jamk

# Reverse Engineering

## Lab 06

Timo Lehosvuo, M3426

Report
Reverse Engineering, Marko Silokunnas
7.3.2021
ICT

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

# Sisällys

**1 Lab 06 ..... 3**

**2. Time spent ..... 7**

# 1 Lab 06

Unlike the previous labs there were a lot going on in the main function, but the main point is the "_memcpy" where it copies "109" bytes from the location of "byte_8048810" and at the end where it sets "var_B0" to "0". It also sets "30" bytes to "0" in location of "var_AA" but this was not essential to solve the lab:


Figure 1: Main function.

Afterwards it jumps to "checker" that checks if the value of "eax" is 109 and if not, it goes to a loop:

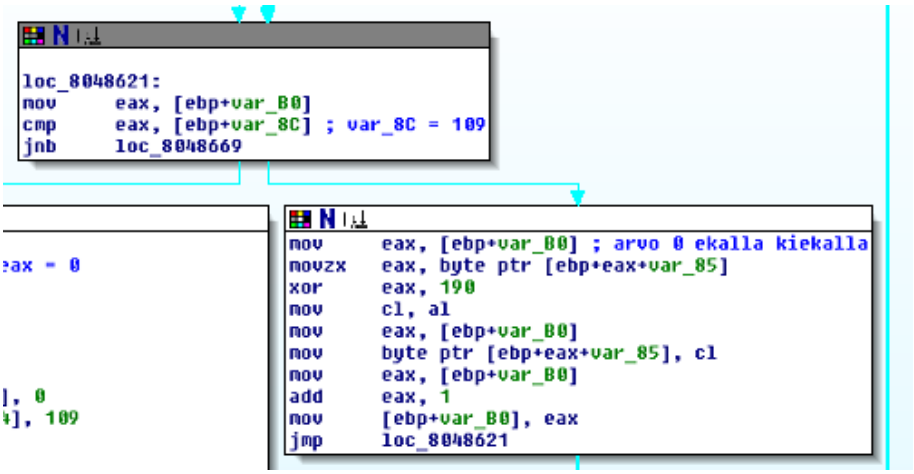

Figure 2: check_serial function.

The loop takes the byte indicated by the value of “eax” (in the first run its 0) from the location of “var_85” and it XORs this with 190. The output of this is then put to 8-bit data register “al” which is moved to “cl” and later in the code it is moved back to the location of “var_85” and to the same byte index indicated by the value of “eax”. Basically, this loop XORs the values in “var_85” and puts the XORed values back to “var_85”. Value of “eax” is incremented by one and the loop starts over until “eax” is “109”. When the value of “eax” is “109” the code moves to a different section where a lot of interesting things happen:


Figure 3: check_serial function loops.

The functions that I needed to solve to solve the lab are “_mmap”, “_membcpy” and I needed to figure out what is inside “ecx” at the end where it is called. The “_mmap” function maps files or devices into memory and in this case it takes 6 arguments: addr (0) tells the starting address of the mapping, length (109) tells the length of the mapping, prot (7) describes memory protection, flags (34) this tells whether updates to the mapping are visible to other processes and are they carried to underlying files, file descriptor (4294967295) and offset (0) tells where the “length” starts

("mapping starts at 0 with length of 109"). This new mapped area is used at the function "_memcpy" where it copies "109" bytes from "var_85" (XORed content from earlier) to the mapped are pointed by "mmap" (mmap returns a pointer to the mapped are). Now that I started to have an understanding what's going on, I still did not know what the content inside "mmap" is used for. I knew that the code calls "ecx" which holds the result of "_memcpy" so I figured that it had to hold a function inside. I went and copied the values inside "byte_8048810" and XORed them using cyberchef:

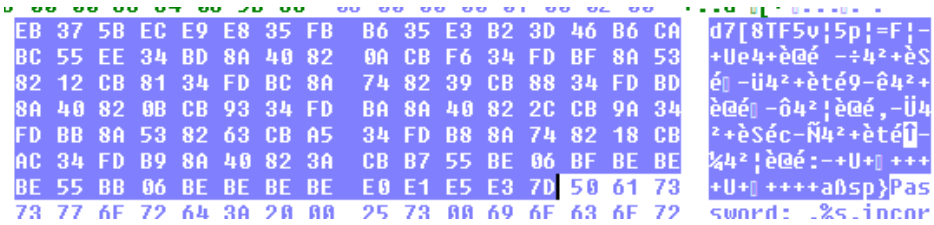


Figure 4: HEX values.

