FIRST EDITION – CHAPTER 12 REV 1

Kevin Thomas

Forward

This book is not associated or endorsed by the Rust Foundation. This is purely a FREE educational resource going step-by-step on how to build and reverse engineer Rust binaries.

Rust redefines how a binary is compiled. It does not operate with similar patterns like we see in older languages and is simply the most difficult language to reverse engineer in history.

Many of the C-style decompilers within the popular reversing tools are literally rendered useless with Rust.

Malware Authors are going the route of Rust as there are few tools that exist yet to properly statically analyze the compiled binaries which coupled with its speed and memory safety, it is very difficult to understand what it is actually doing.

The aim of this book is to teach basic Rust and step-by-step reverse engineer each simple binary to understand what is going on under the hood.

We will develop within the Windows architecture (Intel x64 CISC) as most malware targets this platform by orders of magnitude.

Let’s begin...

Table Of Contents

Chapter 1: Hello Rust

We begin our journey with developing a simple hello world program in Rust on a Windows 64-bit OS.

We will then reverse engineer the binary in IDA Free.

Let’s first download Rust for Windows.

https://www.rust-lang.org/tools/install

Let’s download IDA Free.

https://hex-rays.com/ida-free/#download

Let’s download Visual Studio Code which we will use as our integrated development environment.

https://code.visualstudio.com/

Once installed, let’s add the Rust extension within VS Code.

https://marketplace.visualstudio.com/items?itemName=rust-lang.rust-analyzer

It is important to update Rust so I would encourage you to run this before every project.

```
rustup update
```

Let’s create a new project and get started by following the below steps.

```
cargo new one_hello
```

Now let’s populate our **main.rs** file with the following.

```
fn main() {
    println!("Hello, world!");
}
```

Let’s open up the terminal by click CTRL+SHIFT+` and type the following.

```
cargo build
```

Let’s run the binary!

```
.\target\debug\one_hello.exe
```

Output…

```
Hello, world!
```

Congratulations! You just created your first hello world code in Rust. Time for cake!

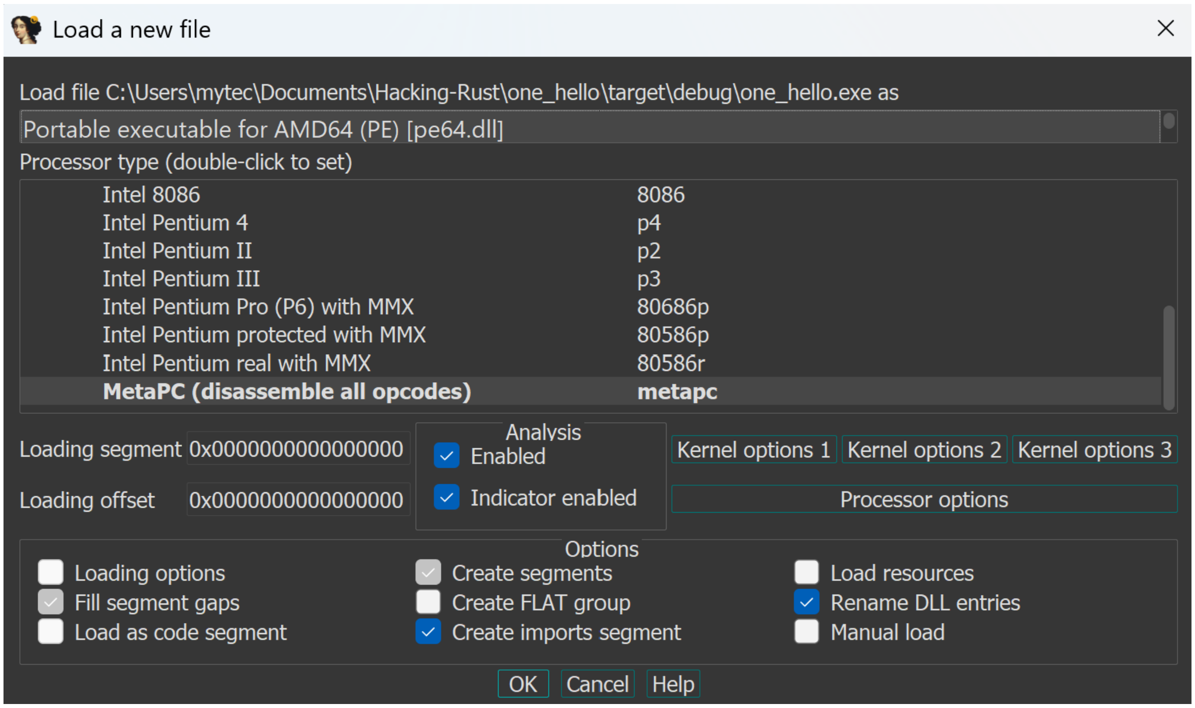
We simply created a hello world style example to get us started.

In our next lesson we will debug this in IDA Free!

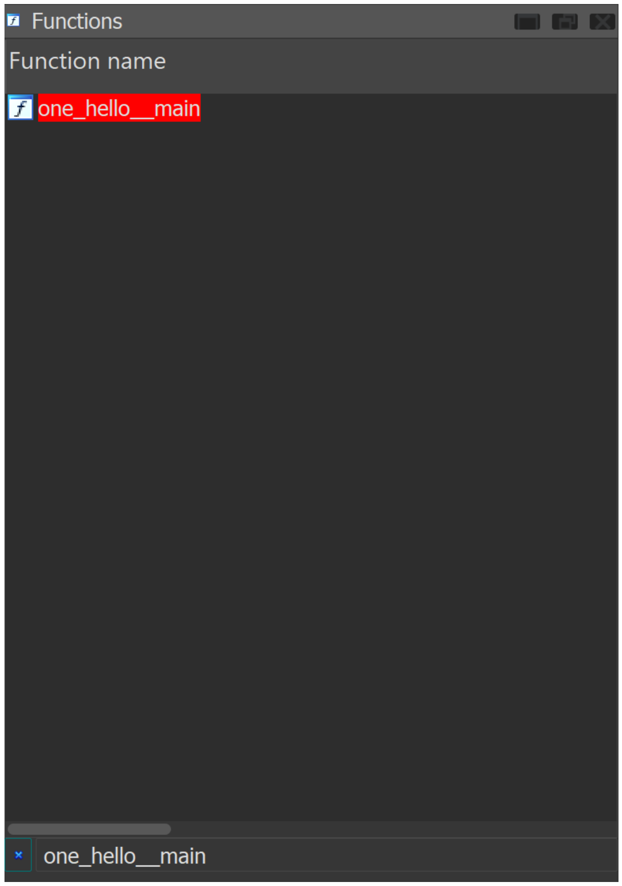
Chapter 2: Debugging Hello Rust

Let’s debug our app within IDA Free.

Open IDA Free and we see the load screen. We can keep all the defaults and simply click OK.

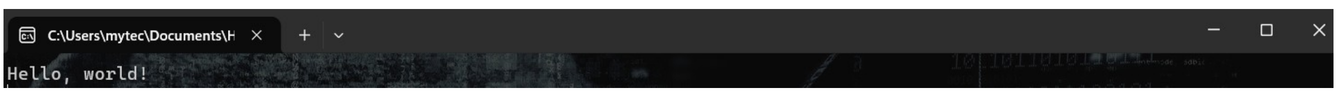


In Rust at the assembler level we will need to search for the entry point of our app. This is the *one_hello__main* function. You can use CTRL+F to search.

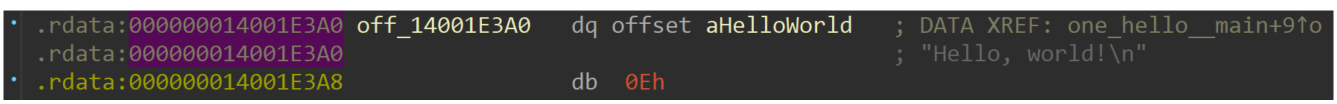


Now we can double-click on the one_hello__main to launch the focus to this function and graph.

We can see in the terminal our *"Hello, world!\n"* text.



If we double-click on *off_14001E3A0* it will take us to a new window where the string lives within the binary.



Similar to Go, we can see the strings are in a string pool however we do see a 0Ah, 0 so it contains the newline and null terminator as Go handles it slightly different.

These lessons are designed to be short and digestible so that you can code and hack along.

In our next lesson we will learn how to hack this string and force the binary to print something else to the terminal of our choosing.

This will give us the first taste on hacking Rust!

Chapter 3: Hacking Hello Rust

Let’s hack our app with IDA Free.

Let’s load up IDA and revisit the binary.

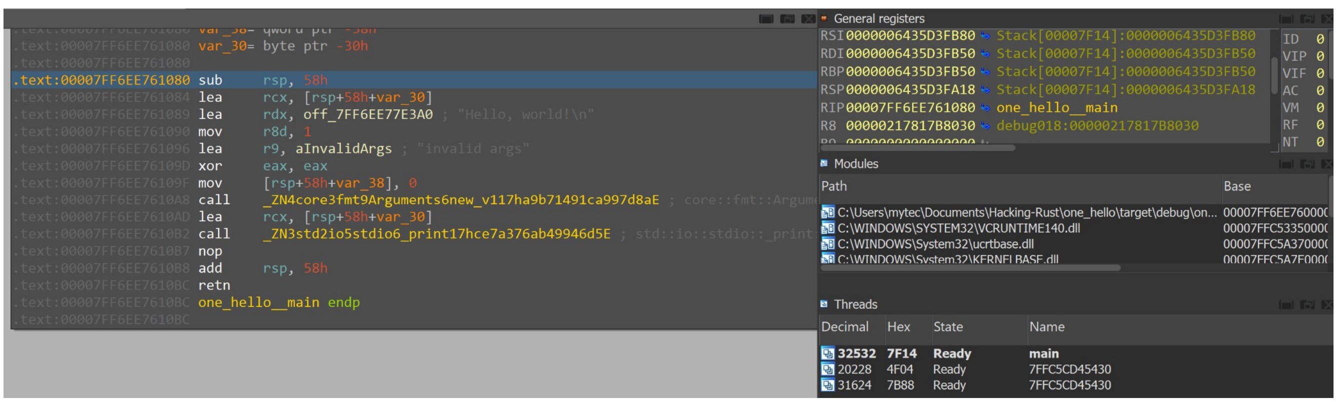
```
; void __fastcall one_hello::main()
one_hello__main proc near

var_38= qword ptr -38h
var_30= byte ptr -30h

sub     rsp, 58h
lea     rcx, [rsp+58h+var_30]
lea     rdx, off_14001E3A0 ; "Hello, world!\n"
mov     r8d, 1
lea     r9, aInvalidArgs ; "invalid args"
xor     eax, eax
mov     [rsp+58h+var_38], 0
call    _ZN4core3fmt9Arguments6new_v117ha9b71491ca997d8aE ; core::fmt::Arguments::new_v1::ha9b71491ca997d8a
lea     rcx, [rsp+58h+var_30]
call    _ZN3std2io5stdio6_print17hce7a376ab49946d5E ; std::io::stdio::_print::hce7a376ab49946d5
nop
add     rsp, 58h
retn
one_hello__main endp
```

We start off with subtracting *0x58* bytes from *RSP*. We do not see the C-style prologue here. What we are seeing is what the Rust compiler chose to do at compile time.

This is a VERY simple example but I want to take the time to step through this step-by-step.

