PESU Center for
Information Security,
Forensics and
Cyber Resilience

Cybsec Workshop

PESU
ISFCR

PESU Center for
Information Security,
Forensics and
Cyber Resilience

Introduction to reversing

PESU Center for Information Security, Forensics and Cyber Resilience

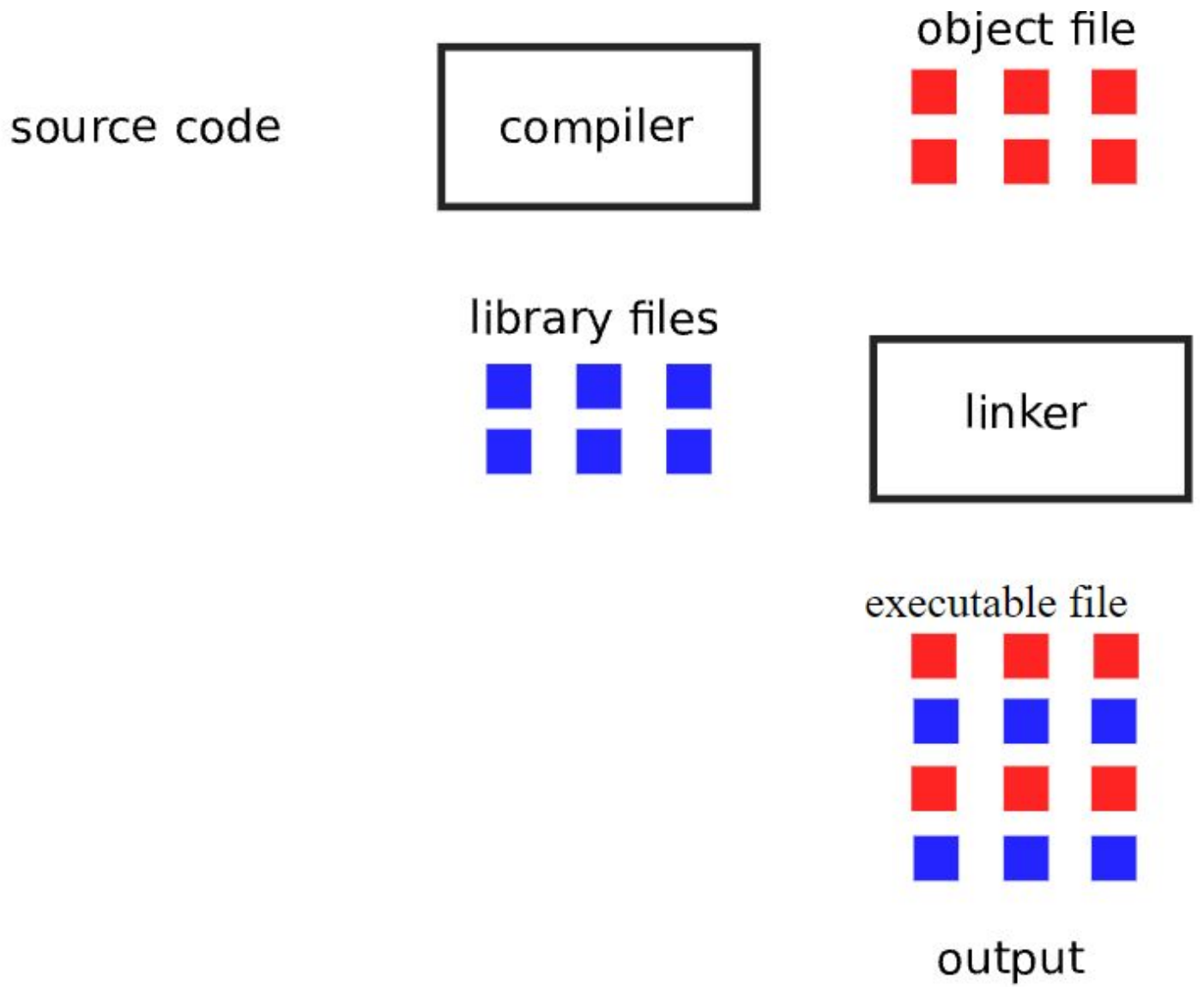
Contents

What is Reverse Engineering

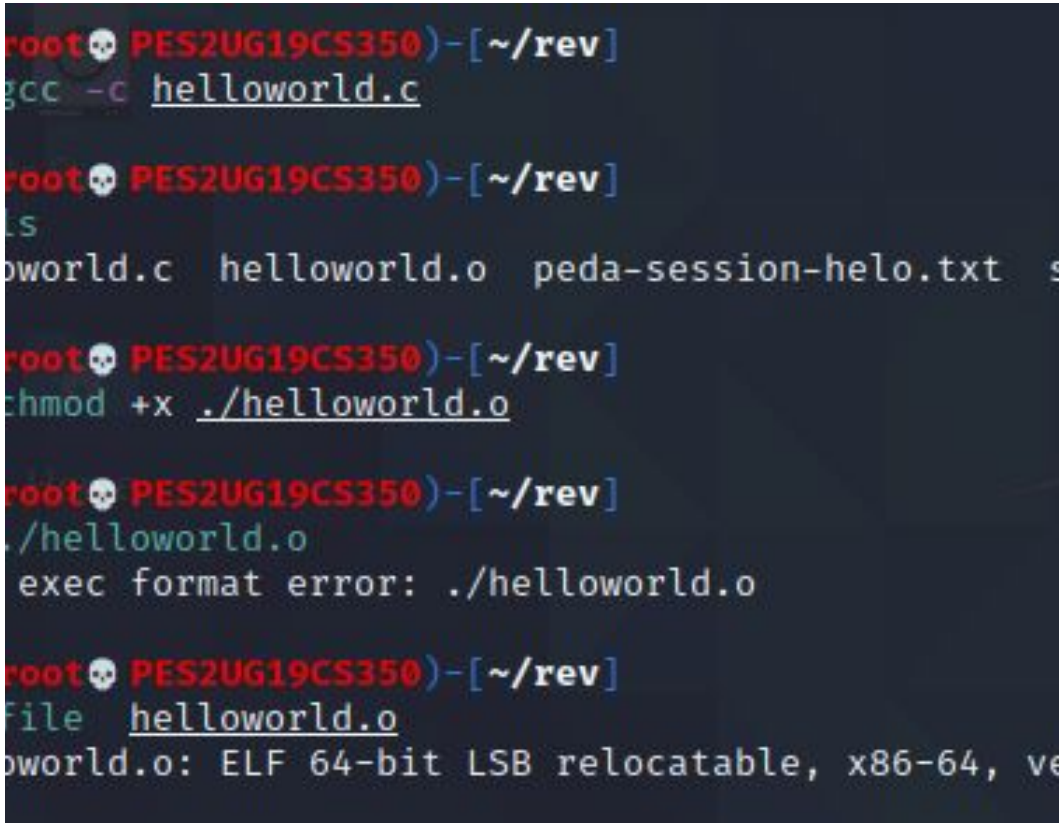
PESU Center for Information Security, Forensics and Cyber Resilience

- Generally To understand what Is reversing engineering.one must first understand how any higher level compiled languages is converted into machine readable / byte code,Ever Wondered what happens on the inside , how The computer reads and gives output magically from source code

- The Compiler which is a special program converts Source Code into machine language module called object file . And Another specialized program called the Linker combines this object file with other previously compiled object files (in particular run-time modules) to create an executable file

