An Analysis of the Rust Programming Practice for Memory Safety Assurance

Baowen Xu(✉), Bei Chu, Hongcheng Fan, and Yang Feng

Nanjing University, Nanjing, China
bwxu@nju.edu.cn

Abstract. Memory safety is a critical concern in software development, as related issues often lead to program crashes, vulnerabilities, and security breaches, leading to severe consequences for applications and systems. This paper provides a detailed analysis of how Rust effectively addresses memory safety concerns. The paper first introduces the concepts of ownership, reference and lifetime in Rust, highlighting how they contribute to ensuring memory safety. It then delves into an examination of common memory safety issues and how they manifest in popular programming languages. Rust's solutions to these issues are compared to those of other languages, emphasizing the benefits of using Rust for enhanced memory safety. In conclusion, this paper offers a comprehensive exploration of prevalent memory safety issues in programming and demonstrates how Rust effectively addresses them. With its encompassing mechanisms and strict rules, Rust proves to be a reliable choice for developers aiming to achieve enhanced memory safety in their programming endeavors.

Keywords: Memory safety · Rust · Ownership · Reference

1 Introduction

Memory safety has always been a critical concern in software development [16]. In many programming languages, memory safety issues often result in severe consequences for applications and systems.

In the 2022 Common Weakness Enumeration (CWE) Top 25 Most Dangerous Software Weaknesses list, there are four software defects related to memory safety, including Out-of-bounds Write, Out-of-bounds Read, Use After Free, and NULL Pointer Dereference. Among these, Out-of-bounds Write holds the top position in terms of prevalence [9].

In order to address these issues, the Rust programming language emerged [7, 12]. It has garnered significant attention in recent years as a programming language known for building efficient and secure system software [3]. It offers a unique combination of low-level control over system resources, similar to languages like C and C++, while also ensuring memory and concurrency safety through mechanisms like ownership and lifetimes [4].

One of the most distinctive and innovative features of Rust is its ownership system [5]. To facilitate programming, Rust introduces the concept of borrowing [5]. Ownership can be borrowed by creating references to values, allowing multiple parts of the code to access and work with the data without transferring ownership. Besides, Rust uses the concept of lifetimes to track the relationships between references and ensure they remain valid [5].

In this paper, we aim to explain how Rust addresses issues related to memory safety. We presents an investigation into various memory safety issues commonly encountered in programming. By studying these examples, we then proceeded to showcase how Rust tackles these problems through its innovative ownership and lifetime mechanisms. Through our analysis and demonstrations, we provide insights into how Rust effectively mitigates memory safety concerns and promotes safer programming practices.

The remainder of the paper is organized as follows: Sect. 2 presents the mechanisms introduced in Rust to ensure memory safety, including ownership, references, and lifetimes. Section 3 analyzes common memory safety issues and their manifestations in popular programming languages, followed by an exploration of Rust's solutions to these issues. Finally, Sect. 4 provides a concise summary of the article's findings and highlights Rust as a dependable choice for developers seeking robust memory safety guarantees.

2 Backgrounds

In this section, we will introduce the concept of ownership and lifetime in Rust.

2.1 Ownership and Reference

Ownership. In Rust language, ownership is a fundamental concept that governs how memory is handled in Rust programs. While these rules are strictly checked in the complication, Rust programmers obtain a executable program with guaranteed memory safety [5].

With Rust's ownership model, each value has a single owner, typically represented by a variable, at any given time. The owner is responsible for the lifetime and deallocation of the value. When the owner goes out of scope, Rust automatically frees the memory associated with the value.

In contrast to many other programming languages, Rust employs move semantics as the default behavior for assignment operations. This represents a typical case of ownership transfer in Rust. Ownership transfer also occurs when variables are passed as function parameters or returned as function results, adhering to Rust's ownership principles. The ownership system in Rust ensures that memory is properly managed throughout the program's execution, enhancing memory safety and reducing the likelihood of runtime errors related to memory management.

Reference. As previously explained, ownership transfers are not limited to assignment operations. If the caller needs to retain access to the variable passed to the callee after the function call, the variable must be returned to the caller as part of the function's return value.

Undoubtedly, such code lacks conciseness and elegance for developers. Thankfully, Rust has taken this into consideration during its design and offers a feature known as *references* [5]. References allow the usage of values without transferring ownership, providing a more flexible and convenient approach in Rust programming.

In Rust, references do not own the value they point to, which ensures compliance with the ownership rules. Within Rust's ownership mechanism, the process of creating a reference is commonly known as borrowing.

Just like variables, references in Rust can be categorized into two types: immutable references and mutable references.