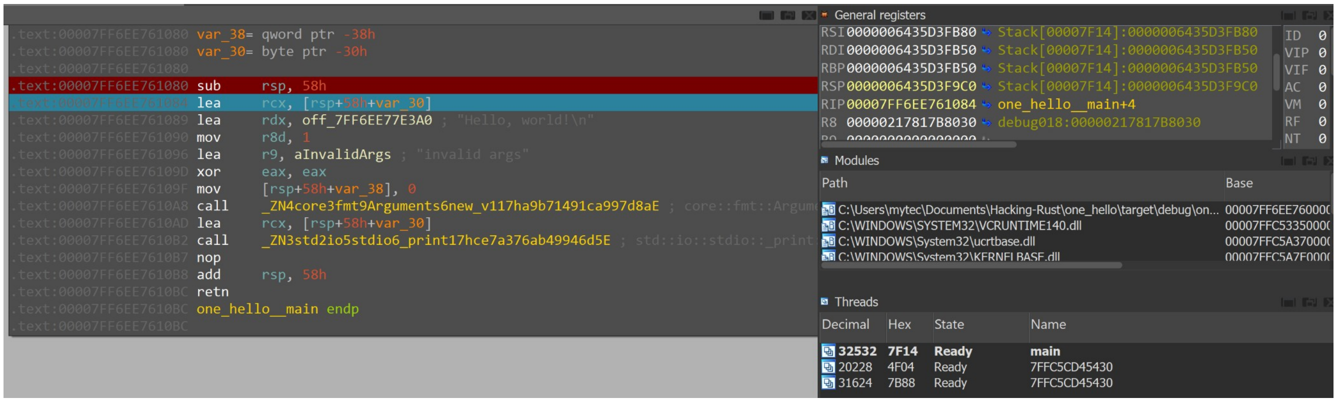
Let’s set a breakpoint at the entry.

Lets run.



Lets take note where *RSP* is *0x0000006435d3fa18*.

Lets step once.



We now see the value of *RSP* at *0x0000006435df9c0*. This makes sense as *0x0000006435d3fa18 - 0x58 = 0x0000006435d3f9c0*.

Next we see *lea rcx, [rsp+58h+var_30]* so let's break this down.

LEA stands for "Load Effective Address." It is a versatile instruction that can be used to perform arithmetic calculations on addresses without actually accessing memory.

RCX is the destination register where the effective address will be stored.

[rsp+58h+var_30] is the source operand, representing the address that needs to be calculated. It consists of multiple parts:
* *RSP* is the stack pointer register.
* *58h* is a hexadecimal constant value, which represents an offset from the stack pointer.
* *var_30* refers to a variable or memory location. In this case, it is *-30h*, indicating a byte-sized variable or memory location located at an offset of *-30h* (or *-48* in decimal) from the stack pointer.

Therefore, *lea rcx, [rsp+58h+var_30]* calculates the effective address by adding the stack pointer (*RSP*) with an offset of *58h* and the variable *var_30* located at an offset of *-30h* from the stack pointer. The resulting effective address is stored in the *RCX* register.

Next we see an offset value being loaded into *RDX*.

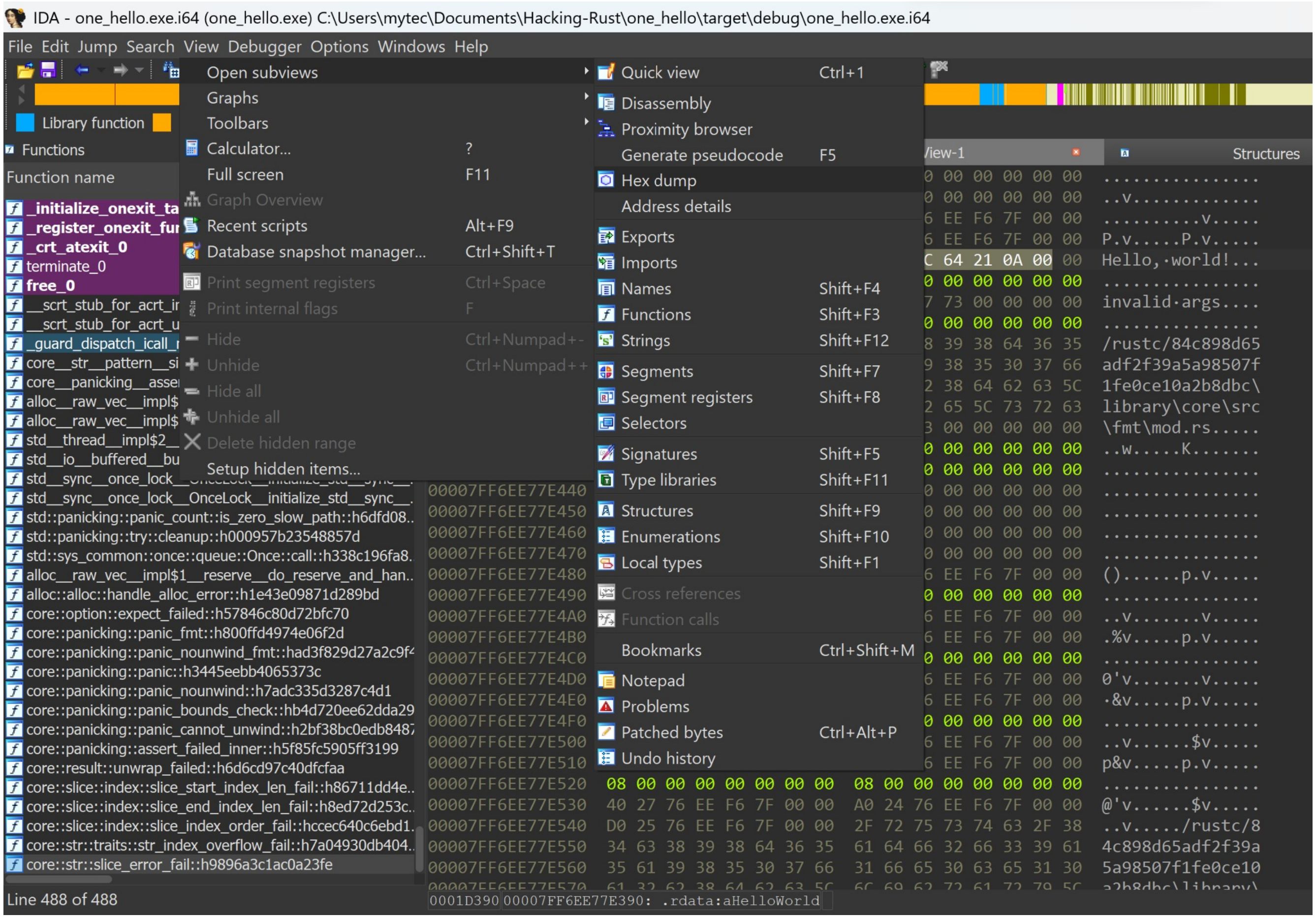
```
.text:00007FF6EE761080
.text:00007FF6EE761080
.text:00007FF6EE761080
.text:00007FF6EE761080 ; void __fastcall one_hello::main()
.text:00007FF6EE761080 one_hello__main proc near
.text:00007FF6EE761080
.text:00007FF6EE761080 var_38= qword ptr -38h
.text:00007FF6EE761080 var_30= byte ptr -30h
.text:00007FF6EE761080
.text:00007FF6EE761080 sub     rsp, 58h
.text:00007FF6EE761084 lea     rcx, [rsp+58h+var_30]
.text:00007FF6EE761089 lea     rdx, off_7FF6EE77E3A0 ; "Hello, world!\n"
.text:00007FF6EE761090 mov     r8d, 1
.text:00007FF6EE761096 lea     r9, aInvalidArgs ; "invalid args"
.text:00007FF6EE76109D xor     eax, eax
.text:00007FF6EE76109F mov     [rsp+58h+var_38], 0
.text:00007FF6EE7610A8 call    _ZN4core3fmt9Arguments6new_v117ha9b71491ca997d8aE ; core::fmt::Argu
.text:00007FF6EE7610AD lea     rcx, [rsp+58h+var_30]
.text:00007FF6EE7610B2 call    _ZN3std2io5stdio6_print17hce7a376ab49946d5E ; std::io::stdio::_prin
.text:00007FF6EE7610B7 nop
.text:00007FF6EE7610B8 add     rsp, 58h
.text:00007FF6EE7610BC retn
.text:00007FF6EE7610BC one_hello__main endp
```

This is our string. Let's double-click on the offset.

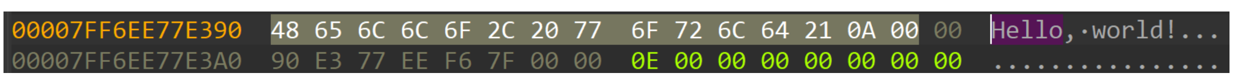
```
.rdata:00007FF6EE77E3A0 off_7FF6EE77E3A0 dq offset aHelloWorld  ; DATA XREF: one_hello__main+9↑o
.rdata:00007FF6EE77E3A0                                         ; "Hello, world!\n"
.rdata:00007FF6EE77E3A8 db  0Eh
```

We see 13 chars and the null terminator therefore *0x0e*.

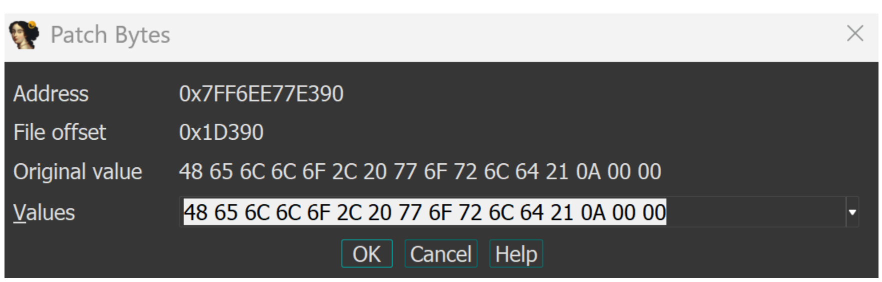
Let's stop the debug and click on the *"Hello, world!\n"* text and view the *Hex View-1*.



Here we see the hex view.

