

# Reverse Engineering

Thursday, February 17, 2022 9:11 AM

## Crackme1

Throw the crackme in ghidra goto the main function

```
***** undefined main() *****  
undefined undefined8 L:1 <RETURN>  
undefined undefined1 Stack[-0x10...local_10] XREF[2]: 0010077b(W),  
undefined undefined2 Stack[-0x16...local_16] XREF[2]: 0010077f(R),  
undefined undefined4 Stack[-0x18...local_18] XREF[1]: 001007a1(*),  
undefined undefined4 Stack[-0x1c...local_1c] XREF[2]: 0010077d(W),  
undefined undefined4 Stack[-0x20...local_20] XREF[2]: 00100793(W),  
main XREF[4]: Entry Point(*),  
main _start:0010067d(*), 001008fc,  
main 001009a8(*)  
0010076a 55 PUSH RBP  
0010076b 48 89 e5 MOV RBP, RSP  
0010076e 48 83 ec SUB RSP, 0x20  
20  
00100772 64 48 8b MOV RAX, qword ptr FS:[0x28]  
04 25 28  
00 00 00  
0010077b 48 89 45 MOV qword ptr [RBP + local_10], RAX  
f8  
0010077f 31 c0 XOR EAX, EAX  
00100781 48 8d 3d LEA RDI, [s_enter_password_00100894] = "enter password"  
0c 01 00  
00  
00100788 e8 83 fe CALL <EXTERNAL>:::puts  
ff ff  
0010078d b8 05 3d MOV EAX, dword ptr [DAT_001008d0] = 30786168h
```

Scanning in user input into local\_16 and comparing it to local\_1c

```
001007c4 48 89 c7 MOV RDI, RAX  
001007c7 e8 64 fe CALL <EXTERNAL>:::strcmp int strcmp(char * __sl, char * __sr)  
ff ff  
001007cc 89 45 e8 MOV dword ptr [RBP + local_20], EAX  
001007c1 83 7d e8 CMP dword ptr [RBP + local_20], 0x0  
00  
001007d3 75 13 JNZ LAB_001007e8
```

```
15 local_1c = 0x30786168;  
16 local_18 = 0x72;  
17 __isoc99_scanf(&DAT_001008a3, local_16);  
18 iVar1 = strcmp(local_16, (char *)local_1c);  
19 if (iVar1 == 0) {  
20 puts("password is correct");  
21 }  
22 else {
```

If we look in the decompiler we can see the key immediately, but we can also trace back the key to

```
72 64 20...  
001008d0 68 61 78 DAT_001008d0 XREF[1]: main:0010078d(R)  
30 undef... 30786168h
```

and then convert this to a string to get the password

```
08d0 68 61 78 s_r_001008d4 XREF[1,1]: main:0010078d(R),  
s_hexOr_001008d0 main:00100796(R)  
30 72 00 ds "hexOr"
```

## Crackme2

We again have a comparison

```
00 0010074b b8 00 00 MOV EAX, 0x0  
00 00  
00100750 e8 9b fe CALL <EXTERNAL>::__isoc99_scanf undefined __isoc99_scanf()  
ff ff  
00100755 8b 45 f4 MOV EAX, dword ptr [RBP + local_14]  
00100758 3d 7c 13 CMP EAX, 0x137c
```

```
10 puts("enter your password");  
11 __isoc99_scanf(DAT_00100838, &local_14);  
12 if (local_14 == 0x137c) {  
13 puts("password is valid");  
14 }  
15 else {  
16 puts("password is incorrect");  
17 }
```

But converting this to a string gives us garbled nonsense

```
00100755 8b 45 f4 MOV EAX, dword ptr [RBP + local_14]  
00100758 3d 7c 13 CMP EAX, 0x137c  
00 00
```

So we can pretty safely guess that the input is integer type

and we're correct!

```
00 0010074b b8 00 00 MOV EAX, 0x0  
00 00  
00100750 e8 9b fe CALL <EXTERNAL>::__isoc99_scanf undefined __isoc99_scanf()  
ff ff  
00100755 8b 45 f4 MOV EAX, dword ptr [RBP + local_14]  
00100758 3d 7c 13 CMP EAX, 4988
```

```
10 puts("enter your password");  
11 __isoc99_scanf(&DAT_00100838, &local_14);  
12 if (local_14 == 4988) {  
13 puts("password is valid");  
14 }  
15 else {  
16 puts("password is incorrect");  
17 }
```

## Crackme3

In this one we have a couple of constants that are probably up to no good

```

0010072f 31 c0    XOR    EAX,EAX
00100731 66 c7 45 MOV    word ptr [RBP + local_2b],0x7a61
dd 61 7a
00100737 c6 45 df MOV    byte ptr [RBP + local_29],0x74
74

```

I find the "password is correct" string and work my way backwards from there

```

02
0010079b 7e cb    JLE    LAB_00100763
0010079d 48 8d 3d  LEA    RDI,[s_password_is_correct_00100881] = "password is correct"
dd 00 00

