout development, reinforcing the principle of design for security. Developers should employ a range of tailored tools to mitigate and remove as many vulnerabilities as possible throughout development. For a list of recommended tools relating to Data Type, specifically, please see the 'Automation' section below.
6. **Adopt Secure Standard:** Adopting and maintaining a secure standard is important throughout development. In relation to Data Type, this will help the developer to avoid incompliant code, as illustrated above, as well as adhere to best practices and allow efficient work within a team. By adopting and applying a secure standard one can ensure the most efficient, secure development possible.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Unlikely	Medium	High	2

Automation

Tool	Version	Checker	Description Tool
CPPCheck	2.12	Comprehensive Static Analysis	CPPCheck can perform a comprehensive static analysis when all checks are turned on. This will check for a variety of violations and style errors in C++ code. [https://cppcheck.sourceforge.io/]
Microsoft/MSVS Code Analysis	Visual Studio 17.8.2	Static Analysis	Microsoft Visual Studios comes with a built-in static analysis tool that should be enabled in the project

Tool	Version	Checker	Description Tool
			properties for further coverage. https://learn.microsoft.com/en-us/cpp/build/reference/analyze-code-analysis?view=msvc-170
Clang-Tidy	18.0.0	Static Analysis	Clang-Tidy is a static analysis tool that can check for a variety of errors. For implementation in C++ see the following link. This tool can be used in conjunction with the previous. https://chat.openai.com/c/26243feb-b3a1-418f-9ef2-de1c3d2c60f5

Coding Standard 2

Coding Standard	Label	Name of Standard
Data Value	[STD-002-CPP]	This coding standard aims to establish consistent and secure practices for handling data value. Adhering to this standard will reduce the risk of vulnerabilities and exploits as well as enhance code quality. Some overlap between this standard and the previous is to be expected. 1. **Use Descriptive Constants**: A reiteration of the previous 'Avoid Magic Numbers' rule, employ named constants following proper naming convention instead of raw numeric literals. 2. **Choose Data Types Appropriately**: Choose the correct data type (int, double, string…) based on the requirements, including correct sizing (short, long, signed, unsigned). This will help ensure efficient memory usage and prevent unnecessary over/underflow. 3. **Ensure Conversions do not Result in Loss of Data:** It is important to remain mindful of data type conversions to avoid loss of precision or data. Ensure that the destination data type can accommodate the values being converted. This will help prevent vulnerability and unexpected behavior. **4.** **Initialize Variables Before Use:** Always initialize variables before use to avoid unpredictable behavior and potential vulnerability. **5.** **Validate User Input:** Validate and sanitize user input to ensure conformity with expected data types. This will help prevent vulnerability and unexpected behavior. **6.** **Ensure Operations Do Not Result in Divide-By-Zero Error:** Do not permit operations that result in a divide-by-zero error. This can lead to unpredictable behavior, runtime errors, crashes, and vulnerabilities. It is heavily advised to implement checks to ensure denominators are non-zero prior to performing division operations. This will enhance code robustness and prevent a host of potential errors and vulnerabilities.

Noncompliant Code

The following code demonstrates improper adherence to the standard. Rather than declaring a variable for pi a magic number is utilized violating the principle.

```
// Rule 1: Use Descriptive Constants
double area = 3.1415 * 5 * 5;  // Violates rule by using magic number 3.1415 directly in the code…
                               // use a variable instead.
```

The following code demonstrates improper adherence to the standard.

```
// Rule 2: Choose Data Types Appropriately
unsigned short int incorrectSize = 100000;  // Violates rule by using an invalid size
```

The following code demonstrates improper adherence to the standard. The example below may lose data during conversion in violation of the principle.

Noncompliant Code

```
// Rule 3: Ensure Conversions do not Result in Loss of Data
int intValue = 1000;
char charValue = static_cast<char>(intValue);  // Violates rule by potentially losing data during conversion
```

The following code demonstrates improper adherence to the standard. The example below attempts to utilize an uninitialized variable in violation of the principle.

```
// Rule 4: Initialize Variables Before Use
cout << "Uninitialized Variable: " << uninitializedVariable << endl;  // Violates rule by using uninitialized variable
```

The following code demonstrates improper adherence to the standard. The following example contains no user input validation in violation of the principle.

```
// Rule 5: Validate User Input
string userInput;
cout << "Enter a word: ";
cin >> userInput;  // Violates rule by not validating user input
```

The following code demonstrates improper adherence to the standard. The following example attempts to divide by 0, leading to an undefined error. This is in violation of the principle.

```
// Rule 6: Avoid Divide-By-Zero Error
 int numerator = 10;
 int denominator = 0;
 int result = numerator/denominator; // Violates rule as denominator is 0
 cout << “Result: “ << result << endl;
```

Compliant Code

The following code demonstrates proper adherence to the standard. The example below utilizes a named constant rather than a magic number.

```
// Rule 1: Use Descriptive Constants
const double PI = 3.1415;
double radius = 5;
double area = PI * radius * radius;  // Using a named constant instead of a magic number
cout << "Area: " << area << endl;
```

The following code demonstrates proper adherence to the standard. In the example below, the variable is resized to accommodate the number.

```
// Rule 2: Choose Data Types Appropriately
unsigned int correctSize= 10;  // Amends rule by using a valid size
```

The following code demonstrates proper adherence to the standard. This example utilizes two types that will guarantee no loss of data. Correct implementation will depend on specific instances.

```
// Rule 3: Ensure Conversions do not Result in Loss of Data
```

Compliant Code

```
    int intValue = 24;
    double doubleValue = static_cast<double>(intValue);  // Ensures no data loss during conversion
    // Please note the responsibility to avoid data loss in specific scenarios falls upon the developer
```

The following code demonstrates proper adherence to the standard. This example properly initializes the variable before usage.

```
    // Rule 4: Initialize Variables Before Use
    string initializedVariable = “IT IS INITIALIZEEEED”;  // Initializing variable before use
    cout << "Dr. Initialize says: " << initializedVariable << endl;
```

The following code demonstrates proper adherence to the standard. The example below includes a form of input validation, specific implementation will vary based on needs.

```
   // Rule 5: Validate User Input
/* This implementation is valid, as an example it checks for numbers present in the entered string.)
    string userInput;
    cout << "Enter a word: ";
    getline(cin, userInput);  // Read the entire line

    // Check if userInput contains any numeric characters
    bool containsNumbers = false;
    for (char c : userInput) {
        if (isdigit(c)) {
            containsNumbers = true;
            break;
        }
    }
    if (containsNumbers) {
        cout << "Invalid input. Please enter a word without numbers." << endl;
    } else {
        cout << "Valid input: " << userInput << endl;
    }
```

The following code demonstrates proper adherence to the standard. The example below does not divide by 0 and will therefore not prompt an undefined error.

```
    // Rule 6: Avoid Divide-By-Zero Error
     int numerator = 10;
     int denominator = 2; // Ensure a non-zero denominator
     int result = numerator/denominator; // Amends rule as denominator is 2
     cout << “Result: “ << result << endl;

    return 0;
}
```

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

Standard: Data Value
Principles(s):

1. **Validate Input Data:** Validating input is an important principle to be aware of and adhere to when concerning the standard Data Value. Ensuring to practice input validation will help to align the developer and their work with best security practices, mitigate unexpected behavior and vulnerability, as well as bolster a robust DiD system. Specifically, when concerning Data Value this will help the developer avoid a variety of attacks including injection attacks.
2. **Heed Compiler Warnings:** Paying close attention to compiler warnings is important for most every standard, including Data Value. Addressing compiler warnings is critical to addressing potential errors and vulnerabilities present in code. Improper handling or neglecting of warnings can lead to unexpected behavior, vulnerability, and improper alignment with the standard and secure principles. As data value manipulation is a common avenue of attack, it is especially important for developers to heed all compiler warnings.
3. **Architect for Security:** Developers should remain cognisant of the principle of architect and design for security policies when handling Data Value. Building a robust codebase from the ground-up with common vulnerabilities in mind and mitigation strategies implemented throughout development will result in a more layered and secure application. Developers should be mindful to adhere to this principle regardless of the standard.
4. **Sanitize Data Sent:** When data is being transmitted, it is imperative to apply data sanitization. In order to adhere to best security policies and protect the integrity of data, validating data sent and received is essential for maintaining a healthy, secure system. This is particularly true when the data is sensitive in nature, ensuring proper validation and cleansing will support robust DiD.
5. **Utilize Quality Assurance:** Utilizing robust quality assurance methods is critical throughout development, reinforcing the principle of design for security. Developers should employ a range of tailored tools to mitigate and remove as many vulnerabilities as possible throughout development. For a list of recommended tools relating to Data Value specifically, please see the ‘Automation’ section below.
6. **Adopt a Secure Standard:** Adopting and maintaining a secure standard is important throughout development. In relation to Data Value, this will help the developer to avoid noncompliant code, as well as adhere to best practices and allow efficient work within a team. By adopting and applying a secure standard one can ensure the most efficient, secure development possible.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium - High	High	3

Automation

Tool	Version	Checker	Description Tool
CPPCheck	2.12	Comprehensive Static Analysis	CPPCheck can perform a comprehensive static analysis when all checks are turned on. This will check for a variety of violations and style errors in C++ code. [https://cppcheck.sourceforge.io/]
Microsoft/MSVS Code Analysis	Visual Studio 17.8.2	Static Analysis	Microsoft Visual Studios comes with a built-in static analysis tool that should be enabled in the project properties for further coverage. To

Tool	Version	Checker	Description Tool
			see extended usage see link below. https://learn.microsoft.com/en-us/cpp/build/reference/analyze-code-analysis?view=msvc-170
Clang-Tidy	18.0.0	Static Analysis	Clang-Tidy is a static analysis tool that can check for a variety of errors. For implementation in C++ see the following link. This tool can be used in conjunction with the previous. https://chat.openai.com/c/26243feb-b3a1-418f-9ef2-de1c3d2c60f5

Coding Standard 3

Coding Standard	Label	Name of Standard
String Correctness	[STD-003-CPP]	This coding standard aims to establish consistent and secure practices for handling strings. Adhering to this standard will reduce the risk of vulnerabilities and exploits as well as enhance code quality. **1. Allocate Adequate String Storage Space for Char Data and Null Terminator:** It is essential when allocating memory for strings that there is enough space to accommodate the character data AND the null terminator. This will help prevent buffer overflows and ensure the integrity of string operations. **2. Do Not Create a String from a Null Pointer:** Strings must not be created from null pointers, as this can lead to undefined behavior or runtime errors. Developers should check for null pointers prior to string operations to ensure a more predictable and secure code execution. **3. Use Valid References and Pointers:** Validate references and pointers prior to utilizing them in string operations. This can help ensure that only valid memory locations are accessed as well as reduce the risk of null pointer dereference. **4. Range Check Element Access:** Attempting to modify out-of-range objects can result in undefined behavior and should be avoided. Before accessing the elements of a string, performing a range check can help ensure that the index is within the bounds of the string. This will help prevent buffer overflows and undefined behavior. **5. Do Not Modify String Literals:** As string literals are often stored in read-only memory and are immutable, modifying them can lead to undefined behavior and compromise code stability. 6. **Ensure Conversions do not Result in Loss of Data:** In cases where character data can be interpreted as negative, signed character data must be converted into unsigned character data before being assigned or converted to a larger signed type. This rule emphasizes preserving data integrity during conversions. **7. Understand Narrow vs. Wide Strings and Functions:** Passing narrow string arguments to wide string functions and vice-versa can lead to unexpected and undefined behavior. Understanding these distinctions can help ensure compatibility and proper handling in different contexts. **8. Validate User Input:** Validate and sanitize user input to ensure conformity with expected data types. This will help prevent vulnerability and unexpected behavior.

