Thesis Dissertation

EXPLORING COMPILER ENFORCED MEMORY SAFETY IN RUST

Antonis Louca

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2022

UNIVERSITY OF CYPRUS
COMPUTER SCIENCE DEPARTMENT

Exploring Compiler Enforced Memory Safety in Rust

Antonis Louca

Supervisor
Dr. Elias Athanasopoulos

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

May 2022

Acknowledgments

First of all, I would like to extend my gratitude to my thesis supervisor Dr. Elias Athanasopoulos, for his advice, support and guidance throughout this project. His support helped me achieve this great milestone.

I would also like to express my gratitude to all of my university professors, here at the University of Cyprus, who these past four years tutored us and provided us with comprehensive knowledge and increased my curiosity about the Computer Science field.

Last but not least, a huge thank you to my family and friends who provided unconditional support and love during this learning adventure.

Abstract

Programming systems can be safe and unsafe. Safe systems guarantee secure memory access, while unsafe systems have no security guarantees. Unsafe systems suffer from programming bugs that can be exploited by attackers. Safe systems depend on their run-time environment to avoid such bugs. While safe systems are very useful, their run-time environment imposes a significant amount of performance overhead, which can affect high-performance applications. In these situations, the use of unsafe systems is inevitable. To provide some security for unsafe systems, we use hardening techniques, to thwart attackers. The hardened applications built using unsafe systems can still be compromised. New systems arose that created a middle category. These systems try to provide memory safety guarantees with no or very minimal run-time, thus maintaining performance. Such a system is the Rust programming language, where all checks that guarantee secure memory access are performed at compile time.

In this thesis, we investigate if such systems' safe code, can be exploited after compilation. If the program succeeds in compilation, then the binary is valid and free of memory access bugs. Our goal is to make a valid binary invalid by inserting bugs after the compilation stage. We provide scenarios where binaries' spatial and temporal safety is violated by creating artificial bugs on a binary level. We also provide a light validator script that searches for patterns in the disassembly of the binary. The validator we provide searches only for buffer overflow checks, and it's evaluated for its accuracy. Finally, we propose some ideas, to potentially improve the validator's, accuracy and create a second validator used for locating temporal safety artificial bugs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer programming languages are the key tools used when implementing an application that performs some kind of computation. Throughout the years many types of computer programming systems have been developed. These systems range from compiled time languages to interpreter and run-time dependent languages. As time went by, and these systems became a part of the industry the first programming bugs were found in programs. In fact, compiled languages that offer no run-time support still suffer from programming bugs. Exploiting these bugs can cause major defence holes in systems. The concern about security holes, in programs, sparked the motivation of both the art of exploitation and the need to build up defences for already existing systems like C/C++. It also created a wave of new system programming languages that tried to offer security guarantees, whilst being as lightweight as possible trying to match the speed of C/C++. Such examples are Rust and GO languages.

This thesis will focus on the Rust language [11]. Rust language tries a different approach than both run time-dependent languages and hardening security features of C++ like Resource Acquisition Is Initialization (RAII). In RAII/C++ we can bind the construction process of an object, to resource allocation and the destruction process to resource release, thus for initialization to succeed, we first need resource acquisition to succeed.

Rust uses stricter rules that are enforced by its compiler at compile time. The Rust compiler will evaluate the program and either mark it as valid and compile it or mark it as invalid and fail compilation. In addition, the Rust compiler adds code to provide security guarantees for some types of bugs. This makes Rust's binaries safe and fast. The main concern of this thesis is if these concepts can be bypassed after the compilation of the program. In other words, can these supposedly valid executables, that succeeded compilation phase, be exploited on a binary level? Can the same bugs that exist in a C/C++ binary be created on a Rust binary, after the compilation phase, by stealthy binary modification? Finally, can another program; a "validator", check the Rust binary, understand if a given binary satisfies the same criteria enforced by the Rust compiler, and mark it as valid or

invalid?

Imagine a scenario of an app store like Android's "Google Play" and iOS's "App Store". Android applications include a run-time which places them in the fully safe system category. On the contrary, iOS uses Objective-C [2] which is fully unsafe and Swift [1] which uses Automatic Reference Counting (ARC) to provide memory safety. Consider now that this store uses Rust instead to enforce memory safety for the applications hosted in the store while maintaining high performance. Now, an attacker develops a Rust application that passed the compilation state and is considered valid and safe. The attacker proceeds in creating artificial bugs on the binary level, making the safe application unsafe. This new binary in reality is now invalid and unsafe but is still considered valid and safe and it's added to the app store. [27]

In this thesis, we also explore the possibility of tracking these modifications. Specifically, if the app store from the previous example can create a tool for validating each binary before publishing it, such attacks can be avoided.

Contributions

1. We explore and exploit spatial safety measures. By removing checks for buffer and integer overflows and reintroducing bugs.

2. We explore and exploit temporal safety measures. Create prohibited concepts like double mutable references, use-after-free bugs, dangling pointers etc.

3. We provide a light validator for spatial safety.

Thesis Structure

In the next chapters of this thesis, we cover more on Rust concepts, terminology, and rules along with the methodology we will follow. Specifically, we explain some more terminologies and problems, unsafe systems face (Chapter 2). This chapter can be skipped if the reader has a technical background. In Chapter 3 we discuss different Rust concepts that enforce spatial or temporal safety. Chapters 4 and 5 are the more technical chapters, that include different attempts to bypass Rust's security measures. Additionally, in Chapter 6 we evaluate the results of a light validator for spatial safety provided in this thesis. Finally, in Chapter 7 we propose improvements to the existing validator and suggest ways to evaluate temporal safety violations.

Chapter 2

Background

2.1 Memory Safety, Safe and Unsafe Systems

2.1.1 Memory Safety

Memory safety is the state where our programs are protected from software bugs and vulnerabilities when dealing with memory accesses. Such bugs include buffer overflows, integer overflows, and dangling pointers. In other words, a memory-safe system is a system where memory accesses are well defined. Most of the programming languages we use are memory safe meaning they have some form of memory management, like a garbage collector.

2.1.2 Safe Systems

Safe systems are systems that use a form of memory management. These systems include scripting languages like Python, Ruby, JavaScript, and languages with run-time environments like Java or C#. The languages mentioned above use garbage collection logic, to manage memory accesses, and thus they have a heavy run-time. The main problem with these programming languages is that since they have this run time environment to enforce memory safety, they are slow, making them unsuitable for system programming applications.

2.1.3 Unsafe Systems

Unsafe systems are systems that do nothing to enforce safe memory access. They have no run time environment, to keep track of memory accesses. These systems include C/C++ languages and are usually used for system-level programming (i.e., writing OS kernels, networking protocols etc.). These applications cannot afford to have a run time environment as they mainly focus on performance and offer no memory safety.

2.2 Problems With Unsafe Systems

After defining types of systems and their usage, we need to address different kinds of problems, that occur when we have no memory safety. Let's consider a program in Java where an array is accessed in some way. If the index we use is not in arrays' legal bounds, Java's run-time catches that and creates a run-time error. In a C/C++ equivalent program, we can have arbitrary pointer arithmetic and thus we can index outside of the array's bounds creating either an over-read or overwrite bug [7].

An overread bug is when the potential attacker can access arbitrary amounts of memory and reveal sensitive data, like stack canaries and addresses, that will eventually help the attacker bypass defences like canaries, Address Space Layout Randomization (ASLR) etc. An overwrite bug or buffer overflow, is when the potential attacker has the ability to change memory outside of the legal bounds of each operation. For example, the user can provide an input that eventually will be stored in memory, now consider that the input provided is larger than its buffer container. Since there is no bound checking this causes the copy operation to overwrite and corrupt adjacent memory.

Another important bug that can potentially appear is the use-after-free bug. A use-after-free occurs when we try to dereference a pointer that its memory has been freed. This can cause undefined behaviour and motivate exploitation through techniques like heap spraying and heap Feng sui.

The problems mentioned above are only some of the errors that occur and belong to the subset of access errors, which are needed to understand this thesis. An error that occurs and belongs to the subset of memory leak errors is the double-free bug. In a double-free bug, memory is freed twice, causing premature free of a new object in memory.

In the next chapters, we examine how memory safety techniques in Rust try to solve the two categories of violations. These categories are Spatial Safety (ensuring memory accesses are within bounds of the object being accessed) and Temporal Safety (making sure that pointers point to valid memory when they are being accessed).

2.3 Basic Motivation Of Rust

Rust was ranked as the most loved language by programmers in StackOverflow [25]. It is a statically typed systems programming language, that aims to achieve high performance in applications that need similar speed to what unsafe systems offer while maintaining memory safety. Rust removes the run time support and tries to provide memory safety at compile time. Thus, Rust needs to enforce some rules about how its code is written and structured. Rust language also enforces some concepts that are not present in other widely used languages like ownership, borrowing and lifetimes, which will be explained

in the next chapter. Most of these concepts are enforced by the compiler at compile time. Specifically, spatial safety bugs are prevented by snippets of code added by the compiler and temporal safety bugs are prevented by a subroutine of Rust's compiler called the borrow checker. Rust as a system sits between the fully unsafe systems like C/C++ and the run-time safe environments like Java/C#.

When writing in Rust, we can have unsafe Rust code and safe Rust code bundled together. In this document, we explore only the safe Rust code. To distinguish the two; the Safe Rust code applies restrictions to the programmer enforced by the concepts we will talk about later, and the unsafe Rust provides more autonomy to the programmer. Since unsafe Rust can co-exist with C/C++ code, it operates like code written in C thus, the same possible errors can occur as in an unsafe system.

2.4 Advantages and Disadvantages When Using Rust

2.4.1 Advantages

Some advantages of Rust include memory and thread-safety. Since there is no run-time support speed and performance are not affected. Rust has an easy high-level syntax, that supports most of the concepts used in other high-level languages. Lastly, Rust provides friendly documentation and compiler error messages along with a very good package manager [12].

2.4.2 Disadvantages

The main disadvantage of Rust is the rules that the compiler enforces on the programmer, to ensure safety. Rules like lifetimes, ownership and borrowing, are not present in other languages, which creates a larger learning curve for the developer.

Chapter 3

Methodology

This chapter dives into the various ideas of Rust's memory safety, such as bound checking, the borrow checker and lifetimes. These techniques are used to prevent bugs that belong to memory safety violations like temporal and spatial safety. In other languages that are run-time environment-dependent. The run time environment executes checks that enforce spatial and temporal safety. These languages use bound checking, or metadata bookkeeping, like counters that count references for a piece of data to prevent dangling pointers. Rust tries to enforce these safety qualities at compile-time, reducing run time overhead as much as possible. Our goal in this thesis and later chapters is to see if we can modify the "safe" binary produced after the compilation and reintroduce spatial and temporal safety violations.

3.1 Spatial Safety

As it was briefly explained in the previous chapter, spatial safety is when the pointer used to access an object can access data in memory, that is outside of the bounds of its allocated space. Usually, when we talk about enforcing spatial safety, in languages like C# or Java, we mean executing run-time checks before dereferencing the pointer (e.g., comparison of the index with the bounds of allocated memory).

Rust's approach to these checks is similar. When exploring buffer overflow and integer overflow bugs, the Rust compiler adds code to perform some safety checks at run-time. Buffer overflow checks include snippets of added code that perform bounds checking when accessing an array. Rust compiler adds a check that compares the index to the array's length, and either allows access or crashes the program. Using this kind of checks Rust can prevent overwrite or overread bugs [7]. The added check causes the program to crash before corrupting/overwriting memory or before over-reading memory that does not belong to the accessed object.

When compiling a binary using rustc we can either build the application in debug or release mode. In debug mode, we have debug symbols and is suitable for development. Instead in release mode, we have no debug symbols, and the binary is optimized, which is more suitable for released applications. In both cases, the programmer can choose different optimization levels. The format of each added check depends on the chosen optimization level.

For integer overflows, the Rust compiler adds a similar check. In release mode the check is present, only if the programmer opts for it. When Rust applications run in debug mode, this check is always enabled. With integer overflow bugs there is an extra instruction added by the Rust compiler, which checks a register for overflow, and crashes the program if the register has indeed overflowed.

3.2 Temporal Safety

When talking about how Rust enforces temporal safety, we need to address more than just run-time checks. The Rust compiler uses no run-time checks for temporal safety. To get a better understanding, we need to talk about the different concepts of Rust. More specifically in this section, we go through variables and mutability, and some other functionalities of the borrow checker like ownership and borrowing concepts and finally explain what the lifetime concept is. These concepts are checked for validity by the compiler and if any of these are violated, the compiler will mark our program as invalid and won't compile it.

3.2.1 Variables and Mutability

Some of the easier to explain quirks of Rust is that all variables are immutable by default. We can still have mutable variables by manually declaring them as mutable using the "mut" keyword. In this case, if we try to change the value of a variable that is not declared as mutable the compiler won't compile our program and will produce an error. Variables declared as constants are also immutable and can be used in the global scope too.

Copying Values

In Rust when we move primitive values from one variable to the other, we create a copy of the variable in memory. Primitive values are copied over to a different location in memory.

```
1 let x = 10;
2 let y = x;
```

Listing 3.1: Copying Values Example

In the above example, x and y variables point to different locations in memory that both contain the value 10. Since 10 is an integer and a primitive type, its value is copied to another location.

3.2.2 Ownership

Ownership is maybe the most unique feature of Rust. Using ownership Rust guarantees memory safety without needing a garbage collector. Ownership consists of a set of rules that dictate how a program manages memory. In the previous chapter, we talked about other systems' approaches to memory safety. Some use run-time support and some other languages like C/C++ let the programmer manage memory, by explicit allocation and deallocation. Rust's approach to memory management is automatic allocation and deallocation by using the compiler to enforce ownership rules. Again, if any rule is violated the program won't compile. Before explaining the rules, we need to define what a "scope" is. A scope in Rust is specified with curly brackets " { } " and can be either a function or a pair of curly brackets inside the function that marks a smaller scope. Scope marks how long and where a variable is available.

Ownership Rules

1. Each variable is called the owner of its value.
2. There is only one owner for each value at a time.
3. When the owner of the value goes out of scope the value is dropped/freed.

```
1 {//scope
2         let s = String::from("hello world");
3         /*s is now valid, s is the owner of the value "hello world" */
4
5         //use s in different operations
6
7 }  /*s is dropped because the scope ends here
8     we cannot use s after the end of scope*/
```

Listing 3.2: Ownership Example

Moving Values

```
1 {//scope
2 let s1 = String::from("hello");
3 let s2 = s1;
4 println!("{}" s1);
5 }
```

Listing 3.3: Moving Values Example

With non-primitive values, Rust does not create a copy. In other words, Rust won't copy the value "hello" in another location and make s2 point to that location. Instead, memory representation should look like Figure 3.1.

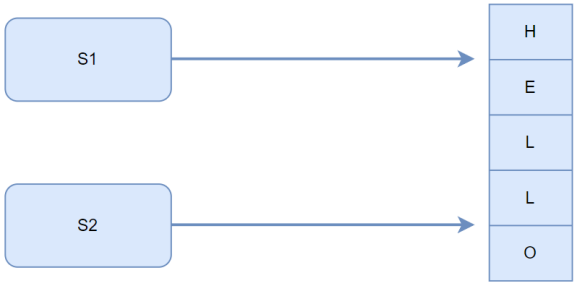

Figure 3.1: Moving Values Memory Representation Before Moving

Since now the same value has two different owners at the same time, Rust won't compile the program. Rust avoids this by moving values. Meaning, we can no longer use variable s1 to access the "hello" string. The reason is that the value of s1 was moved to s2. S1 is no longer the owner of value "hello", instead s2 is the new owner.Actual memory representation is shown in Figure 3.2.

