Discussion 4: iostrem

Discussion Topic:

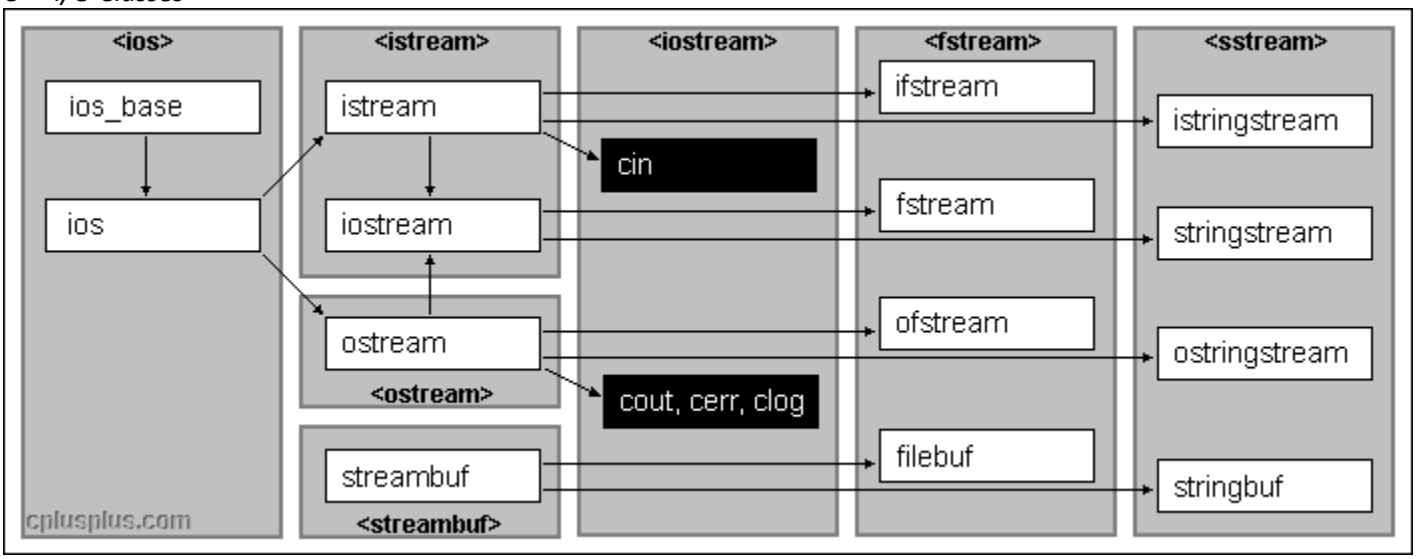
In creating C++ applications, you have the ability to utilize various formatting functions in the iostream library.
What are some of the formatting vulnerabilities that can be encountered in using the iostream library in C++?
What tips can be utilized to identify these vulnerabilities?
Be sure to provide an appropriate source code example to illustrate your points.

My Post:

Hello Class,

In C++, the 'iostrem' library is an object-oriented library that provides input and output functionality using streams (cpluplus, n.d.). A stream is an abstraction representing a device on which input and output of operations can be performed. In other words, it is a representation of a source or destination of characters with indefinite length (can handle an unknown or variable amount of data). Streams can be many things, for example, they can handle input and output from a disk file, keyboard, console, or even in-memory strings. The diagram below provides an illustration of the C++ I/O classes and how they relate to each other.

Figure 1
C++ I/O Classes



Note: The diagram shows the hierarchical structure and relationships between different C++ I/O classes, such as input streams ('istream'), output streams ('ostream'), and file streams ('fstream'), as well as the buffers associated with of them. From "Input/Output" by cpluplus (n.d.).

The ‘iostrem’ library brings flexibility on how to manipulate input and output operations. However, it also introduces vulnerabilities, especially when formatting input data is handled improperly. Below is a list of the common formatting vulnerabilities that can emerge when using the ‘iostrem’ library.

1. **Buffer overflow and data truncation,** this can happen by not having a built-in mechanism that checks boundaries (Snyk Security Research Team, 2022). For example, user input can lead to data truncated or bindery overflow when formatting functions like ‘std::setw’ are used improperly.
Truncated Code example, what *not* to do:

```
#include <iostream>
#include <iomanip>
#include <string>

void printInput(const std::string& input) {
    // Using std::setw without checking input length
    std::cout << "Formatted Input: " << std::setw(10) << input << std::endl;
}

int main() {
    std::string userInput;
    std::cout << "Enter a word: ";
    std::cin >> userInput;

    // This could cause data truncation if input is longer than 10 characters
    printInput(userInput);

    return 0;
}
```

2**. Inconsistent data formatting**, the I/O streams have formatting states that once set need to be reset whenever manipulating the data with different manipulators (Moria, 2017). For example when using a combination of manipulators like ‘std::hex’, ‘std::oct’, or ‘std::dec’. If the states are not reset, the outputs may retain the format of the previous manipulator, resulting in incorrect data representation.