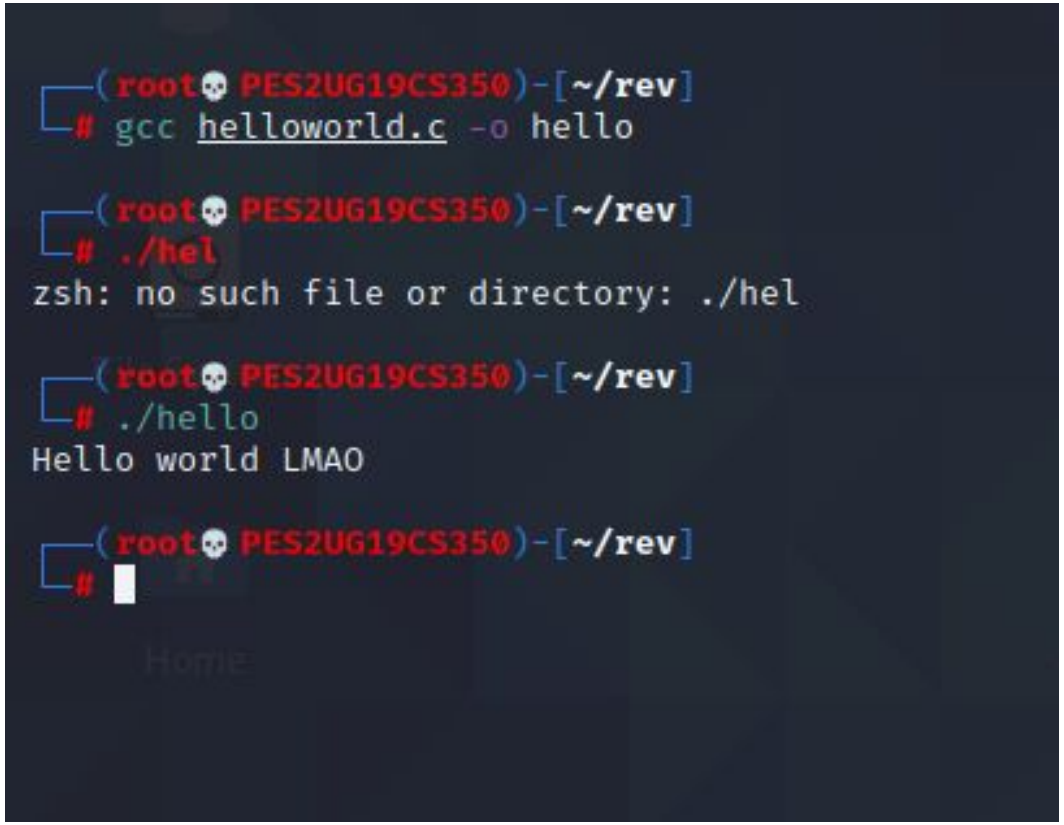

source code
compiler
object file
library files
linker
executable file
output

- To see the compiler and linker in action you can use “-c” option in gcc to compile a source without linking



- However you Cant run the executable file Yet because The Linker has not yet linked the Run Time Modules To the Object file (hence the exec Format error)

You Can Link the object file with the Linker Using the Standard gcc command



- Since Now you Have a Basic understanding Of how pseudo-readable code is converted into machine language by the compiler and Interpreter
- Reverse Engineering is typically the process of taking a compiled (machine code, bytecode) program and trying to understand what the program does and converting it back into a more human readable format.

Decompilers and Disassemblers

PESU ISFCR
PESU Center for Information Security, Forensics and Cyber Resilience

- A Diassembler is a Program which converts Machine / byte code into intermediate assembly instructions

```
Dump of assembler code for function main:
   0x000000000040052d <+0>:     push   %rbp
   0x000000000040052e <+1>:     mov    %rsp,%rbp
   0x0000000000400531 <+4>:     sub    $0x10,%rsp
   0x0000000000400535 <+8>:     movq   $0x4005e4,-0x10(%rbp)
   0x000000000040053d <+16>:    movq   $0x0,-0x8(%rbp)
   0x0000000000400545 <+24>:    mov    -0x10(%rbp),%rax
   0x0000000000400549 <+28>:    lea    -0x10(%rbp),%rcx
   0x000000000040054d <+32>:    mov    $0x0,%edx
   0x0000000000400552 <+37>:    mov    %rcx,%rsi
   0x0000000000400555 <+40>:    mov    %rax,%rdi
   0x0000000000400558 <+43>:    callq  0x400420 <execve@plt>
   0x000000000040055d <+48>:    leaveq
   0x000000000040055e <+49>:    retq
End of assembler dump.
```