1. Immutable references, also referred to as shared references, provide read-only access to the referenced value.
   Rust enables multiple immutable references to coexist for the same value concurrently, promoting a shared and concurrent access model.
2. By using `&mut`, we can create mutable references that allow both reading and modifying values. However, Rust enforces a strict rule that only one valid mutable reference can exist for a particular value at any given time.

2.2 Lifetimes

To manage references, Rust introduces the concept of lifetimes [5]. In Rust, every reference has its own lifetime, which can be either explicitly specified by the developer or implicitly inferred by the compiler. The purpose of lifetimes is to ensure that references are valid for as long as we need them to be.

Lifetime annotations in Rust are denoted by an apostrophe (`'`) followed by an identifier, such as `'a`, `'b`, `'c`, and so on. These annotations describe the scope of a reference, indicating how long the reference remains valid within the program.

When writing code, it is typically necessary to follow certain rules regarding lifetimes, which include:

1. References must not outlive the values they refer to.
2. If you store a reference in a variable, the reference must remain valid for the entire duration of the variable's lifetime.
3. If there are multiple references, their lifetimes must intersect properly to satisfy validity requirements.

Developers can utilize explicit lifetime annotations or rely on the compiler's inference to ensure the validity and safety of references.

3 Common Memory Safety Issues and Solutions of Rust

In this section, we will discuss common memory safety issues and how they manifest in existing programming languages.

3.1 NULL Pointer Dereference

Description. In computing, a NULL pointer is a special value assigned to a pointer or reference to indicate that it does not point to a valid object or memory location. A NULL pointer dereference happens when an application attempts to access or manipulate data through a pointer that is expected to point to a valid memory address, but is NULL [11].

Manifestations. Below are some examples of NULL Pointer Dereference in common languages from CWE [11].

```c
void host_lookup(char *user_supplied_addr){
    struct hostent *hp;
    in_addr_t *addr;
    char hostname[64];
    in_addr_t inet_addr(const char *cp);
    validate_addr_form(user_supplied_addr);
    addr = inet_addr(user_supplied_addr);
    hp = gethostbyaddr(addr, sizeof(struct in_addr), AF_INET);
    strcpy(hostname, hp->h_name);
}
```

In this example, the program accepts an IP address input from the user, validates its format, and proceeds to perform a hostname lookup. The hostname is then copied into a buffer. If an attacker supplies an apparently valid address that fails to resolve to a hostname, the `gethostbyaddr()` function would return NULL. Since the code does not verify the return value of `gethostbyaddr()`, a null pointer dereference would subsequently occur in the `strcpy()` function call.

This Android application has registered to handle a URL when sent an intent:

```java
IntentFilter filter = new IntentFilter("com.example.URL");
MyReceiver receiver = new MyReceiver();
registerReceiver(receiver, filter);
public class UrlHandlerReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        if("com.example.URL".equals(intent.getAction())) {
            String URL = intent.getStringExtra("URLToOpen");
            int length = URL.length();
        }
    }
}
```

The application assumes that the URL will always be included in the intent. However, when the URL is not present, the call to `getStringExtra()` will return null, thus causing a null pointer exception when `length()` is called.

Solution of Rust. In order to address the issue of NULL pointer dereference, a different approach is taken in Rust, compared to traditional programming languages. In safe Rust, there is no concept of a null pointer (NULL). Instead, it uses the `Option` type to represent values that may be absent.

The Option type is an enumeration with two variants: `Some` and `None`. Some wraps a concrete value, indicating its presence, while None represents the absence of a value.

In Rust, when attempting to dereference an Option type, it must be pattern matched first to determine whether it is Some or None. Only when the Option type is Some, can its value be safely dereferenced.

Below is a simple Rust code snippet demonstrating how to handle Option types:

```rust
fn main() {
    let number: Option<i32> = Some(35);
    match number {
        Some(num) => {
            println!("The number is: {}", num);
        }
        None => {
            println!("There is no number.");
        }
    }
}
```

Number is an `Option<i32>` type assigned the value `Some(42)`. By pattern matching, we can safely dereference the value and perform corresponding actions based on its type.

3.2 Wild Pointer

Description. Wild pointers are pointers that have not been properly initialized before their first use. Strictly speaking, in programming languages that do not enforce initialization, every pointer is considered a wild pointer initially [17]. The key issue lies in whether the pointer is initialized before its first usage.