Noncompliant Code

The following code demonstrates improper adherence to the standard. The following example did not leave space for the null terminator in violation of the principle.

```
// Rule 1: Allocate Adequate String Storage Space for Char Data and Null Terminator
```

Noncompliant Code

```
char insufficientBuffer[5];
strcpy(insufficientBuffer, "Hello");  // Violates rule by not allocating enough space for the null terminator
```

The following code demonstrates improper adherence to the standard. The following example attempts to create a string from a null pointer, this violates the principle.

```
// Rule 2: Do Not Create a String from a Null Pointer
char* nullPointer = nullptr;
char* stringFromNull = strdup(nullPointer);  // Violates rule by creating a string from a null pointer
```

The following code demonstrates improper adherence to the standard. The following example attempts to utilize a null pointer in string operations. This is in violation of the principle.

```
// Rule 3: Use Valid References and Pointers
char* invalidPointer = nullptr;
strcpy(invalidPointer, "Invalid");  // Violates rule by using a null pointer in string operations
```

The following code demonstrates improper adherence to the standard. The example below attempts to access an element that is out of range, creating an error and violating the principle.

```
// Rule 4: Range Check Element Access
char myString[] = "Example";
char outOfRangeChar = myString[10];  // Violates rule by accessing out-of-range elements
```

The following code demonstrates improper adherence to the standard. The following example attempts to modify a string literal. This is in violation of the principle.

```
// Rule 5: Do Not Modify String Literals
const char* immutableString = "Immutable";
immutableString[0] = 'X';  // Violates rule by attempting to modify a string literal
```

The following code demonstrates improper adherence to the standard. The following example does not convert a signed type into unsigned prior to conversion resulting in a potential loss of data. This is also in violation of the principle.

```
// Rule 6: Ensure Conversions do not Result in Loss of Data
signed char negativeChar = -128;
unsigned char lossOfData = negativeChar;  // Violates rule by not converting signed to unsigned
```

The following code demonstrates improper adherence to the standard. The following example illustrates improper usage of narrow versus wide strings in violation of the principle.

```
// Rule 7: Understand Narrow vs. Wide Strings and Functions
const char* narrowString = "Narrow";
wchar_t wideChar = static_cast<wchar_t>(narrowString[0]);  // Violates rule by not using wide string functions
```

The following code demonstrates improper adherence to the standard. The example below contains no user input validation, in violation of the principle.

Noncompliant Code

```
// Rule 8: Validate User Input
string userInput;
cout << "Enter a word: ";
cin >> userInput;  // Violates rule by not validating user input
```

Compliant Code

The following code demonstrates proper adherence to the standard. The example below adequately allocates enough space for the character data and the null terminator.

```
// Rule 1: Allocate Adequate String Storage Space for Char Data and Null Terminator
char sufficientBuffer[6];  // Allocates enough space for the null terminator
strcpy(sufficientBuffer, "Hello");
```

The following code demonstrates proper adherence to the standard. The following example no longer creates a string from a null pointer, instead allocating sufficient memory for a valid string.

```
// Rule 2: Do Not Create a String from a Null Pointer
char* validPointer = new char[10];  // Allocates memory for a valid string
strcpy(validPointer, "Valid");
```

The following code demonstrates proper adherence to the standard. The following example utilizes a valid pointer in the string operation. This example abides by Rule 3.

```
// Rule 3: Use Valid References and Pointers
char validString[10];
strcpy(validString, "Valid");  // Uses a valid pointer in string operations
```

The following code demonstrates proper adherence to the standard. The following example includes a range checker to ensure out-of-bounds errors are mitigated.

```
// Rule 4: Range Check Element Access
char myString[] = "Example";
if (strlen(myString) > 10) {
    char inRangeChar = myString[10];  // Checks range before accessing elements
}
```

The following code demonstrates proper adherence to the standard. The following example no longer attempts to modify a string literal, rather it modifies a mutable string.

```
// Rule 5: Do Not Modify String Literals
char mutableString[] = "Mutable";
mutableString[0] = 'M';  // Modifies a mutable string
```

The following code demonstrates proper adherence to the standard. The following example converts a signed char with a negative value to an unsigned char, this does not result in a loss of data. It is worth noting in this specific example, the representation of the data may change.

Compliant Code

```
   // Rule 6: Ensure Conversions do not Result in Loss of Data
   signed char negativeChar = -128;
   unsigned char noLossOfData = static_cast<unsigned char>(negativeChar);  // Converts signed to unsigned
```

The following code demonstrates proper adherence to the standard. In the following example, a proper utilization of narrow vs wide string functions can be seen that abides by the principle.

```
   // Rule 7: Understand Narrow vs. Wide Strings and Functions
   const char* narrowString = "Narrow";
   wchar_t wideChar = L'W';  // Uses wide string functions
```

The following code demonstrates proper adherence to the standard. The example below illustrates one specific example of input validation, specific implementation will vary depending on needs.

```
   // Rule 8: Validate User Input
/* This implementation is valid, as an example it checks for numbers present in the entered string.)
   string userInput;
   cout << "Enter a word: ";
   getline(cin, userInput);  // Read the entire line

   // Check if userInput contains any numeric characters
   bool containsNumbers = false;
   for (char c : userInput) {
      if (isdigit(c)) {
         containsNumbers = true;
         break;
      }
   }
   if (containsNumbers) {
      cout << "Invalid input. Please enter a word without numbers." << endl;
   } else {
      cout << "Valid input: " << userInput << endl;
      // Proceed with the rest of your program logic
   }
```

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

Standard: String Correctness
Principles(s):

1. **Validate Input Data:** Validating user input is an important principle to be aware of and adhere to when concerning the standard String Correctness. Ensuring to practice input validation will help to align the developer and their work with best security practices, mitigate unexpected behavior and vulnerability, as well as bolster a robust DiD system. Specifically, when concerning String Correctness this will help the developer avoid loss of data, mitigate serious error, and avoid potential injection attacks.
2. **Heed Compiler Warnings:** Paying close attention to compiler warnings is important for most every standard, including String Correctness. Addressing compiler warnings is critical to addressing potential errors and vulnerabilities present in code. Improper handling or neglecting of warnings can lead to unexpected behavior, vulnerability, and improper alignment with the standard and secure principles. As illustrated in the examples above, there are many improper handlings of String Correctness, and developers should be mindful to heed all compiler warnings and mitigate against common attack avenues utilizing strings.
3. **Architect for Security:** Developers should remain cognisant of the principle of architect and design for security policies when handling String Correctness. Building a robust codebase from the ground-up with common vulnerabilities in mind and mitigation strategies implemented throughout development will result in a more layered and secure application. Developers should be mindful to adhere to this principle regardless of the standard, but particularly with strings, as they are common avenues of attack and vulnerability.
4. **Keep it Simple:** Developers should be mindful of the principle of keeping it simple. This mundane statement, if considered adequately, can help mitigate vulnerability and error. By practicing the principle of simplicity a developer can ensure alignment with the principle, secure standard set in this document, as well as bolster team efficiency. Applying this to String Correctness will help a developer ensure mitigation of noncompliant and vulnerable code.
5. **Sanitize Data Sent:** When data is being transmitted it is imperative to apply data sanitization. In order to adhere to best security policies and protect the integrity of data, validating data sent and received is essential for maintaining a healthy, secure system. This is particularly true with strings, as they are a common format for sensitive information and must be handled properly, in accordance with the principle and standard.
6. **Utilize Quality Assurance:** Utilizing robust quality assurance methods is critical throughout development, reinforcing the principle of design for security. Developers should employ a range of tailored tools to mitigate and remove as many vulnerabilities as possible throughout development. For a list of recommended tools relating to String Correctness specifically, please see the 'Automation' section below.
7. **Adopt a Secure Standard:** Adopting and maintaining a secure standard is important throughout development. In relation to String Correctness, this will help the developer to mitigate common vulnerabilities and boost team efficiency. By adopting and applying a secure standard one can ensure the most efficient, secure development possible.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	High	2 - 3

Automation

Tool	Version	Checker	Description Tool
CPPCheck	2.12	Comprehensive Static Analysis	CPPCheck can perform a comprehensive static analysis when all checks are turned on. This will check for a variety of violations and

Tool	Version	Checker	Description Tool
			style errors in C++ code. [https://cppcheck.sourceforge.io/]
Microsoft/MSVS Code Analysis	Visual Studio 17.8.2	Static Analysis	Microsoft Visual Studios comes with a built-in static analysis tool that should be enabled in the project properties for further coverage. To see extended usage see link below. https://learn.microsoft.com/en-us/cpp/build/reference/analyze-code-analysis?view=msvc-170
Clang-Tidy	18.0.0	Static Analysis	Clang-Tidy is a static analysis tool that can check for a variety of errors. For implementation in C++ see the following link. This tool can be used in conjunction with the previous. https://chat.openai.com/c/26243feb-b3a1-418f-9ef2-de1c3d2c60f5
PVS-Studio	7.22	Static Analysis, V579	PVS-Studio is a static analyzer that aids in code quality, security (SAST) and safety. More info on specific versions and languages, including C++, can be found below. https://pvs-studio.com/en/

Coding Standard 4

Coding Standard	Label	Name of Standard
SQL Injection	[STD-004-CPP]	This coding standard aims to establish consistent and secure practices for handling attempted SQL Injection. Adhering to this standard will reduce the risk of vulnerabilities and exploits as well as enhance code quality. **1. Use Parameterized Queries:** The utilization of prepared statements with variable binding (parameterized query) while interacting with the database (db) will help prevent SQL vulnerability. Parameterization separates SQL code from input, heightening security. Parameterized queries force the developer to define all the SQL code before passing each parameter to the query. This subsequently allows the db to distinguish between code and data, regardless of user input. Parameterized Queries are a crucial step in preventing SQL attack. **2. Sanitize and Validate Input:** Sanitizing and validating user input remains an important layer of defense when protecting against SQL injection attack. The developer should remove or escape characters that can be used in an SQL injection attack. What to exclude is up to the discretion of the developer, but common keyphrases and characters to exclude are “and”, “or”, ‘, and “. **3. Implement Robust Error Handling:** Much like in other categories, implementing robust error handling is critical for mitigating SQL injection attack. It is imperative the developer avoids exposing detailed error messages to the user, as it is possible to accidentally provide valuable information to attackers. **4. Secure Sensitive Information:** As part of implementing a robust DiD strategy, developers must secure sensitive information in additional layers of security. Sensitive information must not be hardcoded in the software. Sensitive information should be stored on a db with strong hashing algorithms to secure data. **5. Utilize Stored Procedures:** Similar to the utilization of parameterized queries, stored procedures have the same potential for SQL mitigation as parameterized queries, if utilized correctly. Stored Procedures require the developer to build SQL statements with parameters that are typically automatically parameterized. The difference between stored procedures and parameterized queries is that stored procedures are defined and stored in the db itself, and called from the application.

Noncompliant Code

The following code demonstrates improper adherence to the standard. The example below directly concatenates the user supplied input into the SQL query. This is incorrect and violates the principle.

```
// Rule 1: Use Parameterized Queries
// Non-compliant: Directly concatenating user input into the SQL query
std::string userInput = "Jane Doe";
```

Noncompliant Code

```
std::string query = "SELECT * FROM users WHERE username='" + userInput + "';";
```

The following code demonstrates improper adherence to the standard. The example below provides no method of sanitization or validation of input, lacking a key link of the DiD chain. This is in violation of the principle.

```
// Rule 2: Sanitize and Validate Input
// Non-compliant: Not sanitizing user input, risking SQL injection
std::string unsafeInput = "'; DROP TABLE users; --";
std::string unsafeQuery = "SELECT * FROM users WHERE username='" + unsafeInput + "';";
```

The following code demonstrates improper adherence to the standard. The following example does not supply any error handling. In addition to weakening the resilience of the program, this is in violation of the principle.

```
// Rule 3: Implement Robust Error Handling
// Non-compliant: Not handling SQL injection attempts in user input
std::string errorProneInput = "John Doe' OR 1=1; --";
std::string errorProneQuery = "SELECT * FROM users WHERE username='" + errorProneInput + "';";
```

The following code demonstrates improper adherence to the standard. The following example hardcodes a password into the program. This approach lacks security and is in violation of the principle.

```
// Rule 4: Secure Sensitive Information
// Non-compliant: Storing sensitive information in the application code
std::string password = "secretpassword";
```

The following code demonstrates improper adherence to the standard. The following example does not utilize stored procedures to prevent SQL injection, in actuality it provides no protection whatsoever. This is in violation of the principle.

```
// Rule 5: Utilize Stored Procedures
// Non-compliant: Not using stored procedures to prevent SQL injection
cout << "Enter Input";
cin >> userInput;
```

Compliant Code

The following code demonstrates proper adherence to the standard. The following code block displays a simple example with a placeholder parameter. Following this, guidelines-specific parameters can be implemented based on need.

```
// Rule 1: Use Parameterized Queries
// Compliant: Using parameterized query to prevent SQL injection
std::string userInput = "Jane Doe";
std::string query = "SELECT * FROM users WHERE username=?";  // Placeholder for parameter
```

The following code demonstrates proper adherence to the standard. The following includes an example of a Regular expression meant to sanitize input and prevent SQL injection. Specific implementation will vary.