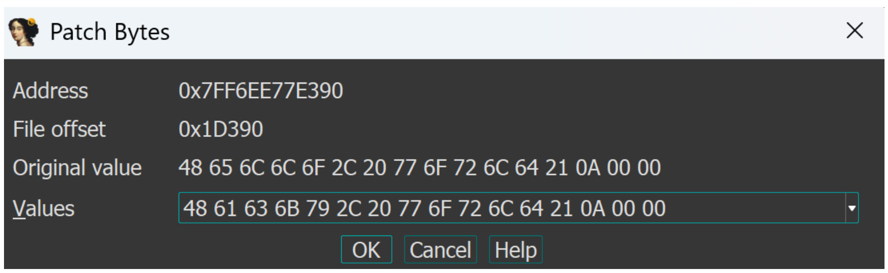


Click edit – Patch program – Change byte…

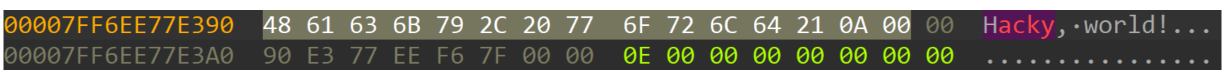


Let’s hack our binary to say “Hacky, world!\n” instead!

```
'H' -> 0x48
'a' -> 0x61
'c' -> 0x63
'k' -> 0x6B
'y' -> 0x79
',' -> 0x2C
' ' -> 0x20
'w' -> 0x77
'o' -> 0x6F
'r' -> 0x72
'l' -> 0x6C
'd' -> 0x64
'!' -> 0x21
```



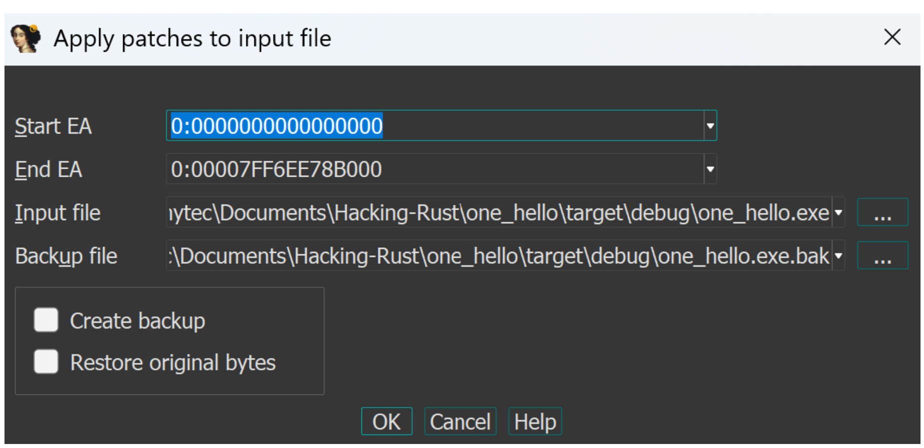
Click OK.



SUCCESS!

Press the red stop button to stop debugging.

Click edit – Patch program – Apply patches to input file…



Let’s run again…

Let’s F8, step over, until the *NOP*.

Observe the command window.

SUCCESS! We have successfully hacked our first Rust binary!

The rest of the Assembler is trivial as we are just calling std::io::stdio::_print to echo the line to STDOUT.

Stay tuned as this will get significantly more challenging. With all things we start small and build!

In our next chapter we will discuss the Scalar Data Types.

Chapter 4: Scalar Data Types

We continue our journey with a scalar data types example program in Rust on a Windows 64-bit OS.

Let’s create a new project and get started by following the below steps.

```
cargo new two_scalar-data-types
```

Now let’s populate our **main.rs** file with the following.

```
fn main() {
    let my_integer: i32 = 42;
    println!("integer: {}", my_integer);

    let my_float: f64 = 3.14;
    println!("float: {}", my_float);

    let my_char: char = 'A';
    println!("character: {}", my_char);
}
```

Let’s open up the terminal by click CTRL+SHIFT+` and type the following.

```
cargo build
```

Let’s run the binary!

```
.\target\debug\two_scalar-data-types.exe
```

Output…

```
integer: 42
float: 3.14
character: A
```

Congratulations! You just created another program in Rust. Time for cake!

We simply created an example of each of the scalar data types which are int, float and char.

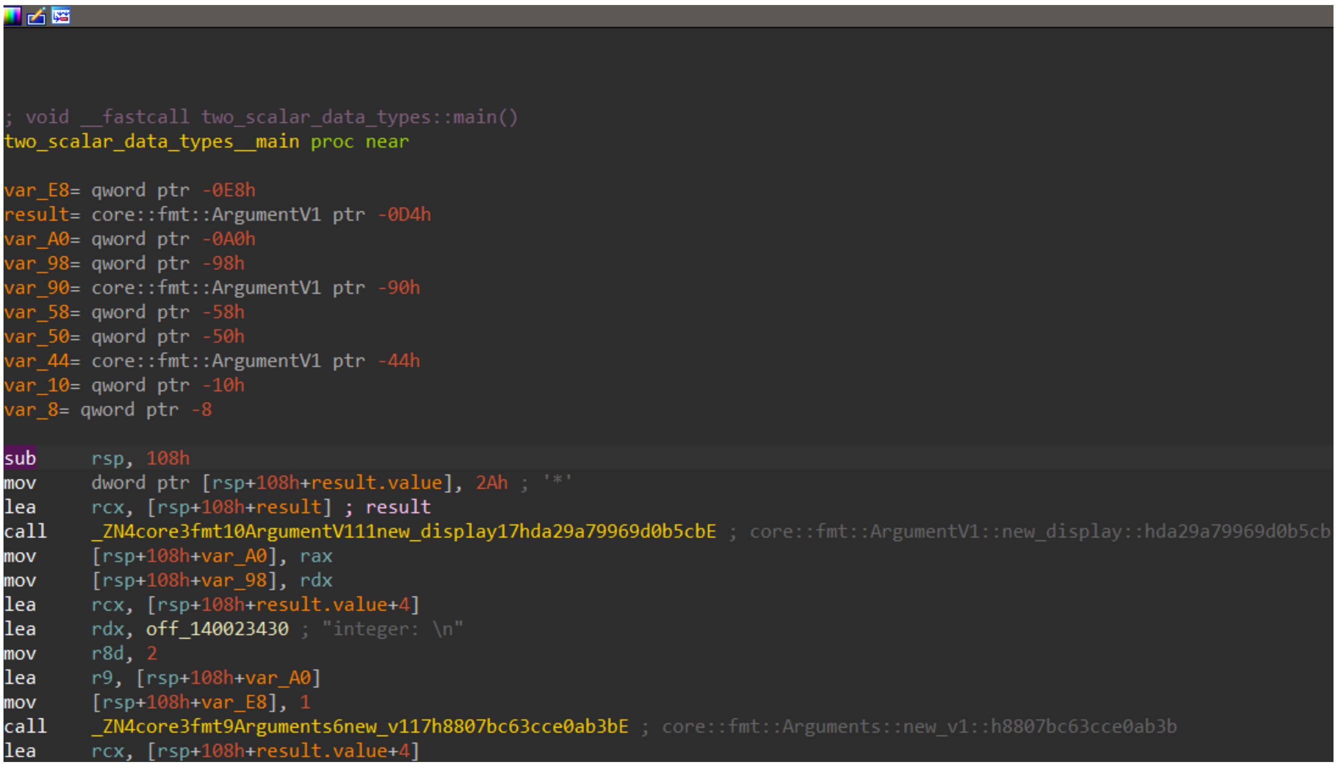
In our next lesson we will debug this in IDA Free!

Chapter 5: Debugging Scalar Data Types

Let’s debug our app within IDA Free.

Open IDA Free and we see the load screen. We can keep all the defaults and simply click OK.

On the left-hand side we see the Function name window and we want to search for *two_scalar_data_types__main*. Double-click it and we will see the function.



```
; void __fastcall two_scalar_data_types::main()
two_scalar_data_types__main proc near

var_E8= qword ptr -0E8h
result= core::fmt::ArgumentV1 ptr -0D4h
var_A0= qword ptr -0A0h
var_98= qword ptr -98h
var_90= core::fmt::ArgumentV1 ptr -90h
var_58= qword ptr -58h
var_50= qword ptr -50h
var_44= core::fmt::ArgumentV1 ptr -44h
var_10= qword ptr -10h
var_8= qword ptr -8

sub     rsp, 108h
mov     dword ptr [rsp+108h+result.value], 2Ah ; '*'
lea     rcx, [rsp+108h+result] ; result
call    _ZN4core3fmt10ArgumentV111new_display17hda29a79969d0b5cbE ; core::fmt::ArgumentV1::new_display::hda29a79969d0b5cb
mov     [rsp+108h+var_A0], rax
mov     [rsp+108h+var_98], rdx
lea     rcx, [rsp+108h+result.value+4]
lea     rdx, off_140023430 ; "integer: \n"
mov     r8d, 2
lea     r9, [rsp+108h+var_A0]
mov     [rsp+108h+var_E8], 1
call    _ZN4core3fmt9Arguments6new_v117h8807bc63cce0ab3bE ; core::fmt::Arguments::new_v1::h8807bc63cce0ab3b
lea     rcx, [rsp+108h+result.value+4]
```

```
call    _ZN4core3fmt10ArgumentV111new_display17h0ac0f3d1bf4c49e3E ; core::fmt::ArgumentV1::new_display::h0ac0f3d1bf4c49e3
mov     [rsp+108h+var_58], rax
mov     [rsp+108h+var_50], rdx
lea     rcx, [rsp+108h+var_90.formatter]
lea     rdx, off_140023458 ; "float: "
mov     r8d, 2
lea     r9, [rsp+108h+var_58]
mov     [rsp+108h+var_E8], 1
call    _ZN4core3fmt9Arguments6new_v117h8807bc63cce0ab3bE ; core::fmt::Arguments::new_v1::h8807bc63cce0ab3b
lea     rcx, [rsp+108h+var_90.formatter]
call    _ZN3std2io5stdio6_print17hce7a376ab49946d5E ; std::io::stdio::_print::hce7a376ab49946d5
mov     dword ptr [rsp+108h+var_44.value], 41h ; 'A'
lea     rcx, [rsp+108h+var_44] ; result
call    _ZN4core3fmt10ArgumentV111new_display17h2cdf30f7c1587727E ; core::fmt::ArgumentV1::new_display::h2cdf30f7c1587727
mov     [rsp+108h+var_10], rax
mov     [rsp+108h+var_8], rdx
lea     rcx, [rsp+108h+var_44.value+4]
lea     rdx, off_140023488 ; "character: "
mov     r8d, 2
lea     r9, [rsp+108h+var_10]
mov     [rsp+108h+var_E8], 1
call    _ZN4core3fmt9Arguments6new_v117h8807bc63cce0ab3bE ; core::fmt::Arguments::new_v1::h8807bc63cce0ab3b
lea     rcx, [rsp+108h+var_44.value+4]
call    _ZN3std2io5stdio6_print17hce7a376ab49946d5E ; std::io::stdio::_print::hce7a376ab49946d5
nop
add     rsp, 108h
retn
two_scalar_data_types__main endp
```

We see the below instruction.

```
mov     dword ptr [rsp+108h+result.value], 2Ah ; '*'
```

The value of *2Ah* is *42* so we know that the literal value is being stored inside *RSP* + the offset value above.