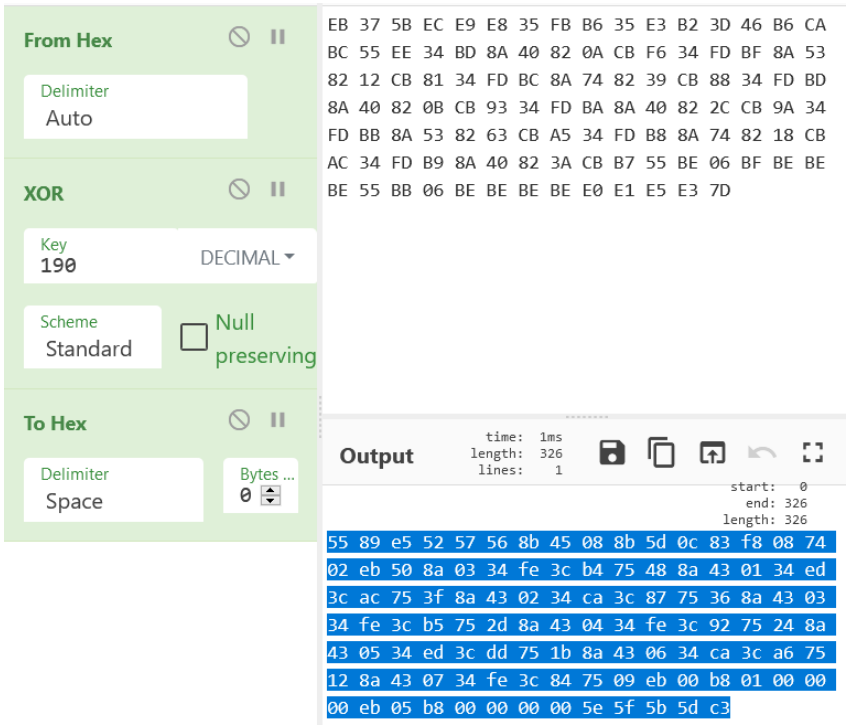


Figure 5: HEX after XOR.

Now that I had the content of the "mmap" memory I had to disassemble this somehow, so I googled "online disassembler" and use the first one I found (https://onlinedisassembler.com/static/home/index.html):

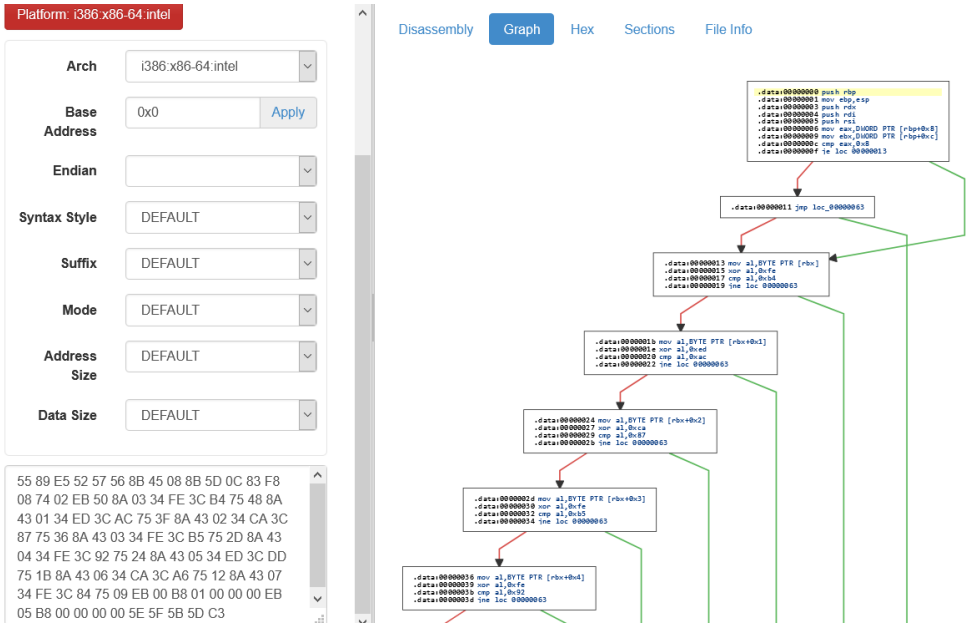


Figure 6: Online disassembler.