Compliant Code

```
// Rule 2: Sanitize and Validate Input
// Compliant: Using Regex to prevent SQL injection
std::regex theRegulator("[^a-z0-9](and|or)[\\'\"]{0}", std::regex_constants::icase); // Example Regex
```

The following code demonstrates proper adherence to the standard. The following example includes error handling that can be utilized in conjunction with the previous Regex, as a form of error handling and SQL injection defense.

```
// Rule 3: Implement Robust Error Handling
// Compliant: Implementing error handling for SQL injection attempts
// detect if SQL injection pattern matched
if (std::regex_search(sql, theRegulator)) {
   // display error message
   std::cout << "Access Denied! SQL Injection Attack Detected!" << std::endl;
   return false;
}
```

The following code demonstrates proper adherence to the standard. The following example illustrates a secure password being retrieved from a separate db rather than hardcoded in. This approach aligns with the principle and provides greater security.

```
// Rule 4: Secure Sensitive Information
// Compliant: Not storing sensitive information in the application code
std::string password;  // Password should be retrieved securely, not hardcoded
```

The following code demonstrates proper adherence to the standard. The following code block illustrates how a specific procedure may be implemented. Implementation will vary depending on goals and purpose.

```
// Rule 5: Utilize Stored Procedures
// Compliant: Using stored procedures to prevent SQL injection
std::string userInput;
cout << "Enter Input: ";
cin >> userInput;
// Call a stored procedure instead of directly using user input in a query
```

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

Standard: SQL Injection
Principles(s):

1. **Validate Input Data:** Validating Input is an important principle to be aware of and adhere to, especially when concerning SQL Injection. Ensuring to practice input validation will help to align the developer and their work with best security practices, mitigate unexpected behavior and vulnerability, as well as bolster a robust DiD system. As SQL Injection is a type of attack where the attacker manipulates or injects SQL code into input fields to gain unauthorized access or manipulate the DB, validating input is particularly important.
2. **Architect for Security:** Developers should remain especially cognisant of the principle to architect and design for security policies when handling SQL Injection. Building a robust codebase from the ground-up with common attack strategies in mind and mitigation strategies implemented throughout development will result in a more layered and secure application. Developers should be especially mindful to adhere to this principle when concerning SQL Injection, as vulnerabilities can occur without explicit error in the code, mitigation is essential.
3. **Sanitize Data Sent:** Sanitizing Data before transmission is particularly important when concerning SQL Injection. As data sanitization will allow any harmful content to be removed, this principle will work hand-in-hand with input validation to ensure the best possible mitigation against injection attack.
4. **Adopt a Secure Standard:** Adopting and maintaining a secure standard is important throughout development, particularly for SQL Injection. As vulnerability can occur without explicit error in the code, it is important to adhere to the provided standard and mitigate against known injection attack avenues.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	High - Medium	High	3

Automation

Tool	Version	Checker	Description Tool
CPPCheck	2.12	Comprehensive Static Analysis	CPPCheck can perform a comprehensive static analysis when all checks are turned on. This will check for a variety of violations and style errors in C++ code. [https://cppcheck.sourceforge.io/]
Microsoft/MSVS Code Analysis	Visual Studio 17.8.2	Static Analysis	Microsoft Visual Studios comes with a built-in static analysis tool that should be enabled in the project properties for further coverage. To see extended usage see link below. https://learn.microsoft.com/en-us/cpp/build/reference/analyze-code-analysis?view=msvc-170
Clang-Tidy	18.0.0	Static Analysis	Clang-Tidy is a static analysis tool that can check for a variety of errors. For implementation in C++ see the following link. This tool can be used in conjunction with the previous.

Tool	Version	Checker	Description Tool
			https://chat.openai.com/c/26243feb-b3a1-418f-9ef2-de1c3d2c60f5
Checkmarx	3.0	Checkmarx One https://checkmarx.com/product/application-security-platform/?	Checkmarx One is a cloud-native platform that provides a full suite of application security testing (AST).

Coding Standard 5

Coding Standard	Label	Name of Standard
Memory Protection	[STD-005-CPP]	This coding standard aims to establish consistent and secure practices for handling Memory Protection. Adhering to this standard will reduce the risk of vulnerabilities and exploits as well as enhance code quality. 1. **Use Valid References and Pointers:** It is imperative to avoid storing an iterator, reference, or pointer to an element within a container where there is a risk of modification such that the pointer, iterator, or reference becomes invalid. Invalid or null references can lead to undefined behavior, security vulnerability, and inefficient resource usage. 2. **Adopt Proper Read/Write Permissions:** Assigning appropriate read, write and execute permissions aligns with the principle of Least Privilege discussed earlier. Ensuring that code and data have the minimum access necessary for function will help mitigate unauthorized access or modification of memory areas. 3. **Utilize Address Space Layout Randomization (ASLR):** ASLR is a common technique utilized to randomize the memory addresses used by system components. Employing this security measure will make it more challenging for attackers to predict the specific location of code or data. ASLR is an important aspect of a comprehensive DiD strategy. 4. **Utilize Stack Canaries:** Although Stack Canaries do very little to actually prevent modification or attack, they can serve as valuable alert systems to when an attack has happened. Similar to ASLR, they are a valuable addition to a robust DiD strategy. 5. **Utilize the Data Execution Prevention (DEP) Method:** Also in line with the principle of least privilege, DEP prevents the execution of code from certain memory regions. This is done to help mitigate the risk of code injection attacks by marking certain regions of memory as non-executable. DEP is another important piece of a robust DiD strategy. 6. **Ensure Adequate Bounds Allocation/Checking:** It is imperative of program security and health to ensure adequate bounds allocation and checking when dealing with memory protection and management. Proper bounds allocation and checking involves validating the array or buffer accesses to ensure they are within and stay within the allocated boundaries. Improperly handling either the allocation or checking step can lead to buffer overflow and data corruption.

Noncompliant Code

The following code demonstrates improper adherence to the standard. The following example illustrates a potential violation of the standard where a reference to a local variable goes out of scope.

```
// Rule 1: Use Valid References and Pointers
// Non-compliant: Storing a reference to a local variable that goes out of scope
int* nonCompliantPointer;
```

Noncompliant Code

```
{
    int localVariable = 42;
    nonCompliantPointer = &localVariable;  // Violates rule by storing a reference to a local variable
}
std::cout << *nonCompliantPointer << std::endl;  // Undefined behavior
```

The following code demonstrates improper adherence to the standard. The following example attempts to write character data outside the bounds of the array. This is in violation of the principle and will cause error.

```
// Rule 2: Adopt Proper Read/Write Permissions
// Non-compliant: Assigning inappropriate permissions to memory
char* nonCompliantMemory = new char[10];
nonCompliantMemory[15] = 'A';  // Violates rule by writing to memory beyond allocated bounds
std::cout << nonCompliantMemory[15] << std::endl;  // Undefined behavior
```

The following code demonstrates improper adherence to the standard. The following example illustrates the importance of utilizing ASLR. ASLR will make it more difficult for attackers to predict the location of sensitive data.

```
// Rule 3: Utilize Address Space Layout Randomization (ASLR)
// Non-compliant: Assuming predictable memory layout
// Violates rule by assuming a fixed memory address
int* predictableMemory = reinterpret_cast<int*>(0x00000000);
std::cout << *predictableMemory << std::endl;  // Potential security vulnerability
```

The following code demonstrates improper adherence to the standard. The following code example does not implement a stack canary in violation of the principle.

```
// Rule 4: Utilize Stack Canaries
// Non-compliant: Lack of stack canaries
char buffer[100];
// Code without stack canary protection
```

The following code demonstrates improper adherence to the standard. The following example does not properly implement DEP and thus violates the principle. DEP is a common tactic utilized to prevent malicious attack or vulnerability and should be utilized.

```
// Rule 5: Utilize the Data Execution Prevention (DEP) Method
// Non-compliant: Executing code from non-executable memory
typedef void (*FunctionPointer)();
FunctionPointer nonCompliantFunction = reinterpret_cast<FunctionPointer>(buffer);
nonCompliantFunction();  // Violates rule by executing code from buffer
```

The following code demonstrates improper adherence to the standard. The following example attempts to access character data outside the bounds of the array, in violation of the principle.

```
// Rule 6: Ensure Adequate Bounds Allocation/Checking
// Non-compliant: Insufficient bounds checking
char unsafeArray[5];
for (int i = 0; i <= 5; ++i) {
```

Noncompliant Code

```
    unsafeArray[i] = 'A';  // Violates rule by not checking bounds properly
  }
```

Compliant Code

The following code demonstrates proper adherence to the standard. The following example correctly references a valid variable and abides by the principle.

```
// Rule 1: Use Valid References and Pointers
// Compliant: Avoid storing a reference to a local variable that goes out of scope
int* compliantPointer = new int(42);  // Dynamically allocated variable with a valid scope
std::cout << *compliantPointer << std::endl;  // Reference to a valid variable
// Remember to deallocate the dynamically allocated memory
delete compliantPointer;
```

The following code demonstrates proper adherence to the standard. The following example correctly addresses an element within the array, and no longer violates the principle.

```
// Rule 2: Adopt Proper Read/Write Permissions
// Compliant: Assign appropriate permissions to memory
char* compliantMemory = new char[10];
compliantMemory[5] = 'A';  // Correctly writes to allocated memory
std::cout << compliantMemory[5] << std::endl;  // Outputs the expected value
```

Utilizing ASLR will vary depending on implementation and system. Developers are cautioned that it is required to adhere to the principle.

```
// Rule 3: Utilize Address Space Layout Randomization (ASLR)
// Compliant: Avoid assuming predictable memory layout
// Use dynamic memory allocation or rely on system functions, implementation will vary
```

The following code demonstrates proper adherence to the standard. The following example illustrates how a stack canary may be implemented. Specific implementation will vary depending on the instance and need.

```
// Rule 4: Utilize Stack Canaries
// Compliant: Implement stack canaries if required by the platform
char bufferWithCanary[104];  // Includes space for a stack canary
// Code with stack canary protection, implementation will vary
…
```

The following code demonstrates proper adherence to the standard. The following example illustrates an instance where executing non-executable memory was avoided. Thus it is in line with the principle and valid.

```
// Rule 5: Utilize the Data Execution Prevention (DEP) Method
// Compliant: Avoid executing code from non-executable memory
// Ensure function pointers point to valid executable code
typedef void (*FunctionPointer)();
FunctionPointer compliantFunction = reinterpret_cast<FunctionPointer>(&std::memset);
```

Compliant Code

```
compliantFunction();  // Executes a standard library function
```

The following code demonstrates proper adherence to the standard. The following example correctly implements bounds checking and will not prompt an out-of-bounds error.

```
// Rule 6: Ensure Adequate Bounds Allocation/Checking
// Compliant: Perform bounds checking to avoid buffer overflow
char safeArray[5];
for (int i = 0; i < 5; ++i) {
    safeArray[i] = 'A';  // Correctly checks bounds and writes to the array
}
```

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

Standard: Memory Protection
Principles(s):

