FIRST EDITION – CHAPTER 3 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.

In later chapters we will within a Raspberry Pi 64-bit ARM OS so that you can get a perspective of what hacking that architecture looks like in Golang at the binary level.

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!

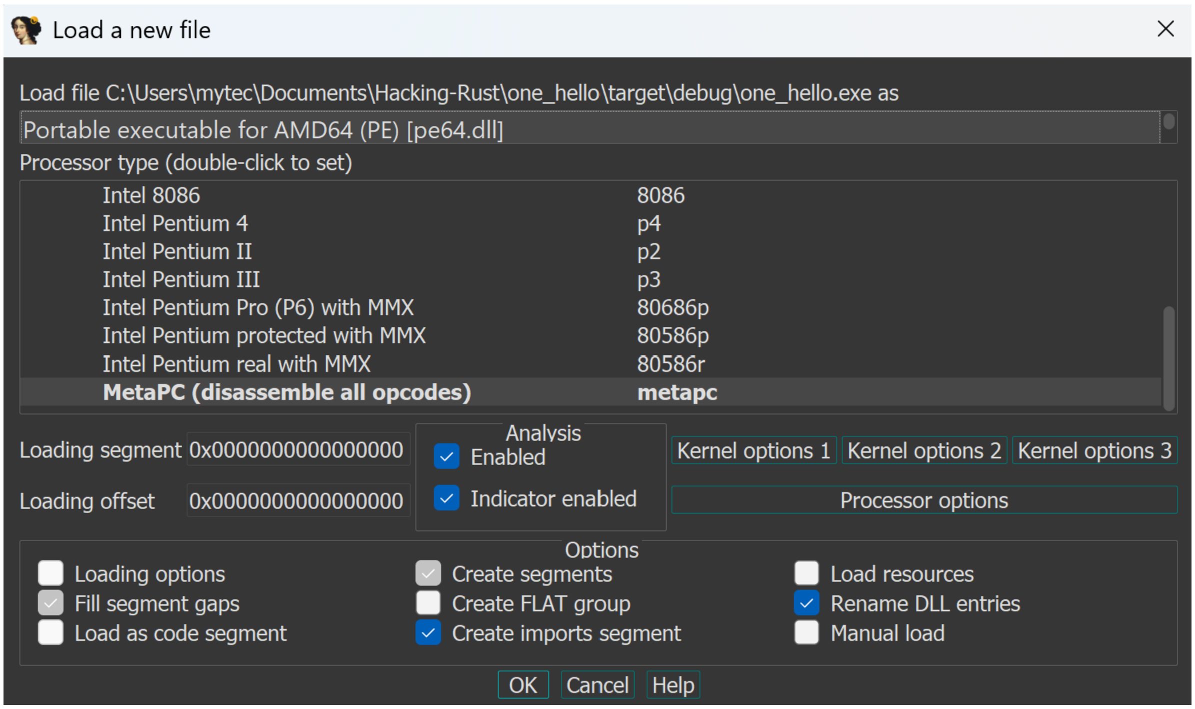
We simply created a hello world style example to get us started.

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

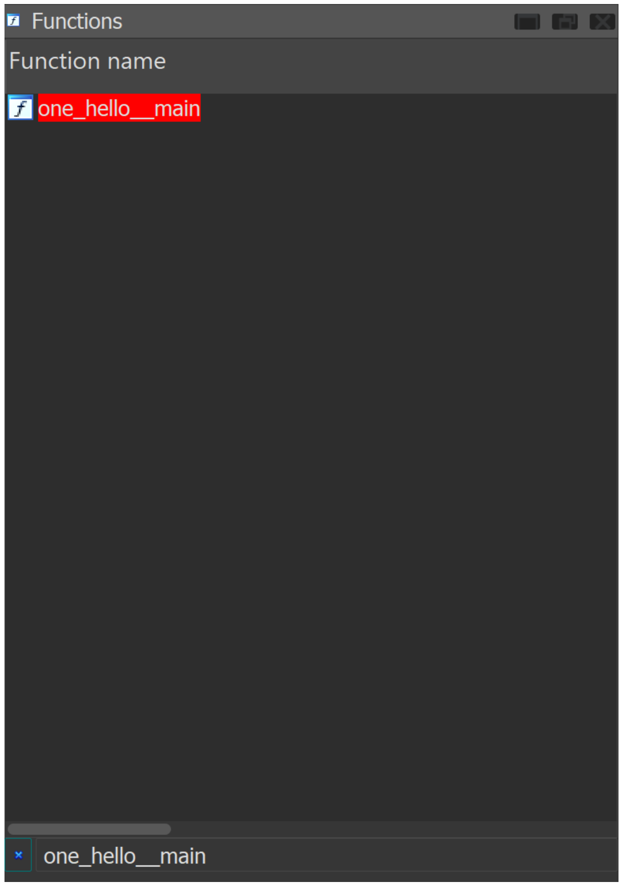
Chapter 2: Debugging Hello Rust

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.

```
; 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 can see in the box 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.

