

Top System Programming Vulnerabilities



Top System Programming Vulnerabilities

| | |
|--|-----------|
| 1. Buffer overflow | 2 |
| Noncompliant Code (Vulnerable to Buffer Overflow): | 2 |
| CPP | 2 |
| C | 3 |
| Rust | 4 |
| Compliant Code (Safe from Buffer Overflow): | 5 |
| CPP | 5 |
| C | 6 |
| Rust | 7 |
| 2. Integer overflow and underflow | 8 |
| Noncompliant Code (Vulnerable to Integer Overflow): | 8 |
| CPP | 8 |
| C | 9 |
| Rust | 10 |
| Compliant Code (Protected from Integer Overflow): | 11 |
| CPP | 11 |
| C | 12 |
| Rust | 13 |
| 3. Pointer initialization | 13 |
| Noncompliant Code (Vulnerable to Dereferencing Uninitialized Pointer): | 14 |
| CPP | 14 |
| C | 14 |
| Rust | 15 |
| Compliant Code (Proper Pointer Initialization): | 16 |
| CPP | 16 |
| C | 17 |
| Rust | 17 |
| 4. Incorrect type conversion | 18 |
| Noncompliant Code (Vulnerable to Incorrect Type Conversion): | 18 |
| CPP | 18 |
| C | 19 |
| Rust | 20 |
| Compliant Code (Proper Type Conversion): | 21 |
| CPP | 21 |
| C | 22 |
| Rust | 23 |

| | |
|--|-----------|
| 5. Format string vulnerability | 24 |
| Noncompliant Code (Vulnerable to Format String Attacks): | 24 |
| CPP | 24 |
| C | 25 |
| Rust | 26 |
| Compliant Code (Safe against Format String Attacks): | 26 |
| CPP | 26 |
| C | 27 |
| Rust | 28 |

1. Buffer overflow

Noncompliant Code (Vulnerable to Buffer Overflow):

CPP

```
#include <stdio.h>

int main(int argc, char **argv)
{
    volatile int important_data = 0;
    char user_input[10];

    gets(user_input); // Vulnerable function

    if(important_data != 0) {
        printf("Warning !!!, the 'important_data' was changed\n");
    } else {
        printf("the 'important_data' was not changed\n");
    }
}
```

In the code above, the `gets()` function doesn't perform any bounds checking on the input, which allows a buffer overflow if the user provides an input larger than `user_input` array's size.

C

```
#include <stdio.h>
#include <string.h>

int main() {
    char buffer[10];
    // The data_to_corrupt variable is placed right after the buffer in
    the stack.
    int data_to_corrupt = 0;

    printf("Enter a long string to cause a buffer overflow: ");
    // gets() is a dangerous function that does not check the length of
    the input.
    // It will continue to write data past the end of the buffer if the
    input is too long.
    gets(buffer);

    // If a buffer overflow occurred, data_to_corrupt may have been
    altered.
    if (data_to_corrupt != 0) {
        printf("Buffer overflow occurred! data_to_corrupt = %d\n",
data_to_corrupt);
    } else {
        printf("No buffer overflow. data_to_corrupt = %d\n",
data_to_corrupt);
    }

    return 0;
}
```

- ☐ buffer is a character array that can hold 10 bytes.
- ☐ gets(buffer) is used to read a string from the standard input without checking the size, which can lead to a buffer overflow if the input is longer than 9 characters (plus the null terminator).
- ☐ data_to_corrupt is an integer variable that will be placed in memory directly after buffer on the stack. If buffer overflows, data_to_corrupt can be overwritten with arbitrary values.