1. **Heed Compiler Warnings:** Paying close attention to compiler warnings is important for most every standard, and critically so for Memory Protection. Addressing compiler warnings is essential for mitigating potential errors and vulnerabilities present in code. Improper handling or neglecting of warnings can lead to unexpected behavior, vulnerability, and improper alignment with the standard and secure principles. As Memory Protection is a critical aspect of security, it is especially important for developers to heed all compiler warnings.
2. **Keep it Simple:** Practicing simplicity in design and implementation is critical for Memory Protection. As complex systems can introduce more opportunities for vulnerability or attack, simplicity of design in Memory Protection is a layer of security itself, and should be kept in mind by all developers.
3. **Default Deny:** Adopting the principle of Default Deny is necessary for secure Memory Protection. Only providing users with the need to access memory the ability to do so is an important step in eliminating many potential avenues of attack, and provides another layer in a robust DiD implementation.
4. **Practice Least Privilege:** Similar to the logic behind Default Deny, adopting the Principle of Least Privilege is essential within Memory Protection. If a user needs access to memory, they should only be allowed access to what is necessary for their purpose. This narrowed scope will help mitigate from attack as well as potentially limit the damage of a breach, and is an essential layer in a robust DiD implementation.
5. **Practice DiD:** Adopting DiD in relation to Memory Protection is critical. As many attacks ultimately seek to gain access to this portion of the system, implementing a robust and layered defensive strategy is paramount in mitigating and limiting vulnerability and attack. Providing a diverse and effective array of security measures is a must for any developer dealing with Memory Protection.
6. **Use Quality Assurance:** Utilizing robust quality assurance methods is critical throughout development, reinforcing the principle of design for security. Developers should employ a range of tailored tools to mitigate and remove as many vulnerabilities as possible throughout development. For a list of recommended tools relating to Memory Protection specifically, please see the ‘Automation’ section below.
7. **Adopt a Standard:** Adopting and maintaining a secure standard is important throughout development. In relation to Memory Protection, this will help the developer to better handle allocation, management, and freeing of memory. By adopting and applying a secure standard, we can ensure the most efficient, error-free development possible.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Critical	Likely	High - Medium	Maximum	3

Automation

Tool	Version	Checker	Description Tool
CPPCheck	2.12	Comprehensive Static Analysis	CPPCheck can perform a comprehensive static analysis when all checks are turned on. This will check for a variety of violations and style errors in C++ code. [https://cppcheck.sourceforge.io/]
Checkmarx	3.0	Checkmarx One https://checkmarx.com/product/application-security-platform/?	Checkmarx One is a cloud-native platform that provides a full suite of application security testing (AST).
AddressSanitizer	Clang 18.0.0	Memory Sanitizer https://clang.llvm.org/docs/MemorySanitizer.html	AddressSanitizer with Memory Sanitizer capabilities can detect various memory-related issues enhancing protection.

Coding Standard 6

Coding Standard	Label	Name of Standard
Assertions	[STD-006-CPP]	This coding standard aims to establish consistent and secure practices for handling assertions. Adhering to this standard will reduce the risk of vulnerabilities and exploits as well as enhance code quality. 1. **Utilize Assertions to Document Assumptions:** Developers should employ assertions strategically. Assertions act as concise documentation, explicitly defining the expected behavior at points in the program. This will aid in debugging and comprehension. 2. **Utilize Assertions for Debugging Not Error Handling:** As assertions are typically disabled for a production build, assertions should be utilized as a debugging tool, not an error handling tool. Error handling should be accomplished separately of assertions. 3. **Disable Assertions in Production Builds:** Developers should disable assertions for production builds. Assertions can lead to unnecessary overhead and potential security risks, and as such, they should be utilized for debugging during the development phase and removed afterward. 4. **Avoid Side Effects:** Developers should ensure that assertions do not alter the program or introduce unintended effects. Assertions that induce effect may introduce modification that can obscure an issue. 5. **Review and Update Assertions Regularly:** Assertions should be regularly reviewed and updated. As assertions serve as a form of debugging and documentation, as the project continues development or requirements evolve, Assertions should be updated to reflect this. Failure to remove or update assertions can lead to inefficient or confusing development.

Noncompliant Code

The following code demonstrates improper adherence to the standard. The following example fails to utilize assertions to determine behavior. This is in violation of the principle.

```
// Rule 1: Utilize Assertions to Document Assumptions
// Non-compliant: Does not utilize assertion to ensure proper behavior.
int x = 5;
int y = 10;
int z = 20;
x = y;
z = x;
return z;
```

The following code demonstrates improper adherence to the standard. The following example attempts to utilize an assertion for error handling. This is a faulty practice and in violation of the principle.

```
// Rule 2: Utilize Assertions for Debugging Not Error Handling
// Non-compliant: Incorrect use of assertions for error handling
```

Noncompliant Code

```
int* ptr = nullptr;
assert(ptr != nullptr);  // Violates rule by using assertion for error handling
```

The following code demonstrates improper adherence to the standard. The following example illustrates a mock build version where the developer forgot to disable assertions. This is in violation of the principle.

```
// Rule 3: Disable Assertions in Production Builds
// Non-compliant: Leaving assertions enabled in a production build
#ifndef NDEBUG
assert(false);  // Violates rule by not disabling assertions for production
#endif
```

The following code demonstrates improper adherence to the standard. The following example illustrates an assertion that has side effects, this is bad practice and in violation of the principle.

```
// Rule 4: Avoid Side Effects
// Non-compliant: Assertion with side effect
int y = 10;
assert(++y > 0);  // Violates rule by having a side effect in the assertion
```

The following code demonstrates improper adherence to the standard. The following example illustrates how outdated assertions can lead to confusing code and reduce readability.

```
// Rule 5: Review and Update Assertions Regularly
// Non-compliant: Outdated assertion
int z = 15;
assert(x < 10);  // Violates rule by not updating the assertion with changing requirements
```

Compliant Code

The following code demonstrates proper adherence to the standard. The example properly implements an assertion to determine behavior.

```
// Rule 1: Utilize Assertions to Document Assumptions
// Compliant: Utilize assertion to document assumptions.
int x = 5;
int y = 10;
int z = 20;
x = y;
z = x;
assert(z == x); // Proper use of assertion to determine behavior
return z;
```

The following code demonstrates proper adherence to the standard. The example does not attempt to utilize assertions for error handling, and would contain additional code when implemented. Specific implementation will depend on requirements.

```
// Rule 2: Utilize Assertions for Debugging Not Error Handling
```

Compliant Code

```
// Compliant: Assertions are used for debugging, not error handling.
int* ptr = new int;
assert(ptr != nullptr);  // Assuming ptr should not be null
// Continue with the rest of the code...
delete ptr;
```

The following code demonstrates proper adherence to the standard. The example correctly disables assertions for the product build.

```
// Rule 3: Disable Assertions in Production Builds
// Compliant: Assertions are disabled in a production build.
#ifndef NDEBUG
// DISABLE THE ASSERTIONS
#endif
```

The following code demonstrates proper adherence to the standard. The example correctly refactors the assertion so that it no longer has any side effects.

```
// Rule 4: Avoid Side Effects
// Compliant: Assertion without side effect.
int y = 10;
assert(y > 0);  // Proper assertion, does not incur effects
```

The following code demonstrates proper adherence to the standard. The example illustrates the correct updating of the assertion, and how this will boost readability and lower confusion.

```
// Rule 5: Review and Update Assertions Regularly
// Compliant: Updated assertion with changing requirements.
int updatedX = 15;  // Assuming x is updated
assert(updatedX > 10);  // Assertion properly updated
```

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

Standard: Assertions
Principles(s):

1. **Heed Compiler Warnings:** Paying close attention to compiler warnings is important for most every standard, including Assertions. Addressing compiler warnings is critical in addressing potential errors and vulnerabilities present in code. Compiler Warnings may warn the developer of potential issues with their Assertions, such as unreachable code or conflicting conditions. Paying attention to all compiler warnings is essential.
2. **Keep it Simple:** When utilizing Assertions, it is important that developers keep the implementation simple and specific in scope. Overly complex Assertions can be less effective and confusing for a team of developers. This principle will also aid in reinforcing the developer not to create Assertions with side effects.
3. **Practice DiD:** Utilizing Assertions as part of a robust DiD strategy is important and can help mitigate conditions or vulnerabilities not covered elsewhere in the code. Utilizing Assertions in debugging is an important step to ensuring a product's security and should be utilized as part of a developer's DiD strategy and security by design.
4. **Utilize Quality Assurance:** Utilizing robust quality assurance methods is critical throughout development, reinforcing the principle of design for security. Developers should employ Assertions alongside static analysis to mitigate and remove as many vulnerabilities as possible throughout development. For a list of recommended tools relating to Assertions specifically, please see the 'Automation' section below.
5. **Adopt a Secure Standard:** Adopting and maintaining a secure standard is important throughout development. In relation to Assertions, this will help the developer to utilize Assertions judiciously, helping to define when and how to utilize them to ensure they enhance code security. By adopting and applying a secure standard to Assertions, we can ensure the most efficient, error-free development possible.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Low	Medium	1

Automation

Tool	Version	Checker	Description Tool
CPPCheck	2.12	Comprehensive Static Analysis	CPPCheck can perform a comprehensive static analysis when all checks are turned on. This will check for a variety of violations and style errors in C++ code, including unneeded or improper assertions. [https://cppcheck.sourceforge.io/]

Coding Standard 7

Coding Standard	Label	Name of Standard
Exceptions	[STD-007-CPP]	This coding standard aims to establish consistent and secure practices for handling exceptions. Adhering to this standard will reduce the risk of vulnerabilities and exploits as well as enhance code quality. 1. **Handle Exceptions Appropriately:** Developers should employ exception-handling judiciously, ensuring exceptions are caught and handled at the appropriate level. This will involve the developer identifying specific types of exceptions that can be thrown and handling them in a manner suitable to the specific project. This will help enhance program robustness and facilitate debugging. **2. Clean up Resources in Exception Handling:** It is imperative that developers clean up any resources that may have been acquired before the exception was thrown. Failing to do so can result in resource leaks and degrade system performance. **3. Avoid Unnecessary Try-Catch Blocks:** Developers should avoid the usage of excessive or unnecessary try-catch blocks. Exception-handling carries overhead and should therefore be applied only where exceptions are likely to occur; unnecessary exceptions can clutter the code reducing readability and efficiency. **4. Rethrow Exceptions Accordingly:** When catching exceptions, developers will need to make informed decisions about whether to handle the exception at the current level, or propagate it up the stack. Rethrowing exceptions should be done selectively, preserving relevant information and providing useful context where necessary. This will help ensure the code's behavior remains consistent with requirements. **5. Handle All Exceptions:** It is crucial that developers handle all exceptions appropriately in their work. This involves both identifying and handling exceptions within the code. Additionally, it is beneficial to plan for unforeseen circumstances, as failing to handle all exceptions can lead to unexpected program behavior and vulnerability.

