Rust 언어 메모리 안전 모델에서 스마트 포인터의 역할에 대한 연구

카욘도마틴 [1], 방인영 [2], 백윤흥 [3]
[1,2] 서울대학교 전기정보공학과 박사과정
[3] 서울대학교 전기정보공학과 교수

kayondo@snu.ac.kr, iybang@sor.snu.ac.kr, ypaek@snu.ac.kr

Understanding The Role of Smart Pointers in the Rust Memory

Martin Kayondo[1], Inyoung Bang[1], Yunheung Paek[1]
[1]Dept. of Electrical and Computer Engineering and Inter-University Semiconductor Research Center(ISRC), Seoul National University

Abstraction

Rust has gained popularity as a memory safe systems programming language. At the center of its memory safety is a strict memory ownership model with stringent rules enforced by the compiler. This paper aims to shed light on this memory safety model and the role smart pointers play towards its success. We study specific smart pointers, their purposes and contribution to Rust's memory safety. We further explore weaknesses of these smart pointers and their APIs, and provide scenarios under which they may lead to memory vulnerabilities in Rust programs.

1. Introduction

Rust is a systems programming language that has gained popularity over the years, mainly because of the memory safety it promises, while offering competitive runtime performance close, and sometimes better than C/C++. For this, it has been adopted in the Linux kernel and Android OS. Rust's memory safety model is unique because it uses ownership and borrowing concepts to enforce strict rules that prevent common memory bugs such as use-after-free, buffer overflow and data races. This is especially attractive because compared to C/C++ these rules are enforced automatically by the compiler rather than having memory managed by the programmer as in C/C++. One of the essential features of this memory safety model is the use of smart pointers. The role of smart pointers in Rust is so well defined that in a purely safe Rust program (Safe Rust), all memory operations are performed using smart pointers. Rust code is considered unsafe if there is use of *unsafe blocks*, where the programmer calls an unsafe function, dereferences a raw pointer or makes a routine call to a function from an external library written in a foreign language (FFI). In this work, unsafe Rust and FFI are considered orthogonal because pointers sent to FFI as arguments are strapped into raw pointers, and unsafe Rust has no major link to smart pointers other than exposing this APIs for custom use by programmers – which we explore in section 3.

Unlike other languages that use explicit memory allocation and management or garbage collection, Rust assigns a single owner to each memory object and all other pointers are considered borrowers. This is Rust's ownership model. Under this model, an object may be borrowed either mutable or immutably, but only a single mutable borrower exists at a time. The model also defines lifetimes on borrows to check against the owner's liveliness. When an owner goes out of scope, the owned object is freed and the borrowed object must not outlive the owner, otherwise this would lead to a use-after-free (UAF) bug. Ownership can be transferred to another variable, causing the original owner and all borrowers to lose access to the memory object. This model enforces restricted aliasing, and any violation of these rules leads to a compile error. With a single owner, Rust fundamentally mitigates UAF bugs by ensuring dangling pointers never exist in the program. Thus, all dereferences in Rust are always valid. Lifetimes and ownership rules guarantee that access on any memory object is always valid in terms of liveliness. Similarly, data races are deterred by this model since there is only a single legal writer at a time.

To enforce these rules, the memory model relies on smart pointers. A smart pointer in Rust is a data object with awareness of the Rust memory model. It holds a raw pointer and, in some cases, additional metadata, and its interface (API) provides for borrowing, ownership transfer and cleanup. As expected, however, the rules enforced by the model are too stringent for Rust to offer a reasonable programming experience and to build large scale programs that match today's demands. For example, with the single ownership rule, shared memory either among threads or process is virtually impossible. Therefore, depending on the needs of the programmer, specific smart pointers are available either to ease programmability while withholding ownership rules or to provide additional memory safety.

2. Smart Pointers in Rust

Box: This is the basic heap pointer and the most elementary smart pointer in Rust. It is equivalent to a *malloced* pointer in C/C++, although its API provides automatic cleanup where by the wrapped pointer is free once the owner goes out of scope. All references to a *boxed* object are technically valid and there are no dangling pointers as the *Box* API supports both borrowing and ownership transfer as explained above.

Vec: This is a commonly used versatile smart pointer in Rust for dynamic arrays. It acts as a heap-allocated buffer and stores not only a pointer to the buffer, but also metadata such as buffer capacity and currently used length. The *Vec* API provides for borrowing, buffer *slicing*, and ownership transfer. It also offers secure indexing, where every index is checked against the length metadata to prevent overflows.