Rust

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

fn main() {
    let mut important_data = 0;
    let mut buffer = [0u8; 10];

    // Mimic unsafe behavior similar to gets() in C
    // This is unsafe and not recommended!
    unsafe {
        let buffer_ptr = buffer.as_mut_ptr();
        let buffer_size = buffer.len();
        let mut input = io::stdin().bytes();

        // Read bytes directly into the buffer without bounds checking
        for i in 0.. {
            if let Some(Ok(byte)) = input.next() {
                *buffer_ptr.add(i) = byte; // Potential buffer overflow
                if input > 10 bytes
                    if byte == b'\n' {
                        break;
                    }
                } else {
                    break;
                }
            }
        }

        // Simulate a situation where the buffer overflow could
        // overwrite important_data
        if buffer_size < 12 {
            // This is where the overflow would affect important_data
            *buffer_ptr.add(11) = 0xFF; // Deliberately writing outside
            // of the buffer
        }
    }

    if important_data != 0 {
        println!("Warning !!!, the 'important_data' was changed");
    } else {

```

```
        println!("The 'important_data' was not changed");
    }
}
```

- ☐ We use an unsafe block to allow us to perform raw pointer operations.
- ☐ We create a raw pointer to the buffer and manually index into it, which is not bounds-checked.
- ☐ We deliberately write outside the bounds of the buffer to simulate a buffer overflow that overwrites important_data.

Compliant Code (Safe from Buffer Overflow):

CPP

```
#include <stdio.h>

int main(int argc, char **argv)
{
    volatile int important_data = 0;
    char user_input[10];

    fgets(user_input, sizeof(user_input), stdin); // Safe function

    if(important_data != 0) {
        printf("Warning !!!, the 'important_data' was changed\n");
    } else {
        printf("the 'important_data' was not changed\n");
    }
}
```

- ☐ In this compliant version, the `fgets()` function is used instead of `gets()`. The `fgets()` function takes a size argument which prevents reading more characters than the buffer can hold, hence preventing a buffer overflow.

C

```
#include <stdio.h>
#include <string.h>

#define BUFFER_SIZE 10

int main() {
    char buffer[BUFFER_SIZE];
    int data_to_protect = 0;

    printf("Enter a string up to 9 characters: ");
    // Use fgets to prevent buffer overflow. It reads up to
    BUFFER_SIZE-1 characters
    // and appends a null terminator.
    if (fgets(buffer, BUFFER_SIZE, stdin) == NULL) {
        printf("Error reading input.\n");
        return 1; // Return an error code if input fails
    }

    // Remove newline character if present, as fgets includes it in the
    buffer.
    buffer[strcspn(buffer, "\n")] = 0;

    // The rest of your code remains unchanged
    if (data_to_protect != 0) {
        printf("Buffer overflow occurred! data_to_protect = %d\n",
data_to_protect);
    } else {
        printf("No buffer overflow. data_to_protect = %d\n",
data_to_protect);
    }

    return 0; // Return a success code
}
```

```
}
```

- ☐ BUFFER_SIZE is defined as the size of the buffer, including space for the null terminator.
- ☐ fgets(buffer, BUFFER_SIZE, stdin) is used to read input from stdin. It reads up to BUFFER_SIZE - 1 characters and appends a null terminator, preventing buffer overflow.
- ☐ buffer[strcspn(buffer, "\n")] = 0; is used to remove the newline character that fgets() reads and includes in the buffer if there is one.
- ☐ The data_to_protect variable should remain unchanged if the input is within the buffer's limits, demonstrating that no buffer overflow has occurred.

Rust

```
use std::io;

fn main() {
    let mut important_data = 0;
    let mut user_input = String::new();

    println!("Please enter some input (up to 9 characters): ");

    // Read input from the user safely
    match io::stdin().read_line(&mut user_input) {
        Ok(_) => {
            // Truncate the input to the first 9 characters to prevent
            overflow
            user_input.truncate(9);

            // Here we would have logic that could potentially change
```



```

important_data
    // For the sake of this example, let's say if the user types
    "change",
        // we change the important_data to 1.
        if user_input.trim() == "change" {
            important_data = 1;
        }
    },
    Err(error) => println!("Error reading input: {}", error),
}

