Week 6(Reverse Engineering) Report

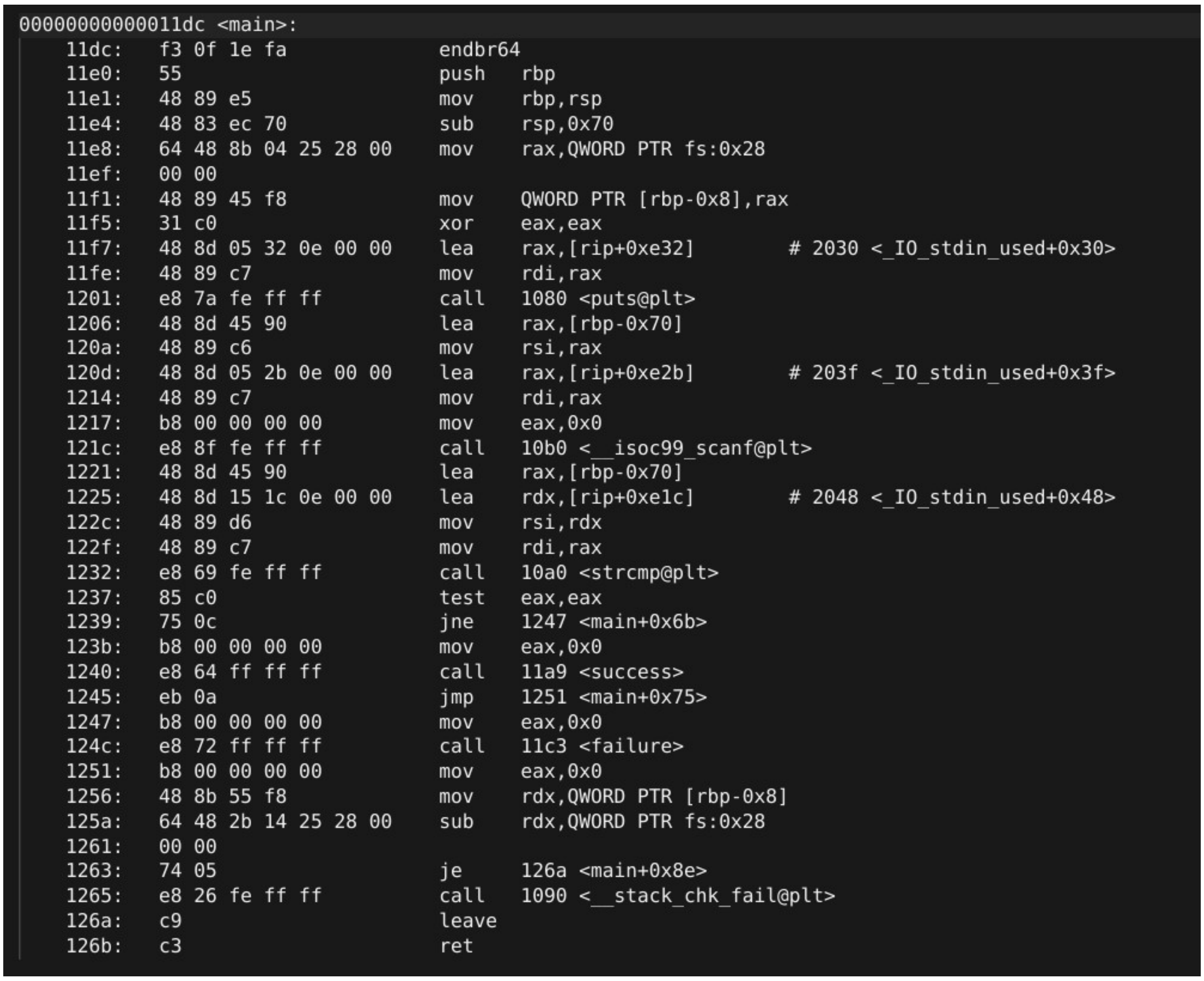
Problem 1:

First of all I convert the binary to Assembly code to understand the program and stored it in a file prob1_asm.txt using the command:

```
objdump -d -M intel Prob1 > prob1_asm.txt
```

here I have used "-M intel" with the disassemble -d flag to get the code in intel syntax instead of AT&T syntax for better readability and understandability.