- A Decompiler is program which “converts back” machine code into human readable format and pseudo-code

X86_64 architecture

PESU Center for Information Security, Forensics and Cyber Resilience

- x86 Assembly is the assembly instruction code used by the non ARM (Intel/AMD) processors

- There are 8 general purpose registers in both 32bit and 64 bit types
- And a special instruction pointer

- You can think of registers as special variables of the assembly world
- Which can be used to hold any type of data to which some have acquired specific use which are used in programs

- 32 bit arch has 6 general purpose registers and 2 special registers
- The 8 registered are named as EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
- In 64 bit they are named as RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP
- RSP is the stack pointer always points to the top of the process stack

- You Can perform basic assembly operations on these Registers Such as
- Mov
- Cmp
- Test
- lea
- Push
- Pop

Analyzing and Reversing Simple progr

PESU Center for Information Security, Forensics and Cyber Resilience

- Lets see a Simple compiled program Aka executable File which just prints text to output
- This is our source code for Reference

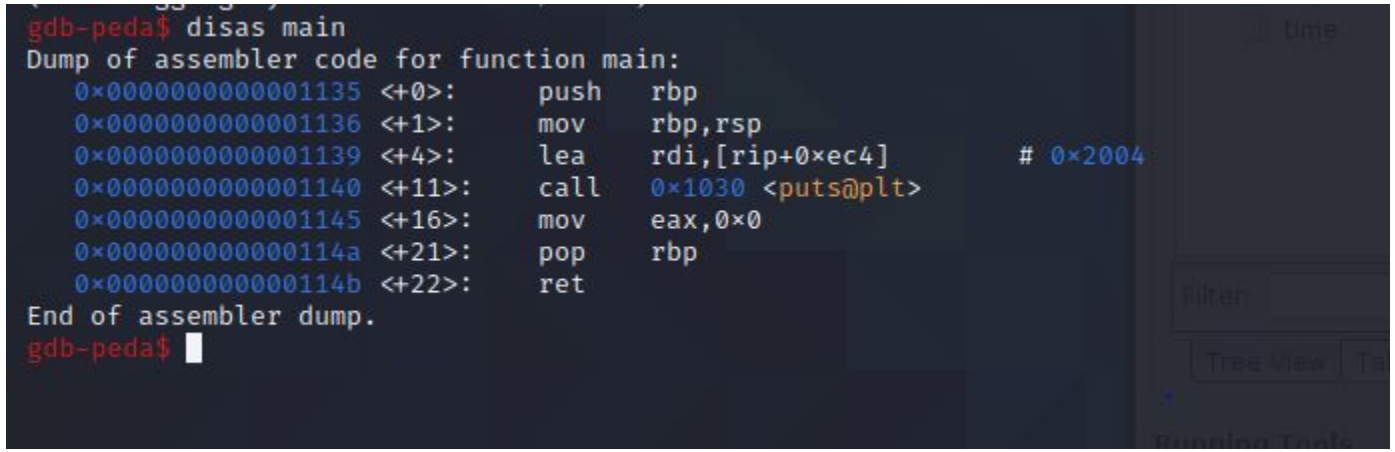
```c
#include<stdio.h>

int main(){
printf("Hello world LMAO\n");
return 0; }
```

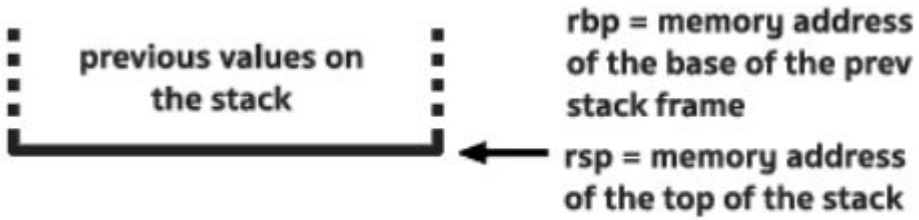
```
┌──(root💀PES2UG19CS350)-[~/rev]
└─# ./hello
Hello world LMAO
```

- Now Lets use gdb(disassembler) to take a look inside and Try to understand the Assembly!