```
.rdata:000000014001E3A0 off_14001E3A0   dq offset aHelloWorld   ; DATA XREF: one_hello__main+9↑o
.rdata:000000014001E3A0                                         ; "Hello, world!\n"
.rdata:000000014001E3A8                 db  0Eh
```

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.

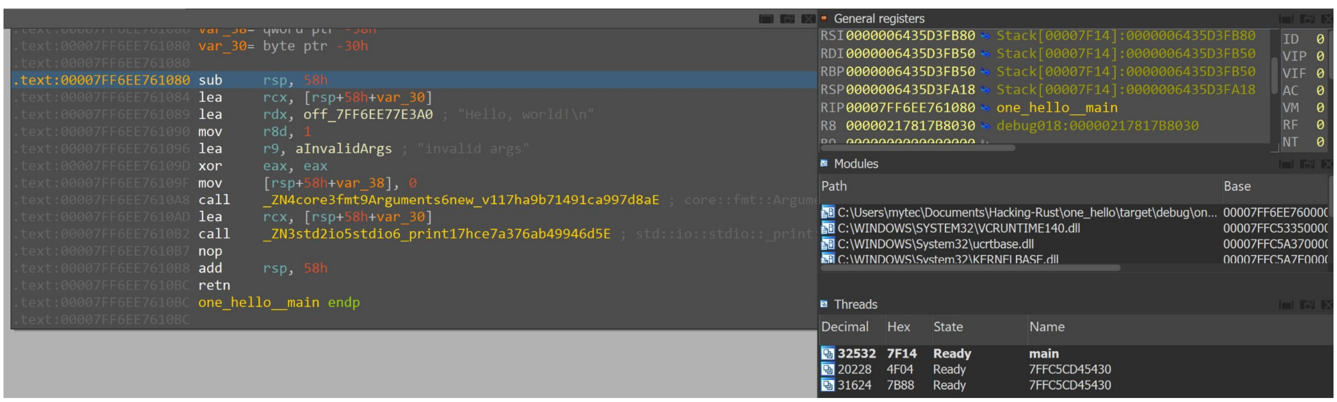
Let’s load up IDA and revisit the binary.

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.

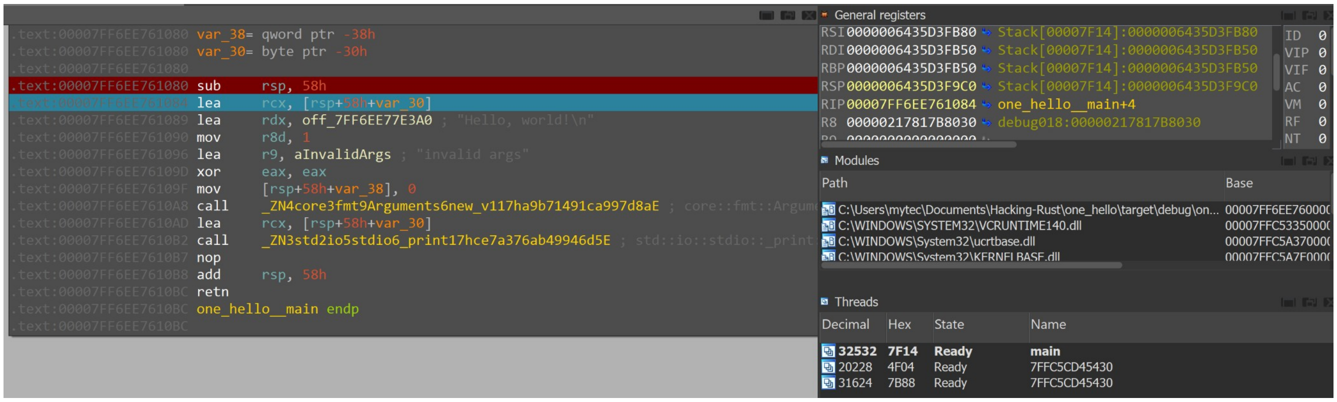
