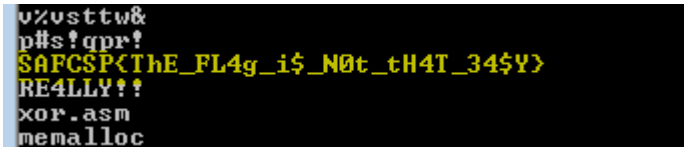
Introduction:

During the Kaizen Arabia CTF, I found interesting challenge (Little F0rt) under Reverse Engineering category, which request for a flag starting with the format “SAFCSP{}” in a given executable called “Bombs_Landed”

Step #1: Strings

First thing of course starting with the strings of the executable file, I found a string with thet given flag format:

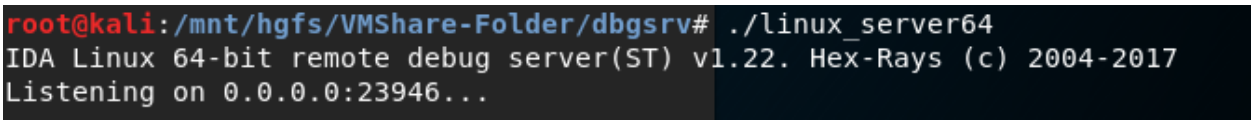


But unfortunately it was not the correct flag.

Step #2: Type of the file

To reverse executable file, first we need to know its type, using Kali command I found that it is ELF 64-bit executable file for Linux Operating System:

And for that I used IDA Pro Linux server to start debugging it remotely.

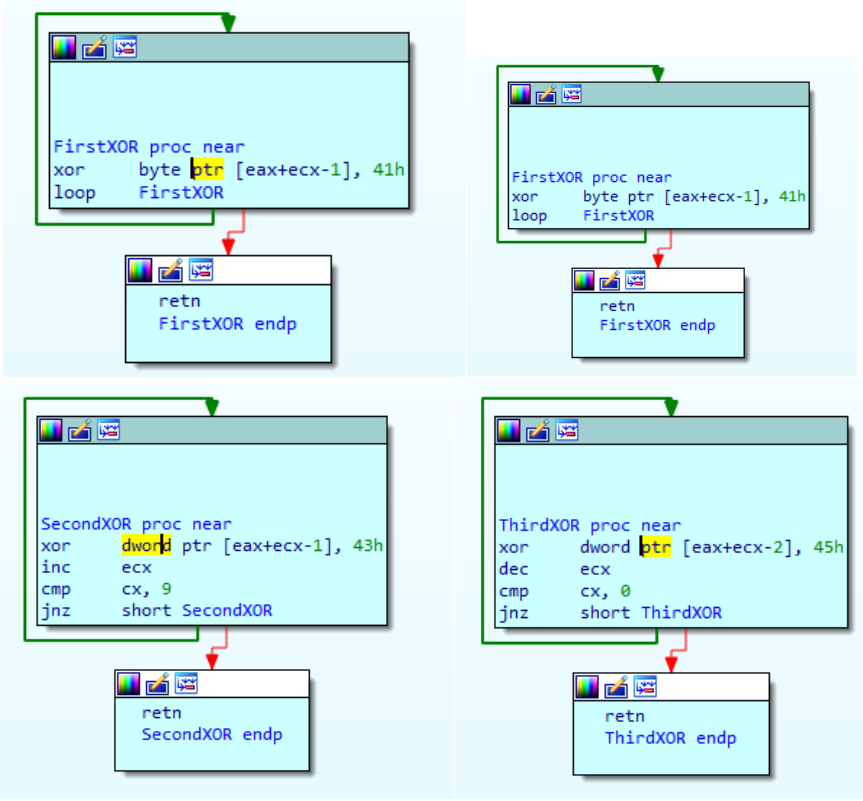


And then from Windows machine, I used IDA Pro remotely to execute Bombs_Landed executable and start debugging it.

Step #3: Debugging:

I Did some tracing with the execution until if found a strange behavior, there was four functions with name end with XOR, and after each call for those functions it overwrites one of the parameters passed to that function, I realized that it could be manipulating the content of that parameter and then overwrite the new result (which is what every function of those functions did).

The following graph for each one of these functions:



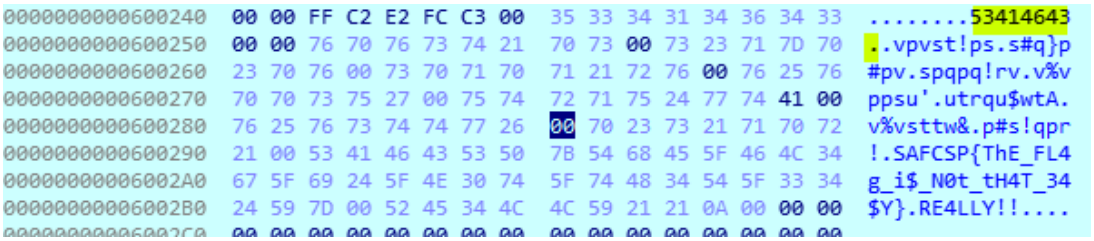
What is important is not what the functions do, but what are the results of them, so I set a breakpoint after each call of these functions:

```
; Segment permissions: Read/Execute
_text           segment para public 'CODE' use64
                assume cs:_text
                ;org 400080h
                assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing



                public _start
_start          proc near
                xor     eax, eax
                xor     ecx, ecx
                mov     cx, 9
                mov     eax, offset _GLOBAL_OFFSET_TABLE_
                call    FirstXOR
                mov     _GLOBAL_OFFSET_TABLE_, 0
                mov     dword_60024C, 0
                xor     eax, eax
                xor     ecx, ecx
                mov     eax, offset virtualalloc
                call    SecondXOR
                mov     virtualalloc, 0
                mov     dword_600256, 0
                xor     eax, eax
                xor     ecx, ecx
                mov     cx, 9
                mov     eax, offset heapfree
                call    ThirdXOR
                mov     heapfree, 0
                mov     dword_60025F, 0
                xor     eax, eax
                xor     ecx, ecx
                mov     cx, 9
                mov     eax, offset INFINITE
                call    ThirdXOR
                mov     INFINITE, 0
                mov     dword_600268, 0
                xor     eax, eax
                xor     ecx, ecx
                mov     eax, offset writememory
                call    SecondXOR
                mov     writememory, 0
                mov     dword_600271, 0
                xor     eax, eax
                xor     ecx, ecx
                mov     cx, 9
                mov     eax, offset heapalloc
                call    FirstXOR
                mov     heapalloc, 0
                mov     dword_60027A, 0
                xor     eax, eax
                xor     ecx, ecx
                mov     eax, offset waitforsingelobject
                call    SecondXOR
                mov     waitforsingelobject, 0
                mov     dword_600284, 0
                xor     eax, eax
                mov     eax, offset remotethread
                call    FinalXOR
                mov     remotethread, 0
                mov     dword_60028D, 0
                mov     eax, 1
                mov     edi, 1 ; fd
                mov     rsi, offset play ; "RE4LLY!!\n"
                mov     edx, 9 ; count
                syscall ; LINUX - sys_write
                mov     eax, 3Ch
                mov     edi, 0 ; error_code
                syscall ; LINUX - sys_exit
_start          endp
```

And since the executable overwrite every parameter it passes to the function, using the Hex view on the address 0x0000000000600248 (which is the address of _GLOBAL_OFFSET_TABLE_):

```
0000000000600240  00 00 FF C2 E2 FC C3 00  74 72 75 70 75 77 75 72  ........trupuwur
0000000000600250  41 00 76 70 76 73 74 21  70 73 00 73 23 71 7D 70  A.vpvst!ps.s#q}p
0000000000600260  23 70 76 00 73 70 71 70  71 21 72 76 00 76 25 76  #pv.spqpq!rv.v%v
0000000000600270  70 70 73 75 27 00 75 74  72 71 75 24 77 74 41 00  ppsu'.utrqu$wtA.
0000000000600280  76 25 76 73 74 74 77 26  00 70 23 73 21 71 70 72  v%vsttw&.p#s!qpr
0000000000600290  21 00 53 41 46 43 53 50  7B 54 68 45 5F 46 4C 34  !.SAFCSP{ThE_FL4
00000000006002A0  67 5F 69 24 5F 4E 30 74  5F 74 48 34 54 5F 33 34  g_i$_N0t_tH4T_34
00000000006002B0  24 59 7D 00 52 45 34 4C  4C 59 21 21 0A 00 00 00  $Y}.RE4LLY!!....
00000000006002C0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

From this the only thing that is interesting the fake flag and word “RE4LLY!!” but when we run the executable and we stop on the first breakpoint after FirstXOR, we can see that the content of the Hex view changed:



Here we can see the changed highlighted “**53414643**”, and then the next instruction “**mov _GLOBAL_OFFSET_TABLE_, 0**” will overwrite it with zeros.

All next functions do the same, XOR the content of the given parameter and then after the call it overwrite it. I collected the result of every function before overwriting, the results combined as following:

- **53414643**
- **53507b30**
- **6f485f53**
- **65454d73**
- **5f53306d**
- **45304e65**
- **5f50774e**
- **5f6d457d**

From what it looks like it seems an ASCII test in HEX format, and using tools or online web sites (like.rapidtables.com) to convert HEX to ASCII, the result will be the flag which is:

SAFCSP{0oH_SeEMs_S0mE0Ne_PwN_mE}

Note: some results of the functions will be more than 4 bytes; you need take only the last four bytes of the edited bytes.

Conclusion:

This idea of this challenge, it writes the Hex value of the flag on the memory and then overwrite it, and it use XOR to not be clear on the strings result and confuse with the fake flag. Also, we can see that it uses some system function’s names (like *heapalloc*, *writememory*) so that you may not notice that it contains the important data. Another way to solve this without debugging is by extracting the content of the functions parameter and perform manual XOR instead of making the debugger execute it for you.

I hope this explain how I solved this challenge.