- Pretty Small disassembly as expected let us go step by step to understand Whats happening



```
0x0000000000001135 <+0>:     push   rbp
```

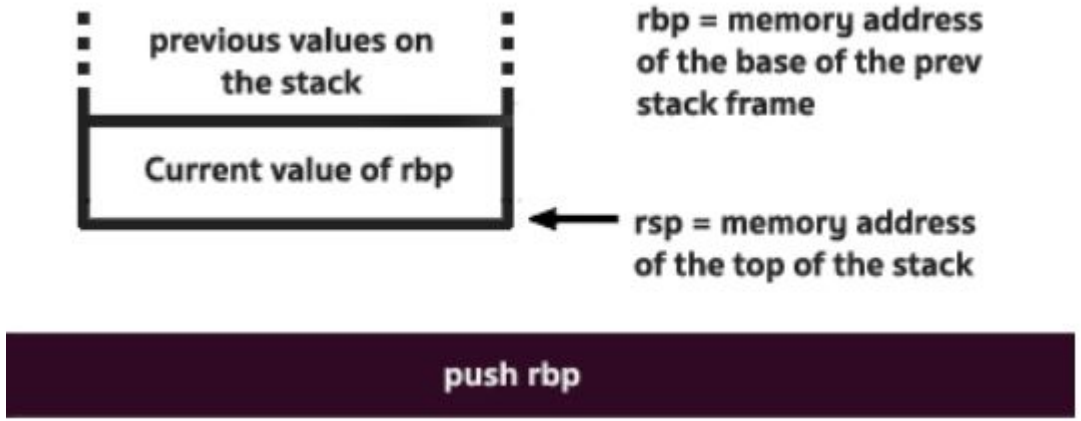


- The first Two instructions are Used for Setting up the Stack frame Of the process.so that it creates Space for the local variables and function parameters of the program

- Push Rbp: pushing the Current value present in RBP to the stack
- Because it “pushes” onto the stack, now the value of RSP contains memory address on top of new stack.

```
0x0000000000001136 <+1>:     mov    rbp,rsp
```

- The next instruction mov rbp,rsp copies the value of rsp into rbp
- And both point to top of the stack now.this is Done so that the process can subtract rsp by offset to make space for local variables
- while keeping rbp intact



```
0x0000000000001139 <+4>:     lea    rdi,[rip+0xec4]        # 0x2004
```


- The third instruction Uses Lea (load effective address) . In simple Terms it takes value present in that specific address and loads it into rdi general purpose register

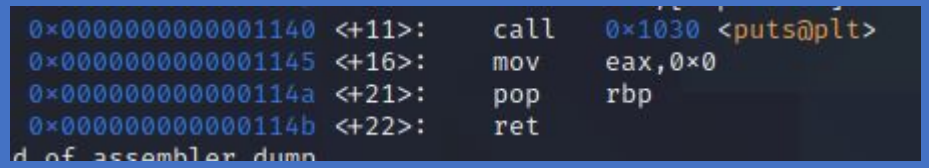
- The [] in assembly is used for dereferencing or in simple terms finds value present in that address .Similar to The * operator in c and c++

- so the third instruction.takes the value present in the address rip+0xec4 and puts it in the rdi register in our case it’s the hello world string in memory

- Lets see It live on gdb!
- As You can see the Register Rdi contains our string “hello World”
- Alternatively we can also check the RIP offset address just to make sure

```
RAX: 0×401122 (<main>:  push   rbp)
RBX: 0×0
RCX: 0×7ffff7fac718 —▸ 0×7ffff7faeb00 —▸ 0×0
RDX: 0×7fffffffe178 —▸ 0×7fffffffe482 ("COLORFGBG=15;0")
RSI: 0×7fffffffe168 —▸ 0×7fffffffe473 ("/root/rev/test")
RDI: 0×402004 ("Hello world LMAO")
```

```
gdb-peda$ x/s $rip+0×ed7
0×402004:       "Hello world LMAO"
gdb-peda$
```