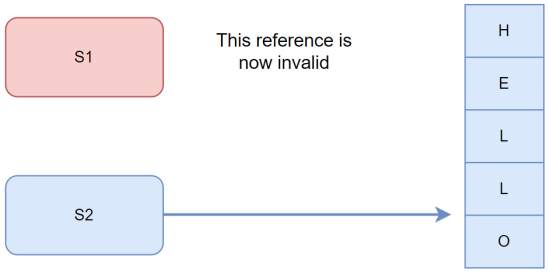

Figure 3.2: Moving Values Memory Representation After Moving

In the last two subsections, we explored moving and copying variables to other variables. Similarly, the same moving and copying principles exist when passing a variable as an argument to a function. Thus, on a primitive type, the function gets a copy of the value and on a non-primitive type, the function takes ownership of the value and causes a move operation. Using the same logic when returning a value from a function the function transfers the ownership of the value to the destination variable.

```
{ //scope
let s1 = String::from("hello world");  // create a string
takes_ownership(s1);  //the function takes ownership of s1's value,
// s1 is dropped when function scope ends, its now unusable in this scope

let s2=gives_ownership(); //s2 is the destination variable and
     // takes ownership of the return value from the function
}
```

Listing 3.4: Ownership Using Functions

3.2.3 Borrowing

Sometimes though we want to give access to a piece of data without moving the piece of data out of scope or giving up the ownership. In the same way, as in C/C++, we can create a reference to a function, or another variable, using the symbol "&", we can also create a reference in Rust. In Rust when creating a reference, to a variable that variable does not own that value. Thus, when the scope of the variable ends, the value won't be dropped. Borrowing works the same way as in life; we borrow something to use and then we must give it back. The action of creating a reference is called borrowing and to use borrowing we need to follow some rules. The way borrowing works is similar to the readers-writers problem in concurrency.

References in Rust language similarly to its variables are immutable by default. Meaning when creating a regular reference to a value, the new reference can read but not modify that value in memory. Mutable references can be created by explicit declaration using "&mut".

Borrowing Rules

1. We can have as many immutable references as we want at the same time.

2. We can only have one mutable reference at a time for the same value in one scope. Code with two mutable references on the same value in the same scope fails compilation.

```
{ // scope
    let mut s = String::from("hello world");
    let s1 = &mut s;  //1st mutable reference in scope
    let s2 = &mut s;  //2nd mutable reference in same scope
} //this code will not compile
```

Listing 3.5: Example of Borrowing Rule 2

3. A mutable reference cannot coexist with an immutable one. Code with immutable and mutable references to the same value fails compilation.

```
1 { // scope
2     let mut s = String::from("hello world");
3     let s1 = &mut s;  //1st mutable reference in scope
4     let s2 =  &s;  //2nd immutable reference in same scope
5 } //this code will not compile
```

Listing 3.6: Example of Borrowing Rule 3

3.2.4 Lifetimes

The last Rust feature to cover in this chapter is lifetimes. Lifetimes are a feature that prevents the creation of dangling pointers. Using lifetimes Rust makes sure that during compilation no reference can be a dangling reference. In other words, the compiler makes sure that if a reference to a data exists, that piece of data won't go out of scope before all the references to that data go out of scope.

Before we continue, we need to clarify what lifetimes are. Lifetimes are a difficult operation of Rust's compiler and something we are not familiar with from other languages. We try to explain lifetimes as simply as possible.

Lifetimes are used when giving references to a function, or when a struct contains references, to objects that the struct does not own. Lifetimes help the compiler understand how long it can hold on to a reference. When talking about lifetimes we have a couple of rules, called the Elision rules. We may not understand it but lifetimes are always used and implicitly declared, by the compiler and checked by the borrow checker. Sometimes though, we may need to explicitly declare them in functions or structs, to assure the compiler that given references are going to live a certain amount of time. The lifetime of a reference lasts for as long as a value lives, which is until either the value is moved or dropped. Before listing some code, we need to explain the elision rules we mentioned earlier.

Elision Rules

1. Each elided lifetime becomes a distinct lifetime parameter.
2. If there is only one input lifetime that lifetime is assigned to all output lifetimes.
3. For multiple input lifetimes, if there is one lifetime of "&self" or "&mut self". That lifetime is assigned to the output references/variables.

From the above rules, we can extract the following

1. Lifetimes are stubs issued by the compiler, on the code. The borrow checker runs on modified code and marks our code as valid or not.

2. Each input reference is assigned a distinct lifetime automatically, and based on inputs, the output is assigned its lifetime.

3. Lifetimes help Rust enforce its ownership model and aim to prevent dangling references.

In the below listing we present an example of implicit lifetime declaration and why sometimes, we need to declare our own lifetimes.

```
fn foo(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

In function foo, we do not declare lifetimes. Thus, the compiler tries to apply the elision rules to enforce lifetimes automatically and produces something like the following listing.

```
fn foo<'a,'b>(x: &'a str, y: &'b str) -> &'? str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Since x and y are assigned a different lifetime, the compiler does not know what kind of lifetime the output should get. To solve this, we need to assure the compiler that both x and y variables are going to live for at least the same lifetime. Following listing calms the compiler down.

```rust
fn foo<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

3.3 Our Methodology

In this section, we describe the methodology followed in exploiting Rust's concepts explained in previous sections. In spatial safety, we create and explore examples that use statically allocated arrays and find the check injected in our code by the Rust compiler. We create integer overflow examples and similarly locate the check. After locating the check, we try to bypass it and recreate the equivalent bugs as in C/C++. In temporal safety, we create double mutable references on a binary level and exploit concepts like lifetimes and the borrow checker. The approach followed on both occasions spatial and temporal safety is the sneakiest and dirtiest one since we want to prove that by making slight modifications, memory safety problems can reappear.

3.3.1 Spatial Safety

Specifically, in spatial safety, we create buffer overflow proofs of concept and modify the binaries using a disassembler like radare2 to create some kind of bug. In the first proof of concept, we try to locate the buffer overflow check inside the disassembly and bypass it. In the next proof of concept, we produce a shellcode, that utilizes that buffer overflow bug to overwrite the return address and jump to an already existing function in the program. Proof of concept 3 has a different goal, which is to investigate if the check exists when changing a single element inside the array, for which the index is provided by user input. Finally, in proof of concept 4, integer overflows are investigated. Similarly, we need to locate and modify the checks using radare2 to bypass them. In all the POCs mentioned above, we use the following process.

1. Create the toy example based on each POC.
2. Compile, run and observe initial results.
3. Explore low-level instructions and disassembly, of the binary using GDB and Radare2.
4. Locate the instructions that perform bound checking.

5. Modify the binary using Radare2.

6. Run the modified binary with the right input to exploit the bug.

3.3.2 Temporal Safety

In temporal safety, the tasks that need to be conducted are not as straightforward. The main goal of the temporal safety investigation is to render the borrow checker obsolete. We start the investigation by trying to create a double mutable reference of the same value, for a simple type, like an integer wrapper and then move to more complex objects like Vectors, and custom-made structures. After creating a double mutable reference, we advance by creating a use after free bug and a dangling pointer. Lastly, in proof of concept 5, the lifetimes concept is exploited, by creating a dangling pointer. All these concepts are exploited, in a dirty and stealthy way. We use existing padding code to write our assembly code inside. Then we find a command that can be used to jump to padding code without shifting the binary, execute the injected commands and jump back to the next regular instruction. A general approach we used in the more involving POCs is the following.

1. Create the toy example based on each POC.

2. Compile, run and observe initial results.

3. Explore low-level instructions and disassembly, of the binary using GDB and Radare2.

4. Locate in memory the addresses of the pointers.

5. Locate padding code where we add new assembly instructions.

6. Locate instruction that can be used to jump to padding code without shifting the binary.

7. Modify the binary using Radare2.

   (a) Replace instruction with a jump to padding section.

   (b) Add new code in padding section.

8. Run the modified binary with the right input to exploit the bug.

Chapter 4

Spatial Safety

4.1 Proof Of Concept 1: Locate And Create First Buffer Overflow Bug

4.1.1 Exploring Assembly Of a Simple Program

To explore checks enforcing buffer overflow prevention, we create a simple program, that copies elements from one array to the other. To do that we use a small function called "copy over", as seen below.

```
fn copy_over(mut arr_a: [i32; 10], arr_b: [i32; 15]) -> [i32; 10] {
    let len = arr_b.len();
    for i in 0..len {
        arr_a[i] = arr_b[i];
    }
    return arr_a;
}
```

This function contains a for loop to copy from one array to the other, which is a classic way to introduce buffer overflows in C/C++. The function copies 15 integer numbers, from *array b* to *array a*. But unfortunately, *array a* is only 10 integers long. This is a classic buffer overflow error and the program compiles and runs.

Fortunately, the program crashes with the following error: **" thread 'main' panicked 'index out of bounds: the len is 10 but the index is 10' "**

From the above experiment, we are sure that Rust, has a way to produce a panic error, and prevent overwrite bugs. Let's take a closer look into assembly code and try to spot if any code was added for this kind of check. We are using gdb [10] with dashboard add-on [8], to help with the inspection.

```
1  Dump of assembler code for function _ZN4toy19copy_over17h2218b9e201ef9b60E:
2     0x000055555555c060 <+0>:     sub    $0x98,%rsp
3     0x000055555555c067 <+7>:     mov    %rdx,0x28(%rsp)
4     ===Skip Some Instructions===
5     0x000055555555c12e <+206>:   mov    %rax,0x88(%rsp)
6     0x000055555555c136 <+214>:   mov    %rax,0x90(%rsp)
7     0x000055555555c13e <+222>:   cmp    $0xf,%rax
8     0x000055555555c142 <+226>:   setb   %al
9     0x000055555555c145 <+229>:   test   $0x1,%al
10    0x000055555555c147 <+231>:   jne    0x55555555c14b
11    0x000055555555c149 <+233>:   jmp    0x55555555c17f
12    0x000055555555c14b <+235>:   mov    0x8(%rsp),%rax
13    0x000055555555c150 <+240>:   mov    0x28(%rsp),%rcx
14    0x000055555555c155 <+245>:   mov    (%rcx,%rax,4),%ecx
15    0x000055555555c158 <+248>:   mov    %ecx,0x4(%rsp)
16    0x000055555555c15c <+252>:   cmp    $0xa,%rax
17    0x000055555555c160 <+256>:   setb   %al
18    0x000055555555c163 <+259>:   test   $0x1,%al
19    0x000055555555c165 <+261>:   jne    0x55555555c169
20    0x000055555555c167 <+263>:   jmp    0x55555555c19b
21    0x000055555555c169 <+265>:   mov    0x20(%rsp),%rax
22    0x000055555555c16e <+270>:   mov    0x8(%rsp),%rcx
23    0x000055555555c173 <+275>:   mov    0x4(%rsp),%edx
24    0x000055555555c177 <+279>:   mov    %edx,(%rax,%rcx,4)
25    0x000055555555c17a <+282>:   jmp    0x55555555c0d7
26    0x000055555555c17f <+287>:   mov    0x8(%rsp),%rdi
27    0x000055555555c184 <+292>:   lea    0x3a3cd(%rip),%rdx        # 0x555555596558
28    0x000055555555c18b <+299>:   lea    -0x14a2(%rip),%rax        # call panic
29    0x000055555555c192 <+306>:   mov    $0xf,%esi
30    0x000055555555c197 <+311>:   call   *%rax
31    0x000055555555c199 <+313>:   ud2
32    0x000055555555c19b <+315>:   mov    0x8(%rsp),%rdi
33    0x000055555555c1a0 <+320>:   lea    0x3a3c9(%rip),%rdx        # 0x555555596570
34    0x000055555555c1a7 <+327>:   lea    -0x14be(%rip),%rax        # call panic
35    0x000055555555c1ae <+334>:   mov    $0xa,%esi
36    0x000055555555c1b3 <+339>:   call   *%rax
37    0x000055555555c1b5 <+341>:   ud2
38 End of assembler dump.
```

Listing 4.1: Initial Disassembly

As shown in the above listing the triplet marked with red colour is the one we consider to be the bounds check added by the compiler. The jump commands highlighted in blue lead to the corresponding panic section of the binary that causes the crash. The command triplet seen consists of a compare, a set and test commands, and jumps to panic block if the %rax register is larger than the fixed magic value in the CMP command. More specifically, the program checks current index with magic value (bounds) and sets %al based on CF flag. Jne command jumps to the specified address if ZF flag is not set (ZF == 0). If (%al AND 0x1) = 0 then the ZF flag is set else ZF = 0. As long as %al equals to 1 then the program avoids the panic error block. This becomes clearer by observing cmp commands in these sections, the magic value for each cmp is 0xf and 0xa, which equals

to 15 and 10; the length of each array respectively. To investigate this more we create a smaller example with only one array, and a loop to initialize that array. The simplified copy_over() function is seen below.

```
fn copy_over(mut arr_a: [i32; 10], range: i32) -> [i32; 10] {
    for i in 0..range {
        println!("{}", i);
        arr_a[i as usize] = i as i32;
    }
    return arr_a;
}
```

We must investigate the disassembly of the simplified example. Once again, the same check is present, with the same triplet of commands. Shown in red color.

```
   0x000055555555e817 <+279>:   lea    0x68(%rsp),%rdi
   0x000055555555e81c <+284>:   call   *0x421ae(%rip)        # 0x5555555a09d0
   0x000055555555e822 <+290>:   mov    0x64(%rsp),%eax
   0x000055555555e826 <+294>:   mov    %eax,0xc(%rsp)
   0x000055555555e82a <+298>:   movslq 0x64(%rsp),%rax
   0x000055555555e82f <+303>:   mov    %rax,0x10(%rsp)
   0x000055555555e834 <+308>:   cmp    $0xa,%rax
   0x000055555555e838 <+312>:   setb   %al
   0x000055555555e83b <+315>:   test   $0x1,%al
   0x000055555555e83d <+317>:   jne    0x55555555e841
   0x000055555555e83f <+319>:   jmp    0x55555555e857
   0x000055555555e841 <+321>:   mov    0x28(%rsp),%rax
   0x000055555555e846 <+326>:   mov    0x10(%rsp),%rcx
   0x000055555555e84b <+331>:   mov    0xc(%rsp),%edx
   0x000055555555e84f <+335>:   mov    %edx,(%rax,%rcx,4)
   0x000055555555e852 <+338>:   jmp    0x55555555e74e
   0x000055555555e857 <+343>:   mov    0x10(%rsp),%rdi
   0x000055555555e85c <+348>:   lea    0x3fba5(%rip),%rdx        # 0x55555559e408
   0x000055555555e863 <+355>:   lea    -0x2b6a(%rip),%rax        # call panic function
   0x000055555555e86a <+362>:   mov    $0xa,%esi
   0x000055555555e86f <+367>:   call   *%rax
   0x000055555555e871 <+369>:   ud2
End of assembler dump.
```

Listing 4.2: Exploring Simpler Disassembly Example

4.1.2 Binary Modification

The goal is to make the check unusable by patching the binary. We use the following code.

```rust
fn main() {
    let mut arr_a: [i32; 10] = [10; 10];
    let overwrite: [i32; 10] = [0; 10];
    let arr_b: [i32; 15] = [15; 15];
    println!("overwritten array before : {:?}", overwrite);
    copy_over(&mut arr_a, &arr_b);
    println!("array A: {:?}", arr_a);
    println!("array B: {:?}", arr_b);
    println!("overwritten array after: {:?}", overwrite);
}

fn copy_over(arr_a: &mut [i32; 10], arr_b: &[i32; 15]) {
    let len = arr_b.len();

    for i in 0..len {
        arr_a[i] = arr_b[i];
    }
}
```

Copy over function is the same as seen before, but now we have three arrays. One array is the destination array (arr_a), and the other is the source array (arr_b). The code uses one more array with the name "overwrite" allocated between the other two. The main goal of this toy example is to create an overwrite bug, by modifying the binary. The results we expect from this example are to finish the loop without panic exceptions and write all 15 integers of arr_b to arr_a but since arr_a, can hold up to 10 integers, the next 5 elements are written in overwrite array. Since we are talking about spatial safety and stack structure, we expect the following arrangement of the arrays.

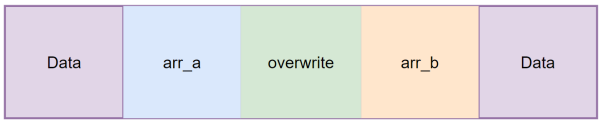

Figure 4.1: Array Arrangement In Stack

To modify the binary, we use radare2 disassembler. We locate the check inside the disassembly of the copy_over function and then we patch the cmp command. Specifically, we change the compare instruction for the destination array (arr_a), from cmp 0xa, %rax to cmp 0xf, %rax (Listing 4.1). By doing this we know that the loop will finish copying all 15 integers of the source array.

4.1.3 Results After Patching

```
1 overwritten array before : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
2 array A: [15, 15, 15, 15, 15, 15, 15, 15, 15, 15]
3 array B: [15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15]
4 overwritten array after: [15, 15, 15, 15, 15, 0, 0, 0, 0, 0]
```

Listing 4.3: Final Results With Overwrite Array

From the above results, we confirm our speculations. The destination array was overflowed, and the remaining elements were written in overwrite array. We can now remove the overwrite array and perform the same process as explained above with the same program, the expected results are that arr_b will overwrite itself.