Rc (Reference Counted): As aforementioned, the ownership model usually only allows for a single owner for a memory object. This makes it impossible to develop certain data structures such as doubly linked lists in Rust. As a workaround, *Rc* provides for multiple owners of a single memory object at the same time. It maintains a reference counter to keep track of the number of owners of the shared objects, which is automatically incremented when a new pointer to the same object is formed and decremented when any of the owners goes out of scope. When the counter drops to zero, the memory object is finally deallocated. It is important to note that although *Rc* provides for multiple ownership, in order to support the single mutability rule, all references it provides are immutable and only have read access to the memory object.

Cell and RefCell: These are referred to as interior mutability pointers. They are special in a way that they cannot be dereferenced directly and the programmer would have to use their APIs to read or write to them. Their APIs provides a hidden mutability, making it implicitly legal to write they store. A *Cell or RefCell* appears immutable on the surface, but the programmer may use the *set* method to write to them. *Cell* allows for direct modification of the stored value using its *get* and *set* methods, while *RefCell* allows for the creation of borrowed references using *borrow* and *borrow_mut* methods. When used with *Rc, Cell* and *RefCell* can provide safe access to mutable data even in presence of multiple references. *Cell* enforces ownership and borrowing rules at compile time through its API, whereas *RefCell* tracks borrow counts through metadata and enforces rules at runtime.

Smart pointers discussed above ensure enforce memory safety rules only in single-threaded environments. In multithreaded environments, shared objects are inevitable. With the single ownership rule, it is difficult to design shared objects among threads. An obvious solution would be using *Rc,* but since it relies on the reference counter metadata, synchronization issues in the would affect the validity of the reference counter, resulting in memory bugs. Rust provides more smart pointers specifically for multithreaded environments as explored below:

Arc (Atomically reference counted pointer): Like *Rc*, *Arc* allows multiple parts of a program to hold references to a shared value. However, unlike *Rc, Arc* uses atomic reference counting and locking to ensure that these references are safely managed across multiple threads. This makes it safe to share ownership of a value between threads, as long as all references are declared as *Arc.*

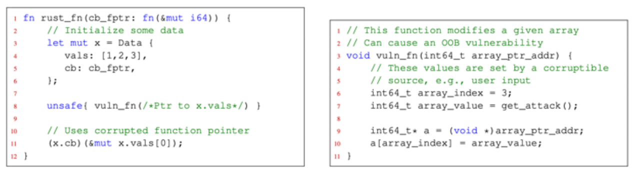
Mutex and **RwLock**: Like *Rc, Arc* enables only immutable shared references. To solve this limitation, *Mutex* can be used. *Mutex* locks the data before access and unlocks it after. This ensures that data races are prevented in multithreaded environments. However, *Mutex* comes with the overhead of locking, even for read-only access. *RwLock* is a more efficient alternative as it only locks on write accesses, making it a better choice for memory objects with more read operations compared to write operations. *Mutex* relies on the lock and poison flag metadata to enforce safety, where the position flag is set when a thread holding the lock *panics*, alerting other threads sharing the object about possible invalidity of the pointer due to the *panic.*

3. Smart Pointer Weaknesses

In the previous section, we explored the different smart pointers in Rust and how they help enforce ownership rules. However, as mentioned, some of these smart pointers store metadata on which the memory model depends to enforce memory safety. On the other hand, some smart pointers solely depend on their API and the programmer's proficiency to ensure memory safety rules are withheld. In this section, we investigate two cases in which smart pointers may be used by an attacker to mislead the memory model and cause vulnerabilities.

Metadata Overwriting:

Consider a case where an attacker is able to overwrite the metadata of a smart pointer. As a result, such an attack controls the operation of the memory model at runtime and may cause the program the execute on their plans. This is especially possible in Rust programs that make calls into FFI routines. An FFI function with a buffer overflow bug maybe exploited to overwrite smart pointer metadata as shown in Figure 1.



```
fn rust_fn(cb_fptr: fn(&mut i64)) {
    // Initialize some data
    let mut x = Data {
       vals: [1,2,3],
       cb: cb_fptr,
    };

    unsafe{ vuln_fn(/*Ptr to x.vals*/) }

    // Uses corrupted function pointer
    (x.cb)(&mut x.vals[0]);
}
```

