Reverse Engineering

Lab 03

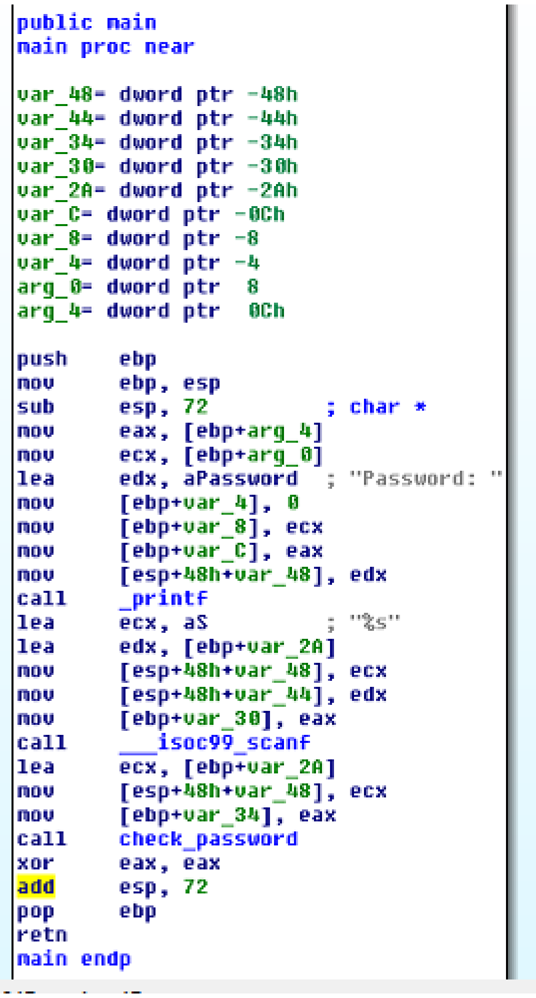
Timo Lehosvuo, M3426

Report
Reverse Engineering, Marko Silokunnas
20.2.2021
ICT

Sisällys

1 Lab 03

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



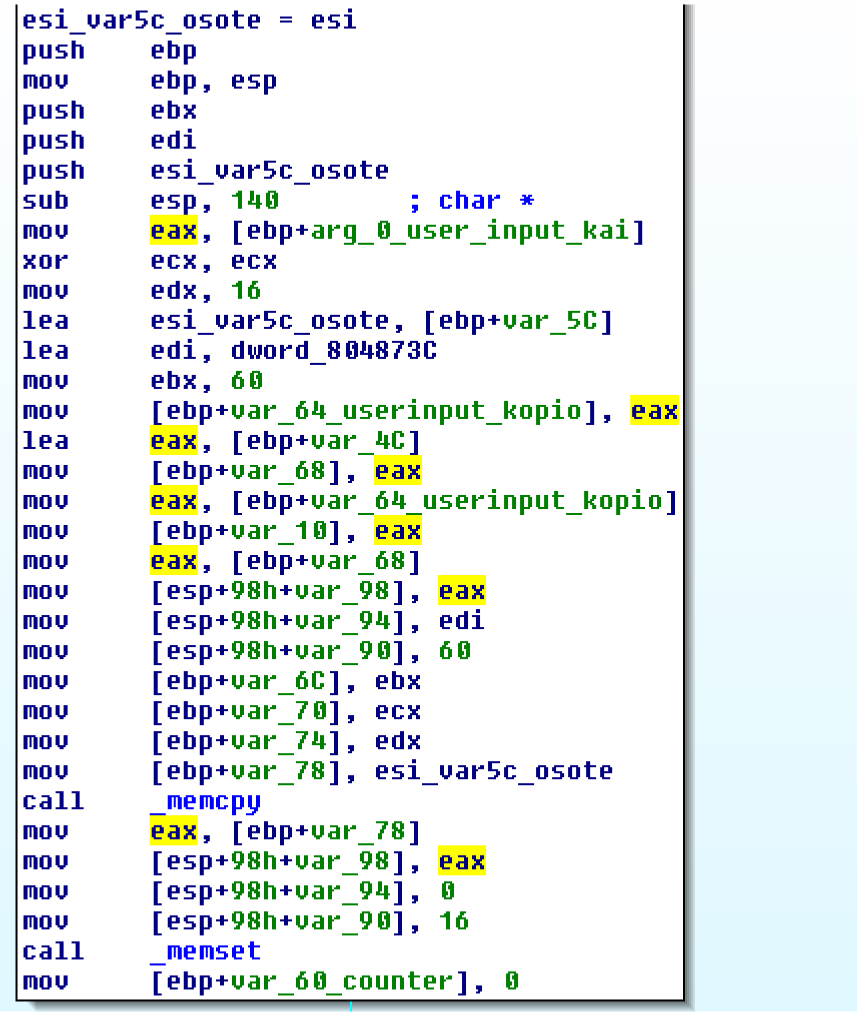
```
public main
main proc near

var_48= dword ptr -48h
var_44= dword ptr -44h
var_34= dword ptr -34h
var_30= dword ptr -30h
var_2A= dword ptr -2Ah
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4
arg_0= dword ptr  8
arg_4= dword ptr  0Ch

push    ebp
mov     ebp, esp
sub     esp, 72          ; char *
mov     eax, [ebp+arg_4]
mov     ecx, [ebp+arg_0]
lea     edx, aPassword   ; "Password: "
mov     [ebp+var_4], 0
mov     [ebp+var_8], ecx
mov     [ebp+var_C], eax
mov     [esp+48h+var_48], edx
call    _printf
lea     ecx, aS          ; "%s"
lea     edx, [ebp+var_2A]
mov     [esp+48h+var_48], ecx
mov     [esp+48h+var_44], edx
mov     [ebp+var_30], eax
call    ___isoc99_scanf
lea     ecx, [ebp+var_2A]
mov     [esp+48h+var_48], ecx
mov     [ebp+var_34], eax
call    check_password
xor     eax, eax
add     esp, 72
pop     ebp
retn
main endp
```