We then see a call to the **println!** macro.

```
call    _ZN3std2io5stdio6_print17hce7a376ab49946d5E ; std::io::stdio::_print::hce7a376ab49946d5
```

Bare with me as I explain the floating point number. Let's first review the actual code we are going to statically reverse.

```
    let my_float: f64 = 3.14;
    println!("float: {}", my_float);
```

The first instruction loads the floating-point value *3.14* from memory into the XMM0 register. The XMM registers are 128-bit registers that are used to store floating-point values and vectors.

```
movsd   xmm0, cs:__real@40091eb851eb851f
```

The next instruction stores the value in XMM0 to the memory location rsp+108h+var_90.value. This memory location is part of the stack frame that is allocated for the *println!* macro call.

```
movsd   [rsp+108h+var_90.value], xmm0
```

The next instruction sets the register *rcx* to point to the memory location *rsp+108h+var_90*. This memory location contains the *ArgumentV1* struct that will be passed to the new_display function.

```
lea     rcx, [rsp+108h+var_90] ; result
```

The next instruction calls the new_display function. The *new_display* function is responsible for converting the *ArgumentV1* struct into a string representation.

```
call    _ZN4core3fmt10ArgumentV111new_display17h0ac0f3d1bf4c49e3E ; core::fmt::ArgumentV1::new_display::h0ac0f3d1bf4c49e3
```

The *new_display* function returns the string representation in the *RAX* and *RDX* registers. The next instruction stores the value in *RAX* to the memory location *rsp+108h+var_58*. This memory location contains the *Formatter* struct that will be passed to the *println!* macro.

```
mov     [rsp+108h+var_58], rax
```

The next instruction stores the value in *RDX* to the memory location *rsp+108h+var_50*. This memory location contains the slice of bytes that represents the string representation of the *ArgumentV1* struct.

```
mov     [rsp+108h+var_50], rdx
```

The next instruction sets the register *RCX* to point to the memory location *rsp+108h+var_90.formatter*. This memory location contains the *Formatter* struct that will be passed to the *println!* macro.

```
lea     rcx, [rsp+108h+var_90.formatter]
```

The next instruction sets the register *RDX* to point to the string literal "*float:* ".

```
lea     rdx, off_140023458 ; "float: "
```

Finally we see the *println!* macro print the value.

```
call    _ZN3std2io5stdio6_print17hce7a376ab49946d5E ; std::io::stdio::_print::hce7a376ab49946d5
```

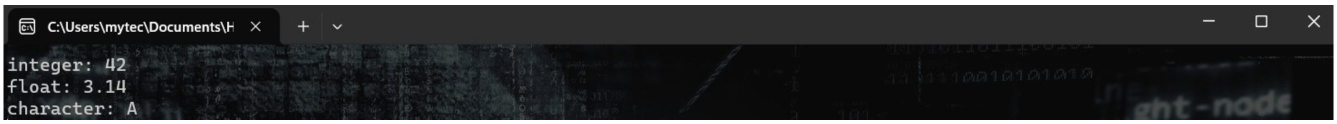
The last item was the char ‘A’ which is simply moved into RSP and an offset below.

```
mov     dword ptr [rsp+108h+var_44.value], 41h ; 'A'
```

Then we print.

```
call    _ZN3std2io5stdio6_print17hce7a376ab49946d5E ; std::io::stdio::_print::hce7a376ab49946d5
```

Below are the results in the terminal.



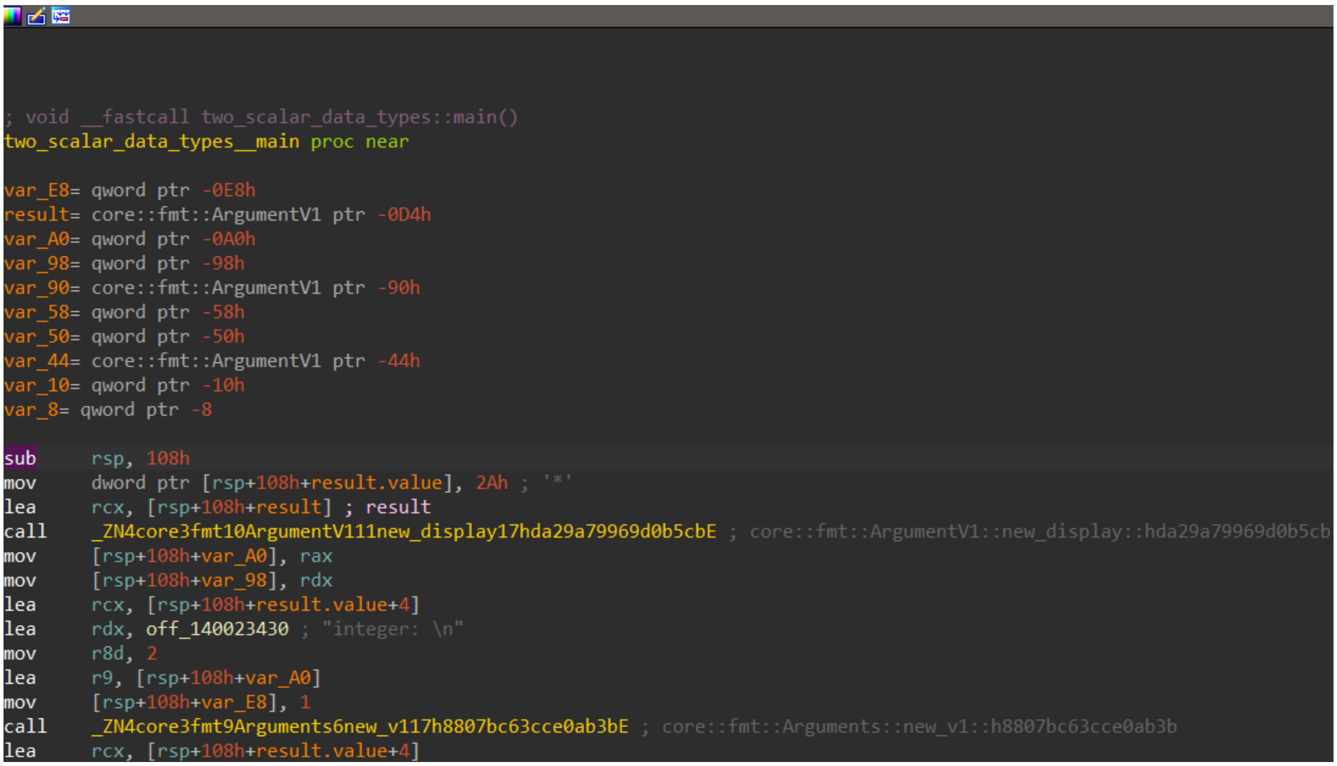
In our next lesson we will hack this!

Chapter 6: Hacking Scalar Data Types

Let’s hack our app within IDA Free.

Open IDA Free and we see the load screen. We can keep all the defaults and simply click OK.

On the left-hand side we see the Function name window and we want to search for *two_scalar_data_types__main*. Double-click it and we will see the function.



```
; void __fastcall two_scalar_data_types::main()
two_scalar_data_types__main proc near

var_E8= qword ptr -0E8h
result= core::fmt::ArgumentV1 ptr -0D4h
var_A0= qword ptr -0A0h
var_98= qword ptr -98h
var_90= core::fmt::ArgumentV1 ptr -90h
var_58= qword ptr -58h
var_50= qword ptr -50h
var_44= core::fmt::ArgumentV1 ptr -44h
var_10= qword ptr -10h
var_8= qword ptr -8

sub     rsp, 108h
mov     dword ptr [rsp+108h+result.value], 2Ah ; '*'
lea     rcx, [rsp+108h+result] ; result
call    _ZN4core3fmt10ArgumentV111new_display17hda29a79969d0b5cbE ; core::fmt::ArgumentV1::new_display::hda29a79969d0b5cb
mov     [rsp+108h+var_A0], rax
mov     [rsp+108h+var_98], rdx
lea     rcx, [rsp+108h+result.value+4]
lea     rdx, off_140023430 ; "integer: \n"
mov     r8d, 2
lea     r9, [rsp+108h+var_A0]
mov     [rsp+108h+var_E8], 1
call    _ZN4core3fmt9Arguments6new_v117h8807bc63cce0ab3bE ; core::fmt::Arguments::new_v1::h8807bc63cce0ab3b
lea     rcx, [rsp+108h+result.value+4]
```

```
call    _ZN4core3fmt10ArgumentV111new_display17h0ac0f3d1bf4c49e3E ; core::fmt::ArgumentV1::new_display::h0ac0f3d1bf4c49e3
mov     [rsp+108h+var_58], rax
mov     [rsp+108h+var_50], rdx
lea     rcx, [rsp+108h+var_90.formatter]
lea     rdx, off_140023458 ; "float: "
mov     r8d, 2
lea     r9, [rsp+108h+var_58]
mov     [rsp+108h+var_E8], 1
call    _ZN4core3fmt9Arguments6new_v117h8807bc63cce0ab3bE ; core::fmt::Arguments::new_v1::h8807bc63cce0ab3b
lea     rcx, [rsp+108h+var_90.formatter]
call    _ZN3std2io5stdio6_print17hce7a376ab49946d5E ; std::io::stdio::_print::hce7a376ab49946d5
mov     dword ptr [rsp+108h+var_44.value], 41h ; 'A'
lea     rcx, [rsp+108h+var_44] ; result
call    _ZN4core3fmt10ArgumentV111new_display17h2cdf30f7c1587727E ; core::fmt::ArgumentV1::new_display::h2cdf30f7c1587727
mov     [rsp+108h+var_10], rax
mov     [rsp+108h+var_8], rdx
lea     rcx, [rsp+108h+var_44.value+4]
lea     rdx, off_140023488 ; "character: "
mov     r8d, 2
lea     r9, [rsp+108h+var_10]
mov     [rsp+108h+var_E8], 1
call    _ZN4core3fmt9Arguments6new_v117h8807bc63cce0ab3bE ; core::fmt::Arguments::new_v1::h8807bc63cce0ab3b
lea     rcx, [rsp+108h+var_44.value+4]
call    _ZN3std2io5stdio6_print17hce7a376ab49946d5E ; std::io::stdio::_print::hce7a376ab49946d5
nop
add     rsp, 108h
retn
two_scalar_data_types__main endp
```

We see the below instruction.

```
mov     dword ptr [rsp+108h+result.value], 2Ah ; '*'
```

The value of *2Ah* is *42* so we know that the literal value is being stored inside *RSP* + the offset value above.

Let's set a breakpoint by highlighting the line and pressing F2.

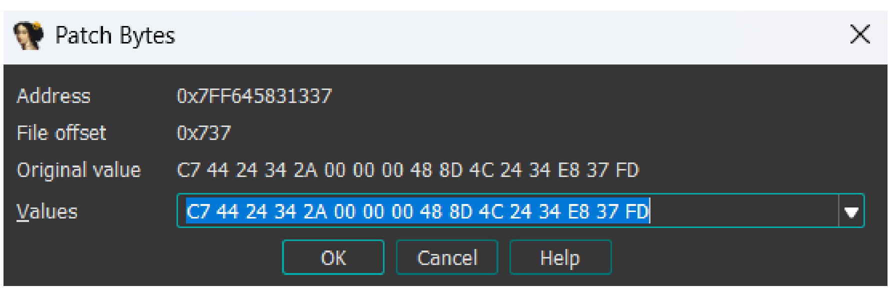
```
mov     dword ptr [rsp+108h+result.value], 2Ah ; '*'
```

Press play to debug.

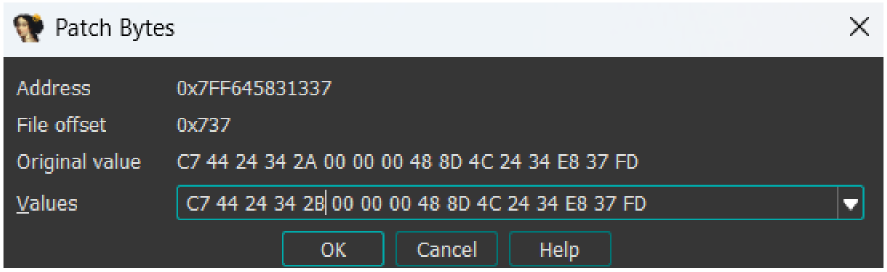
We now see the our breakpoint triggered.

```
.text:00007FF645831337 mov     dword ptr [rsp+108h+result.value], 2Ah
```

Click edit – Patch program – Change byte…



Change the 2A to 2B.