```
1 array A: [10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
2 array B: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
3 array A: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
4 array B: [10, 11, 12, 13, 14, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Listing 4.4: Final Results Without Overwrite Array

Before we explore another example, we need to note that we can modify the command instruction iff the bytes we are going to add won't shift the binary. If modifying the cmp instruction shifts the binary, we cannot perform this modification.

4.2 Proof of Concept 2: Use buffer Overflow Bug To Transfer Control Flow To Another Function

In this section, we create a buffer overflow like the one mentioned in the previous section, but now we exploit that buffer overflow bug, to modify the return address. The goal is to change the return address and make the program return to a different destination than the one it was supposed to return to.

4.2.1 Creating Toy Example

We create a similar toy example as before, but it needs to follow some specifications. Firstly, it needs to accept input from the user. Additionally, we need to call the function that serves as the malicious destination at least once, as the Rust compiler performs dead code elimination by default. The code for the initial toy example is presented in the listing below. The program takes a list of integers from the user as input arguments. Copies the input to another array, using the copy method called copy_arr(). The function show_message() serves as the malicious destination function. We want to use a shellcode to transfer the program's control flow to that function.

```rust
fn main() {
    show_message();
    let args: Vec<String> = env::args().collect();
    println!("{:?}", args);

    if args.len() == 1 {
        return;
    }

    let mut input_vec: Vec<u8> = Vec::new();
    for arg in env::args().skip(1) {
        // println!("{}", arg);
        input_vec.push(arg.parse::<u8>().unwrap());
    }
    println!("{:?}", input_vec);
    bug(input_vec);
}

fn bug(input_vect: Vec<u8>) {
    let marker: u128 = 0xdeadbeef;
    let mut arr_a: [u8; 256] = [0; 256];
    println!("input: {:?}", input_vect);
    copy_arr(&mut arr_a, input_vect);
    println!("array A: {:?}", arr_a);
} //we want this function to return to show_message() function

fn copy_arr(arr_a: &mut [u8; 256], input_vect: Vec<u8>) {
    let len = input_vect.len();
    for i in 0..len {
        println!("temp: {:?}, i= {}", input_vect.get(i), i);
        arr_a[i] = input_vect.get(i).unwrap().to_be_bytes()[0];
    }
}
fn show_message() {
    println!("I was not supposed to be called but here we go!!!");
}
```

4.2.2 Performing The Attack

Usually, we can use the printf built-in function in Linux systems to pass hexadecimal values as input. Instead, we are going to make a small python script that reads a file and transform it to the input format we want and use it as a shellcode. The input format for the script is the same as the output when using the x/w command in gdb [10] without the

address column.

As a naïve approach, we use the safe environment of gdb [10], and for now, we consider the whole stack until the return address as our shellcode. Later in this section, we discover the minimum values that are needed in the shellcode to succeed.

4.2.3 Steps Used To Exploit Toy Example

1. Run the program in gdb [10].
2. Insert a breakpoint in Bug() function.
3. Get the Return address of that function.
4. Copy stack image of that function until return address (use info frame to see RA in GDB [10]).
5. Use the script and stack image to generate a valid input, in terms of length (We explain why this step is important later).
6. Rerun the program using the input to get the valid stack image. Find the address of the function we want to jump to. We do that by using the print command in gdb [10] and the name of the function. This works since we have debug symbols available.
7. Generate the new input, with the return address of the function we want to jump to using the script. The script provides the length of the input, in terms of bytes. This number is the minimum number that the cmp instruction we talked about in the previous section needs as bounds.
8. Modify the binary, as explained, using the hex value for the length of the input, provided by the script.
9. Run the program, using the new input.

Since we are now aware of the basic procedure, we explain the procedure in more detail. In the above process, we mentioned that we run the program once, and then we must run it again to get a valid image stack.

Using the whole stack as shellcode is not always the best idea. We also need to address what are the types of bugs needed, to perform this exploitation process.

To answer these questions, we perform the attack based on the steps above, but when we have the valid stack image, we start zeroing elements from the stack until the program crashes. This process aims to leave only the necessary elements in our shellcode, that without these values, the exploitation won't work.

```
1  0x7fffffffc2b0: 0x00000000 0x00000000 0x00000000 0x00000000
2  0x7fffffffc2c0: 0x00000000 0x00000000 0x00000000 0x00000000
3  0x7fffffffc2d0: 0x00000000 0x00000000 0x00000000 0x00000000
4  0x7fffffffc2e0: 0x00000000 0x00000000 0x00000000 0x00000000
5  0x7fffffffc2f0: 0x00000000 0x00000000 0x00000000 0x00000000
6  0x7fffffffc300: 0x00000000 0x00000000 0x00000000 0x00000000
7  0x7fffffffc310: 0x00000000 0x00000000 0x00000000 0x00000000
8  0x7fffffffc320: 0x00000000 0x00000000 0x00000000 0x00000000
9  0x7fffffffc330: 0x00000000 0x00000000 0x00000000 0x00000000
10 0x7fffffffc340: 0x00000000 0x00000000 0x00000000 0x00000000
11 0x7fffffffc350: 0x00000000 0x00000000 0x00000000 0x00000000
12 0x7fffffffc360: 0x00000000 0x00000000 0x00000000 0x00000000
13 0x7fffffffc370: 0x00000000 0x00000000 0x00000000 0x00000000
14 0x7fffffffc380: 0x00000000 0x00000000 0x00000000 0x00000000
15 0x7fffffffc390: 0x00000000 0x00000000 0x00000000 0x00000000
16 0x7fffffffc3a0: 0x00000000 0x00000000 0x00000000 0x00000000
17 0x7fffffffc3b0: 0x55591000 0x00005555 0x555a1390 0x00005555
18 0x7fffffffc3c0: 0x00000002 0x00000000 0x00000000 0x00000000
19 0x7fffffffc3d0: 0x5555e490 0x00005555 0xffffc3e8 0x00007fff
20 0x7fffffffc3e0: 0x00000001 0x00000000 0xffffc6f0 0x00007fff
21 0x7fffffffc3f0: 0x5555e490 0x00005555 0xffffc6f0 0x00007fff
22 0x7fffffffc400: 0x555b34b0 0x00005555 0x00000200 0x00000000
23 0x7fffffffc410: 0x000001f0 0x00000000 0x00000000 0x00000000
24 0x7fffffffc420: 0x555a8968 0x00005555 0x555a4088 0x00005555
25 0x7fffffffc430: 0x555a1358 0x00005555 0x00000002 0x00000000
26 0x7fffffffc440: 0x00000000 0x00000000 0x5555e490 0x00005555
27 0x7fffffffc450: 0xffffc6d8 0x00007fff 0x00000001 0x00000000
28 0x7fffffffc460: 0x55593f07 0x00005555 0xdeadbeef 0x00000000
29 0x7fffffffc470: 0x00000000 0x00000000 0xffffc6f0 0x00007fff
30 0x7fffffffc480: 0x555a12f0 0x00005555 0xffffc8d8 0x00007fff
31 0x7fffffffc490: 0x55591000 0x00005555 0x5555fa24 0x00005555
```

Listing 4.5: Initial Stack Image

By inspecting memory, we look at the stack image of the function bug(). In the figure above, we have marked some points of interest. Starting from the easy-to-understand addresses, we have the return address at the bottom right corner marked with red and the 0xdeadbeef marker variable seen in green.

Through trial and error, we start removing the rest of the addresses and zero them. As it turns out everything can be zeroed except the contents that are highlighted with blue colour. But why are these values important?

By examining the input vector variable, we can see that the pointer to that vector is the same address as the one stored in the stack (Listing 4.6). The hexadecimal value 1f0 represents the length of the vector. By converting it to decimal we find out that it is equal to 496. Thus, our shellcode needs to include the return address of the malicious function, the address of the pointer holding the input and the length of the input. This explains why we needed to run the program twice. The first run was providing the input length implicitly. Generally, we can easily calculate this value.

```
>>> p input_vect
$1 = alloc::vec::Vec<u8, alloc::alloc::Global> {
  buf: alloc::raw_vec::RawVec<u8, alloc::alloc::Global> {
    ptr: core::ptr::unique::Unique<u8> {
      pointer: 0x5555555b34b0,
      _marker: core::marker::PhantomData<u8>
    },
    cap: 512,
    alloc: alloc::alloc::Global
  },
  len: 496
}
```

Listing 4.6: Exploring Blue Values Using GDB [10]

Thus, in the following listing, we provide an example of the minimum content we need to have in our shellcode. Everything else can be zeroed.

Concluding, along with the overwrite bug from the buffer overflow we created, we also need an overread bug to get the metadata for the variable that holds the input. In this case, we need the address of the pointer. The input length can be calculated easily and added to the shellcode.

4.2.4 Minimal Shellcode Example

```
0x00000000  0x00000000  0x00000000  0x00000000
==========  SKIP SOME ZEROED VALUES ==========
0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
0x00000002  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
0x555b34b0  0x00005555  0x00000000  0x00000000
0x000001f0  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x5555fa24  0x00005555
```

Listing 4.7: Minimum Number Of Values We Need To Have In Shellcode. Final Stack Image

After returning from the bug function, the main program returns to the show_message() function. Inspecting the following snippet of gdb's [10] log, we can see the code of the function and it's not main. Program crashes when executing line 22 which corresponds to the print function.

```
1  +ni
2  toy2d::show_message () at src/main.rs:47
3  47      }
4  +disas
5  Dump of assembler code for function _ZN5toy2d12show_message17heae569a4c5c67091E:
6  => 0x0000555555560830 <+0>:     sub    $0x38,%rsp
7     0x0000555555560834 <+4>:     lea    0x8(%rsp),%rdi
8     0x0000555555560839 <+9>:     lea    0x40c60(%rip),%rsi        # 0x5555555a14a0
9     0x0000555555560840 <+16>:    mov    $0x1,%edx
10    0x0000555555560845 <+21>:    lea    0x31a5c(%rip),%rcx        # 0x5555555922a8
11    0x000055555556084c <+28>:    xor    %eax,%eax
12    0x000055555556084e <+30>:    mov    %eax,%r8d
13    0x0000555555560851 <+33>:    call   0x5555555622a0
14    0x0000555555560856 <+38>:    lea    0x8(%rsp),%rdi
15    0x000055555556085b <+43>:    call   *0x4315f(%rip)        # 0x5555555a39c0
16    0x0000555555560861 <+49>:    add    $0x38,%rsp
17    0x0000555555560865 <+53>:    ret
18 End of assembler dump.
```

Listing 4.8: Final Results

4.3 Proof Of Concept 3: Overwrite Bug When Accessing An Array

In this proof of concept, we create a new toy example, to explore and answer some important questions. Firstly, if we have a buffer, and we want to access it, does it contain similar bounds check. The second question we need to answer is; if there is such a check can we modify the binary, the same way we did before, and then create an overwrite or overread bug?

4.3.1 Creating Toy Example

```
1      fn bug() {
2          let marker: u128 = 0xdeadbeef;
3          let mut arr_a: [u8; 256] = [10; 256];
4          change_elem(&mut arr_a);
5          println!("array A: {:?}", arr_a);
6      }
7
8      /*change element on a specified index given by user.*/
```

```
9       fn change_elem(arr_a: &mut [u8; 256]) {
10          let mut input = String::new();
11          println!("give index of element to access:");
12
13          io::stdin()
14              .read_line(&mut input)
15              .expect("error: with user input");
16
17          println!("input: {:?}", input);
18          let index = input.trim().parse::<usize>().unwrap();
19
20          println!("index: {}", index);
21          arr_a[index as usize] = 65;
22      }
```

The example we created uses the change_elem() function to get user input, index the given array, and write the number 65 to that slot. In this example, the user controls where the modification will take place. No programmer enforced bounds checking is performed for the index. Thus, this is an overwrite bug that an attacker could easily exploit, in C/C++. As we have seen previously in Rust there is bound checking, for arrays.

4.3.2 Locating The Check

When running the toy example, the program crashes if the input is larger than 255. This is normal as the buffer given can hold only up to 256 elements. The program panics and we get the following error which creates the suspicion that a similar check is present when the program runs.

```
1  index for accessing element:
2  257
3  input: "257\n"
4  index: 257
5  thread 'main' panicked at 'index out of bounds: the len is 256 but the index is 257',
       src/main.rs:40:5
6  note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Listing 4.9: Initial Results When Running The Program

By taking a closer look at the assembly we detect the same check as before. We know this is a valid check as the decimal value of 0x100 equals 256 which is the length of our buffer.

```
1    0x000055555555dab0 <+640>:   call   0x55555555d380
2    0x000055555555dab5 <+645>:   jmp    0x55555555dab7
3    0x000055555555dab7 <+647>:   lea    0xee22(%rip),%rcx        # 0x55555556c8e0
4    0x000055555555dabe <+654>:   lea    0x148(%rsp),%rdi
5    0x000055555555dac6 <+662>:   call   *%rcx
6    0x000055555555dac8 <+664>:   jmp    0x55555555daca
7    0x000055555555daca <+666>:   mov    0x128(%rsp),%rax
8    0x000055555555dad2 <+674>:   mov    %rax,0x8(%rsp)
9    0x000055555555dad7 <+679>:   cmp    $0x100,%rax
10   0x000055555555dadd <+685>:   setb   %al
11   0x000055555555dae0 <+688>:   test   $0x1,%al
12   0x000055555555dae2 <+690>:   jne    0x55555555dae6
13   0x000055555555dae4 <+692>:   jmp    0x55555555db35
```

Listing 4.10: Inspecting Memory and Locating Check

Thus, the check shown in the above listing is the one that performs the bounds checking. The next step is to patch the binary and bypass this check.

4.3.3 Performing The Attack

We use radare2 to patch the binary, the assembly code for the check after modification is shown in the below listing. We decided to change the check to be equal to 265 and thus, we expect to see the value 0x109 in the cmp command's magic value.

```
1    0x000055555555d73e <+654>:   lea    0x148(%rsp),%rdi
2    0x000055555555d746 <+662>:   call   *%rcx
3    0x000055555555d748 <+664>:   jmp    0x55555555d74a
4    0x000055555555d74a <+666>:   mov    0x128(%rsp),%rax
5    0x000055555555d752 <+674>:   mov    %rax,0x8(%rsp)
6    0x000055555555d757 <+679>:   cmp    $0x109,%rax
7    0x000055555555d75d <+685>:   setb   %al
8    0x000055555555d760 <+688>:   test   $0x1,%al
9    0x000055555555d762 <+690>:   jne    0x55555555d766
10   0x000055555555d764 <+692>:   jmp    0x55555555d7b5
11   0x000055555555d766 <+694>:   mov    0x70(%rsp),%rax
12   0x000055555555d76b <+699>:   mov    0x8(%rsp),%rcx
13   0x000055555555d770 <+704>:   movb   $0x41,(%rax,%rcx,1)
14   0x000055555555d774 <+708>:   lea    0x78(%rsp),%rdi
```

Listing 4.11: Disassembly Of Patched Binary

Final Results

When providing the number 257 as an input the program does not crash and exits successfully.

```
1  index for accessing element:
2  257
3  input: "257\n"
4  index: 257
```

```
array A: [10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
     10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
      10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
     10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
      10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
     10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
      10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
     10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
      10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
     10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
      10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
     10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
      10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
```

Listing 4.12: Final Results

By inspecting the stack again, we can spot the difference. In the first stack representation, we can spot the array which is initialized with the value 10 (0xa0). Everything else seems regular, at least as regular as hexadecimal addresses can be.

