UNIVERSITY OF THE FRASER VALLEY

COMP 325 FINAL PROJECT

Fall 2021

Submitted to -
Prof. Talia Q

Submitted By -
Shubham Rishi
Aayush Damathia
Khushpreet Singh
Jaskaran Singh Khangura

Table of Contents

Introduction

The Final report for the Project reverse-challenge which was found to be a double xor encrypted file in Assembly language which we tried to decrypt using different means and websites such as xor decoder, kali Linux to find the password. This was found to be a fun project and included a lot of observations of the assembly language and was broken down into numerous sections for decryption purposes.

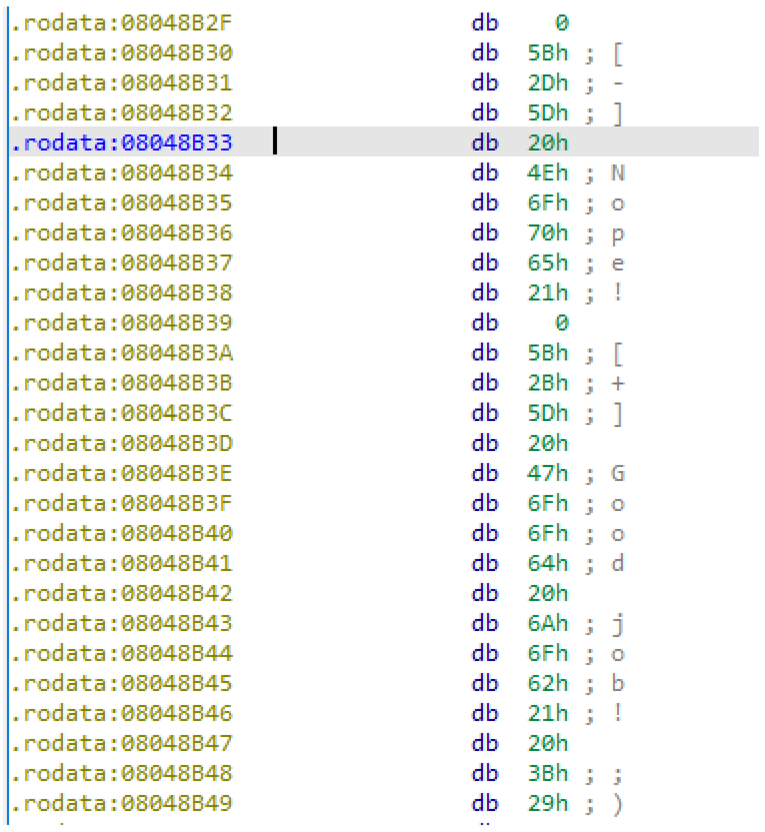
Plan Used

The basic plan we used was to first decode what each and every string in the code was doing, getting called, moved, added, subtracted, etc which we did manually by writing it in easy words. Apart from this we have used IDA Pro to check the IDA view for any clues and Hex view to find any string which can include any tiny bit of detail, afterwards we loaded the code in a Linux machine to understand what kind of output will show up in which kind of conditions(more info below), throughout the code we traced “xor” as it is a beloved encryption and decryption method used by hackers.Throughout the decryption process se have put our work force in different section debugging and exploration in the expectation of finding the password.

Methods used:

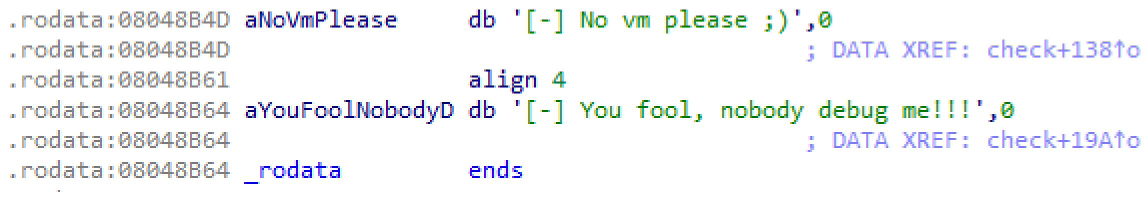
<u>Beginning Phase</u>

The first thing we did was open IDA and execute the program to see if there were any indications that may aid us. After a few minutes of scouring the code, we discovered the following:



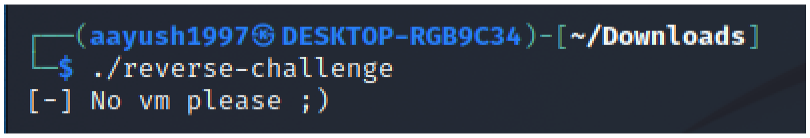
```
.rodata:08048B2F                 db    0
.rodata:08048B30                 db  5Bh ; [
.rodata:08048B31                 db  2Dh ; -
.rodata:08048B32                 db  5Dh ; ]
.rodata:08048B33                 db  20h
.rodata:08048B34                 db  4Eh ; N
.rodata:08048B35                 db  6Fh ; o
.rodata:08048B36                 db  70h ; p
.rodata:08048B37                 db  65h ; e
.rodata:08048B38                 db  21h ; !
.rodata:08048B39                 db    0
.rodata:08048B3A                 db  5Bh ; [
.rodata:08048B3B                 db  2Bh ; +
.rodata:08048B3C                 db  5Dh ; ]
.rodata:08048B3D                 db  20h
.rodata:08048B3E                 db  47h ; G
.rodata:08048B3F                 db  6Fh ; o
.rodata:08048B40                 db  6Fh ; o
.rodata:08048B41                 db  64h ; d
.rodata:08048B42                 db  20h
.rodata:08048B43                 db  6Ah ; j
.rodata:08048B44                 db  6Fh ; o
.rodata:08048B45                 db  62h ; b
.rodata:08048B46                 db  21h ; !
.rodata:08048B47                 db  20h
.rodata:08048B48                 db  3Bh ; ;
.rodata:08048B49                 db  29h ; )
```

It is quite visible that these strings provide some insight into whether or not the password we entered is correct. We also discovered that there are more strings, which we don't know what they're for at the moment:



```
.rodata:08048B4D aNoVmPlease     db '[-] No vm please ;)',0
.rodata:08048B4D                                         ; DATA XREF: check+138↑o
.rodata:08048B61                 align 4
.rodata:08048B64 aYouFoolNobodyD db '[-] You fool, nobody debug me!!!',0
.rodata:08048B64                                         ; DATA XREF: check+19A↑o
.rodata:08048B64 _rodata         ends
```

The next step would be to execute the code on Linux, for this project we specifically used Kali Linux. To launch the application, we must first chmod u+x the file to grant it full permissions, after which we can run it using the usual command. After running the application, we can see that it can detect whether we are running it on a virtual machine environment or a physical Linux computer. Unfortunately, when we ran it on our virtual machine, we got the following message.