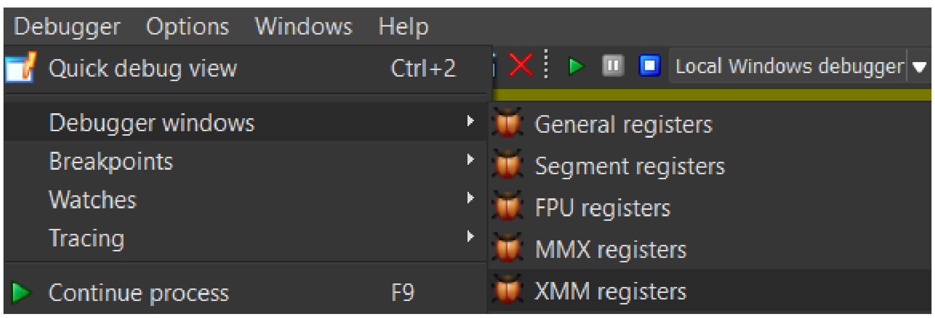


Click ok. We have now hacked our 42 value to 43!

Let’s put a breakpoint on the below line.

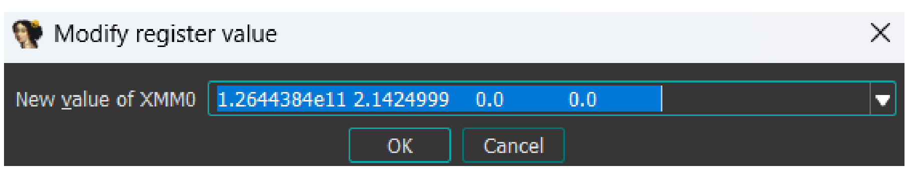
Lets press F9 to run to it.

Click on Debugger – Debugger windows – XMM registers

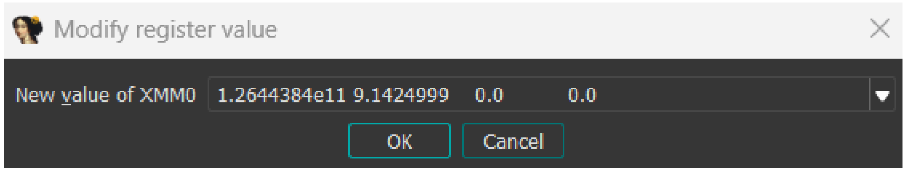


It will load the XMM registers.

Double-click on the 2.1424999.



Change the value to 9 instead of 2.



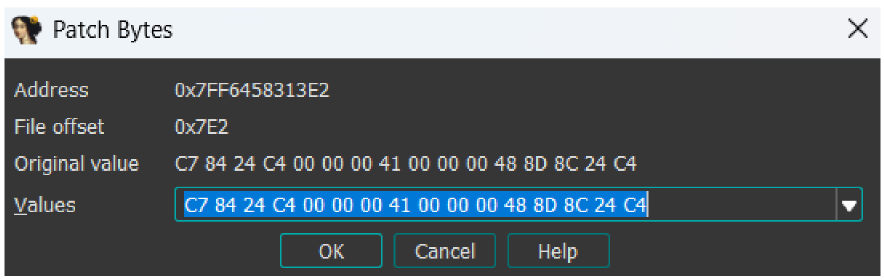
Press OK.

We have successfully hacked 3.14 to 299499.58!

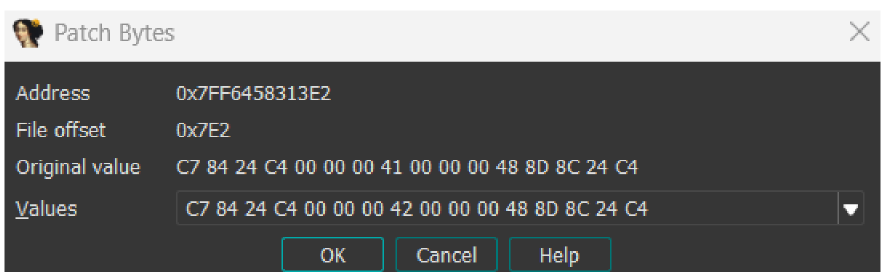
Put a breakpoint on the below line and press F9.

```
.text:00007FF6458313E2 mov     dword ptr [rsp+108h+var_44.value], 41h
```

Click edit – Patch program – Change byte…



Change 41 to 42.



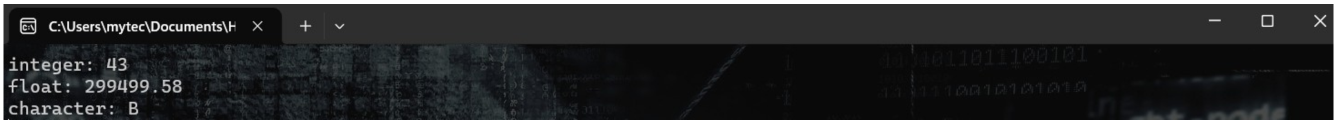
Press the red stop button to stop debugging.

Click edit – Patch program – Apply patches to input file…

Place a breakpoint on the NOP under the final call to *println!* macro.

```
call    _ZN3std2io5stdio6_print17hce7a376ab49946d5E ; std::io::stdio::_print::hce7a376ab49946d5
nop
```

BOOM!



In our next lesson we will handle Compound Data Types.

Chapter 7: Compound Data Types

We continue our journey with a compound data types example program in Rust on a Windows 64-bit OS.

Let’s create a new project and get started by following the below steps.

```
cargo new three_compound-data-types
```

Now let’s populate our **main.rs** file with the following.

```
fn main() {
    let tup = (1337, 3.14, 42);
    let (x, y, z) = tup;
    println!("x: {x}");
    println!("y: {y}");
    println!("z: {z}");

    let x = [1, 2, 3];
    let one = x[0];
    let two = x[1];
    let three = x[2];
    println!("one: {one}");
    println!("two: {two}");
    println!("three: {three}");
}
```

Let’s open up the terminal by click CTRL+SHIFT+` and type the following.

```
cargo build
```

Let’s run the binary!

```
.\target\debug\three_compound-data-types.exe
```

Output…

```
x: 1337
y: 3.14
z: 42
one: 1
two: 2
three: 3
```

Congratulations! You just created another program in Rust. Time for cake!

We simply created an example of each of the compound data types which are tuple and array.

In our next lesson we will debug this in IDA Free!

Chapter 8: Debugging Scalar Data Types

Let’s debug our app within IDA Free.

Open IDA Free and we see the load screen. We can keep all the defaults and simply click OK.