Noncompliant Code

The following code demonstrates improper adherence to the standard. This example attempts to handle all exceptions without specificity, violating the principle.

```
// Rule 1: Handle Exceptions Appropriately
// Non-compliant: Catching all exceptions without specific handling
void handleExceptionsNonCompliant() {
   try {
      // Some code that may throw exceptions
   } catch (...) {
      // Handling all exceptions without specific actions
      // Violates rule by not handling exceptions appropriately
```

Noncompliant Code

```
    }
}
```

The following code demonstrates improper adherence to the standard. The example fails to adequately clean resources after an exception, in violation of the principle.

```
// Rule 2: Clean up Resources in Exception Handling
// Non-compliant: Not cleaning up resources in the presence of an exception
void cleanupResourcesNonCompliant() {
    int* dynamicMemory = new int;
    try {
        // Some code that may throw exceptions
        throw std::runtime_error("An exception occurred");
    } catch (const std::exception&) {
        // Exception caught, but resources are not cleaned up
        // Violates rule by not cleaning up resources in exception handling
    }
}
```

The following code demonstrates improper adherence to the standard. The example utilizes unnecessary try-catch blocks that will increase overhead and code complexity. This is in violation of the principle.

```
// Rule 3: Avoid Unnecessary Try-Catch Blocks
// Non-compliant: Excessive use of try-catch blocks for every statement
void unnecessaryTryCatchNonCompliant() {
    try {
        // Statement 1 that may throw exceptions
    } catch (const std::exception&) {
        // Exception handling for Statement 1

    }
    try {
        // Statement 2 that may throw exceptions
    } catch (const std::exception&) {
        // Exception handling for Statement 2
        // Violates rule by using unnecessary try-catch blocks
    }
}
```

The following code demonstrates improper adherence to the standard. The example rethrows the exception without preserving relevant information. This is in violation of the principle and is bad practice.

```
// Rule 4: Rethrow Exceptions Accordingly
// Non-compliant: Rethrowing exceptions without preserved information
void rethrowExceptionsNonCompliant() {
    try {
        // Some code that may throw exceptions
        throw std::runtime_error("An exception occurred");
```

Noncompliant Code

```
    } catch (const std::exception& ex) {
        std::cerr << "Caught exception: " << ex.what() << std::endl;
        throw ex; // Rethrowing without preserving relevant information
    }
}
```

The following code demonstrates improper adherence to the standard. The example illustrates a complete ignoring of the exception and exception handling. This is in direct violation of the principle, and should be avoided.

```
// Rule 5: Handle All Exceptions
// Non-compliant: Ignores the exception
void cleanupResourcesNonCompliant() {
    int* dynamicMemory = new int;
    try {
        // Some code that may throw exceptions
        throw std::runtime_error("An exception occurred");
    } catch (...) {
        // No handling or cleanup, the exception is ignored
    }
    // Additional code...
}
    return 0;
}
```

Compliant Code

The following code demonstrates proper adherence to the standard. The example illustrates how a developer might structure their code to adequately handle exceptions. Specific implementation will vary.

```
// Rule 1: Handle Exceptions Appropriately
// Compliant: Handling specific exceptions with appropriate actions
void handleExceptionsCompliant() {
    try {
        // Some code that may throw exceptions
        throw std::runtime_error("An exception occurred");
    } catch (const std::exception& ex) {
        // Handling specific exceptions with appropriate actions
        std::cerr << "Caught exception: " << ex.what() << std::endl;
    }
}
```

The following code demonstrates proper adherence to the standard. The example creates a structure where a developer might include specific cleaning syntax. Specific implementation will vary.

```
// Rule 2: Clean up Resources in Exception Handling
// Compliant: Cleaning up resources in the presence of an exception
void cleanupResourcesCompliant() {
```

Compliant Code

```
    std::unique_ptr<int> dynamicMemory(new int);
    try {
        // Some code that may throw exceptions
        throw std::runtime_error("An exception occurred");
    } catch (const std::exception&) {
        // Cleaning up resources in the presence of an exception
    }
}
```

The following code demonstrates proper adherence to the standard. The example removes the second unnecessary try-catch block, reducing clutter and abiding by the principle.

```
// Rule 3: Avoid Unnecessary Try-Catch Blocks
// Compliant: Using try-catch blocks only where exceptions are likely
void necessaryTryCatchCompliant() {
    try {
        // Statement 1 that may throw exceptions
    } catch (const std::exception&) {
        // Exception handling for Statement 1
    }

    // Statement 2 that may throw exceptions
    // No unnecessary try-catch block for Statement 2
}
```

The following code demonstrates proper adherence to the standard. The example rethrows the exception but does so while preserving relevant information. This is in accordance with the principle.

```
// Rule 4: Rethrow Exceptions Accordingly
// Compliant: Rethrowing exceptions with preserved information
void rethrowExceptionsCompliant() {
    try {
        // Some code that may throw exceptions
        throw std::runtime_error("An exception occurred");
    } catch (const std::exception& ex) {
        std::cerr << "Caught exception: " << ex.what() << std::endl;
        throw; // Rethrowing with preserved relevant information
    }
}
```

The following code demonstrates proper adherence to the standard. The example no longer ignores the handling of the exception and illustrates an adequate structure for exception handling. Specific implementation will vary.

```
// Rule 5: Handle All Exceptions
// Compliant: Handles the exception and cleans up resources
void cleanupResourcesCompliant() {
    int* dynamicMemory = nullptr;
    try {
```

Compliant Code

```
        dynamicMemory = new int;
        // Some code that may throw exceptions
        throw std::runtime_error("An exception occurred");
    } catch (const std::exception& ex) {
        // Handle the exception, log, or perform necessary cleanup
        std::cerr << "Caught exception: " << ex.what() << std::endl;
    }

    // Cleanup resources, even in case of an exception
    if (dynamicMemory != nullptr) {
        delete dynamicMemory;
    }
    // Additional code...
}
```

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

Standard: Exceptions
Principles(s):

1. **Heed Compiler Warnings:** Paying close attention to compiler warnings is important for most every standard, including Exceptions. Addressing compiler warnings is critical to addressing potential errors and vulnerabilities present in code. Compiler Warnings may alert the developer of potential issues with Exceptions, such as unchecked exceptions or inadequate error handling. Paying attention to all compiler warnings is essential.
2. **Keep it Simple:** When utilizing Exceptions, it is important that developers keep the implementation simple and specific in scope. Overly complex Exceptions can be less effective and confusing for a team of developers. Prioritizing clarity and readability to facilitate debugging aligns with the principle and standard.
3. **Practice DiD:** Utilizing Exceptions as part of a robust DiD strategy is important and can help mitigate conditions or vulnerabilities not covered elsewhere in the code. Implementing layers within the exceptions is recommended, such as a combination of structured exception handling.
4. **Utilize Quality Assurance:** Utilizing robust quality assurance methods is critical throughout development, reinforcing the principle of design for security. Developers should employ Exceptions to mitigate and remove as many vulnerabilities as possible throughout development. For a list of recommended tools relating to Exceptions specifically, please see the 'Automation' section below.
5. **Adopt a Secure Standard:** Adopting and maintaining a secure standard is important throughout development. In relation to Exceptions, this will help the developer to apply Exceptions properly. This will then help to define when and how to utilize Exceptions to ensure they enhance code security. By adopting and applying a secure standard for Exceptions, we can ensure the most efficient, error-free development possible.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Possible	Medium	Medium	2

Automation

Tool	Version	Checker	Description Tool
CPPCheck	2.12	Comprehensive Static Analysis	CPPCheck can perform a comprehensive static analysis when all checks are turned on. This will check for a variety of violations and style errors in C++ code. [https://cppcheck.sourceforge.io/]
Microsoft/MSVS Code Analysis	Visual Studio 17.8.2	Static Analysis	Microsoft Visual Studios comes with a built-in static analysis tool that should be enabled in the project properties for further coverage. To see extended usage see link below. https://learn.microsoft.com/en-us/cpp/build/reference/analyze-code-analysis?view=msvc-170
Clang-Tidy	18.0.0	CWE-398: Indicator of Poor Code Quality	Clang-Tidy is a static analysis tool that can check for a variety of errors. For implementation in C++ see the following link. This tool can help identify issues relating to exception handling. https://chat.openai.com/c/26243feb-b3a1-418f-9ef2-de1c3d2c60f5

Coding Standard 8

Coding Standard	Label	Name of Standard
Memory Management	[STD-008-CPP]	This coding standard aims to establish consistent and secure practices for handling Memory Management. Adhering to this standard will reduce the risk of vulnerabilities and exploits as well as enhance code quality. 1. **Ensure Adequate Bounds Allocation/Checking:** It is imperative of program security and health to ensure adequate bounds allocation and checking when dealing with memory protection and management. Proper bounds allocation and checking involves validating the array or buffer accesses to ensure they are within and stay within the allocated boundaries. Improperly handling either the allocation or checking step can lead to buffer overflow and data corruption. 2. **Free Dynamically Allocated Memory when Not Needed:** Before the lifetime of the last pointer that stores the return value of a call to a standard memory allocation function has ended, it must be matched by a call to free() with that pointer value. Failure to free dynamically allocated memory can result in memory leaks and a degradation of system performance. 3. **Do Not Access Freed Memory:** Developers must avoid accessing memory that has been freed. Accessing freed memory is undefined behavior. Undefined behavior should be avoided at all costs, as it can lead to crashes or security vulnerabilities. 4. **Only Free Dynamically Allocated Memory:** Developers must not free memory that has not been allocated dynamically, this can result in heap corruption and other undefined behavior and serious errors. This rule does not apply to null pointers. 5. **Do Not Modify Object Alignment with realloc():** Developers must not invoke realloc() to modify the size of allocated objects that have stricter alignment requirements than those of malloc(). Doing so can result in memory corruption and is considered an unsafe practice.

Noncompliant Code

The following code demonstrates improper adherence to the standard. This example array does not ensure that the function will go out-of-bounds. As the array is declared with five elements (0-4), element 5 does not exist, thus it is improper.

```
// Rule 1: Ensure Adequate Bounds Allocation/Checking
// Non-compliant: Insufficient bounds checking
char unsafeArray[5];
for (int i = 0; i <= 5; ++i) {
   unsafeArray[i] = 'A';  // Violates rule by not checking bounds properly
}
```

Noncompliant Code

The following code demonstrates improper adherence to the standard. In this example, memory is not freed before return, thus violating the rule.

```
// Rule 2: Free Dynamically Allocated Memory when Not Needed
#include <stdlib.h>

enum { BUFFER_SIZE = 32 };

int f(void) {
  char *text_buffer = (char *)malloc(BUFFER_SIZE);
  if (text_buffer == NULL) {
    return -1;  // Non-compliance: Memory not freed before return
  }
  return 0;
}
```

The following code demonstrates improper adherence to the standard. In this example, the program is attempted to access memory that has been freed, thus violating the rule.

```
//Rule 3: Do Not Access Freed Memory
#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);  // Non-compliant: Accessing freed memory
  }
}
```

The following code demonstrates improper adherence to the standard. In this example, the program attempts to free statically allocated memory, in violation of the rule.

```
//Rule 4: Only Free Dynamically Allocated Memory
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

enum { MAX_ALLOCATION = 1000 };

int main(int argc, const char *argv[]) {
  char *c_str = NULL;
  size_t len;

  if (argc == 2) {
```

Noncompliant Code

```
    len = strlen(argv[1]) + 1;
    if (len > MAX_ALLOCATION) {
      /* Handle error */
    }
    c_str = (char *)malloc(len);
    if (c_str == NULL) {
      /* Handle error */
    }
    strcpy(c_str, argv[1]);
  } else {
    c_str = "usage: $>a.exe [string]";
    printf("%s\n", c_str);
  }
  free(c_str);  // Non-compliant: Freeing statically allocated memory
  return 0;
}
```

The following code demonstrates improper adherence to the standard. In this example, it is possible that realloc may change the alignment, thus it violates the rule.

```
//Rule 5: Do Not Modify Object Alignment with realloc()
#include <stdlib.h>

void func(void) {
  size_t resize = 1024;
  size_t alignment = 1 << 12;
  int *ptr;
  int *ptr1;

  if (NULL == (ptr = (int *)aligned_alloc(alignment, sizeof(int)))) {
    /* Handle error */
  }

  if (NULL == (ptr1 = (int *)realloc(ptr, resize))) {
    /* Handle error */
  }
  // Non-compliant: realloc() may change the alignment, violating the rule
}
```

Compliant Code

The following code demonstrates proper adherence to the standard. The example below correctly checks the bounds of and writes to the array.

```
// Rule 1: Ensure Adequate Bounds Allocation/Checking
// Compliant: Perform bounds checking to avoid buffer overflow
char safeArray[5];
```

Compliant Code

```
for (int i = 0; i < 5; ++i) {
    safeArray[i] = 'A';  // Correctly checks bounds and writes to the array
}
```

