

Final Project
COMP 325
University of the Fraser Valley
Instructor: Talia Q

By: Armaan Chima
Date: December 5th, 2025

What is the purpose of this project?

The following assignment is given to help improve reverse engineering skills, improve general understanding of Assembly language, familiarize students with red pills, more challenging decryption, and the process of constructing a plan and knowledge of extracting a password from the final-project file using effective tools, methods, and strategies.

The assignment will include the plan and methods I used to approach this assignment, the challenges faced, lessons learned, a conclusion, and screenshots of my terminal and IDA (assembly breakdown).

The Steps I Took to Discover the Password

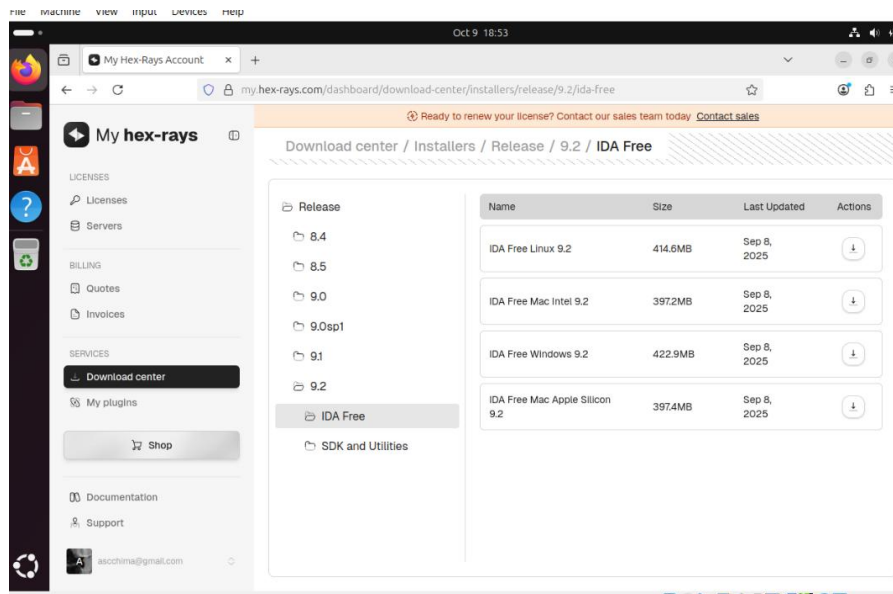
In order to discover what the password is from the final-project file, I used tools such as IDA, GDB (debugger), and Ubuntu (VirtualBox). To approach this assignment, one must conduct this assignment in a controlled environment, where the following tools (IDA and GDB) will help unveil the password. This will be done through a mix of static analysis, dynamic analysis, and references (internal and external resources).

Methods Used

The first thing I did was install a Ubuntu server on VirtualBox by downloading an .iso image from the Ubuntu website. After that, I configured and installed GDB and IDA onto the Ubuntu server, so that the examination of the reverse-ex file could be conducted on a virtual machine instead of my own personal laptop environment. On the other hand, IDA will help convert the file to Assembly language. As for GDB, this will aid in debugging the final-project file.

The following photos show the installation process of the tools that I will be using.

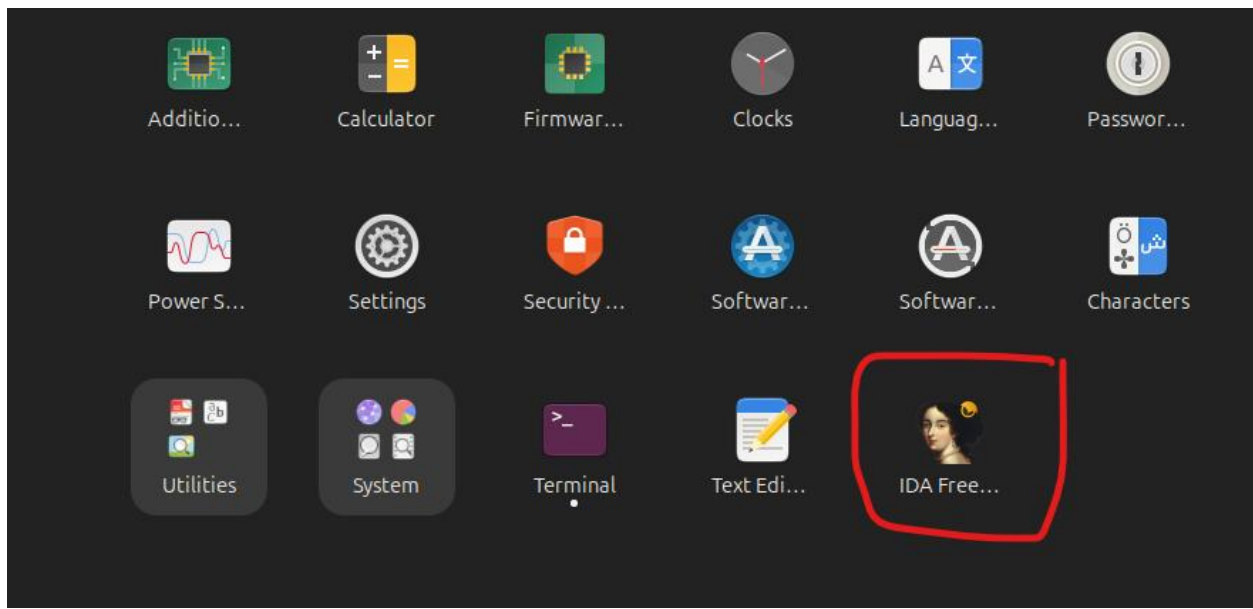
Installation of IDA on VM



Installation of GDB

```
Get:12 http://ca.archive.ubuntu.com/ubuntu questing-backports/multiverse amd64 C
omponents [216 B]
Get:13 http://security.ubuntu.com/ubuntu questing-security/main amd64 Components
[208 B]
Get:14 http://security.ubuntu.com/ubuntu questing-security/restricted amd64 Comp
onents [212 B]
Get:15 http://security.ubuntu.com/ubuntu questing-security/universe amd64 Compon
ents [208 B]
Get:16 http://security.ubuntu.com/ubuntu questing-security/multiverse amd64 Comp
onents [212 B]
Fetched 107 kb in 1s (360 kB/s)
All packages are up to date.
~/Downloads$ sudo apt install -y gdb
gdb is already the newest version (16.3-1ubuntu2).
gdb set to manually installed.
Summary:
  Upgrading: 0, Installing: 0, Removing: 0, Not Upgrading: 0
~/Downloads$ gdb --version
GNU gdb (Ubuntu 16.3-1ubuntu2) 16.3
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
~/Downloads$
```