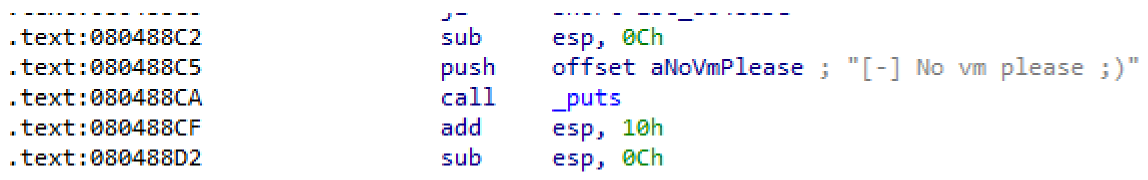


```
┌──(aayush1997㉿DESKTOP-RGB9C34)-[~/Downloads]
└─$ ./reverse-challenge
[-] No vm please ;)
```

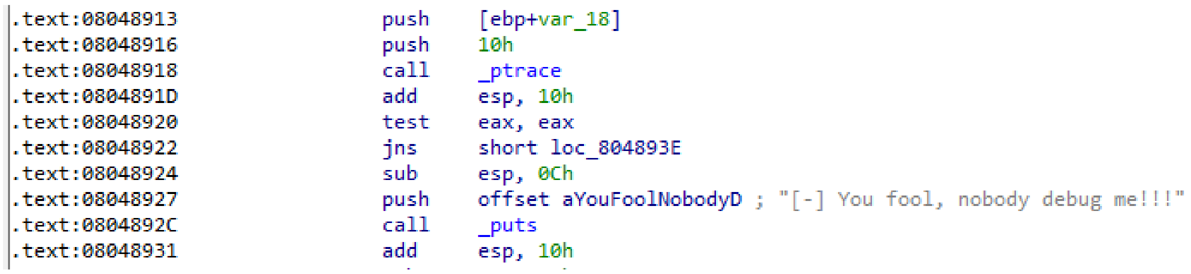
So now after configuring a bit about the program in the terminal using the virtual environment the next step is to examine the program using IDA.

<u>Code Analysis</u>

So it was time to return to IDA and investigate some big functions. So first we started with **check** method, which seems to be responsible for the entire program's output. As we can see below, it was simple to locate the output when we use a virtual machine(“[-] No vm please ;)”) and when we try to debug it(“[-] you fool, nobody debug me!!!”) in the check function:

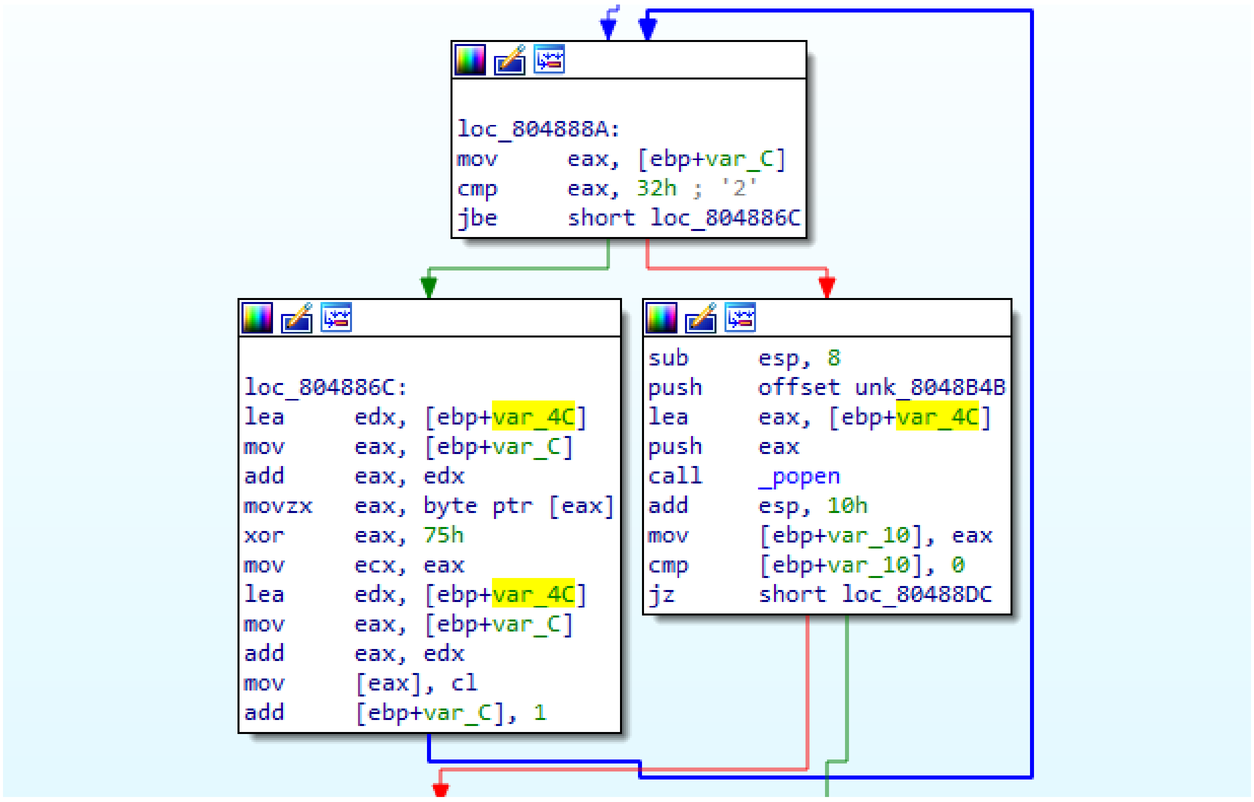


```
.text:080488C2                 sub     esp, 0Ch
.text:080488C5                 push    offset aNoVmPlease ; "[-] No vm please ;)"
.text:080488CA                 call    _puts
.text:080488CF                 add     esp, 10h
.text:080488D2                 sub     esp, 0Ch
```



```
.text:08048913                 push    [ebp+var_18]
.text:08048916                 push    10h
.text:08048918                 call    _ptrace
.text:0804891D                 add     esp, 10h
.text:08048920                 test    eax, eax
.text:08048922                 jns     short loc_804893E
.text:08048924                 sub     esp, 0Ch
.text:08048927                 push    offset aYouFoolNobodyD ; "[-] You fool, nobody debug me!!!"
.text:0804892C                 call    _puts
.text:08048931                 add     esp, 10h
```

Currently, We can consider this thing just a tiny clue towards our exploration. before these labels that we discussed above and just after the function's start, this function (check) executes roughly mov 32h (50 in ASCII) instructions, and then the function executes a loop, as seen below:



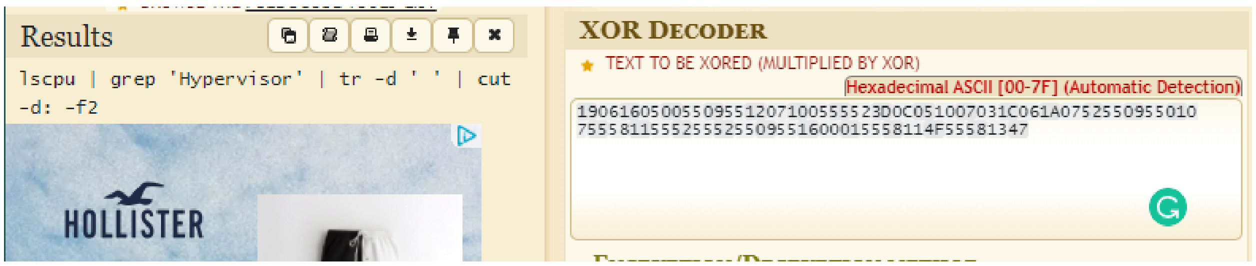
```
loc_804888A:
mov     eax, [ebp+var_C]
cmp     eax, 32h ; '2'
jbe     short loc_804886C
```

```
loc_804886C:
lea     edx, [ebp+var_4C]
mov     eax, [ebp+var_C]
add     eax, edx
movzx   eax, byte ptr [eax]
xor     eax, 75h
mov     ecx, eax
lea     edx, [ebp+var_4C]
mov     eax, [ebp+var_C]
add     eax, edx
mov     [eax], cl
add     [ebp+var_C], 1
```

```
sub     esp, 8
push    offset unk_8048B4B
lea     eax, [ebp+var_4C]
push    eax
call    _popen
add     esp, 10h
mov     [ebp+var_10], eax
cmp     [ebp+var_10], 0
jz      short loc_80488DC
```

In the above, we can see that the function var_c acts as the loop counter and we can see the code loops between itself and loc_804886C 32h times which is 50 times in ASCII value and this number originates from the mov instructions we referred to in the previous paragraph.