Code example, what not to do:

```
#include <iostream>
#include <iomanip>

void displayValues() {
    int number = 255;

    // Apply std::hex to print the number in hexadecimal format
    std::cout << "Hexadecimal: " << std::hex << number << std::endl;

    // Print the number again without resetting the manipulator
    std::cout << "Decimal (Incorrect): " << number << std::endl;

    // Correct approach: reset to std::dec for decimal output
    std::cout << "Decimal (Correct): " << std::dec << number << std::endl;
}

int main() {
    displayValues();
    return 0;
}
```

Outputs:

```
Hexadecimal: ff
Decimal (Incorrect): ff
Decimal (Correct): 255
```

3. **Stream adjustor side-effects,** when using manipulators that adjust stream such as width, precision, or fill characters (e.g., 'std::setprecision', 'std::setfill'), if not used properly, can result in unpredictable results. This can affect the subsequent output throughout the program.

Code example, what not to do:

```
#include <iostream>
#include <iomanip>

void displayNumbers() {
    double num1 = 3.14159;
    double num2 = 42;

    // Set precision and fill for num1
    std::cout << "Formatted num1: " << std::setprecision(4) << std::fixed << num1 <<
std::endl;

    // no resetting, these settings affect num2 as well
    std::cout << "Formatted num2 (Incorrect): " << num2 << std::endl;

    // Correct approach: reset manipulators for num2
    std::cout << "Formatted num2 (Correct): " << std::setprecision(0) << num2 <<
std::endl;
}

int main() {
    displayNumbers();
    return 0;
}
```

Output:

```
Formatted num1: 3.1416
Formatted num2 (Incorrect): 42.0000
Formatted num2 (Correct): 42
```

4. **Thread safety concerns**, 'std::cout' is thread-safe for individual character operations; However concurrent writes from multiple threads can result in interleaved output, see code example below:

Code example, what not to do:

```
#include <iostream>
#include <thread>

void print_message(const std::string& message) {
    std::cout << message << std::endl;
}

int main() {
    std::thread t1(print_message, "Hello from thread 1");
    std::thread t2(print_message, "Hello from thread 2");

    t1.join();
    t2.join();

    return 0;
```

```
}
```

Outputs

```
Hello from thread 1Hello from thread 2
```

A solution is to add a mutex locks

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex cout_mutex;

void print_message(const std::string& message) {
    std::lock_guard<std::mutex> lock(cout_mutex);
    std::cout << message << std::endl;
}

int main() {
    std::thread t1(print_message, "Hello from thread 1");
    std::thread t2(print_message, "Hello from thread 2");

    t1.join();
    t2.join();

    return 0;
}
```

Output:

```
Hello from thread 1
Hello from thread 2
```

All the examples above illustrate vulnerabilities that can be exploited by malicious actors if not properly addressed. Below is a list of tips for identifying and mitigating these vulnerabilities:

- Use tools for static analysis to detect buffer-related issues, tools such as Polyspace, Astrée, Splint, Asan, etc. These tools are very useful in identifying such vulnerabilities, especially when the source code consists of millions of lines.
- Enforce data type boundaries by integrating built-in boundary checks into the program code and performing tests to validate that the data is handled properly.
- After modifying stream states with manipulators, reset them or use local copies of streams.
- Use proper thread safety by using mechanics such as ‘mutex’ locks when accessing shared resources like in multithreaded applications.

To summarize, the ‘iostrem’ library is an object-oriented library that provides input and output functionality using streams, it provides flexible input/output operations. However, it also introduces vulnerabilities if not managed carefully. Vulnerabilities like buffer overflow, inconsistent formatting, and thread safety concerns. By applying best practices such as resetting stream states, performing boundary checks, and using thread-safety mechanisms, these Vulnerabilities can be mitigated.

-Alex

References:

cpluplus (n.d.). *Input/Output*. Cpluplus. https://cplusplus.com/reference/iolibrary/

Moria (2017, May 13). *IOStream Is Hopelessly Broken.* Moria. https://www.moria.us/articles/iostream-is-hopelessly-broken/

Snyk Security Research Team (2022, August 16). *Top 5 C++ security risks*. Snyk. https://snyk.io/blog/top-5-c-security-risks/