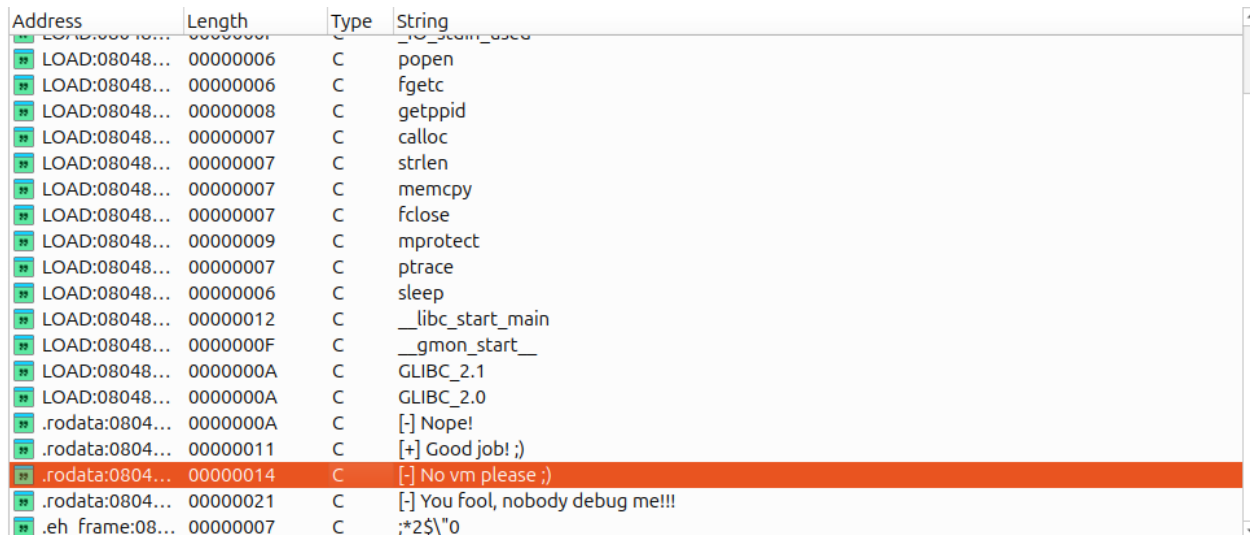
IDA is officially an application on the VM



When executing the final-project file in my terminal, I noticed that the output displayed said “[-] No vm please ;)” and no prompt was given to the user. This assignment is definitely a step up from the reverse-ex decryption assignment.

```
~$ ./final-project
bash: ./final-project: Permission denied
~$ chmod 777 final-project
~$ ./final-project
[-] No vm please ;)
~$
```

As a result, I turned to IDA to help me analyze and examine the Assembly code, as well as the process, to determine how the password/string is being enforced. Furthermore, what I did was I used the command shift+F12 in IDA, which presented many read-only data (rodata) sections.



Address	Length	Type	String
LOAD:08048...	00000006	C	popen
LOAD:08048...	00000006	C	fgetc
LOAD:08048...	00000008	C	getppid
LOAD:08048...	00000007	C	calloc
LOAD:08048...	00000007	C	strlen
LOAD:08048...	00000007	C	memcpy
LOAD:08048...	00000007	C	fclose
LOAD:08048...	00000009	C	mprotect
LOAD:08048...	00000007	C	ptrace
LOAD:08048...	00000006	C	sleep
LOAD:08048...	00000012	C	__libc_start_main
LOAD:08048...	0000000F	C	__gmon_start__
LOAD:08048...	0000000A	C	GLIBC_2.1
LOAD:08048...	0000000A	C	GLIBC_2.0
.rodata:0804...	0000000A	C	[-] Nope!
.rodata:0804...	00000011	C	[+] Good job! ;)
.rodata:0804...	00000014	C	[-] No vm please ;)
.rodata:0804...	00000021	C	[-] You fool, nobody debugs me!!!
.eh_frame:08...	00000007	C	;*2\$\"0

Looking at the above screenshot, we can see four key strings/clues, those being “[-] No vm please ;)”, “[-] Nope”, “[-] You fool, nobody debugs me!!!”, and “[+] Good job ;)”. When I first tried to execute the file, I got the “[-] No vm please ;)” message. As a result, I will start my search there.

Below we can see an image in hex view showing a break down of where all the 4 messages are placed. I also discovered that the memory addresses of “[-] No vm please ;)” and “[-] You fool, nobody debugs me!!!” were found at the memory addresses 0x0804B4D and 0x08048B64.

```

08048AD0 74 27 8D B6 00 00 00 00 8B 44 24 38 89 2C 24 89 t'.....D$8.,$.
08048AE0 44 24 08 8B 44 24 34 89 44 24 04 FF 94 BB 04 FF D$..D$4.D$.....
08048AF0 FF FF 83 C7 01 39 F7 75 DF 83 C4 1C 5B 5E 5F 5D .....9.....[^_]
08048B00 C3 EB 0D 90 90 90 90 90 90 90 90 90 90 90 90 .....
08048B10 F3 C3 00 00 53 83 EC 08 E8 A3 FA FF FF 81 C3 DB ....S.....
08048B20 12 00 00 83 C4 08 5B C3 03 00 00 00 01 00 02 00 .....[.....
08048B30 5B 2D 5D 20 4E 6F 70 65 21 00 5B 2B 5D 20 47 6F [-].Nope!..[+].Go
08048B40 6F 64 20 6A 6F 62 21 20 3B 29 00 72 00 5B 2D 5D od·job!.;).r.[-]
08048B50 20 4E 6F 20 76 6D 20 70 6C 65 61 73 65 20 3B 29 ·No·vm·please·;)
08048B60 00 00 00 00 5B 2D 5D 20 59 6F 75 20 66 6F 6F 6C ....[-].You·fool
08048B70 2C 20 6E 6F 62 6F 64 79 20 64 65 62 75 67 20 6D ,·nobody·debug·m
08048B80 65 21 21 21 00 00 00 00 01 1B 03 3B 40 00 00 00 e!!!.....;@...
08048B90 07 00 00 00 F8 F8 FF FF 5C 00 00 00 03 FB FF FF .....\.
08048BA0 80 00 00 00 5A FB FF FF B4 00 00 00 05 FC FF FF ....Z.....
08048BB0 D4 00 00 00 E0 FD FF FF F4 00 00 00 18 FF FF FF .....
08048BC0 20 01 00 00 88 FF FF FF 5C 01 00 00 14 00 00 00 .....\.
08048BD0 00 00 00 00 01 7A 52 00 01 7C 08 01 1B 0C 04 04 .....zR..|.
08048BE0 88 01 00 00 20 00 00 00 1C 00 00 00 94 F8 FF FF .....'.
08048BF0 10 01 00 00 00 0E 08 46 0E 0C 4A 0F 0B 74 04 78 .....F..J..t.x
08048C00 00 3F 1A 3B 2A 32 24 22 30 00 00 00 40 00 00 00 .?.;*2$"0...@...
08048C10 7B FA FF FF 57 00 00 00 00 44 0C 01 00 47 10 05 {...W....D...G..
08048C20 02 75 00 44 0F 03 75 78 06 10 03 02 75 7C 02 42 ..u.D..ux....u|.B
08048C30 C1 0C 01 00 41 C3 41 C5 43 0C 04 04 1C 00 00 00 ....A.....
08048C40 74 00 00 00 9E FA FF FF AB 00 00 00 00 41 0E 08 t.....A..
08048C50 85 02 42 0D 05 02 A7 C5 0C 04 04 00 1C 00 00 00 ..B.....
08048C60 94 00 00 00 79 FB FF FF 0B 01 00 00 00 41 0F 08 .....A..
00000B53 08048B53: .rodata:s+6 (Synchronized with IDA View-A)

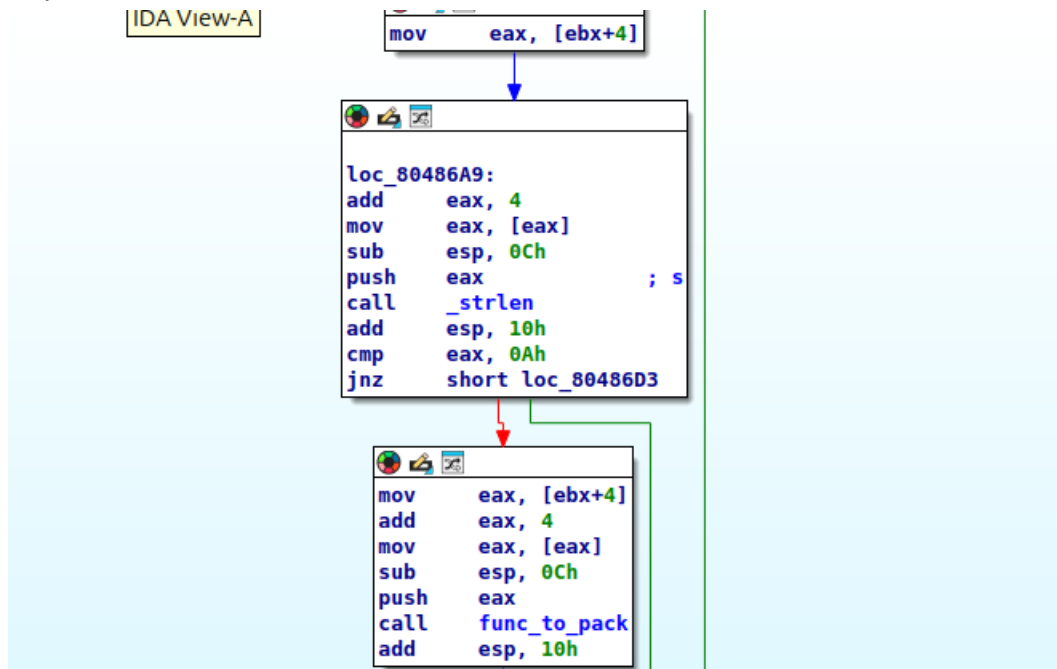
```