Looking above, we can see that the loop uses 75h to execute an XOR operation on all of its 50 (19h,6,16h,5,0,55h,9,55h,12h,7,10h,5,55h,52h,3Dh,0Ch,5,10h,7,3,1Ch,6,1Ah,7,52h,55h,9,55h,1,7,55h,58h,11h,55h,52h,55h,52h,55h,9,55h,16h,0,1,55h,58h,11h,4Fh,55h,58h,13h,47h) values individually. This decryption was easy enough to solve due to the previous experience gained through this course and to decrypt this we have used an online tool “XOR Decoder” which can be found :“https://www.dcode.fr/xor-cipher”.


Results
XOR DECODER
TEXT TO BE XORED (MULTIPLIED BY XOR)
Hexadecimal ASCII [00-7F] (Automatic Detection)
HOLLISTER

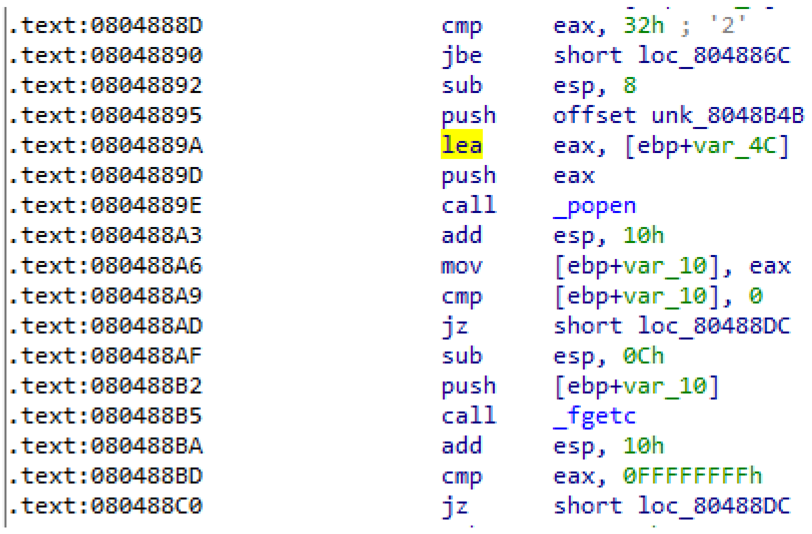
```
lscpu | grep 'Hypervisor' | tr -d ' ' | cut
-d: -f2
```

```
19061605005509551207100555523D0C051007031C061A0752550955010
7555811555255525509551600015558114F55581347
```

The Value Decoded from it found to be `lscpu | grep 'Hypervisor' | tr -d ' ' | cut -d: -f2.`

So we can assume that this application clearly indicates whether or not we are utilizing a virtual machine. Though we were able to decode that still we couldn't figure out what was going on in the loop that we have mentioned above. So to evaluate it this time we have to go back and try to figure out what occurs once the loop ends.

And we can see what occurs when the loop is closed down below:



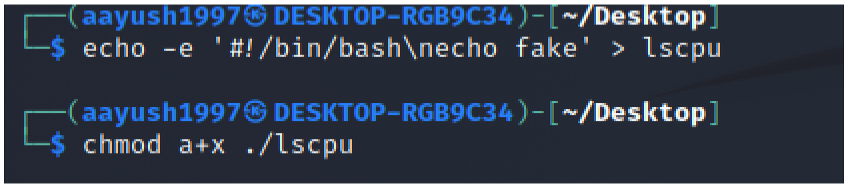
```
.text:0804888D                 cmp     eax, 32h ; '2'
.text:08048890                 jbe     short loc_804886C
.text:08048892                 sub     esp, 8
.text:08048895                 push    offset unk_8048B4B
.text:0804889A                 lea     eax, [ebp+var_4C]
.text:0804889D                 push    eax
.text:0804889E                 call    _popen
.text:080488A3                 add     esp, 10h
.text:080488A6                 mov     [ebp+var_10], eax
.text:080488A9                 cmp     [ebp+var_10], 0
.text:080488AD                 jz      short loc_80488DC
.text:080488AF                 sub     esp, 0Ch
.text:080488B2                 push    [ebp+var_10]
.text:080488B5                 call    _fgetc
.text:080488BA                 add     esp, 10h
.text:080488BD                 cmp     eax, 0FFFFFFFFh
.text:080488C0                 jz      short loc_80488DC
```

This code invokes the **popen** function, of which details may be found online, and is used to start pipe streams to or from a processor. It also calls the function **fgetc** and compares the return value when the stream is opened. It was simple for us to figure out what was going on at the command: `lscpu | grep 'Hypervisor' | tr -d ' ' | cut -d: -f2` If the command returns a value, it will be sent to the function, which will show "No vm please ;)" before exiting. If the command produces no output, it shuts the stream and exits.

Closer look to NoVm

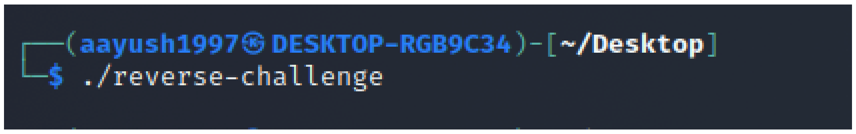
Now after all that effort we finally know how the program configures if we are using a virtual machine and close it. As we know the various commands the loop runs (discussed above), it was easy to tell that the program depended on the lscpu command to determine if the CPU was present or not. But what if there is a filename lscpu already present in the virtual machine?. so this time we tried to find a way to bypass it and then run the program. So the best solution was to get to our virtual machine and create a new lscpu file. So we did that on our Kali Linux.

Which is demonstrated below:



```
┌──(aayush1997㉿DESKTOP-RGB9C34)-[~/Desktop]
└─$ echo -e '#!/bin/bash\necho fake' > lscpu

┌──(aayush1997㉿DESKTOP-RGB9C34)-[~/Desktop]
└─$ chmod a+x ./lscpu
```

After that, we export the directory inside the PATH variable thus completing the job. Now we tried running the program after taking away permissions from everyone and the following was the output which was delayed like it took a second or two to execute.