```
+x/50gx arr_a
0x7fffffffd9c8: 0x0a0a0a0a0a0a0a0a      0x0a0a0a0a0a0a0a0a
0x7fffffffd9d8: 0x0a0a0a0a0a0a0a0a      0x0a0a0a0a0a0a0a0a
0x7fffffffd9e8: 0x0a0a0a0a0a0a0a0a      0x0a0a0a0a0a0a0a0a
0x7fffffffd9f8: 0x0a0a0a0a0a0a0a0a      0x0a0a0a0a0a0a0a0a
0x7fffffffda08: 0x0a0a0a0a0a0a0a0a      0x0a0a0a0a0a0a0a0a
0x7fffffffda18: 0x0a0a0a0a0a0a0a0a      0x0a0a0a0a0a0a0a0a
0x7fffffffda28: 0x0a0a0a0a0a0a0a0a      0x0a0a0a0a0a0a0a0a
0x7fffffffda38: 0x0a0a0a0a0a0a0a0a      0x0a0a0a0a0a0a0a0a
0x7fffffffda48: 0x0a0a0a0a0a0a0a0a      0x0a0a0a0a0a0a0a0a
0x7fffffffda58: 0x0a0a0a0a0a0a0a0a      0x0a0a0a0a0a0a0a0a
0x7fffffffda68: 0x0a0a0a0a0a0a0a0a      0x0a0a0a0a0a0a0a0a
0x7fffffffda78: 0x0a0a0a0a0a0a0a0a      0x0a0a0a0a0a0a0a0a
0x7fffffffda88: 0x0a0a0a0a0a0a0a0a      0x0a0a0a0a0a0a0a0a
0x7fffffffda98: 0x0a0a0a0a0a0a0a0a      0x0a0a0a0a0a0a0a0a
0x7fffffffdaa8: 0x0a0a0a0a0a0a0a0a      0x0a0a0a0a0a0a0a0a
0x7fffffffdab8: 0x0a0a0a0a0a0a0a0a      0x0a0a0a0a0a0a0a0a
0x7fffffffdac8: 0x0000000000000001      0x000055555559a4a8
0x7fffffffdad8: 0x00007fffffffdcd8      0x000055555558cf20
0x7fffffffdae8: 0x000055555555d811      0x0000000000000000
0x7fffffffdaf8: 0x000055555559a420      0x0000000000000001
0x7fffffffdb08: 0x0000000000000000      0x00000000deadbeef
0x7fffffffdb18: 0x0000000000000000      0x0000000000000000
0x7fffffffdb28: 0x000055555555d3db      0x00007ffff7ff000
0x7fffffffdb38: 0x000055555555ed0b      0x000055555559ea40
0x7fffffffdb48: 0x000055555559ea60      0x000055555555d3d0
```

Listing 4.13: Inspecting Memory Before Input

After the modification, we observe that a new value appeared shown in red. The hex value 41 equals to 65 in decimal. This means that the stack was modified, out of the legal bounds, of the array. Thus, we created an overwrite bug.

```
1  +x/50gx arr_a
2  0x7fffffffd9c8: 0x0a0a0a0a0a0a0a0a       0x0a0a0a0a0a0a0a0a
3  0x7fffffffd9d8: 0x0a0a0a0a0a0a0a0a       0x0a0a0a0a0a0a0a0a
4  0x7fffffffd9e8: 0x0a0a0a0a0a0a0a0a       0x0a0a0a0a0a0a0a0a
5  0x7fffffffd9f8: 0x0a0a0a0a0a0a0a0a       0x0a0a0a0a0a0a0a0a
6  0x7fffffffda08: 0x0a0a0a0a0a0a0a0a       0x0a0a0a0a0a0a0a0a
7  0x7fffffffda18: 0x0a0a0a0a0a0a0a0a       0x0a0a0a0a0a0a0a0a
8  0x7fffffffda28: 0x0a0a0a0a0a0a0a0a       0x0a0a0a0a0a0a0a0a
9  0x7fffffffda38: 0x0a0a0a0a0a0a0a0a       0x0a0a0a0a0a0a0a0a
10 0x7fffffffda48: 0x0a0a0a0a0a0a0a0a       0x0a0a0a0a0a0a0a0a
11 0x7fffffffda58: 0x0a0a0a0a0a0a0a0a       0x0a0a0a0a0a0a0a0a
12 0x7fffffffda68: 0x0a0a0a0a0a0a0a0a       0x0a0a0a0a0a0a0a0a
13 0x7fffffffda78: 0x0a0a0a0a0a0a0a0a       0x0a0a0a0a0a0a0a0a
14 0x7fffffffda88: 0x0a0a0a0a0a0a0a0a       0x0a0a0a0a0a0a0a0a
15 0x7fffffffda98: 0x0a0a0a0a0a0a0a0a       0x0a0a0a0a0a0a0a0a
16 0x7fffffffdaa8: 0x0a0a0a0a0a0a0a0a       0x0a0a0a0a0a0a0a0a
17 0x7fffffffdab8: 0x0a0a0a0a0a0a0a0a       0x0a0a0a0a0a0a0a0a
18 0x7fffffffdac8: 0x0000000000004101       0x000055555559a4a8
19 0x7fffffffdad8: 0x00007fffffffdcd8       0x000055555558cf20
20 0x7fffffffdae8: 0x000055555555d811       0x0000000000000000
21 0x7fffffffdaf8: 0x000055555559a420       0x0000000000000001
22 0x7fffffffdb08: 0x00007fffffffd9c8       0x00000000deadbeef
23 0x7fffffffdb18: 0x0000000000000000       0x00007fffffffd9c8
24 0x7fffffffdb28: 0x000055555555d3db       0x00007fffff7ff000
25 0x7fffffffdb38: 0x000055555555ed0b       0x000055555559ea40
26 0x7fffffffdb48: 0x000055555559ea60       0x000055555555d3d0
```

Listing 4.14: Inspecting Memory After Input

4.4 Documenting Bounds Check For Release Mode

In previous sections of this chapter, we examined a check that exists when statically allocated arrays are used. Before moving into the next section of this chapter where we talk about integer overflow bugs, we need to address an important technicality.

In previous sections when examining the assembly code, the check added by the compiler, consisted of a triplet of instructions (cmp, setb, test). In this section, we want to make the distinction between checks found in debug mode and release mode. The exploitation trials in previous sections were performed, on the debug version of each binary. The release version of the binary contains a different format, of the same check, from the one seen in the debug version.

We use the program below, compiled in release mode, and explore the disassembly of the new binary.

```
fn main() {
    //get input args
    let args: Vec<String> = env::args().collect();
    println!("{:?}", args);
    let range = args[1].parse::<i32>().unwrap();
    println!("{}", range);
    let mut arr_a: [i32; 10] = [0; 10];

    arr_a = copy_over(arr_a, range);
    println!("{:?}", arr_a);
}

fn copy_over(mut arr_a: [i32; 10], range: i32) -> [i32; 10] {
    for i in 0..range {
        println!("{}", i);
        arr_a[i as usize] = i as i32;
    }
    return arr_a;
}
```

In the listing below, we examine a small disassembly snippet. The commands that are of utmost importance are shown in green colour. This is the optimized version of the check we have explored in previous sections.

```
0x000055555555c8fe <+430>:   movq    $0x1,0x38(%rsp)
0x000055555555c907 <+439>:   mov     %r13,%rdi
0x000055555555c90a <+442>:   call    *%r12
0x000055555555c90d <+445>:   mov     0xc(%rsp),%eax
0x000055555555c911 <+449>:   cmp     $0x9,%rax
0x000055555555c915 <+453>:   ja      0x55555555ca2a
0x000055555555c91b <+459>:   add     $0x1,%ebx
0x000055555555c91e <+462>:   mov     %eax,0x40(%rsp,%rax,4)
0x000055555555c922 <+466>:   cmp     %ebx,%ebp
0x000055555555c924 <+468>:   jne     0x55555555c8c0
0x000055555555c926 <+470>:   mov     0x60(%rsp),%rax
0x000055555555c92b <+475>:   mov     %rax,0xb0(%rsp)
0x000055555555c933 <+483>:   movaps  0x40(%rsp),%xmm0
0x000055555555c938 <+488>:   movaps  0x50(%rsp),%xmm1
0x000055555555c93d <+493>:   movaps  %xmm1,0xa0(%rsp)
```

Listing 4.15: Observing Optimized Disassembly

By observing the check, we can extract its functionality. It compares %rax which is our index with a fixed value (the length of the array -1 (10 - 1 = 9)). Then if the %rax value is greater than 9, the program jumps to the specified address. By running the program step by step in GDB [10], we know that after jumping to that address the program terminates with a panic error. We continue this distinction between binaries built in debug mode and release mode in the evaluation chapter.

4.5 Proof Of Concept 4: Integer Overflow Check

In this section, we examine integer overflows and potential checks added by the compiler. This section is split into two parts debug mode and release mode.

To explore this case, we use the following toy example. The program gets two numbers and uses the function unchecked to add them. We are using unsigned 8-bit types, so we don't need large numbers to cause overflow. Thus, in this case, if the sum of arguments surpasses 255 we have an integer overflow.

```
use std::env;

fn unchecked(x: u8, y: u8) {
    let z = x + y;
    println!("{} + {} = {:?}\n", x, y, z);
}

fn main() {
    let args: Vec<String> = env::args().collect();
    if args.len() == 1 {
        return;
    }
    let arg1 = args.get(1).unwrap().parse::<u8>().unwrap();
    let arg2 = args.get(2).unwrap().parse::<u8>().unwrap();
    println!("{:?}, {:?}", arg1, arg2);
    unchecked(arg1, arg2);
}
```

By reading the documentation and RFC 560 [21] we learn that in debug mode arithmetic operations like +, or – of primitive types are checked for overflow, but in release mode that checking is disabled and the result wraps into two's complement. Understandably, this kind of check is disabled as it can impose a high overhead in performance if is in a basic block that is frequently executed. The good news is that it can be manually enabled. Nevertheless, the goal of this section is to try and locate how the check happens. We then try to remove it and see if we can cause undefined behaviour in a program that supposedly has this check enabled.

4.5.1 Debug Mode

Firstly, we use debug mode, as it's the standard compilation and provides debug symbols, which will give us a better understanding.

Running First Example

```
cargo run 255 1
255, 1
thread 'main' panicked at 'attempt to add with overflow', src/main.rs:7:13
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Listing 4.16: Initial Results

By running the program with arguments 255 and 1 we cause an overflow to type u8 since its max value is 255. We get the above error and dive into the assembly code to locate the check for this overflow bug.

Exploring Disassembly

```
1  Dump of assembler code for function ZN16toy19_ovrfl_expl9unchecked17h08ff5b5dd30130a0E:
2     0x0000555555561090 <+0>:     sub    $0xd8,%rsp
3     0x0000555555561097 <+7>:     mov    %sil,%al
4     0x000055555556109a <+10>:    mov    %dil,%cl
5     0x000055555556109d <+13>:    mov    %cl,0x45(%rsp)
6     0x00005555555610a1 <+17>:    mov    %al,0x46(%rsp)
7  => 0x00005555555610a5 <+21>:    mov    0x45(%rsp),%al
8     0x00005555555610a9 <+25>:    add    0x46(%rsp),%al
9     0x00005555555610ad <+29>:    mov    %al,0x44(%rsp)
10    0x00005555555610b1 <+33>:    setb   %al
11    0x00005555555610b4 <+36>:    test   $0x1,%al
12    0x00005555555610b6 <+38>:    jne    0x5555555611ee
13    0x00005555555610bc <+44>:    mov    0x44(%rsp),%al
14    ===skip some instructions===
15    0x00005555555611db <+331>:   lea    0x48(%rsp),%rdi
16    0x00005555555611e0 <+336>:   call   *0x3e7e2(%rip)        # 0x55555559f9c8
17    0x00005555555611e6 <+342>:   add    $0xd8,%rsp
18    0x00005555555611ed <+349>:   ret
19    0x00005555555611ee <+350>:   lea    0x2e0eb(%rip),%rdi   # 0x55555558f2e0
20    0x00005555555611f5 <+357>:   lea    0x3c26c(%rip),%rdx   # 0x55555559d468
21    0x00005555555611fc <+364>:   lea    -0x5553(%rip),%rax   #call panic
22    0x0000555555561203 <+371>:   mov    $0x1c,%esi
23    0x0000555555561208 <+376>:   call   *%rax
24    0x000055555556120a <+378>:   ud2
25 End of assembler dump.
```

Listing 4.17: Observing Disassembly

In the above listing we have the disassembly, of the unchecked function. We highlight two points of interest one in red and one in green colour. Let's analyse the red segment first. The program performs the addition of the two numbers in line 8. Then moves the result in the stack from the %al register. After storing the result in the stack, the program checks the result to see if an overflow occurred. This check is performed by a triplet of control flow instructions (setb, test, jne).

What does the check do?

1. Set byte on %al if below: if CF flag equals to 1.
2. Test %al with 0x1. Perform a logical AND operation and set the ZF flag.
3. Jump to address if not equal to zero. Jump if ZF=0.

The address that jumps to in case of overflow is the green section, which is also the section that calls the panic error handler and crashes the program.

Patching And Bypassing The Check

From the above example, we understand that what happens can be "fixed" by bypassing the jump to the panic section. In other words, what we need to do is convert the jne instruction to a regular jump instruction, that will always jump unconditionally to the next instruction.

```
1     0x00005555555610a1 <+17>:  mov    %al,0x46(%rsp)
2  => 0x00005555555610a5 <+21>:  mov    0x45(%rsp),%al
3     0x00005555555610a9 <+25>:  add    0x46(%rsp),%al
4     0x00005555555610ad <+29>:  mov    %al,0x44(%rsp)
5     0x00005555555610b1 <+33>:  setb   %al
6     0x00005555555610b4 <+36>:  test   $0x1,%al
7     0x00005555555610b6 <+38>:  jmp    0x5555555610bc
8     0x00005555555610b8 <+40>:  xor    (%rcx),%al
9     0x00005555555610ba <+42>:  add    %al,(%rax)
10    0x00005555555610bc <+44>:  mov    0x44(%rsp),%al
11    0x00005555555610c0 <+48>:  mov    %al,0x47(%rsp)
12    0x00005555555610c4 <+52>:  lea    0x45(%rsp),%rax
```

Listing 4.18: Disassembly Of Binary After Patching

Now the program always jumps to the specified address and ignores the flags set by the setb and test commands. In this case, the jmp instruction was smaller than the previous jne instruction, thus we have some junk instructions created below the jmp instruction. Fortunately, the binary does not shift, and these commands don't cause any implications as the program always jumps past them. If the jmp was larger than the replaced command, we would not have been able to bypass the check this way.

Results After Patching

```
1  255, 1
2  255 + 1 = 0
```

Listing 4.19: Results After Patching

In the above output, we can see that the result was wrapped around and instead of crashing which is the expected result, the result of the addition is zero.

4.5.2 Release Mode

In this subsection, we bypass the check in the release version of the program. Since the check was bypassed the same way as in debug mode we won't go into much detail. But since the compiler applies optimization techniques in release mode, we want to investigate how much these optimizations change the check.

```
1  Dump of assembler code for function _ZN16toy19_ovrfl_expl4main17h0be36dcdc195531aE:
2  => 0x000055555555cbb0 <+0>:     push   %r15
3     0x000055555555cbb2 <+2>:     push   %r14
4     0x000055555555cbb4 <+4>:     push   %r13
5     ===skip some instructions===
6     0x000055555555ccf7 <+327>:   mov    %al,0xd(%rsp)
7     0x000055555555ccfb <+331>:   mov    %cl,0xe(%rsp)
8     0x000055555555ccff <+335>:   add    %cl,%al
9     0x000055555555cd01 <+337>:   jb     0x55555555cde8
10    0x000055555555cd07 <+343>:   mov    %al,0xf(%rsp)
11    0x000055555555cd0b <+347>:   lea    0xd(%rsp),%rax
12    0x000055555555cd10 <+352>:   mov    %rax,0x10(%rsp)
13    0x000055555555cd15 <+357>:   lea    0x2c794(%rip),%rax        # 0x5555555894b0
14    ===skip some instructions===
15    0x000055555555cde3 <+563>:   pop    %r14
16    0x000055555555cde5 <+565>:   pop    %r15
17    0x000055555555cde7 <+567>:   ret
18    0x000055555555cde8 <+568>:   lea    0x2e291(%rip),%rdi  # 0x55555558b080 <str.0>
19    0x000055555555cdef <+575>:   lea    0x3a652(%rip),%rdx        # 0x555555597448
20    0x000055555555cdf6 <+582>:   mov    $0x1c,%esi
21    0x000055555555cdfb <+587>:   call   *0x3cd67(%rip)        # 0x555555599b68
22    0x000055555555ce01 <+593>:   jmp    0x55555555ce90
```

In the above snippet, we observe that instead of three instructions, only one is used (seen in red colour). The "jb" instruction jumps to the address if the carry flag is set to 1 (CF=1) and calls the panic error handler. Thus, we can bypass that check again by adding a regular jmp command, that will always jump to the next instruction, and handle the check the same way we did in debug mode.