PESU ISFCR

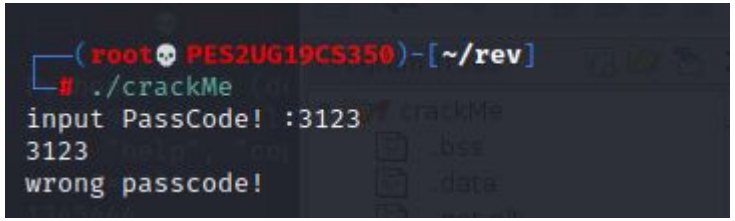
PESU Center for Information Security, Forensics and Cyber Resilience

- In x64 assembly the function parameters are passed through the registers. In certain order.Visit https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-160 to know more about this
- The 4th instruction calls the function Puts. But wait A minute you must have a doubt! Even though we used printf It is shown as puts in the disassembly.why?
- The compiler replaces printf with puts if there are no format specifiers such as %d for optimization reasons
- And after 4th instruction ends it prints out string to output
- The rest two instructions just clear the rax register and pop rbp from the stack
- And Voila we have just analyzed our first disassembly of a simple program

Solving a simple crackme

PESU Center for Information Security, Forensics and Cyber Resilience

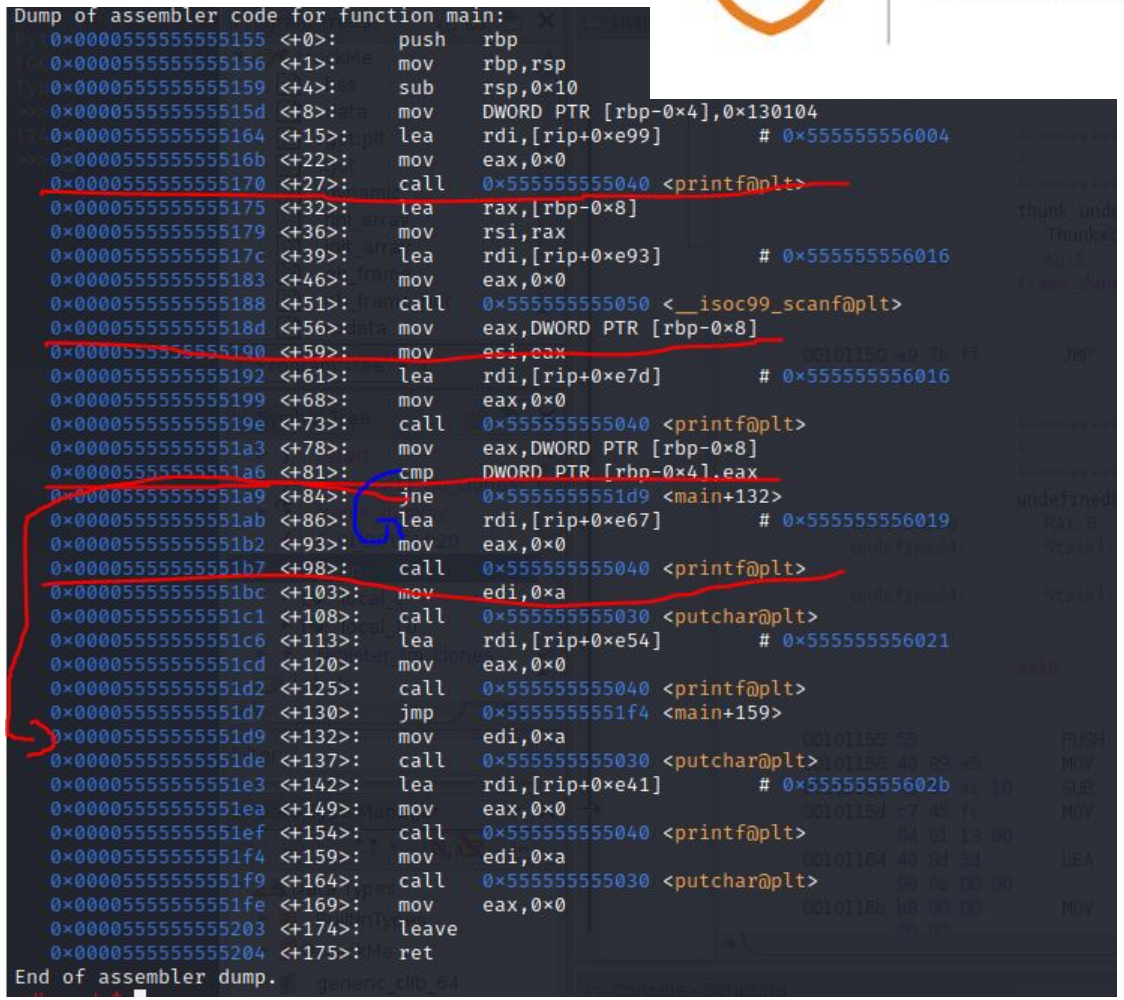
- Lets solve a simple crackme.Using Reverse Engineering