// Check if important_data has been changed
if important_data != 0 {
    println!("Warning: the 'important_data' was changed");
} else {
    println!("The 'important_data' was not changed");
}
}

```

- ☐ We use `String::new()` to create a new, empty `String` for `user_input`.
- ☐ We use `io::stdin().read_line(&mut user_input)` to safely read a line of input from the user into `user_input`. This method handles buffer sizing and grows as needed, preventing buffer overflow.
- ☐ We use `user_input.truncate(9)` to ensure that only the first 9 characters of the input (if any) are considered, preventing any logic that follows from acting on unexpected data.
- ☐ We use `user_input.trim()` to remove any leading or trailing whitespace, which is common when dealing with user input from the console.

2. Integer overflow and underflow

Noncompliant Code (Vulnerable to Integer Overflow):

CPP

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

int main() {
    unsigned int nresp;
    char **response;

    scanf("%u", &nresp); // Read an unsigned integer from input

    // This may cause an integer overflow if nresp is very large
    response = (char**) malloc(nresp * sizeof(char*));

    if (response == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // ... rest of the code ...

    free(response);
    return 0;
}
```

If `nresp` is a large value, multiplying it by `sizeof(char*)` can cause an overflow, resulting in a smaller memory allocation than expected.

C

```
#include <stdio.h>
#include <limits.h>

int main() {
    unsigned int uint_max = UINT_MAX;
    unsigned int overflowed_value = uint_max + 1;
}
```

```

printf("Maximum value for unsigned int: %u\n", uint_max);
printf("Value after overflow: %u\n", overflowed_value);

int int_max = INT_MAX;
int overflowed_int_value = int_max + 1;

printf("Maximum value for int: %d\n", int_max);
printf("Value after overflow: %d\n", overflowed_int_value);

return 0;
}

```

UINT_MAX is the maximum value for an unsigned integer, and INT_MAX is the maximum value for a signed integer.

Adding 1 to UINT_MAX causes an overflow in the unsigned integer, which is well-defined in C: it wraps around to 0.

Adding 1 to INT_MAX causes an overflow in the signed integer, which is undefined behavior in C. The program may wrap around to INT_MIN, but since it's undefined, it could also crash or behave unpredictably.

Rust

```

fn main() {
    let large_value: u32 = std::u32::MAX - 1;
    let addend: u32 = 2;

    // This is unsafe and will cause undefined behavior if the
    // assumption is wrong
    unsafe {
        let result = large_value.unchecked_add(addend);
        println!("Result of the overflow: {}", result);
    }
}

```

unchecked_add is used within an unsafe block. This method is unsafe because it does not check for overflow, allowing for silent wraparound.

This code will compile and run without panicking even in debug mode, but it is considered undefined behavior according to Rust's safety rules.

Compliant Code (Protected from Integer Overflow):

C++

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

int main() {
    unsigned int nresp;
    char **response;

    scanf("%u", &nresp); // Read an unsigned integer from input

    // Check for potential integer overflow
    if (nresp > (UINT_MAX / sizeof(char*))) {
        printf("Input too large!\n");
        return 1;
    }

    response = (char**) malloc(nresp * sizeof(char*));

    if (response == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // ... rest of the code ...

    free(response);
    return 0;
}
```

- ❑ In this compliant code, we added a condition to check if multiplying nresp with sizeof(char*) will cause an overflow by dividing the maximum unsigned integer (UINT_MAX) by sizeof(char*) and comparing it against nresp. If nresp exceeds the result of this division, then the multiplication will result in an overflow, and the program will output an error message and exit.

C

```
#include <stdio.h>
#include <limits.h>

int main() {
    unsigned int uint_max = UINT_MAX;
    unsigned int a = uint_max - 1;
    unsigned int b = 2;