(a) Rust code that calls C/C++ to modify a Rust struct.

```
// This function modifies a given array
// Can cause an OOB vulnerability
void vuln_fn(int64_t array_ptr_addr) {
    // These values are set by a corruptible
    // source, e.g., user input
    int64_t array_index = 3;
    int64_t array_value = get_attack();

    int64_t* a = (void *)array_ptr_addr;
    a[array_index] = array_value;
}
```

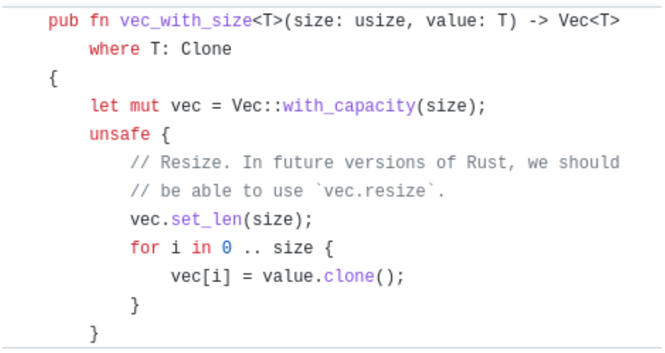
(b) C/C++ code that performs an Out-of-Bounds (OOB) error.

(Figure 1) A possible overflow from an FFI function can affect smart pointer metadata [1]

Suppose *x* in Figure 1 (a) is a *Vec* smart pointer, an attacker may use leverage the overflow in the FFI function in Figure 1 (b) to overwrite the length metadata of the smart pointer. Further checks on such a pointer to prevent heap overflows by the memory model would be unsound, hence an attacker can mislead the model to cause a heap-buffer overflow. This kind of attack is possible on other smart pointers that rely on metadata to enforce memory safety.

API misuse:

Another scenario is when programmers misuse the smart pointer APIs, causing unsound behavior at run time. Some smart pointers expose their APIs for experienced programmers to modify the metadata at will. However, such uses are usually considered unsafe and must be wrapped by the *unsafe* keyword. In the Rust world, it is etiquette to add a comment on such blocks explaining why the programmer believes what they are doing is still safe, nonetheless, some programmers misuse these APIs in many cases. Figure 2 shows one such case in which the programmer uses the *set_len* function of the *Vec* smart pointer, setting the length metadata with a potentially unsound value. This example is a reported CVE [2], proving that such bugs are prevalent in Rust programs.



```
pub fn vec_with_size<T>(size: usize, value: T) -> Vec<T>
    where T: Clone
{
    let mut vec = Vec::with_capacity(size);
    unsafe {
        // Resize. In future versions of Rust, we should
        // be able to use `vec.resize`.
        vec.set_len(size);
        for i in 0 .. size {
            vec[i] = value.clone();
        }
    }
```

(Figure 2) Smart pointer API misuse leading to potential unsound behavior.

Several API misuses cases such as that in Figure 2 have been reported. It is evident that most of these issues are due to use of unsafe Rust, or FFI. As a result, several works [3, 4] have proposed solutions to improve Rust's memory safety by isolation, but to best of our knowledge, there has not been any work outlining the need to review smart pointer safety.

4. Conclusion

In this work, we have explored Rust's memory model and its reliance on smart pointers. We explain how different smart pointers are used and how they contribute to Rust's memory safety. However, we also show that even though these a core to Rust's memory safety model, they themselves are susceptible to corruption and may rather mislead a Rust program into undefined behavior or even make it susceptible to memory attacks. With this, we advise Rust programmers to use unsafe smart pointer APIs with extra care, and possibly isolate FFI with any of the cited previous works to enhance provided safety.

5. Acknowledgement

This work was supported by the National Information Society Agency, funded by the Ministry of Science and ICT in 2023 (No. 2021-0-00528), to develop standard protocols for hardware-based trusted computing and distributed data protection platforms.

References

[1] Mergendahl Samuel, Nathan Burow, and Hamed Okhravi, "Cross-language attacks.", In Proceedings 2022 Network and Distributed System Security Symposium., NDSS, vol 22, pp.1-17, 2022

[2] https://nvd.nist.gov/vuln/detail/CVE-2021-28037 "CVE-2021-28037 Detail"

[3] Liu Peiming, Gang Zhao, and Jeff Huang, "Securing unsafe rust programs with XRust", Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 234-245, 2020

[4] Kirth, Paul, et al, "PKRU-Safe: automatically locking down the heap between safe and unsafe languages.", Proceedings of the Seventeenth European Conference on Computer Systems, pp. 132-148, 2022.