Chapter 5

Temporal Safety

In this chapter, we will discuss temporal safety. As we already know from the previous chapter dedicated in spatial safety the compiler of Rust adds checks in assembly code, to prevent buffer overflows, int overflows, etc. Regarding temporal safety, we need to dive into another concept of the Rust compiler, called the borrow checker. The borrow checker is a program run at compile-time, that aims to enforce temporal safety.

For example, in Rust, we cannot declare two mutable references for the same variable/object. If we do that the borrow checker will catch this and mark our code as invalid. In this chapter, we try to create some bugs, regarding temporal safety and try to bypass these rules enforced by the borrow checker. In each section of this chapter, we try to exploit a different concept of the borrow checker.

Throughout this chapter we try to answer some of the following questions

1. Can a double mutable reference be created by modifying the binary?
2. Can use-after-free bugs be created?
3. Can the concept of lifetimes be exploited?

5.1 Proof Of Concept 1: Double Mutable References In Integer Wrapper Object

5.1.1 Concept

In this section, we create the first proof of concept. For this proof of concept, we want to construct a double mutable reference to the same object by modifying the binary. Before diving into a more complex example, we will use a wrapper object for the integer primitive type.

5.1.2 Toy Example Program And Initial Results

The toy example for this POC is the following. Let us go through what this example aims to achieve before we dive into the assembly code. We have two different functions, int_correct_values and int_box_values. The first function prints the correct values for the two integers and the other one allocates two integers in heap. In the second function, we want to make pointer p2, to access the data of pointer p1 so that when we manipulate p2 we manipulate data of p1. This concept is the concept of having two references to the same object in C/C++. In Rust, we cannot have two mutable references to the same memory. Many immutable references are fine, but only one mutable reference is allowed to the same object at a time. Much like the readers-writers problem when we think about concurrency concepts. We modify the binary, in function int_box_references, to make pointer p2 point at data of p1, in this case, point to integer 5. After the modification, we expect the following results: p1=11 and p2=11.

```
fn int_correct_values() {
    let mut p1 = 5;
    p1 += p1;
    let mut p2 = 6;
    p2 += 1;
    println!("pointer p1: {:?}\npointer p2: {:?}", p1, p2);
}

fn int_box_values() {
    let mut p1 = Box::new(5);
    *p1 += *p1;
    let mut p2 = Box::new(6);
    *p2 += 1;
    println!("pointer p1: {:?}\npointer p2: {:?}", p1, p2);
}

fn main() {
    println!("Correct values:");
    int_correct_values();
    println!("Changed references:");
    int_box_values();
}
```

Initial Results

When running the binary we get the following results. Both functions are identical in this case so their results must match.

```
1  Correct values:
2  pointer p1: 10
3  pointer p2: 7
4  Changed references:
5  pointer p1: 10
6  pointer p2: 7
```

Listing 5.1: Initial Results

5.1.3 Disassembly And Course Of Action

Now that we investigated the first results we dive into the assembly and find the changes needed, to make the double mutable reference a possibility. We want to do as few changes as possible.

```
1     0x000055555555c09d <+141>:   mov    0x48(%rsp),%rax
2     0x000055555555c0a2 <+146>:   mov    %rax,0x68(%rsp)
3  => 0x000055555555c0a7 <+151>:   mov    0x68(%rsp),%rax
4     0x000055555555c0ac <+156>:   mov    (%rax),%eax
5     0x000055555555c0ae <+158>:   inc    %eax
6     0x000055555555c0b0 <+160>:   mov    %eax,0x44(%rsp)
7     0x000055555555c0b4 <+164>:   seto   %al
8     0x000055555555c0b7 <+167>:   test   $0x1,%al
9     0x000055555555c0b9 <+169>:   jne    0x55555555c23f
10    ===Skip Some Instructions===
11    0x000055555555c1c6 <+438>: jmp    0x55555555c1c8
12    0x000055555555c1c8 <+440>: lea    0x68(%rsp),%rdi
13    0x000055555555c1cd <+445>: call   0x55555555be20  #call destructor
14    0x000055555555c1d2 <+450>: jmp    0x55555555c1d4
15    0x000055555555c1d4 <+452>: lea    0x60(%rsp),%rdi
16    0x000055555555c1d9 <+457>: call   0x55555555be20  #call destructor
```

Listing 5.2: Disassembly Before Modification

The code in the above listing is the code executed before incrementing p2 by one. See line 5 highlighted with green colour. Fortunately, between the storing of p2 (line 1) and the modification (line 5), we can patch commands and make the program store the address of p1 in the reserved address for p2. Simply by modifying line 1, we can create the double mutable reference.

By examining the blue segment of the assembly, we can see where in the stack the two references are stored. The blue part shows the deconstruction phase of the two objects. Object p2 is stored in offset 0x68 and p1 in offset 0x60. To create the double mutable reference, we need to store in 0x68 the value stored in 0x60 offset (object p1).

We use the more convenient way, which is to patch the command in line 1 with a "mov 0x60 (%rsp), %rax" command and then let the program continue with regular execution storing %rax in 0x68 as it's supposed to.

5.1.4 Patching And Bypassing

```
0x000055555555c09d <+141>:   mov    0x60(%rsp),%rax
0x000055555555c0a2 <+146>:   mov    %rax,0x68(%rsp)
0x000055555555c0a7 <+151>:   mov    0x68(%rsp),%rax
0x000055555555c0ac <+156>:   mov    (%rax),%eax
0x000055555555c0ae <+158>:   inc    %eax
0x000055555555c0b0 <+160>:   mov    %eax,0x44(%rsp)
0x000055555555c0b4 <+164>:   seto   %al
0x000055555555c0b7 <+167>:   test   $0x1,%al
0x000055555555c0b9 <+169>:   jne    0x55555555c23f
```

Listing 5.3: Disassembly After Modification

This is the same snippet of code after the patching. The change is small, but we can successfully bypass an important restriction of the borrow checker.

5.1.5 Results and Conclusions

After running the patched binary, we get the following results:

```
Correct values:
pointer p1: 10
pointer p2: 7
Changed references:
pointer p1: 11
pointer p2: 11
free(): double free detected in tcache 2
Aborted
```

Listing 5.4: Results After Modification

We succeeded in creating a double reference and achieved the expected results for the two pointers. The results are correct since we created a double mutable reference of p1 in pointer p2. Thus, if we remember our initial code.

```
fn int_box_values() {
    let mut p1 = Box::new(5);
    *p1 += *p1;
    let mut p2 = Box::new(6);
    *p2 += 1;
    println!("pointer p1: {:?}\npointer p2: {:?}", p1, p2);
}
```

Pointer p1 is added to itself, thus after the creation of the p2 object, we have p1=10 and p2=6. Then we perform the attack as described above and both p1 and p2 are equal to 10. Pointer p2 is incremented by one which increments the memory of object p1 which now equals 11. Finally, since both objects point to the same memory, they are both equal to 11.

But as we can see the program crashes with a double-free error. This happens because, at the end of the scope/ function of our example, both objects are dropped. To fix this we just bypass one of the destructors.

```
1    0x000055555555c1c6 <+438>:   jmp    0x55555555c1c8
2    0x000055555555c1c8 <+440>:   lea    0x68(%rsp),%rdi
3    0x000055555555c1cd <+445>:   call   0x55555555be20  #call destructor
4    0x000055555555c1d2 <+450>:   jmp    0x55555555c1de
5    0x000055555555c1d4 <+452>:   lea    0x60(%rsp),%rdi
6    0x000055555555c1d9 <+457>:   call   0x55555555be20  #call destructor
7    0x000055555555c1de <+462>:   add    $0xf8,%rsp
8    0x000055555555c1e5 <+469>:   ret
```

Listing 5.5: Fixing Double Free Error

The program now jumps from line 4 to line 7 bypassing the call to the second destructor in line 6. In the final results, the program produces the expected results without crashing.

```
1  Results:
2  Correct values:
3  pointer p1: 10
4  pointer p2: 7
5  Changed references:
6  pointer p1: 11
7  pointer p2: 11
```

Listing 5.6: Final results

5.2 Proof Of Concept 2: Double Mutable References With Vector Objects

5.2.1 Concept

In this example, we try to recreate the double mutable reference bug of the previous section but instead of integer wrappers, we introduce double mutable references using vectors.

5.2.2 Toy Example Program

The toy example we explore is constructed using the same manner as our previous toy example.

We want to create an artificial double mutable reference for vector v1. The expected results for this example are the following: we want to create 2 different objects v1 and v2 where v1=1 and v2=2. After the mutable reference creation, we should be able to use v2 as a handle for v1. Thus, we expect the result to be v1=1,2,2,1 and v2=1,2,2,1. Below we present the code for the toy example.

```
fn vector_references() {
    let mut v1: Vec<u8> = Vec::new();
    v1.push(1);
    let mut v2: Vec<u8> = Vec::new();
    v2.push(2);
    println!("Vectors init:\nvector1: {:?}\nvector2: {:?}", v1, v2);
    v2.push(2);
    v2.push(2);
    v1.push(1);
    println!("\nvector1: {:?}\nvector2: {:?}", v1, v2);
}

fn main() {
    vector_references();
}
```

Initial Results

Running the program above we get the following initial results.

```
Vectors init:
vector1: [1]
vector2: [2]

vector1: [1, 1]
vector2: [2, 2, 2]
```

Listing 5.7: Initial Results

5.2.3 Disassembly And Course Of Action

In this part of the POC, we dive into the assembly and create the artificial bug. First of all, we need to locate where the two objects are allocated. Below we introduce snippets of the assembly code for the function. Before examining the code, we need to make sure we know exactly what we want to do. To get the result described above we need to create a double reference at a certain point (after the first println call, and before the next push instruction at line 7). This will ensure that all push instructions affect the same piece of memory, meaning that both handles v1 and v2 point to the same memory. To do that we isolate the important pieces from the disassembly, and explain them before making any changes to the binary.

```
1     0x000055555555e0b0 <+0>:     sub    $0x198,%rsp
2  => 0x000055555555e0b7 <+7>:     lea    0x78(%rsp),%rdi
3     0x000055555555e0bc <+12>:    call   0x55555555d750
4     0x000055555555e0c1 <+17>:    lea    0x78(%rsp),%rdi
5     0x000055555555e0c6 <+22>:    mov    $0x1,%esi
6     0x000055555555e0cb <+27>:    call   0x55555555d7b0
7     0x000055555555e0d0 <+32>:    jmp    0x55555555e0d2
8     0x000055555555e0d2 <+34>:    lea    0x90(%rsp),%rdi
9     0x000055555555e0da <+42>:    call   0x55555555d750
10    0x000055555555e0df <+47>:    jmp    0x55555555e0e1
11    0x000055555555e0e1 <+49>:    lea    0x90(%rsp),%rdi
12    0x000055555555e0e9 <+57>:    mov    $0x2,%esi
13    ===Skip Some Instructions===
14    0x000055555555e1f0 <+320>:   lea    0xa8(%rsp),%rdi
15    0x000055555555e1f8 <+328>:   call   *%rcx
16    0x000055555555e1fa <+330>:   jmp    0x55555555e1fc
17    0x000055555555e1fc <+332>:   lea    0x90(%rsp),%rdi
18    0x000055555555e204 <+340>:   mov    $0x2,%esi
19    0x000055555555e209 <+345>:   call   0x55555555d7b0
20    0x000055555555e20e <+350>:   jmp    0x55555555e210
21    ===Skip Some Instructions===
22    0x000055555555e33a <+650>:   jmp    0x55555555e33c
23    0x000055555555e33c <+652>:   lea    0x90(%rsp),%rdi
24    0x000055555555e344 <+660>:   call   0x55555555e990 #call destructor
25    0x000055555555e349 <+665>:   jmp    0x55555555e34b
26    0x000055555555e34b <+667>:   lea    0x78(%rsp),%rdi
27    0x000055555555e350 <+672>:   call   0x55555555e990 #call destructor
28    0x000055555555e355 <+677>:   add    $0x198,%rsp
29    0x000055555555e35c <+684>:   ret
30    0x000055555555e35d <+685>:   lea    0x90(%rsp),%rdi
31    0x000055555555e365 <+693>:   call   0x55555555e990
32    0x000055555555e36a <+698>:   lea    0x78(%rsp),%rdi
33    0x000055555555e36f <+703>:   call   0x55555555e990
34    0x000055555555e374 <+708>:   mov    0x168(%rsp),%rdi
35    0x000055555555e37c <+716>:   call   0x555555559050
36    0x000055555555e381 <+721>:   ud2
37    0x000055555555e383 <+723>:   mov    %rax,%rcx
38    0x000055555555e386 <+726>:   mov    %edx,%eax
39    0x000055555555e388 <+728>:   mov    %rcx,0x168(%rsp)
40    0x000055555555e390 <+736>:   mov    %eax,0x170(%rsp)
41    0x000055555555e397 <+743>:   jmp    0x55555555e36a
42    0x000055555555e399 <+745>:   mov    %rax,%rcx
43    0x000055555555e39c <+748>:   mov    %edx,%eax
44    0x000055555555e39e <+750>:   mov    %rcx,0x168(%rsp)
45    0x000055555555e3a6 <+758>:   mov    %eax,0x170(%rsp)
46    0x000055555555e3ad <+765>:   jmp    0x55555555e35d
47 End of assembler dump.
```

Listing 5.8: Disassembly Before Modification

In the above listing, we isolate three parts: green, red, and blue. We go through them one by one. First, we need to locate the objects v1 and v2. An easy way to do so is to look at the blue-coloured instructions which are followed by calls to the destructor (lines 23 and 26). The two objects we are searching for are stored in those addresses. If we look at the start of the code lines 4 and 5 the mov instruction uses the fixed value of 1 which

corresponds to the first vector and lines 11 and 12 correspond to the second vector (mov instruction uses a fixed value of 2).

Now that we know which handle corresponds to which address, we need to locate the ideal place to patch the binary. By inspecting the code, we want to execute the first print command and then execute the second print after constructing the double reference.

We focus on two parts the red-coloured part and the green one. The green one is just padding that we use to write some instructions and create the double mutable reference. We plan on jumping to the padding code, executing those instructions, and then jumping back. The red-coloured part represents the ideal instruction to use, to create the double mutable reference. Specifically, at that point of the program, we already have the address of vector v2 in %rdi and, we can use the mov instruction (line 18) to jump to the padding section without shifting the binary (the mov instruction accommodates the 5-byte jump we need to jump to the padding code).

In the next subsection, we provide the changes made, in the points of interest shown in this listing.

5.2.4 Patching And Bypassing

```
1   ===Skip Some Instructions===
2   0x000055555555e1f8 <+328>:   call   *%rcx
3   0x000055555555e1fa <+330>:   jmp    0x55555555e1fc
4   0x000055555555e1fc <+332>:   lea    0x90(%rsp),%rdi
5   0x000055555555e204 <+340>:   jmp    0x55555555e35d
6   0x000055555555e209 <+345>:   call   0x55555555d7b0
7   ===Skip Some Instructions===
8   0x000055555555e35c <+684>:   ret
9   0x000055555555e35d <+685>:   mov    0x78(%rsp),%rdi
10  0x000055555555e362 <+690>:   mov    %rdi,0x90(%rsp)
11  0x000055555555e36a <+698>:   lea    0x90(%rsp),%rdi
12  0x000055555555e372 <+706>:   mov    $0x2,%esi
13  0x000055555555e377 <+711>:   jmp    0x55555555e209
14  0x000055555555e37c <+716>:   call   0x555555559050 <_Unwind_Resume@plt>
```

Listing 5.9: Disassembly After Modification

Again, the two parts are highlighted with red and green colours. In line 5 we replaced the previous mov with a jump instruction to the address where the padding code starts.