Here is the code snippet of the main function of the converted Assembly code:



```
00000000000011dc <main>:
    11dc:   f3 0f 1e fa             endbr64
    11e0:   55                      push   rbp
    11e1:   48 89 e5                mov    rbp,rsp
    11e4:   48 83 ec 70             sub    rsp,0x70
    11e8:   64 48 8b 04 25 28 00    mov    rax,QWORD PTR fs:0x28
    11ef:   00 00
    11f1:   48 89 45 f8             mov    QWORD PTR [rbp-0x8],rax
    11f5:   31 c0                   xor    eax,eax
    11f7:   48 8d 05 32 0e 00 00    lea    rax,[rip+0xe32]        # 2030 <_IO_stdin_used+0x30>
    11fe:   48 89 c7                mov    rdi,rax
    1201:   e8 7a fe ff ff          call   1080 <puts@plt>
    1206:   48 8d 45 90             lea    rax,[rbp-0x70]
    120a:   48 89 c6                mov    rsi,rax
    120d:   48 8d 05 2b 0e 00 00    lea    rax,[rip+0xe2b]        # 203f <_IO_stdin_used+0x3f>
    1214:   48 89 c7                mov    rdi,rax
    1217:   b8 00 00 00 00          mov    eax,0x0
    121c:   e8 8f fe ff ff          call   10b0 <__isoc99_scanf@plt>
    1221:   48 8d 45 90             lea    rax,[rbp-0x70]
    1225:   48 8d 15 1c 0e 00 00    lea    rdx,[rip+0xe1c]        # 2048 <_IO_stdin_used+0x48>
    122c:   48 89 d6                mov    rsi,rdx
    122f:   48 89 c7                mov    rdi,rax
    1232:   e8 69 fe ff ff          call   10a0 <strcmp@plt>
    1237:   85 c0                   test   eax,eax
    1239:   75 0c                   jne    1247 <main+0x6b>
    123b:   b8 00 00 00 00          mov    eax,0x0
    1240:   e8 64 ff ff ff          call   11a9 <success>
    1245:   eb 0a                   jmp    1251 <main+0x75>
    1247:   b8 00 00 00 00          mov    eax,0x0
    124c:   e8 72 ff ff ff          call   11c3 <failure>
    1251:   b8 00 00 00 00          mov    eax,0x0
    1256:   48 8b 55 f8             mov    rdx,QWORD PTR [rbp-0x8]
    125a:   64 48 2b 14 25 28 00    sub    rdx,QWORD PTR fs:0x28
    1261:   00 00
    1263:   74 05                   je     126a <main+0x8e>
    1265:   e8 26 fe ff ff          call   1090 <__stack_chk_fail@plt>
    126a:   c9                      leave
    126b:   c3                      ret
```

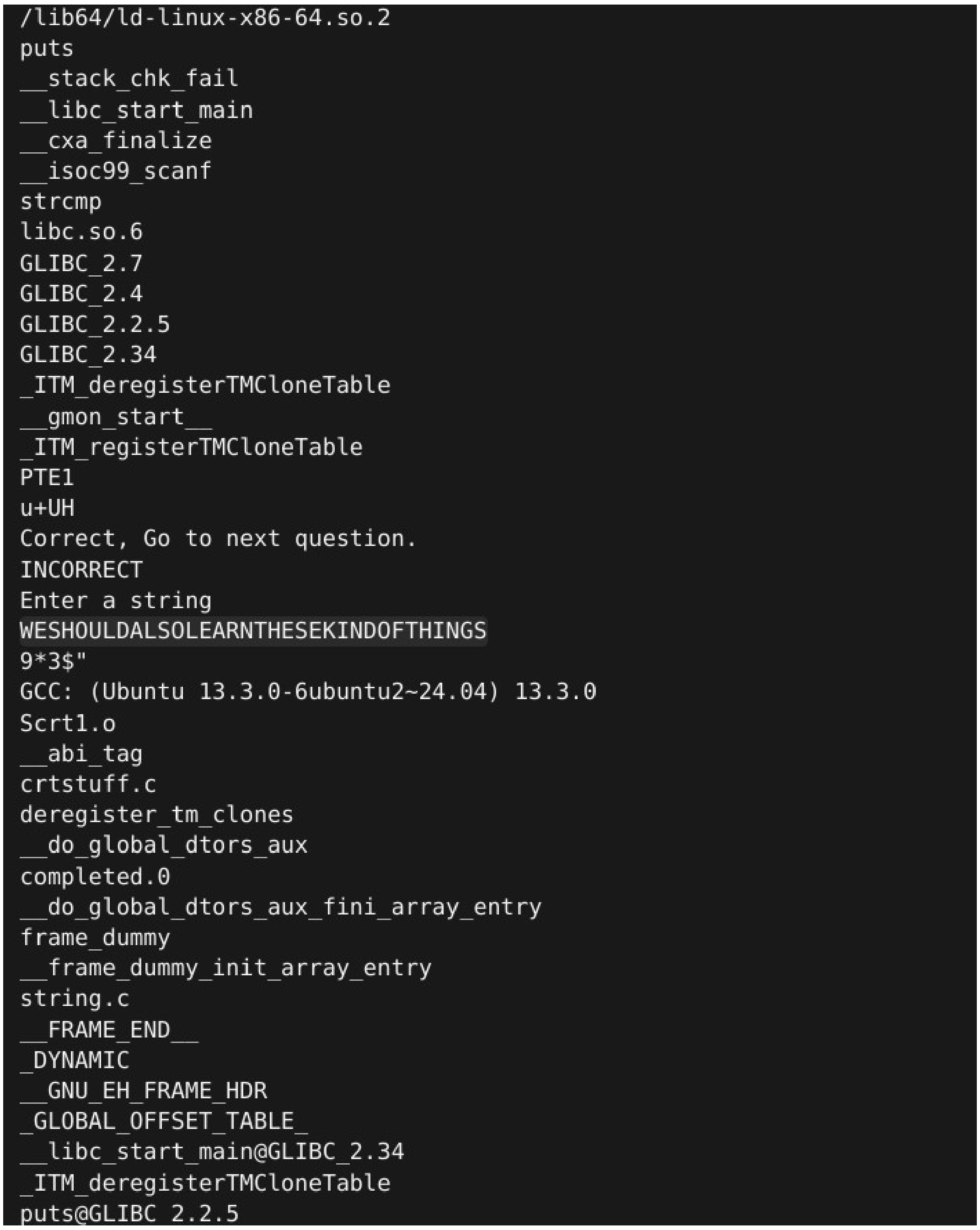
here we saw that the main function is starting at location 11dc.
A string is loaded at 0x2030 (lea rax, [rip+0xe32] at 0x11f7) and printed via puts@plt at 0x1201.
This string could be the prompt string that is getting printed after running the problem like "Enter a string".
Then a buffer is allocated at rbp-0x70 (lea rax, [rbp-0x70] at 0x1206) for user input via scanf.
The input buffer is compared with a hardcoded string at 0x2048 (lea rdx, [rip+0xe1c] at 0x1225) using strcmp at 0x1232.