Now we must bypass the red pill detection. We need to first start by analyzing the check function in further detail.

After further analysis, I noticed that the main function plays a large part in this assignment as it turns to 3 primary functions, these are...

- func_to_pack
- _strlen

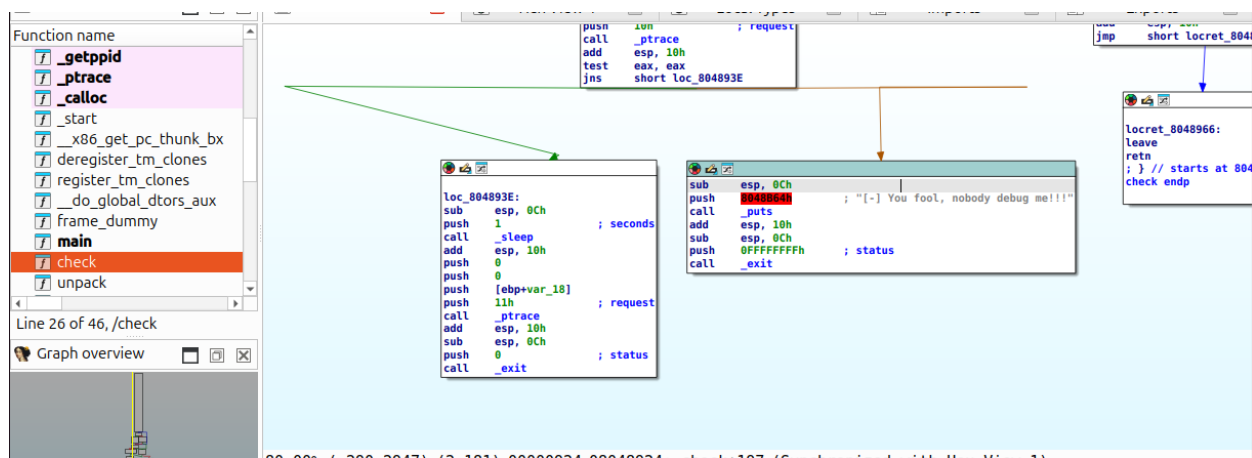
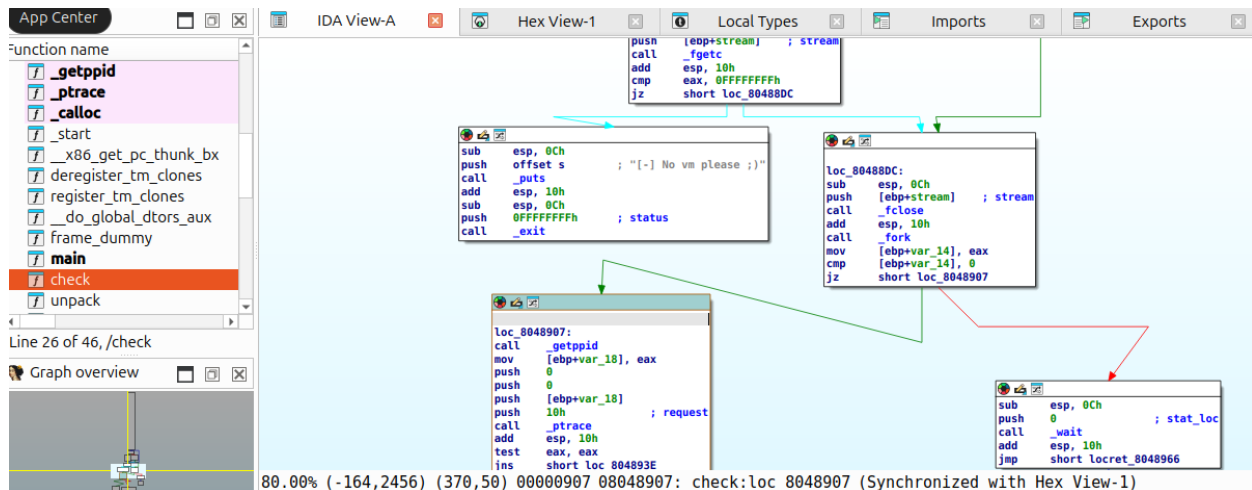
- unpack