On the left-hand side we see the Function name window and we want to search for *three_compound_data_types__main*. Double-click it and we will see the function.

```
; void __fastcall three_compound_data_types::main()
three_compound_data_types__main proc near

var_2C8= qword ptr -2C8h
var_2B8= qword ptr -2B8h
var_2B0= qword ptr -2B0h
var_2A8= qword ptr -2A8h
var_2A0= qword ptr -2A0h
var_298= qword ptr -298h
var_290= qword ptr -290h
var_288= qword ptr -288h
var_280= qword ptr -280h
var_278= dword ptr -278h
var_274= dword ptr -274h
var_26C= dword ptr -26Ch
var_268= qword ptr -268h
var_25C= dword ptr -25Ch
var_258= byte ptr -258h
var_228= qword ptr -228h
var_220= qword ptr -220h
var_218= byte ptr -218h
var_1E8= qword ptr -1E8h
var_1E0= qword ptr -1E0h
var_1D8= byte ptr -1D8h
var_1A8= qword ptr -1A8h
var_1A0= qword ptr -1A0h
var_198= dword ptr -198h
var_194= dword ptr -194h
```

```
var_194= dword ptr -194h
var_190= dword ptr -190h
var_18C= dword ptr -18Ch
var_188= dword ptr -188h
var_184= dword ptr -184h
var_180= byte ptr -180h
var_150= qword ptr -150h
var_148= qword ptr -148h
var_140= byte ptr -140h
var_110= qword ptr -110h
var_108= qword ptr -108h
var_100= byte ptr -100h
var_D0= qword ptr -0D0h
var_C8= qword ptr -0C8h
var_C0= qword ptr -0C0h
var_B8= qword ptr -0B8h
var_B0= qword ptr -0B0h
var_A8= qword ptr -0A8h
var_A0= qword ptr -0A0h
var_98= qword ptr -98h
var_90= qword ptr -90h
var_88= qword ptr -88h
var_80= qword ptr -80h
var_78= qword ptr -78h
var_70= qword ptr -70h
var_68= qword ptr -68h
var_60= qword ptr -60h
var_58= qword ptr -58h
var_50= qword ptr -50h
var_48= qword ptr -48h
var_40= qword ptr -40h
var_38= qword ptr -38h
```

```
var_38= qword ptr -38h
var_30= qword ptr -30h
var_28= qword ptr -28h
var_20= qword ptr -20h
var_18= qword ptr -18h
var_10= qword ptr -10h
var_8= qword ptr -8

sub     rsp, 2E8h
mov     [rsp+2E8h+var_278], 539h
mov     rax, 40091EB851EB851Fh
mov     [rsp+2E8h+var_280], rax
mov     [rsp+2E8h+var_274], 2Ah ; '*'
mov     eax, [rsp+2E8h+var_278]
mov     [rsp+2E8h+var_26C], eax
movsd   xmm0, [rsp+2E8h+var_280]
movsd   [rsp+2E8h+var_268], xmm0
mov     eax, [rsp+2E8h+var_274]
mov     [rsp+2E8h+var_25C], eax
lea     rcx, [rsp+2E8h+var_26C]
mov     [rsp+2E8h+var_30], rcx
lea     rax, _ZN4core3fmt3num3imp52_$LT$impl$u20$core__fmt__Display$u20$for$u20$i32$GT$3fmt17hc51309275f9f171fE ; core::fmt::num::imp::_$LT$impl$u20$core..fmt..Display
mov     [rsp+2E8h+var_288], rax
mov     [rsp+2E8h+var_28], rax
mov     [rsp+2E8h+var_40], rcx
mov     [rsp+2E8h+var_38], rax
mov     rcx, [rsp+2E8h+var_40]
mov     rax, [rsp+2E8h+var_38]
mov     [rsp+2E8h+var_228], rcx
mov     [rsp+2E8h+var_220], rax
mov     rax, rsp
mov     qword ptr [rax+20h], 1
```

```
lea     rdx, off_140022420 ; "x: \n"
lea     rcx, [rsp+2E8h+var_258]
mov     [rsp+2E8h+var_2B8], rcx
mov     r8d, 2
mov     [rsp+2E8h+var_298], r8
lea     r9, [rsp+2E8h+var_228]
call    _ZN4core3fmt9Arguments6new_v117h646cbffa619ae3b3E ; core::fmt::Arguments::new_v1::h646cbffa619ae3b3
mov     rcx, [rsp+2E8h+var_2B8]
call    _ZN3std2io5stdio6_print17h445fdab5382e0576E ; std::io::stdio::_print::h445fdab5382e0576
mov     r8, [rsp+2E8h+var_298]
lea     rcx, [rsp+2E8h+var_268]
mov     [rsp+2E8h+var_10], rcx
lea     rax, _ZN4core3fmt5float52_$LT$impl$u20$core__fmt__Display$u20$for$u20$f64$GT$3fmt17hf255ac95ce93dfbbE ; core::fmt::float::_$LT$impl$u20$core..fmt..Display
mov     [rsp+2E8h+var_8], rax
mov     [rsp+2E8h+var_20], rcx
mov     [rsp+2E8h+var_18], rax
mov     rcx, [rsp+2E8h+var_20]
mov     rax, [rsp+2E8h+var_18]
mov     [rsp+2E8h+var_1E8], rcx
mov     [rsp+2E8h+var_1E0], rax
mov     rax, rsp
mov     qword ptr [rax+20h], 1
lea     rdx, off_140022448
lea     rcx, [rsp+2E8h+var_218]
mov     [rsp+2E8h+var_2B0], rcx
lea     r9, [rsp+2E8h+var_1E8]
call    _ZN4core3fmt9Arguments6new_v117h646cbffa619ae3b3E ; core::fmt::Arguments::new_v1::h646cbffa619ae3b3
mov     rcx, [rsp+2E8h+var_2B0]
call    _ZN3std2io5stdio6_print17h445fdab5382e0576E ; std::io::stdio::_print::h445fdab5382e0576
mov     r8, [rsp+2E8h+var_298]
mov     rax, [rsp+2E8h+var_288]
lea     rcx, [rsp+2E8h+var_25C]
```

```
mov     [rsp+2E8h+var_50], rcx
mov     [rsp+2E8h+var_48], rax
mov     [rsp+2E8h+var_60], rcx
mov     [rsp+2E8h+var_58], rax
mov     rcx, [rsp+2E8h+var_60]
mov     rax, [rsp+2E8h+var_58]
mov     [rsp+2E8h+var_1A8], rcx
mov     [rsp+2E8h+var_1A0], rax
mov     rax, rsp
mov     qword ptr [rax+20h], 1
lea     rdx, off_140022470
lea     rcx, [rsp+2E8h+var_1D8]
mov     [rsp+2E8h+var_2A8], rcx
lea     r9, [rsp+2E8h+var_1A8]
call    _ZN4core3fmt9Arguments6new_v117h646cbffa619ae3b3E ; core::fmt::Arguments::new_v1::h646cbffa619ae3b3
mov     rcx, [rsp+2E8h+var_2A8]
call    _ZN3std2io5stdio6_print17h445fdab5382e0576E ; std::io::stdio::_print::h445fdab5382e0576
mov     r8, [rsp+2E8h+var_298]
mov     rax, [rsp+2E8h+var_288]
mov     [rsp+2E8h+var_198], 1
mov     [rsp+2E8h+var_194], 2
mov     [rsp+2E8h+var_190], 3
mov     ecx, [rsp+2E8h+var_198]
mov     [rsp+2E8h+var_18C], ecx
mov     ecx, [rsp+2E8h+var_194]
mov     [rsp+2E8h+var_188], ecx
mov     ecx, [rsp+2E8h+var_190]
mov     [rsp+2E8h+var_184], ecx
lea     rcx, [rsp+2E8h+var_18C]
mov     [rsp+2E8h+var_70], rcx
mov     [rsp+2E8h+var_68], rax
mov     [rsp+2E8h+var_80], rcx
```

```
mov     [rsp+2E8h+var_78], rax
mov     rcx, [rsp+2E8h+var_80]
mov     rax, [rsp+2E8h+var_78]
mov     [rsp+2E8h+var_150], rcx
mov     [rsp+2E8h+var_148], rax
mov     rax, rsp
mov     qword ptr [rax+20h], 1
lea     rdx, off_140022498 ; "one: "
lea     rcx, [rsp+2E8h+var_180]
mov     [rsp+2E8h+var_2A0], rcx
lea     r9, [rsp+2E8h+var_150]
call    _ZN4core3fmt9Arguments6new_v117h646cbffa619ae3b3E ; core::fmt::Arguments::new_v1::h646cbffa619ae3b3
mov     rcx, [rsp+2E8h+var_2A0]
call    _ZN3std2io5stdio6_print17h445fdab5382e0576E ; std::io::stdio::_print::h445fdab5382e0576
mov     r8, [rsp+2E8h+var_298]
mov     rax, [rsp+2E8h+var_288]
lea     rcx, [rsp+2E8h+var_188]
mov     [rsp+2E8h+var_90], rcx
mov     [rsp+2E8h+var_88], rax
mov     [rsp+2E8h+var_A0], rcx
mov     [rsp+2E8h+var_98], rax
mov     rcx, [rsp+2E8h+var_A0]
mov     rax, [rsp+2E8h+var_98]
mov     [rsp+2E8h+var_110], rcx
mov     [rsp+2E8h+var_108], rax
mov     rax, rsp
mov     qword ptr [rax+20h], 1
lea     rdx, off_1400224C0 ; "two: "
lea     rcx, [rsp+2E8h+var_140]
mov     [rsp+2E8h+var_290], rcx
lea     r9, [rsp+2E8h+var_110]
call    _ZN4core3fmt9Arguments6new_v117h646cbffa619ae3b3E ; core::fmt::Arguments::new_v1::h646cbffa619ae3b3
```

```
mov     rcx, [rsp+2E8h+var_290]
call    _ZN3std2io5stdio6_print17h445fdab5382e0576E ; std::io::stdio::_print::h445fdab5382e0576
mov     rax, [rsp+2E8h+var_288]
lea     rcx, [rsp+2E8h+var_184]
mov     [rsp+2E8h+var_B0], rcx
mov     [rsp+2E8h+var_A8], rax
mov     [rsp+2E8h+var_C0], rcx
mov     [rsp+2E8h+var_B8], rax
mov     rcx, [rsp+2E8h+var_C0]
mov     rax, [rsp+2E8h+var_B8]
mov     [rsp+2E8h+var_D0], rcx
mov     [rsp+2E8h+var_C8], rax
lea     rcx, [rsp+2E8h+var_100]
lea     rdx, off_1400224E8 ; "three: "
mov     r8d, 2
lea     r9, [rsp+2E8h+var_D0]
mov     [rsp+2E8h+var_2C8], 1
call    _ZN4core3fmt9Arguments6new_v117h646cbffa619ae3b3E ; core::fmt::Arguments::new_v1::h646cbffa619ae3b3
lea     rcx, [rsp+2E8h+var_100]
call    _ZN3std2io5stdio6_print17h445fdab5382e0576E ; std::io::stdio::_print::h445fdab5382e0576
nop
add     rsp, 2E8h
retn
three_compound_data_types__main endp
```

Ok… I know this can be completely overwhelming but again like all of the other Reverse Engineering & Hacking (REDAH) tutorials I have made over the years we will take it one step at a time!

We see the below instruction.

```
mov     [rsp+2E8h+var_278], 539h
```

We remember in our source code our first tuple value is 1337. Therefore 0x539 is 1337.

We then see a call to the *println!* macro to print this value.

```
call    _ZN3std2io5stdio6_print17h445fdab5382e0576E ; std::io::stdio::_print::h445fdab5382e0576
```

Next we are going to handle the next tuple value of 3.14.

Let’s set a breakpoint at the below and run.

```
.text:00007FF66657135F lea     rcx, [rsp+2E8h+var_268]
```

Let’s take a step with F8 and double-click on the *[rsp+2E8h+var_268]* value.

We see the values on the stack below.

```
Stack[0000398C]:0000007B3375F970 db  1Fh
Stack[0000398C]:0000007B3375F971 db  85h
Stack[0000398C]:0000007B3375F972 db 0EBh
Stack[0000398C]:0000007B3375F973 db  51h
Stack[0000398C]:0000007B3375F974 db 0B8h
Stack[0000398C]:0000007B3375F975 db  1Eh
Stack[0000398C]:0000007B3375F976 db    9
Stack[0000398C]:0000007B3375F977 db  40h
```

Let’s convert the hex values to binary.

```
1Fh -> 00011111
85h -> 10000101
0EBh -> 000011101011
51h -> 01010001
0B8h -> 000010111000
1Eh -> 00011110
09 -> 00001001
40h -> 01000000
```

```
00011111 10000101 000011101011 01010001 000010111000 00011110 00001001 01000000
```

This binary sequence represents the 64-bit double-precision floating-point number according to the IEEE 754 standard.

* Sign bit: 0 (positive)
* Exponent: 00011111101 (1021 in decimal)
* Significand: 1.0100001100010000000000000000000000000000000000000000

When you convert this binary representation to decimal, it is approximately equal to 3.14.

We then see a call to the *println!* macro to print this value.

```
.text:00007FF6665713DF call    _ZN3std2io5stdio6_print17h445fdab5382e0576E ; std::io::stdio::_print::h445fdab5382e0576
```

Next we are going to handle the next tuple value of 42.

Let’s set a breakpoint on the below.

```
.text:00007FF6665713EE lea     rcx, [rsp+2E8h+var_25C]
```

Let’s F8 and step over.

Let’s examine the values within *[rsp+2E8h+var_25C]*.

```
Stack[0000398C]:0000007B3375F97C db  2Ah
```

We know 42 is 0x2a in hex.

We then see a call to the *println!* macro to print this value.

```
.text:00007FF666571467 call    _ZN3std2io5stdio6_print17h445fdab5382e0576E ; std::io::stdio::_print::h445fdab5382e0576
```

Next we are going to handle the array values of 1, 2 and 3.

We see them all being placed into offsets of the stack.

```
.text:00007FF666571476 mov     [rsp+2E8h+var_198], 1
.text:00007FF666571481 mov     [rsp+2E8h+var_194], 2
.text:00007FF66657148C mov     [rsp+2E8h+var_190], 3
```

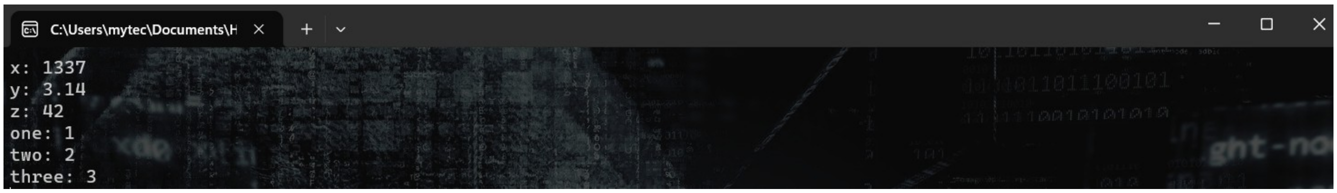
We then see the three calls to the *println!* macro which are spaced out in the Assembler.

```
.text:00007FF66657153A call    _ZN3std2io5stdio6_print17h445fdab5382e0576E ; std::io::stdio::_print::h445fdab5382e0576
```

```
.text:00007FF6665715C2 call    _ZN3std2io5stdio6_print17h445fdab5382e0576E ; std::io::stdio::_print::h445fdab5382e0576
```

```
.text:00007FF666571647 call    _ZN3std2io5stdio6_print17h445fdab5382e0576E ; std::io::stdio::_print::h445fdab5382e0576
```

This prints the following to the terminal.



In our next lesson we will hack this!

Chapter 9: Hacking Compound Data Types

Let’s hack our app within IDA Free.

Open IDA Free and we see the load screen. We can keep all the defaults and simply click OK.

On the left-hand side we see the Function name window and we want to search for *three_compound_data_types__main*. Double-click it and we will see the function.

We see the first part of our program which is the first tuple value of 1337.