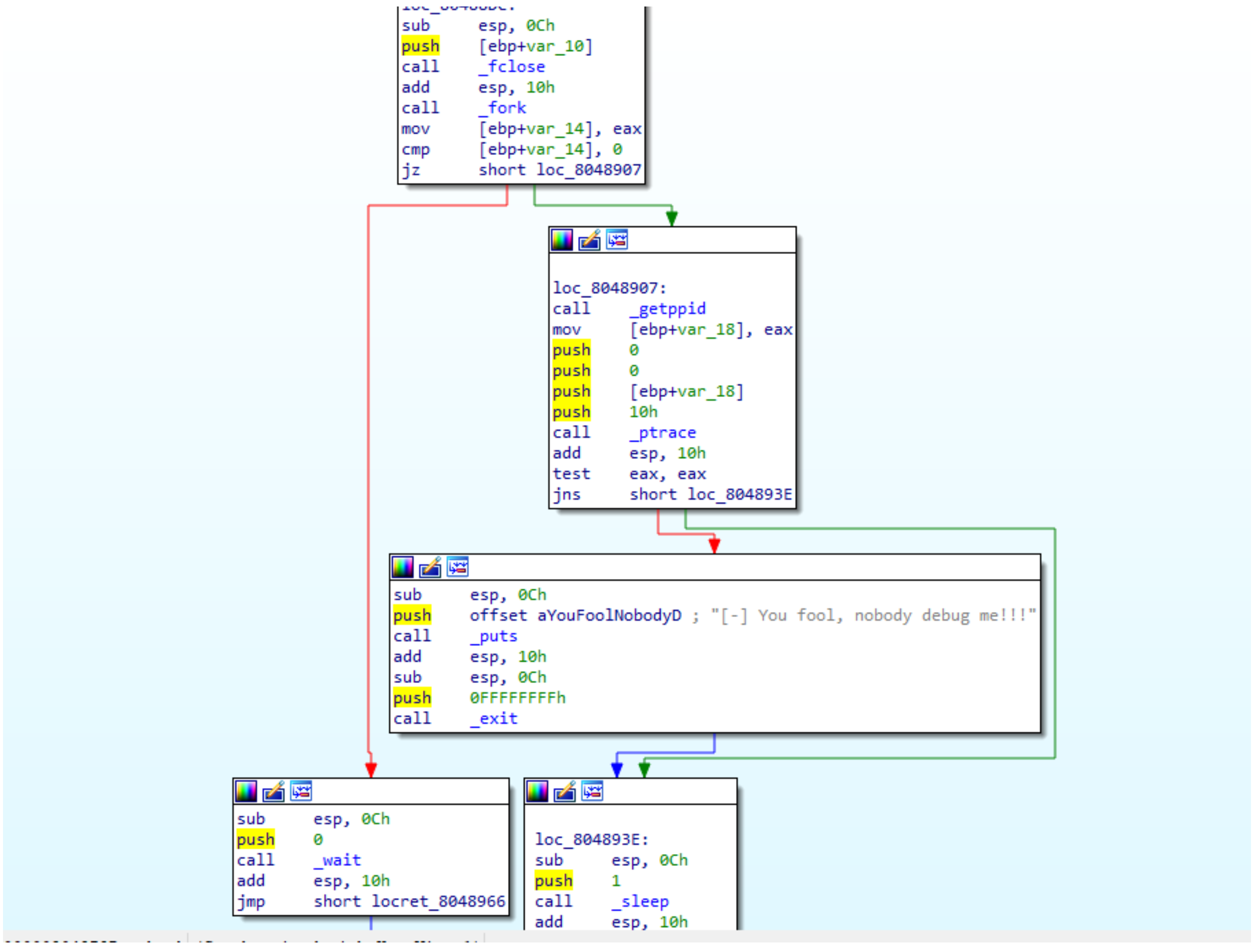


```
┌──(aayush1997㉿DESKTOP-RGB9C34)-[~/Desktop]
└─$ ./reverse-challenge
```

Now the reverse-challenge file doesn’t give any virtual machine message so it must be bypassed. So now it was time to go to the next step which was the debug message (aYouFoolNobodyD).

Taking care of Debug

After changing from a virtual machine to an actual Linux Machine, what we end up with was that there was still a lot to get rid of before starting the decryption process, so when we tried running the program we didn’t get the no vm error, but this time we got “You fool Nobody Debug me” as an error. So now it was time to understand how the anti debugger detects that debugging is going on, as we can see this function is pretty simple and easy to trace using IDA it just uses standard library functions which are **fork** which creates a new process by duplicating the calling process which is called the child which the same duplicate of the calling process which is called parent, and the next function is the **getppid** which helps get the parent processes ID and **wait** which is the function which waits for the child process to stop. And the next function was **ptrace** which at this moment doesn’t seem that interesting but this function helps in observing and controlling the execution of another process. Below we can see the anti debugger IDA breakdown:



```
sub     esp, 0Ch
push    [ebp+var_10]
call    _fclose
add     esp, 10h
call    _fork
mov     [ebp+var_14], eax
cmp     [ebp+var_14], 0
jz      short loc_8048907
```

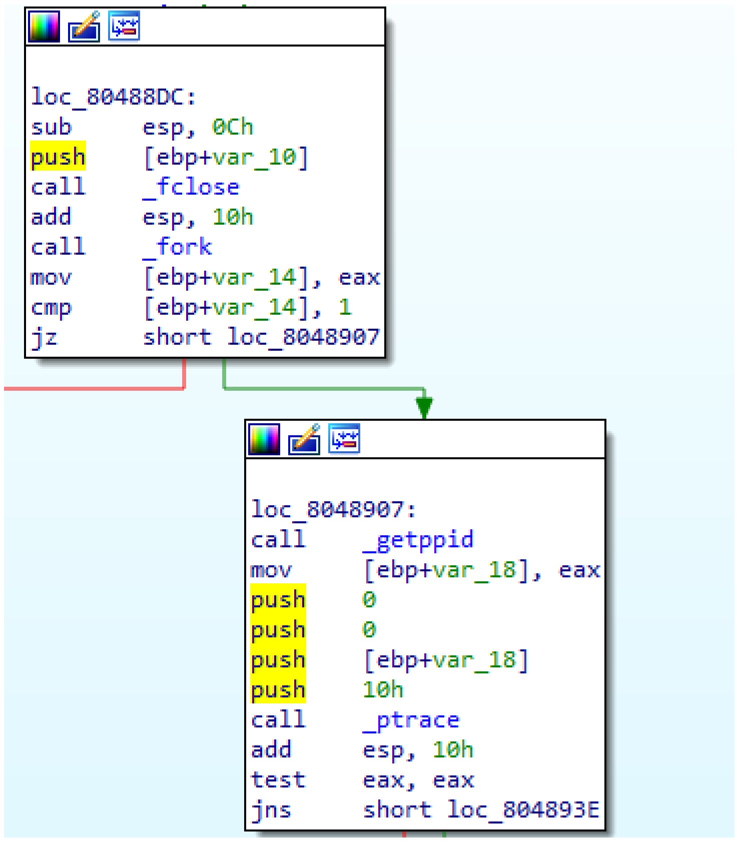
```
loc_8048907:
call    _getppid
mov     [ebp+var_18], eax
push    0
push    0
push    [ebp+var_18]
push    10h
call    _ptrace
add     esp, 10h
test    eax, eax
jns     short loc_804893E
```

```
sub     esp, 0Ch
push    offset aYouFoolNobodyD ; "[-] You fool, nobody debug me!!!"
call    _puts
add     esp, 10h
sub     esp, 0Ch
push    0FFFFFFFFh
call    _exit
```

```
sub     esp, 0Ch
push    0
call    _wait
add     esp, 10h
jmp     short locret_8048966
```

```
loc_804893E:
sub     esp, 0Ch
push    1
call    _sleep
add     esp, 10h
```

