CWE-119: C17 vs C++20 vs Rust

CWE-119 is a common weakness in software development that involves buffer overflow or buffer overrun. It occurs when a program tries to write data beyond the boundaries of a buffer, leading to memory corruption and potential security vulnerabilities.

C17

C17 does not have built-in protection against buffer overflow. However, modern features such as bounds checking and address sanitization can help mitigate the risk of CWE-119.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    char buffer[10];
    int index = 0;

    printf("Enter a string: ");
    fgets(buffer, sizeof(buffer), stdin); // read input from user

    // bounds checking
    if (index < sizeof(buffer)) {
        buffer[index] = '\0'; // add null terminator
    }

    printf("Your string is: %s\n", buffer);
    return 0;
}
```

In this example, we use `fgets()` to read input from the user and store it in a buffer of size 10. We also use bounds checking to ensure that the index is within the size of the buffer before adding a null terminator to the string.

C++20

C++20 introduces the `std::span` class, which provides a safe and efficient way to work with arrays and buffers. It allows us to specify the size of the buffer and provides bounds checking to prevent buffer overflow.

```cpp
#include <iostream>
#include <span>

int main() {
    char buffer[10];
    int index = 0;
    std::cout << "Enter a string: ";
    std::cin.get(buffer, sizeof(buffer)); // read input from user

    // bounds checking
    std::span<char> span(buffer, index);
    if (span.size() < sizeof(buffer)) {
        buffer[index] = '\0'; // add null terminator
    }

    std::cout << "Your string is: " << buffer << std::endl;
    return 0;
}
```

In this example, we use `std::cin.get()` to read input from the user and store it in a buffer of size 10. We also use `std::span` to provide bounds checking and ensure that the index is within the size of the buffer before adding a null terminator to the string.

Rust

Rust provides built-in protection against buffer overflow through its ownership and borrowing system. It ensures that memory is managed safely and prevents common programming errors such as null pointer dereferencing and buffer overflow.

```
use std::io::{self, Read};

fn main() -> io::Result<()> {
    let mut buffer = [0; 10];
    let mut index = 0;

    print!("Enter a string: ");
    io::stdin().read(&mut buffer)?; // read input from user

    // bounds checking
    if index < buffer.len() {
        buffer[index] = 0; // add null terminator
    }

    println!("Your string is: {}", String::from_utf8_lossy(&buffer));
    Ok(())
}
```

In this example, we use `io::stdin().read()` to read input from the user and store it in a buffer of size 10. We also use bounds checking to ensure that the index is within the size of the buffer before adding a null terminator to the string.

Rust’s ownership and borrowing system ensures that memory is managed safely and prevents buffer overflow.