```

\*\*important to remember that local\_28 is holding the user input\*\*

```

LEA    RAX=>local_28,[RBP + -0x20]
MOV    RSI,RAX
LEA    RDI,[DAT_00100868] = 25h %
MOV    EAX,0x0
CALL   <EXTERNAL>::__isoc99_scanf undefined __isoc99_scanf()

```

The following is key to understand

```

00100766 eb 2f    JMP    LAB_00100797
                LAB_00100768 XREF[1]: 0010079b(j)
00100768 8b 45 d8 MOV    EAX,dword ptr [RBP + local_30]
0010076b 48 98    CDQE
0010076d 0f b6 54 MOVZX EDX,byte ptr [RBP + RAX*0x1 + -0x20]
05 e0
00100772 8b 45 d8 MOV    EAX,dword ptr [RBP + local_30]
00100775 48 98    CDQE
00100777 0f b6 44 MOVZX EAX,byte ptr [RBP + RAX*0x1 + -0x23]
05 dd
0010077c 38 c2    CMP    DL,AL
0010077e 74 13    JZ    LAB_00100793
00100780 48 8d 3d  LEA    RDI,[s_password_is_incorrect_0010086b] = "password is incorrect"
e4 00 00
00
00100787 e8 44 fe CALL   <EXTERNAL>::puts int puts(char * __s)
ff ff
0010078c b8 00 00  MOV    EAX,0x0
00 00
00100791 eb 1b    JMP    LAB_001007ae
                LAB_00100793 XREF[1]: 0010077e(j)
00100793 83 45 d8 ADD    dword ptr [RBP + local_30],0x1
01
                LAB_00100797 XREF[1]: 00100766(j)
00100797 83 7d d8  CMP    dword ptr [RBP + local_30],0x2
02
0010079b 7e cb    JLE    LAB_00100763

```

local\_30 is a look counter, and it looks like unless the first three loops match a comparison, the program calls "password is incorrect", we want to avoid that

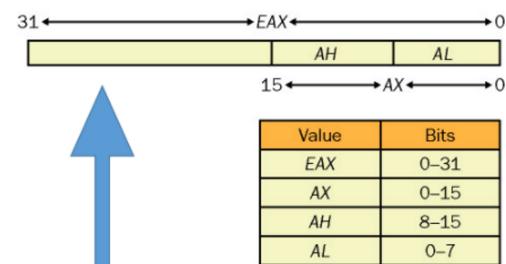
Ultimately, we want to pass this check three times

```

MOVZX EDX,byte ptr [RBP + RAX*0x1 + -32]
MOV    EAX,dword ptr [RBP + local_30]
CDQE
MOVZX EAX,byte ptr [RBP + RAX*0x1 + -35]
CMP    DL,AL

```

Recall that DL and AL are structured as follows



So we're comparing the least significant bits of EAX and EDX

We also know based on our reading of the assembly, that local\_28 is stored at RBP-32

```

00100747 48 8d 45    LEA      RAX=>local_28,[RBP + -32]
e0
0010074b 48 89 c6    MOV      RSI,RAX
0010074e 48 8d 3d    LEA      RDI,[DAT_00100868]
13 01 00
00
00100755 b8 00 00    MOV      EAX,0x0
00 00
0010075a e8 91 fe ff  CALL    <EXTERNAL>::__isoc99_scanf
ff ff
0010075f c7 45 d8    MOV      dword ptr [RBP + local_30],0x0
00 00 00
00
00100766 eb 2f    JMP     LAB_00100797

LAB_00100768          XREF[1]:
00100768 8b 45 d8    MOV      EAX,dword ptr [RBP + local_30]
0010076b 48 98        CDQE
0010076d 0f b6 54    MOVZX   EDX,byte ptr [RBP + RAX*0x1] + -32
05 e0

```

So the above shows us that our input is getting stored into EDX

```

EDX,byte ptr [RBP + RAX*0x1] + -32
EAX,dword ptr [RBP + local_30]
EAX,byte ptr [RBP + RAX*0x1] + -32

```

Those mystery characters come to mind right around now

```

MOV      word ptr [RBP + local_2b],"za"
MOV      byte ptr [RBP + local_29],'t'

```

Lets spin up gdb and see what's going on

and it looks like the first character that gets loaded to be compared to our input is 0x61(❶)...and the second...

```

*RAX 0x61
RBX 0x5555554007d0 (__libc_csu_init) ← push r15
RCX 0x0
RDX 0x61

```

and the second character is 0x7a(❷)

```

*RAX 0x7a
RBX 0x5555554007d0 (__libc_csu_init) ← push r15
RCX 0x0
RDX 0x61

```

And the last value is 0x74(❸), so it looks like our assessment was correct, the password is ❶❷❸

```

*RAX 0x74
RBX 0x5555554007d0 (__libc_csu_init) ← push r15
RCX 0x0
*RDX 0x74

```

```

[(kali㉿kali)-[~/THM/RE]]
$ ./crackme3.bin
enter your password
❶❷❸
password is correct

```