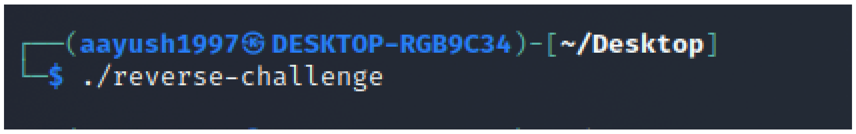
If we look from the top of the image we can see that the program calls **fork** and the value which gets returned is saved into the EAX register. Then we can see in the next step that the value in EAX is then copied into an empty variable. This value is then compared with zero if it gets the value as FALSE then it will wait for the child process to end then it would jump to short loc_804893E. Else if it's true it will go and call **getppid**. Once the program gets the parent ID, it then calls the function **ptrace** while containing the parent ID and ptrace_attach request. It will have two outputs. If it fails the program will display the anti-debug message (aYouFoolNobodyD) and go straight to that function. The second is if it succeeds then it skips the anti-debug message and goes to loc_804893E. At this location, the program would sleep for a moment and then detach using the **ptrace** function with the ptrace_detach request. As you can see from the snapshot above the program calls the **ptrace** function using two values 10h and 11h. Now to get past this, it was easy as in loc_80488DC is the function from which the function calls **fork** which returns zero which is used to determine the jump to loc_8048907 so we just changed the value of 0 to 1 and was able to bypass the debugger (shown below).



```
loc_80488DC:
sub     esp, 0Ch
push    [ebp+var_10]
call    _fclose
add     esp, 10h
call    _fork
mov     [ebp+var_14], eax
cmp     [ebp+var_14], 1
jz      short loc_8048907
```

```
loc_8048907:
call    _getppid
mov     [ebp+var_18], eax
push    0
push    0
push    [ebp+var_18]
push    10h
call    _ptrace
add     esp, 10h
test    eax, eax
jns     short loc_804893E
```

Now we can see above the value cmp [ebp+var_14], 1 the value here is 1 instead of zero. When we run this we get the same output as before but without any delay, so that means that we were successful in bypassing the debugger message.

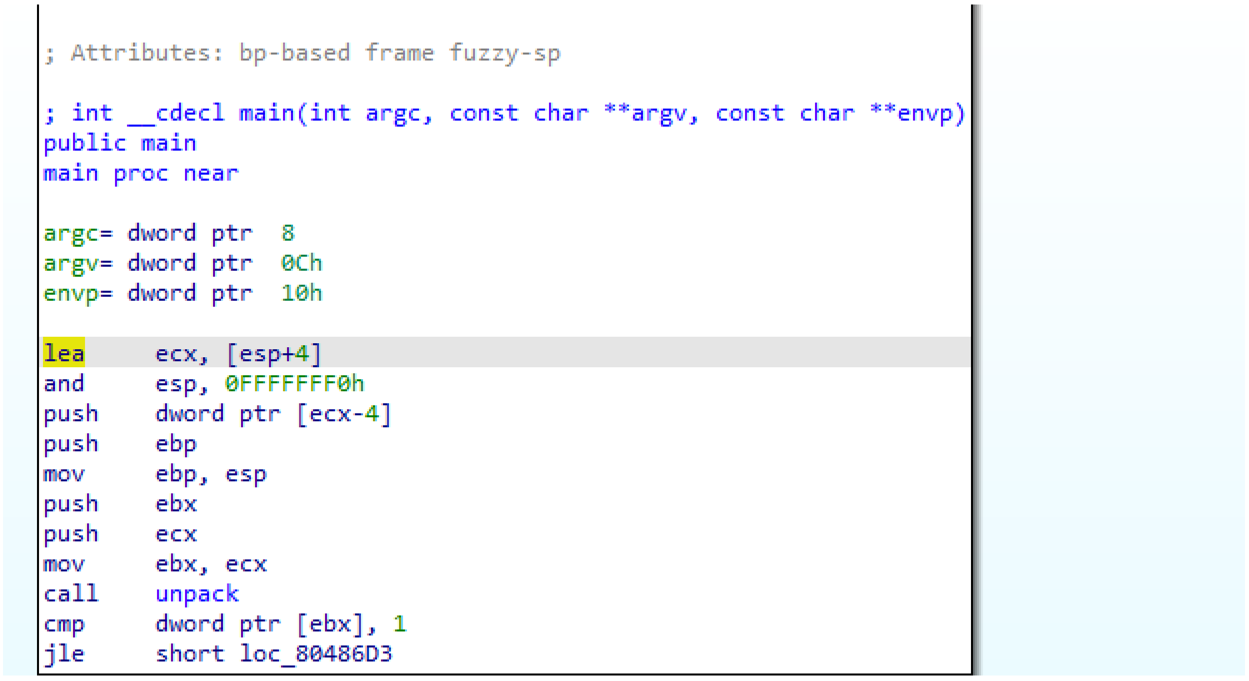


```
┌──(aayush1997㉿DESKTOP-RGB9C34)-[~/Desktop]
└─$ ./reverse-challenge
```

So now that we got rid of the two main hurdles which were not letting us search for the password namely the virtual machine message and the debug message. We can now move ahead to the Main Functions.

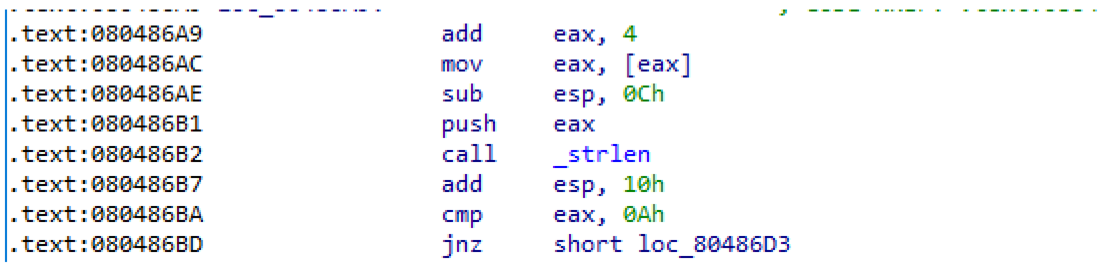
Main Clues

At the beginning of the program in the **main** function, the program aligns the stack pointer at an offset multiple of 10h which is 16 in decimals. And pushes the return address which sets up the stack frame as we can see below:



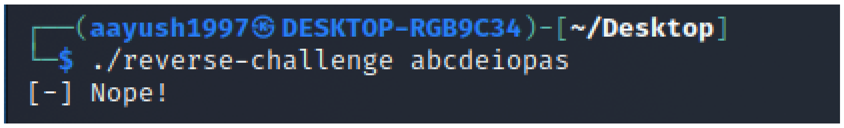
```
; Attributes: bp-based frame fuzzy-sp

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

argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

lea     ecx, [esp+4]
and     esp, 0FFFFFFF0h
push    dword ptr [ecx-4]
push    ebp
mov     ebp, esp
push    ebx
push    ecx
mov     ebx, ecx
call    unpack
cmp     dword ptr [ebx], 1
jle     short loc_80486D3
```

After that the program calls a function called **unpack** and afterward it checks if there is more than one argument and after which it checks if the second argument length equals 10 as you can see below in the screenshot attached:



```
.text:080486A9                 add     eax, 4
.text:080486AC                 mov     eax, [eax]
.text:080486AE                 sub     esp, 0Ch
.text:080486B1                 push    eax
.text:080486B2                 call    _strlen
.text:080486B7                 add     esp, 10h
.text:080486BA                 cmp     eax, 0Ah
.text:080486BD                 jnz     short loc_80486D3
```

So just to check the authenticity of what we found above we decided to send random 10 digits as an input to the program which gave us the following output:



```
┌──(aayush1997㉿DESKTOP-RGB9C34)-[~/Desktop]
└─$ ./reverse-challenge abcdeiopas
[-] Nope!
```

