jamk

Reverse Engineering TTC6510-3002

Lab 6

Michael Herman

Student number: AB8910
Instructor: Joonatan Ovaska
Return date: 4.10.2023
Group: TIC21S1

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

Contents

Figures

1 Introduction

This report outlines the steps involved in determining the password of the lab06-ver2 32-bit ELF binary. The task was performed using the IDA64 disassembler on a VM running an instance of Kali Linux. The program is intended to obtain user input for a password. The report begins with an overview of its critical features and workflow. A technical discussion of the lab solution then proceeds. Concluding observations provide an overall impression of the task.

2 Overview

The *unk_804A008* array contains CPU instructions encoded with XOR encryption. The instruction is decrypted using a hex key of 0x99. The *mmap* function allocates memory for these decrypted instructions. Memory is configured with protected flags for reading, writing and execution. Additional flags for anonymity and privacy appear. CPU instructions are stored in the allocated memory once decrypted.

Instructions are executing using the *call* operand. This is where the password check takes place. Password verification uses XOR operations. The result of XORing character ASCII hexes should match predefined values. The program prints "correct!" if the returned values match. It prints "incorrect!" if they do not.

Password decryption takes place in a loop. It iterates through an array containing the encrypted password character by character. Each character is decrypted with XOR encryption using the key 0x99. The array is then updated with the decrypted character. The loop continues until it has processed all characters in the array or until iterator is greater than or equal to the *array_size* variable.

3 Technical discussion

3.1 Main

The primary arguments set the destination, source and size of the operation. The destination for the operation is determined by loading the address stored in the *cpu_pwd_check* variable into the

esi register using the instruction *lea esi, [ebp+cpu_pwd_check]*. This address is saved to a memory location within the *esp* stack register.

The source is specified by loading the address of the *unk_804A008* variable. This is the source argument for the *memcpy* operation. It is moved into the esi register using the *lea esi, unk_804A008* instruction. The address of the source argument is then copied to esp using the instruction *mov [esp+4], esi*.

The size of the operation is determined by moving the value 114 into a memory location. This is done with the instruction mov *dword ptr [esp+8], 72h* (114). It is the precise number of encoded instructions located in the unk_804A008 array. The *[ebp+array_size], 72h* (114) operation sets the size of the array for the loop.

Figure 1: Main function block

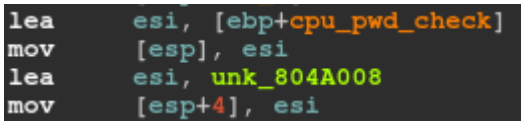
```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

mmap_return2= dword ptr -0C4h
input_pass2= dword ptr -0C0h
var_BC= dword ptr -0BCh
var_B8= dword ptr -0B8h
var_B4= dword ptr -0B4h
var_d180= dword ptr -0B0h
var_d176= dword ptr -0ACh
var_d172= dword ptr -0A8h
iterator= dword ptr -0A4h
pword_input= byte ptr -9Eh
array_size= dword ptr -80h
cpu_pwd_check= byte ptr -7Ah
var_8= dword ptr -8
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
push    esi
sub     esp, 0E4h
mov     eax, [ebp+argv]
mov     ecx, [ebp+argc]
xor     edx, edx
mov     [ebp+var_8], 0
lea     esi, [ebp+cpu_pwd_check]
mov     [esp], esi
lea     esi, unk_804A008
mov     [esp+4], esi
mov     dword ptr [esp+8], 72h ; 'r'
mov     [ebp+var_d176], eax
mov     [ebp+var_d180], ecx
mov     [ebp+var_B4], edx
call    _memcpy
mov     [ebp+array_size], 72h ; 'r'
lea     eax, [ebp+pword_input]
mov     [esp], eax
mov     dword ptr [esp+4], 0
mov     dword ptr [esp+8], 1Eh
call    _memset
lea     eax, aPassword  ; "Password: "
mov     [esp], eax
call    _printf
lea     ecx, [ebp+pword_input]
lea     edx, aS         ; "%s"
mov     [esp], edx
mov     [esp+4], ecx
mov     [ebp+var_B8], eax
call    ___isoc99_scanf
mov     [ebp+iterator], 0
```

Figure 2: Arguments

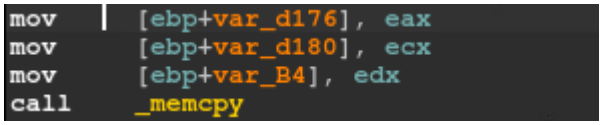


3.1.1 Misdirection

Values in the *ebp* register appear at first glance to be function arguments. Closer inspection indicates this is not the case.

