Milestone 2, Group-7

Group Members:

Arnav Goyal - 2021A7PS2596G
Aryan Nambiar - 2021A7PS2619G
Pranav Bajpai - 2021A7PS2062G
Amey Patil - 2021A7PS2740G

Our Project

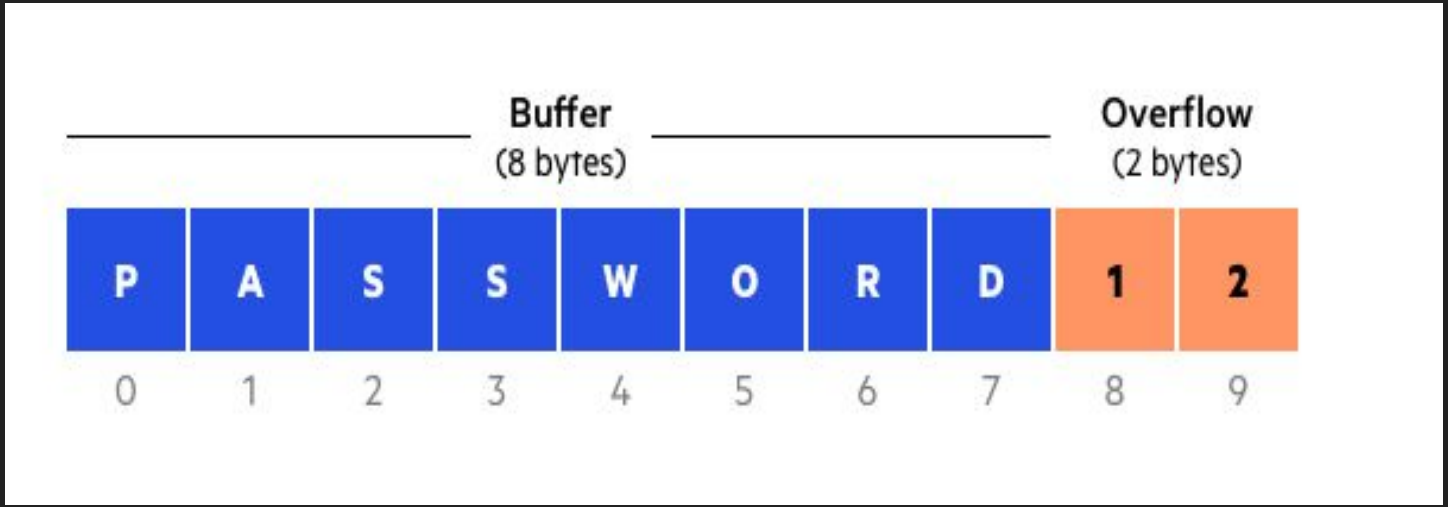
Implementing a multithreaded web server in Rust and showcasing its concurrency and security advantages over C.

Advantages Over C

1. Prevention of Buffer Overflow Attack
2. Dangling Pointer Vulnerability
3. Concurrency Advantages
4. Size of Code

Buffer Overflow Attack

A buffer overflow attack occurs when a program writes more data to a buffer than it can hold. Since buffers are created to contain a finite amount of data, the extra information—which has to go somewhere—can overflow into adjacent buffers, corrupting or overwriting the data they contain. In the context of security, this behavior can be exploited to cause a program to behave in an unexpected way, including the execution of malicious code.



How does Rust Help?

Why is C prone to buffer overflow attacks?
C is particularly vulnerable to buffer overflows because it does not perform automatic bounds checking on arrays.