    // Check for overflow before performing the addition
    if (a > UINT_MAX - b) {
        printf("Error: unsigned int overflow detected\n");
    } else {
        unsigned int result = a + b;
        printf("Result of unsigned int addition: %u\n", result);
    }

    int int_max = INT_MAX;
    int c = int_max - 1;
    int d = 2;

    // Check for overflow before performing the addition
    if (c > 0 && d > INT_MAX - c) {
        printf("Error: signed int overflow detected\n");
    } else {
        int result = c + d;
        printf("Result of signed int addition: %d\n", result);
    }

    return 0;
}
```

- ☐ Before adding two unsigned integers, we check if the first integer *a* is greater than `UINT_MAX - b`. If this is true, adding *b* to *a* would cause an overflow.
- ☐ Before adding two signed integers, we check if *c* is positive and if *d* is greater than `INT_MAX - c`. If this is true, adding *d* to *c* would cause an overflow.

Rust

```
fn main() {  
    let a: u32 = std::u32::MAX;  
    let b: u32 = 1;  
  
    // Use checked_add to prevent overflow  
    match a.checked_add(b) {  
        Some(result) => println!("Result of addition: {}", result),  
        None => println!("Error: unsigned integer overflow detected"),  
    }  
  
    let c: i32 = std::i32::MAX;  
    let d: i32 = 1;  
  
    // Use checked_add to prevent overflow  
    match c.checked_add(d) {  
        Some(result) => println!("Result of addition: {}", result),  
        None => println!("Error: signed integer overflow detected"),  
    }  
}
```

- ☐ `checked_add` is used to perform the addition. If the addition does not overflow, it returns `Some(result)`, where `result` is the sum of *a* and *b*.
- ☐ If the addition would cause an overflow, `checked_add` returns `None`, and the program prints an error message instead of panicking or wrapping around.

3. Pointer initialization

Noncompliant Code (Vulnerable to Dereferencing Uninitialized Pointer):

CPP

```
#include <stdio.h>

void main() {
    int *ptr; // uninitialized pointer

    if (nullptr != ptr) { // incorrect check, can lead to undefined
behavior
        *ptr = 5;
        printf("%d\n", *ptr);
    }
}
```

In the above code, the pointer `ptr` is uninitialized. Dereferencing an uninitialized pointer can result in undefined behavior. The check against `nullptr` is also incorrect since `nullptr` is a C++ keyword and not valid in C. The behavior is unpredictable since `ptr` might contain a garbage value.

C

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

int main() {
    int *initialized_pointer = malloc(sizeof(int)); // Allocate memory
    for an integer
    if (initialized_pointer == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }

    *initialized_pointer = 42; // Dereference the initialized pointer
    safely

    printf("Value of the data pointed to by initialized_pointer: %d\n",
    *initialized_pointer);

    free(initialized_pointer); // Free the allocated memory

    return 0;
}
```

`uninitialized_pointer` is declared but not initialized; it points to some arbitrary memory location. The program then attempts to write the value 42 to this location, which is unsafe because the pointer could be pointing anywhere. Finally, the program attempts to print the value from the same location, which is also unsafe.

Rust

```
fn main() {
    let uninitialized_pointer: *mut i32; // Declare a raw pointer but do
    not initialize it

    unsafe {
        // UNSAFE: Dereferencing an uninitialized pointer is undefined
    }
}
```



```

behavior
    *uninitialized_pointer = 42; // Attempt to write through the
    uninitialized pointer

    println!("Value of the data pointed to by uninitialized_pointer:
    {} ", *uninitialized_pointer);
    }
}

```

data is a valid variable with an initial value.

initialized_pointer is a raw pointer that is initialized to the address of data.

The unsafe block is used to dereference the raw pointer, but this time it is safe because the pointer is initialized to a valid memory address.