Figure 2 shows operations copying values from *eax*, *ecx* and *edx* to addresses in the ebp register with offsets. The destination argument points to the address of the *argv* variable. But there is no data stored at this location. The variable does not appear elsewhere in the code. The same is true for *argc.* The size argument is derived from the EDX register. Here a self-XOR operation is performed. This effectively resets the size parameter in the register to zero.

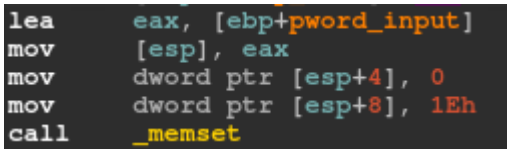
Figure 3: Empty arguments



3.1.2 User input

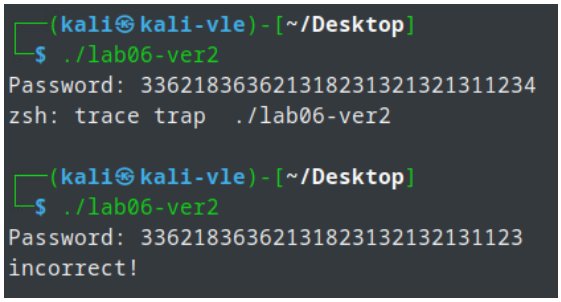
Memory is prepared for user input using *memset* arguments. The destination is set with *mov [esp], eax*. Values are set to 0 with *mov dword ptr [esp+4], 0*. The maximum number of values is set effectively by *mov dword ptr [esp+8], 1Eh* (30). This suggests an error will be thrown if this value is exceeded.

Figure 4: Setting the memory



This is confirmed when values in excess of 30 characters are entered.

Figure 5: Maximum value confirmation



The *printf* function then displays the "Password: " prompt. The arguments are then prepared for *scanf*. The *%s* format specifier is provided as an argument. This indicates user input is stored as a string. There is a redundant instruction with an unused variable immediately prior to scanf. User input is stored at the *[ebp+password_input]* memory address.

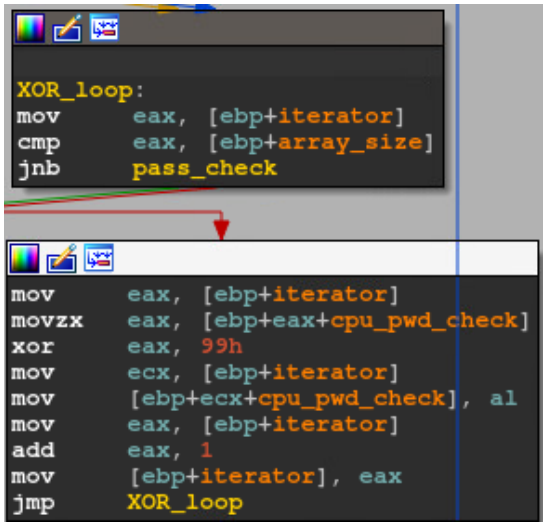
3.2 XOR loop

The iterator is initially set to 0 with the instruction *mov [ebp+iterator], 0*. Its value is then moved to the EAX register using *mov eax, [ebp+iterator]*. The value of iterator is compared to the size of the array using *cmp eax, [ebp+array_size]*. The loop is controlled by the *jnb* (jump if not below or equal) operand. The loop will continue as long as the iterator is greater than or equal to the array size (114).