The following code demonstrates proper adherence to the standard. The code properly frees the dynamic memory once it no longer has need for it.

```
// Rule 2: Free Dynamically Allocated Memory when Not Needed
#include <stdlib.h>

enum { BUFFER_SIZE = 32 };

int f(void) {
  char *text_buffer = (char *)malloc(BUFFER_SIZE);
  if (text_buffer == NULL) {
    return -1;
  }

  free(text_buffer);  // Compliant: Freeing dynamically allocated memory
  return 0;
}
```

The following code demonstrates proper adherence to the standard. The example below avoids accessing memory that has been freed, and is therefore compliant with the principle.

```
#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);  // Compliant: Avoids accessing freed memory
  }
}
```

The following code demonstrates proper adherence to the standard. The example below does not attempt to free static memory, and rather, frees dynamically allocated memory. Thus, it is in line with the principle.

```
//Rule 4: Only Free Dynamically Allocated Memory
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
```

Compliant Code

```
enum { MAX_ALLOCATION = 1000 };

int main(int argc, const char *argv[]) {
  char *c_str = NULL;
  size_t len;

  if (argc == 2) {
    len = strlen(argv[1]) + 1;
    if (len > MAX_ALLOCATION) {
      /* Handle error */
    }
    c_str = (char *)malloc(len);
    if (c_str == NULL) {
      /* Handle error */
    }
    strcpy(c_str, argv[1]);
  } else {
    printf("%s\n", "usage: $>a.exe [string]");
    free(c_str);  // Compliant: Only frees dynamically allocated memory
    return EXIT_FAILURE;
  }
  free(c_str);
  return 0;
}
```

The following code demonstrates proper adherence to the standard. Alligned_alloc is used for allocations and memcpy is used for copying. Thus it aligns with the standard.

```
// Rule 5: Do Not Modify Object Alignment with realloc()
#include <stdlib.h>
#include <string.h>

void func(void) {
  size_t resize = 1024;
  size_t alignment = 1 << 12;
  int *ptr;
  int *ptr1;

  if (NULL == (ptr = (int *)aligned_alloc(alignment,
                                          sizeof(int)))) {
    /* Handle error */
  }

  if (NULL == (ptr1 = (int *)aligned_alloc(alignment,
                                           resize))) {
    /* Handle error */
  }
```

Compliant Code

```
  if (NULL == memcpy(ptr1, ptr, sizeof(int))) {
    /* Handle error */
  }

  free(ptr);
  // Compliant: Aligned_alloc is used for both allocations, and memcpy is used for copying
}
```

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

Standard: Memory Management
Principles(s):

1. **Heed Compiler Warnings:** Paying close attention to compiler warnings is important for most every standard, including Memory Management. Addressing compiler warnings is critical in mitigating potential errors and vulnerabilities present in code. Improper handling or neglecting of warnings can lead to unexpected behavior, memory leak, vulnerability, and improper alignment with the standard and secure principles. As Memory Management is a critical aspect of security, it is especially important for developers to heed all compiler warnings.
2. **Keep it Simple:** Practicing simplicity in design and implementation is important for Memory Management. As complex systems can introduce more opportunities for vulnerability or attack, simplicity of design in Memory Management is a layer of security itself, and will help adhere to the DiD standard.
3. **Default Deny:** Adopting the principle of Default Deny is necessary for secure Memory Management. Only allowing users with the need to access memory the ability to do so is an important step in eliminating many potential avenues of attack, and provides another layer in a robust DiD implementation.
4. **Practice Least Privilege:** Similar to the logic behind Default Deny, adopting the Principle of Least Privilege is essential within Memory Management. If a user needs access to memory, they should only be allowed access to what is necessary for their purpose. This narrowed scope will help mitigate from attack as well as potentially limit the damage of a breach, and is an essential layer in a robust DiD implementation.
5. **Sanitize Data Sent to Other Systems:** Sanitizing data is essential for Memory Management. When interacting with an external system, developers must ensure that data passed to and received from these systems is properly sanitized to adhere to the standard. This will help prevent memory-related vulnerabilities and injection attacks.
6. **Practice DiD:** Adopting DiD in relation to Memory Management is critical. As many attacks ultimately seek to gain access to the memory portion of the system, implementing a robust and layered defensive strategy is paramount in mitigating and limiting vulnerability and attack. Providing a diverse and effective array of security measures is a must for any developer dealing with Memory Management.
7. **Utilize Quality Assurance:** Utilizing robust quality assurance methods is critical throughout development, reinforcing the principle of design for security. Developers should employ a range of tailored tools to mitigate and remove as many vulnerabilities as possible throughout development. For a list of recommended tools relating to Memory Management specifically, please see the 'Automation' section below.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	High - Medium	High	3

Automation

Tool	Version	Checker	Description Tool
CPPCheck	2.12	Comprehensive Static Analysis	CPPCheck can perform a comprehensive static analysis when all checks are turned on. This will check for a variety of violations and style errors in C++ code. [https://cppcheck.sourceforge.io/]
Microsoft/MSVS Code Analysis	Visual Studio 17.8.2	Static Analysis	Microsoft Visual Studios comes with a built-in static analysis tool that should be enabled in the project properties for further coverage. To see extended usage see link below. https://learn.microsoft.com/en-us/cpp/build/reference/analyze-code-analysis?view=msvc-170
Clang-Tidy	18.0.0	Static Analysis	Clang-Tidy is a static analysis tool that can check for a variety of errors. For implementation in C++ see the following link. This tool can be used in conjunction with the previous. https://chat.openai.com/c/26243feb-b3a1-418f-9ef2-de1c3d2c60f5
Valgrind	3.22.0	Memcheck	Valgrind with Memcheck can aid in detecting memory-related errors and leaks, and should be utilized in conjunction with the prior tools.

Coding Standard 9

Coding Standard	Label	Name of Standard
File Input/Output (FIO)	[STD-009-CPP]	This coding standard aims to establish consistent and secure practices for handling FIO. Adhering to this standard will reduce the risk of vulnerabilities and exploits as well as enhance code quality. 1. **Exclude User Input from Format Strings:** Never call a formatted I/O function with a format string containing a tainted value. Malicious attackers may fully or partially control the contents of a format string and crash a vulnerable process, view stack or memory contents, or overwrite memory. 2. **Do Not Perform Operations on Devices Compatible Only with Files:** Developers must avoid performing operations on devices compatible only with files. Such device names include AUX, CON, COM1, LPT1 or paths using \\.\ device namespace. Performing operations on such devices can result in crashes, undefined behavior, and denial-of-service attacks. 3. **Do Not Copy a FILE Object:** FILE represents file streams. Developers should not attempt to copy or duplicate a file object. Improper duplication or copying can result in undefined behavior and file handling issues. 4. **Close Files when No Longer Needed:** A call to the fopen() or freopen() function must be matched with a call to fclose() before the lifetime of the last pointer that stores the return value of the call has ended or before the normal program termination, whichever is first. Failure to do so can lead to resource leaks as well as unexpected behavior. 5. **Do Not Access a Closed File:** Developers must not attempt to access a file after it has been closed, as doing so results in undefined behavior and program errors.

Noncompliant Code

The following code demonstrates improper adherence to the standard. The example directly incorporates the user input into a format string violating the principle.

```
//Rule 1: Exclude User Input from Format Strings
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void incorrect_password(const char *user) {
  int ret;
  /* User names are restricted to 256 or fewer characters */
  static const char msg_format[] = "%s cannot be authenticated.\n";
  size_t len = strlen(user) + sizeof(msg_format);
  char *msg = (char *)malloc(len);
  if (msg == NULL) {
    /* Handle error */
```

Noncompliant Code

```
  }
  ret = snprintf(msg, len, msg_format, user);
  if (ret < 0) {
    /* Handle error */
  } else if (ret >= len) {
    /* Handle truncated output */
  }
  fprintf(stderr, msg);
  free(msg);
  // Non-compliant: User input is directly incorporated into the format string
}
```

The following code demonstrates improper adherence to the standard. The example assumes that FILE is a file, not a device. This can cause unintended behavior and violates the principle.

```
// Rule 2: Do Not Perform Operations on Devices Compatible Only with Files
#include <stdio.h>

void func(const char *file_name) {
  FILE *file;
  if ((file = fopen(file_name, "wb")) == NULL) {
    /* Handle error */
  }

  /* Operate on the file */

  if (fclose(file) == EOF) {
    /* Handle error */
  }
  // Non-compliant: The code assumes that 'file' is a regular file,
  // but it may be a device or special file, leading to unintended behavior.
}
```

The following code demonstrates improper adherence to the standard. The example attempts to copy the FILE object in violation of the principle. This can additionally cause undefined and unexpected behavior.

```
// Rule 3: Do Not Copy a FILE Object
#include <stdio.h>

int main(void) {
  FILE my_stdout = *stdout;
  if (fputs("Hello, World!\n", &my_stdout) == EOF) {
    /* Handle error */
  }
  // Non-compliant: The code copies the FILE object, which can lead to undefined behavior.
  // FILE objects have internal state, and copying them may result in unexpected behavior.
  return 0;
}
```

Noncompliant Code

The following code demonstrates improper adherence to the standard. The example does not close the file after use, this is in violation of the principle.

```
// Rule 4: Close Files when No Longer Needed
#include <stdio.h>

int func(const char *filename) {
  FILE *f = fopen(filename, "r");
  if (NULL == f) {
    return -1;
  }
  // Non-compliant: The file is not closed after use, which can lead to resource leaks.
  /* ... */
  return 0;
}
```

The following code demonstrates improper adherence to the standard. The example attempts to access a closed file in violation of the principle. This can also lead to undefined behavior.

```
// Rule 5: Do Not Access a Closed File
#include <stdio.h>

int close_stdout(void) {
  if (fclose(stdout) == EOF) {
    return -1;
  }

  // Non-compliant: Accessing a closed file (stdout) after closing it.
  printf("stdout successfully closed.\n");
  return 0;
}
```

Compliant Code

The following code demonstrates proper adherence to the standard. In the corrected example, user input is not directly included in the format string, thus it abides by the principle.

```
//Rule 1: Exclude User Input from Format Strings
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void incorrect_password(const char *user) {
  int ret;
  /* User names are restricted to 256 or fewer characters */
  static const char msg_format[] = "%s cannot be authenticated.\n";
  size_t len = strlen(user) + sizeof(msg_format);
```

Compliant Code

```
  char *msg = (char *)malloc(len);
  if (msg == NULL) {
    /* Handle error */
  }
  ret = snprintf(msg, len, msg_format, user);
  if (ret < 0) {
    /* Handle error */
  } else if (ret >= len) {
    /* Handle truncated output */
  }
  fputs(msg, stderr);
  free(msg);
  // Compliant: User input is not directly included in the format string
}
```

The following code demonstrates proper adherence to the standard. The corrected example first checks whether FILE is an actual file before performing operations. This is in concordance with the principle and emphasizes good practice.

```
// Rule 2: Do Not Perform Operations on Devices Compatible Only with Files
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#ifdef O_NOFOLLOW
  #define OPEN_FLAGS O_NOFOLLOW | O_NONBLOCK
#else
  #define OPEN_FLAGS O_NONBLOCK
#endif

void func(const char *file_name) {
  struct stat orig_st;
  struct stat open_st;
  int fd;
  int flags;

  if ((lstat(file_name, &orig_st) != 0) ||
      (!S_ISREG(orig_st.st_mode))) {
    /* Handle error */
  }

  /* Race window */

  fd = open(file_name, OPEN_FLAGS | O_WRONLY);
  if (fd == -1) {
    /* Handle error */
  }
```

Compliant Code

```
  if (fstat(fd, &open_st) != 0) {
    /* Handle error */
  }

  if ((orig_st.st_mode != open_st.st_mode) ||
      (orig_st.st_ino  != open_st.st_ino) ||
      (orig_st.st_dev  != open_st.st_dev)) {
    /* The file was tampered with */
  }

  /*
   * Optional: drop the O_NONBLOCK now that we are sure
   * this is a good file.
   */
  if ((flags = fcntl(fd, F_GETFL)) == -1) {
    /* Handle error */
  }

  if (fcntl(fd, F_SETFL, flags & ~O_NONBLOCK) == -1) {
    /* Handle error */
  }

  /* Operate on the file */

  if (close(fd) == -1) {
    /* Handle error */
  }
  // Compliant: The code checks whether the file is a regular file
  // before performing file operations, preventing unintended behavior on devices or special files.
}
```

