jamk

Reverse Engineering

Lab 04

Timo Lehosvuo, M3426

Report
Reverse Engineering, Marko Silokunnas
20.2.2021
ICT

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

Sisällys

1 Lab 04

As usual I started reverse engineering the file by trying to understand what the main function does:

```
mov     [esp+68h+var_68], ebx
mov     [esp+68h+var_64], 0
mov     [esp+68h+var_60], 30
mov     [ebp+var_3C], edi
mov     [ebp+var_40], esi
mov     [ebp+var_44], edx
call    _memset
mov     eax, [ebp+var_44]
mov     [esp+68h+var_68], eax
call    _printf
lea     ecx, aS          ; "%s"
lea     edx, [ebp+var_36]
mov     [esp+68h+var_68], ecx
mov     [esp+68h+var_64], edx
mov     [ebp+var_48], eax
call    ___isoc99_scanf
lea     ecx, [ebp+var_36]
mov     [esp+68h+var_68], ecx
mov     [ebp+var_4C], eax
call    check_password
and     al, 1
movzx   ecx, al
cmp     ecx, 1
jnz     loc_8048622
```

Figure 1: Main function.

Just like in lab03 this has *_memset* function but in this case, it is not necessary to clarify what it does to solve the lab, so I moved on to *check_password* function:

```
check_password proc near

var_14= dword ptr -14h
var_10_pw_pituus_kai= dword ptr -10h
var_9= byte ptr -9
var_8_user_input_kopio= dword ptr -8
var_1= byte ptr -1
arg_0_userinput= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 24         ; char *
mov     eax, [ebp+arg_0_userinput]
mov     [ebp+var_8_user_input_kopio], eax
mov     [ebp+var_9], 66
mov     eax, esp
mov     dword ptr [eax], offset PASSWORD ; ";r7$r7,&/'c%0v68c"
call    _strlen
mov     [ebp+var_10_pw_pituus_kai], eax
mov     eax, [ebp+var_8_user_input_kopio]
mov     ecx, esp
mov     [ecx], eax
call    _strlen
cmp     eax, [ebp+var_10_pw_pituus_kai]
jz      loc_804850E
```

Figure 2: check_password function.

My focus went to checking what the *_strlen* functions did and I realized that both the user input and the string ";r7$r7,&/'c%0v68c" lengths are being compared and depending on the result the

code jumps to different part on the code. The interesting part of the code could be found from the next section of the code:

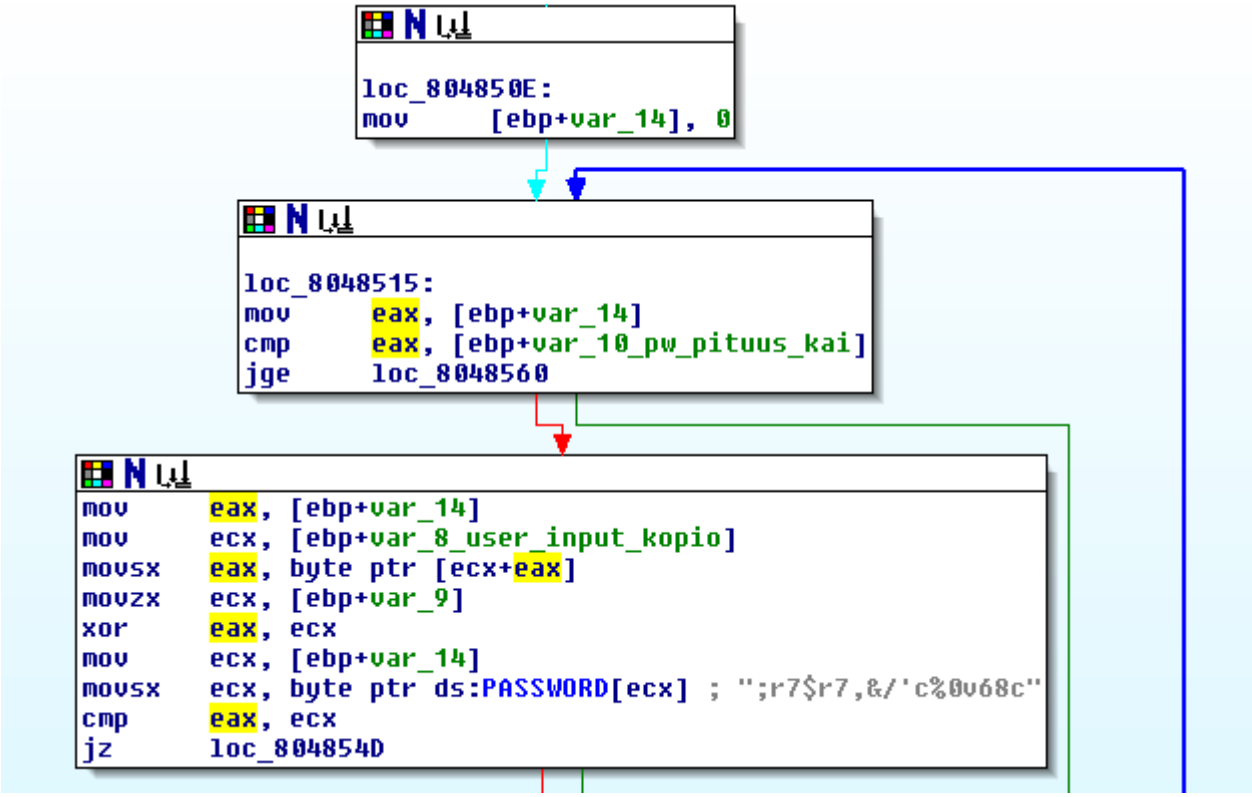


Figure 3: check_password function xor.