The comparison result is tested at 0x1237 (test eax, eax). If equal (0), it calls <success> at 0x1240, which loads the success message at 0x2008 (lea rax, [rip+0xe50] at 0x11b1) and calls puts(printf) and prints some string. If not equal, it calls <failure> at 0x124c, which loads some string at 0x2026 (lea rax, [rip+0xe54] at 0x11cb) and loops infinitely (jmp 0x11cb at 0x11da).

Now we have to get the correct string. So for this I used this command to extract all strings from the program and stored it in a file prob1_strings.txt using the command:

```
strings Prob1 > prob1_strings.txt
```

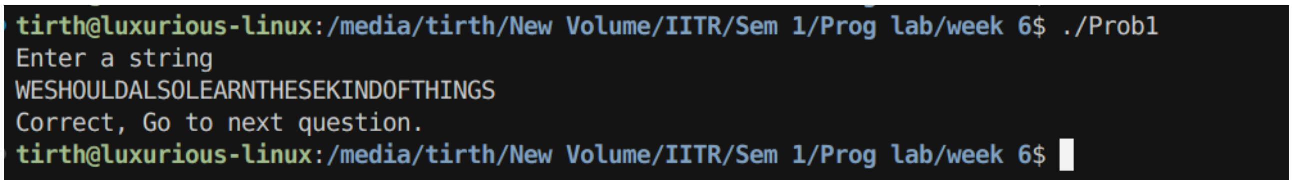
The file looks like:



```
/lib64/ld-linux-x86-64.so.2
puts
__stack_chk_fail
__libc_start_main
__cxa_finalize
__isoc99_scanf
strcmp
libc.so.6
GLIBC_2.7
GLIBC_2.4
GLIBC_2.2.5
GLIBC_2.34
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
PTE1
u+UH
Correct, Go to next question.
INCORRECT
Enter a string
WESHOULDALSOLEARNTHESEKINDOFTHINGS
9*3$"
GCC: (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
Scrt1.o
__abi_tag
crtstuff.c
deregister_tm_clones
__do_global_dtors_aux
completed.0
__do_global_dtors_aux_fini_array_entry
frame_dummy
__frame_dummy_init_array_entry
string.c
__FRAME_END__
_DYNAMIC
__GNU_EH_FRAME_HDR
_GLOBAL_OFFSET_TABLE_
__libc_start_main@GLIBC_2.34
_ITM_deregisterTMCloneTable
puts@GLIBC_2.2.5
```

Here there are very less human readable strings like “Correct, Go to next question.”, “INCORRECT”, “Enter a string”, “WESHOULDALSOLEARNTHESEKINDOFTHINGS”.

From this we can say that the other three are the output strings. So the only string left is “ WESHOULDALSOLEARNTHESEKINDOFTHINGS” which is our input string at which the program will halt and print the correct message.