Manifestations

```c
1  int main() {
2      int *p; /* wild pointer */
3      *p = 32;
4  }
```

This is a very simple example of wild pointer in C language. We declare a pointer variable, `p`, without specifying its address. In this case, `p` becomes a wild pointer. Since `p` can potentially point to any address, the assignment in line 3 has a high chance of corrupting important and protected memory space, leading to program crashes.

```cpp
#include <iostream>
int main() {
    int *arr;
    for(int i = 0; i < 5; i++)
        std::cout << arr[i] << " ";
    return 0;
}
```

Here is another example of wild pointers in C++ language. In the above program, a pointer `arr` is declared but not initialized. As a result, it is displaying the contents of random memory locations. If we compile and run this program, depending on the compiler and compilation options, it may output five numbers or result in a segmentation fault.

Solution of Rust. Due to Rust's enforcement of RAII (Resource Acquisition Is Initialization), the compiler in Rust checks whether pointers and references are initialized before their first usage.

It is worth noting that in safe Rust, dereferencing raw pointers is not allowed. Therefore, the error message mentioned above is only a part of the story.

3.3 Dangling Pointer

Description. Dangling pointers occur when an object is deleted or deallocated without modifying the value of a pointer that still points to the memory location of the deallocated object [1]. This can lead to unpredictable behavior when the dangling pointer is dereferenced, as the memory may now contain different data.

Manifestations. In many languages, when an object is deleted from memory explicitly or when the stack frame is destroyed upon return, the associated pointers are not automatically modified [17]. As a result, the pointers still point to the same memory location, even though that memory may now be used for other purposes.

```c
{
    char *dp = NULL;
    {
        char c;
        dp = &c;
    }
}
```

For example, in the above code, we declare a variable `c` within an inner scope and assign the address of `c` to the pointer `dp`. However, once `c` goes out of its scope, the memory it occupied is deallocated, leaving `dp` as a dangling pointer.

Another common source of dangling pointers arises from improper usage of memory allocation and deallocation functions, such as `new` and `delete` in `C++`, as demonstrated below.

```cpp
void func() {
    char *dp = new char[SIZE];
    delete[] dp;
}
```

One frequently encountered error is returning the addresses of stack-allocated local variables. When a called function returns, the memory space for these variables is deallocated, resulting in technically undefined values or 'garbage values'.

Solution of Rust. Rust solves the problem of dangling pointers through its ownership and borrowing mechanisms. When a value is bound to a variable in Rust, the variable takes ownership of the value, and when the variable goes out of scope, Rust automatically releases the owned value.

Next comes the difference between Rust and other languages. In languages like C/C++, when we use a pointer or reference to a variable after it has gone out of scope, also known as a dangling pointer or reference, those languages do not prevent us from doing so. However, Rust performs compile-time checks and throws an error.

Rust can perform such checks because it manages references through lifetimes. As mentioned in the context, there are two crucial constraints for lifetimes. During compilation, the Rust compiler attempts to select appropriate lifetimes for references based on these constraints.

Clearly, these two constraints result in conflicting lifetime ranges. Therefore, the Rust compiler detects such situations and throws an error.

3.4 Double Free

Description. Double free errors occur when the memory deallocation function is invoked multiple times with the same memory address as an argument [1,10]. This can result in the corruption of the program's memory management data structures, potentially allowing a malicious user to write values in arbitrary memory locations.

Manifestations

```
char* ptr = (char*)malloc(SIZE);
if (abrt) {
    free(ptr);
}
free(ptr);
```

While some double free vulnerabilities may be as straightforward as the example provided, many are scattered across hundreds of lines of code or even different files.

```
public final class AssetManager {
    @Override
    protected void finalize() throws Throwable {
        if (mObject != 0) {
            nativeDestroy(mObject);
        }
    }
    void xmlBlockGone(int id) {
        if (mNumRefs == 0 && mObject != 0) {
            nativeDestroy(mObject);
            mObject = 0;
        }
    }
}
final class XmlBlock {
    private @Nullable final AssetManager mAssets;
    private void finalize() throws Throwable {
        if (mAssets != null) {
            mAssets.xmlBlockGone(hashCode());
        }
    }
}
```

The above code demonstrates an example of a double free in Java [8]. Assuming we have a XmlBlock `X` created by a AssetManager `A`. After calling

A.close and both X and A are ready to be garbage collected, if A.finalize is called first (nativeDestroy), the subsequent invocation of X.finalize will trigger A.xmlBlockGone, causing the second nativeDestroy of A and resulting in a crash.

Solution of Rust. Rust addresses the issue of double free through its ownership mechanism. In Rust, when an owner goes out of scope, the value is automatically released, eliminating the need for manual resource deallocation by the developer. Rust also enforces the rule, checked at compile time, that each value has only one owner, ensuring that double free does not occur.