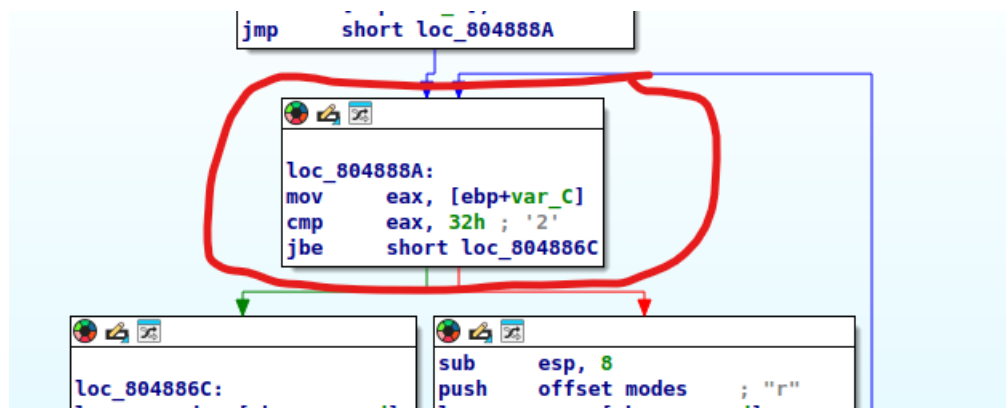
What I notice is that the arrangement in the check function originally brings about the following message “[-] No vm please ;)” and after the fact, the message “[-] You fool, nobody debugs me!!!”. After recognizing this, it has become clear that the first red pill is the virtual machine detection resulting in the message “[-] No vm please ;)”

Next, the XOR function’s duty inside the program should be further inspected, and there should also be an effort made to deduce the encoded string it interprets.

Note: These screenshots below are all in the check function (had to take separate photos as the graphical view was too large).

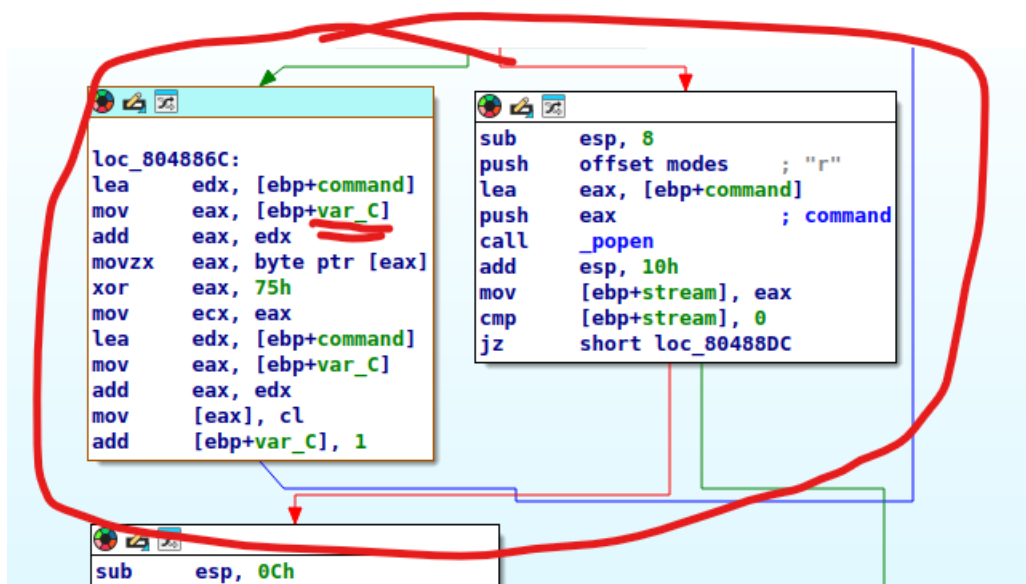


Let's recap! It's clear that the first red pill is in the check function (this has become obvious). Below, we can see a hexadecimal value that jumps to the address 0x0804888A. Furthermore, the image below shows us an instruction `cmp eax, 32h` and the hexadecimal value 32h. Now, if we were to translate this 32h value from ASCII to a decimal value, we would get the decimal value of 50 (hmmm interesting).



But what does this mean? This means that there is a recurring process that is happening in a loop, and there are 50 cycles of it, and var_C (the variable) is keeping track of the iterations.

I would also like to add that the 75h hexadecimal value is linked to the XOR operation (something to keep in mind). The next thing on the list is we need to find a way to reveal the hidden string. This can be done by implementing XOR processes with the 75h hexadecimal value to the 50 hexadecimal values that we just discussed.



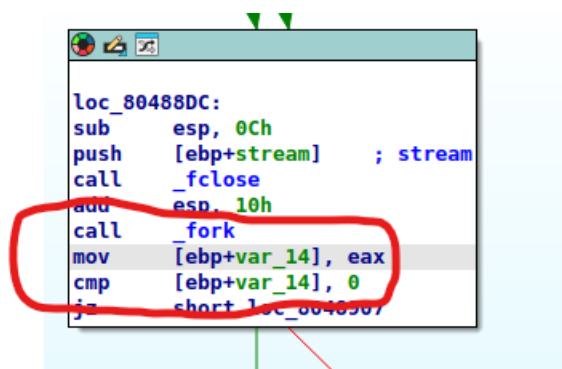
After the XOR operation was managed, I got the command -> lscpu | grep 'Hypervisor' | tr -d ' ' | cut -d: -f2, which, when run, produces the output of KVM as we can see in the screenshot produced below.

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

Now that that is dealt with, we need to deal with the [-] You fool, nobody debugs me!!!” message (this is a red pill).

Something to address is at the memory address 0x080488DC; fork is used, which plays a part as this produces a process (specifically a child one).

Another thing to note is that the mov command predetermines content in eax, when contrasting the zero present in the following line. Once certain circumstances are reached, the memory address 0x08048907 is pursued.

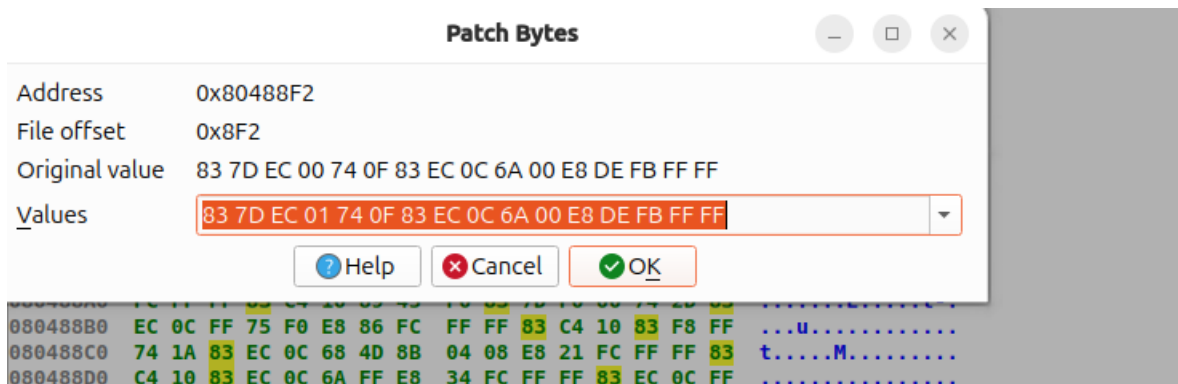


Let's switch our focus to the getppid function. The getppid function is called, which results in a PPID coming back. Furthermore, the ptrace function is being included in this final project, which implies the presence of crucial information to solve this red pill problem.