Compliant Code (Proper Pointer Initialization):

CPP

```

#include <stdio.h>

void main() {
    int *ptr = NULL; // properly initialized pointer

    if (ptr != NULL) { // correct check
        *ptr = 5; // This code will not be executed because ptr is NULL
        printf("%d\n", *ptr);
    }
}

```

- ☐ In this compliant code, the pointer ptr is properly initialized to NULL. The check against NULL ensures that the pointer is not dereferenced when it is null, preventing any undefined behavior.

C

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

int main() {
    int *initialized_pointer = malloc(sizeof(int)); // Allocate memory
    for an integer
    if (initialized_pointer == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }

    *initialized_pointer = 42; // Dereference the initialized pointer
    safely

    printf("Value of the data pointed to by initialized_pointer: %d\n",
    *initialized_pointer);

    free(initialized_pointer); // Free the allocated memory

    return 0;
}
```

- ☐ Memory is allocated for an integer using malloc, and initialized_pointer is set to point to this memory.
- ☐ Before dereferencing the pointer, the code checks if malloc returned NULL, which would indicate that the memory allocation failed.
- ☐ After using the allocated memory, free is called to deallocate the memory, preventing a memory leak.

Rust

```
fn main() {  
    let mut value = 0; // A mutable variable with an initial value  
    let value_ptr = &mut value as *mut i32; // Create a raw pointer to  
    `value`  
  
    unsafe {  
        // SAFE: Dereferencing a pointer to memory we own is okay within  
        an unsafe block  
        *value_ptr = 42; // Dereference the pointer to write to `value`  
        println!("Value: {}", *value_ptr); // Dereference the pointer to  
        read `value`  
    }  
}
```

- ☐ value is a variable that is properly initialized.
- ☐ value_ptr is a raw pointer that is created by taking the address of value.
- ☐ An unsafe block is used to dereference value_ptr. This is necessary because all operations on raw pointers are considered unsafe in Rust. However, this is compliant with Rust's safety rules because we know that value_ptr points to valid memory.

4. Incorrect type conversion

Noncompliant Code (Vulnerable to Incorrect Type Conversion):

CPP

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string str;
    cout << "Please enter your string: \n";
    getline(cin, str);
    unsigned int len = str.length();
    if (len > -1)
    {
        cout << "string length is " << len << " which is bigger than -1 " << std::endl;
    }
    else
    {
        cout << "string length is " << len << " which is less than -1 " << std::endl;
    }
    return 0;
}
```

As you've explained, due to the incorrect type conversion, comparing an unsigned int with -1 will always result in the else branch being executed.

C

```
#include <stdio.h>

int main() {
    long large_number = 9223372036854775807; // Maximum value for a
    signed long on a 64-bit system
    int truncated_number = (int)large_number; // Incorrectly casting to
    a smaller type
}
```

```

printf("Original large number: %ld\n", large_number);
printf("Truncated number: %d\n", truncated_number);

// Incorrect pointer type conversion
double pi = 3.141592653589793;
int *pi_int_pointer = (int *)&pi; // Cast a double pointer to an int
pointer

printf("Incorrectly interpreted value of pi: %d\n",
*pi_int_pointer);

return 0;
}

```

large_number is a long that contains the maximum value that can be stored in a signed long on a 64-bit system. Casting it to an int can truncate the value, leading to data loss and undefined behavior.

pi is a double, but its address is cast to an int*, and then dereferenced. This violates strict aliasing rules and leads to undefined behavior because the memory representation of a double is being incorrectly interpreted as an int.

Rust

```

fn main() {
    let large_number: i64 = 9223372036854775807; // Maximum value for
i64
    let truncated_number = large_number as i32; // Unsafe truncation

    println!("Original large number: {}", large_number);
    println!("Truncated number: {}", truncated_number);

    // Unsafe pointer type conversion
    let pi: f64 = 3.141592653589793;
    let pi_pointer = &pi as *const f64 as *const i32; // Cast a f64

```

pointer to an i32 pointer

```
unsafe {
    println!("Incorrectly interpreted value of pi: {}",
*pi_pointer);
}
```

large_number is cast to i32, which can lead to truncation and data loss because i64 can represent a wider range of values than i32.

pi is a f64, and its pointer is cast to a pointer to i32. Dereferencing this pointer is undefined behavior because f64 and i32 have different sizes and memory representations.

