

## **Reverse Engineering TTC6510-3002**

Joonatan Ovaska

A K M MAHMUDUL HAQUE  
AB0208

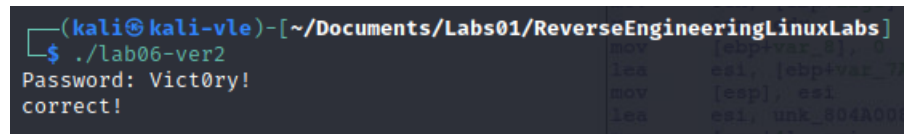
**Student number:** 2110841

### **Lab06**

Date: 04.10.2023

## First Step

- Overcoming the challenge posed by the program, initially, not much information was apparent.
- Initial observations included the presence of a loop and XOR decryption logic.
- The array **unk\_804A008** contains CPU instructions encoded with XOR encryption.
- Decryption of these instructions occurs using the hexadecimal key 0x99.
- Memory allocation via the **mmap** function is employed to store these decrypted instructions with specific flags:
  - o Protection flags: READ, WRITE, EXECUTE
  - o Additional flags: ANONYMOUS and PRIVATE
- Decrypted CPU instructions are stored within this allocated memory space.
- Execution of instructions takes place using the call operand, where the password verification process unfolds.
  - o The CPU instructions employ XOR operations to check the password validity.
  - o Correct password criteria involve XORing the ASCII hex values of characters to match a specified comparison value.
- The program evaluates the instructions' return, determining whether the entered password is correct or incorrect.
- Upon successful validation, the password is revealed: **Vict0ry!**



```
(kali@kali-vle) - [~/Documents/Labs01/ReverseEngineeringLinuxLabs]
$ ./lab06-ver2
Password: Vict0ry!
correct!
```

## Second Step

- There are indications that the programmer intentionally introduced distractions to perplex reverse engineering attempts.
- A notable observation is the manipulation of EBP registers, seemingly designed to appear as function arguments.
- Certain instructions marked as "fake" have been identified, suggesting deliberate attempts to mislead. Visual aids, such as images, have been utilized to highlight these instructions.
- **Identification of Potentially Fake Instructions:** Preceding the **memcpy** operation, three specific instructions are suspected to be fake.
- **Reasoning:**
  - o The operations involve copying values from registers EAX, ECX, and EDX to addresses in EBP registers with specified offsets.
- **Destination Argument:** Points to the address of **argv**, but this location remains unused in the code.
- **Source Argument:** Refers to **argc**, which follows a similar unused pattern.
- **Size Argument:** Derived from an XOR operation on the EDX register, resetting its value to **0**.

```

var_B4= dword ptr -0B4h
var_B0= dword ptr -0B0h
var_AC= dword ptr -0ACh
var_A8= dword ptr -0A8h
var_A4= dword ptr -0A4h
var_9E= byte ptr -9Eh
var_80= dword ptr -80h
var_7A= 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+var_7A]
mov     [esp], esi
lea     esi, unk_804A008
mov     [esp+4], esi
mov     dword ptr [esp+8], 72h ; 'r'
mov     [ebp+var_AC], eax
mov     [ebp+var_B0], ecx
mov     [ebp+var_B4], edx
call    _memcpy
mov     [ebp+var_80], 72h ; 'r'
lea     eax, [ebp+var_9E]
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+var_9E]
lea     edx, aS ; "%s"
mov     [esp], edx
mov     [esp+4], ecx
mov     [ebp+var_B8], eax
call    __isoc99_scanf
mov     [ebp+var_A4], 0

```

### Contextual Clarity:

#### - Real Arguments Setup:

- **Destination:** Initialized by `lea esi, [ebp+CPU_pwd_check]`, storing the address for CPU instructions. This address is then saved to a memory location in the ESP stack register.
- **Source:** Established by `lea esi, unk_804A008`, serving as the source argument for `memcpy`. The source argument's address is then copied to ESP register via `mov [esp+4], esi`.
- **Size:** Configured by `mov dword ptr [esp+8], 114`, representing the exact number of encoded instructions within the `unk_804A008` array.

### Rationale for Assumption:

- Certain instructions in this segment appear nonsensical, leading to wasted analysis time.
- Based on this assumption, the analysis was able to progress smoothly.

### Post-Memcpy Operations:

- **Array Size Setting:** `mov [ebp+array_size], 114` establishes the array size for the upcoming loop.
- **Memory Preparation (memset arguments setup):**
  - `lea eax, [ebp+password_input]`: Designates the destination address.

- **mov [esp], eax:** Copies the destination address to the stack register.
- **mov dword ptr [esp+4], 0:** Initializes values to 0.
- **mov dword ptr [esp+8], 30:** Specifies the count of values to be set.