The padding code was modified with the following logic. First, we remember that object v1 is stored in offset 0x78 and v2 is stored in offset 0x90. We use only one register, %rdi to minimize the impact on the state of the program. We move the address of the v1 pointer to %rdi and then store the value of %rdi in the address with offset 0x90 (lines 9 and 10) which makes the double reference a reality as v2 now points to v1. Following the mutable reference creation, we need to restore the state before jumping back to the function. Thus, we restore the value in %rdi and execute the mov command we replaced.

5.2.5 Results And Conclusions

```
Vectors init:
vector1: [1]
vector2: [2]

vector1: [1, 1]
vector2: [1, 1, 2]
free(): double free detected in tcache 2
Aborted
```

Listing 5.10: Results After Modification

Firstly, we have the same double-free error, which we know how to bypass from the previous proof of concept, thus we won't cover that. The peculiar thing is we don't get the expected results. On the contrary, we have very different results from the expected. The expected results were v1= 1,2,2,1 and v2= 1,2,2,1.

By observing the results in the above listing in second place we have number 1 instead of 2, in both vectors. To make things worse we do not have a vector that is of length 4. To further investigate this, we use gdb and observe operations step by step.

If we examine the two vectors before the jump to the padding code, both have different pointer addresses.

```
+p v1
$3 = alloc::vec::Vec<u8, alloc::alloc::Global> {
  buf: alloc::raw_vec::RawVec<u8, alloc::alloc::Global> {
    ptr: core::ptr::unique::Unique<u8> {
      pointer: 0x55555559dad0,
      _marker: core::marker::PhantomData<u8>
    },
    cap: 8,
    alloc: alloc::alloc::Global
  },
  len: 1
}
>>> p v2
$4 = alloc::vec::Vec<u8, alloc::alloc::Global> {
  buf: alloc::raw_vec::RawVec<u8, alloc::alloc::Global> {
    ptr: core::ptr::unique::Unique<u8> {
      pointer: 0x55555559daf0,
      _marker: core::marker::PhantomData<u8>
    },
    cap: 8,
    alloc: alloc::alloc::Global
  },
  len: 1
}
```

Listing 5.11: Examining Vector Before Mutable Reference Creation

After creating the mutable references, everything seems fine as both vectors point to the same address.

```
>>> p v1
+p v1
$5 = alloc::vec::Vec<u8, alloc::alloc::Global> {
  buf: alloc::raw_vec::RawVec<u8, alloc::alloc::Global> {
    ptr: core::ptr::unique::Unique<u8> {
      pointer: 0x55555559dad0,
      _marker: core::marker::PhantomData<u8>
    },
    cap: 8,
    alloc: alloc::alloc::Global
  },
  len: 1
}
>>> p v2
+p v2
$6 = alloc::vec::Vec<u8, alloc::alloc::Global> {
  buf: alloc::raw_vec::RawVec<u8, alloc::alloc::Global> {
    ptr: core::ptr::unique::Unique<u8> {
     pointer: 0x55555559dad0,
      _marker: core::marker::PhantomData<u8>
    },
    cap: 8,
    alloc: alloc::alloc::Global
  },
  len: 1
}
```

Listing 5.12: Examining Vector After Mutable Reference Creation

After some inspection, we find the problem, in the metadata information for each vector. Basically, after adding each element, only the metadata for the handle we use is updated. This results in one handle having obsolete metadata values and causing problems. The following snippet from gdb shows exactly the problem and how we achieve the expected results by manually updating the metadata (setting the length of each vector manually).

```

v2.push(2);
v2.push(2);
>>> p v1
$7 = alloc::vec::Vec<u8, alloc::alloc::Global> {
  buf: alloc::raw_vec::RawVec<u8, alloc::alloc::Global> {
    ptr: core::ptr::unique::Unique<u8> {
      pointer: 0x55555559dad0,
      _marker: core::marker::PhantomData<u8>
    },
    cap: 8,
    alloc: alloc::alloc::Global
  },
  len: 1
}
>>> p v2
$8 = alloc::vec::Vec<u8, alloc::alloc::Global> {
  buf: alloc::raw_vec::RawVec<u8, alloc::alloc::Global> {
    ptr: core::ptr::unique::Unique<u8> {
```

```
20          pointer: 0x55555559dad0,
21          _marker: core::marker::PhantomData<u8>
22        },
23        cap: 8,
24        alloc: alloc::alloc::Global
25      },
26      len: 3
27    }
28     >>> set v1.len=3
29    v1.push(1);
30    >>> p v1
31    $11 = alloc::vec::Vec<u8, alloc::alloc::Global> {
32      buf: alloc::raw_vec::RawVec<u8, alloc::alloc::Global> {
33        ptr: core::ptr::unique::Unique<u8> {
34          pointer: 0x55555559dad0,
35          _marker: core::marker::PhantomData<u8>
36        },
37        cap: 8,
38        alloc: alloc::alloc::Global
39      },
40      len: 4
41    }
42    >>> p v2
43    $12 = alloc::vec::Vec<u8, alloc::alloc::Global> {
44      buf: alloc::raw_vec::RawVec<u8, alloc::alloc::Global> {
45        ptr: core::ptr::unique::Unique<u8> {
46          pointer: 0x55555559dad0,
47          _marker: core::marker::PhantomData<u8>
48        },
49        cap: 8,
50        alloc: alloc::alloc::Global
51      },
52      len: 3
53    }
54    >>> set v2.len=4
55
56    >>> p v1
57    $13 = alloc::vec::Vec<u8, alloc::alloc::Global> {
58      buf: alloc::raw_vec::RawVec<u8, alloc::alloc::Global> {
59        ptr: core::ptr::unique::Unique<u8> {
60          pointer: 0x55555559dad0,
61          _marker: core::marker::PhantomData<u8>
62        },
63        cap: 8,
64        alloc: alloc::alloc::Global
65      },
66      len: 4
67    }
68    >>> p v2
69    $14 = alloc::vec::Vec<u8, alloc::alloc::Global> {
70      buf: alloc::raw_vec::RawVec<u8, alloc::alloc::Global> {
71        ptr: core::ptr::unique::Unique<u8> {
72          pointer: 0x55555559dad0,
73          _marker: core::marker::PhantomData<u8>
74        },
75        cap: 8,
76        alloc: alloc::alloc::Global
```

```
77      },
78    len: 4
79  }
80
81  vector1: [1, 2, 2, 1]
82  vector2: [1, 2, 2, 1]
```

Listing 5.13: Metadata Not Updating And Setting Them Manually Through GDB

5.3 Proof Of Concept 3: Use After Free bug In Integer Wrapper

5.3.1 Concept

In this proof of concept, we construct a use after free bug. We create this bug in two stages. In the first stage, we explore if it's possible to create a use-after-free bug and in the second stage we fabricate an object and try to create the use after free for that object. Since this POC is just an introduction to get familiar with what we want to exploit in the next POC we describe it very briefly.

5.3.2 Toy Example Program

```
1  fn int_box_ref_v2() -> Box<i32> {
2      let mut p1 = Box::new(5);
3      *p1 += *p1;
4      let mut p2 = Box::new(6);
5      *p2 += 1;
6      println!("In function: pointer p1: {:?}, pointer p2: {:?}", p1, p2);
7      return p2;
8  }
9
10 fn main() {
11     let mut p2 = int_box_ref_v2();
12     *p2 += 1;
13     println!("Main: pointer p2: {:?}", p2);
14 }
```

We create a double reference, using the method shown in the previous POC. The goal is to be able to access the dropped memory space using the second handle that is returned from the function. Pointer p2 now point to the data of p1 which lives only in the scope of function int_box_ref_v2(). Thus, if we can access the memory outside, we will create a use-after-free bug.

Initial Results

```
In function: pointer p1: 10, pointer p2: 7
Main: pointer p2: 8
```

Listing 5.14: Initial Results

5.3.3 Disassembly And Course Of Action

Taking the same course of action as in POC 1, we only need to modify one instruction to change the addresses between the two objects (p1, p2)

```
   0x000055555555bf48 <+120>: jmp    0x55555555bf4a
   0x000055555555bf4a <+122>: mov    0x50(%rsp),%rax
   0x000055555555bf4f <+127>: mov    %rax,%rcx
   0x000055555555bf52 <+130>: mov    %rcx,0x48(%rsp)
   0x000055555555bf57 <+135>: movl   $0x6,(%rax)
   0x000055555555bf5d <+141>: mov    0x48(%rsp),%rax
   0x000055555555bf62 <+146>: mov    %rax,0x60(%rsp)
   0x000055555555bf67 <+151>: mov    0x60(%rsp),%rax
   0x000055555555bf6c <+156>: mov    (%rax),%eax
   0x000055555555bf6e <+158>: inc    %eax
   0x000055555555bf70 <+160>: mov    %eax,0x44(%rsp)
   0x000055555555bf74 <+164>: seto   %al
   0x000055555555bf77 <+167>: test   $0x1,%al
   0x000055555555bf79 <+169>: jne    0x55555555c0f8
```

Listing 5.15: Disassembly Before Modification

We modify the instruction that loads the address of the pointer p2 from a temporary address into %rax. Then, in the next command %rax is stored in the address reserved for the whole scope of the function to access the p2 object. This way we have successfully created a double mutable reference.

5.3.4 Patching And Bypassing

```
   0x000055555555bf48 <+120>: jmp    0x55555555bf4a
   0x000055555555bf4a <+122>: mov    0x50(%rsp),%rax
   0x000055555555bf4f <+127>: mov    %rax,%rcx
   0x000055555555bf52 <+130>: mov    %rcx,0x48(%rsp)
   0x000055555555bf57 <+135>: movl   $0x6,(%rax)
   0x000055555555bf5d <+141>: mov    0x68(%rsp),%rax
   0x000055555555bf62 <+146>: mov    %rax,0x60(%rsp)
   0x000055555555bf67 <+151>: mov    0x60(%rsp),%rax
   0x000055555555bf6c <+156>: mov    (%rax),%eax
   0x000055555555bf6e <+158>: inc    %eax
   0x000055555555bf70 <+160>: mov    %eax,0x44(%rsp)
   0x000055555555bf74 <+164>: seto   %al
   0x000055555555bf77 <+167>: test   $0x1,%al
   0x000055555555bf79 <+169>: jne    0x55555555c0f8
```

Listing 5.16: Disassembly After Modification

The instruction in the previous listing (line 6) changed as shown in the listing above. Moreover, to avoid crashing, we bypassed the destructor in main to avoid a double-free error. The program jumps from line 1 to line 4, bypassing the destructor.

```
1    0x000055555555c1de <+174>:   jmp    0x55555555c1ea
2    0x000055555555c1e0 <+176>:   lea    0x20(%rsp),%rdi
3    0x000055555555c1e5 <+181>:   call   0x55555555be40    #call destructor
4    0x000055555555c1ea <+186>:   add    $0x88,%rsp
5    0x000055555555c1f1 <+193>:   ret
```

Listing 5.17: Bypassing Destructor

5.3.5 Results And Conclusions

After running the modified binary, we are treated with a peculiar result once again. Inside the function, results seem fine, as the double reference seemed to work correctly, like in POC 1. In main though we have an unexpected result. Instead of printing the expected result which is 12 (11+1), we get result 1.

```
1  In function: pointer p1: 11, pointer p2: 11
2  Main: pointer p2: 1
```

Listing 5.18: Final Results

We easily understand why by inspecting memory using gdb. In line 4 we examine the memory of the address that p2 points to, memory does not contain the value we expected, instead, it's zero. Thus, adding value 1 to zero results in value 1. In the next section we will explain this phenomenon more, but long story short, after the destructor is executed the memory state of the object changes in some way. In real-life scenarios, we don't really care as we would want to fill memory with the data of choice and then use the dangling pointer which we successfully created, to access it.

```
1  +p p2
2  $5 = (*mut i32) 0x55555559aad0
3  +x/gx 0x55555559aad0
4  0x55555559aad0:    0x0000000000000000
5  +n
6  0x000055555555c146    23      *p2 += 1;
7  +n
8  24      println!("Main: pointer p2: {:?}", p2);
9  +p p2
10 $6 = (*mut i32) 0x55555559aad0
11 +x/gx 0x55555559aad0
12 0x55555559aad0:    0x0000000000000001
13 +n
```

Listing 5.19: Verify Results

5.4 Proof Of Concept 4: Use After Free Bug In Custom Struct

5.4.1 Concept

In this POC we want to expand on the knowledge gained from the previous proof of concept. This time we use a more complex object. We want to create a use-after-free bug but in this POC we use the struct User.

5.4.2 Toy Example Program

```
#[derive(Debug)]
pub struct User {
    logged: bool,
    name: String,
    password: String,
}

impl User {
    fn new(uname: &str, pass: &str) -> User {
        User {
            name: uname.to_string(),
            password: pass.to_string(),
            logged: false,
        }
    }

    pub fn log_in(&mut self, pass: String) {
        if self.logged == true {
            println!("\nAlready logged in\n");
            return;
        }

        if self.password.eq(&pass) {
            println!("\nWelcome user {}\n", self.name);
            self.logged = true;
        } else {
            println!("\nWrong password");
        }
    }
}

fn create_users() -> Box<User> {
```

```
33      println!("====In function====");
34      let mut u1 = Box::<User>::new(User::new("Antonis", "1234"));
35      let mut u2 = Box::<User>::new(User::new("Cassandra", "myfavpassword"));
36      println!("User 1: {:?}\nUser 2: {:?}", u1, u2);
37      u1.log_in("1234".to_string());
38      println!("User 1: {:?}\nUser 2: {:?}", u1, u2);
39      return u2;
40  }
41
42  fn main() {
43      let mut u2 = create_users();
44      println!("\n========Main========\nUser 2: {:?}", u2);
45      u2.log_in("myfavpassword".to_string());
46      println!("User 2: {:?}", u2);
47  }
```

The toy example is constructed as follows. We created a struct named User with three members; the name, the password, and a Boolean value that marks the user as logged in or not. In the struct, there is also a function that performs the login operation. We have three possible outcomes from this function. If the password is correct, the user logs in successfully (printing a “welcome user" message). If the password is wrong, it prints “Wrong password”, or if the user is already logged in the function prints “Already logged in”.

The goal of the toy example is to create a use-after-free bug. To do so we use the function "create_users" which constructs two users (u1, u2) and logs the first one in (u1 logs in).

The second user is returned to the caller function; in this case, the main function, where the second user also performs the login functionality.

Initial Results

```
1  ====In function====
2  User 1: User { logged: false, name: "Antonis", password: "1234" }
3  User 2: User { logged: false, name: "Cassandra", password: "myfavpassword" }
4  Welcome user Antonis
5  User 1: User { logged: true, name: "Antonis", password: "1234" }
6  User 2: User { logged: false, name: "Cassandra", password: "myfavpassword" }
7  ========Main========
8  User 2: User { logged: false, name: "Cassandra", password: "myfavpassword" }
9  Welcome user Cassandra
10 User 2: User { logged: true, name: "Cassandra", password: "myfavpassword" }
```

Listing 5.20: Initial Results For Unmodified Example

Before creating the bug, we explain the correct results. The two users Antonis and Cassandra are created. User Antonis logs in and the program prints the message “Welcome user Antonis”. The two users are printed once more just so we can monitor their state. Finally, User Cassandra is returned to the main function where she logs in as well.

After creating the use-after-free bug, we expect that instead of user Cassandra, user Antonis is returned to main, and when the user tries to log in, we should get the message “Already logged in”. Another expected outcome is the crash of the program after we create the use-after-free bug.

5.4.3 Disassembly And Course Of Action

To create the bug, we need to modify the binary. We take a look into the disassembly of the function "create_users()" before modification. We have highlighted some points of interest in the following snippet of code.

For this exploitation process, we follow the same concepts as previous POCs. Specifically, we will use line 9 to jump to the padding section which is marked with green colour. There are a couple of reasons to use that instruction, firstly modifying that instruction won’t affect the result of the program and we do not need to fix the state of the program after creating the double reference. Another reason is that we need a jump instruction that consists of 5 bytes to jump to padding code, thus replacing a regular jmp won’t do the trick. The mov instruction in line 9 reserves the exact number of bytes we want.