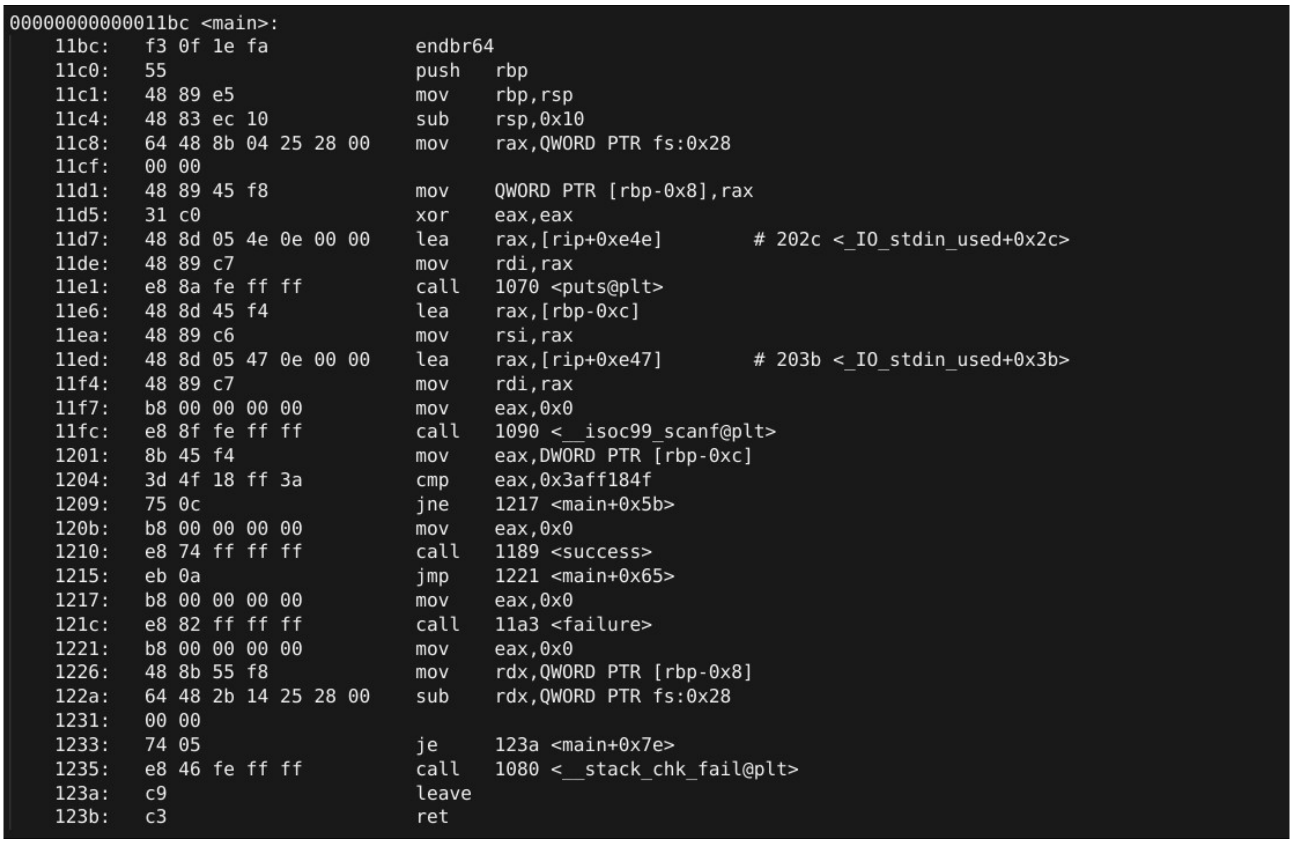


```
tirth@luxurious-linux:/media/tirth/New Volume/IITR/Sem 1/Prog lab/week 6$ ./Prob1
Enter a string
WESHOULDALSOLEARNTHESEKINDOFTHINGS
Correct, Go to next question.
tirth@luxurious-linux:/media/tirth/New Volume/IITR/Sem 1/Prog lab/week 6$
```

Problem 2:

First, I converted the binary to assembly and stored the output in a file prob2_asm.txt using the cmd:
objdump -d -M intel Prob2 > prob2_asm.txt