Figure 1: Main function.

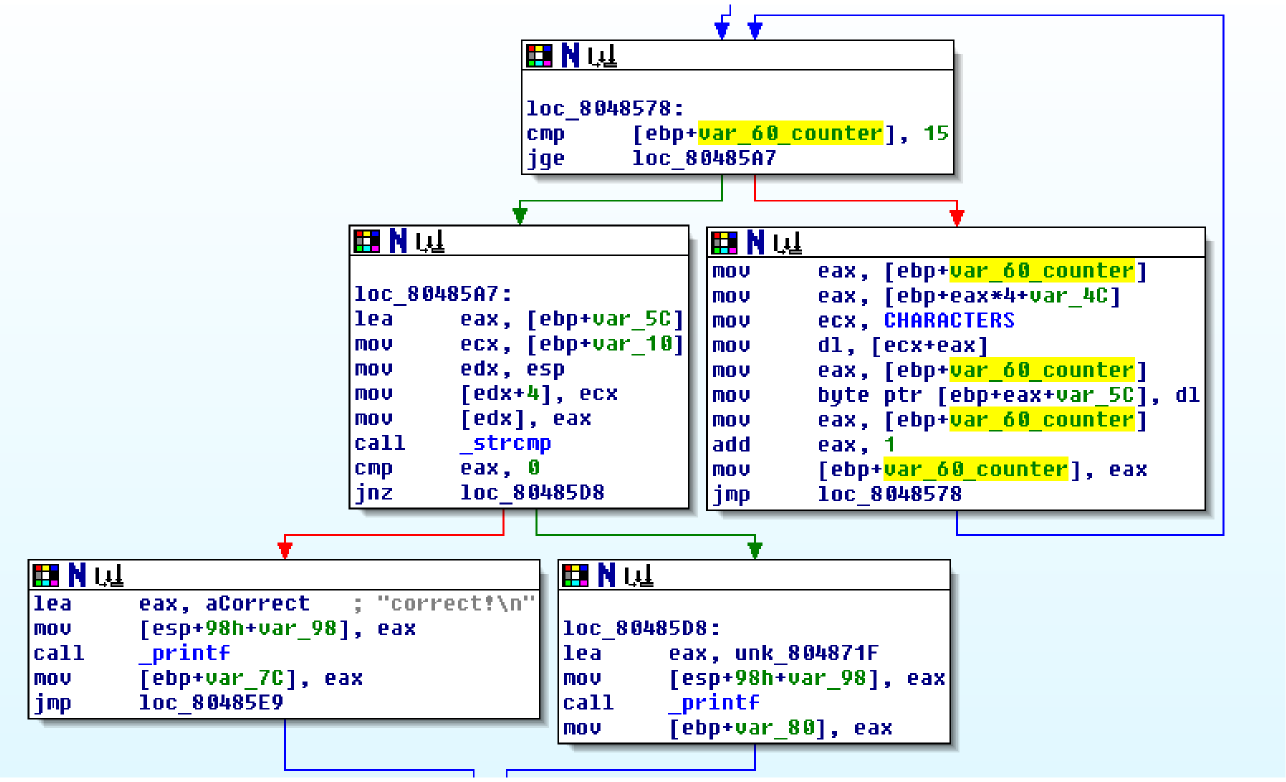
I realized quickly that I can’t find the solution here and move on to the “check_password” function:



```
esi_var5c_osote = esi
push    ebp
mov     ebp, esp
push    ebx
push    edi
push    esi_var5c_osote
sub     esp, 140         ; char *
mov     eax, [ebp+arg_0_user_input_kai]
xor     ecx, ecx
mov     edx, 16
lea     esi_var5c_osote, [ebp+var_5C]
lea     edi, dword_804873C
mov     ebx, 60
mov     [ebp+var_64_userinput_kopio], eax
lea     eax, [ebp+var_4C]
mov     [ebp+var_68], eax
mov     eax, [ebp+var_64_userinput_kopio]
mov     [ebp+var_10], eax
mov     eax, [ebp+var_68]
mov     [esp+98h+var_98], eax
mov     [esp+98h+var_94], edi
mov     [esp+98h+var_90], 60
mov     [ebp+var_6C], ebx
mov     [ebp+var_70], ecx
mov     [ebp+var_74], edx
mov     [ebp+var_78], esi_var5c_osote
call    _memcpy
mov     eax, [ebp+var_78]
mov     [esp+98h+var_98], eax
mov     [esp+98h+var_94], 0
mov     [esp+98h+var_90], 16
call    _memset
mov     [ebp+var_60_counter], 0
```