Let’s set a breakpoint at the entry.

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*.

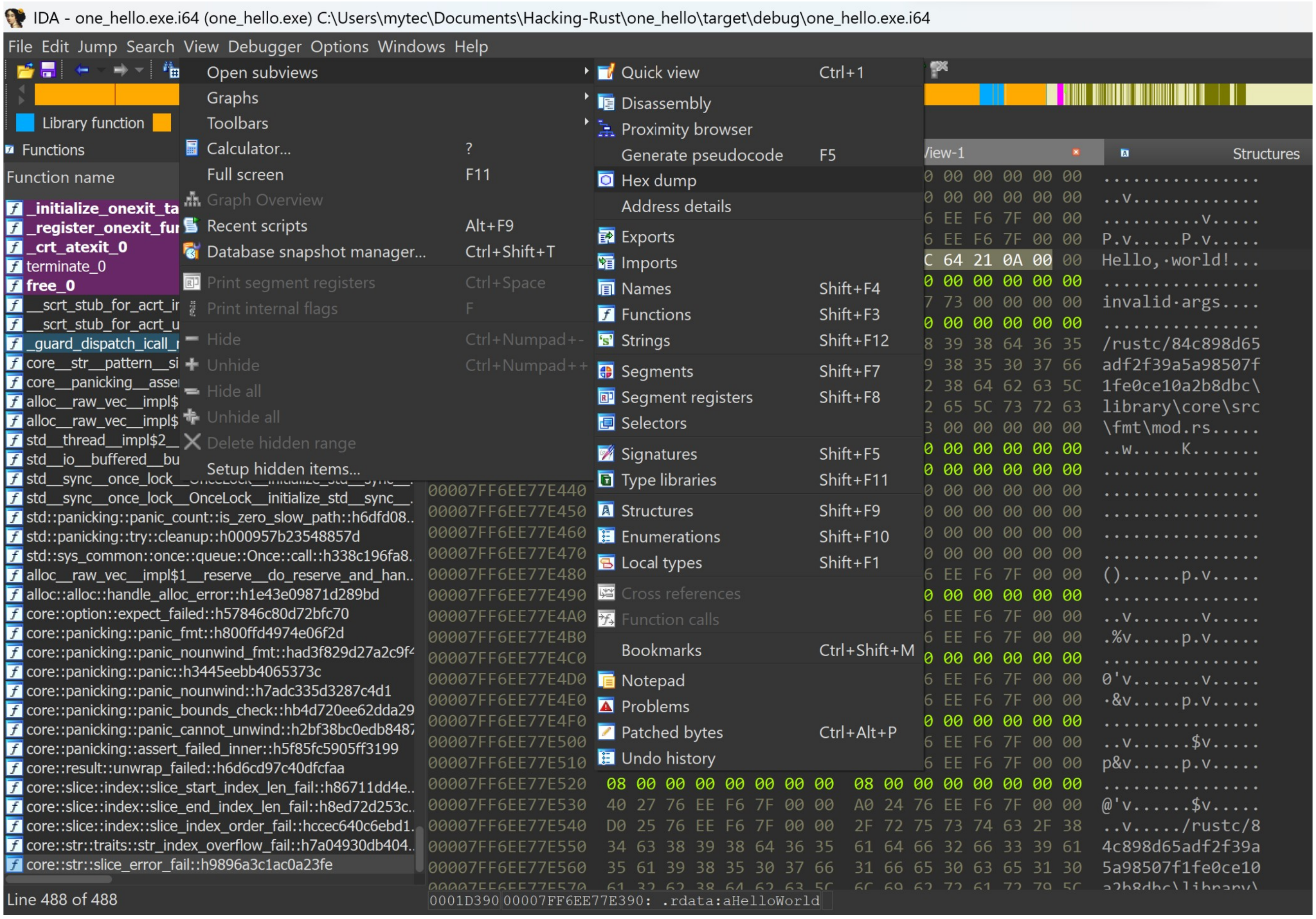
```
.text:00007FF6EE761080
.text:00007FF6EE761080
.text:00007FF6EE761080
.text:00007FF6EE761080 ; void __fastcall one_hello::main()
.text:00007FF6EE761080 one_hello__main proc near
.text:00007FF6EE761080
.text:00007FF6EE761080 var_38= qword ptr -38h
.text:00007FF6EE761080 var_30= byte ptr -30h
.text:00007FF6EE761080
.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.