This comparison uses the Carry Flag (*CF*). CF is set when there is a carry or borrow in arithmetic operations and cleared when there is no carry or borrow (Sikorski & Honig, 2012). If the CF is not set the jump is taken and loop exits. If the CF is set the jump is not taken and the loop continues.

A typical XOR operation is performed in the lower block on the array of hex values stored in unk_804A008. There are 114 elements to be XORed. The iterator is incremented by the *add eax, 1* instruction. Each array element is accessed and XORed using the key 0x99.

Figure 6: XOR loop



3.3 Password verification

3.3.1 Memory allocation

Memory is allocated in the *pass_check* block using the *mmap* function. The function establishes memory allocation parameters. This includes the following:

- The starting address (set to zero). This allows the kernel to set the address anywhere.
- The size of the allocated memory (114 bytes).
- Protection flags (read, write, and execute permissions) set access privileges.
- Flags related to memory mapping (*MAP_ANONYMOUS* and *MAP_PRIVATE*).
- The file descriptor is set to -1. This indicates there is no associated file.
- The *var_BC* variable is cleared to zero and unused. It is not stored in the register otherwise associated with mmap (eax).

The mmap function is called with these parameters to allocate memory.

Private in this context signifies the memory map is isolated from other processes. Any modifications made to it are not reflected in other processes. Changes would not be written back to that file if it is associated with memory mapping. *Anonymous* indicates memory mapping is not linked to any specific file (Sikorski and Honig, 2012). Its contents are instead initialized to zero. The file descriptor and offset arguments become irrelevant in this mode. It is worth noting that certain

implementations may necessitate setting the file descriptor to -1 even when anonymous mapping is used.

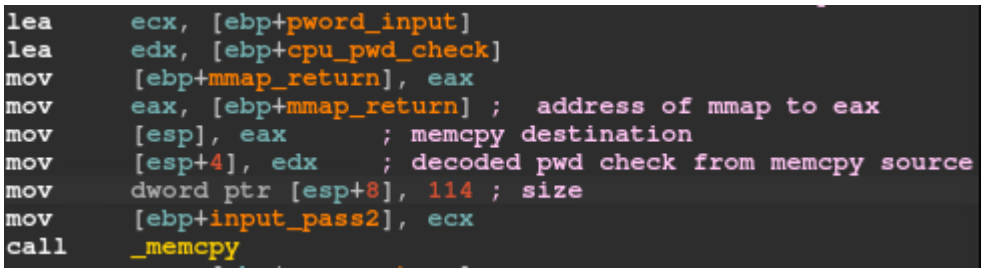
Figure 7: Memory mapping

The descriptor of the file to be mapped is typically a positive integer. The value is negative in this case because the process is executed anonymously (Sikorski & Honig, 2012). The offset parameter determines the position within the mapped area. This is specified as an offset from the beginning of the mapping for a given size. The mmap function returns a pointer to the mapped memory area. This provides for interaction with and manipulation of allocated memory. These instructions lay the groundwork for the receipt of decrypted CPU instructions from the password verification process.

3.3.2 Memory locations

The *instruction lea ecx, [ebp+password_input]* effectively loads the memory address of the password input into the ecx register. The memory address of the *lea edx, [ebp+cpu_pwd_check*] transfers the instruction to the EDX register. This contains the decrypted CPU instructions. The return value of the mmap function is copied from the EAX register using these instructions to the memory address *[ebp+mmap_return]*. The address of the mmap return value is stored in the eax register.

Figure 8: Memory copying sequence



The sequence of operations above sets up the arguments required for the memcpy function. This is used to copy the XORed values to the specified memory location.

3.3.3 Input processing

The instruction *mov eax, [ebp+mmap_return]* transfers the address stored in mmap_return to the EAX register. The destination address for memcpy is set with the *mov [esp], eax* instruction. The instruction uses the value contained in eax. This is the address returned by mmap. The value stored in edx is copied to [esp+4] with the *mov [esp+4]* instruction. The address of the decrypted (*cpu_pwd_check*) is held in edx.

This is of direct relevance to the values decrypted in the XOR block. The size of memory to be manipulated is set with *mov dword ptr [esp+8], 114*. This matches the size of the XORed CPU instructions. The address of the password input is copied from ecx prior to invoking the memcpy function. This time it is sent to [ebp+ input_pass2].