I changed the “Arch” to i386:x86-64:intel and started to look at the result. The code starts with some “push”, “mov” and “cmp” instructions until it jumps to a loop and the loop interested me the most so I shifted my focus there. I realised that it did some XORing and just like in the previous labs I “reverse” XORed the inputs in each loop and got the result “JAMKl0lz”:

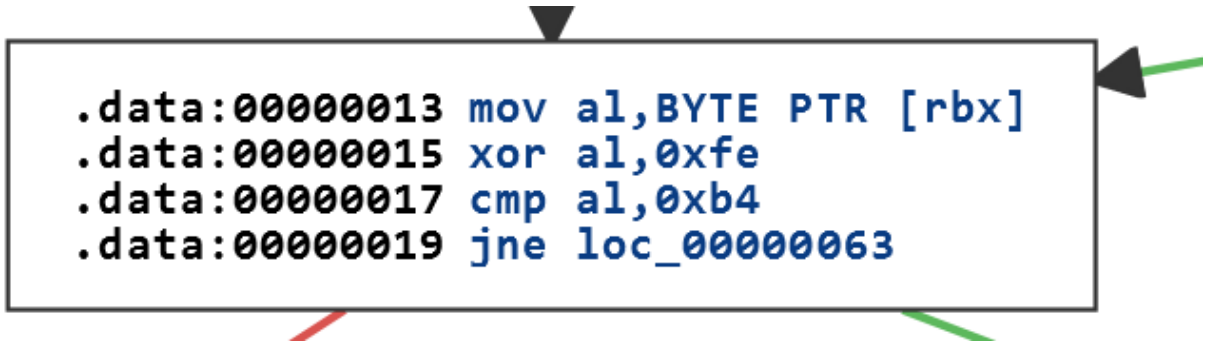


Figure 7: Loop.

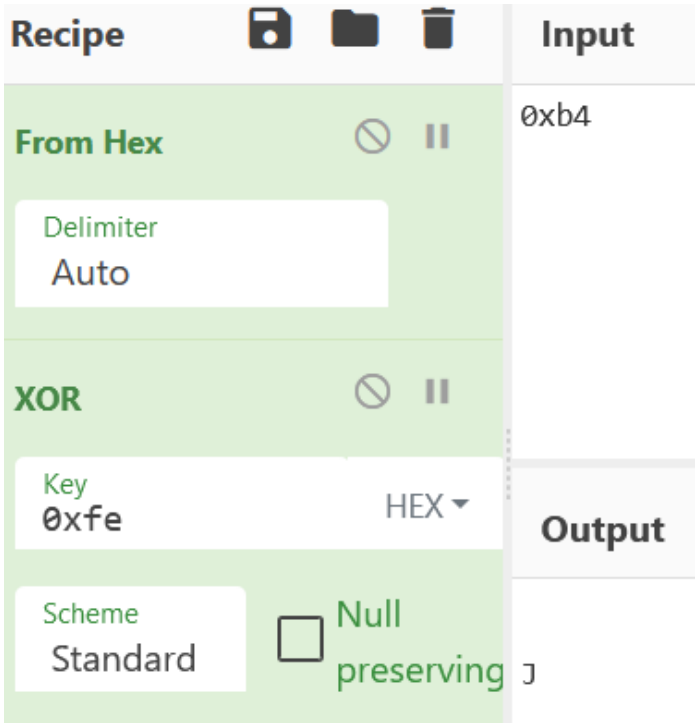


Figure 8: Single loops input XOR

Tested the possible password and it worked:

```
root@kali:~/Desktop/labs# ./lab06
Password: JAMKl0lz
correct!
root@kali:~/Desktop/labs#
```

Figure 9: Password test.

## 2. Time spent

| | |
|---|---|
| Report: | 1.5 h |
| Solving the lab: | 7 h |
| Total: | 8.5 h |