The following code demonstrates proper adherence to the standard. The corrected example utilizes a pointer to the FILE object, no longer copying it. This is in concordance with the principle.

```
// Rule 3: Do Not Copy a FILE Object
#include <stdio.h>

int main(void) {
  FILE *my_stdout = stdout;
  if (fputs("Hello, World!\n", my_stdout) == EOF) {
    /* Handle error */
  }
  // Compliant: The code uses a pointer to the original FILE object, avoiding copying.
  // This adheres to the standard behavior of FILE objects and prevents undefined behavior.
  return 0;
}
```

Compliant Code

The following code demonstrates proper adherence to the standard. The corrected example properly closes the file after usage as per the principle.

```
// Rule 4: Close Files when No Longer Needed
#include <stdio.h>

int func(const char *filename) {
  FILE *f = fopen(filename, "r");
  if (NULL == f) {
    return -1;
  }
  // Compliant: The file is closed after use, preventing resource leaks.
  /* ... */
  if (fclose(f) == EOF) {
    return -1;
  }
  return 0;
}
```

The following code demonstrates proper adherence to the standard. The corrected example includes a handler for the event that stdout was closed, and utilizes fputs rather than printf, avoiding a call to a closed file.

```
// Rule 5: Do Not Access a Closed File
#include <stdio.h>

int close_stdout(void) {
  if (fclose(stdout) == EOF) {
    return -1;
  }

  // Compliant: Accessing a closed file (stdout) through fputs instead of printf.
  fputs("stdout successfully closed.", stderr);
  return 0;
}
```

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

Standard: FIO Operations
Principles(s):

1. **Heed Compiler Warnings:** Paying close attention to compiler warnings is important for most every standard, including FIO Operations. Addressing compiler warnings is critical to addressing potential errors and vulnerabilities present in code. Compiler Warnings may warn the developer of potential issues with FIO Operations, such as forgetting to close a file. Paying attention to all compiler warnings is essential.
2. **Keep it Simple:** When utilizing FIO Operations, it is important that developers keep the implementation simple. Overly complex FIO Operation implementations can be less effective and confusing for a team of developers.
3. **Default Deny:** Adopting the principle of Default Deny is necessary for secure FIO Operations. Only allowing users with the need to access memory the ability to do so is an important step in eliminating many potential avenues of attack, and provides another layer in a robust DiD implementation.
4. **Practice Least Privilege:** Similar to the logic behind Default Deny, adopting the Principle of Least Privilege is essential within FIO Operations. If a user needs access to memory, they should only be granted read/write capability based on their purpose. This narrowed scope will help mitigate from attack as well as potentially limit the damage of a breach, and is an essential layer in a robust DiD implementation.
5. **Sanitize Data Sent to Other Systems:** Sanitizing data is essential for FIO Operations. When interacting with an external system, developers must ensure that data passed to and received from these systems is properly sanitized to adhere to the standard. This will help prevent FIO vulnerabilities like traversal attacks.
6. **Practice DiD:** Adopting DiD in relation to FIO Operations is paramount. Proper validation, error handling, permissions, and tamper detecting all reinforce a DiD system. Providing a diverse and effective array of security measures is a must for any developer dealing with FIO Operations.
7. **Utilize Quality Assurance:** Utilizing robust quality assurance methods is critical throughout development, reinforcing the principle of design for security. Developers should employ a range of tailored tools to mitigate and remove as many vulnerabilities as possible throughout development. For a list of recommended tools relating to FIO specifically, please see the ‘Automation’ section below.
8. **Adopt a Secure Coding Standard:** Adopting and maintaining a secure standard is important throughout development. In relation to FIO Operations, this will help the developer to manage operations properly. By adopting and applying a secure standard for FIO Operations, we can ensure the most efficient, error-free development possible.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Possible	Medium - High	Medium	2

Automation

Tool	Version	Checker	Description Tool
CPPCheck	2.12	Comprehensive Static Analysis	CPPCheck can perform a comprehensive static analysis when all checks are turned on. This will check for a variety of violations and style errors in C++ code. [https://cppcheck.sourceforge.io/]
Microsoft/MSVS Code Analysis	Visual Studio 17.8.2	Static Analysis	Microsoft Visual Studios comes with a built-in static analysis tool that should be enabled in the project

Tool	Version	Checker	Description Tool
			properties for further coverage. To see extended usage see link below. https://learn.microsoft.com/en-us/cpp/build/reference/analyze-code-analysis?view=msvc-170
Clang-Tidy	18.0.0	Static Analysis	Clang-Tidy is a static analysis tool that can check for a variety of errors. For implementation in C++ see the following link. This tool can be used in conjunction with the previous. https://chat.openai.com/c/26243feb-b3a1-418f-9ef2-de1c3d2c60f5

Coding Standard 10

Coding Standard	Label	Name of Standard
Naming, Documentation, Comments Standard	[STD-010-CPP]	This coding standard aims to establish consistent and secure practices for documentation, commenting, and naming convention. Adhering to this standard will reduce the risk of vulnerabilities and exploits as well as enhance code quality and readability. 1. **Developers Must Utilize Comments to Enhance Readability:** Developers are required to utilize comments judiciously to improve code readability and company efficiency. Comments should be utilized to provide insight into complex algorithms or logic not immediately clear from viewing the code. 2. **Developers Must Provide Documentation (multiple options available):** Developers are required to supply documentation for their code, offering insight into the purpose, usage, and potential vulnerabilities. Potential options include commenting, README files, usage and explanation of exceptions, or specific documentation tools. 3. **Developers Must Maintain a Consistent and Logical Naming Convention:** Developers must follow a clear and consistent naming convention for variables, functions, classes, and other code elements. This is in an effort to improve code readability and increase company efficiency. 4. **Do Not Over-Comment:** Developers should be cautious to not over-comment their work, as this can have the opposite intended effect, reducing overall code readability and company efficiency. 5. **Regularly Update Comments:** Developers are required to keep their comments up-to-date with the code in an effort to maintain readability and efficiency. Out-of-date comments can lead to discrepancy and confusion, and should be avoided.

Noncompliant Code

The following code demonstrates improper adherence to the standard. This example does not include any comments, in violation of the principle.

```
// Rule 1: Developers Must Utilize Comments to Enhance Readability
int calculateSum(int a, int b) {
    return a + b;
} // Non-compliant, no use of commentating.
```

The following code demonstrates improper adherence to the standard. Similar to the previous example, this code includes no form of documentation, in violation of the principle.

```
// Rule 2: Developers Must Provide Documentation
int calculateSum(int a, int b) {
    return a + b;
} // Non-compliant, no form of documentation
```

Noncompliant Code

The following code demonstrates improper adherence to the standard. The example employs an illogical and confusing naming convention, in violation of the principle.

```
// Rule 3: Developers Must Maintain a Consistent and Logical Naming Convention
//
int cAlCuLaTeSuM(int a, int b) {
    return a + b;
} // Non-compliant, does not follow adequate or logical naming convention.
```

The following code demonstrates improper adherence to the standard. The example includes excessive comments, causing clutter, reducing readability, and violating the principle.

```
// Rule 4: Do Not Over-Comment
// Violation: Needless commenting clutters code.
// type int utilized
// function named calculateSum
int calculateSum(int a, int b) { // int a and b utilized to calculate sum
    return a + b; // returns the sum of a + b
} // ends the function
```

The following code demonstrates improper adherence to the standard. The example includes out-of-date comments causing confusion and reducing readability.

```
// Rule 5: Regularly Update Comments
// Non-compliant: Comment is not updated after code changes
int calculateSum(int a, int b) {
    // Add a, b, and c
    return a + b;
}
```

Compliant Code

The following code demonstrates proper adherence to the standard. The example includes the addition of a comment to describe the purpose or operation of the function. This is in concordance with the principle.

```
// Rule 1: Developers Must Utilize Comments to Enhance Readability
// Compliant: Adding a comment to explain the purpose of the function
int calculateSum(int a, int b) {
    // Add a and b
    return a + b;
}
```

The following code demonstrates proper adherence to the standard. Similar to the previous example, this code includes a comment, a form of documentation, abiding by the principle.

```
// Rule 2: Developers Must Provide Documentation
// Compliant: Adding a comment to explain the purpose of the function
```

Compliant Code

```
int calculateSum(int a, int b) {
   // Add a and b
   return a + b;
}
```

The following code demonstrates proper adherence to the standard. The corrected example applies the conventional naming convention and is in alignment with the principle.

```
// Rule 3: Developers Must Maintain a Consistent and Logical Naming Convention
int calculateSum(int a, int b) {
   return a + b;
} // Compliant, utilizes proper naming convention.
```

The following code demonstrates proper adherence to the standard. The corrected example removes unnecessary and redundant comments, and it now abides by the principle.

```
// Rule 4: Do Not Over-Comment
// Compliant: Does not include unnecessary code (example is purposefully simple)
// Function to calculate the sum of a and b
int calculateSum(int a, int b) {
   return a + b;
}
```

The following code demonstrates proper adherence to the standard. The corrected example features comments that have been updated to reflect the current state of the code. This is in alignment with the principle.

```
// Rule 5: Regularly Update Comments
// Compliant: Updated comment to reflect changes in code
int calculateSum(int a, int b) {
   // Add a and b
   return a + b;
}
```

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

Standard: Naming, Documentation, Comments Standard
Principles(s):

1. **Keep it Simple:** Choosing clear, straightforward names for variables, functions, and other documentary entities is essential for team efficiency. Avoid unnecessary or overly complex explanations in the code, and comments should be descriptive yet concise.
2. **Utilize Effect Quality Assurance:** Although not directly related to commenting, many static analysis tools can alert the developer when comments are incorrect to the program's function, or take on a different style than the majority. As such, tools should be configured to include this functionality. For a list of recommended tools relating to Naming, Documentation, and Comments specifically, please see the 'Automation' section below.
3. **Adopt a Secure Standard:** Adhering to the set standard for documentation is required of all developers. This will be essential for team and project efficiency as well as help to mitigate vulnerability within the code.