To understand which user is which we look at the lines 20 to 22, marked with blue colour. By observing these instructions, we understand that the destructor is called on the address stored in the stack at offset 0xc8 (line 20). And the address stored in the stack at offset 0x90 is returned to the caller function. Thus, in our case u1’s address is stored in 0xc8 offset, and u2 is stored in 0x90 offset.

```
1  Dump of assembler code for function _ZN5toy2412create_users17h19d1f2b069f7ae84E:
2     0x000055555555e270 <+0>:    sub    $0x248,%rsp
3  => 0x000055555555e277 <+7>:    lea    0x98(%rsp),%rdi
4     0x000055555555e27f <+15>:   lea    0x3c202(%rip),%rsi        # 0x55555559a488
5     ===Skip Some Instructions===
6     0x000055555555e363 <+243>: lea    0x108(%rsp),%rsi
7     0x000055555555e36b <+251>: mov    $0x38,%edx
8     0x000055555555e370 <+256>: call   0x55555555a060 <memcpy@plt>
9     0x000055555555e375 <+261>: mov    0x78(%rsp),%rax
10    0x000055555555e37a <+266>: mov    %rax,0x90(%rsp)
11    0x000055555555e382 <+274>: lea    0xc8(%rsp),%rax
12    0x000055555555e38a <+282>: mov    %rax,0x190(%rsp)
13    0x000055555555e392 <+290>: lea    0x90(%rsp),%rax
14    0x000055555555e39a <+298>: mov    %rax,0x198(%rsp)
15    ===Skip Some Instructions===
16    0x000055555555e5bd <+845>: lea    0xdd4c(%rip),%rcx         # 0x55555556c310
17    0x000055555555e5c4 <+852>: lea    0x1b8(%rsp),%rdi
```

```
18    0x000055555555e5cc <+860>: call   *%rcx
19    0x000055555555e5ce <+862>: jmp    0x55555555e5d0
20    0x000055555555e5d0 <+864>: lea    0xc8(%rsp),%rdi
21    0x000055555555e5d8 <+872>: call   0x55555555efd0 #call destructor
22    0x000055555555e5dd <+877>: mov    0x90(%rsp),%rax
23    0x000055555555e5e5 <+885>: add    $0x248,%rsp
24    0x000055555555e5ec <+892>: ret
25    0x000055555555e5ed <+893>: lea    0x90(%rsp),%rdi
26    0x000055555555e5f5 <+901>: call   0x55555555efd0 #call destructor
27    0x000055555555e5fa <+906>: lea    0xc8(%rsp),%rdi
28    0x000055555555e602 <+914>: call   0x55555555efd0 #call destructor
29    0x000055555555e607 <+919>: mov    0x218(%rsp),%rdi
30    0x000055555555e60f <+927>: call   0x55555555a050 <_Unwind_Resume@plt>
31    0x000055555555e614 <+932>: ud2
32    0x000055555555e616 <+934>: mov    %rax,%rcx
33    0x000055555555e619 <+937>: mov    %edx,%eax
34    0x000055555555e61b <+939>: mov    %rcx,0x218(%rsp)
35    0x000055555555e623 <+947>: mov    %eax,0x220(%rsp)
36    0x000055555555e62a <+954>: jmp    0x55555555e5fa
37    0x000055555555e62c <+956>: mov    %rax,%rcx
38    0x000055555555e62f <+959>: mov    %edx,%eax
39    0x000055555555e631 <+961>: mov    %rcx,0x218(%rsp)
40    0x000055555555e639 <+969>: mov    %eax,0x220(%rsp)
41    0x000055555555e640 <+976>: jmp    0x55555555e5ed
42  End of assembler dump.
```

Listing 5.21: Disassembly Before Modification

5.4.4 Patching And Bypassing

We modify the binary as shown in the following listing. Line 3 shows the modified mov command to jmp. This jmp command will jump to the padding code section of the function where the added instructions reside. In the padding code lines 10-12 we perform the following operations. Move the address of u1 (at offset 0xc8) into %rax, store it in the address of u2 (at offset 0x90), and jump back to the regular code of the function.

```
1     0x000055555555e36b <+251>: mov    $0x38,%edx
2     0x000055555555e370 <+256>: call   0x55555555a060 <memcpy@plt>
3     0x000055555555e375 <+261>: jmp    0x55555555e5ed
4     0x000055555555e37a <+266>: mov    %rax,0x90(%rsp)
5     0x000055555555e382 <+274>: lea    0xc8(%rsp),%rax
6     ===Skip Some Instructions===
7     0x000055555555e5dd <+877>: mov    0x90(%rsp),%rax
8     0x000055555555e5e5 <+885>: add    $0x248,%rsp
9     0x000055555555e5ec <+892>: ret
10    0x000055555555e5ed <+893>: mov    0xc8(%rsp),%rax
11    0x000055555555e5f5 <+901>: mov    %rax,0x90(%rsp)
12    0x000055555555e5fd <+909>: jmp    0x55555555e37a
13    0x000055555555e602 <+914>: call   0x55555555efd0
```

Listing 5.22: Disassembly After Modification

5.4.5 Results And Conclusions

When running the program we are treated with the results below.

```
1  ====In function====
2  User 1: User { logged: false, name: "Antonis", password: "1234" }
3  User 2: User { logged: false, name: "Antonis", password: "1234" }
4  Welcome user Antonis
5  User 1: User { logged: true, name: "Antonis", password: "1234" }
6  User 2: User { logged: true, name: "Antonis", password: "1234" }
7  ========Main========
8  Segmentation fault
```

Listing 5.23: Final Results

We need to address a couple of things first; the reference exchange was performed before the first print thus both pointers point to user 1 (user Antonis). When returning to the main function the program crashes with a segmentation fault. This is something we expected because of the execution of the destructor. In previous POC we mentioned that the memory state changed after executing the destructor, resulting in unexpected results.

The same thing happens in this POC after the execution of the destructor of u1 user. If we take a closer look inside a log from gdb we understand that the reference exchange was performed correctly, but after the destruction process, memory state changes. Changes in memory are marked with red colour (lines 6 and 15)

```
1  +p u1
2  $4 = (*mut toy24::User) 0x55555559eb10
3  +p u2
4  $5 = (*mut toy24::User) 0x55555559eb10
5  +x/30gx 0x55555559eb10
6  0x55555559eb10: 0x000055555559ead0  0x0000000000000007
7  0x55555559eb20: 0x0000000000000007  0x000055555559eaf0
8  0x55555559eb30: 0x0000000000000004  0x0000000000000004
9  0x55555559eb40: 0x0000000000000001  0x0000000000000021
10 0x55555559eb50: 0x72646e6173736143  0x0000000000000061
11 ....
12 +p u2
13 $6 = (*mut toy24::User) 0x55555559eb10
14 +x/30xg 0x55555559eb10
15 0x55555559eb10: 0x0000000000000000  0x000055555559e010
16 0x55555559eb20: 0x0000000000000007  0x000055555559eaf0
17 0x55555559eb30: 0x0000000000000004  0x0000000000000004
18 0x55555559eb40: 0x0000000000000001  0x0000000000000021
19 ....
```

Listing 5.24: Observing Memory State

As explained in a previous proof of concept, this does not matter in real-life situations, because when we attack this example, we would make sure to have our data in these memory addresses, using methods like Heap Feng Sui, and prevent the crash.

Just to prove our point though we bypass the destructor and rerun the program. This time we are greeted with the expected result which is the error message “Already logged

in", since user Antonis is already logged in the system.

```
====In function====
User 1: User { logged: false, name: "Antonis", password: "1234" }
User 2: User { logged: false, name: "Antonis", password: "1234" }
Welcome user Antonis
User 1: User { logged: true, name: "Antonis", password: "1234" }
User 2: User { logged: true, name: "Antonis", password: "1234" }
========Main========
User 2: User { logged: true, name: "Antonis", password: "1234" }
Already logged in
User 2: User { logged: true, name: "Antonis", password: "1234" }
```

Listing 5.25: Bypassing Destructor To Prove A Point

5.5 Proof Of Concept 5: Exploiting Lifetimes Concept

5.5.1 Concept

In this section, we exploit the lifetimes concept which was explained in the methodology chapter. We create an example, where modifying the binary creates a dangling reference. We know from previous sections that this is possible, but in this example, we want to show that explicit lifetime rules can also be exploited and used to introduce bugs on a binary level.

5.5.2 Toy Example Program

```
fn main() {
    let mut s1 = String::from("short");
    let mut s2 = String::from("longer_Str");
    let mut result;
    { //scope 1
        let mut s3 = String::from("This should not be returned");
        result = longest(&mut s1, &mut s2, &mut s3);
    }
    println!("Longest string is: {}", result);
}
fn longest<'a, 'b>(x: &'a mut str, y: &'a mut str, z: &'b mut str) -> &'a mut
 ↪  str {
    if x.len() > y.len() {
        x
    } else {
        y
    } }
```

In this case, we will try to return variable s3 instead of s2. Since s3 lives only during scope 1 it's expected to have a different lifetime, than s1 and s2. Since s1 and s2 can have the same lifetime, we can call the longest function as in the above listing and the compiler won't complain.

If we call the function replacing line 7 with the following command "result = longest (&mut s1, &mut s3, &mut s3);" we get the error below when building the program. The compiler understands that s1 and s3 have two different lifetimes and marks our code as invalid.

```
1   error[E0597]: `s3` does not live long enough
2     --> src/main.rs:54:31
3      |
4   54 |         result = longest(&s1, &s3, &s3);
5      |                               ^^^ borrowed value does not live long enough
6   55 |     }
7      |     - `s3` dropped here while still borrowed
8   56 |
9   57 |     println!("Return lifetime a variable: {}", result);
10     |                                                 ------ borrow later used here
11
12  error: aborting due to previous error; 6 warnings emitted
```

Listing 5.26: Error Message Using Wrong Lifetime Arguments

To bypass the lifetime measures, we modify the binary and return s3, thus creating a dangling pointer in the result variable. The result variable will point to s3 and s3 will be freed before the result is used in the print function.

Initial Results

```
1   Longest string is: longer_Str
```

Listing 5.27: Initial Results For Unmodified Program

5.5.3 Disassembly And Course Of Action

In this subsection, we take a look into the assembly code of the function longest().

```
1   Dump of assembler code for function _ZN9toy27_lft7longest17hdae9d195a8f59dd6E:
2   0x000055555555c970 <+0>:   sub    $0x78,%rsp
3   0x000055555555c974 <+4>:   mov    %rdi,0x10(%rsp)
4   0x000055555555c979 <+9>:   mov    %rsi,0x18(%rsp)
5   0x000055555555c97e <+14>:  mov    %rdx,0x20(%rsp)
6   0x000055555555c983 <+19>:  mov    %rcx,0x28(%rsp)
7   0x000055555555c988 <+24>:  mov    %rdi,0x48(%rsp)
8   0x000055555555c98d <+29>:  mov    %rsi,0x50(%rsp)
9   0x000055555555c992 <+34>:  mov    %rdx,0x58(%rsp)
10  0x000055555555c997 <+39>:  mov    %rcx,0x60(%rsp)
11  0x000055555555c99c <+44>:  mov    %r8,0x68(%rsp)
12  0x000055555555c9a1 <+49>:  mov    %r9,0x70(%rsp)
```

```
13     0x000055555555c9a6 <+54>:   call   0x55555555cfc0
14     0x000055555555c9ab <+59>:   mov    %rax,0x30(%rsp)
15     ===Skip Some Instructions===
16     0x000055555555c9ee <+126>: mov    0x20(%rsp),%rcx
17     0x000055555555c9f3 <+131>: mov    %rcx,0x38(%rsp)
18     0x000055555555c9f8 <+136>: mov    %rax,0x40(%rsp)
19     0x000055555555c9fd <+141>: mov    0x38(%rsp),%rax
20     0x000055555555ca02 <+146>: mov    0x40(%rsp),%rdx
21  => 0x000055555555ca07 <+151>: add    $0x78,%rsp
22     0x000055555555ca0b <+155>: ret
```

Listing 5.28: Disassembly Before Modification

The information we discover by looking at the assembly is the following

1. In green colour, we can see the destination addresses in the stack, for the z variable (its pointer at line 11 and its length at line 12).

2. In red colour we observe two store instructions to %rax and %rdx registers, this is luckily the preparation of the return values. The course of action we take is to modify these two instructions, that instead of returning what is stored in 0x38 and 0x40 they return, what is stored in the stack in the addresses 0x68 and 0x70, which contain the information about the z variable, we mentioned earlier.

5.5.4 Patching And Bypassing

The assembly code of the function after modification is shown in the listing below:

```
1      0x000055555555c970 <+0>:    sub    $0x78,%rsp
2      0x000055555555c974 <+4>:    mov    %rdi,0x10(%rsp)
3      0x000055555555c979 <+9>:    mov    %rsi,0x18(%rsp)
4      0x000055555555c97e <+14>:   mov    %rdx,0x20(%rsp)
5      0x000055555555c983 <+19>:   mov    %rcx,0x28(%rsp)
6      0x000055555555c988 <+24>:   mov    %rdi,0x48(%rsp)
7      0x000055555555c98d <+29>:   mov    %rsi,0x50(%rsp)
8      0x000055555555c992 <+34>:   mov    %rdx,0x58(%rsp)
9      0x000055555555c997 <+39>:   mov    %rcx,0x60(%rsp)
10     0x000055555555c99c <+44>:   mov    %r8,0x68(%rsp)
11     0x000055555555c9a1 <+49>:   mov    %r9,0x70(%rsp)
12     ===Skip Some Instructions===
13     0x000055555555c9e9 <+121>: mov    0x28(%rsp),%rax
14     0x000055555555c9ee <+126>: mov    0x20(%rsp),%rcx
15     0x000055555555c9f3 <+131>: mov    %rcx,0x38(%rsp)
16     0x000055555555c9f8 <+136>: mov    %rax,0x40(%rsp)
17     0x000055555555c9fd <+141>: mov    0x68(%rsp),%rax
18     0x000055555555ca02 <+146>: mov    0x70(%rsp),%rdx
19     0x000055555555ca07 <+151>: add    $0x78,%rsp
20     0x000055555555ca0b <+155>: ret
```

Listing 5.29: Disassembly After Modification

Focusing on the green parts, we modified the assembly code, so that the return values are loaded from the addresses where the metadata of variable z reside.

5.5.5 Results And Conclusions

After running the example, we can see that we created, a use-after-free bug, printing the result variable, prints bogus data.

Figure 5.1: Lifetimes Result After Modification

We can verify that this operation was successful using gdb to examine the result and z variables respectively.

```
+p z
$1 = &mut str {
  data_ptr: 0x55555559ca10,
  length: 27
}
+p y
$2 = &mut str {
  data_ptr: 0x55555559caf0,
  length: 10
}
+p x
$3 = &mut str {
  data_ptr: 0x55555559cad0,
  length: 5
}

===Skip Some Instructions===
+ni
0x000055555555c863  82  println!("Longest string is: {}", result);
+p result
$7 = &mut str {
  data_ptr: 0x55555559ca10,
  length: 27
}
```

Listing 5.30: GDB Verification

Chapter 6

Evaluation

This chapter analyses artificial or real Rust applications for buffer overflow checks. The main idea is to use a python script, and search for buffer overflow checks in binaries. The script uses radare2 [17] python binding and attempts to locate overflow checks in debug and release mode. The script focuses on instruction patterns that use 32-bit or 64-bit registers.

In the first section of this chapter, we examine the results for artificial binaries, and in the second section, we use the script to explore applications found on GitHub.

In the examples below we compare executable binaries with non-executable versions of the same binaries (object files). We refer to them as executable and non-executable versions for simplicity. The difference between them is that the executable version contains the extra code needed for the binary to run, along with code from Rust shared libraries.

6.1 Validator Details

Our validator uses heuristics, to match assembly code. During our exploration, we discovered that in debug format checks, at least one "mov" instruction precedes them (the one that prepares the index). Then the 3 instruction pattern we talked about in Chapter 4 follows which is succeeded by "jne" and "jmp" instructions. On the contrary for the release format, we cannot say for sure that a "mov" instruction precedes the "cmp" command.