The main function of the above assembly code looks like:



```
00000000000011bc <main>:
    11bc:	f3 0f 1e fa          	endbr64
    11c0:	55                   	push   rbp
    11c1:	48 89 e5             	mov    rbp,rsp
    11c4:	48 83 ec 10          	sub    rsp,0x10
    11c8:	64 48 8b 04 25 28 00 	mov    rax,QWORD PTR fs:0x28
    11cf:	00 00 
    11d1:	48 89 45 f8          	mov    QWORD PTR [rbp-0x8],rax
    11d5:	31 c0                	xor    eax,eax
    11d7:	48 8d 05 4e 0e 00 00 	lea    rax,[rip+0xe4e]        # 202c <_IO_stdin_used+0x2c>
    11de:	48 89 c7             	mov    rdi,rax
    11e1:	e8 8a fe ff ff       	call   1070 <puts@plt>
    11e6:	48 8d 45 f4          	lea    rax,[rbp-0xc]
    11ea:	48 89 c6             	mov    rsi,rax
    11ed:	48 8d 05 47 0e 00 00 	lea    rax,[rip+0xe47]        # 203b <_IO_stdin_used+0x3b>
    11f4:	48 89 c7             	mov    rdi,rax
    11f7:	b8 00 00 00 00       	mov    eax,0x0
    11fc:	e8 8f fe ff ff       	call   1090 <__isoc99_scanf@plt>
    1201:	8b 45 f4             	mov    eax,DWORD PTR [rbp-0xc]
    1204:	3d 4f 18 ff 3a       	cmp    eax,0x3aff184f
    1209:	75 0c                	jne    1217 <main+0x5b>
    120b:	b8 00 00 00 00       	mov    eax,0x0
    1210:	e8 74 ff ff ff       	call   1189 <success>
    1215:	eb 0a                	jmp    1221 <main+0x65>
    1217:	b8 00 00 00 00       	mov    eax,0x0
    121c:	e8 82 ff ff ff       	call   11a3 <failure>
    1221:	b8 00 00 00 00       	mov    eax,0x0
    1226:	48 8b 55 f8          	mov    rdx,QWORD PTR [rbp-0x8]
    122a:	64 48 2b 14 25 28 00 	sub    rdx,QWORD PTR fs:0x28
    1231:	00 00 
    1233:	74 05                	je     123a <main+0x7e>
    1235:	e8 46 fe ff ff       	call   1080 <__stack_chk_fail@plt>
    123a:	c9                   	leave
    123b:	c3                   	ret
```

here the main function is starting at 0x11bc.

A string is loaded at 0x202c (lea rax, [rip+0xe4e] at 0x11d7) and printed via puts(printf) at 0x11e1.

A buffer is allocated at rbp-0xc (lea rax, [rbp-0xc] at 0x11e6) for user input via scanf at 0x11fc.

The input is loaded into eax at 0x1201 (mov eax, DWORD PTR [rbp-0xc]).

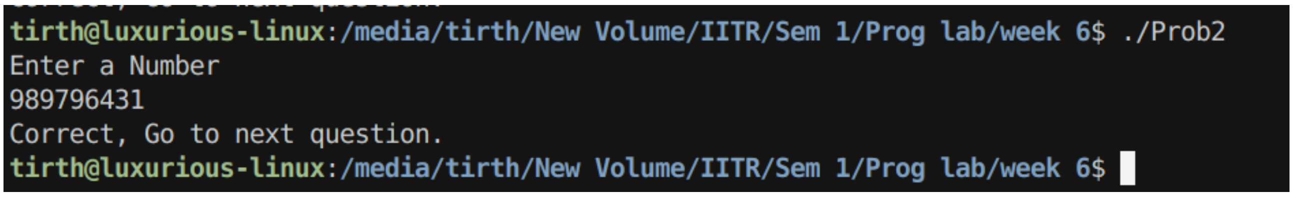
Then the input at eax is compared with 0x3aff184f using cmp at 0x1204.

If they're equal it calls <success> at 0x1210 and prints the success message and if they're not equal it calls <failure> at 0x121c and prints the failure message.

So, now we have to convert the hexadecimal number 0x3aff184f to decimal to get the correct input.