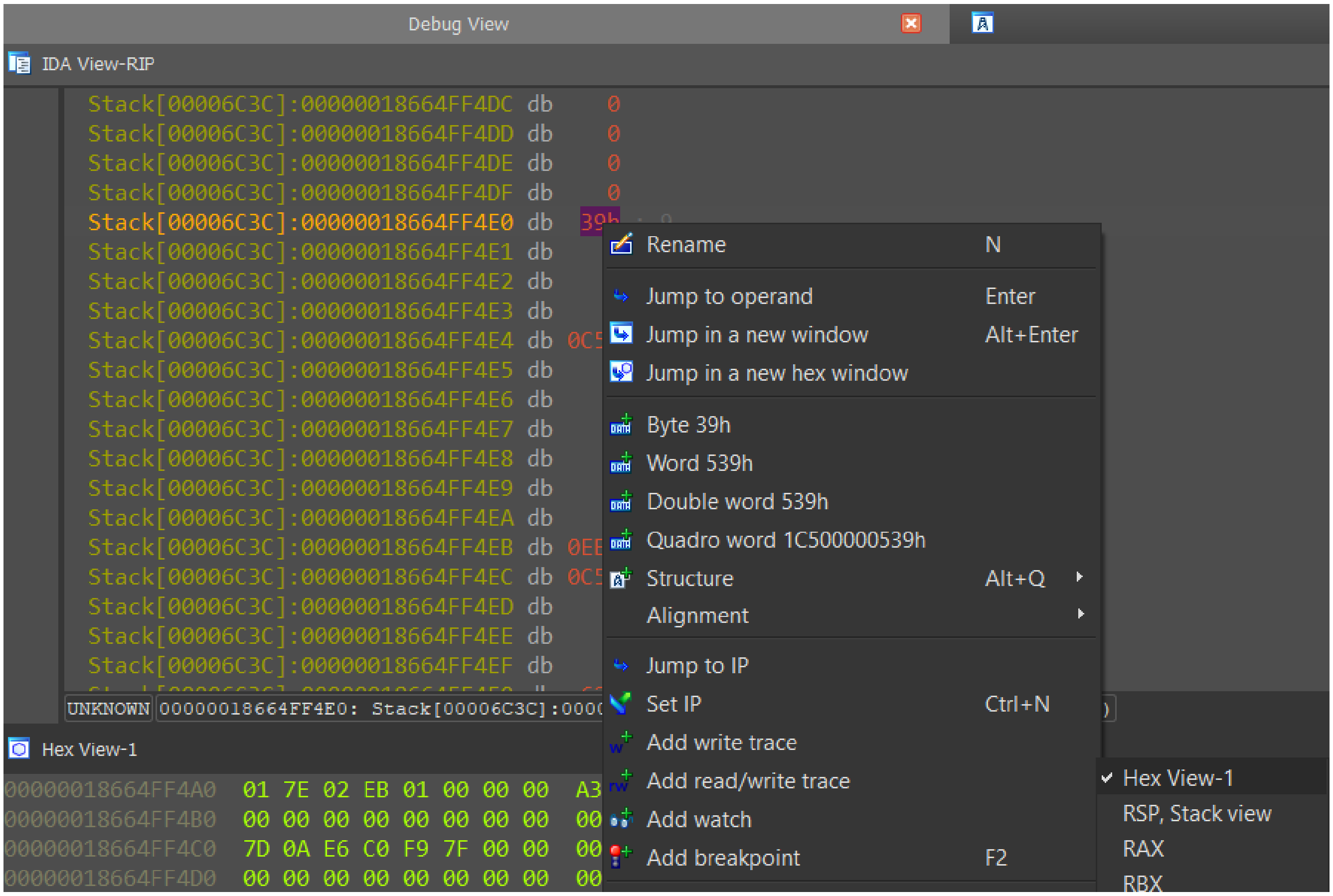
```
mov     [rsp+2E8h+var_278], 539h
```

Let’s set a breakpoint on the next line after this and run.

```
.text:00007FF75DF41287 mov     [rsp+2E8h+var_278], 539h
.text:00007FF75DF4128F mov     rax, 40091EB851EB851Fh
```

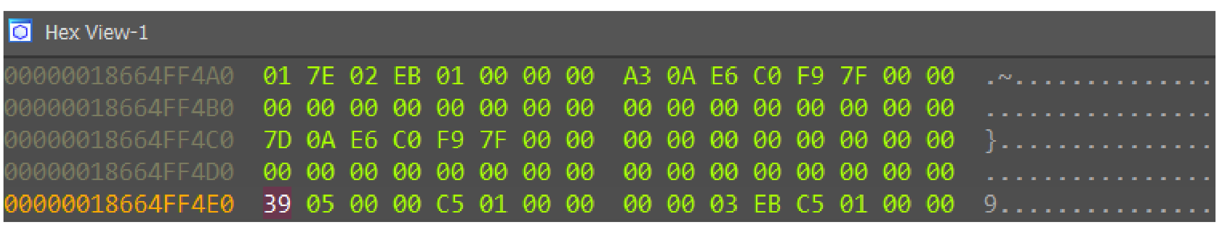
Let’s double-click in the *[rsp+2E8h+var_278]*.

It is important to have the hex view synced with our instructions.



We see a *db 5* and then above it *db 39* so this is our *0x539* value we are going to hack.

Right-click on the hex view where the *39* is.



Press F2 and it will turn grey and type 40.

Then press F2 again and it will turn green again.

Let's continue and see the result!

Success! We hacked the first part!

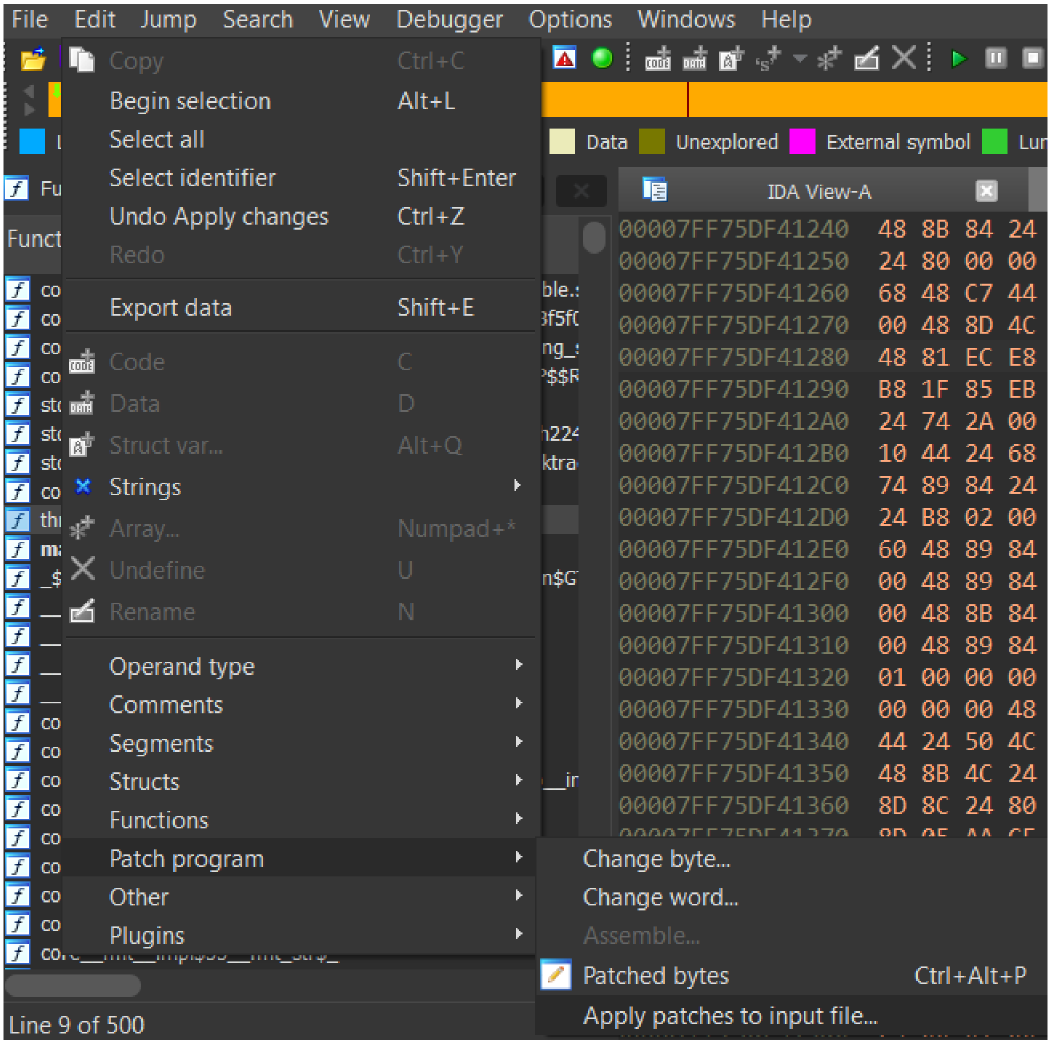
This is a different way to hack the variable as this was a RAM hack within our stack. This will not stay persistent on a re-run of the program but I wanted to show you yet another way to make a hack if you are attached to a running binary.

Let's stop our debugging session and click on the Hex View-1 tab and make sure the below line is selected first.

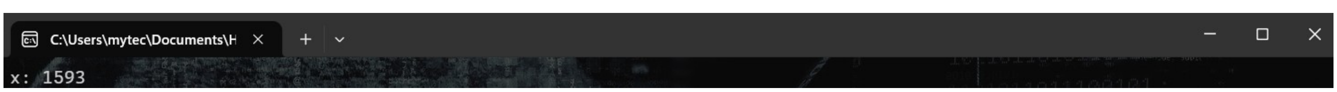
In the Hex View-1 tab.

Press F2 and type *06* and then press F2 again as it should go grey and then back to orange.

Then click *Edit* then *Patch program* then *Apply patches to input file...*



Now it will permanently change the binary. I deliberately changed the *06* this time to show the difference when we run.



To hack the value of the 3.14 tuple we have to take a step back and see what is happening in our binary. We know that in our debugging session we found the values inside *var_268* to be the below.

```
1Fh -> 00011111
85h -> 10000101
0EBh -> 000011101011
51h -> 01010001
0B8h -> 000010111000
1Eh -> 00011110
09 -> 00001001
40h -> 01000000

00011111 10000101 000011101011 01010001 000010111000 00011110 00001001 01000000
```

This binary sequence represents the 64-bit double-precision floating-point number according to the IEEE 754 standard.

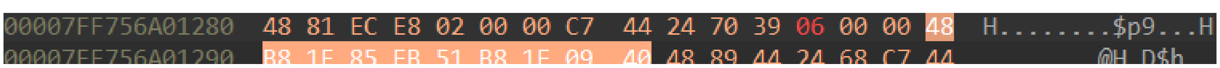
* Sign bit: 0 (positive)
* Exponent: 00011111101 (1021 in decimal)
* Significand: 1.0100001100010000000000000000000000000000000000000000

When you convert this binary representation to decimal, it is approximately equal to 3.14.

At the beginning of our main function we see variables being init with offsets and then we see this strange value being moved into rax below.

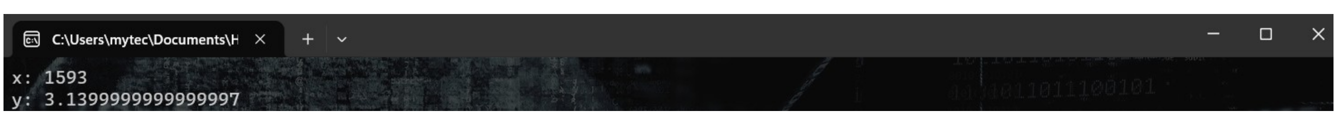
```
mov     rax, 40091EB851EB851Fh
```

Let’s highlight it and look in our Hex View-1 tab.



BOOM!

Let’s change *1F* to *1E* and patch and run.