3.3.4 Input size

Input size is calculated using the *strlen* function. Arguments are set before the function is called. Another apparent attempt at obfuscation appears in this section. The *mov eax, [ebp+mmap_return]* is employed to copy the return value from the mmap function into the eax register. This step duplicates the mmap return address shortly before the strlen function call. The *mov [ebp+mmap_return2], eax* instruction preserves the return information from mmap at a specified memory address. The *mov edx, [ebp+input_pass2]* and *mov [ecx], edx* instructions calculate the size of the user input. It is these instructions which effectively prepare the argument string passed to the strlen function.

Figure 9: Input size calculation

```
mov     eax, [ebp+mmap_return]
mov     ecx, esp
mov     edx, [ebp+input_pass2]
mov     [ecx], edx
mov     [ebp+mmap_return2], eax
call    _strlen
```

3.3.5 Setup and execution

The *mov [esp], eax* instruction is pivotal in the setup and execution process. This is because it assigns the return value from the strlen function. This is utilized within the instructions themselves. The address of input_pass2 is copied by *mov eax, [ebp+password_input2]* and stored in the EAX register.

The *mov [esp+4], eax* instruction takes this address and adds it into the designated memory location for use. The *mov ecx, [ebp+mmap_return2]* then comes into play. It copies the address contained in mmap_return2 into the ecx register. The *call ecx* instruction is then executed. This effectively invokes the instructions residing at the previously established address. This sequence of operations facilitates the execution of the CPU instructions. The arguments can now be called.

Figure 10: Setup and execution

```
mov     [esp], eax
mov     eax, [ebp+input_pass2]
mov     [esp+4], eax
mov     ecx, [ebp+mmap_return2]
call    ecx
cmp     eax, 0
jnz     loc_8049357
```

3.4 Decryption

The instructions are decrypted and converted using a tool located at https://defuse.ca. Decryption requires a hex value. The first character can be found by XORing 0xA8 with 0xFE. Here the character is 56. This is equivalent to **V** in ASCII encoding. It is the first letter of the password. This is the final logic underpinning the encryption. The decryption process is presented below in its entirety. Annotations are included.

```
0:  55                  push   ebp
1:  89 e5               mov    ebp,esp
3:  52                  push   edx
4:  57                  push   edi
5:  56                  push   esi
6:  8b 45 08            mov    eax,DWORD PTR [ebp+0x8]   | strlen return
9:  8b 5d 0c            mov    ebx,DWORD PTR [ebp+0xc]   | user input
c:  ba fe ca ed fe      mov    edx,0xfeedcafe
11: 83 f8 08            cmp    eax,0x8                   | required character length
14: 74 02               je     0x18                      | 0x18 if password length correct
```

```
16: eb 50              jmp     0x68                        | else exit to position 0x68
18: 8a 03              mov     al,BYTE PTR [ebx]           | 1st character input
1a: 34 fe              xor     al,0xfe                     | XOR with 0xFE
1c: 3c a8              cmp     al,0xa8                     | XOR 0xfe to 0xa8 = 0x54; V
1e: 75 48              jne     0x68
20: 8a 43 01           mov     al,BYTE PTR [ebx+0x1]       | 2nd character
23: 34 ed              xor     al,0xed
25: 3c 84              cmp     al,0x84                     | XOR 0xed & 0x84 = 0x68; i
27: 75 3f              jne     0x68
29: 8a 43 02           mov     al,BYTE PTR [ebx+0x2]       | 3rd character
2c: 34 ca              xor     al,0xca
2e: 3c a9              cmp     al,0xa9                     | XOR 0xca & 0xa9 = 0x63; c
30: 75 36              jne     0x68
32: 8a 43 03           mov     al,BYTE PTR [ebx+0x3]       | 4th character
35: 34 fe              xor     al,0xfe
37: 3c 8a              cmp     al,0x8a                     | XOR 0xfe & 0x8a = 0x74; t
39: 75 2d              jne     0x68
3b: 8a 43 04           mov     al,BYTE PTR [ebx+0x4]       | 5th character
3e: 34 fe              xor     al,0xfe
40: 3c ce              cmp     al,0xce                     | XOR 0xfe & 0xce = 0x30; 0
42: 75 24              jne     0x68
44: 8a 43 05           mov     al,BYTE PTR [ebx+0x5]       | 6th character
47: 34 ed              xor     al,0xed
49: 3c 9f              cmp     al,0x9f                     | 0x72 = r
4b: 75 1b              jne     0x68
4d: 8a 43 06           mov     al,BYTE PTR [ebx+0x6]       | 7th character
50: 34 ca              xor     al,0xca
52: 3c b3              cmp     al,0xb3                     | 0x79 = y
54: 75 12               jne     0x68
56: 8a 43 07           mov     al,BYTE PTR [ebx+0x7]       | 8th character
59: 34 fe              xor     al,0xfe
5b: 3c df              cmp     al,0xdf                     | 0x21 = !
5d: 75 09              jne     0x68
5f: eb 00              jmp     0x61
61: b8 01 00 00 00     mov     eax,0x1                     | Exit for correct password
66: eb 05              jmp     0x6d
68: b8 00 00 00 00     mov     eax,0x0                     | Exit for incorrect password
6d: 5e                 pop     esi
6e: 5f                 pop     edi
6f: 5b                 pop     ebx
70: 5d                 pop     ebp
71: c3                 ret
```