To further increase the accuracy of the validator, we used another observation. This heuristic ensures that all checks printed in the log not only point to a panic exception call, but that call performs a form of bound checking. Based on the Rust panic documentation [19] the "panic_bounds_check" is called when we have out-of-bounds access to arrays or slices. Using this reasoning every check that is added to the result set, functions as a bound check for an array or a slice.

In linked binaries, radare2 produces an artefact for each panic exception call. Sim-

ilarly, it fixes an artifact to the call for panic_bound_check function. We can use that artefact to our advantage. Specifically, while searching for a check, we gather the addresses that lead to a panic error artefact dedicated for bounds checking. This process is performed approximately, by collecting the addresses of the 4 instructions preceding the artefact. Based on experience the number 4 is a good middle ground to use. At the same time, the validator searches for checks that match one of the two formats implemented. When all overflow check candidates are found, the script uses the heuristic rule we mentioned above to reduce false positive results.

Specifically, the validator filters all found checks, based on the target addresses of "jmp" or "ja" commands respectively. If the target address of "jmp" instruction for a candidate check appears in the set of addresses that lead to an overflow check, then that candidate is marked as valid and is printed in the log file. On the contrary, if the target does not appear in that set then is probably a false positive, and it is excluded from the result set.

Overall, we need to set some criteria for what we consider successful results from the script. For now, we focus on checks that use 64 or 32-bit registers and a fixed value, for both debug and release modes.

The results in the sections below present the number of debug and release format checks found in each binary. These checks follow the debug and release format explained in Chapter 3 and obey the heuristic rules as explained in this section.

Each check found that violates one of these rules is filtered out of the final result set and is not present in the log file. We present the number of checks for each binary, that follow the debug format and the number of checks that follow the release format in each binary. Some binaries are compiled in debug mode and others are compiled in release mode.

6.2 Validating Artificial Binaries

This section explores artificially constructed toy examples. This section contains two tables that present the checks found in debug and release format respectively. In the tables below we compare known checks, with checks found in the executable version of the example, and checks found in the non-executable version of each toy example. This reasoning isolates the checks that appear only in the code written by the programmer and are not products of a shared library. At this point, there is a disclaimer to be made. The checks explored in Chapter 3 are nothing more than assembly commands thus it is normal to miss some checks, especially when generalizing the matching format.

Binary	Known checks	Executable version	Non - Executable version
Toy0	2	2	2
Toy1	1	1	1
Toy1Release	0	0	0
Toy2	1	1	1
Toy2Release	0	0	0
Toy3	1	1	1
Toy3Release	0	0	0
Toy4	3	3	3
Toy5	1	1	1

Table 6.1: Results For Artificial Binaries - Debug Format

Binary	Known checks	Executable version	Non - Executable version
Toy0	0	23	0
Toy1	0	23	0
Toy1Release	1	25	1
Toy2	0	23	0
Toy2Release	1	24	1
Toy3	0	24	1
Toy3Release	1	25	1
Toy4	0	23	0
Toy5	0	23	0

Table 6.2: Results For Artificial Binaries - Release Format

Comments on Results

1. The validator managed to discover known checks for each scenario. Additional checks were found using the release format in the executable versions.

2. Non-executable results are identical to the expected results.

3. Results on the release format are the same across all the tested binaries. This is because of the common shared libraries linked in each executable.

4. Noise from false positive results reduced by using heuristics.

6.3 Validating Real Life Binaries

In this section, we use the script to investigate potential overflow checks in binaries found on GitHub. In the previous section, we compared known results, with the results found by the validator running on the executable and then running on the non-executable version. In this section, we do not have known results, we test the validator on real-life Rust applications. These applications were chosen based on their functionalities, including CSV parsers, a grep clone etc. To get proper results, we need to run the script, find which checks correspond to actual buffer checks and then perform taint analysis to discover which buffers found by the script are input dependent, and can cause security problems if modified. In the tables below we present the statistics found by the validator.

Binary	Executable version	Non - Executable version
Weld [24]	84	7
Ripgrep [5]	276	160
Xsv [6]	136	8
Dust [3]	90	1
Fblog [4]	105	0
Runiq [28]	68	0

Table 6.3: Results For Real Applications - Debug Format

Binary	Executable version	Non - Executable version
Weld [24]	41	5
Ripgrep [5]	47	9
Xsv [6]	45	19
Dust [3]	46	10
Fblog [4]	48	7
Runiq [28]	46	2

Table 6.4: Results For Real Applications - Release Format

Comments on Results

1. Some checks belong to shared libraries and are not present in the object file.
2. Higher number of checks follow the debug format than the release format.
3. False-positive checks are avoided using heuristics.

6.4 Evaluation Conclusions

Initially, the validator was tested against artificial toy examples, with known expected results. Then the validator was tested against real-life applications from GitHub. The script manages to find many different checks in both controlled and uncontrolled tests. Certainly, these checks do not always correspond to real functional or exploitable overflow checks. The same patterns used to check buffer overflows can be used as checks in many different situations. To solve this we used heuristics to make sure the result set of the validator consists of checks that guard buffers. Moreover, some buffers may not be user input dependent.

Even though we don't have a way to measure them, we know that the validator can also produce false-negative results. In other words, the validator misses checks that secure buffer bounds. This is true as the validator scans for context-dependent patterns. This means that patterns differ based on the context they are used. This can lead to many different patterns that are not yet implemented in the script, and are disregarded.

6.4.1 False Positives

The validator searches for patterns in code to find buffer overflow checks. The script was specifically implemented to search for the two varieties explored in the spatial safety chapter.

Since the validator searches for patterns in code, it's very likely to find many of these patterns present in the assembly. These patterns might not be functioning as overflow checks, instead, they might check for a very different operation. This is a normal concept in assembly, as very different high-level code operations can produce near-identical assembly commands.

To avoid false positives, we use a heuristic that checks if the "jmp" command targets a panic exception block dedicated for bound checking. Thus, for a check to be in the result log file, the check must obey one of the two formats (debug or release) and target a bound checking exception block. Even with the use of that heuristic, we cannot be certain that these buffers are user input dependent. To do that we need to perform taint analysis, and focus only on the harmful discoveries, that lead to exploits.

6.4.2 False negatives

False-negative results are when the validator missed a check entirely, and it was a functional overflow check. In the results investigated above, we don't mention false negatives, as we cannot measure them. False-negative results surely exist for our validator and in this subsection, we explain why. The validator searches for checks by matching code

patterns. At the beginning of this section, we mentioned that patterns are context-based. Many factors can change the context of the code; thus, many factors can cause false negatives.

More specifically, when a program is compiled, the compiler chooses from many different patterns to perform a check for a value. We have many instructions that are used in comparison operations and many ways to examine the flags set by the “cmp” command (e.g., test, ja, jne, je etc). Having all these commands that can be shuffled together, the spectrum of check patterns gets exponentially larger, which leads to false negatives.

Another factor is the registers used in each pattern. For example, in Chapter 3 we mainly saw patterns using the %rax register. This does not mean that all patterns use the %rax register. As discovered in this section some checks use other registers like %rdi. This problem was solved using regular expressions to include as many registers as possible.

Moreover, in our tests all the patterns contain the form of “cmp value, reg”, but in reality, this check can be performed using in the format of “cmp reg, reg” as well. Even though compilers opt for the first way whenever possible, this does not exclude the second format from being a possibility.

Finally, heuristics are based on experience. They focus on the majority of results, excluding some edge cases. An example is that we chose to use the last 4 addresses before the bounds checking artefact added by radare2. That is just an observation, and it excludes addresses that might be targeted but are not included in the set (e.g., they belong to last 5 instructions). Another scenario is that an attacker might be able to modify the "jmp" command to have a different target than the one pointing to the panic block, making the check untraceable.

6.4.3 Final Comments

All in all, the different kinds of possible patterns mentioned above, cause different and unique checks that the validator hasn’t seen before; resulting in false negatives. On the one hand, regular expressions largely assist in generalizing, matching and locating more checks, but on the other hand, if we add to the equation the different compiler optimization levels, and how each pattern can change through optimization then this pattern matching problem becomes a lot more complicated.

Chapter 7

Future Work

As seen in the previous chapter, there are some improvements to be done. The spatial safety validator should be improved in terms of accuracy and functionality. There is also a need to create a validator for temporal safety, to validate binaries and discover artificial dangling pointers and use-after-free bugs.

7.1 Spatial Safety Improvements

To further solidify our findings on real-life applications, we could perform taint analysis, to discover how many of these checks guard a buffer that is user input dependent. As stated before, our spatial safety validator suffers from accuracy problems. We could try to increase the accuracy of the validator, by adding more patterns to the pattern checker component. The validator discovers only buffer overflow checks. A good idea is to expand the validator's functionality to search for integer overflow checks as well.

7.2 Temporal Safety Improvements

Constructing a mechanism for validating temporal safety is not as straightforward as spatial safety. The borrow checker mechanism of the Rust compiler is performed at compile time, and no signs of the borrow checker, survive the compilation phase. A good idea to validate the temporal safety of Rust binaries would be to lift the Rust binary into Intermediate Representation code (IR) since rustc uses LLVM. This means there is a good chance that the borrow checker is a library of passes on the LLVM IR. Based on "Guide to Rust development" [20] the borrow checker runs on Middle-level IR (MIR) which is a control flow graph used for borrow checking, checking uninitialized variables and performing optimizations. Even if we cannot recreate the MIR, there is a good chance we can emulate the functionality of the borrow checker, on the lifted IR.

Chapter 8

Related Work

Memory safety bugs in unsafe languages are one of the most important and old problems faced by security systems. The attacker can exploit the bug and take control of the program. This problem exists for many years now and despite the countermeasures provided, by software hardening, all currently deployed countermeasures can be defeated [26].

Unsafe systems like C/C++ have hardening techniques. For example, stack canaries contribute to stack protection. Also, in state-of-the-art compilers like Clang, we have SafeStack, which prevents data corruption by stack overflows [13, 15]. While SafeStack is based on information hiding by using a safe and an unsafe stack, some interesting work is conducted so systems are hardened without the need of hiding. Instead, commodity hardware is used to protect memory regions [14].

Papers also, state that bounds checking can also be done with lower overhead [9, 22]. Unsafe systems suffer from temporal safety bugs, as much as they suffer from spatial safety ones. Research to enforce temporal safety is conducted, along with attempts to try and pinpoint temporal safety violations in C/C++ [16, 23, 29].

This thesis does not offer new ways of detecting spatial or temporal safety violations, nor do we optimize existing ones. Instead, we explore the safe code of Rust and use the reasoning that this safe environment can be modified on a binary level to reintroduce bugs. We also offer a script which discovers the compiler's checks that guard buffers.

The idea of artificially unsafe binaries in a safe app store was successfully implemented, by Jekyll applications, in the iOS store [27]. These applications contain remotely exploitable vulnerabilities, which lead to eligible unsafe binaries added to the store.

Rust offers the ability to integrate safe code written only in Rust, with unsafe code written in C/C++. These executables are called mixed binaries. Interesting works show how these binaries can be exploited, based on the fact that Rust and C/C++ code share the same address space, but each one enforces different rules on their memory model [18]. This thesis relies on the fact that the binary contains only safe code emitted by the Rust compiler and after modification, the binary becomes unsafe.

Chapter 9

Conclusion

This thesis investigated the safety of binaries written in Rust language. Throughout Chapters 4 and 5 we explored scenarios exploiting spatial and temporal safety in Rust binaries. We validated our reasoning for modifying and introducing bugs at the binary level. Our attempts were successful since Rust binaries do not have run-time support and are only checked at compile time.

We exploited spatial safety, by removing buffer overflow and integer overflow checks added by the compiler. As proven in this thesis this leads to over-read or over-write bugs, that can be used to perform exploitations. Checks added in debug and release mode differ and we need to address them individually.

A similar logic was followed for temporal safety exploitation. More specifically we tried to bypass concepts of Rust that enforce temporal safety. Such concepts are Ownership, Mutable References, Lifetimes and Borrowing. These concepts provide memory safety guarantees and prevent bugs like dangling pointers, use-after-free etc. We proved that we could reintroduce such bugs on a binary level after compilation by manipulating different binaries.

Furthermore, we created a light validator for locating buffer overflow checks and patching them to a new offset (this feature is implemented only for the “cmp value, reg” format of the check). As seen in Chapter 6 the validator was tested against artificial scenarios and real-life applications. In Chapters 6 and 7 we stated the weaknesses of the validator, and as discussed there is still plenty of work to perfect the validator. Lastly, a potential way of building a validator for temporal safety was provided in Chapter 7.

Bibliography

[1] Apple. Swift.org. https://docs.swift.org/swift-book/.

[2] Apple. Programming with objective-c. https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithObjects/WorkingwithObjects.html, Sep 2014.

[3] Bootandy. Bootandy/dust: A more intuitive version of du in rust. https://github.com/bootandy/dust.

[4] Brocode. Brocode/fblog: Small command-line json log viewer. https://github.com/brocode/fblog.

[5] BurntSushi. Burntsushi/ripgrep: Ripgrep recursively searches directories for a regex pattern while respecting your gitignore. https://github.com/BurntSushi/ripgrep.

[6] BurntSushi. Burntsushi/xsv: A fast csv command line toolkit written in rust. https://github.com/BurntSushi/xsv.

[7] cwe. Common weakness enumeration. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html, 2021.

[8] Cyrus-And. Cyrus-and/gdb-dashboard: Modular visual interface for gdb in python. https://github.com/cyrus-and/gdb-dashboard.

[9] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for ... - irisa.fr. https://www.irisa.fr/lande/lande/icse-proceedings/icse/p162.pdf.

[10] T. G. Foundation. gdb. https://www.gnu.org/software/gdb/.

[11] T. R. Foundation. The rust programming language. https://doc.rust-lang.org/book/.

[12] T. R. Foundation. rustprod. https://www.rust-lang.org/production.

[13] L. D. Group. Clang 15.0.0git documentation. https://clang.llvm.org/docs/SafeStack.html.

[14] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware - vusec. https://download.vusec.net/papers/memsentry_eurosys17.pdf.

[15] S. Nagarakatte, J. Zhao, M. Martin, and S. Zdancewic. Highly compatible and complete spatial memory safety for c. https://repository.upenn.edu/cgi/viewcontent.cgi?article=1941&context=cis_reports, Jan 2009.

[16] S. Nagarakatte, J. Zhao, M. Martin, and S. Zdancewic. Cets: Compiler-enforced temporal safety for c. https://acg.cis.upenn.edu/papers/ismm10_cets.pdf, Jan 2010.

[17] S. A. (pancake) and the community. Radare org. https://github.com/radareorg.

[18] M. Papaevripides and E. Athanasopoulos. Exploiting mixed binaries - cs.ucy.ac.cy. http://www.cs.ucy.ac.cy/~elathan/papers/tops21.pdf, Dec 2020.

[19] Rust-Lang. https://doc.rust-lang.org/src/core/panicking.rs.html.

[20] Rust-Lang. Guide to rustc development. https://rustc-dev-guide.rust-lang.org/borrow_check.html?highlight=borrow#mir-borrow-check.

[21] Rust-Lang. Rfcs/0560-integer-overflow.md at master · rust-lang/rfcs. https://github.com/rust-lang/rfcs/blob/master/text/0560-integer-overflow.md.

[22] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow ... - stanford university. https://suif.stanford.edu/papers/tunji04.pdf.

[23] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos. Vtpin: Practical vtable hijacking protection for binaries. http://static.cs.brown.edu/people/vpk/papers/vtpin.acsac16.pdf.

[24] Serayuzgur. Serayuzgur/weld: Full fake rest api generator written with rust. https://github.com/serayuzgur/weld.

[25] StackOveflow. Stack overflow survey. https://insights.stackoverflow.com/survey/2020.

[26] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory - university of california ... https://people.eecs.berkeley.edu/~dawnsong/papers/Oakland13-SoK-CR.pdf, 2013.

[27] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on IOS: When benign apps become evil. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/wang_tielei, Aug 2013.

[28] Whitfin. Whitfin/runiq: An efficient way to filter duplicate lines from input, à la uniq. https://github.com/whitfin/runiq.

[29] S. H. Yong and S. Horwitz. Protecting c programs from attacks via invalid pointer ... https://www.cse.psu.edu/~trj1/cse597-s11/docs/horowitz_c_pointers.pdf.