Compliant Code (Proper Type Conversion):

CPP

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string str;
    cout << "Please enter your string: \n";
    getline(cin, str);
    size_t len = str.length(); // 'size_t' is an unsigned type
    representing the size of objects in bytes.
    if (len > static_cast<size_t>(-1)) // Casting -1 to size_t for
    proper comparison.
    {
        cout << "string length is " << len << " which is bigger than -1
" << std::endl;
    }
    else
    {

```

```

        cout << "string length is " << len << " which is less than -1 "
    <<std::endl;
    }
    return 0;
}

```

- ☐ In the compliant code, we utilize `size_t` which is the appropriate type for representing sizes. Additionally, instead of comparing directly with `-1`, we do a `static_cast` to ensure a correct type conversion and avoid issues related to signed/unsigned mismatch.

C

```

#include <stdio.h>
#include <limits.h>

int main() {
    long large_number = 9223372036854775807; // Maximum value for a
    signed long on a 64-bit system

    // Check before casting
    if (large_number > INT_MAX || large_number < INT_MIN) {
        printf("Error: large_number is out of range for an int\n");
    } else {
        int safe_number = (int)large_number;
        printf("Safely casted number: %d\n", safe_number);
    }

    return 0;
}

```

- ☐ For the pointer conversion, you should avoid casting pointers to different types unless you are certain of the type of the object being pointed to and that the memory access will not violate alignment requirements or strict aliasing rules. If you need to interpret the bytes of a type as another type, you can use `memcpy`:

```
#include <stdio.h>
#include <string.h>

int main() {
    double pi = 3.141592653589793;
    int pi_int;

    // Use memcpy to safely copy the bytes
    memcpy(&pi_int, &pi, sizeof(pi_int));

    printf("Value of pi copied into an int without type punning: %d\n",
pi_int);

    return 0;
}
```

- ☐ In this compliant version, `memcpy` is used to copy the bytes of `pi` into `pi_int` without violating strict aliasing, although the resulting `pi_int` value will not make much sense as it's just a bit pattern copied from `pi`. It's important to note that this kind of byte-wise copy is rarely meaningful and should be used with caution.

Rust

```
fn main() {
    let large_number: i64 = 9223372036854775807;

    // Safe conversion using `try_into`
    match large_number.try_into() {
        Ok(truncated_number) => println!("Safely converted number: {}",
truncated_number),
        Err(e) => println!("Conversion error: {:?}", e),
    }

    // Safe transmutation of `f64` to `i64` using `to_bits`, which is a
safe operation
    let pi: f64 = 3.141592653589793;
    let pi_bits: u64 = pi.to_bits();

    // Now, if you really need to interpret these bits as i32 (which is
unusual), you can do so safely:
    let pi_bits_high: i32 = (pi_bits >> 32) as i32; // High 32 bits
    let pi_bits_low: i32 = pi_bits as i32; // Low 32 bits

    println!("High 32 bits of pi: {}", pi_bits_high);
    println!("Low 32 bits of pi: {}", pi_bits_low);
}
```

- ☐ Instead of using `as` for casting (which can be unsafe), `try_into()` is used for the conversion, which safely converts types and returns a `Result` indicating success or failure of the conversion.
- ☐ `pi.to_bits()` is used to safely convert the `f64` value into its bit pattern as a `u64`. This is a safe operation in Rust and does not involve any undefined behavior.
- ☐ If you need to work with the bit pattern of `pi` as `i32`, you can safely split the `u64` into two `i32`s. This is still a bit unusual and should be done with a clear understanding of why you're accessing the raw bits of a floating-point number.

5. Format string vulnerability

Noncompliant Code (Vulnerable to Format String Attacks):

CPP