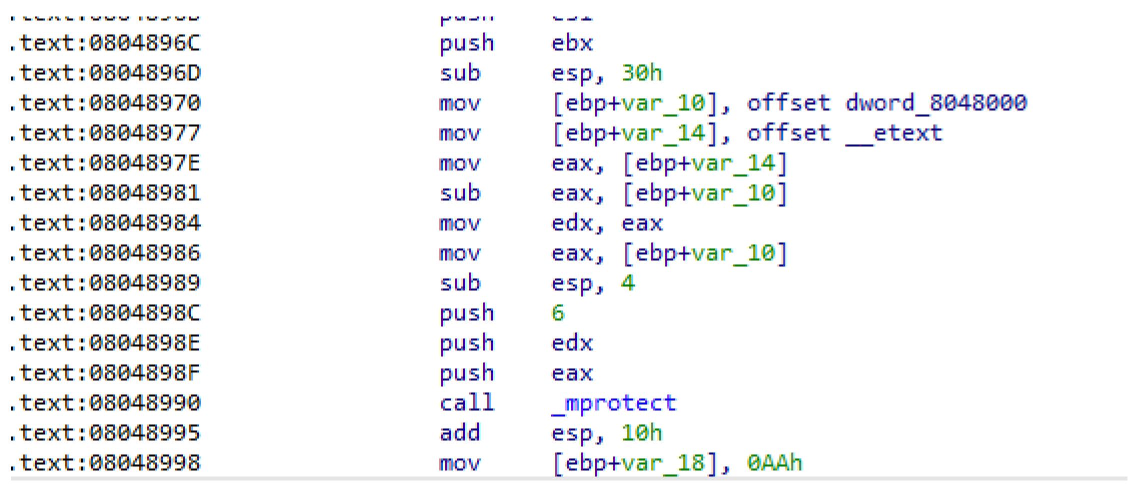
So now we are confirmed that the program would take the second argument which needs to have **10 characters as length** and try to process it. If it is correct, the “[+] Good Job! ;)” otherwise the program would display “[-] Nope !”. But after this, the main question was what was the correct argument? And the only clue we know is that the unpack function mentioned above plays an important role. So we had to do some more analysis.

Unpack function ANALYSIS

The function we encountered above the **unpack function** seems to be the function that could help us solve the password. The function calls three library functions which are as follows:

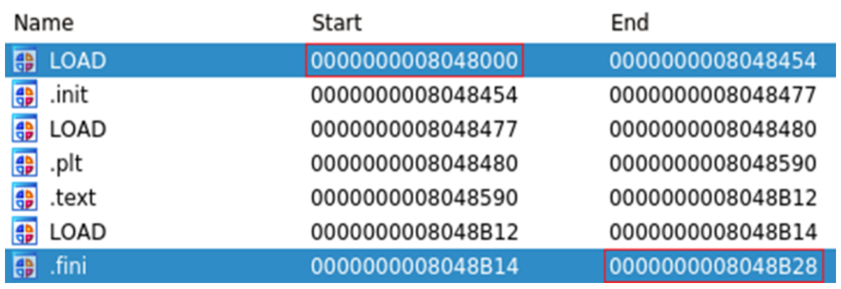
1. **mprotect** which helps in changing the access protection for the caller’s memory page within a certain address range.

   Basically, the function mprotect changes the code to protect while adding the write permission of specific parts of its text section.
2. **calloc** which helps in allocating memory space for an array and then returns a pointer to that allocated space
3. **memcpy** which copies values of n number of bytes from a specific location pointed to by source directly to the memory block pointed by destination.



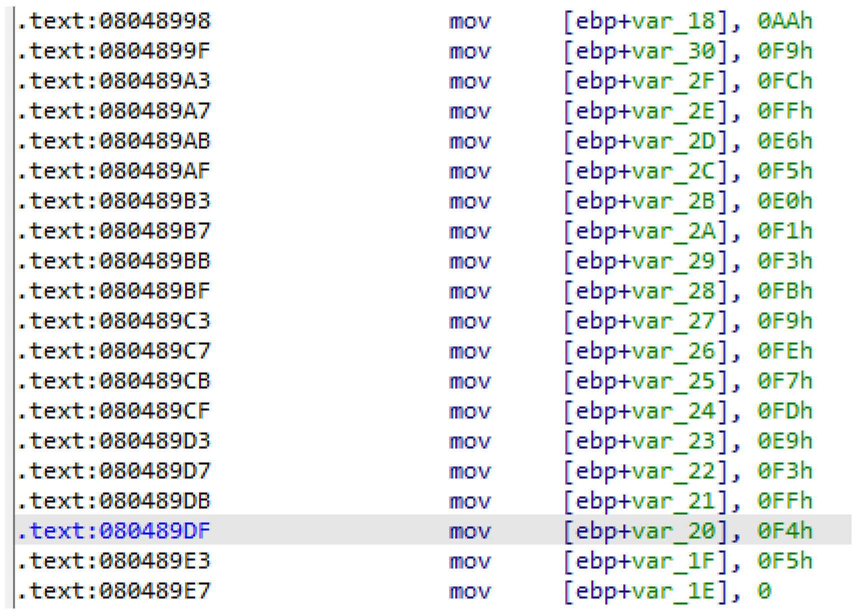
```
.text:0804896C                 push    ebx
.text:0804896D                 sub     esp, 30h
.text:08048970                 mov     [ebp+var_10], offset dword_8048000
.text:08048977                 mov     [ebp+var_14], offset __etext
.text:0804897E                 mov     eax, [ebp+var_14]
.text:08048981                 sub     eax, [ebp+var_10]
.text:08048984                 mov     edx, eax
.text:08048986                 mov     eax, [ebp+var_10]
.text:08048989                 sub     esp, 4
.text:0804898C                 push    6
.text:0804898E                 push    edx
.text:0804898F                 push    eax
.text:08048990                 call    _mprotect
.text:08048995                 add     esp, 10h
.text:08048998                 mov     [ebp+var_18], 0AAh
```

As can be seen, the function starts by using **mprotect** with a length of 2856 bytes, which is 0xB28, starting at dword 8048000. As can be seen in the detailed image below:



Name	Start	End
LOAD	0000000008048000	0000000008048454
.init	0000000008048454	0000000008048477
LOAD	0000000008048477	0000000008048480
.plt	0000000008048480	0000000008048590
.text	0000000008048590	0000000008048B12
LOAD	0000000008048B12	0000000008048B14
.fini	0000000008048B14	0000000008048B28

Then the function sets up a loop by allocating a zero-initialized array via **calloc** call and the loop is set with a number of iterations equal to the allocated array. Inside the loop, there is an XOR operation that is performed on each byte at address **80486E2** which is where the **func _to _pack** is located and uses 90h as a decryption key. But in IDA the key shows as 0xFFFFFF90h, according to us this might be because the integer is unsigned. So once the XOR operation completes once, it is performed again using the same key but this time with 20 values as shown below:

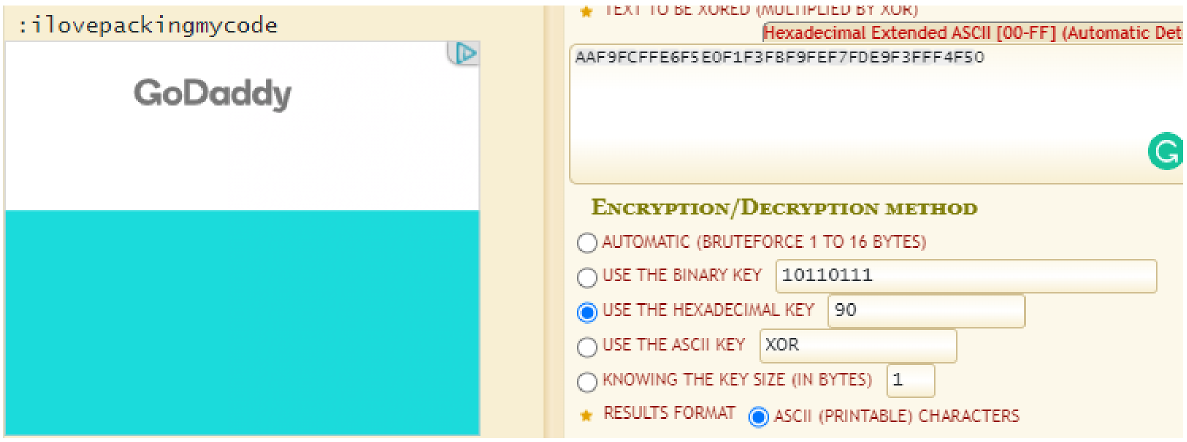


```
.text:08048998                 mov     [ebp+var_18], 0AAh
.text:0804899F                 mov     [ebp+var_30], 0F9h
.text:080489A3                 mov     [ebp+var_2F], 0FCh
.text:080489A7                 mov     [ebp+var_2E], 0FFh
.text:080489AB                 mov     [ebp+var_2D], 0E6h
.text:080489AF                 mov     [ebp+var_2C], 0F5h
.text:080489B3                 mov     [ebp+var_2B], 0E0h
.text:080489B7                 mov     [ebp+var_2A], 0F1h
.text:080489BB                 mov     [ebp+var_29], 0F3h
.text:080489BF                 mov     [ebp+var_28], 0FBh
.text:080489C3                 mov     [ebp+var_27], 0F9h
.text:080489C7                 mov     [ebp+var_26], 0FEh
.text:080489CB                 mov     [ebp+var_25], 0F7h
.text:080489CF                 mov     [ebp+var_24], 0FDh
.text:080489D3                 mov     [ebp+var_23], 0E9h
.text:080489D7                 mov     [ebp+var_22], 0F3h
.text:080489DB                 mov     [ebp+var_21], 0FFh
.text:080489DF                 mov     [ebp+var_20], 0F4h
.text:080489E3                 mov     [ebp+var_1F], 0F5h
.text:080489E7                 mov     [ebp+var_1E], 0
```

And if we try to xor the value with 90h individually and then converting it to Ascii then we will get a value named:

“`:ilovepackingmycode` “

Which was found using the following website:



:ilovepackingmycode

★ TEXT TO BE XORED (MULTIPLIED BY XOR)

Hexadecimal Extended ASCII [00-FF] (Automatic Det

AAF9FCFFE6F5E0F1F3FBF9FEF7FDE9F3FFF4F50

ENCRYPTION/DECRYPTION METHOD

AUTOMATIC (BRUTEFORCE 1 TO 16 BYTES)

USE THE BINARY KEY 10110111

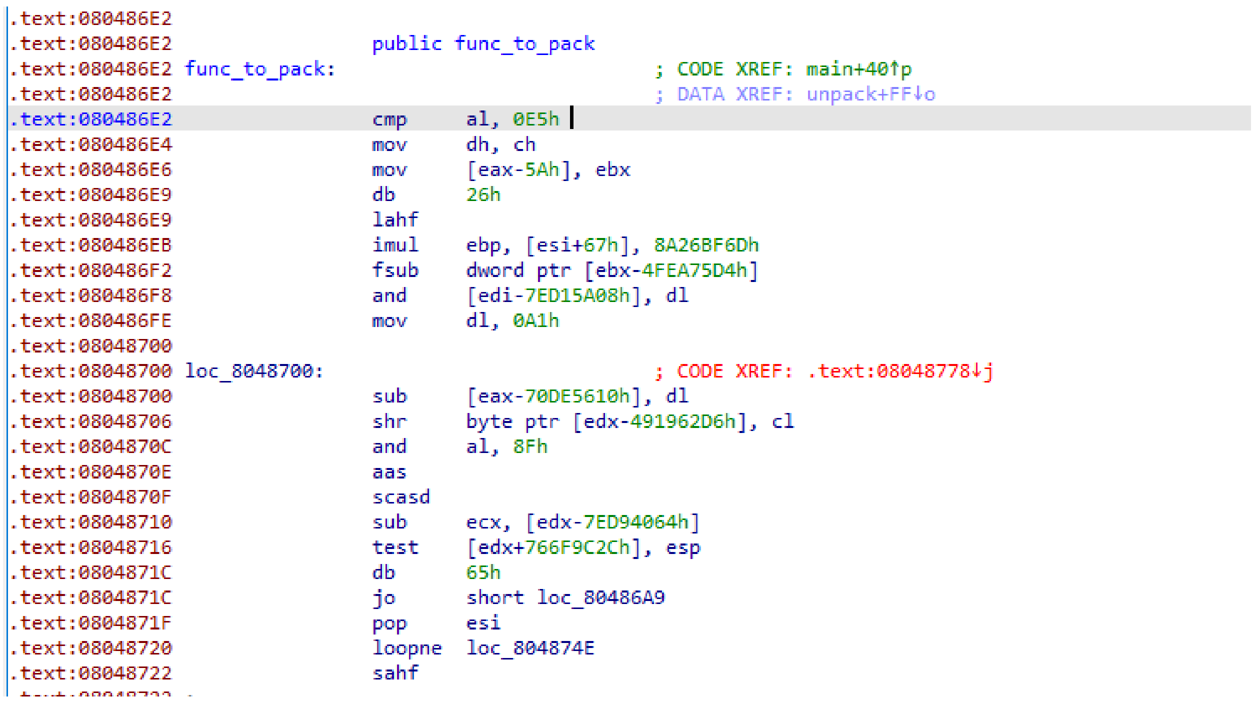
[✓ USE THE HEXADECIMAL KEY] 90

USE THE ASCII KEY XOR

KNOWING THE KEY SIZE (IN BYTES) 1

★ RESULTS FORMAT [✓ ASCII (PRINTABLE) CHARACTERS]

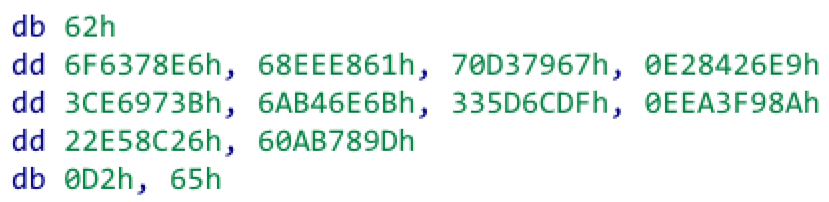
If we go further, The function **func_to_pack** gets overwritten with the unpacked code by the function named **memcpy** whenever the function gets called and once that is completed, **mptotect** gets called in again. So afterward when the packing ends we have a code inside the allocated array.



```
.text:080486E2
.text:080486E2                 public func_to_pack
.text:080486E2 func_to_pack:                           ; CODE XREF: main+40↑p
.text:080486E2                                         ; DATA XREF: unpack+FF↓o
.text:080486E2                 cmp     al, 0E5h
.text:080486E4                 mov     dh, ch
.text:080486E6                 mov     [eax-5Ah], ebx
.text:080486E9                 db      26h
.text:080486E9                 lahf
.text:080486EB                 imul    ebp, [esi+67h], 8A26BF6Dh
.text:080486F2                 fsub    dword ptr [ebx-4FEA75D4h]
.text:080486F8                 and     [edi-7ED15A08h], dl
.text:080486FE                 mov     dl, 0A1h
.text:08048700
.text:08048700 loc_8048700:                            ; CODE XREF: .text:08048778↓j
.text:08048700                 sub     [eax-70DE5610h], dl
.text:08048706                 shr     byte ptr [edx-491962D6h], cl
.text:0804870C                 and     al, 8Fh
.text:0804870E                 aas
.text:0804870F                 scasd
.text:08048710                 sub     ecx, [edx-7ED94064h]
.text:08048716                 test    [edx+766F9C2Ch], esp
.text:0804871C                 db      65h
.text:0804871C                 jo      short loc_80486A9
.text:0804871F                 pop     esi
.text:08048720                 loopne  loc_804874E
.text:08048722                 sahf
```

So what function check does is that it scans through the password entered (User input), each character individually, and then performs xor decryption with the encryption method string stored somewhere in the file. If the password is correct it goes in a loop and then prints the final output string “[+] Good job! ;)” or else it will give an output “[-] Nope!”. So to get the correct password it is necessary to find the flag which contains the required 10 bytes for decryption with which we will perform xor decryption using the string stored in the code.