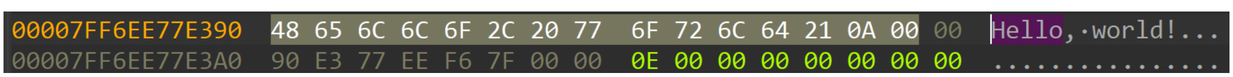
```
.rdata:00007FF6EE77E3A0 off_7FF6EE77E3A0 dq offset aHelloWorld  ; DATA XREF: one_hello__main+9↑o
.rdata:00007FF6EE77E3A0                                          ; "Hello, world!\n"
.rdata:00007FF6EE77E3A8 db  0Eh
```

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

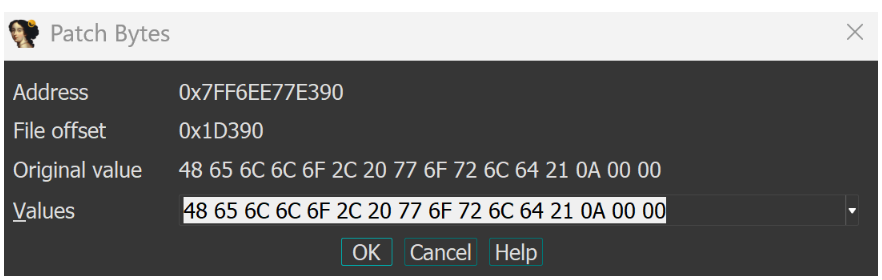
Let's stop the debug and click on the *"Hello, world!\n"* text and view the *Hex View-1*.



Here we see the hex view.

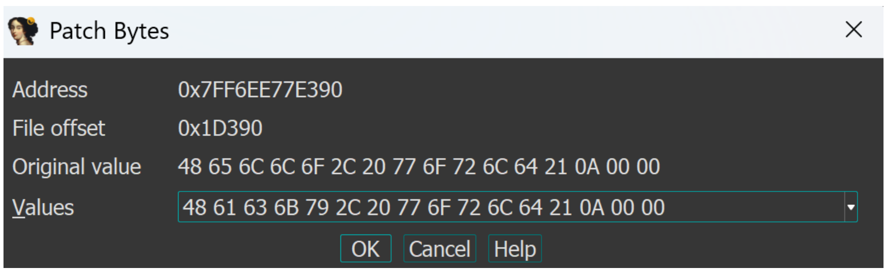


Click edit – Patch program – Change byte…

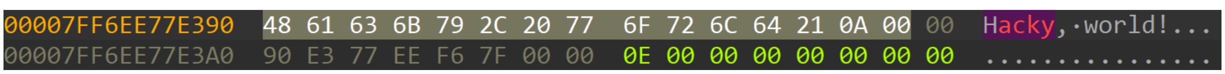


Let’s hack our binary to say “Hacky, world!\n” instead!

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

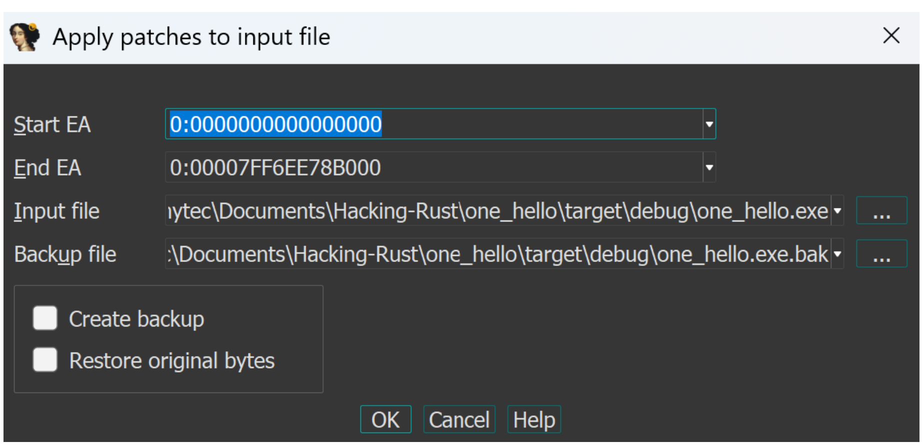


Click OK.



SUCCESS!

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 Type.