Note: If we want to avoid the check (debugging check specifically), we need to switch our attention to the getppid function and the fork function.

How do we get passed this hurdle? Well, we need to change 0x080488F2 since this is where the debug checking is coming in from and once this is solved, we will be able to

bypass the “[-] No vm please ;)” message. Below are photo’s of how this is done



There is also something else to mention, which is that after the ptrace function is called on, 10h (the hexadecimal value) is included, and this is evaluated against the register eax. Next, the program heads to the memory address 0x0804893E. This leads to our next segment, which is taking a deeper look at the pack function and the unpack function.

Let’s analyze the calloc, memcpy, strlen, and mprotect functions!

Notice how dword starts at the 0x08048454 memory address and ends at the 0x08048B28 memory address.

LOAD	08048000	08048454	R	.	X	.	L	mempage	0001	public
.init	<u>08048454</u>	08048477	R	.	X	.	L	dword	0004	public
LOAD	08048477	08048480	R	.	X	.	L	mempage	0001	public
.plt	08048480	08048590	R	.	X	.	L	para	0005	public
.text	08048590	08048B12	R	W	X	.	L	para	0006	public
LOAD	08048B12	08048B14	R	.	X	.	L	mempage	0001	public
.fini	08048B14	<u>08048B28</u>	R	.	X	.	L	dword	0007	public
.rodata	08048B28	08048B85	R	.	.	.	L	dword	0008	public
LOAD	08048B85	08048B88	R	.	X	.	L	mempage	0001	public
.eh_frame_hdr	08048B88	08048BCC	R	.	.	.	L	dword	0009	public
.eh_frame	08048BCC	08048CFC	R	.	.	.	L	dword	000A	public
.init_array	08049CFC	08049D04	R	W	.	.	L	dword	000B	public
.fini_array	08049D04	08049D08	R	W	.	.	L	dword	000C	public
.jcr	08049D08	08049D0C	R	W	.	.	L	dword	000D	public
LOAD	08049D0C	08049DF4	R	W	.	.	L	mempage	0002	public
.got	08049DF4	08049DF8	R	W	.	.	L	dword	000E	public
.got.plt	08049DF8	08049E44	R	W	.	.	L	dword	000F	public
.data	08049E44	08049E4C	R	W	.	.	L	dword	0010	public
.bss	08049E4C	08049E50	R	W	.	.	L	byte	0011	public

Once the mprotect function is called, I noticed that many hexadecimal values appeared. Since mprotect has a say in memory control protection, every byte in the arrangement has the potential to go through an XOR process for the hexadecimal 90h. With this knowledge being known, we might be able to reveal a concealed string.

Something else to note is that the calloc function assigns memory intended for object format to 0; there is potential for a link to the func_to_pack offset and a loop as well. We might need to use XOR decryption methods to get this string. The loop is as follows: decoded character equals encoded character + 112.

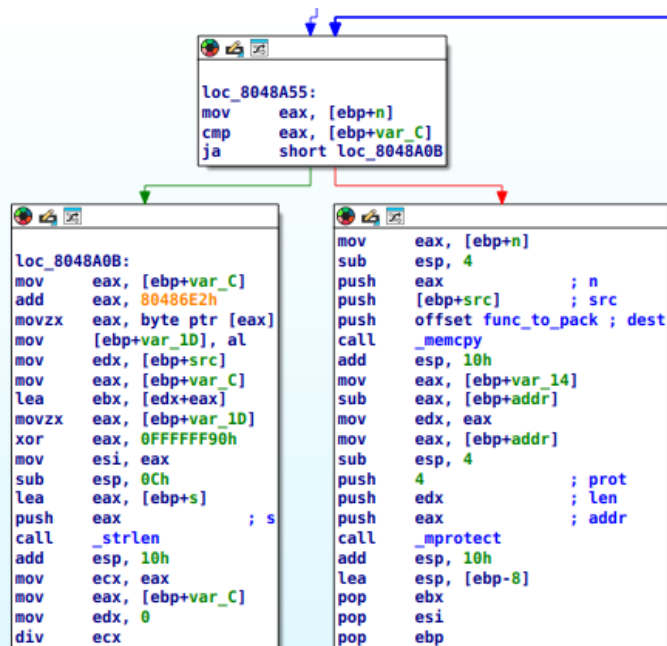
After decryption of the XOR function, we get this -> ilovepackingcode

```

call    _mprotect
add     esp, 10h
mov     [ebp+n], 0AAh
mov     [ebp+s], 0F9h
mov     [ebp+var_2F], 0FCh
mov     [ebp+var_2E], 0FFh
mov     [ebp+var_2D], 0E6h
mov     [ebp+var_2C], 0F5h
mov     [ebp+var_2B], 0E0h
mov     [ebp+var_2A], 0F1h
mov     [ebp+var_29], 0F3h
mov     [ebp+var_28], 0F8h
mov     [ebp+var_27], 0F9h
mov     [ebp+var_26], 0FEh
mov     [ebp+var_25], 0F7h
mov     [ebp+var_24], 0FDh
mov     [ebp+var_23], 0E9h
mov     [ebp+var_22], 0F3h
mov     [ebp+var_21], 0FFh
mov     [ebp+var_20], 0F4h
mov     [ebp+var_1F], 0F5h
mov     [ebp+var_1E], 0
mov     eax, [ebp+n]
add     eax, 1
sub     esp, 8
push    1           ; size
push    eax         ; nmem
call    _calloc
add     esp, 10h
mov     [ebp+src], eax
mov     [ebp+var_C], 0
jmp     short loc_8048A55