To hack the next tuple value of 42 we have to locate the 0x2A value at the top of our binary and patch it to 0x2B. Since you have seen many examples of how to do this I will show the patched binary line.

```
mov     [rsp+2E8h+var_274], 2Bh
```

When we run we see the value of 43 now.

```
x: 1593
y: 3.1399999999999997
z: 43
```

Finally we have our array values of 1, 2 and 3.

```
mov     [rsp+2E8h+var_198], 1
mov     [rsp+2E8h+var_194], 2
mov     [rsp+2E8h+var_190], 3
```

As we look at the hex please look for the patterns of *01 00 00 00* and then within the next byte *02 00 00 00* and then within the next byte *03 00 00 00*.

```
00007FF7F17F11F0  50 48 8B 44 24 60 C7 84  24 50 01 00 00 01 00 00  PH.D$`Ǐ.$P......
00007FF7F17F1200  00 C7 84 24 54 01 00 00  02 00 00 00 C7 84 24 58  .Ǐ.$T.......Ǐ.$X
00007FF7F17F1210  01 00 00 03 00 00 00 8B  8C 24 50 01 00 00 89 8C  .........$P.....
```

We need to change the *01* to *04* and the *02* to *05* and the *03* to *06*.

```
mov     [rsp+2E8h+var_198], 4
mov     [rsp+2E8h+var_194], 5
mov     [rsp+2E8h+var_190], 6
```

We can see the patched bytes.

```
00007FF769DB11F0  50 48 8B 44 24 60 C7 84  24 50 01 00 00 04 00 00  PH.D$`Ǐ.$P......
00007FF769DB1200  00 C7 84 24 54 01 00 00  05 00 00 00 C7 84 24 58  .Ǐ.$T.......Ǐ.$X
00007FF769DB1210  01 00 00 06 00 00 00 8B  8C 24 50 01 00 00 89 8C  .........$P.....
```

Let’s run!

```
x: 1593
y: 3.1399999999999997
z: 43
one: 4
two: 5
three: 6
```

Hooray! We hacked it!

In our next lesson we will cover functions.

Chapter 10: Functions

In this chapter we will cover functions within Rust.

We will then reverse engineer the binary in IDA Free.

Let’s create a new project and get started by following the below steps.

```
rustup update
cargo new four_functions
```

Now let’s populate our **main.rs** file with the following.

```
fn main() {
    let x = plus_one(42);

    println!("The value of x is: {x}");
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

Let’s open up the terminal by click CTRL+SHIFT+` and type the following.

```
cargo build
```

Let’s run the binary!

```
.\target\debug\four_functions.exe
```

Output…

```
The value of x is: 43
```

Here we create a function called *plus_one* and where we have a i32 param called x and it returns and i32.

But wait… There is no return statement!

The code x + 1 is an expression as expressions evaluate to a value and that is implicitly returned when the function ends.

In our next lesson we will debug this in IDA Free!

Chapter 11: Debugging Functions

Let’s debug our app within IDA Free.

Open IDA Free and we see the load screen. We can keep all the defaults and simply click OK.

In Rust at the assembler level we will need to search for the entry point of our app. This is the *four_function__main* function. You can use CTRL+F to search.

```
; void __fastcall four_functions::main()
four_functions__main proc near

var_78= qword ptr -78h
var_64= dword ptr -64h
var_60= byte ptr -60h
var_30= qword ptr -30h
var_28= qword ptr -28h
var_20= qword ptr -20h
var_18= qword ptr -18h
var_10= qword ptr -10h
var_8= qword ptr -8

sub     rsp, 98h
mov     ecx, 2Ah ; '*'  ; int
call    four_functions__plus_one
mov     [rsp+98h+var_64], eax
lea     rcx, [rsp+98h+var_64]
mov     [rsp+98h+var_10], rcx
lea     rax, _ZN4core3fmt3num3imp52_$LT$impl$u20$core__fmt__Display$u20$for$u20$i32$GT$3fmt17h5db180deba73ee2eE ; core::fmt::num::imp::_$LT$impl$u20$core..fmt..Displ
mov     [rsp+98h+var_8], rax
mov     [rsp+98h+var_20], rcx
mov     [rsp+98h+var_18], rax
mov     rcx, [rsp+98h+var_20]
mov     rax, [rsp+98h+var_18]
mov     [rsp+98h+var_30], rcx
```

```
mov     [rsp+98h+var_78], 1
call    _ZN4core3fmt9Arguments6new_v117hf702a91e774c917fE ; core::fmt::Arguments::new_v1::hf702a91e774c917f
lea     rcx, [rsp+98h+var_60]
call    _ZN3std2io5stdio6_print17h7f843005243859e5E ; std::io::stdio::_print::h7f843005243859e5
nop
add     rsp, 98h
retn
four_functions__main endp
```

We see a call to *four_functions__plus_one* and the return value being put into EAX.

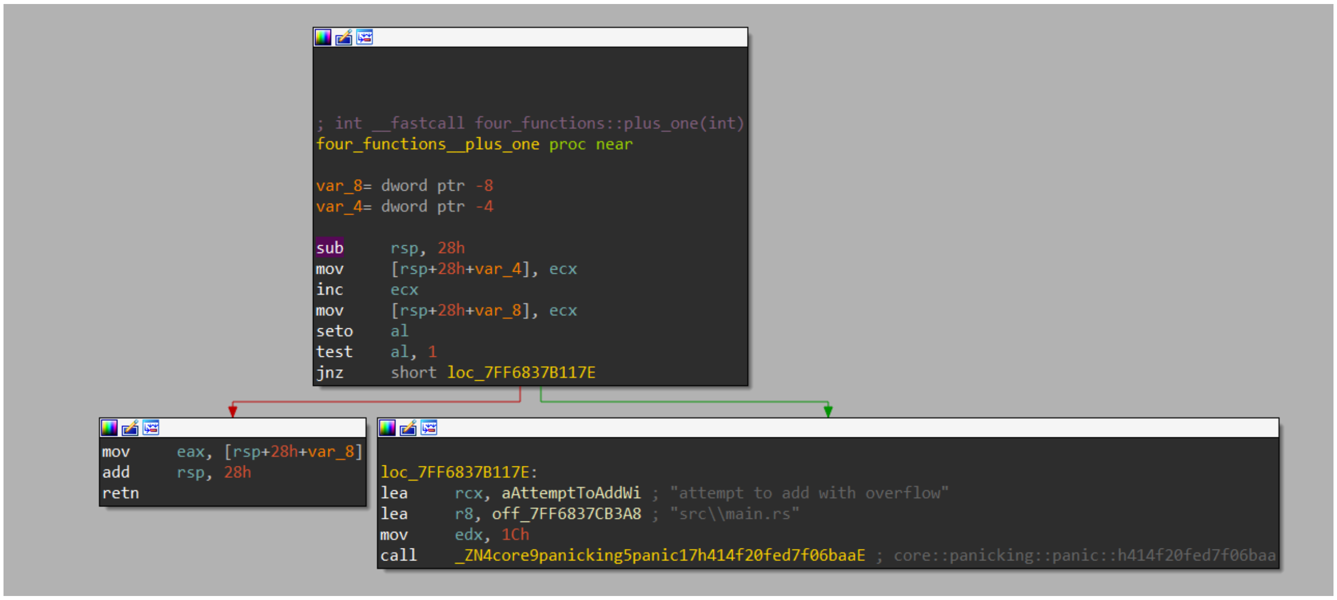
We can test this by putting a breakpoint on the *lea* instruction under and seeing if that is in fact correct.

```
.text:00007FF6837B10D7 mov     ecx, 2Ah ; '*'  ; int        RAX 000000000000002B        ID  0
.text:00007FF6837B10DC call    four_functions__plus_one     RBX 0000017F2554DE40 debug028:0000017F2554DE40   VIP 0
.text:00007FF6837B10E1 mov     [rsp+98h+var_64], eax        RCX 000000000000002B        VIF 0
.text:00007FF6837B10E5 lea     rcx, [rsp+98h+var_64]        RDX 0000000000000001        AC  0
.text:00007FF6837B10EA mov     [rsp+98h+var_10], rcx        RSI 0000000892F6F8B8 Stack[0000358C]:0000000892F6F   VM  0
```

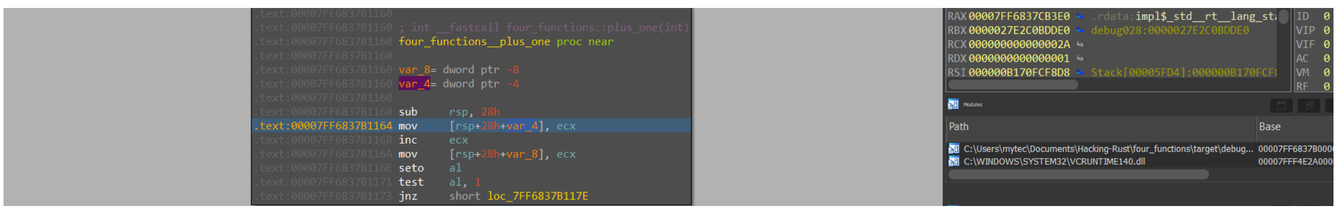
We can verify that *43* is in fact being returned back or *0x2b*.

The rest of main is straight forward as it is jut a call to *println* and some other formatting calls.

Let’s look at our *four_functions__plus_one* function.



Let’s set a breakpoint on the first *mov* instruction.



Here we see that the first param which holds *0x2a* or *42* is being passed in through RCX.

What is interesting here is that there is a *seto al* and *test al, 1* to essentially set the least significant byte of *al* (of the *RAX* register) to *1* and then perform a bitwise *AND* on it as it will set the zero flag if the result is zero otherwise *ZF* will be *0* if the result is non-zero.

If the *jnz* is true, it will panic as it sees and attempt to add with overflow.

If that is not the case, we see the new incremented value within *ECX* being placed into *EAX* to which we proved earlier.

In our next lesson we will learn how to hack this!

Chapter 12: Hacking Functions

Let’s hack our app within IDA Free.

Open IDA Free and we see the load screen. We can keep all the defaults and simply click OK.

On the left-hand side we see the Function name window and we want to search for *four_functions__plus_one*. Double-click it and we will see the function.

Let’s break on the below and .

```
.text:00007FF6837B1164 mov     [rsp+28h+var_4], ecx
```

Let’s set a breakpoint on the next line after this and run.

Let’s double-click *on RCX* and change the value to FF.

```
RCX 00000000000000FF ↳                    VIF 0
```

Let’s step until we are about to return back to *main*.

```
.text:00007FF6837B1179 add     rsp, 28h
```

We see that RAX is now 100 hex or 256. If we let this print to terminal we would now see 256.

Up until now we have hacked and patched our binary which we will do later but you can literally hack at any stage of the process like we just did here which will not be persistent.

Let’s go back to main and select the 2Ah.

```
mov     ecx, 2Ah ; '*'   ; int
```

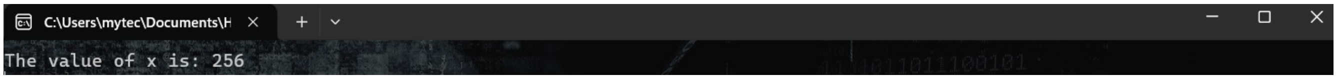
Let’s select the hex view.

```
00007FF6837B10D0  48 81 EC 98 00 00 00 B9  2A 00 00 00 E8 7F 00 00  H.......*.......
```

Let’s click Edit then Change byte and change the 2A to FF.

Then click *Edit* then *Patch program* then *Apply patches to input file...*

Now our change is permanent. Let’s set a breakpoint after the *println* call and look at the terminal.



Boom!

In our next lesson we will cover control flow.