We can convert to decimal using the command:
echo "ibase=16; 3AFF184F" | bc

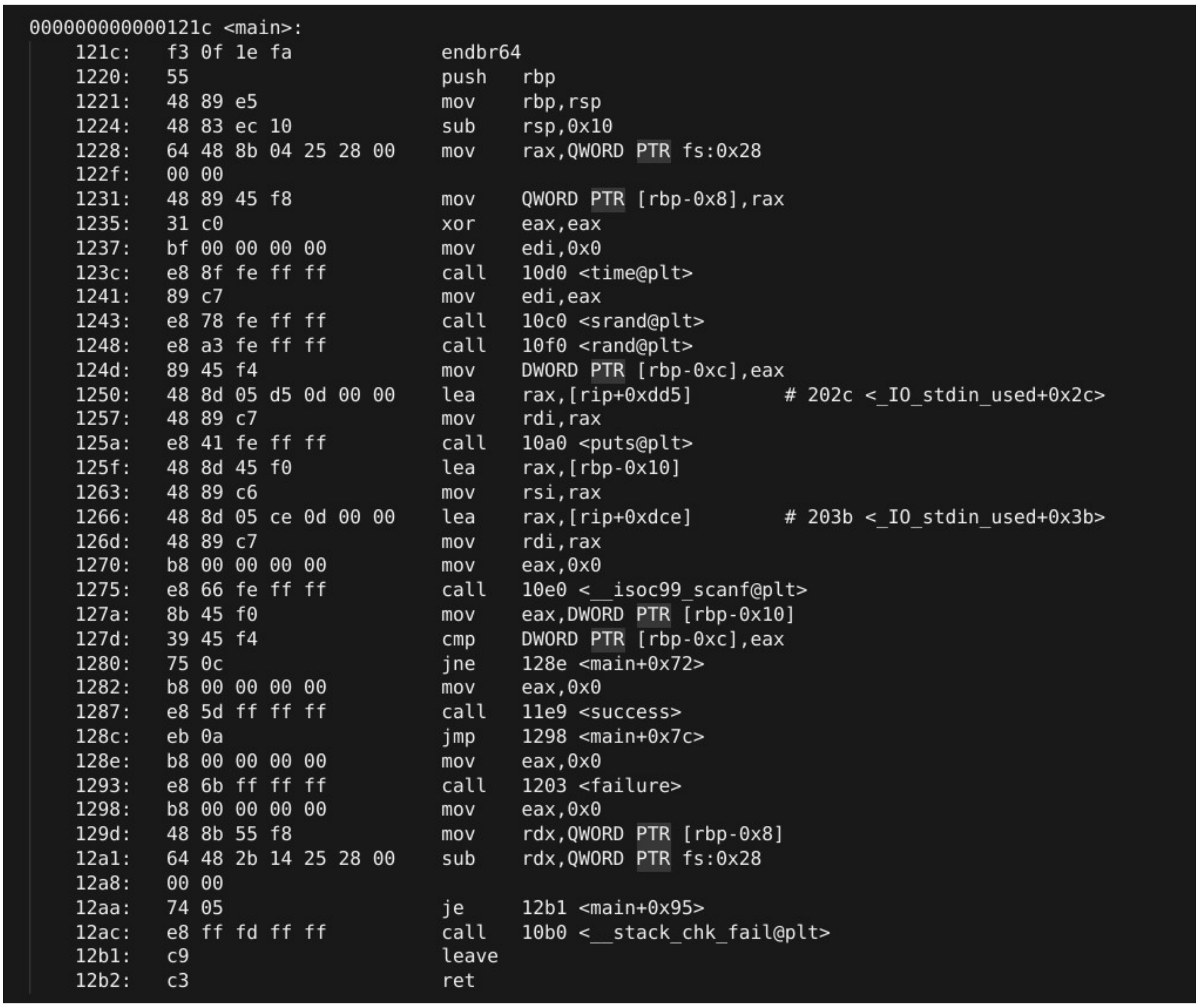
which will give "989796431" which indeed is our input at which the program halts.



```
tirth@luxurious-linux:/media/tirth/New Volume/IITR/Sem 1/Prog lab/week 6$ ./Prob2
Enter a Number
989796431
Correct, Go to next question.
tirth@luxurious-linux:/media/tirth/New Volume/IITR/Sem 1/Prog lab/week 6$
```

Problem 3:

Convert the binary to assembly and saved the output to a file prob3_asm.txt using the command: objdump -d -M intel Prob3 > prob3_asm.txt



```
000000000000121c <main>:
    121c:   f3 0f 1e fa             endbr64
    1220:   55                      push   rbp
    1221:   48 89 e5                mov    rbp,rsp
    1224:   48 83 ec 10             sub    rsp,0x10
    1228:   64 48 8b 04 25 28 00    mov    rax,QWORD PTR fs:0x28
    122f:   00 00
    1231:   48 89 45 f8             mov    QWORD PTR [rbp-0x8],rax
    1235:   31 c0                   xor    eax,eax
    1237:   bf 00 00 00 00          mov    edi,0x0
    123c:   e8 8f fe ff ff          call   10d0 <time@plt>
    1241:   89 c7                   mov    edi,eax
    1243:   e8 78 fe ff ff          call   10c0 <srand@plt>
    1248:   e8 a3 fe ff ff          call   10f0 <rand@plt>
    124d:   89 45 f4                mov    DWORD PTR [rbp-0xc],eax
    1250:   48 8d 05 d5 0d 00 00    lea    rax,[rip+0xdd5]        # 202c <_IO_stdin_used+0x2c>
    1257:   48 89 c7                mov    rdi,rax
    125a:   e8 41 fe ff ff          call   10a0 <puts@plt>
    125f:   48 8d 45 f0             lea    rax,[rbp-0x10]
    1263:   48 89 c6                mov    rsi,rax
    1266:   48 8d 05 ce 0d 00 00    lea    rax,[rip+0xdce]        # 203b <_IO_stdin_used+0x3b>
    126d:   48 89 c7                mov    rdi,rax
    1270:   b8 00 00 00 00          mov    eax,0x0
    1275:   e8 66 fe ff ff          call   10e0 <__isoc99_scanf@plt>
    127a:   8b 45 f0                mov    eax,DWORD PTR [rbp-0x10]
    127d:   39 45 f4                cmp    DWORD PTR [rbp-0xc],eax
    1280:   75 0c                   jne    128e <main+0x72>
    1282:   b8 00 00 00 00          mov    eax,0x0
    1287:   e8 5d ff ff ff          call   11e9 <success>
    128c:   eb 0a                   jmp    1298 <main+0x7c>
    128e:   b8 00 00 00 00          mov    eax,0x0
    1293:   e8 6b ff ff ff          call   1203 <failure>
    1298:   b8 00 00 00 00          mov    eax,0x0
    129d:   48 8b 55 f8             mov    rdx,QWORD PTR [rbp-0x8]
    12a1:   64 48 2b 14 25 28 00    sub    rdx,QWORD PTR fs:0x28
    12a8:   00 00
    12aa:   74 05                   je     12b1 <main+0x95>
    12ac:   e8 ff fd ff ff          call   10b0 <__stack_chk_fail@plt>
    12b1:   c9                      leave
    12b2:   c3                      ret
```

The main is starting at 0x121c.

It calls time@plt at 0x123c (mov edi, 0x0) and srand@plt at 0x1243 (mov edi, eax) to seed the random number generator.

Then it calls rand@plt at 0x1248, storing result at rbp-0xc (mov DWORD PTR [rbp-0xc], eax at 0x124d).

Loads a string at 0x202c (lea rax, [rip+0xdd5] at 0x1250) and printed via puts@plt at 0x125a.

Allocates buffer at rbp-0x10 (lea rax, [rbp-0x10] at 0x125f) for input via scanf at 0x1275.

Loads input into eax at 0x127a (mov eax, DWORD PTR [rbp-0x10]).

Compares random number at rbp-0xc with input at 0x127d (cmp DWORD PTR [rbp-0xc], eax).

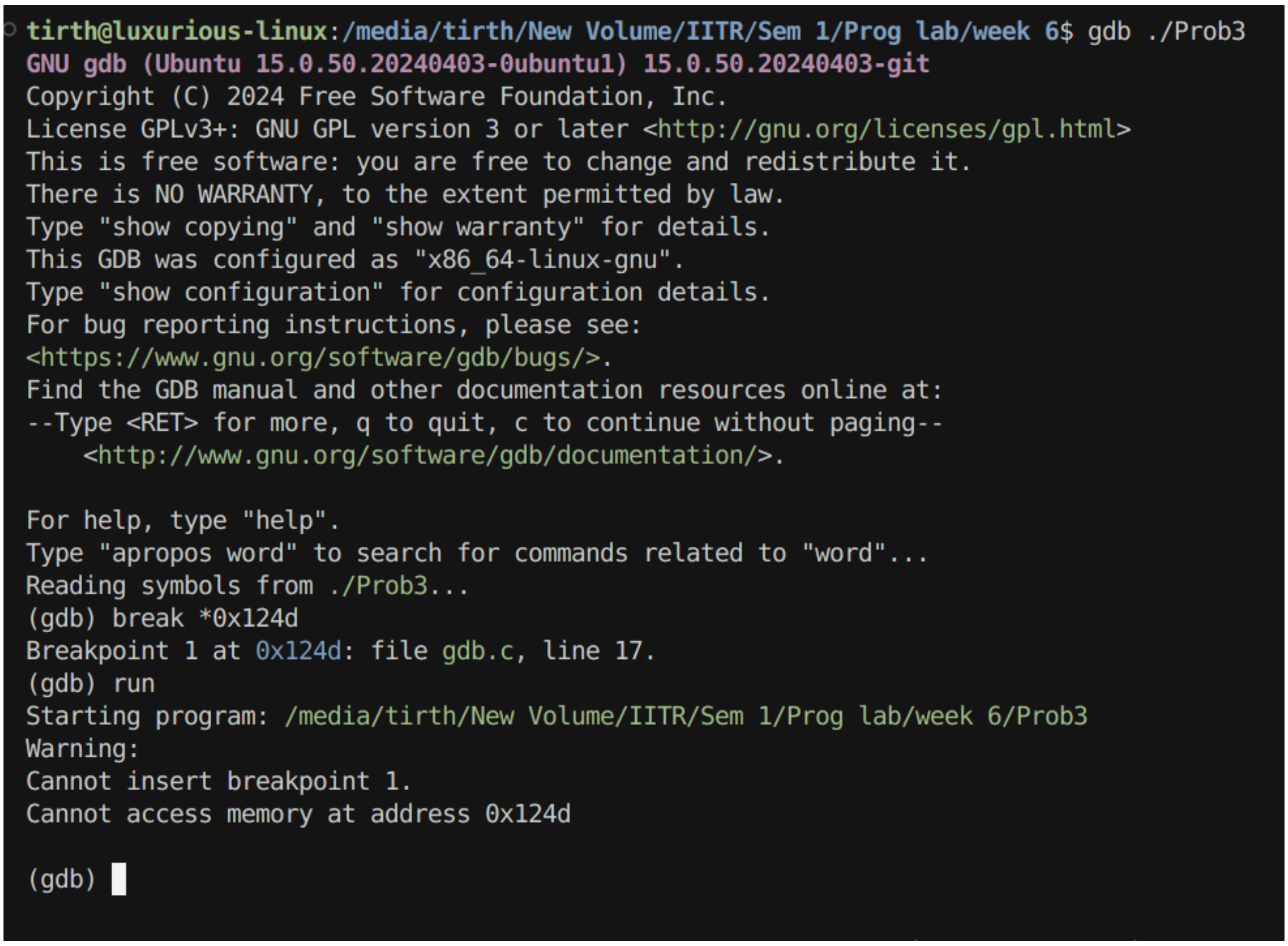
If equal call <success> at 0x1287 and prints the success message. And if not equal call <failure> at 0x1293 and prints the failure message.