```

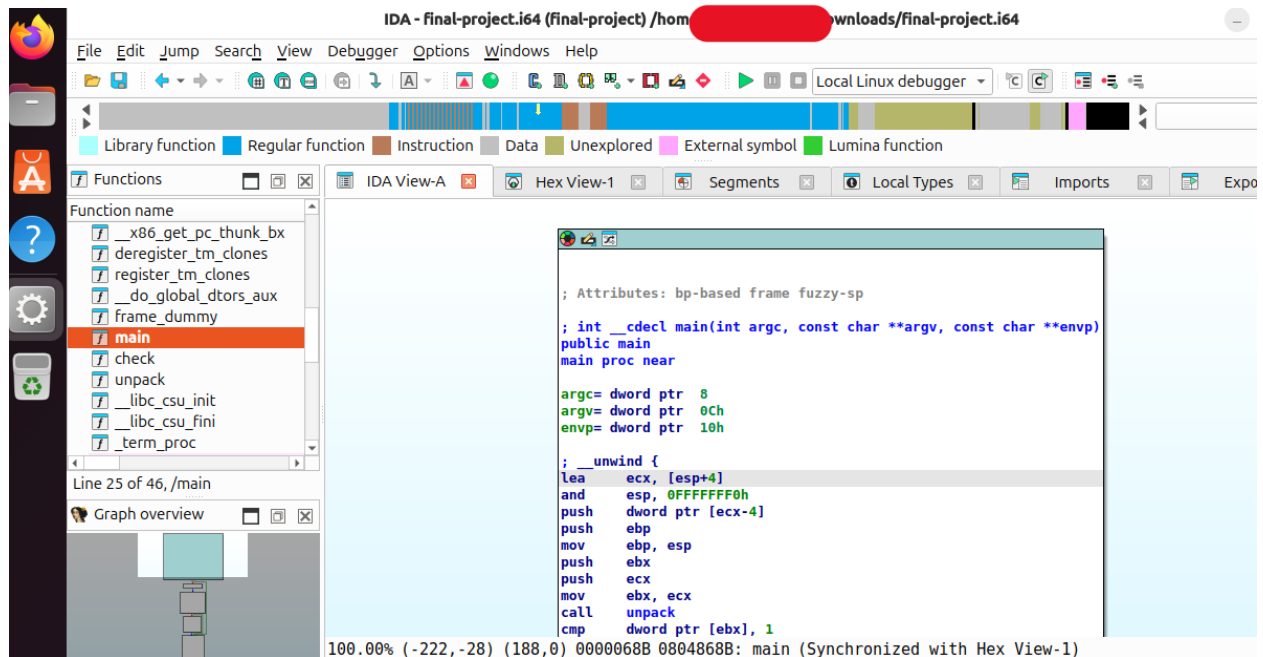
1968 08048968: unpack (Synchronized with Hex



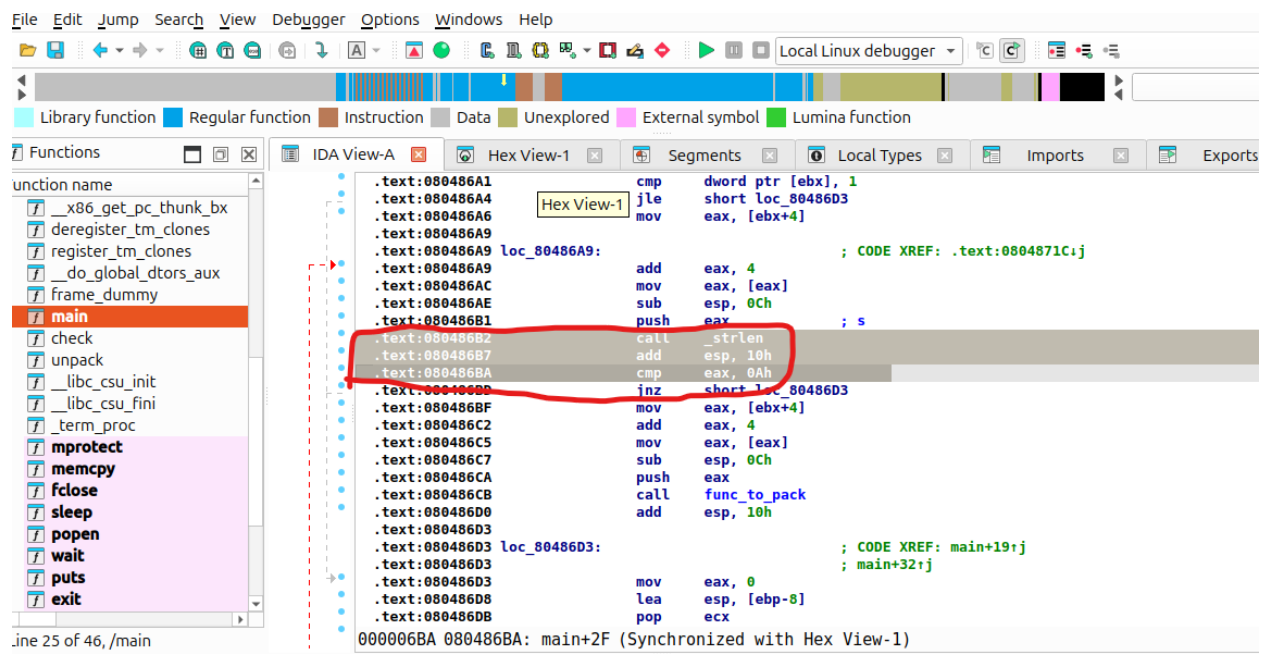
9,1329) (3,21) 00000968 08048968: unpack (Synchronized with Hex View-1)

aded (v9.2.0.250908)

Taking a look at the main function, we can see that envp = 10h, argv = 0Ch, and argc = 8. This is something to keep in mind.

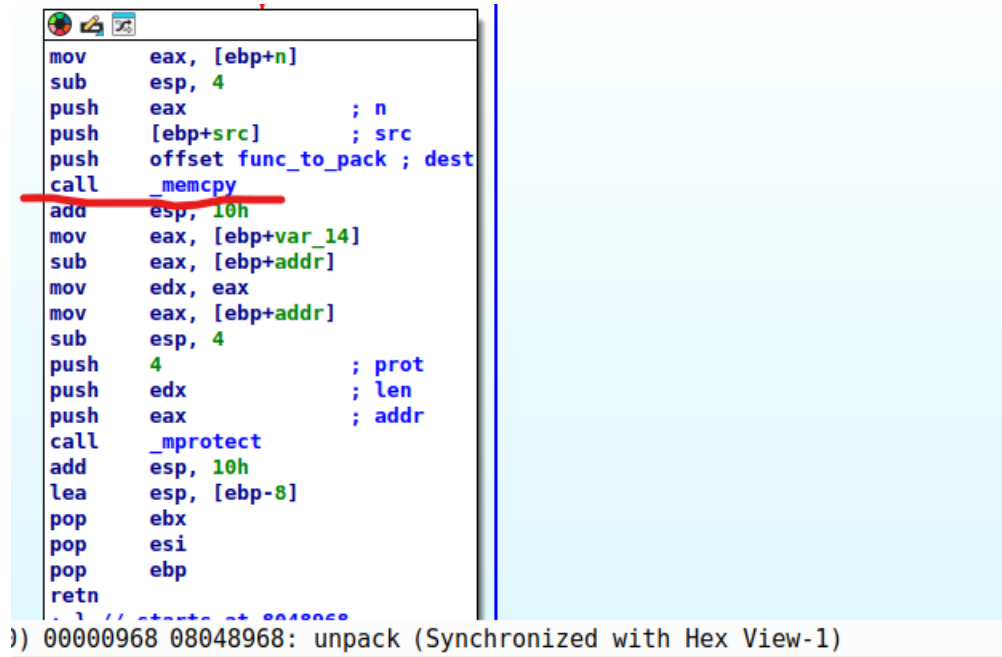


Next let's observe esp 10h, eax 0Ah, and strlen. First things first, the strlen function gauges the string's size, and is compared to 0Ah. However, 0Ah is equivalent to 10 in decimal, so we need to do some experimenting to see if the string's length is 10.