So The executable asks for a passcode and checks based according to that. But Since we Do not have the Source code we do not know the Passcode.But don’t worry lets try to crack this using Reversing engineering

The disassembly

- First lets take a look at the disassembly
- Seems like a lot of disassembly.But don't Worry lets go through only important ones I have underlined



- The first important instruction calls printf with parameters. So it prints “input passcode” string
- ***call 0x555555555040 <printf@plt>***
- Second Important instruction ***mov eax,DWORD PTR [rbp-0x8]***

seems to take Our Input from [rbp-0x8] and store it in eax

- The next most Important Instruction performs a cmp operation. Basically a if condition in Assembly
- So cmp checks if compares two registers by subtracting them
- Its similar to the SUB intstruction but IT does not affect operands
- so the assembly takes two paths one is JNE instruction fails(indicated by blue arrow) and IF JNE succeeds(indicated by red arrow)
- cmp DWORD PTR [rbp-0x4],eax
- The cmp instruction is comparing our value present in eax to value present in the address [rbp-0x4]
- So IF we can find the value present in [rbp-0x4] we have cracked the program!!

- Lets try to find the Value in Gdb By setting breakpoint just before the compare instruction

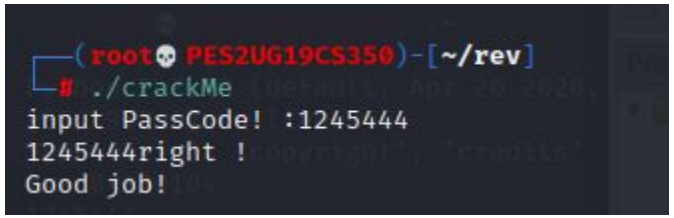
```
gdb-peda$ b *main+81
Breakpoint 2 at 0x555555555a6
gdb-peda$ r
Starting program: /root/rev/crackMe
input PassCode! :1234
```

- We can examine a memory region in gdb by x/wd command ‘w’ is for examining in word length or 4 bytes ‘d’ is for viewing values in integer type
- We have to examine [rbp-0x4] so we can type x/wd $rbp-0x4 to that memory region
- Lets do that in gdb

PESU Center for Information Security, Forensics and Cyber Resilience

- Looks like we Found our passcode present in the Binary

- Lets Input this value to the program and confirm whether this is correct.



- Hooray!! Seems like the Passcode worked. We Have successfully cracked a simple binary using reverse engineering

- Sometimes Reading a Lot of disassembly on gdb can be quite difficult

hence there are abundant of reversing engineering tools that make it easier to reverse

1) Ghidra - Ghidra is a free and open source reverse engineering tool developed by the National Security Agency of the United States.you can get it From their official github page https://github.com/NationalSecurityAgency/ghidra
2) Ida – interactive disassembler is also quite popular and used world wide by a lot of professionals

You can visit their page to download or know more about ida

https://hex-rays.com/ida-pro/

THANK YOU

PESU Center for
Information Security,
Forensics and
Cyber Resilience