Here we're comparing the user input with a randomly generated integer. So it is almost impossible to guess the input to get the success message. Hence we have to use gdb debugger to set the breakpoints after the random number is generated and then we have to give that random number as input to print the success message.

We'll use gdb debugger using:
gdb ./Prob3

and then we'll set breakpoint after the random number is generated at 0x124d

and then we'll start using "run".



```
tirth@luxurious-linux:/media/tirth/New Volume/IITR/Sem 1/Prog lab/week 6$ gdb ./Prob3
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
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.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
--Type <RET> for more, q to quit, c to continue without paging--
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./Prob3...
(gdb) break *0x124d
Breakpoint 1 at 0x124d: file gdb.c, line 17.
(gdb) run
Starting program: /media/tirth/New Volume/IITR/Sem 1/Prog lab/week 6/Prob3
Warning:
Cannot insert breakpoint 1.
Cannot access memory at address 0x124d

(gdb) 
```

And then we have the inspect the random number stored in rbp-0xc using

x/d $rbp-0xc

and then we prompted for input we have to give that number for the program to halt.

(I am not able to set break point and get the random number generated. I tried so so hard but getting memory access error. But I wrote the method what we have to do).

Problem 4:

First I compiled the prob 4 using for 32 bit :
gcc -m32 Prob4.c -o Prob4

but got this compilation error:
/usr/include/stdio.h:28:10: fatal error: bits/libc-header-start.h: No such file or directory

fixed the compilation error using:
sudo apt-get install gcc-multilib g++-multilib libc6-dev-i386

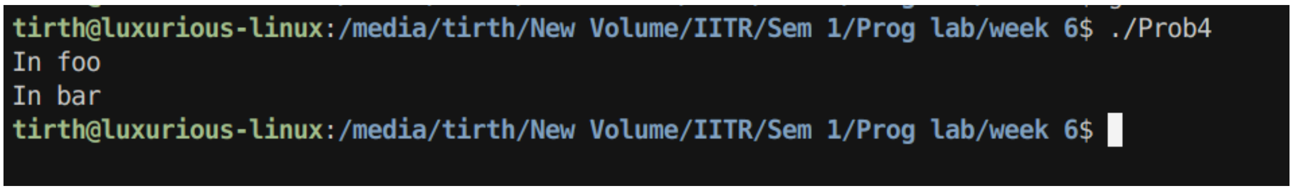
Then I ran using:

./Prob4

and it gave "In bar" in output infinite times.

Compiler for 64 bit using:
gcc Prob4.c -o Prob4

and got this output:



```
tirth@luxurious-linux:/media/tirth/New Volume/IITR/Sem 1/Prog lab/week 6$ ./Prob4
In foo
In bar
tirth@luxurious-linux:/media/tirth/New Volume/IITR/Sem 1/Prog lab/week 6$
```

The reason for this infinite looping issue is that the bar function is overwriting the return address that the address is pushed before calling the bar function. So after the bar function is called the return address is overwritten and again the address of bar function is pushed so it is called infinie times.

Not able to under how to fix this infinite looping issue.