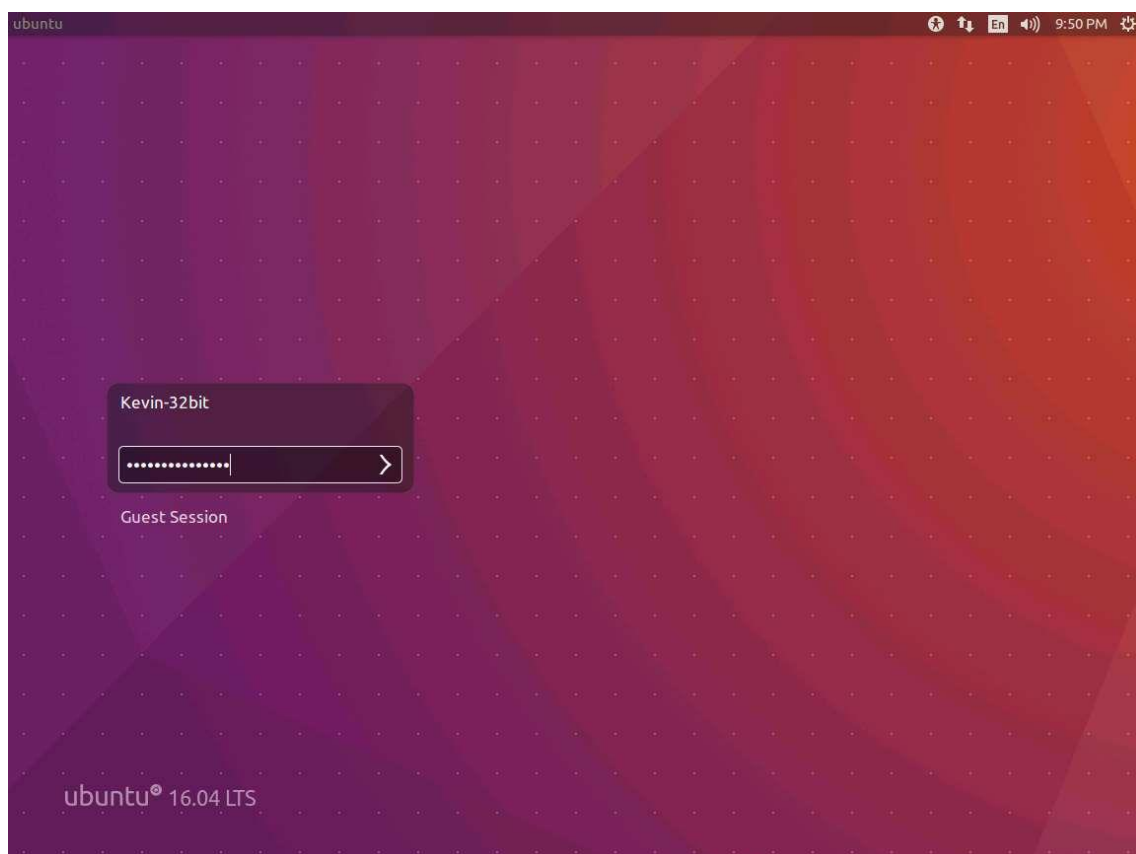


## Module 13: Binary Analysis and Exploitation

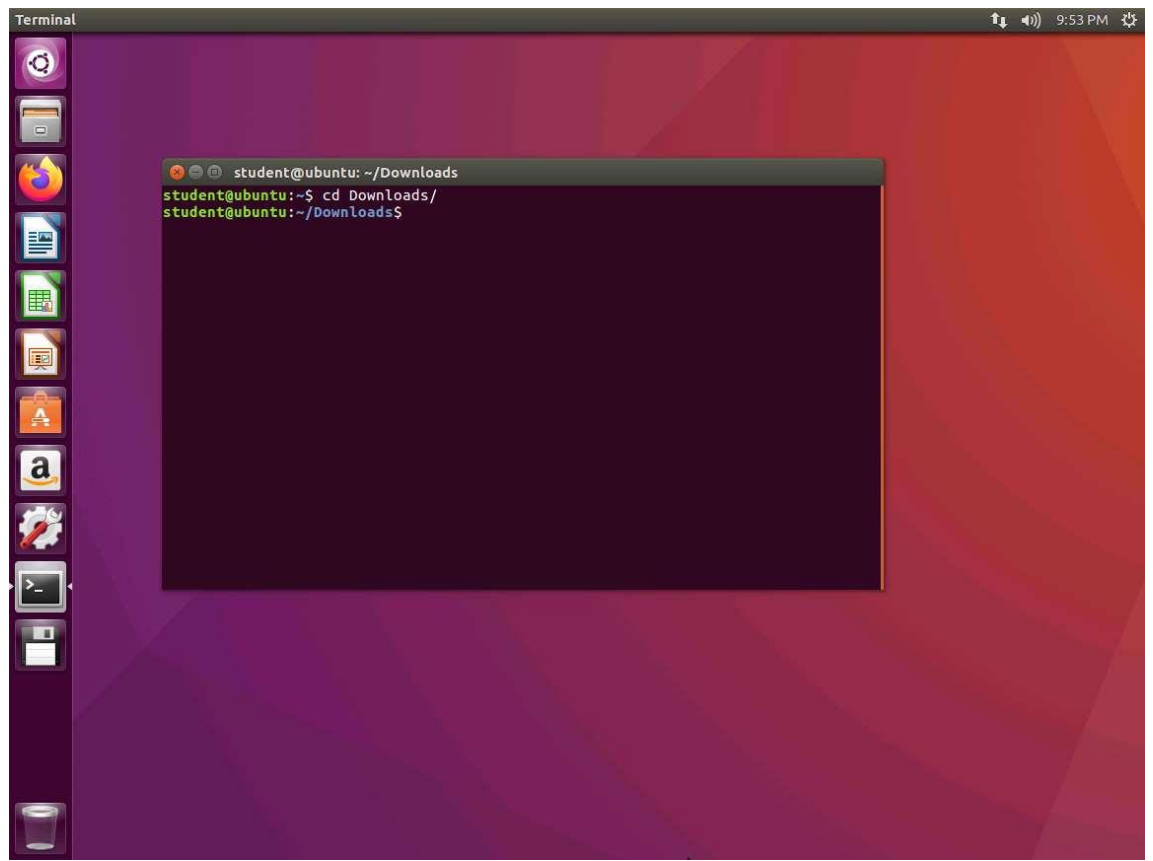
### Exercise 1: Binary Analysis

---

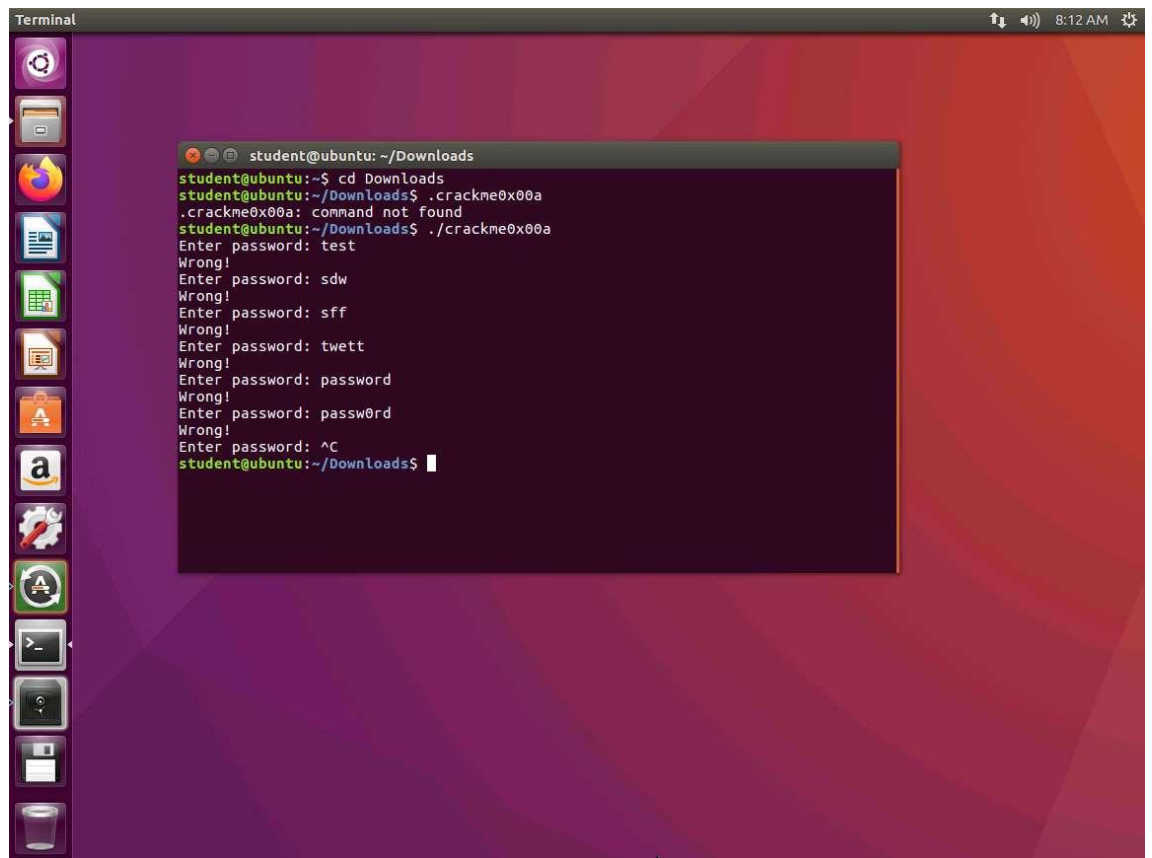
1. ☐ Login to the [Software-Test-Linux-32bit](#) machine using **studentpassword** as Password.



2. ☐ Open a terminal window, and enter **cd Downloads**.

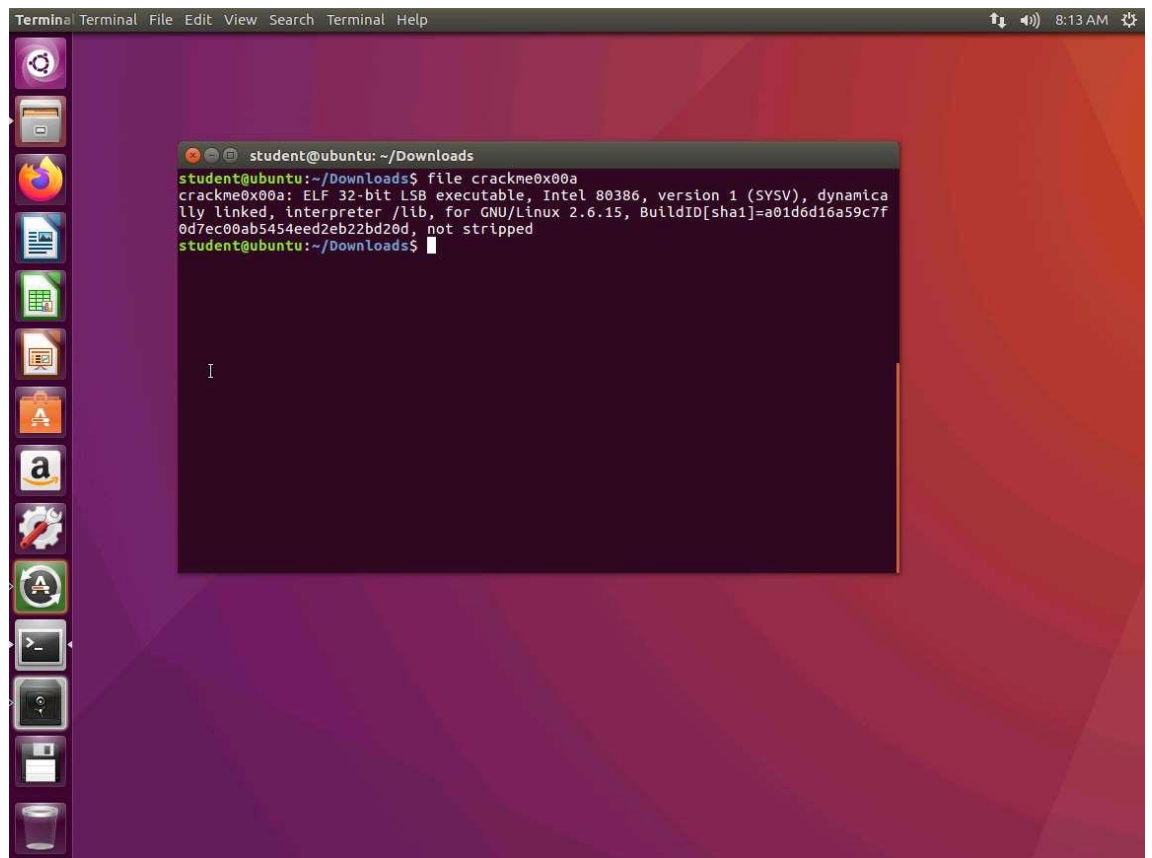


3. ☐ Once you are in the folder, enter **./crackme0x00a**. In the 64-bit machine, the program will not run since it is not built for 64 bit, so we will continue with the 32-bit machine for now. Once you run the program, enter some passwords to see if you can determine what the password is. An example of this is shown in the following screenshot.

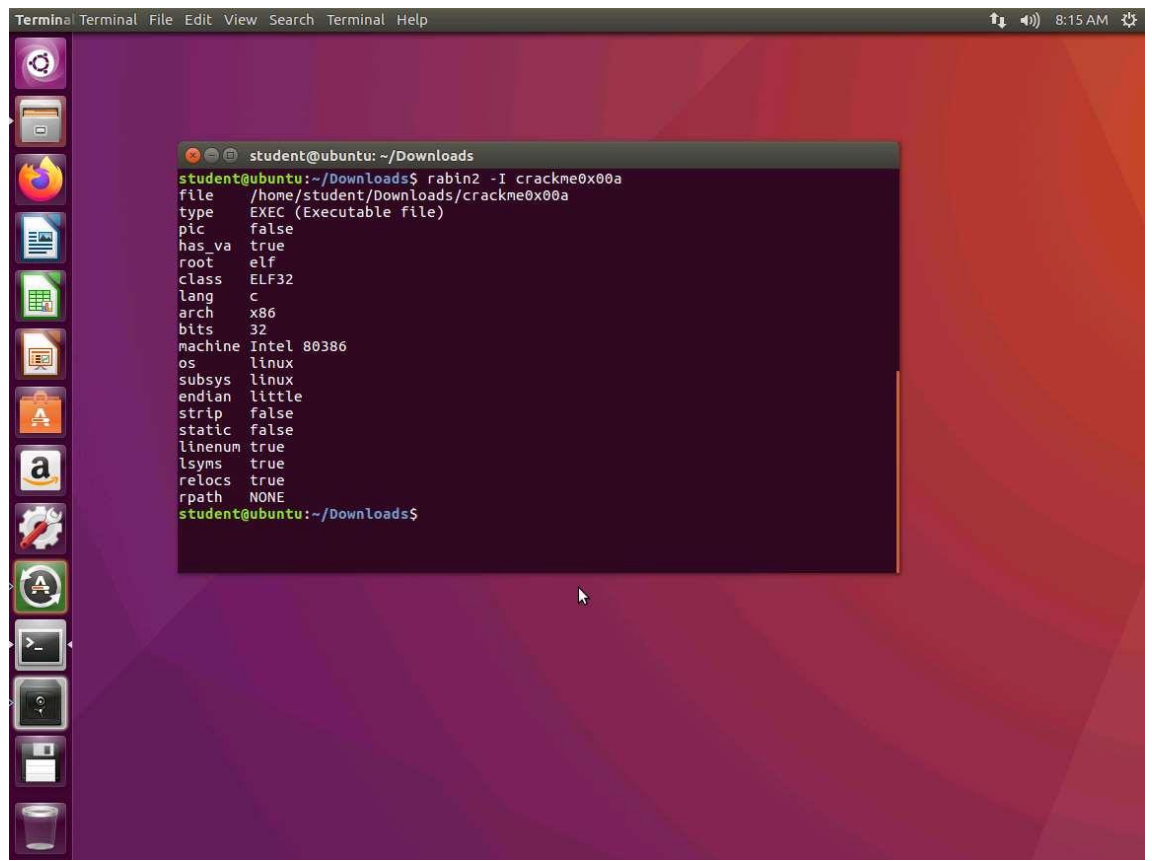
A screenshot of an Ubuntu desktop environment. The background is a purple and red geometric pattern. On the left side, there is a vertical dock with various application icons including Dash, Home Folder, Firefox, LibreOffice Writer, LibreOffice Calc, LibreOffice Impress, Amazon, and others. A terminal window is open in the center, titled 'Terminal'. The terminal shows the following commands and output:

```
student@ubuntu: ~/Downloads
student@ubuntu:~$ cd Downloads
student@ubuntu:~/Downloads$ ./crackme0x00a
./crackme0x00a: command not found
student@ubuntu:~/Downloads$ ./crackme0x00a
Enter password: test
Wrong!
Enter password: sdw
Wrong!
Enter password: sff
Wrong!
Enter password: twett
Wrong!
Enter password: password
Wrong!
Enter password: passw0rd
Wrong!
Enter password: ^C
student@ubuntu:~/Downloads$
```