The code checks if *var_14* matches the length of the password and if it does not it goes to the bottom part of the loop. In this part of the code my focus went to the xor instruction and to the *cmp* instruction part. I figured out that the xor takes the “eax” byte (on the first run eax is 0 and the next one its 1 etc…) from the user input and xored it with *var_9* which is 66. The cmp instruction compares the “ecx” byte from the password function to the byte from the user input.

The loop continues until the length of var_14 is equal to the passwords length and I knew that it was a loop from this section of the code where var_14 is increased by one:

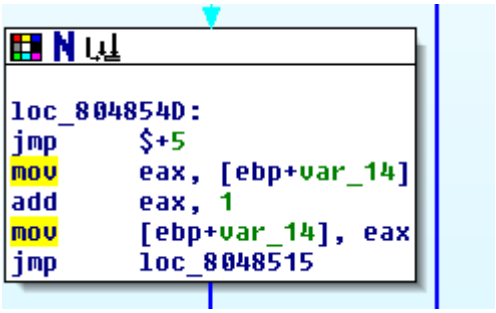


Figure 4: loop.

After figuring all this I still did not know how to get the password out until I realised that I can xor things “backwards”. The function xors the user input (which I don’t know) with decimal 66 so I can just xor the string “;r7$r7,&/'c%0v68c” with 66 and get the user input out. I tested this with a simple python script:

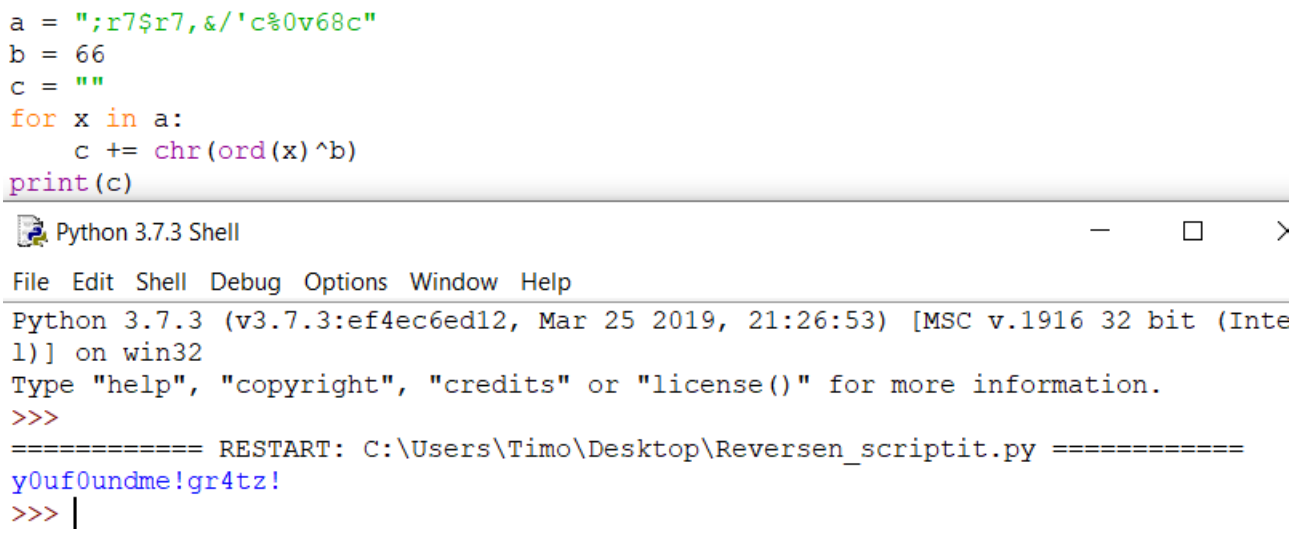
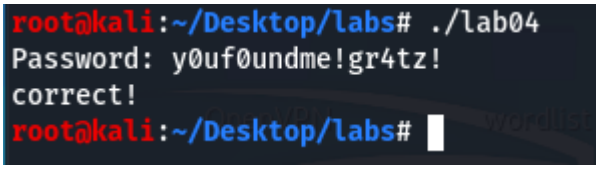


Figure 5: Python script.

Finally, I put the password in to the lab program:



and it was correct! 😊

2. Time spent

Report:	2 h
Solving the lab:	7 h
Total:	9 h