Figure 2: check_password function.

The main things here are the *_memcpy*, *_memset*, *var_4C* and *var_5C*. The function "*_memcpy*" takes three arguments destination (eax), source (edi) and number of bytes (60). The "*eax*" here is a pointer to *var_4C*, "*edi*" is pointer to "*dword_804873C*" and "60" is the number of bytes to copy so basically it takes 60 bytes from dword_804873C and copies them to "var_4C". The *_memset* function takes also three arguments: pointer (eax), int value (0) and number of bytes (16). The "eax" is pointer to var_5C so the function sets the first 16 bytes to 0 in *var_5C* address. After figuring what is going on in the picture above, I moved on and found a loop and a reason for these functions mentioned above:



```
loc_8048578:
cmp     [ebp+var_60_counter], 15
jge     loc_80485A7
```

```
loc_80485A7:
lea     eax, [ebp+var_5C]
mov     ecx, [ebp+var_10]
mov     edx, esp
mov     [edx+4], ecx
mov     [edx], eax
call    _strcmp
cmp     eax, 0
jnz     loc_80485D8
```

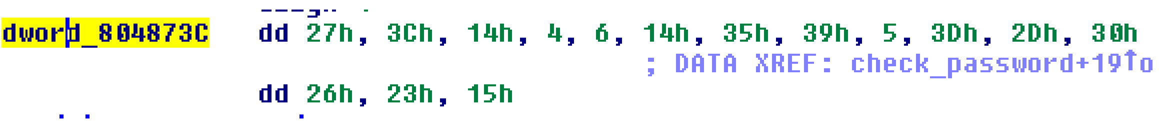
```
mov     eax, [ebp+var_60_counter]
mov     eax, [ebp+eax*4+var_4C]
mov     ecx, CHARACTERS
mov     dl, [ecx+eax]
mov     eax, [ebp+var_60_counter]
mov     byte ptr [ebp+eax+var_5C], dl
mov     eax, [ebp+var_60_counter]
add     eax, 1
mov     [ebp+var_60_counter], eax
jmp     loc_8048578
```

```
lea     eax, aCorrect   ; "correct!\n"
mov     [esp+98h+var_98], eax
call    _printf
mov     [ebp+var_7C], eax
jmp     loc_80485E9
```

```
loc_80485D8:
lea     eax, unk_804871F
mov     [esp+98h+var_98], eax
call    _printf
mov     [ebp+var_80], eax
```

Figure 3: check_password function loop.