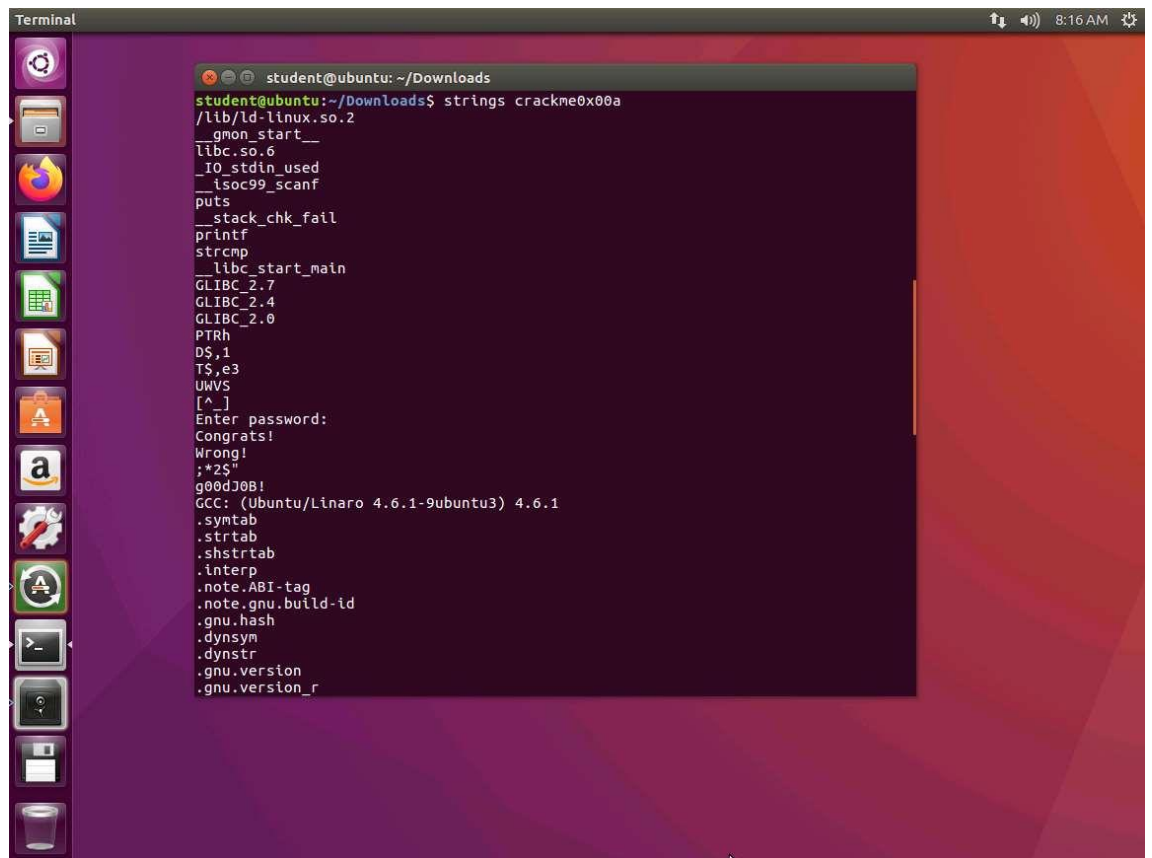
4. ☐ Since we could not guess it, we now need to perform an analysis and see what we can learn about the file. We will use the file command. Enter **file crackme0x00a**. The output of this command is shown in the following screenshot.



5. ☐ As the above screenshot shows, we have an executable and linking format (ELF) 32 bit executable. The file is 32-bit, LSB executable (least-significant byte). It means that the file is little-endian.
6. ☐ We will use another tool. Enter **rabin2 -I crackme0x00a**. An example of the output of this command is shown in the following screenshot.



7. ☐ Let us now use powerful tool strings to see what we can discover in the binary. Enter **strings crackme0x00a**. An example of the output of this command is shown in the following screenshot.

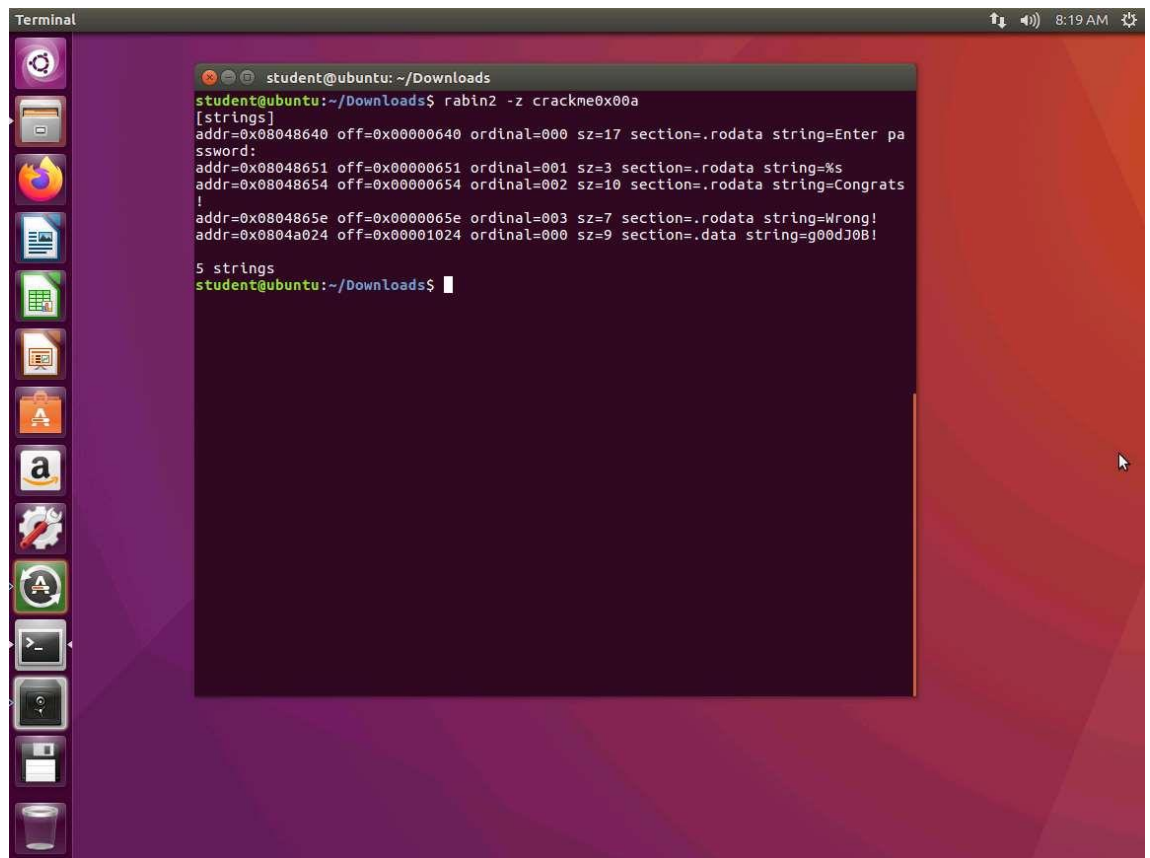


```
Terminal
student@ubuntu: ~/Downloads
student@ubuntu:~/Downloads$ strings crackme0x00a
/lib/ld-linux.so.2
__gmon_start__
libc.so.6
_IO_stdin_used
__isoc99_scanf
puts
__stack_chk_fail
printf
strcmp
__libc_start_main
GLIBC_2.7
GLIBC_2.4
GLIBC_2.0
PTRh
D$,1
T$,e3
UWVS
[^_]
Enter password:
Congrats!
Wrong!
;*2$"
g00dJ0B!
GCC: (Ubuntu/Linaro 4.6.1-9ubuntu3) 4.6.1
.symtab
.strtab
.shstrtab
.interp
.note.ABI-tag
.note.gnu.build-id
.gnu.hash
.dynsym
.dynstr
.gnu.version
.gnu.version_r
```

8. ☐ As you review the strings, do you see anything of interest? We have the prompt for the password followed by what appears to be two responses and then a string. This could be the password, but it seems too easy. We will continue to explore the file further.
9. ☐ In the terminal window, enter **xxd crackme0x00a | more**. The output of this command is shown in the following screenshot.

```
student@ubuntu: ~/Downloads
student@ubuntu:~/Downloads$ xxd crackme0x00a | more
00000000: 7f45 4c46 0101 0100 0000 0000 0000 0000 .ELF.....
00000010: 0200 0300 0100 0000 3084 0408 3400 0000 .....0...4...
00000020: 5811 0000 0000 0000 3400 2000 0900 2800 X.....4...(.
00000030: 1e00 1b00 0600 0000 3400 0000 3480 0408 .....4...4...
00000040: 3480 0408 2001 0000 2001 0000 0500 0000 4.....T...T...
00000050: 0400 0000 0300 0000 5401 0000 5481 0408 T.....
00000060: 5481 0408 1300 0000 1300 0000 0400 0000 .....
00000070: 0100 0000 0100 0000 0000 0000 0080 0408 .....
00000080: 0080 0408 6007 0000 6007 0000 0500 0000 .....
00000090: 0010 0000 0100 0000 140f 0000 149f 0408 .....
000000a0: 149f 0408 1c01 0000 2401 0000 0600 0000 .....S.....
000000b0: 0010 0000 0200 0000 280f 0000 289f 0408 .....(.....
000000c0: 289f 0408 c800 0000 c800 0000 0600 0000 .....
000000d0: 0400 0000 0400 0000 6801 0000 6881 0408 .....h...h...
000000e0: 6881 0408 4400 0000 4400 0000 0400 0000 h...D...D...h...
000000f0: 0400 0000 50e5 7464 6806 0000 6886 0408 ...P.tdh...h...
00000100: 6886 0408 3400 0000 3400 0000 0400 0000 h...4...4...
00000110: 0400 0000 51e5 7464 0000 0000 0000 0000 ...Q.td.....
00000120: 0000 0000 0000 0000 0000 0000 0600 0000 .....R.td.....
00000130: 0400 0000 52e5 7464 140f 0000 149f 0408 ...../lib/ld-linu
00000140: 149f 0408 ec00 0000 ec00 0000 0400 0000 x.so.2.....
00000150: 0100 0000 2f6c 6962 2f6c 642d 6c69 6e75 ...GNU.....
00000160: 782e 736f 2e32 0000 0400 0000 1000 0000 ...GNU...m...
00000170: 0100 0000 474e 5500 0000 0000 0200 0000 ~...EN...+...
00000180: 0600 0000 0f00 0000 0400 0000 1400 0000 .....K...
00000190: 0300 0000 474e 5500 a01d 6d16 a59c 7f0d .....U...
000001a0: 7ec0 0ab5 454e ed2e b22b d20d 0200 0000 .....N...
000001b0: 0800 0000 0100 0000 0500 0000 0020 0020 .....8...
000001c0: 0000 0000 0800 0000 ad4b e3c0 0000 0000 .....
000001d0: 0000 0000 0000 0000 0000 0000 5500 0000 .....
000001e0: 0000 0000 0000 0000 1200 0000 4e00 0000 .....
000001f0: 0000 0000 0000 0000 1200 0000 3800 0000 .....
00000200: 0000 0000 0000 0000 1200 0000 3800 0000 .....
00000210: 0000 0000 0000 0000 1200 0000 0100 0000 .....
00000220: 0000 0000 0000 0000 2000 0000 5c00 0000 .....\\...
```

10. ☐ To use rabin2 to crack the file, you will need to execute with a different parameter than the one we used at the information gathering process. If you refer to the manual, you will see that the parameter **-z** is used to show strings inside **.data** section (similar to gnu strings).
11. ☐ In the terminal window, enter **rabin2 -z crackme0x00a**. An example of this is shown in the following screenshot.

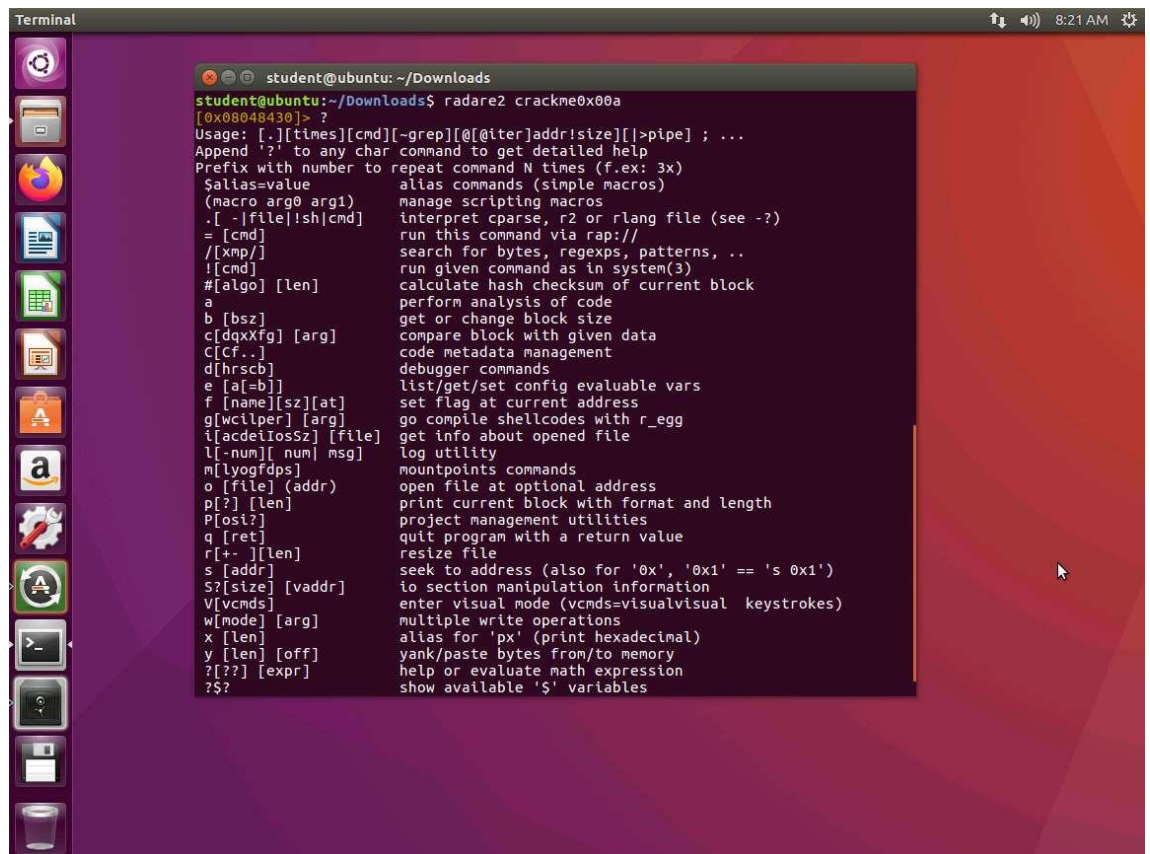


```
Terminal
student@ubuntu: ~/Downloads
student@ubuntu:~/Downloads$ rabin2 -z crackme0x00a
[strings]
addr=0x08048640 off=0x00000640 ordinal=000 sz=17 section=.rodata string=Enter password:
addr=0x08048651 off=0x00000651 ordinal=001 sz=3 section=.rodata string=%s
addr=0x08048654 off=0x00000654 ordinal=002 sz=10 section=.rodata string=Congrats!
addr=0x0804865e off=0x0000065e ordinal=003 sz=7 section=.rodata string=Wrong!
addr=0x0804a024 off=0x00001024 ordinal=000 sz=9 section=.data string=g00dJ0B!

5 strings
student@ubuntu:~/Downloads$
```

12. ☐ Next, we will use the Radare2 tool to look at the executable. In the terminal window, enter **radare2 crackme0x00a**. Once the program is entered, enter **?**. This will allow you to review the different options. An example of the output of this command is shown in the following screenshot.