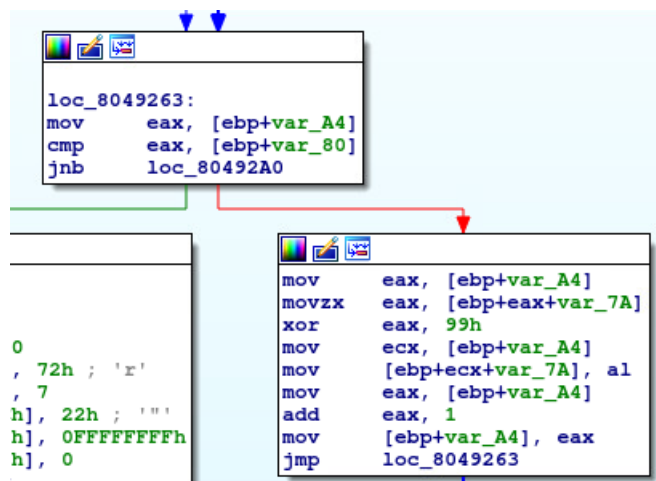
#### Verification and Validation:

- A practical test confirmed this assumption. A password length of 30 characters resulted in a trace trap error. Attempting with one character less triggered an "incorrect" message. Debugging in gdb revealed a SIGSEGV error, indicating a Segmentation Violation where the program attempted to access a restricted memory location.

### Third Step

#### User Input Prompt:

- **printf("Password: "):** Displays "Password: " on the command line.
- **Setting Scanf Arguments:**
  - **Format Specifier: " %s":** Specifies "%s" as the argument, indicating user input will be stored as a string.
- **Identification of a Fake Instruction:**
  - **Pre-Scanf Operation:** An additional suspicious instruction precedes the scanf function call.
  - **Nature:** This instruction is flagged as fake.
  - **Context:** Its purpose seems dubious, possibly serving to mislead reverse engineers.
- **Final Storage of User Input:**
  - **Memory Address: [ebp+password\_input]:** The user input is ultimately stored at this memory location.



### Fourth Step

#### Setting Up XOR Loop:

- **Iterator Initialization:**
  - **mov [ebp+iterator], 0:** Initializes the iterator to 0.
  - **Transfer to EAX: mov eax, [ebp+iterator]** moves the iterator value to the EAX register for comparison.

- **Comparison and Loop Control:**
  - **Comparing Iterator and Array Size: `cmp eax, [ebp+array_size]`:**  
Compares the iterator value with the array size (114 in this case).
  - **Loop Termination:**
    - **Jump Condition:** `jnb` (Jump if Not Below or Equal)
    - **Logic:** The loop ends if the iterator is at least equal to the array size (114).
      - **Jump Condition Details:**
        - If Carry Flag (CF) is not set, indicating no carry or overflow, the jump is taken, terminating the loop.
        - **No Jump:** If CF is set (indicating carry or overflow), the jump is not taken, allowing the loop to continue.
- **Array XOR Operation:**
- **Operation Inside `Array_XOR_block`:**
  - **XOR Operation:** Each element of the array (114 elements in total) is XORed.
  - **Iterator Increment:** `add eax, 1` increments the iterator by 1 in each loop iteration.
  - **XOR Key:** XOR operation is performed with the key **0x99**.
  - **Nature of Array Elements:** These elements represent CPU instructions previously encrypted via XOR encryption.

## Fifth Step

- **Purpose:** `mmap` creates a mapping in the virtual address space of the calling process, enabling efficient data exchange between the process and the kernel.
- **Arguments:**
  - **Address:** Starting address for the new mapping, set to zero. The kernel can allocate suitable memory location.
  - **Size:** Number of bytes to be mapped, in this case, decimal 114, representing the size of the array.
  - **Protection:** Specifies the type of access allowed. Decimal 7 is used, indicating read, write, and execute permissions. Crucial for storing and executing the password checking algorithm.
  - **Flags:** Indicates the nature of the map. In this case, it is **MAP\_PRIVATE** and **MAP\_ANONYMOUS**.
    - **MAP\_PRIVATE:** Ensures the map is not visible to other processes, and changes made are not written back to the file (if applicable).
    - **MAP\_ANONYMOUS:** Specifies that the mapping is not associated with any file, and contents are initialized to zero. File descriptor and offset arguments are ignored, requiring the file descriptor to be set to -1 in some implementations.
- **File Descriptor:** Identifies an open file within a process. Typically a positive integer, but here it is -1, signifying an anonymous map.

- **Offset:** Offset from the start of the mapping for the specified number of bytes.
- **Return Value:** `mmap` returns a pointer to the mapped area.