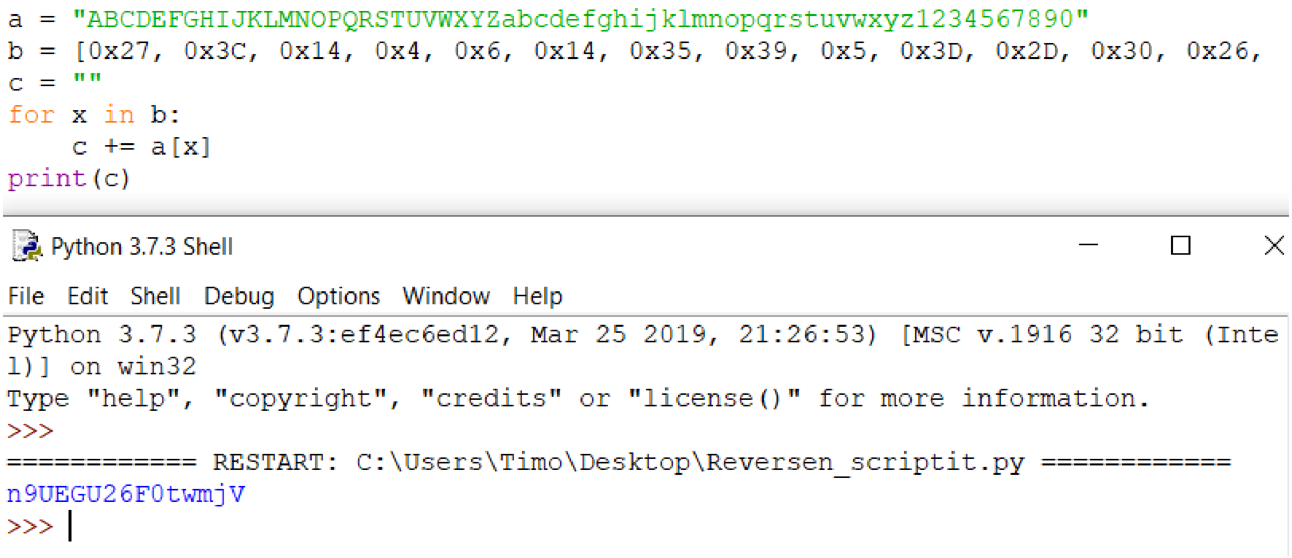
What the loop does it assigns a letter from the function “CHARACTERS” to location of *var_5C.* The letter is decided by the hex value that is located in *var_4C:*



```
dword_804873C   dd 27h, 3Ch, 14h, 4, 6, 14h, 35h, 39h, 5, 3Dh, 2Dh, 30h
                                        ; DATA XREF: check_password+19↑o
                dd 26h, 23h, 15h
```

Figure 4: dword changed to array.

so basically, the first hex value “27h” is in decimal 39 so the function takes the 39th letter from the function *CHARACTERS* which in this case is “n”. After this I started to have a clear understanding how to get the right password and moved on in the assembly code. When the counter reaches 15 it means all the bytes have been moved and the loop breaks and the function moves on to the next function. What it does next it compares the 16 bytes (0-15) with the user input and if they match it prints “correct!”. Now that I figured what the assembly code does, I used a python script to get the right password out:

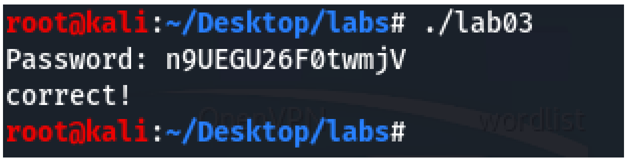


```
a = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890"
b = [0x27, 0x3C, 0x14, 0x4, 0x6, 0x14, 0x35, 0x39, 0x5, 0x3D, 0x2D, 0x30, 0x26,
c = ""
for x in b:
    c += a[x]
print(c)
```

```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Inte
l)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
============ RESTART: C:\Users\Timo\Desktop\Reversen_scriptit.py ============
n9UEGU26F0twmjV
>>> 
```

Figure 5: Python script.

Finally tested if it worked:



```
root@kali:~/Desktop/labs# ./lab03
Password: n9UEGU26F0twmjV
correct!
root@kali:~/Desktop/labs#
```

Figure 6: Right password.

it did ☺

2. Time spent

Report:	2 h
Solving the lab:	10 h
Total:	12 h