The string/password must be 10 characters in length, as when a string of 10 is inserted in completion, we get a “[-] Nope!” response.

Now let’s dive into the unpack function. Followed by the mprotect function, we can see where the memcpy function is called (Underlined in red in the photo below).

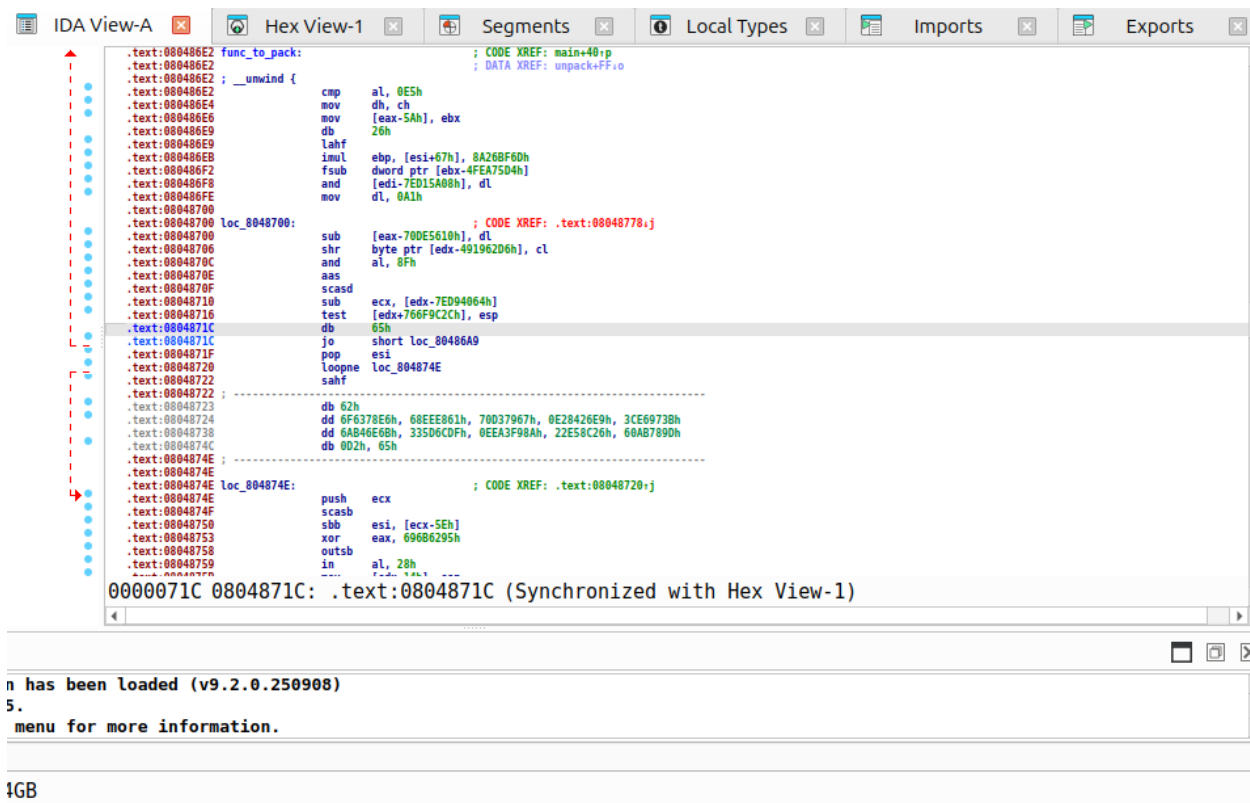


```
mov     eax, [ebp+n]
sub     esp, 4
push    eax                ; n
push    [ebp+src]         ; src
push    offset func_to_pack ; dest
call    _memcpy
add     esp, 10h
mov     eax, [ebp+var_14]
sub     eax, [ebp+addr]
mov     edx, eax
mov     eax, [ebp+addr]
sub     esp, 4
push    4                 ; prot
push    edx               ; len
push    eax               ; addr
call    _mprotect
add     esp, 10h
lea     esp, [ebp-8]
pop     ebx
pop     esi
pop     ebp
retn
```

00000968 08048968: unpack (Synchronized with Hex View-1)

Things are getting interesting now as we need to use GDB (debugger) to disassemble the func_to_pack function. Doing this will allow us to inspect how this function is truly operating, which could expose clues to bring us closer to our goal.

The idea is this: if the password is incorrect, we get the “[-] Nope!” message; else, we get the “[+] Good Job” message if the password is correct. Another thing to consider is that the XOR-encrypted string in the check function could be related to the input string.