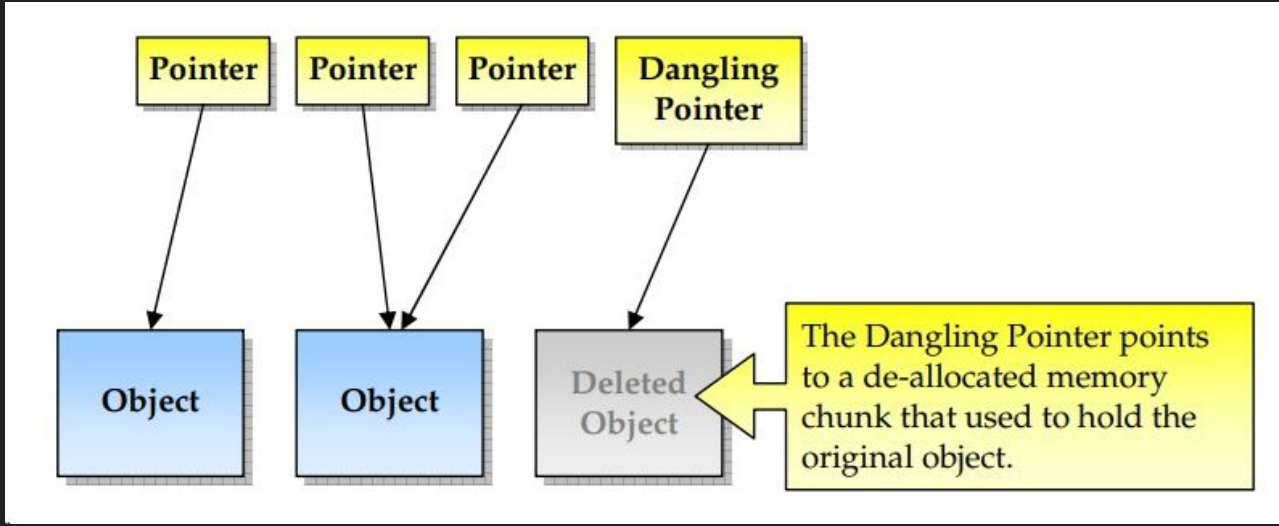
How does Rust prevent this attack?

1. **Bounds Checking**: Rust automatically checks the bounds of arrays. If you try to access an array beyond its length, Rust will panic at compile time, preventing undefined behavior and possible security vulnerabilities.
2. **Slice Type**: Rust's slices are a view into a block of memory represented as a pointer and a length. Slices always keep track of their size, and accessing elements beyond the size is prevented at compile time or causes a panic at runtime.
3. **Pattern Matching**: Rust encourages the use of pattern matching to unpack options and results, which makes the code explicit about cases where there may or may not be a value.
4. **Safe Abstractions**: Rust provides safe abstractions like Vec for a dynamically sized list, which manages buffer sizes and resizing safely behind the scenes.

```rust
match stream {
    Ok(client) => {
        std::thread::spawn(|| {
            handle_client(client);
        });
    }
    Err(e) => {
        eprintln!("Failed to establish a connection: {}", e);
    }
}
```

Dangling Pointer Vulnerability

When writing an application, developers typically include pointers to a variety of data objects. In some scenarios, the developer may accidentally use a pointer or an invalid or deallocated object, causing the application to crash, but can result in far more dangerous behavior. This class of bug is referred to as the Dangling Pointer or the Use-After-Free problem and is every bit as dangerous as buffer overflow.



Dangling pointer vulnerabilities can be a significant concern in web servers developed in C or C++. These vulnerabilities can arise due to the complex and concurrent nature of web server applications. In the context of web servers, several scenarios can lead to dangling pointer vulnerabilities:

1. **Multithreading and Concurrency**: Web servers often handle multiple client requests concurrently using threads or processes. Dangling pointer vulnerabilities can occur when one thread attempts to access a resource that has been freed or modified by another thread, leading to data corruption or crashes.
2. **Dynamic Data Structures**: Many web servers dynamically allocate and manage data structures such as linked lists, trees, and buffers. Improper handling of these data structures, like failing to update pointers when elements are removed or altered, can result in dangling pointers.
3. **Asynchronous I/O**: Some web servers use asynchronous I/O operations, which can complicate memory management. If a pointer to a resource is freed while an asynchronous operation is in progress, the pointer could become dangling when the operation completes.

Rust's combination of ownership, borrow checker, and lifetimes, along with a robust type system, contributes to a significant reduction in memory-related vulnerabilities.

Concurrency Comparison Rust vs C

We outline several benefits of Rust's concurrency features over C:

1. **Compile-Time Restrictions**: The language’s powerful compile-time restrictions aid in writing safe and efficient concurrent programs, catching potential errors before they occur at runtime.

2. **Scoped Threads**: With libraries like ‘crossbeam’, Rust allows the creation of scoped threads, enabling more granular control over thread lifetimes and data ownership, which helps prevent data races.

3. **Proactive Debugging**: Although the initial debugging can be challenging due to strict compiler checks and the learning curve, this process encourages the early identification and resolution of potential concurrency issues.

4. **Concurrency Constructs**: Rust provides advanced concurrency constructs like `Mutex<T>` for synchronization, which, despite their complexity, contribute to safer multithreading compared to manual synchronization in C.

5. **Encouragement of Safe Practices**: By enforcing its strict ownership and concurrency rules, Rust compels programmers to verify and prove the thread safety of their code rigorously.

THANK YOU