#### Preceding Operation (Possibly Fake):

- There is an operation before the `mmap` call where the value at EAX register is stored. Since EAX was reset earlier, this operation might be a distraction and is not utilized elsewhere in the code.
- **Return Value Handling:** The return value of `mmap` is usually stored in the EAX register.

#### Setting Arguments for `strlen` Function:

- **Operation:**
  - o `mov eax, [ebp+mmap_return]`: Copies the return value of `mmap` to EAX register.
  - o `mov [ebp+mmap_return2], eax`: Stores the `mmap` return information at the given address.
  - o `mov edx, [ebp+password_input2]`: Copies the user input address to EDX register.
  - o `mov [ecx], edx`: Sets up the argument string for the `strlen` function, calculating the size of the user input.
- **Setting Arguments for Decrypted CPU Instructions and Execution:**
  - o `mov [esp], eax`: Sets the `strlen` return as an argument for the instructions.
  - o `mov eax, [ebp+password_input2]`: Copies the address of `password_input2` to EAX register.
  - o `mov [esp+4], eax`: Copies to the address for further use.
  - o `mov ecx, [ebp+mmap_return2]`: Copies the address of `mmap_return2` to ECX register.
  - o `call ecx`: Calls the instructions from the address set in the previous step.

### Sixth Step

- The following instructions are decrypted and converted to CPU instructions using an online tool (<https://defuse.ca/online-x86-assembler.htm#disassembly2>).
- The specifics of the instructions are not provided, but it is implied that these instructions are crucial for the password checking process.

#### Disassembly:

0: 5Disassembly:

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] -> This is return of **strlen**.

9: 8b 5d 0c   mov ebx,DWORD PTR [ebp+0xc] -> This is user input.

c: ba fe ca ed fe        mov edx,0xfeedcafe

11: 83 f8 08   cmp eax,0x8 -> Needed user input length. 8 characters. If fails, password in incorrect.

14: 74 02     je 0x18 -> jump to 0x18 if password length is correct.

16: eb 50     jmp 0x68 -> This is to exit algorithm, jump position 0x68.

18: 8a 03     mov al,BYTE PTR [ebx] -> Take the first character of the password input.

1a: 34 fe     xor al,0xfe -> XOR with 0xFE.

1c: 3c a8     cmp al,0xa8 -> compared to 0xa8. So, we need to find the hex value. If we XOR 0xA8 with 0xFE we can find the hex value of the character. Character we are looking for is 56 which is "V" in ASCII list.

1e: 75 48     jne 0x68 13

20: 8a 43 01   mov al,BYTE PTR [ebx+0x1] -> Second character

23: 34 ed     xor al,0xed

25: 3c 84     cmp al,0x84 -> Applying the same logic explained earlier we get 0x69 = "i".

27: 75 3f     jne 0x68

29: 8a 43 02   mov al,BYTE PTR [ebx+0x2] -> Third character

2c: 34 ca     xor al,0xca

2e: 3c a9      cmp al,0xa9 -> 0x63 = "c"

30: 75 36      jne 0x68

32: 8a 43 03   mov al,BYTE PTR [ebx+0x3] -> Fourth character

35: 34 fe      xor al,0xfe

37: 3c 8a      cmp al,0x8a -> 0x74 = "t"

39: 75 2d      jne 0x68

3b: 8a 43 04   mov al,BYTE PTR [ebx+0x4] -> Fifth character

3e: 34 fe      xor al,0xfe

40: 3c ce      cmp al,0xce -> 0x30 = "0" zero

42: 75 24      jne 0x68

44: 8a 43 05   mov al,BYTE PTR [ebx+0x5] -> Sixth 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] -> Seventh 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] -> Eighth 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. State of EAX is check in the main.

66: eb 05     jmp 0x6d

68: b8 00 00 00 00     mov eax,0x0 -> **Exit for wrong password**

6d: 5e        pop esi

6e: 5f        pop edi

6f: 5b        pop ebx

70: 5d        pop ebp

71: c3        ret

- Following the password check operation, the program returns to the main routine.
- **cmp eax, 0:** Compares the value in the EAX register with 0. In the earlier instructions, if the password was correct, EAX was set to 1.
- **jnz loc\_8049357:** This instruction translates to "Jump Not Zero." If the comparison in the previous step results in EAX not being zero (indicating a correct password), the Zero Flag (ZF) is set to 1. If ZF is not zero, the program jumps to the memory address **loc\_8049357**.
- Depending on the outcome of this comparison, either "**correct!**" or "**incorrect!**" is printed to the console.

Topic	Time
Lab06	15 hours
Report writing	4 hours