Rust achieves automatic resource deallocation through the use of the Drop trait, which is automatically implemented for almost all types in Rust. The Drop trait defines the behavior when a value is dropped, allowing Rust to perform necessary cleanup operations.

In the following code, we define a struct named S and implement a custom Drop trait for it. When running this code, we observe that the program outputs "Drop for S.", which demonstrates that Rust automatically invokes the drop method.

```rust
struct S;
impl Drop for S {
    fn drop(&mut self) {
        println!("Drop for S.")
    }
}
fn main() {
    let s = S;
}
```

What if we manually invoke the drop method of the struct to release the resources ahead of time? If we add the line the compiler will provide the following error message:

```
error[E0040]: explicit use of destructor method
 --> src/main.rs:9:7
  |
9 |     s.drop();
  |     --^^^^--
  |     | |
  |     | explicit destructor calls not allowed
  |     help: consider using `drop` function: `drop(s)`
```

This indicates that Rust does not allow us to explicitly call the Drop::drop method.

3.5 Buffer Overflow

Description. A buffer overflow condition occurs when a program tries to store more data in a buffer than its capacity allows or when it attempts to write data beyond the boundaries of a buffer. In this context, a buffer refers to a consecutive section of memory allocated to hold various types of data, ranging from character strings to arrays of integers [6].

Manifestations

```c
#define BUFSIZE 256
int main(int argc, char **argv) {
    char buf[BUFSIZE];
    strcpy(buf, argv[1]);
}
```

In the above example, the buffer size is fixed, but there is no guarantee that the string in `argv[1]` will not exceed this size, potentially causing a buffer overflow.

```c
#define SIZE 8
int main() {
    int id[SIZE];
    for (int i = 0; i <= SIZE; ++i) {
        id[i] = i * 2;
    }
}
```

The above code attempts to store a series of integers in an array. However, the size of the array is `SIZE`, so its indices range from 0 to `SIZE - 1`. The final assignment statement, `id[SIZE] = SIZE * 2;`, causes a buffer overflow issue.

Solution of Rust. Rust addresses the issue of buffer overflow by performing checks on buffer indices during both compile-time and runtime. For regular arrays, Rust requires specifying the size at declaration or infers it based on the code. For dynamic arrays like `Vec`, Rust stores their length and capacity on the stack.

In Rust, there are two ways to access arrays: one is the common method using the `[]` operator, and the other is through the `get` method. When using `[]`, Rust performs compile-time checks to ensure that the index does not exceed the array bounds if both are known. If the size or index is unknown at compile time, Rust inserts runtime checks. If an out-of-bounds access is detected at runtime, it triggers a panic. When accessing arrays using the `get` method, it returns an `Option` value, and it is the responsibility of the developer to manually verify its validity.

3.6 Use of Uninitialized Memory

Description. Use of uninitialized memory means reading data from a previously allocated memory region that has not been filled with initial values.

In some languages such as C and C++, stack variables are not initialized by default. They often contain random or junk data before the function was invoked [15]. In this case, the behavior of the program is unpredictable, and detecting such issues can be challenging. This type of problem is commonly referred to as a "heisenbug" [2].

Manifestations

```php
if (isset($_POST['names'])) {
    $nameArray = $_POST['names'];
}
echo "Hello " . $nameArray['first'];
```

The above PHP code checks if the `names` array in the POST request is set before assigning it to the variable `$nameArray`. However, if the array is not present in the POST request, `$nameArray` will remain uninitialized. This can result in an error when accessing the array to print the greeting message, potentially creating an opportunity for further exploitation.

The following code snippet represents a Use of Uninitialized Memory issue in the C programming language.

```c
char *str;
if (i != err) {
    str = "Hello World!";
}
printf("
```

If the value of the variable `i` is equal to `err`, then the string `str` in the above code will be in an uninitialized and unknown state. In such a scenario, the `printf` function may print junk strings.

Solution of Rust. Similar to C language, in Rust, stack variables are uninitialized by default and need to be explicitly assigned a value. However, unlike C, Rust prevents you from using variables before initializing them. Rust performs branch analysis to ensure that variables are initialized before they are used on each branch [13].

It is important to note that this check does not consider the specific values of conditions; rather, it takes into account the program's dependencies and control flow.

3.7 Data Race

Description. Data race occurs when multiple threads access and modify a shared memory region simultaneously without proper synchronization [14].