```
#include <stdio.h>

int main(int argc, char **argv)
{
    // Unsafe usage of printf
    printf(argv[1]);
    return 0;
}
```

This code directly takes user input (`argv[1]`) and uses it as a format string, opening up the potential for format string attacks. An attacker could provide malicious format specifiers as arguments.

C

```
#include <stdio.h>

int main() {
    char user_input[100];

    // Assume user_input is filled with data from an untrusted source
    printf("Please enter your name: ");
    gets(user_input); // Unsafe use of gets(), which is also deprecated

    // Vulnerable to format string attack
    printf(user_input);
}
```

```
    return 0;
}
```

The gets function is used, which is unsafe because it does not check the length of the input and can lead to buffer overflows.

The printf function is directly passed user_input without format specifiers, which is a format string vulnerability.

Rust

```
extern "C" {
    fn printf(format: *const i8, ...);
}

fn main() {
    let user_input = String::from("user input with %s format specifier");
    unsafe {
        // Unsafe and noncompliant: passing user-controlled input to a C-style printf
        printf(user_input.as_ptr() as *const i8);
    }
}
```

This example assumes you have a C printf function accessible from Rust, which is not normally the case. It's purely hypothetical and for illustrative purposes.

Compliant Code (Safe against Format String Attacks):

CPP

```
#include <stdio.h>

int main(int argc, char **argv)
{
    // Safe usage of printf
    printf("%s\n", argv[1]);
    return 0;
}
```

- ☐ In this version, we specify "%s\n" as the format string, ensuring that the user input is only interpreted as a string and not as additional format specifiers. This prevents format string attacks by displaying the string argument as is, without interpretation of any format specifiers that might be included in the input.
- ☐ For those writing in modern C++, as mentioned, the C++20 std::format offers even better type-safety than the C-style format string functions.

C

```
#include <stdio.h>

#define MAX_INPUT_LENGTH 100

int main() {
    char user_input[MAX_INPUT_LENGTH];
```

```

// Safe way to read strings using fgets()
printf("Please enter your name: ");
if (fgets(user_input, MAX_INPUT_LENGTH, stdin) == NULL) {
    printf("Error reading input.\n");
    return 1;
}

// Secure way to use user input in printf
printf("%s", user_input);

return 0;
}

```

- ☐ fgets is used instead of gets to safely read user input, avoiding buffer overflow by specifying the maximum length of the input.
- ☐ The printf function is used with a format specifier %s, which prevents format string vulnerabilities. The user input is treated as a string and not as a format string.

Rust

```

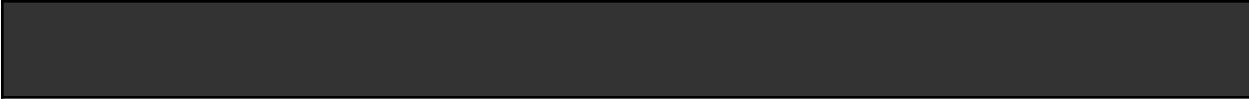
use std::io::{self, Write};

fn main() {
    let mut user_input = String::new();

    print!("Please enter your name: ");
    io::stdout().flush().unwrap(); // Ensure "Please enter your name: "
    is printed before input

    match io::stdin().read_line(&mut user_input) {
        Ok(_) => {
            // Safe and compliant: Rust's macro `println!` requires
            format specifiers
            println!("Hello, {}", user_input.trim_end()); // trim_end
            removes the newline
        },
        Err(error) => println!("Error reading input: {}", error),
    }
}

```

- 
- ☐ Rust's `println!` macro is used, which enforces the use of format specifiers (`{}`) for any inserted content.
 - ☐ User input is safely appended into the format string without the risk of it being interpreted as a format specifier.