A screenshot of a terminal window on an Ubuntu desktop. The terminal title is "Terminal". The prompt is "student@ubuntu: ~/Downloads". The command "radare2 crackme0x00a" has been executed. The output shows the usage and a list of commands with their descriptions. The desktop background is purple, and the left sidebar shows various application icons.

```
student@ubuntu: ~/Downloads$ radare2 crackme0x00a
[0x08048430]> ?
Usage: [.] [times] [cmd] [-grep] [iter] [addr] [size] [pipe] ; ...
Append '?' to any char command to get detailed help
Prefix with number to repeat command N times (f.ex: 3x)
$alias=value      alias commands (simple macros)
(macro arg0 arg1)  manage scripting macros
.[ -|file|!sh|cmd] interpret cparse, r2 or rlang file (see -?)
=[cmd]            run this command via rap://
/[xmp/]           search for bytes, regexps, patterns, ..
![cmd]           run given command as in system(3)
#[algo] [len]    calculate hash checksum of current block
a               perform analysis of code
b [bsz]         get or change block size
c [dqxf] [arg]  compare block with given data
C [cf..]        code metadata management
d [hrscb]       debugger commands
e [a=b]         list/get/set config evaluable vars
f [name][sz][at] set flag at current address
g [wclpr] [arg] go compile shellcodes with r_egg
i [acdeliossz] [file] get info about opened file
l [-num] [num] [msg] log utility
m [lyogfdps]    mountpoints commands
o [file] (addr) open file at optional address
p [?] [len]     print current block with format and length
P [osi?]       project management utilities
q [ret]        quit program with a return value
r [+ -] [len]   resize file
s [addr]       seek to address (also for '0x', '0x1' == 's 0x1')
S? [size] [vaddr] io section manipulation information
V [vcmds]      enter visual mode (vcmds=visualvisual keystrokes)
w [mode] [arg] multiple write operations
x [len]       alias for 'px' (print hexadecimal)
y [len] [off] yank/paste bytes from/to memory
? [??] [expr]  help or evaluate math expression
??           show available '$' variables
```

13. ☐ We now want to run the disassemble function. Enter **pdf @ main**. The output of this command is shown in the following screenshot.

```

Terminal Terminal File Edit View Search Terminal Help
student@ubuntu: ~/Downloads
?@? show help for '@' and '~' suffix
[0x08048430]> pdf @ main
Cannot find function at 0x080484e4
Cannot find function at 0x080484e4
;-- sym.main:
0x080484e4 55 push ebp
0x080484e5 89e5 mov ebp, esp
0x080484e7 83e4f0 and esp, 0xfffffff0
0x080484ea 83ec30 sub esp, 0x30
0x080484ed 65a114000000 mov eax, [gs:0x14]
0x080484f3 8944242c mov [esp+0x2c], eax
0x080484f7 31c0 xor eax, eax
0x080484f9 b840860408 mov eax, str.Enterpassword
0x080484fe 890424 mov [esp], eax
0x08048501 e8cafeffff call 0x1080483d0
0x080483d0(unk) ; sym.imp.printf
0x08048506 b851860408 mov eax, str.s
0x0804850b 8d542413 lea edx, [esp+0x13]
0x0804850f 89542404 mov [esp+0x4], edx
0x08048513 890424 mov [esp], eax
0x08048516 e805ffff call 0x108048420
0x08048420(unk) ; sym.imp._isoc99_scanf
0x0804851b 8d442413 lea eax, [esp+0x13]
0x0804851f 89442404 mov [esp+0x4], eax
0x08048523 c7042424a00. mov dword [esp], sym.pass.1685
0x0804852a e891feffff call 0x1080483c0
0x080483c0(unk) ; sym.imp.strcmp
0x0804852f 85c0 test eax, eax
0x08048531 7521 jnz 0x8048554
0x08048533 c7042454860. mov dword [esp], str.Congrats
0x0804853a e8b1feffff call 0x1080483f0
0x080483f0(unk) ; sym.imp.puts
0x0804853f 90 nop
0x08048540 b800000000 mov eax, 0x0
0x08048545 8b54242c mov edx, [esp+0x2c]
0x08048549 65331514000. xor edx, [gs:0x14]
0x08048550 7415 jz 0x8048567
0x08048552 eb0e jmp 0x8048562
0x08048554 c704245e860. mov dword [esp], str.Wrong
0x0804855b e890feffff call 0x1080483f0
0x080483f0(unk) ; sym.imp.puts
0x08048560 eb97 jmp 0x1080484f9
0x08048562 e879feffff call 0x1080483e0

```

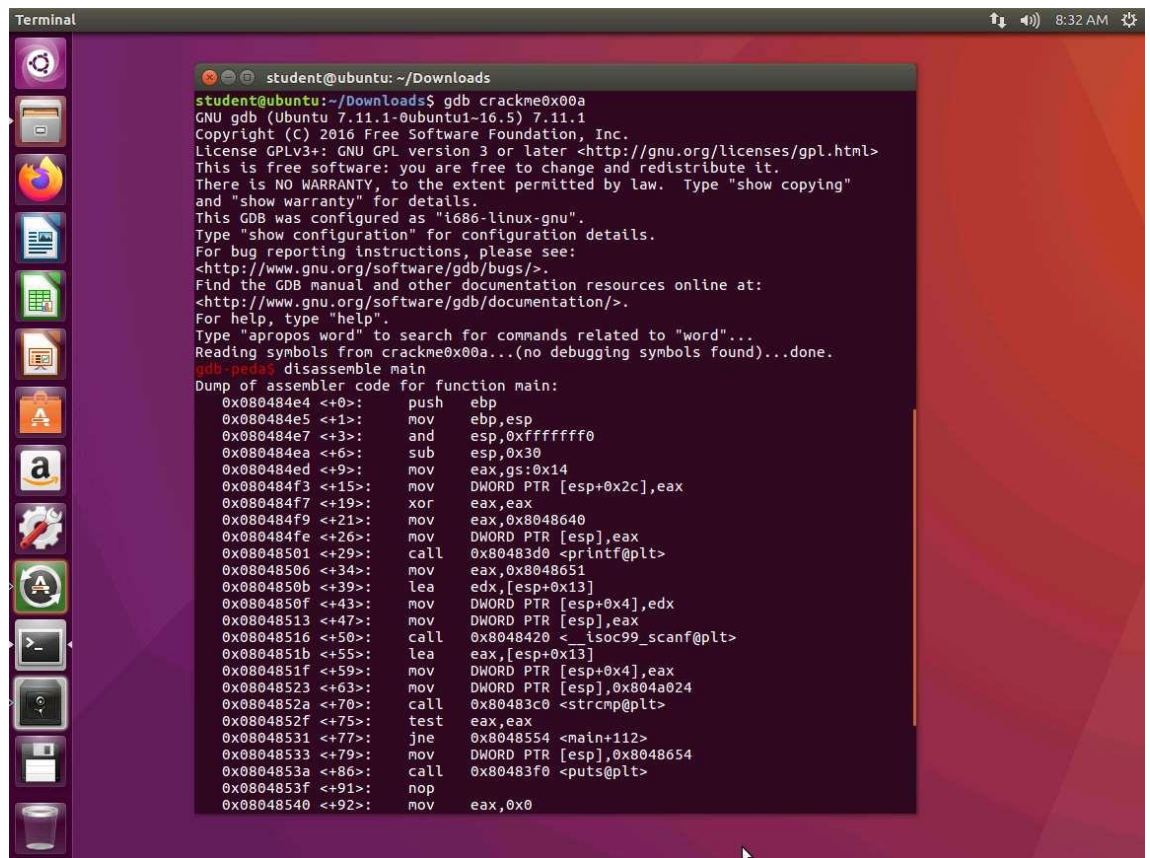
14. ☐ Take a few minutes and look through the disassembled code. An example of this is **strcmp**, which is where our password is evaluated. Please check the following screenshot.

```

0x0804851f 89442404 mov [esp+0x4], eax
0x08048523 c7042424a00. mov dword [esp], sym.pass.1685
0x0804852a e891feffff call 0x1080483c0
0x080483c0(unk) ; sym.imp.strcmp
0x0804852f 85c0 test eax, eax
0x08048531 7521 jnz 0x8048554

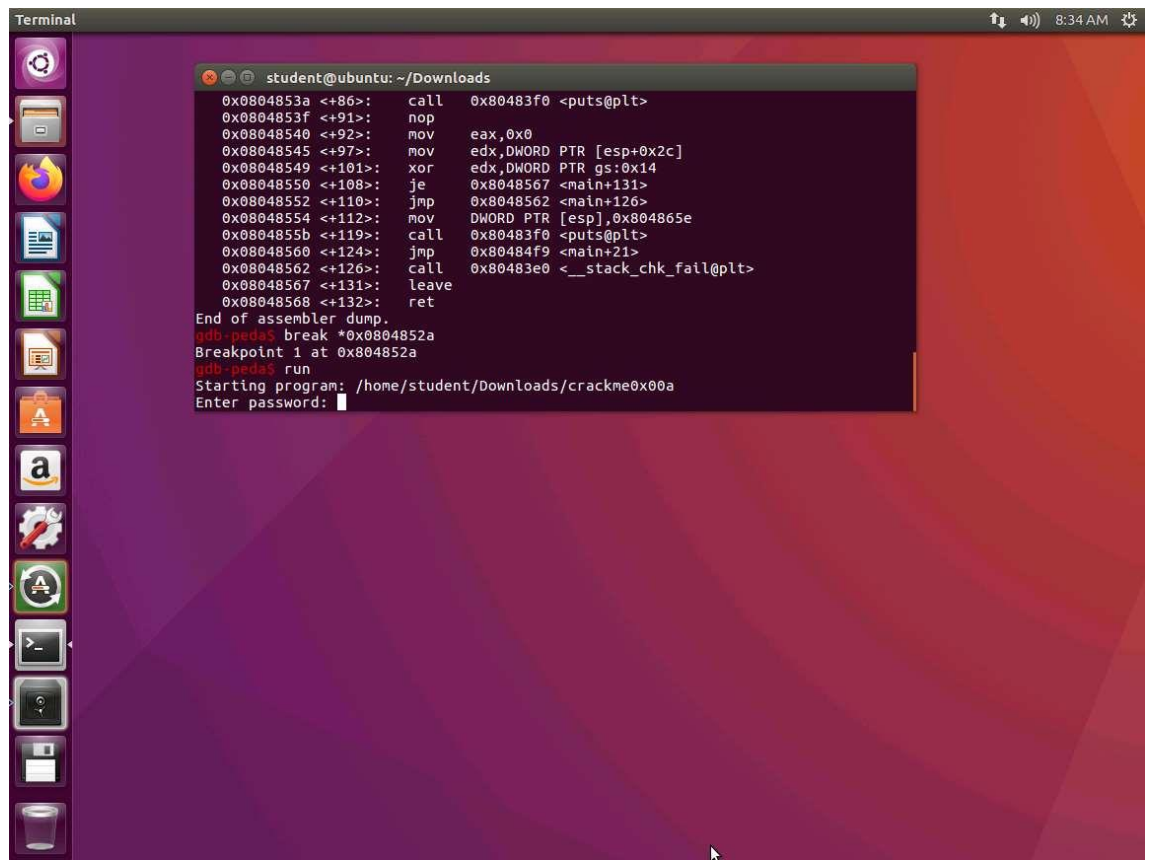
```

15. ☐ We want to look at the code with another tool, which we will now explore.
- In the terminal window, exit from Radare2 and enter **gdb crackme0x00a**. This will load the executable. Next, enter **disassemble main**. An example of the output of this command is shown in the following screenshot.



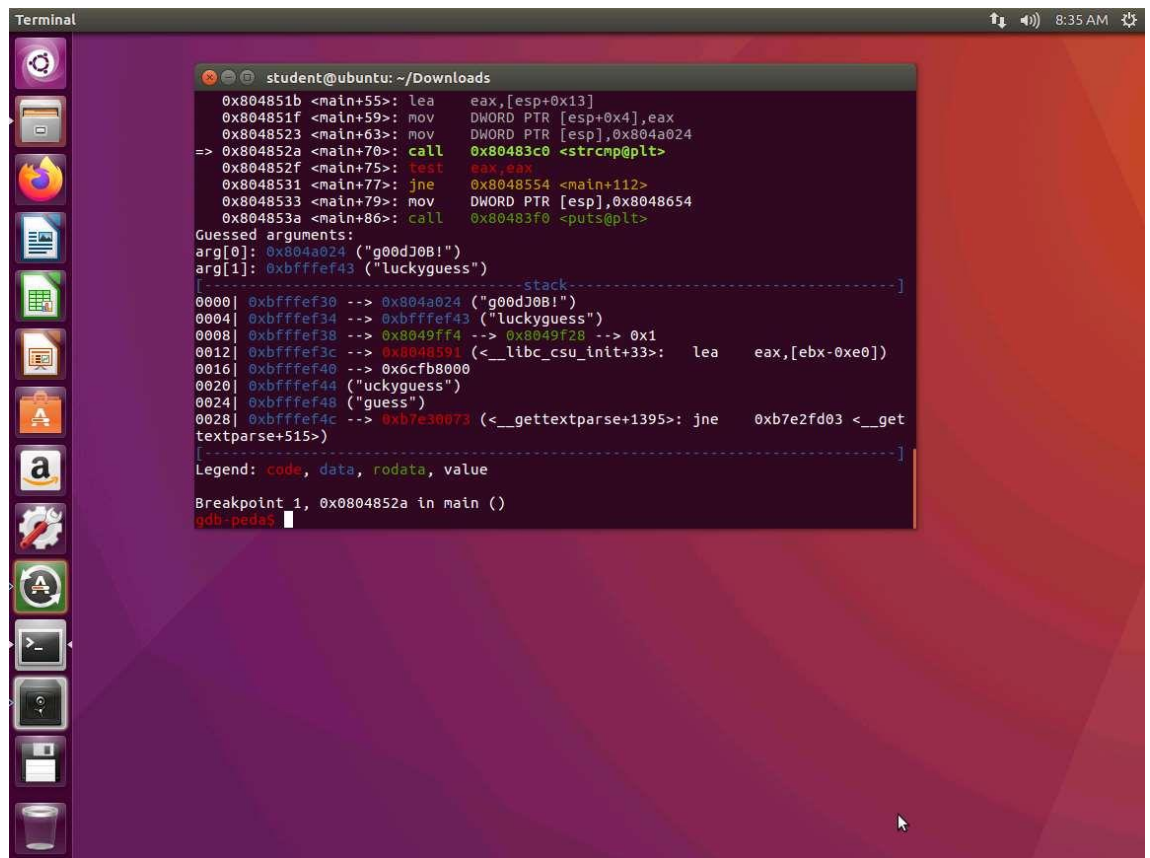
```
Terminal
student@ubuntu: ~/Downloads
student@ubuntu:~/Downloads$ gdb crackme0x00a
GNU gdb (Ubuntu 7.11.1-0ubuntu1-16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change 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 "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from crackme0x00a...(no debugging symbols found)...done.
gdb-peda$ disassemble main
Dump of assembler code for function main:
0x080484e4 <+0>: push    ebp
0x080484e5 <+1>: mov     ebp,esp
0x080484e7 <+3>: and     esp,0xfffffff0
0x080484ea <+6>: sub     esp,0x30
0x080484ed <+9>: mov     eax,gs:0x14
0x080484f3 <+15>: mov     DWORD PTR [esp+0x2c],eax
0x080484f7 <+19>: xor     eax,eax
0x080484f9 <+21>: mov     eax,0x8048640
0x080484fe <+26>: mov     DWORD PTR [esp],eax
0x08048501 <+29>: call    0x80483d0 <printf@plt>
0x08048506 <+34>: mov     eax,0x8048651
0x0804850b <+39>: lea     edx,[esp+0x13]
0x0804850f <+43>: mov     DWORD PTR [esp+0x4],edx
0x08048513 <+47>: mov     DWORD PTR [esp],eax
0x08048516 <+50>: call    0x8048420 <__isoc99_scanf@plt>
0x0804851b <+55>: lea     eax,[esp+0x13]
0x0804851f <+59>: mov     DWORD PTR [esp+0x4],eax
0x08048523 <+63>: mov     DWORD PTR [esp],0x804a024
0x0804852a <+70>: call    0x80483c0 <strcmp@plt>
0x0804852f <+75>: test    eax,eax
0x08048531 <+77>: jne     0x8048554 <main+112>
0x08048533 <+79>: mov     DWORD PTR [esp],0x8048654
0x0804853a <+86>: call    0x80483f0 <puts@plt>
0x0804853f <+91>: nop
0x08048540 <+92>: mov     eax,0x0
```

16. ☐ There is a strcmp instruction on <+70>.Therefore, let us set the breakpoint at the location and run the program using the following commands.
- a. break \*0x0804852a
  - b. run
17. ☐ The program will run until our breakpoint. An example of this is shown in the following screenshot.



```
Terminal
student@ubuntu: ~/Downloads
0x0804853a <+86>: call 0x080483f0 <puts@plt>
0x0804853f <+91>: nop
0x08048540 <+92>: mov eax,0x0
0x08048545 <+97>: mov edx,DWORD PTR [esp+0x2c]
0x08048549 <+101>: xor edx,DWORD PTR gs:0x14
0x08048550 <+108>: je 0x08048567 <main+131>
0x08048552 <+110>: jmp 0x08048562 <main+126>
0x08048554 <+112>: mov DWORD PTR [esp],0x0804865e
0x0804855b <+119>: call 0x080483f0 <puts@plt>
0x08048560 <+124>: jmp 0x080484f9 <main+21>
0x08048562 <+126>: call 0x080483e0 <__stack_chk_fail@plt>
0x08048567 <+131>: leave
0x08048568 <+132>: ret
End of assembler dump.
gdb-peda$ break *0x0804852a
Breakpoint 1 at 0x0804852a
gdb-peda$ run
Starting program: /home/student/Downloads/crackme0x00a
Enter password: 
```

18. ☐ Enter the password **luckyguess**. The comparison will reference the actual password as shown in the following screenshot.