The instruction *jnz loc_8049357* signifies the execution of the Jump Not Zero operation. eax is set to 1 if the password is correct. The cmp operation sets ZF to 1 if the comparison is equal. "[C]orrect!" is printed If the

comparison does not yield 0. A result of 0 indicates an incorrect password. The program prints "incorrect!" and exits.

Figure 11: Jump not zero

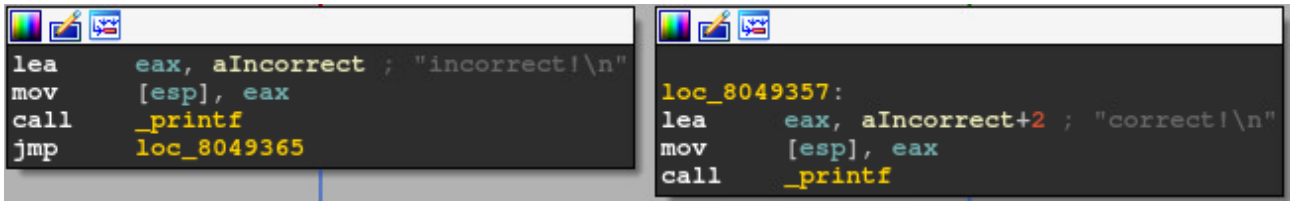
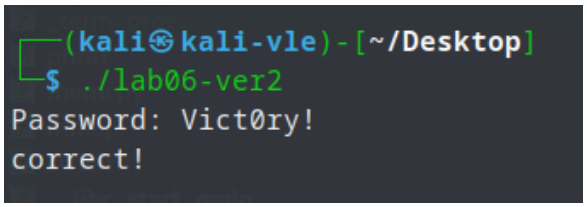


Figure 12: Solution



4 Observations

The task was considerably more advanced than previous labs. The specific nature of the password encryption was especially subtle. Solving the lab was difficult from start to finish. Grasping the encryption logic and identifying its footprint in the code felt at times completely impossible. The numerous attempts at misdirection introduced a novel dimension to the process. It was impossible to go it alone; this report is ultimately the product of group efforts. The lab provided a good sense of the range and sophistication of encryption methods available in x86 assembly. We were by all accounts pushed to the limits of our understanding. It was an elegant solution.

5 Sources

Sikorski, M., & Honig, A. (2012). *Practical malware analysis: A hands-on guide to dissecting malicious software*. No Starch Press, Incorporated.

Defuse (n.d.). *Online x86 and x64 Intel Instruction Assembler*. Retrieved October 4, 2023, from https://defuse.ca/online-x86-assembler.htm#disassembly2