We believe that the Snapshot shown below the function **func_to_pack (loc_8048700)** contains the string we are supposed to use in xor decryption with the other string which is in the file.



```
db 62h
dd 6F6378E6h, 68EEE861h, 70D37967h, 0E28426E9h
dd 3CE6973Bh, 6AB46E6Bh, 335D6CDFh, 0EEA3F98Ah
dd 22E58C26h, 60AB789Dh
db 0D2h, 65h
```

And the other string is supposed to be hardcoded in the function **check**, as the function is supposed to check the password length first and after verifying the length and then pass it to **loc_8048907** where it tests eax register and then if the password is correct then it jumps to **loc_804893E** giving us the desired output.

Challenges Faced:

1. The Main reason was Managing time as a majority of team members were in different countries with different time zones as well as were busy with other heavy load courses,thus spending less time to discuss and make strategy.
2. Majority of the team members lacked knowledge related to Assembly language, so the project time length became longer using the learn and do scheme.
3. As with less experience with IDA and assembly language it was difficult to follow the code to find where what resides, as at first we were expecting it to be an easy one with a single xor encryption,but due to this it took longer to understand.

Resources Used:

1. The main software used for the entire process was IDA Pro which is used for Binary code analysis.
2. Linux Machine: Kali Linux was used in an actual machine to make the program run without the no vm error message.

3. Virtual Machine: At the start of the Project, Ubuntu in Virtual Environment was used but eventually we had to drop that as we were getting the no vm error.

What have we learned?

- ☐ Reverse-Engineering – We learned how to reverse-engineer a piece of code. This was very enlightening since it brings a different approach to understanding programming.
- ☐ Disassembly - We learned how we can utilize a disassembler to take a deep dive into a program to learn about its functionality and extract information from the same. We used IDA in our case to disassemble code and figure out its functionality.
- ☐ Binary hex code analysis – We learned a unique approach to finding the password through direct code analysis. We found quite a few clues and important information but unfortunately, we couldn’t land on the password itself.
- ☐ Bypassing security measures – The analysis of the code showed us that there was an anti-debugger as well as a module to detect a virtual machine and stop program execution. We had employed countermeasures to bypass these to get closer to finding the password.
- ☐ Clever coding – This exercise has been a great insight into clever programming. Every time we thought we nearly got it, new obstacles showed up. We have learned a few techniques that we can use in our own programming to make our programs better and more secure.

- ☐ We learned various terms that were used in the assembly language and in the program file which we have referred below in references.

Why were you not successful?

- ☐ The main problem for us was time. At every stage of cracking the password, a new set of challenges showed up. In our case, we were able to find the anti-debugger in the code along with code to stop execution in case it was used on a virtual machine.
- ☐ We figured out a way to bypass these checks and still were unable to crack the code. Even though we got certain breakthroughs that led us closer to cracking the password, we could not get over the final hurdle to get it done. We couldn’t figure out the last few pieces of the puzzle which would have led us to the password.
- ☐ Our approach was to simply do what we know. If we knew better, maybe we could have found the password. We had to rely on our knowledge and tools and resources on the internet which we were not familiar with. Our approach got us very close, and with time, we felt we would get over the line.

What could have been done differently?

- ☐ Better planning – We could have started off with a more articulate plan. We decided to do whatever we could with the knowledge we had. Maybe delegating tasks to different members and having a set plan and breakdown of what each teammate will do could have helped us. But we were running short on time.

- ☐ Better communication – We could have communicated better with each other. Everyone set off on their own ideas at their own pace. Having teammates in different time zones (Abbotsford, Montreal, and India) did not help since there was a time delay causing delays in communications.
- ☐ More teamwork – An extension of the first point, we could have worked more efficiently as a team. Having team members in different time zones did not help since there was a delay in getting tasks done.
- ☐ Focussing on each other's strengths – Our approach was to tackle the problem head-on with no solid real plan behind it, the plan that we had was not implemented the way we wanted. We could have utilized each member's strengths in a better and more efficient manner.

A plan we believe would be successful if we had more time

- ☐ We feel that during this exercise, we have learned a great deal about how to approach the problem itself.
- ☐ In the beginning, we didn't have a concrete plan on how to approach this problem.
- ☐ Now that we have given it a shot, we have learned a great deal about how we could find the password.
- ☐ Given more time, we could employ strategies and the knowledge that we have gained to crack the code.
- ☐ We are very confident that we are very close to finding the password already.
- ☐ If we act upon these findings, we feel that we would be able to crack the password.

Conclusions

To Conclude, even though we were not able to find the Password we were supposed to find, the knowledge gained in the process of finding the password was immense. throughout the process, we were able to find a lot of clues and processes such as the length of the password was found to be 10 in length as well as we were able to fully understand and tear apart the code for no vm message and you fool debug message while decoding their xor codes and we can assume that we were only one step before finding the correct password, so the experience gained in this project can be considered very beneficial for our future endeavors.

Youtube Video Link:

https://youtu.be/kYnDk4g48Bo

References

“calloc(3): allocate/free dynamic memory - Linux man page.” *Linux Documentation*, https://linux.die.net/man/3/calloc. Accessed 10 December 2021.

“fgetc, getc.” *cppreference.com*, 20 May 2021, https://en.cppreference.com/w/c/io/fgetc. Accessed 9 December 2021.

Kerrisk, Michael. “fork(2) - Linux manual page.” *man7.org*, 27 August 2021, https://man7.org/linux/man-pages/man2/fork.2.html. Accessed 9 December 2021.

Kerrisk, Michael. “getpid(2) - Linux manual page.” *man7.org*, 27 August 2021, https://man7.org/linux/man-pages/man2/getpid.2.html. Accessed 9 December 2021.

Kerrisk, Michael. “popen(3) - Linux manual page.” *man7.org*, 27 August 2021, https://man7.org/linux/man-pages/man3/popen.3.html. Accessed 9 December 2021.

Kerrisk, Michael. “ptrace(2) - Linux manual page.” *man7.org*, 27 August 2021, https://man7.org/linux/man-pages/man2/ptrace.2.html. Accessed 9 December 2021.

Kerrisk, Michael. “wait(2) - Linux manual page.” *man7.org*, 27 August 2021, https://man7.org/linux/man-pages/man2/wait.2.html. Accessed 9 December 2021.

“XOR Cipher - Exclusive OR Calculator - Online Decoder, Encoder.” *dCode*, https://www.dcode.fr/xor-cipher. Accessed 9 December 2021.