Let's analyze the func_to_pack using GDB! In order for this to work, we need to mainly use the disassemble, run, and breakpoints commands. Like the reverse-ex assignment, when the breakpoint is reached, we will be halted. However, this time, when we did it breakpoint func_to_pack, the same steps were needed, but when I inserted "disassemble" into GDB, it gave me the 0x080486ef to 0x08048713. Going back to what I said about the length of the string being 10 characters, these hex codes are possibly related to the ptrace function, which means we are getting close.


```

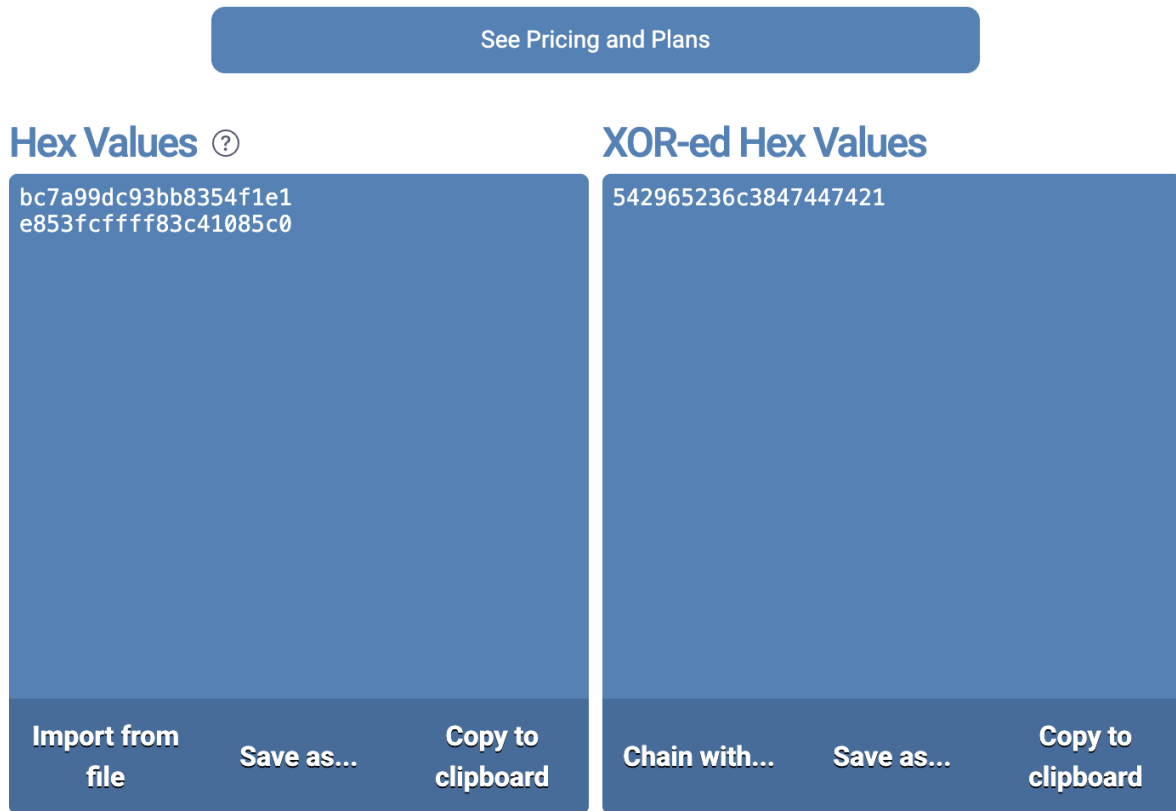
db) disassemble
mp of assembler code for function func_to
0x080486e2 <+0>:      cmp     $0x89,%al
0x080486e4 <+2>:      in      $0x83,%eax
0x080486e6 <+4>:      in      (%dx),%al
0x080486e7 <+5>:      sub     %al,%bh
0x080486e9 <+7>:      inc     %ebp
0x080486ea <+8>:      hlt
0x080486eb <+9>:      add     %al,(%eax)
0x080486ed <+11>:     add     %al,(%eax)
0x080486ef <+13>:     movb    $0xbc,-0x1b(%ebp)
0x080486f3 <+17>:     movb    $0x7a,-0x1a(%ebp)
0x080486f7 <+21>:     movb    $0x99,-0x19(%ebp)
0x080486fb <+25>:     movb    $0xdc,-0x18(%ebp)
0x080486ff <+29>:     movb    $0x93,-0x17(%ebp)
0x08048703 <+33>:     movb    $0xbb,-0x16(%ebp)
0x08048707 <+37>:     movb    $0x83,-0x15(%ebp)
0x0804870b <+41>:     movb    $0x54,-0x14(%ebp)
0x0804870f <+45>:     movb    $0xf1,-0x13(%ebp)
0x08048713 <+49>:     movb    $0xe1,-0x12(%ebp)
0x08048717 <+53>:     movl    $0x0,-0x10(%ebp)
0x0804871e <+60>:     jmp     0x0804875d <func_to+0x5d>
0x08048720 <+62>:     mov     -0x10(%ebp),%eax
0x08048723 <+65>:     add     $0x18b,%eax
0x08048728 <+70>:     add     $0x804878d,%eax
0x0804872d <+75>:     movzbl  (%eax),%eax
0x08048730 <+78>:     mov     %al,-0x11(%ebp)
0x08048733 <+81>:     mov     -0x10(%ebp),%eax

```

The hexadecimal values revealed from the ptrace function, thanks to GDB, are -> **0xE8**, **0x53**, **0xFC**, **0xFF**, **0x83**, **0xC4**, **0x10**, **0x85**, **0xC0**

The hexadecimal values revealed from the func_to_pack function, thanks to GDB, are -> **0xBC, 0x7A, 0x99, 0xDC, 0x93, 0xBB, 0x83, 0x54, 0xF1, 0xE1**

Below is a photo of hexadecimal being converted to a hex string



Below is a screenshot of the hex string being converted to ASCII

From: Hexadecimal To: Text

Open File Sample

Paste hex code numbers or drop file

542965236c3847447421

Character encoding: ASCII

Convert Reset Swap

T)e#l8GDt!

Copy Save

Now, let's test if this password works!

[+] Good job! ;)

Challenges:

Some of the challenges that I faced were...

- Getting the “no vm please ;)” message was pretty annoying as I struggled to get full and proper access when it came to inputting results and getting the correct answer for the password. Every time I tried to patch bytes, I would consistently run into problems, and it was a tedious process.
- Analyzing and breaking down the second red pill was very difficult for me, as it took a lot of analysis and trial and error in terms of looking at hexadecimal values and functions
- Analyzing the functions was a pretty difficult task since I had to go back many times to certain functions and cross-reference, which resulted in a lot of experimentation and testing

Lessons Learned:

Some of the lessons that I have learned are...

- Getting to sharpen my skills with assembly language, as this assignment had a stronger emphasis on variables and loops compared to the previous one.
- Learning how to discover, analyze and use the red pills to my advantage when it came to cracking the passcode.
- I had the opportunity to crack this password, which taught me to stay disciplined, resilient and offered a new perspective on encryption and general security principles.

Conclusion:

Cracking passwords is a challenging process that involves an immense amount of time, effort and a well-thought-out plan in order to succeed. With the help of IDA Free, I had the opportunity to thoroughly analyze the presented Assembly code and find a way to reveal the secret passcode. The `func_to_pack`, `main`, `ptrace`, `check`, and `strlen` functions offer different perspectives and views of the code's formation and movement, which helped in my search for the red pill and passcode. In addition, the hexadecimal values presented in GDB also play a massive part in the end of the assignment. Overall, this was a very interesting and fun project that really put decryption and reverse engineering skills to the test.

References:

ASCII Table - ASCII character codes, HTML, octal, Hex, decimal. (n.d.).

<https://www.asciitable.com/>

Shefali Kumari. (2022, May 2). *Basic Reverse Engineering using GDB* [Video]. YouTube.

<https://www.youtube.com/watch?v=mYY6xHBo4zg>

XOR Hex Numbers. (n.d.). <https://onlinetools.com/hex/xor-hex-numbers>

Hex to String | Hex to ASCII Converter. (n.d.).

<https://www.rapidtables.com/convert/number/hex-to-ascii.html>

Dr Josh Stroschein - The Cyber Yeti. (2023b, December 27). *Patching Binaries with IDA Pro (free)!* [Video]. YouTube. https://www.youtube.com/watch?v=_9swR6zUkWg

Debasish Mandal. (2020, July 31). *Reverse Engineering Tutorial with IDA Pro – An Introduction to IDA Pro.* [Video]. YouTube.

https://www.youtube.com/watch?v=N_3AGB9Vf9E