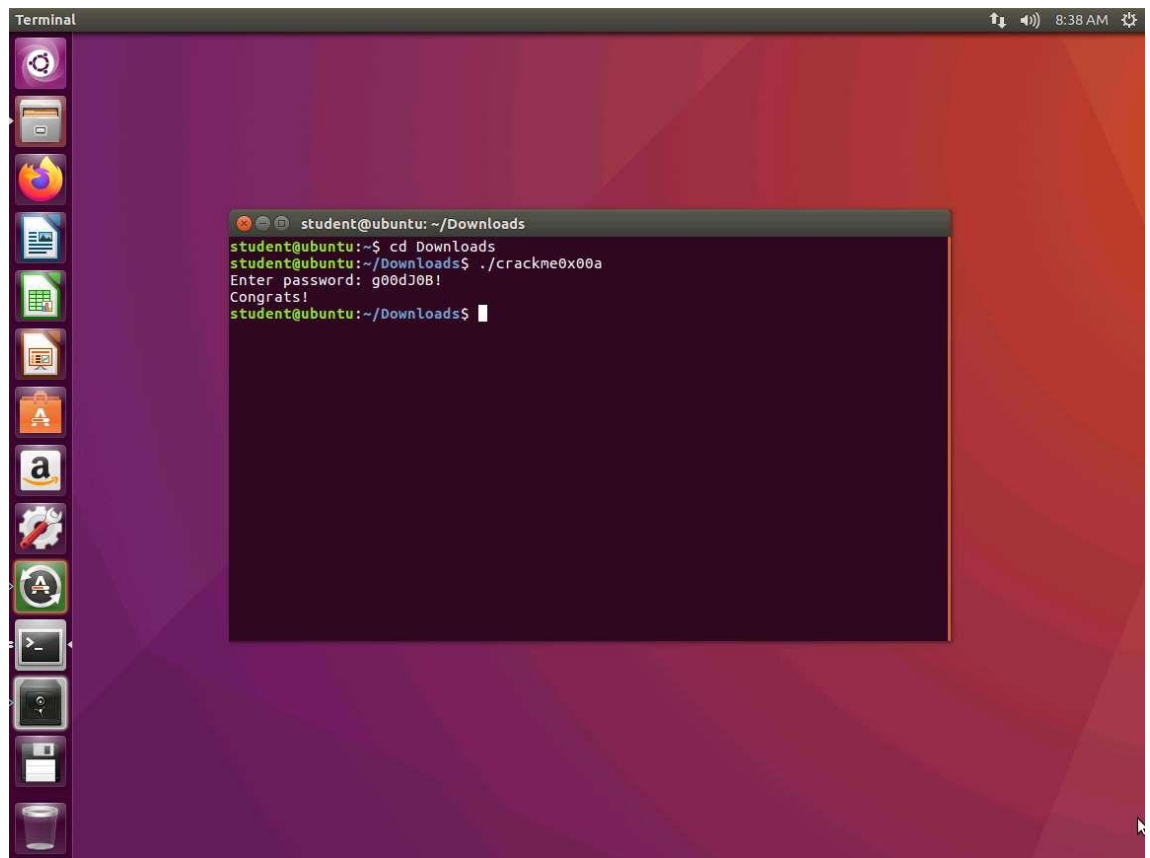


```
Terminal
student@ubuntu: ~/Downloads
0x0804851b <main+55>: lea    eax,[esp+0x13]
0x0804851f <main+59>: mov    DWORD PTR [esp+0x4],eax
0x08048523 <main+63>: mov    DWORD PTR [esp],0x0804a024
=> 0x0804852a <main+70>: call   0x080483c0 <strcmp@plt>
0x0804852f <main+75>: test   eax,eax
0x08048531 <main+77>: jne    0x08048554 <main+112>
0x08048533 <main+79>: mov    DWORD PTR [esp],0x08048654
0x0804853a <main+86>: call   0x080483f0 <puts@plt>
Guessed arguments:
arg[0]: 0x0804a024 ("g00dJ0B!")
arg[1]: 0xbffef43 ("luckyguess")
-----stack-----
0000| 0xbffef30 --> 0x0804a024 ("g00dJ0B!")
0004| 0xbffef34 --> 0xbffef43 ("luckyguess")
0008| 0xbffef38 --> 0x08049ff4 --> 0x08049f28 --> 0x1
0012| 0xbffef3c --> 0x08048591 (<__libc_csu_init+33>: lea    eax,[ebx-0xe0])
0016| 0xbffef40 --> 0x6cfb8000
0020| 0xbffef44 ("luckyguess")
0024| 0xbffef48 ("guess")
0028| 0xbffef4c --> 0xb7e30073 (<__gettextparse+1395>: jne    0xb7e2fd03 <__get
textparse+515>)
Legend: code, data, rodata, value
Breakpoint 1, 0x0804852a in main ()
gdb-peda$
```

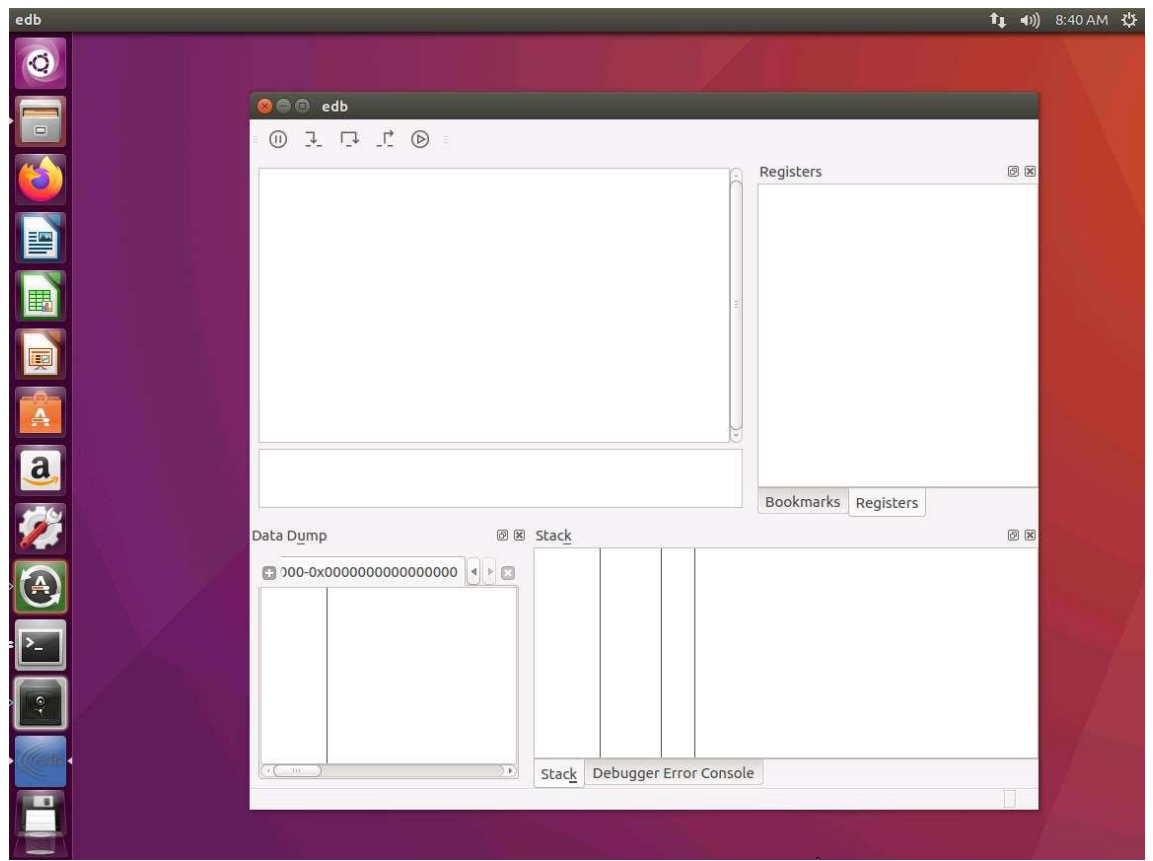
19. ☐ This is successful. To be certain, we need to test the discovered password.

Test the password to check whether it is correct, as shown in the following screenshot.

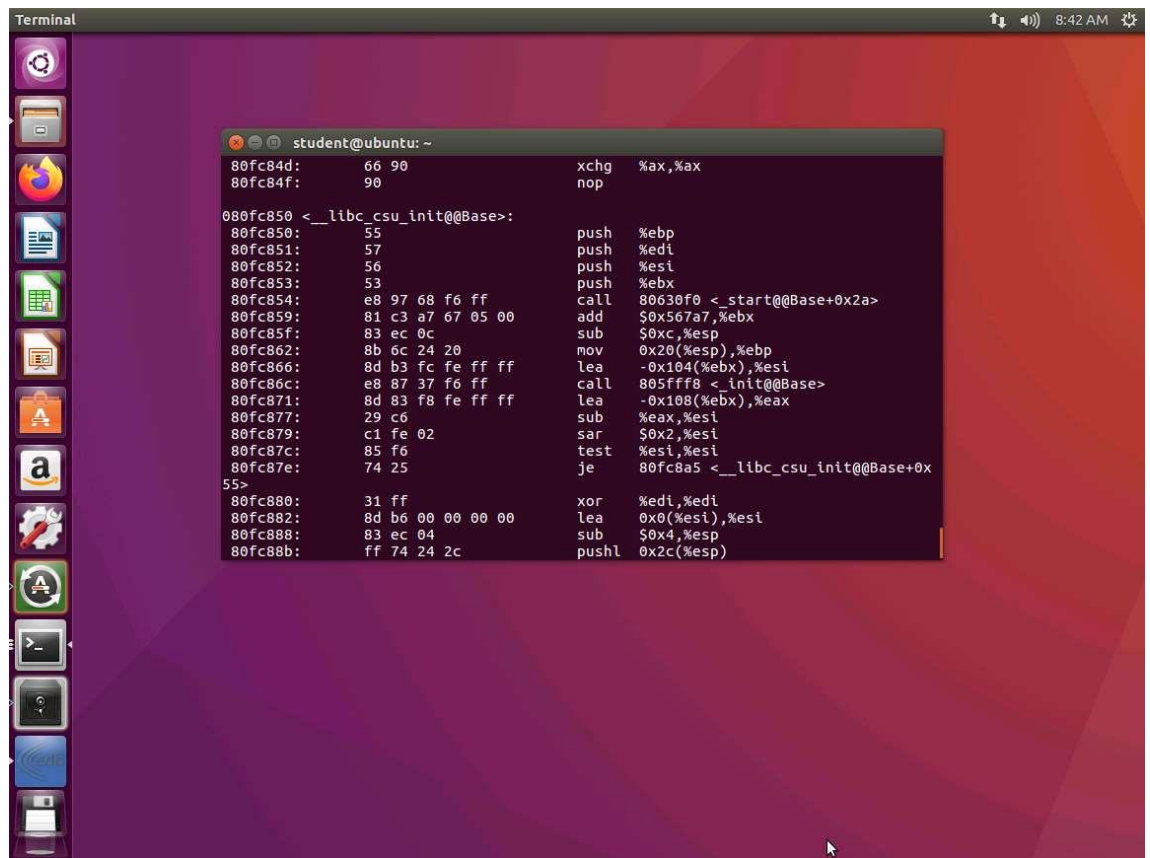




20. ☐ In computing, both hardware and software are reverse engineered.
- However, in this case, we will only refer to reverse engineering software, which usually offers a compiled program that is already in its binary format. The source is not available, but we want to know how it was made, how it works, and how to change it as well.
21. ☐ Now that we have performed the binary analysis of this file using these tools, let us move on and try some other techniques.
22. ☐ We want to look at another tool. Enter **edb**. An example of the output of this command is shown in the following screenshot.



23. ☐ This is the dashboard for Evan's Debugger. We have discussed **edb** very briefly. You are encouraged to read more here: <https://github.com/eteran/edb-debugger/wiki>.
24. ☐ Let us now explore the 32-bit code and its components. In the 32-bit VM, enter **objdump -d /bin/bash**. An example of the output of this command is shown in the following screenshot.



```
Terminal
student@ubuntu: ~
80fc84d: 66 90 xchg %ax,%ax
80fc84f: 90 nop
080fc850 <_libc_csu_init@@Base>:
80fc850: 55 push %ebp
80fc851: 57 push %edi
80fc852: 56 push %esi
80fc853: 53 push %ebx
80fc854: e8 97 68 f6 ff call 80630f0 <_start@@Base+0x2a>
80fc859: 81 c3 a7 67 05 00 add $0x567a7,%ebx
80fc85f: 83 ec 0c sub $0xc,%esp
80fc862: 8b 0c 24 20 mov 0x20(%esp),%ebp
80fc866: 8d b3 fc fe ff ff lea -0x104(%ebx),%esi
80fc86c: e8 87 37 f6 ff call 805fff8 <_init@@Base>
80fc871: 8d 83 f8 fe ff ff lea -0x108(%ebx),%eax
80fc877: 29 c6 sub %eax,%esi
80fc879: c1 fe 02 sar $0x2,%esi
80fc87c: 85 f6 test %esi,%esi
80fc87e: 74 25 je 80fc8a5 <_libc_csu_init@@Base+0x55>
80fc880: 31 ff xor %edi,%edi
80fc882: 8d b6 00 00 00 00 lea 0x0(%esi),%esi
80fc888: 83 ec 04 sub $0x4,%esp
80fc88b: ff 74 24 2c pushl 0x2c(%esp)
```

25. ☐ Next, let us look at intel notation. In the terminal window, enter **objdump** **-d -M intel /bin/bash**. An example of part of the output is shown in the following screenshot.



```
Terminal
student@ubuntu: ~
80fbf32: 88 50 02      mov     BYTE PTR [eax+0x2],dl
80fbf35: 89 fa         mov     edx,edi
80fbf37: 88 50 03      mov     BYTE PTR [eax+0x3],dl
80fbf3a: 89 d8         mov     eax,ebx
80fbf3c: eb 3e         jmp     80fbf7c <_malloc_block_signals@@B
ase+0x91c>
80fbf3e: 66 90         xchg    ax,ax
80fbf40: 8b 6c 24 04   mov     ebp,DWORD PTR [esp+0x4]
80fbf44: 8b 0c 24      mov     ecx,DWORD PTR [esp]
80fbf47: 39 f7         cmp     edi,esi
80fbf49: 89 f8         mov     eax,edi
80fbf4b: 0f 46 f7     cmovbe  esi,edi
80fbf4e: 89 ea         mov     edx,ebp
80fbf50: e8 3b f7 ff ff call    80fb690 <_malloc_block_signals@@B
ase+0x30>
80fbf55: 85 c0         test    eax,eax
80fbf57: 89 c7         mov     edi,eax
80fbf59: 0f 84 c1 00 00 je      80fc020 <_malloc_block_signals@@B
ase+0x9c0>
80fbf5f: 83 ec 04     sub     esp,0x4
80fbf62: 56           push    esi
80fbf63: 53           push    ebx
80fbf64: 50           push    eax
80fbf65: e8 a6 42 f6 ff call    8060210 <memcpy@plt>
```

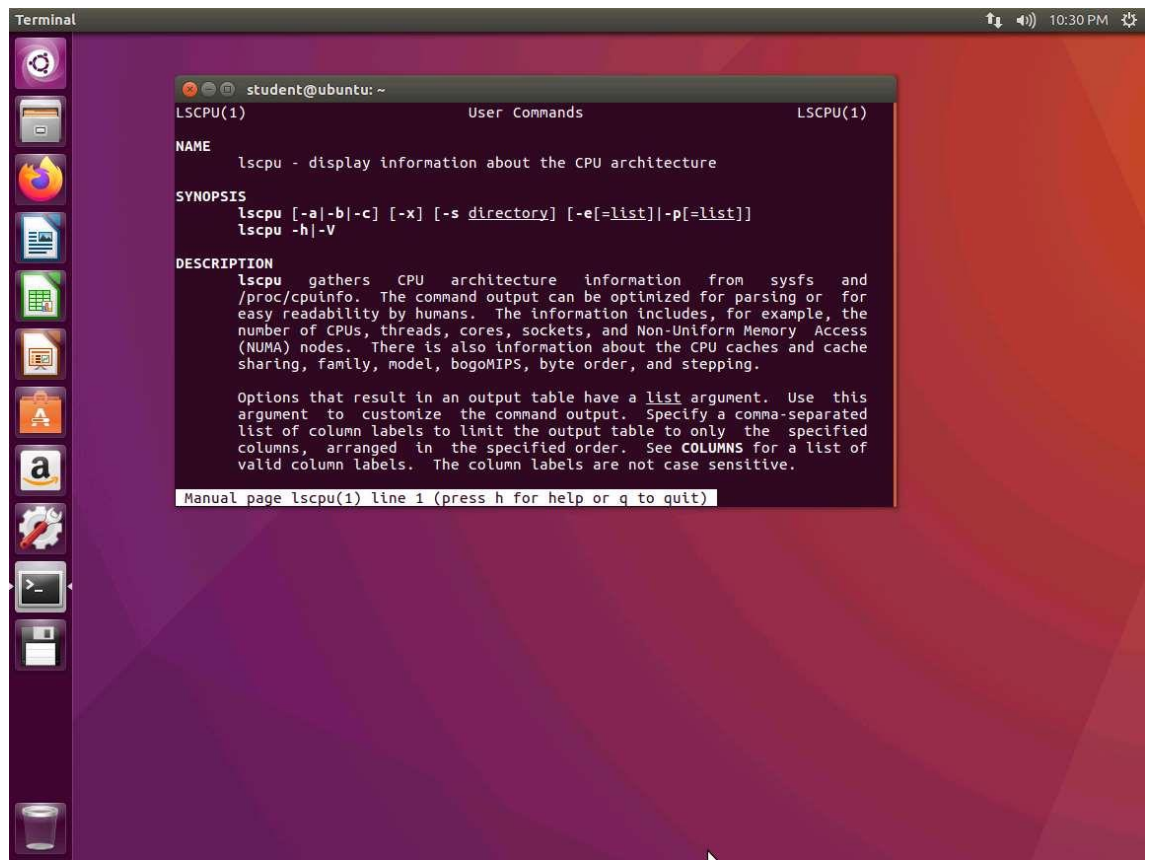
26. ☐ Compare the above two screenshots. One thing that is different is the lack of a % in the intel format.
27. ☐ We used the objdump tool with different arguments in order to highlight the difference between the AT&T syntax and the Intel syntax for, in this case, the 32-bit version of Bash. The first command we issued used the -d command-line argument of objdump to disassemble the Bash binary. The output in the first screenshot shows, from left to right, the address of the instruction, the opcodes for the instruction and operands, the instruction itself, the source operand, a comma, and finally, the destination operand. In short, AT&T syntax is formatted as follows:

AT&T Syntax: <instruction> <source operand>,<destination operand>

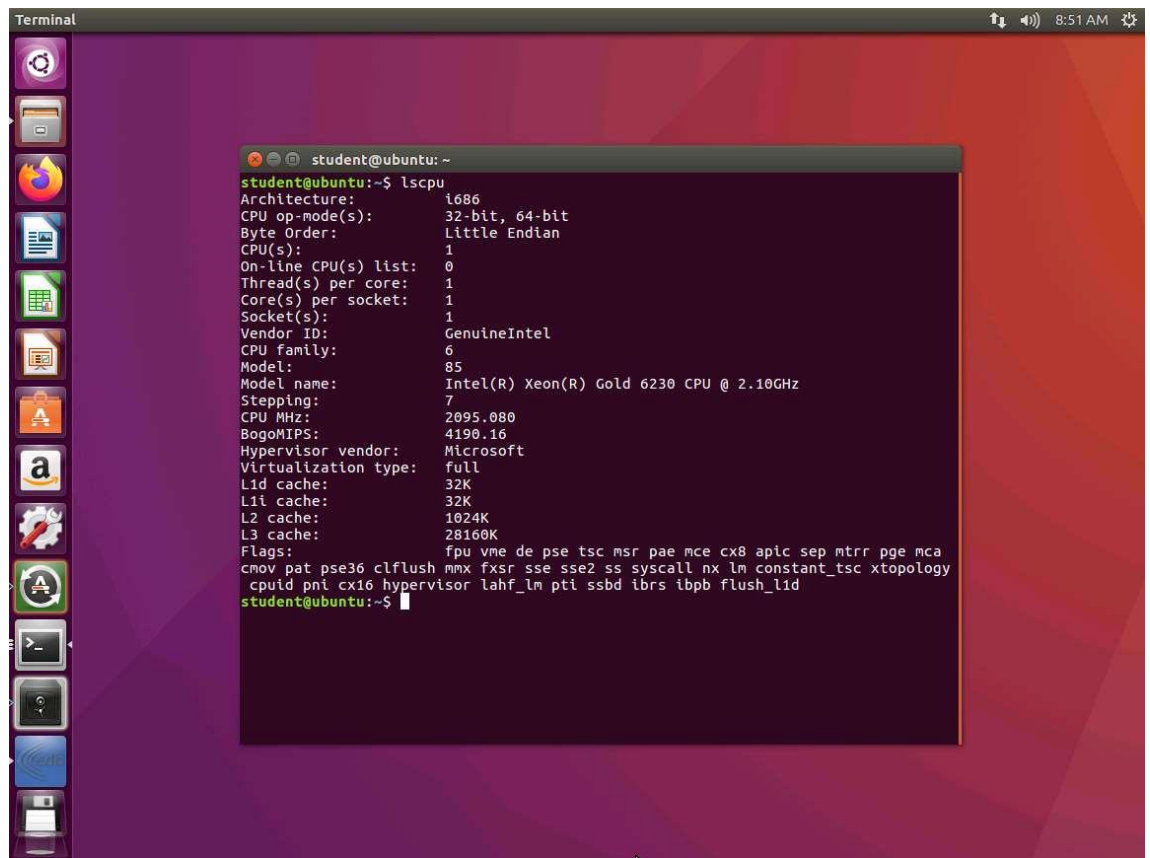
28. ☐ Then, we repeat the first command-line instruction but add the `-M intel` command-line argument, which tells the `objdump` tool to format the output using Intel syntax. The second screenshot is a truncated version of a much larger output and contains the same instructions as the first screenshot, except that it is formatted using the Intel syntax. Moving from left to right across the four columns, the first column shows the address in the memory of the instruction; the second column shows the opcodes for the instruction and operands; the third column shows the instruction itself; and the final column shows the destination operand, a comma, and the source operand. To summarize, the Intel syntax is formatted as follows:

**Intel Syntax:** `<instruction> <destination operand>, <source operand>`

29. ☐ Fortunately, **nasm** will automatically understand which syntax we are using.
30. ☐ You may encounter several different naming conventions for 32-bit and 64-bit Intel assembly. When reading `x86`, `x86-32`, `x86_32`, `IA32`, and `IA-32`, know that this refers to 32-bit Intel assembly. `x86-64`, `x86_64`, `IA64`, and `IA-64` refer to 64-bit Intel assembly. Intel, in this case, refers to the processor-specific instruction set, not necessarily the syntax format.
31. ☐ Let us now explore the different methods to extract information about the machine we are running. In the terminal window of the 32-bit virtual machine, enter **man lscpu**. Take a few minutes and review the information there.



32. ☐ In the terminal window, enter **lscpu**. An example of the output of this command is shown in the following screenshot.

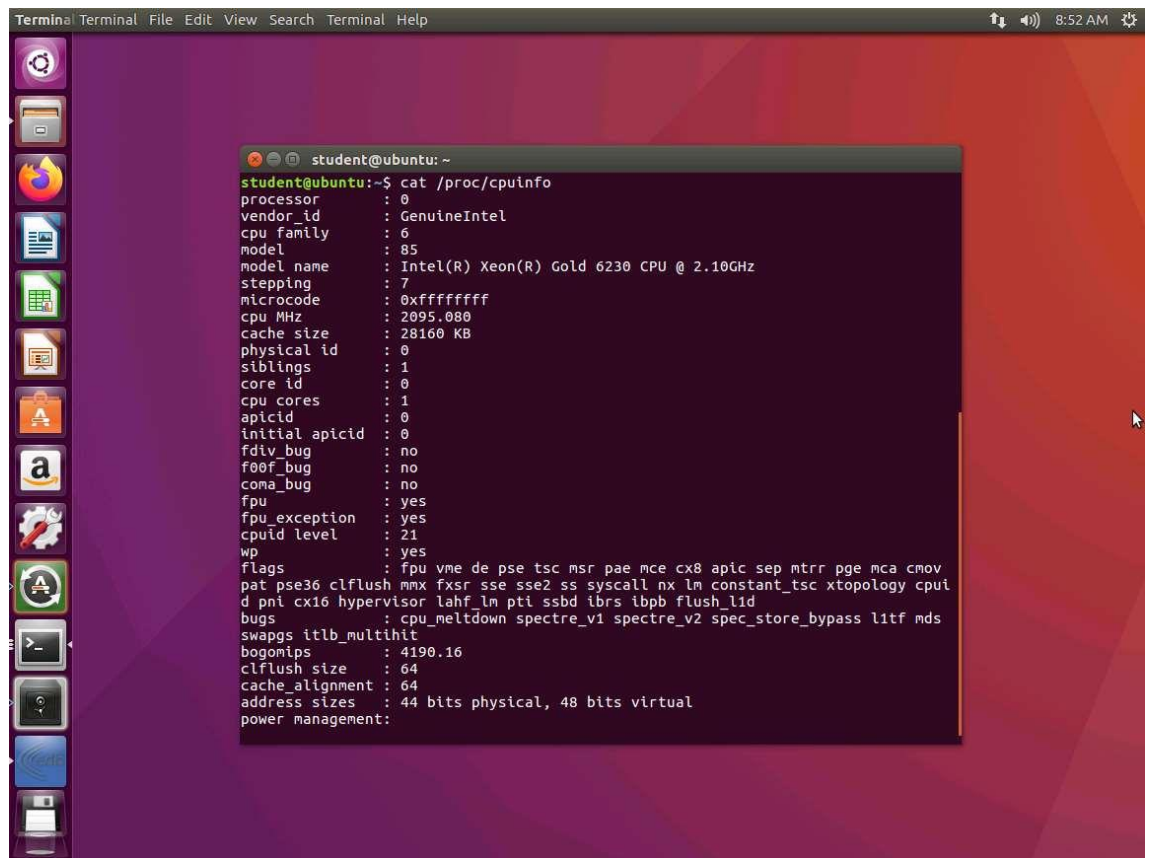


The screenshot shows a terminal window titled 'Terminal' with a dark purple background. The terminal displays the output of the 'lscpu' command. The output provides detailed information about the system's CPU, including architecture (i686), CPU op-mode(s) (32-bit, 64-bit), byte order (Little Endian), and various cache sizes (L1d, L1i, L2, L3). It also identifies the vendor as GenuineIntel, the model as Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz, and the hypervisor as Microsoft. The terminal window is part of a desktop environment with a sidebar of application icons on the left and a system status bar at the top right showing the time as 8:51 AM.

```
student@ubuntu:~$ lscpu
Architecture:          i686
CPU op-mode(s):        32-bit, 64-bit
Byte Order:             Little Endian
CPU(s):                 1
On-line CPU(s) list:   0
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):              1
Vendor ID:              GenuineIntel
CPU family:             6
Model:                  85
Model name:             Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz
Stepping:               7
CPU MHz:                2095.080
BogoMIPS:               4190.16
Hypervisor vendor:     Microsoft
Virtualization type:    full
L1d cache:              32K
L1i cache:              32K
L2 cache:               1024K
L3 cache:               28160K
Flags:                  fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
                        cmov pat pse36 clflush mmx fxsr sse sse2 ss syscall nx lm constant_tsc xtopology
                        cpuid pni cx16 hypervisor lahf_lm pti ssbd ibrs ibpb flush_l1d
student@ubuntu:~$
```

33. ☐ Now, let us look at **proc**. In the terminal window, enter **cat /proc/cpuinfo**.

The output of this command is shown in the following screenshot.



```
student@ubuntu:~$ cat /proc/cpuinfo
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 6
model          : 85
model name     : Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz
stepping       : 7
microcode      : 0xffffffff
cpu MHz        : 2095.080
cache size     : 28160 KB
physical id    : 0
siblings       : 1
core id        : 0
cpu cores      : 1
apicid         : 0
initial apicid : 0
fdiv_bug       : no
f00f_bug       : no
coma_bug       : no
fpu            : yes
fpu exception   : yes
cpuid level    : 21
wp             : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush mmx fxsr sse sse2 ss syscall nx lm constant_tsc xtopology cpui
d pni cx16 hypervisor lahf_lm pti ssbd ibrs ibpb flush_lid
bugs           : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds
swaps itlb_multihit
bogomips       : 4190.16
clflush size   : 64
cache alignment : 64
address sizes   : 44 bits physical, 48 bits virtual
power management:
```

34. ☐ Now, let us use **gdb** to look at the registers. In the terminal window, enter the following commands:

a. `gdb -q /bin/bash`

b. `break main`

c. `run`

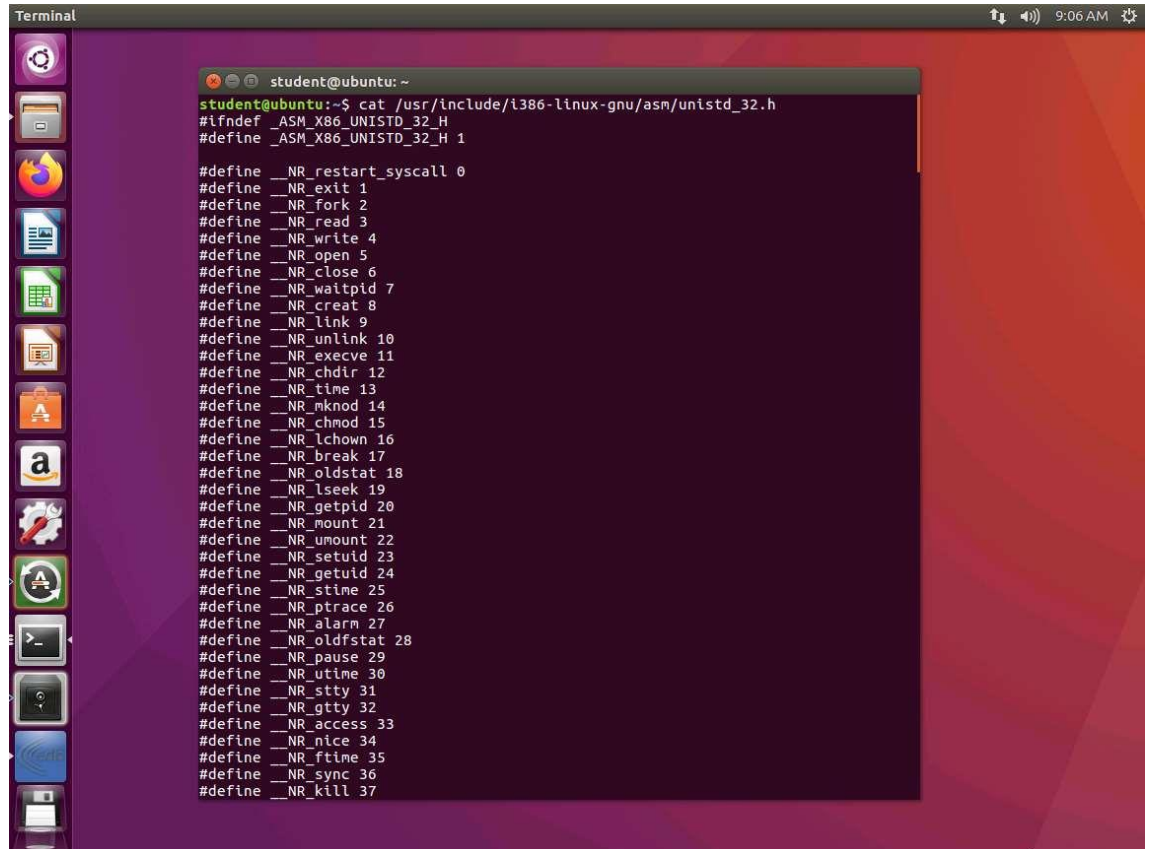
d. `info registers`

An example of the output is shown in the following screenshot.

```
student@ubuntu: ~  
gdb-peda$ info registers  
eax      0x81578c0      0x81578c0  
ecx      0xbffff070    0xbffff070  
edx      0xbffff094    0xbffff094  
ebx      0x0          0x0  
esp      0xbffff048    0xbffff048  
ebp      0xbffff058    0xbffff058  
esi      0xb7f90000    0xb7f90000  
edi      0xb7f90000    0xb7f90000  
eip      0x80618b1      0x80618b1 <main+17>  
eflags   0x286        [ PF SF IF ]  
cs       0x73        0x73  
ss       0x7b        0x7b  
ds       0x7b        0x7b  
es       0x7b        0x7b  
fs       0x0         0x0  
gs       0x33        0x33  
gdb-peda$
```

35. ☐ The important part of this output is the Endianness of our processor. Little-endian means that when we are reviewing or storing data in a register or on the stack, it must be formatted with the least significant byte first. Thus, 0x12345678 will actually look like 0x78563412. This is an extremely important concept to understand and a very important piece of information to know about our processor.
36. ☐ When we discuss assembly and processor architectures, it is important to understand Endianness. When the least significant bit appears in our output first, it is called a little-endian. When the least significant bit is last, we call that a big-endian. Throughout this course, we will use little-endian to display the least significant bit first when storing data in memory. This essentially means that when we deal with strings or immediate values, we need to reverse the order of the bytes. Endianness is one area that usually causes confusion, because we often forget to take it into account when analyzing binaries.
37. ☐ Enter **quit** to exit from gdb.

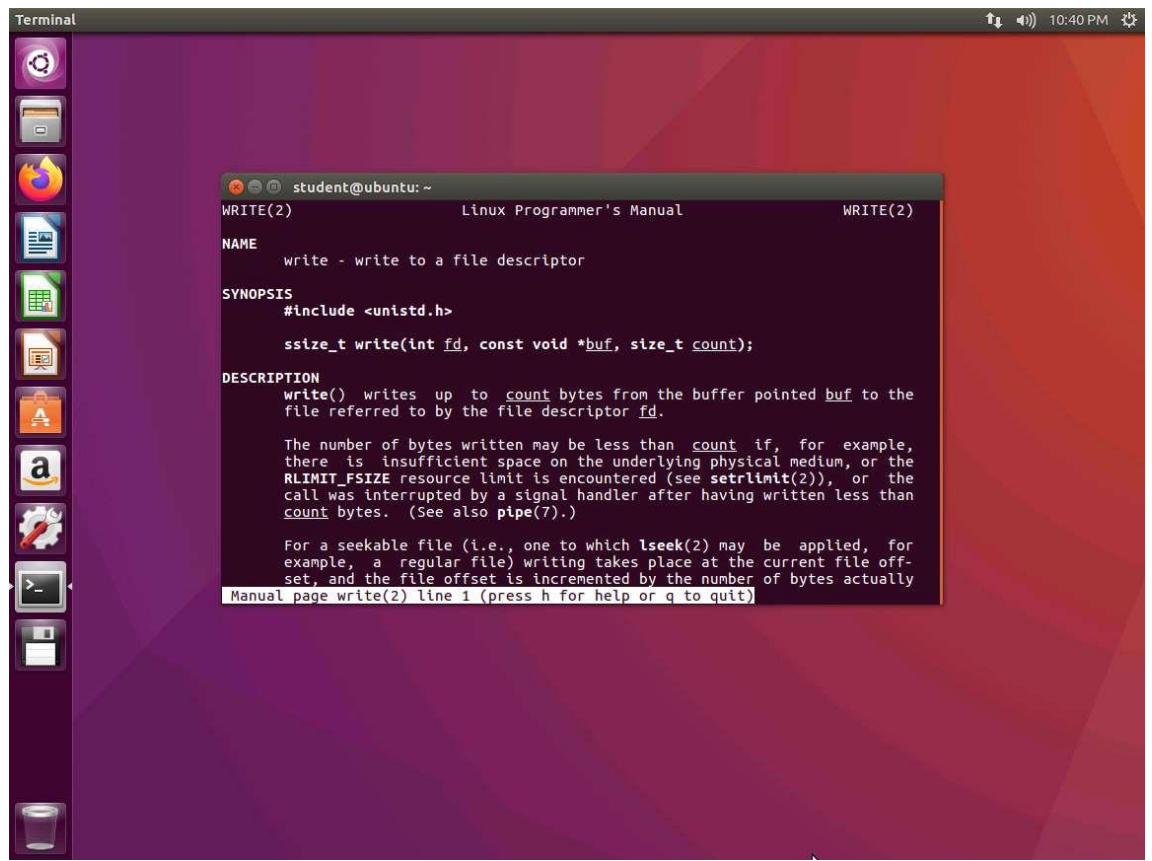
38. ☐ In the 32-bit machine, enter **cat /usr/include/i386-linux-gnu/asm/unistd\_32.h**. An example of the output of this command is shown in the following screenshot.



```
student@ubuntu: ~  
student@ubuntu:~$ cat /usr/include/i386-linux-gnu/asm/unistd_32.h  
#ifndef _ASM_X86_UNISTD_32_H  
#define _ASM_X86_UNISTD_32_H 1  
  
#define __NR_restart_syscall 0  
#define __NR_exit 1  
#define __NR_fork 2  
#define __NR_read 3  
#define __NR_write 4  
#define __NR_open 5  
#define __NR_close 6  
#define __NR_waitpid 7  
#define __NR_creat 8  
#define __NR_link 9  
#define __NR_unlink 10  
#define __NR_execve 11  
#define __NR_chdir 12  
#define __NR_time 13  
#define __NR_mknod 14  
#define __NR_chmod 15  
#define __NR_lchown 16  
#define __NR_break 17  
#define __NR_oldstat 18  
#define __NR_lseek 19  
#define __NR_getpid 20  
#define __NR_mount 21  
#define __NR_umount 22  
#define __NR_setuid 23  
#define __NR_getuid 24  
#define __NR_stime 25  
#define __NR_ptrace 26  
#define __NR_alarm 27  
#define __NR_oldfstat 28  
#define __NR_pause 29  
#define __NR_utime 30  
#define __NR_stty 31  
#define __NR_gtty 32  
#define __NR_access 33  
#define __NR_nice 34  
#define __NR_ftime 35  
#define __NR_sync 36  
#define __NR_kill 37
```

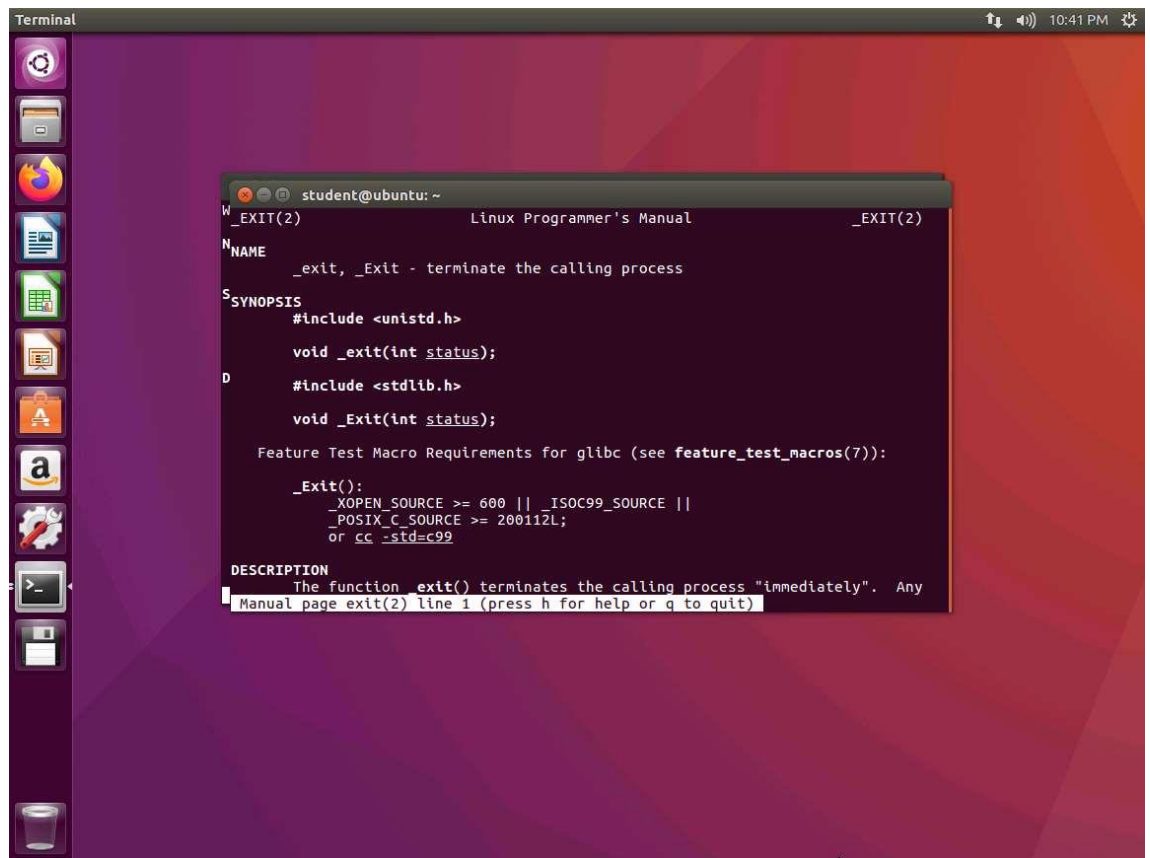
39. ☐ Open another terminal window using the shortcut SHIFT+CTRL+t.
40. ☐ In the terminal window, enter **man 2 write**. Take a few minutes and review the information in the man page.





41. ☐ Open another terminal window. Next, enter **man 2 exit**. Take a few minutes and review the information in the man page.





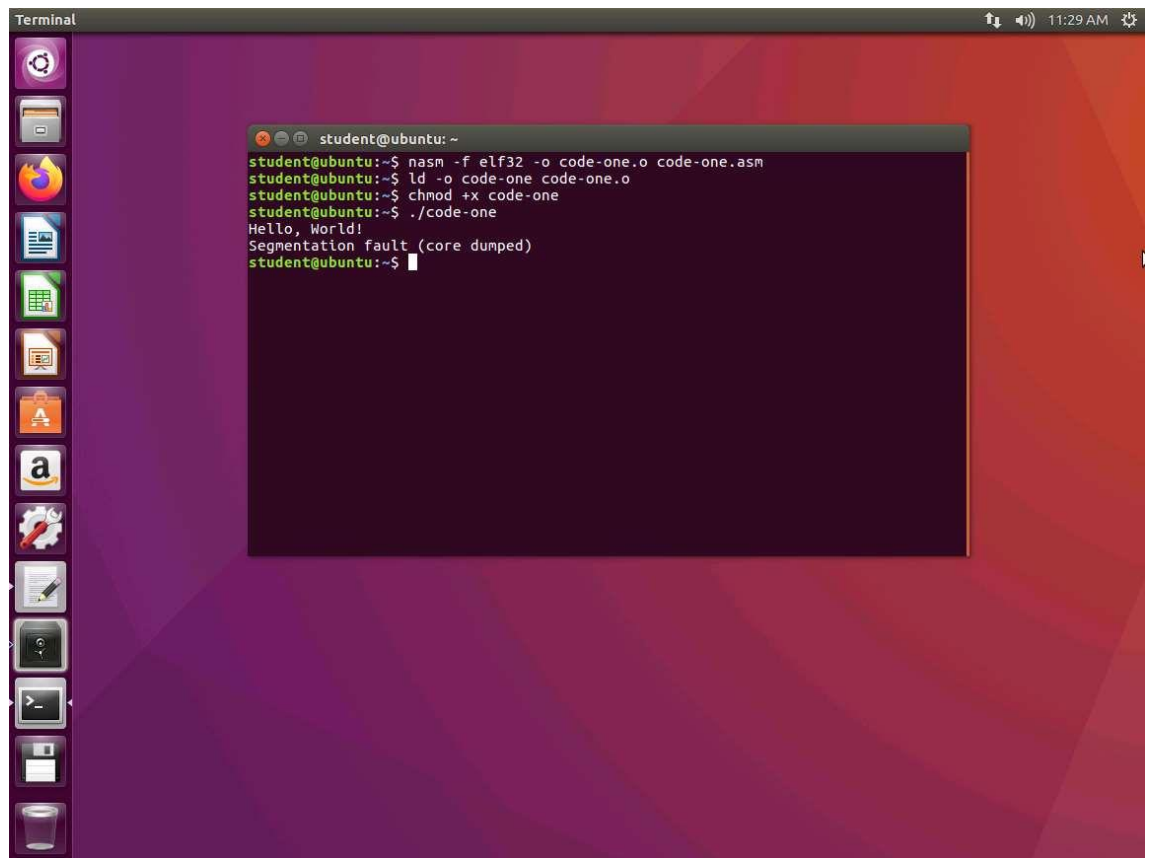
42. ☐ We are ready to create a small assembly program. Open a text editor of

your choice and enter the following:

```
43. global _start
44.
45. section .text
46.
47. _start:
48.     ; write(int fd, const void *buf, size_t count)
49.     xor    eax,eax
50.     xor    ebx,ebx
51.     xor    ecx,ecx
52.     xor    edx,edx
53.     mov    al,0x4
54.     inc    bl
55.     push   0x000a2164
56.     push   0x6c726f57
```

```
57.      push    0x202c6f6c
58.      push    0x6c6548
59.      mov     ecx,esp
60.      mov     dl,0xf
        int     0x80
```

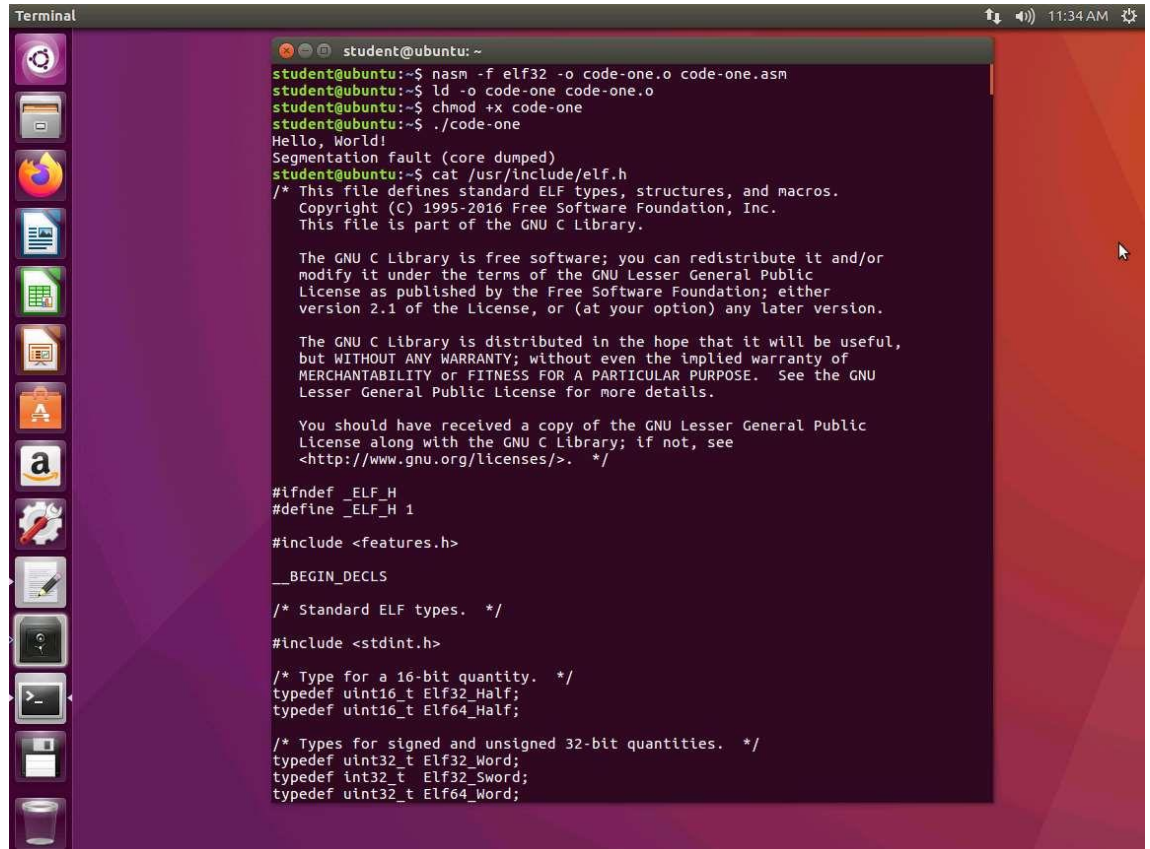
61. ☐ Save the files as **code-one.asm**, and ensure that the indentations are the same as in the example.
62. ☐ Next, enter the following commands:
- a. `nasm -f elf32 -o code-one.o code-one.asm`
  - b. `ld -o code-one code-one.o`
  - c. `chmod +x code-one`
  - d. `./code-one`
63. ☐ An example output of this command is shown in the following screenshot.



```
student@ubuntu: ~  
student@ubuntu:~$ nasm -f elf32 -o code-one.o code-one.asm  
student@ubuntu:~$ ld -o code-one code-one.o  
student@ubuntu:~$ chmod +x code-one  
student@ubuntu:~$ ./code-one  
Hello, World!  
Segmentation fault (core dumped)  
student@ubuntu:~$
```

- 64. ☐ As you see, it takes a lot of assembly to create a simple program, so this is one of the reasons why the C language is so popular.
- 65. ☐ When reviewing disassembled binaries, we may see terms such as byte, word, double word, quad word, and double quad word. These terms represent 8 bits, 16 bits, 32 bits, 64 bits, and 128 bits, respectively.
- 66. ☐ When studying a disassembled binary's output, it is also important to understand how the width of the data within an operand may impact the instruction syntax. For example, **PUSH** may become **PUSH WORD** when pushing a 32-bit wide piece of data onto the stack.
- 67. ☐ This program is based on the ELF-32 (executable and linking format). We will now extract information from the program so that we can understand it better.
- 68. ☐ As with anything, reading the man page is a good start. Enter **man elf**.

69. ☐ Take a few minutes and read the information contained within the man page. Once you have exited the man page, enter **cat /usr/include/elf.h**. The output of this command is shown in the following screenshot.



```
student@ubuntu:~$ nasm -f elf32 -o code-one.o code-one.asm
student@ubuntu:~$ ld -o code-one code-one.o
student@ubuntu:~$ chmod +x code-one
student@ubuntu:~$ ./code-one
Hello, World!
Segmentation fault (core dumped)
student@ubuntu:~$ cat /usr/include/elf.h
/* This file defines standard ELF types, structures, and macros.
   Copyright (C) 1995-2016 Free Software Foundation, Inc.
   This file is part of the GNU C Library.

   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   The GNU C Library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
   Lesser General Public License for more details.

   You should have received a copy of the GNU Lesser General Public
   License along with the GNU C Library; if not, see
   <http://www.gnu.org/licenses/>. */

#ifndef _ELF_H
#define _ELF_H 1

#include <features.h>

_BEGIN_DECLS

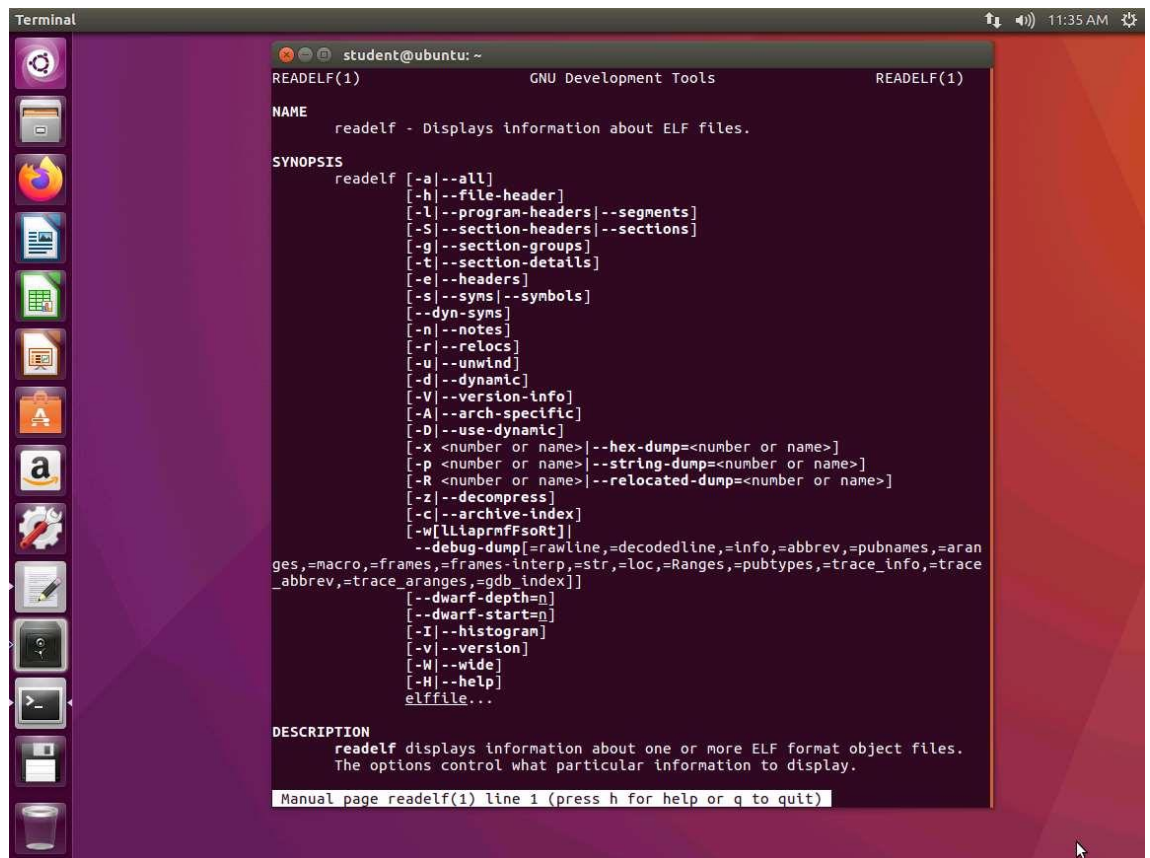
/* Standard ELF types. */

#include <stdint.h>

/* Type for a 16-bit quantity. */
typedef uint16_t Elf32_Half;
typedef uint16_t Elf64_Half;

/* Types for signed and unsigned 32-bit quantities. */
typedef uint32_t Elf32_Word;
typedef int32_t Elf32_Sword;
typedef uint32_t Elf64_Word;
```

70. ☐ Now we are ready to learn more about **ELF** files. Enter **man readelf**. An example of the output of this command is shown in the following screenshot.



```
Terminal
student@ubuntu: ~
readelf(1)                                GNU Development Tools                                readelf(1)

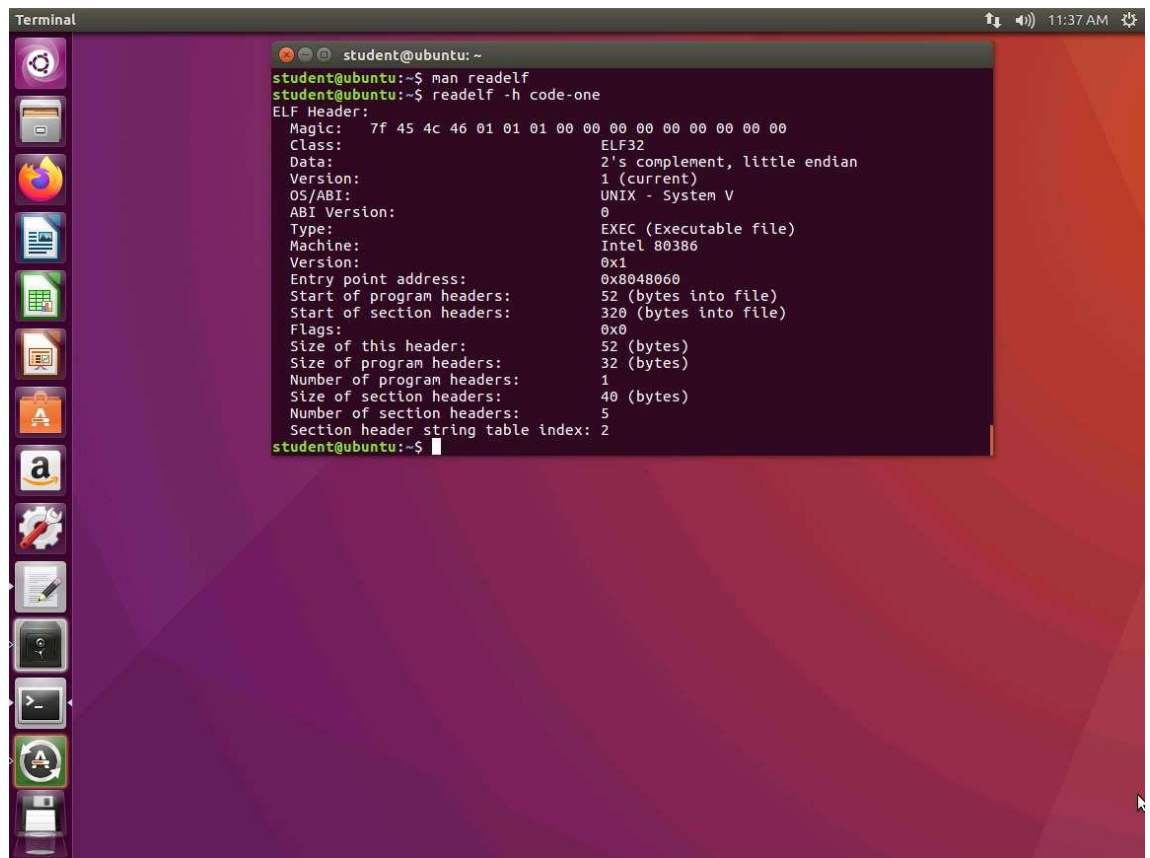
NAME
    readelf - Displays information about ELF files.

SYNOPSIS
    readelf [-a|--all]
            [-h|--file-header]
            [-l|--program-headers|--segments]
            [-S|--section-headers|--sections]
            [-g|--section-groups]
            [-t|--section-details]
            [-e|--headers]
            [-s|--syms|--symbols]
            [--dyn-syms]
            [-n|--notes]
            [-r|--relocs]
            [-u|--unwind]
            [-d|--dynamic]
            [-V|--version-info]
            [-A|--arch-specific]
            [-D|--use-dynamic]
            [-x <number or name>|--hex-dump=<number or name>]
            [-p <number or name>|--string-dump=<number or name>]
            [-R <number or name>|--relocated-dump=<number or name>]
            [-z|--decompress]
            [-c|--archive-index]
            [-w[LLIaprnffsOt]]
            --debug-dump[=rawline,=decodedline,=info,=abbrev,=pubnames,=aran
ges,=macro,=frames,=frames-interp,=str,=loc,=Ranges,=pubtypes,=trace_info,=trace
_abbrev,=trace_aranges,=gdb_index]
            [--dwarf-depth=n]
            [--dwarf-start=n]
            [-I|--histogram]
            [-V|--version]
            [-W|--wide]
            [-H|--help]
            elffile...

DESCRIPTION
    readelf displays information about one or more ELF format object files.
    The options control what particular information to display.

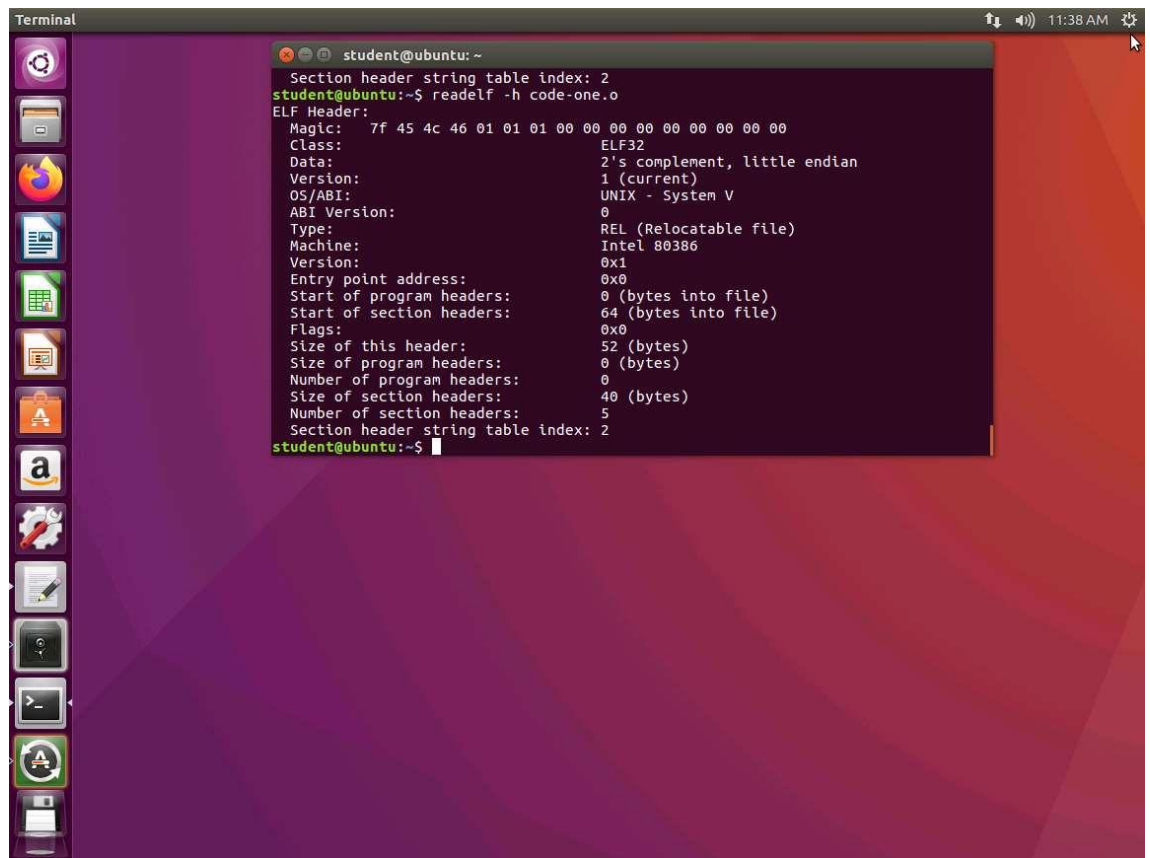
Manual page readelf(1) line 1 (press h for help or q to quit)
```

71. ☐ Next, let us review our code with this tool. Ensure that you are in the folder where you created your program, and enter **readelf -h code-one**. The output of this command, including the start with the ELF Header, is shown in the following screenshot.



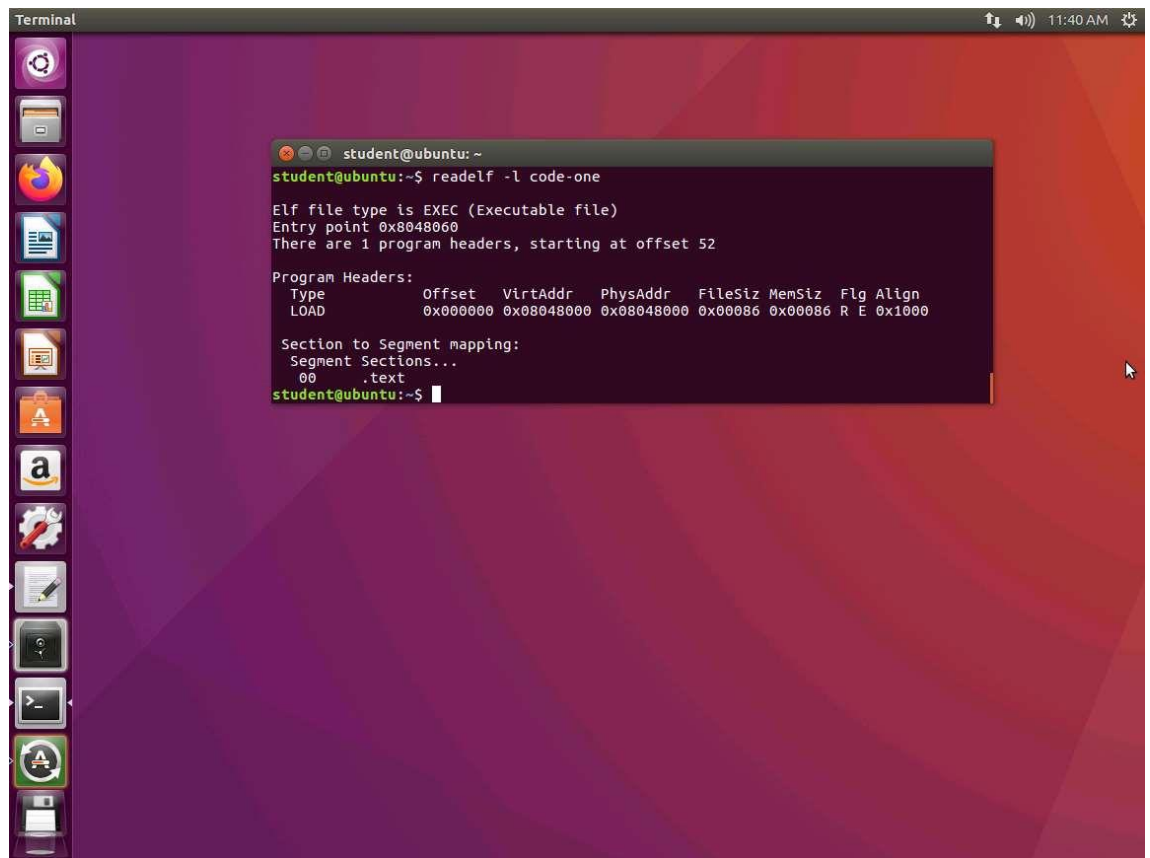
```
student@ubuntu: ~  
student@ubuntu:~$ man readelf  
student@ubuntu:~$ readelf -h code-one  
ELF Header:  
Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00  
Class:                                ELF32  
Data:                                  2's complement, little endian  
Version:                              1 (current)  
OS/ABI:                               UNIX - System V  
ABI Version:                           0  
Type:                                  EXEC (Executable file)  
Machine:                              Intel 80386  
Version:                              0x1  
Entry point address:                   0x8048060  
Start of program headers:              52 (bytes into file)  
Start of section headers:             320 (bytes into file)  
Flags:                                 0x0  
Size of this header:                   52 (bytes)  
Size of program headers:               32 (bytes)  
Number of program headers:              1  
Size of section headers:               40 (bytes)  
Number of section headers:              5  
Section header string table index: 2  
student@ubuntu:~$
```

72. ☐ The Magic is the **7f**, start of the ELF header and 45 (E), 4C(L) and 46(F). So, it starts with ELF. The next number **01** means we have 32 bytes. If we had 64, it would be a 02. Then, the next 01 is for little- endian and a value of 02 would be for big-endian.
73. ☐ Next, we want to examine the object file. Enter **readelf -h code-one.o**. An example of the output of this command is shown in the following screenshot.



```
Terminal
student@ubuntu: ~
Section header string table index: 2
student@ubuntu:~$ readelf -h code-one.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                                ELF32
  Data:                                  2's complement, little endian
  Version:                              1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                  REL (Relocatable file)
  Machine:                               Intel 80386
  Version:                               0x1
  Entry point address:                   0x0
  Start of program headers:              0 (bytes into file)
  Start of section headers:              64 (bytes into file)
  Flags:                                  0x0
  Size of this header:                   52 (bytes)
  Size of program headers:               0 (bytes)
  Number of program headers:              0
  Size of section headers:               40 (bytes)
  Number of section headers:              5
  Section header string table index: 2
student@ubuntu:~$
```

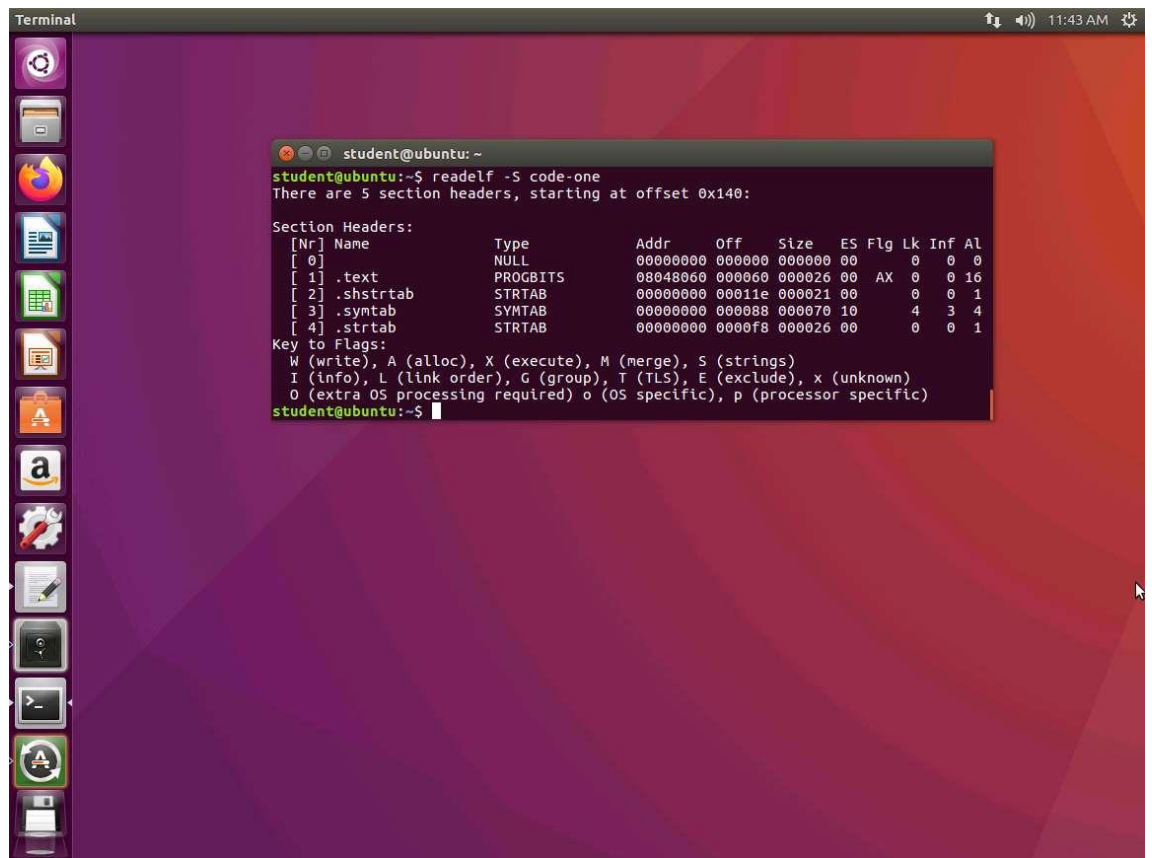
74. ☐ As we can see in the screenshot, Type is **REL**, so this is a relocatable file. As such, there are no program headers. The executable image program headers start 52 bytes in; there are none, so the start is 0.
75. ☐ Next, we will look at the listing. Enter **readelf -l code-one** (that is a n “el”). An example of the output of this command is shown in the following screenshot.



```
student@ubuntu: ~  
student@ubuntu:~$ readelf -l code-one  
Elf file type is EXEC (Executable file)  
Entry point 0x8048060  
There are 1 program headers, starting at offset 52  
  
Program Headers:  
Type           Offset       VirtAddr     PhysAddr     FileSiz MemSiz  Flg Align  
LOAD           0x000000    0x08048000   0x08048000   0x00086 0x00086  R E 0x1000  
  
Section to Segment mapping:  
Segment Sections...  
00      .text  
student@ubuntu:~$
```

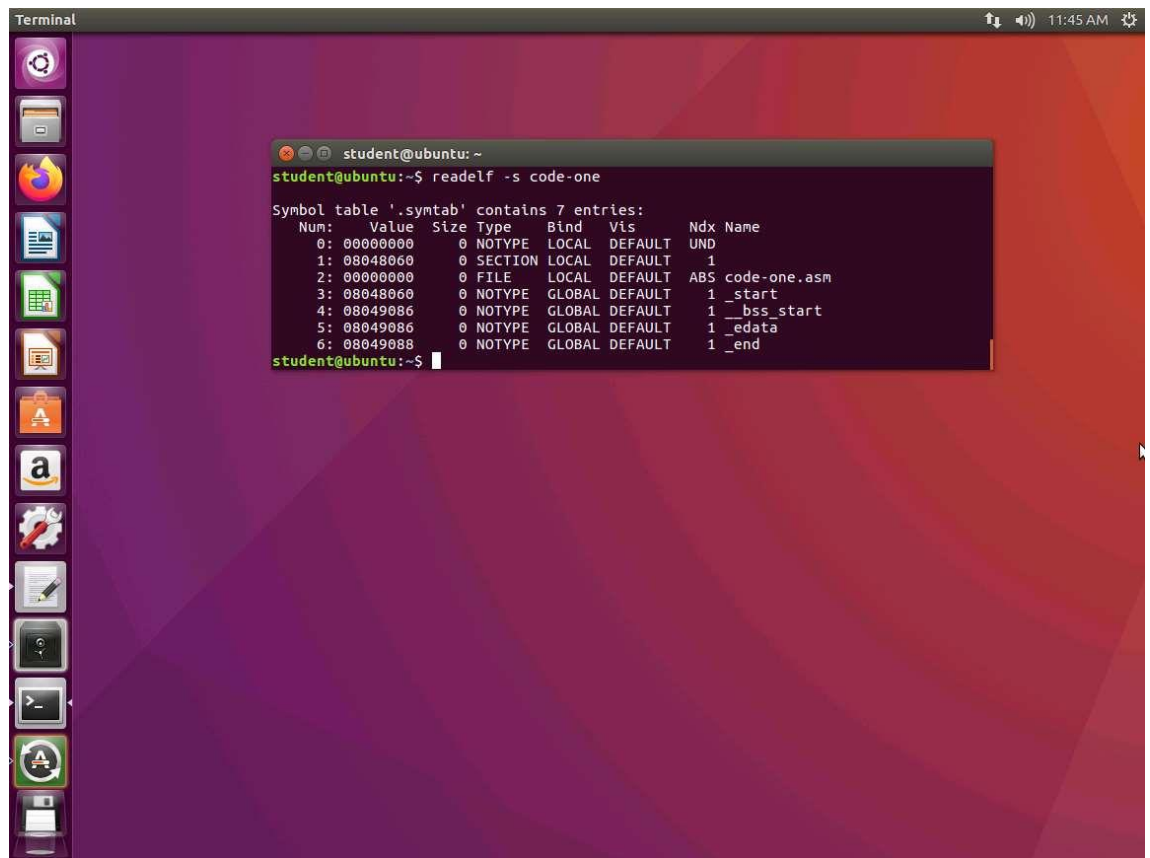
76. ☐ The program header is visible in the screenshot. It starts at virtual address 0x08048000, physical address 0x08048000, has a file size of 0x00086 (142) bytes, and takes up the same amount of memory. It is set with the R and E flags, indicating that segment is set with the permissions read/execute and requires a memory alignment of 0x1000 (4096) bytes. We can also see which sections are mapped to the segment, which is indicated in the program header table. This is the executable **.text** section.
77. ☐ Next, enter **readelf -S code-one**. An example of the output of this command is shown in the following screenshot.





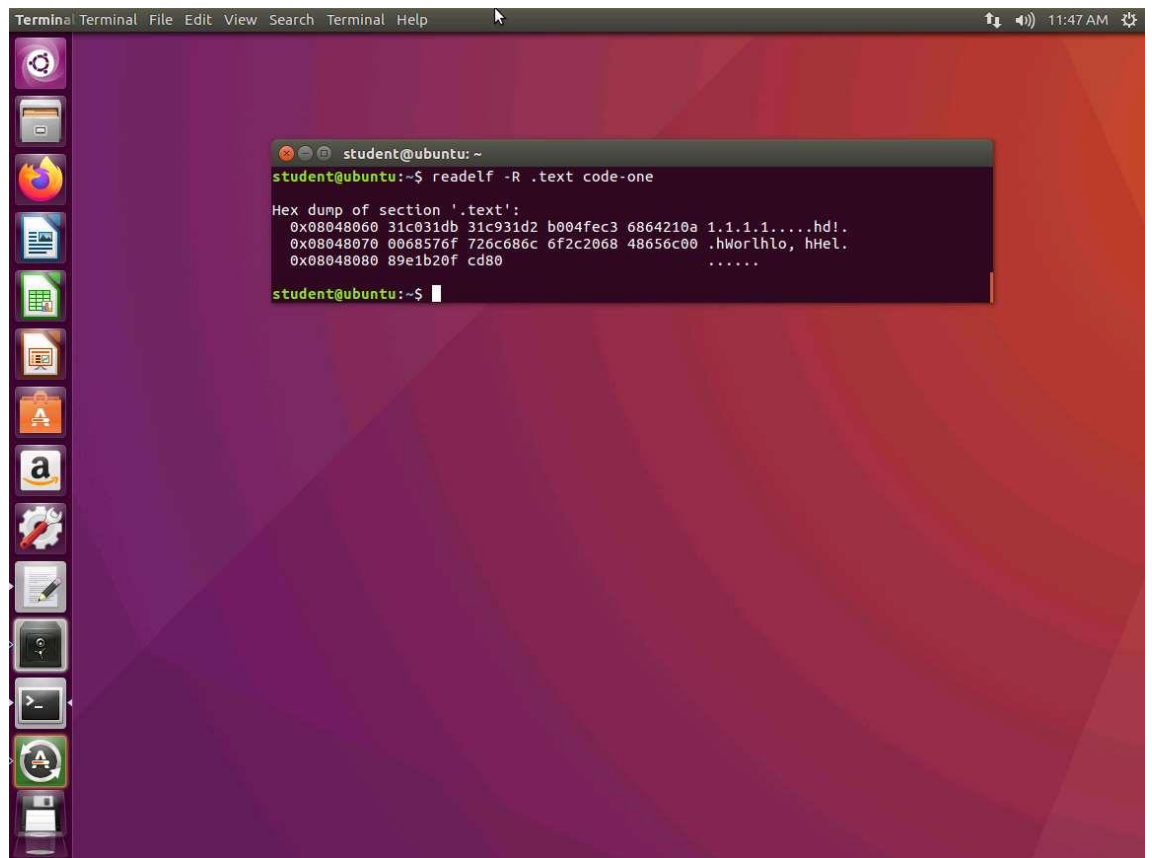
```
student@ubuntu: ~  
student@ubuntu:~$ readelf -S code-one  
There are 5 section headers, starting at offset 0x140:  
  
Section Headers:  
[Nr] Name                Type              Addr      Off      Size    ES Flg Lk Inf Al  
[ 0]                     NULL              00000000  000000  000000  00   0  0  0  
[ 1] .text                  PROGBITS          08048060  000060  000026  00  AX  0  0 16  
[ 2] .shstrtab              STRTAB            00000000  00011e  000021  00   0  0  1  
[ 3] .symtab                SYMTAB            00000000  000088  000070  10   4  3  4  
[ 4] .strtab                STRTAB            00000000  0000f8  000026  00   0  0  1  
  
Key to Flags:  
W (write), A (alloc), X (execute), M (merge), S (strings)  
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)  
O (extra OS processing required) o (OS specific), p (processor specific)  
student@ubuntu:~$
```

78. ☐ This screenshot shows the section headers. We can see that for the **.text** section, the type is indicated as **PROGBITS** and is marked as executable (**X**). The **PROGBITS** type indicates that this section contains program data. This is because we coded the program in the **.text** section.
79. ☐ Next, enter **readelf -s code-one**. An example of the output of this command is shown in the following screenshot.



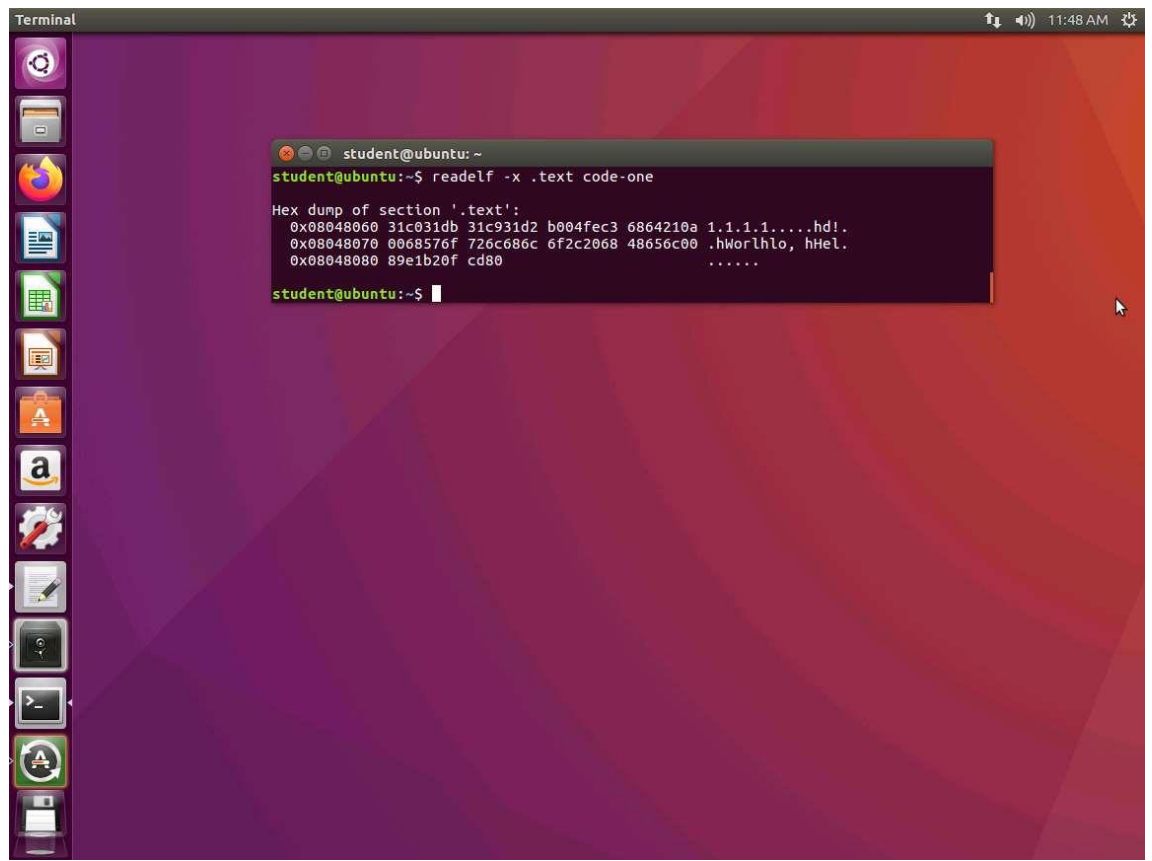
```
student@ubuntu: ~  
student@ubuntu:~$ readelf -s code-one  
Symbol table '.symtab' contains 7 entries:  
Num:  Value      Size Type   Bind   Vis      Ndx Name  
0: 00000000      0 NOTYPE LOCAL DEFAULT UND  
1: 08048060      0 SECTION LOCAL DEFAULT 1  
2: 00000000      0 FILE   LOCAL DEFAULT ABS code-one.asm  
3: 08048060      0 NOTYPE GLOBAL DEFAULT 1 _start  
4: 08049086      0 NOTYPE GLOBAL DEFAULT 1 __bss_start  
5: 08049086      0 NOTYPE GLOBAL DEFAULT 1 _edata  
6: 08049088      0 NOTYPE GLOBAL DEFAULT 1 _end
```

80. ☐ This screenshot shows the symbol table. We have the string table index, the memory location of the symbol itself, the size of the symbol is in bytes, the type of symbol, the symbol's binding, whether it is visible or not, the section index, and the symbol name. Notice that we recognize at least two entries in our output: the `_start` symbol, marked GLOBAL, and the name of our file.
81. ☐ Next, let us look at the `.text` info. Enter **`readelf -R .text code-one`**. An example of the output of this command is shown in the following screenshot.



```
student@ubuntu: ~  
student@ubuntu:~$ readelf -R .text code-one  
Hex dump of section '.text':  
0x08048060 31c031db 31c931d2 b004fec3 6864210a 1.1.1.1.....hd!  
0x08048070 0068576f 726c686c 6f2c2068 48656c00 .hWorlhllo, hHel.  
0x08048080 89e1b20f cd80 .....  
student@ubuntu:~$
```

82. ☐ This screenshot shows the relocated bytes.
83. ☐ Next, enter **readelf -x .text code-one**. The output is the same as the previous command, but this option dumps the hexadecimal.



The screenshot shows an Ubuntu desktop with a purple and red geometric background. On the left is a vertical dock with icons for Dash, Home, Firefox, LibreOffice Writer, LibreOffice Calc, LibreOffice Impress, Amazon, and several system utilities. A terminal window is open in the center, displaying the command `readelf -x .text code-one` and its output, which is a hex dump of the `.text` section of the `code-one` binary. The hex dump shows three lines of memory addresses, hex values, and their corresponding ASCII representations: `1.1.1.1.....hd!.`, `.hWorlhllo, hHel.`, and `.....`.

```
student@ubuntu: ~  
student@ubuntu:~$ readelf -x .text code-one  
Hex dump of section '.text':  
0x08048060 31c031db 31c931d2 b004fec3 6864210a 1.1.1.1.....hd!.  
0x08048070 0068576f 726c686c 6f2c2068 48656c00 .hWorlhllo, hHel.  
0x08048080 89e1b20f cd80 .....  
student@ubuntu:~$
```

84. ☐ The lab objectives have been achieved.