Manifestations

```go
var counter int
func incrementCounter(wg *sync.WaitGroup) {
    defer wg.Done()
    counter++
}
func main() {
    var wg sync.WaitGroup
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go incrementCounter(&wg)
    }
    wg.Wait()
    fmt.Println("Counter value:", counter)
}
```

In this example, multiple goroutines are simultaneously modifying a shared variable called `counter`. Since there is no synchronization mechanism, such as a mutex or a semaphore, in place to coordinate access to the variable, a data race occurs. Data races can lead to inconsistent and unpredictable final values of the `counter` variable.

```c
#include <pthread.h>
#include <stdio.h>
int counter = 0;
void* increment(void* arg) {
    for (int i = 0; i < 1000000; i++) {
        counter++;
    }
    return NULL;
}
int main() {
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, increment, NULL);
    pthread_create(&thread2, NULL, increment, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("Final value of counter:
    return 0;
}
```

In this example, two threads are created, and each thread increments the shared variable `counter` in a loop. Since there is no synchronization mechanism, a data race occurs where both threads are accessing and modifying the variable simultaneously. As a result, the final value of `counter` becomes unpredictable and may vary between different runs of the program.

Solution of Rust. Rust addresses most data race issues through its ownership mechanism. As mentioned earlier, Rust distinguishes between two types of references and imposes corresponding restrictions.

Rust's distinction and restrictions on references serve the purpose of preventing data races during compile time. This concept can be likened to the rules governing readers and writers in concurrent operations. That means it is not possible to have multiple mutable references aliasing the same data simultaneously in Rust.

However, not all types adhere to inherited mutability. Certain types in Rust allow multiple aliases of a memory location while still allowing mutation. Rust addresses this by utilizing the Send and Sync traits to enforce safe concurrent behavior [13].

4 Conclusion

This article provides a comprehensive exploration of prevalent memory safety issues encountered in programming and delves into their manifestations in popular programming languages. It further elucidates how Rust effectively tackles these issues by leveraging the power of ownership, references, and other innovative mechanisms. By enforcing strict rules and guarantees, Rust empowers developers with robust memory safety, ensuring their programs are protected from a range of potential vulnerabilities. With a holistic approach encompassing various mechanisms, Rust stands as a reliable choice for developers seeking enhanced memory safety in their programming endeavors.

References

1. Caballero, J., Grieco, G., Marron, M., Nappa, A.: Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, pp. 133–143 (2012)
2. Grottke, M., Trivedi, K.S.: A classification of software faults. J. Reliab. Eng. Assoc. Jpn. **27**(7), 425–438 (2005)
3. Jiang, H., Wang, L., Tao, X., Hu, H.: RHE: relation and heterogeneousness enhanced issue participants recommendation. In: Xing, C., Fu, X., Zhang, Y., Zhang, G., Borjigin, C. (eds.) WISA 2021. LNCS, vol. 12999, pp. 605–616. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-87571-8_52
4. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: RustBelt: securing the foundations of the rust programming language. Proc. ACM Program. Lang. **2**(POPL), 1–34 (2017)
5. Klabnik, S., Nichols, C.: The Rust Programming Language. No Starch Press (2023)
6. Lhee, K.S., Chapin, S.J.: Buffer overflow and format string overflow vulnerabilities. Softw. Pract. Exp. **33**(5), 423–460 (2003)
7. Matsakis, N.D., Klock, F.S.: The rust language. ACM SIGAda Ada Lett. **34**(3), 103–104 (2014)
8. MITRE: CVE record | CVE. https://www.cve.org/CVERecord?id=CVE-2020-0081. Accessed 25 June 2023
9. MITRE: CWE - 2022 CWE top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html. Accessed 24 June 2023
10. MITRE: CWE - CWE-415: Double free (4.11). https://cwe.mitre.org/data/definitions/415.html. Accessed 24 June 2023
11. MITRE: CWE - CWE-476: null pointer dereference (4.11). https://cwe.mitre.org/data/definitions/476.html. Accessed 24 June 2023
12. Rust Community: Rust programming language. https://www.rust-lang.org/. Accessed 24 June 2023
13. Rust Community: The rustonomicon. https://doc.rust-lang.org/nomicon/. Accessed 25 June 2023
14. Serebryany, K., Iskhodzhanov, T.: ThreadSanitizer: data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications, pp. 62–71 (2009)
15. Stepanov, E., Serebryany, K.: MemorySanitizer: fast detector of uninitialized memory use in C++. In: 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 46–55. IEEE (2015)
16. Szekeres, L., Payer, M., Wei, T., Song, D.: SoK: eternal war in memory. In: 2013 IEEE Symposium on Security and Privacy, pp. 48–62. IEEE (2013)
17. Wikipedia contributors: Dangling pointer—Wikipedia, the free encyclopedia (2023). https://en.wikipedia.org/w/index.php?title=Dangling_pointer&oldid=1155171462. Accessed 24 June 2023