Threat Level

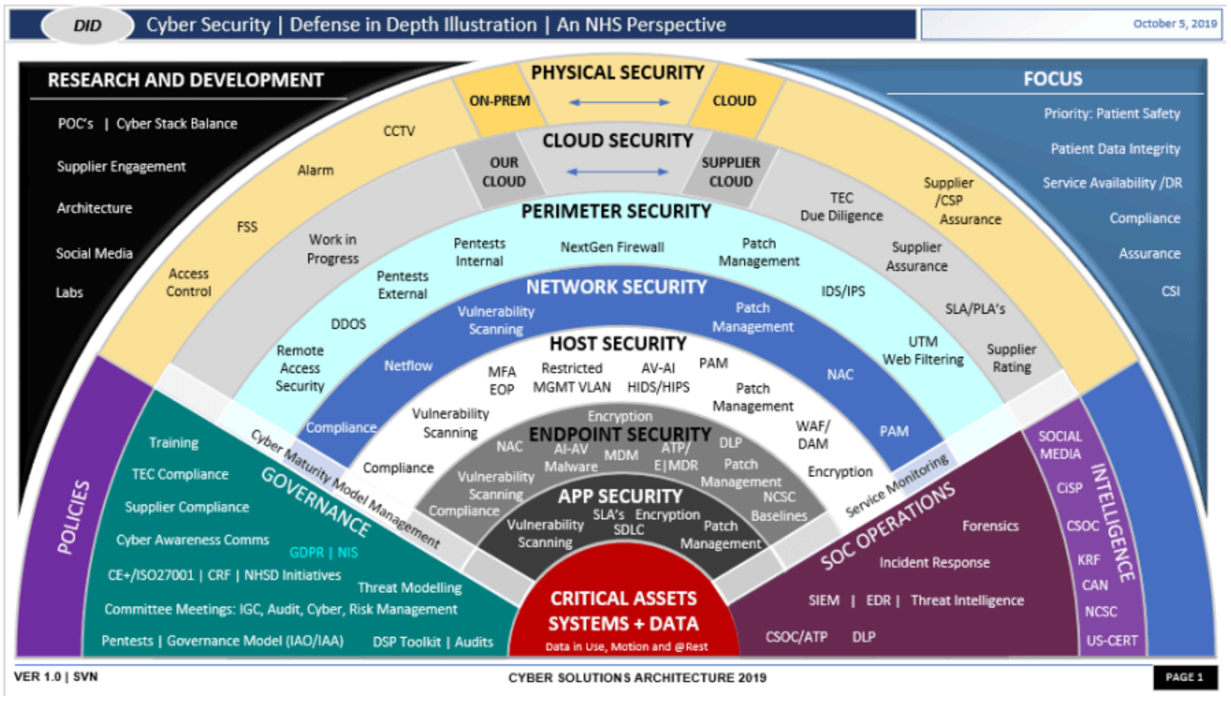
Severity	Likelihood	Remediation Cost	Priority	Level
Low	Possible	Low	Low	1

Automation

Tool	Version	Checker	Description Tool
CPPCheck	2.12	Comprehensive Static Analysis	CPPCheck can perform a comprehensive static analysis when all checks are turned on. This will check for a variety of violations and style errors in C++ code. This can also be utilized to detect incorrect or missing comments. [https://cppcheck.sourceforge.io/]
Clang-Tidy	18.0.0	Static Analysis	Clang-Tidy is a static analysis tool that can check for a variety of errors. For implementation in C++ see the following link. This tool can be similarly used to detect inconsistencies in comments. https://chat.openai.com/c/26243feb-b3a1-418f-9ef2-de1c3d2c60f5

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

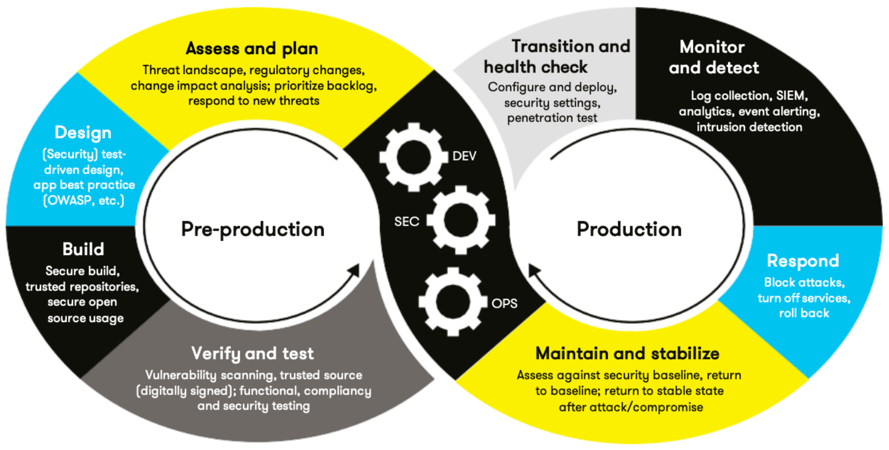
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, 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.


Assess and plan
Threat landscape, regulatory changes, change impact analysis; prioritize backlog, respond to new threats
Design
(Security) test-driven design, app best practice (OWASP, etc.)
Build
Secure build, trusted repositories, secure open source usage
Verify and test
Vulnerability scanning, trusted source (digitally signed); functional, compliancy and security testing
Pre-production
DEV
SEC
OPS
Transition and health check
Configure and deploy, security settings, penetration test
Monitor and detect
Log collection, SIEM, analytics, event alerting, intrusion detection
Production
Respond
Block attacks, turn off services, roll back
Maintain and stabilize
Assess against security baseline, return to baseline; return to stable state after attack/compromise

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.

Transitioning to DevSecOps will ultimately be a wise and beneficial decision for Green Pace. As discussed previously in this document, many of the potential mitigations for common vulnerabilities can, and should be implemented early-on in the development cycle. Considering security from the beginning provides many benefits to an application's health and robustness, including the ability to plan implementation for both software and hardware security features. DevSecOps also features the capability of automating many of the processes involved in secure development, further increasing project efficiency relative to traditional methods of development.

In order to integrate automation into the development process while at the same time maintaining application security and integrity, it is important to understand the DevSecOps pipeline and how automation could be applied to each step. As such, we will analyze how each individual step could implement automation:

Access and Plan: Planning and Assessment involves defining requirements, setting goals, establishing a roadmap, and accessing necessary environments and resources for the work. Although much of this process is creatively motivated and not automated, it can benefit from programs like project management tools, backlog generation, and virtual tracking systems. Additionally, IaC tools can be utilized to aid in the configuration process for project environments.

Design: The design phase of DevSecOps can be heavily automated to detect and mitigate vulnerabilities and deviations from the standard. Although the actual code will need to be implemented largely by-hand on the part of a developer, there are many static analysis tools, such as the ones mentioned previously, that can help a developer quickly identify and solve vulnerabilities and errors. These tools should be heavily utilized throughout design to ensure that defects do not get baked further into development.

Build: Similar to the design phase, the build phase of DevSecOps can be heavily automated. Utilizing continuous integration tools, code can be automatically built and compiled when changes are committed. Concurrently, dynamic analysis tools can be integrated to perform analysis during the build process, working in conjunction with static tools to identify vulnerabilities.

Verify and Test: Automated testing is a key aspect of the verification stage in DevSecOps. This can include, but is not limited to unit testing, integration testing, and security testing. Test automation tools can aid developers by running test suites quickly and consistently, ensuring changes do not introduce vulnerability or error into the code base.
Transition and Health Check: The transition phase of DevSecOps involves deploying the application into production. Automation can feature in this stage in the form of continuous deployment tools. These tools aid developers by automatically validating if changes to a codebase are correct, stable, and ready for immediate autonomous deployment to a production environment.
Monitor and Detect: Automation plays a pivotal role in monitoring and detecting defects within a deployed application. Automated monitoring tools can continuously track the performance and health of a deployed application. These tools can alert developers to any noteworthy changes in the environment, enabling development teams to quickly identify and subsequently respond to issues or threats.
Respond: Although automation is not directly involved in the response portion of DevSecOps, as this will require human intervention, there are roles that automation can play in this phase. Automated systems have the capability to detect and isolate compromised systems, as well as rollback to previous deployments in the result of catastrophic failure. Human intervention is required for this phase, but automation can still play a part in assisting developers to work efficiently.
Maintain and Stabilize: Many routine maintenance processes can be automated, such as updating and patch management. Human intervention will similarly always be required for this phase of DevSecOps, but automation also has a part to play.

Ultimately, DevSecOps is a similar but innovative process to DevOps. By integrating security concerns coupled with automation at the various steps of development, teams are enabled to create robust secure applications as efficiently or even more efficiently than before. Recent development trends have shifted towards iterative, continuous development. DevSecOps is the natural progression of iterative development strategies, only now beneficially incorporating security. Through the use of DevSecOps, Green Pace will be on track to produce applications with the same efficiency and heightened security when compared to a standard DevOps pipeline.

Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STD-001-CPP	High	Unlikely	Medium	High	2
STD-002-CPP	High	Likely	Medium - High	High	3
STD-003-CPP	High	Likely	Medium	High	2 - 3
STD-004-CPP	High	Likely	High - Medium	High	3
STD-005-CPP	Critical	Likely	High - Medium	Maximum	3
STD-006-CPP	Low	Unlikely	Low	Medium	1
STD-007-CPP	Medium	Possible	Medium	Medium	2
STD-008-CPP	High	Likely	High - Medium	High	3
STD-009-CPP	Medium	Possible	Medium - High	Medium	2

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STD-010-CPP	Low	Possible	Low	Low	1

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***.***

a. Explain each type of encryption, how it is used, and why and when the policy applies.
b. 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.

a. Encryptio n	**Explain what it is and how and why the policy applies.**
Encryption in rest	Encryption at rest is a critical part of application security and a robust DiD system. Encryption at rest involves securing data when it is stored in non-volatile storage, such as a DB, file, or backup harddrive. All sensitive and confidential data must be stored in an encrypted format, **NEVER** in plaintext. Developers are required to employ industry standard encryption algorithms, such as AES. Encryption at rest is essential because it mitigates the risk of data breach should a malicious agent gain access to the data. It provides a final layer of security within a DiD system.
Encryption at flight	Encryption in flight is similarly important to that at rest. Encryption in flight refers to protecting data as it travels over a network, preventing unauthorized interception or eavesdropping. Communication over networks, both internal and external, must be encrypted using industry standard secure transport layer protocols (TLS/SSL). Developers must implement end-to-end encryption for data transmission, utilize strong ciphers, and ensure protocols are regularly updated to address vulnerabilities. Encryption at flight safeguards data as it travels. Were this step not taken, any attacker intercepting the information would have access to it, similar to the rationale for encrypting data at rest, it provides a last line of defense should the DiD layers be breached.
Encryption in use	Encryption in use or confidential computing focuses on protecting data while it is being actively processed or utilized by an application. On both the server and client sides, all data being processed in memory or during computational tasks must be encrypted. Developers should employ encryption libraries and technologies that allow for the processing of sensitive data in memory, such as Crypto++. Encryption in use ensures that sensitive data remains protected during active processing, further reducing the risk of vulnerability and attack.

b. Triple-A Framework *	**Explain what it is and how and why the policy applies.**
Authentication	Authentication is an important factor in computer security, and a key portion of AAA. Authentication is the process of verifying the identity of users, systems, or entities attempting to gain access to a network or system (User logins). In-line with Default Deny and Least

b. **Triple-A Framework ***	**Explain what it is and how and why the policy applies.**
	Privilege all users, systems, and entities must undergo authentication before gaining access to network information, resources, or applications (Changes to the db or Addition of new users). Authentication helps to ensure that only authorized individuals gain access to the network or system. This will help minimize the risk of malicious agents taking advantage of the system or its resources.
Authorization	Authorization is an important factor in computer security, and a key portion of AAA. Authorization is the process of determining the level of access or permissions that a viable user has access to within the network (User level of access). Aligned with Default Deny and Least Privilege, all users, systems, and entities are assigned a level of access based on role and responsibility, only being allowed to access what is necessary (Files accessed by users). It is imperative to define and implement role-based access control (RBAC) and attribute-based access control (ABAC) mechanisms within the system. Access should be regularly analyzed to update permissions as is necessary.
Accounting	Accounting is an important factor in computer security, and a key portion of AAA. Accounting refers to the process of tracking and logging activities related to the other A's, providing an auditable trail for accountability and security (Changes to the db). It is important to record and monitor all authentication and authorization events, successful or failed. This will allow maintenance to generate and review logs regularly for suspicious activities or security breach incidents. Developers should implement centralized logging mechanisms, security information and event management systems (SIEM). Configure the systems properly to ensure logs capture relevant information. Accounting serves a crucial role in AAA, providing necessary information to audit, analyze, and monitor the system.

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

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

Map the Principles

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

NOTE: Green Pace has already successfully implemented the following:

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

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

Audit Controls and Management

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

Evidence will include the following:

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

Enforcement

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

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

Exceptions Process

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

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

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

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

Distribution

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

Policy Change Control

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

Policy Version History

Version	Date	Description	Edited By	Approved By
1.0	08/05/2020	Initial Template	David Buksbaum	
1.1	11/11-11/12/ 2023	Milestone 3 Completion. Includes definition of security principles and some completion of coding standard section as per requirements.	Kyle Dale	[Insert text.]
1.2	12/2/2023	Completed the summary of risk assessments table, risk assessment for each standard, and the relevant automation tool for each standard.	Kyle Dale	[Insert text.]
1.2.1	12/3/2023	Completed the principle-standard mapping, DevSecOp automation integration write-up, and the policy for Encryption and AAA. Proofread document.	Kyle Dale	

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV