Write-up

Shellmates Mini-CTF 2018 – BypassME

Amina MEHERHERA
SHELLMATES MEMBER
fa_meherhera@esi.dz

Thanks

Special thanks to KIMOUCHE Mohamed and BOUTHIBA Abderraouf who organized this Mini-CTF and for all the help they gave us and what we learnt from them.

Thanks to ZOUAHI Hafidh and BALI Amina who helped me write this write-up (my very first one).

And of course, thanks to all Shellmates members (ntouma haylin[1] ☺).

[1] BALI Amina ☺

Challenge description

Title: BypassME

Category: Reverse Engineering

Description: none

Points: ?

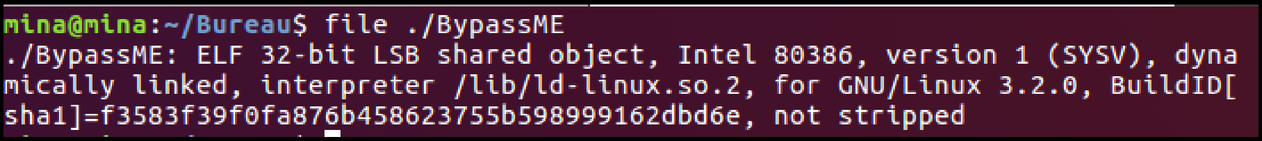
Flag Format: Shellmates{…}

Difficulty: Medium

Author: Raouf or Mohamed ?

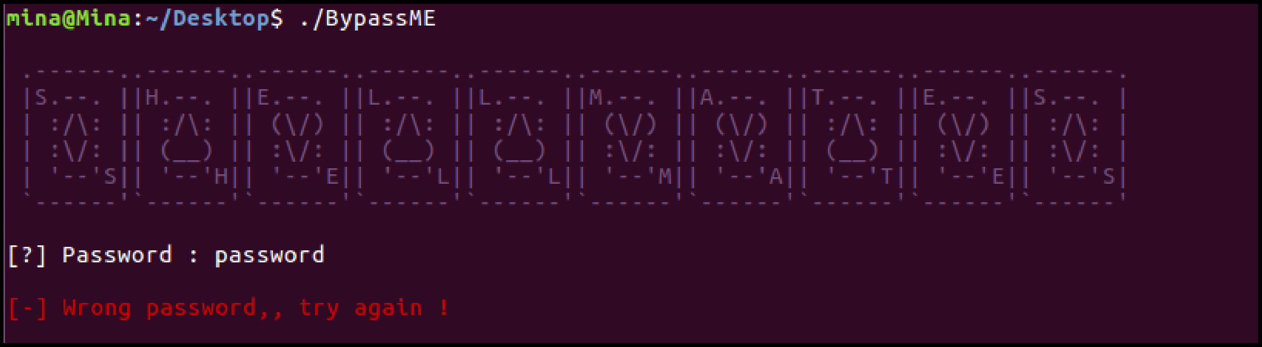
Analysis

We are given an ELF 32-bit non-stripped[2] executable file BypassME as the command file BypassME shows:



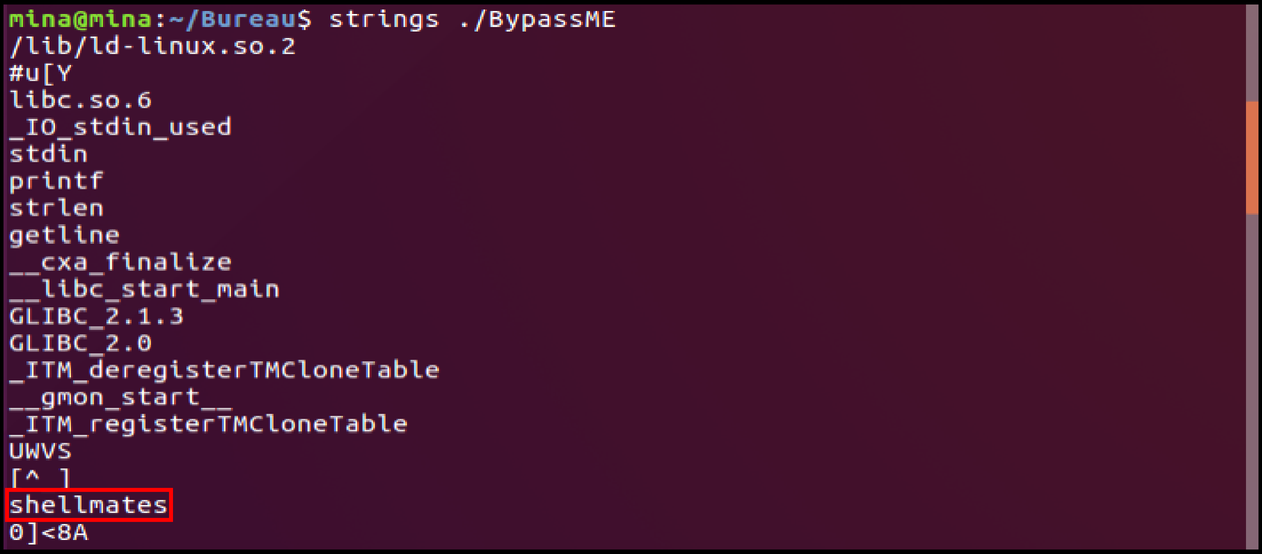
```
mina@mina:~/Bureau$ file ./BypassME
./BypassME: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dyna
mically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.2.0, BuildID[
sha1]=f3583f39f0fa876b458623755b598999162dbd6e, not stripped
```

When we run it, it asks for a password and if we give a random one it displays “Wrong password,, try again”



```
mina@Mina:~/Desktop$ ./BypassME

 .------..------..------..------..------..------..------..------..------..------.
 |S.--. ||H.--. ||E.--. ||L.--. ||L.--. ||M.--. ||A.--. ||T.--. ||E.--. ||S.--. |
 | :/\: || :/\: || (\/) || :/\: || :/\: || (\/) || (\/) || :/\: || (\/) || :/\: |
 | :\/: || (__) || :\/: || (__) || (__) || :\/: || :\/: || (__) || :\/: || :\/: |
 | '--'S|| '--'H|| '--'E|| '--'L|| '--'L|| '--'M|| '--'A|| '--'T|| '--'E|| '--'S|
 `------'`------'`------'`------'`------'`------'`------'`------'`------'`------'

[?] Password : password

[-] Wrong password,, try again !
```

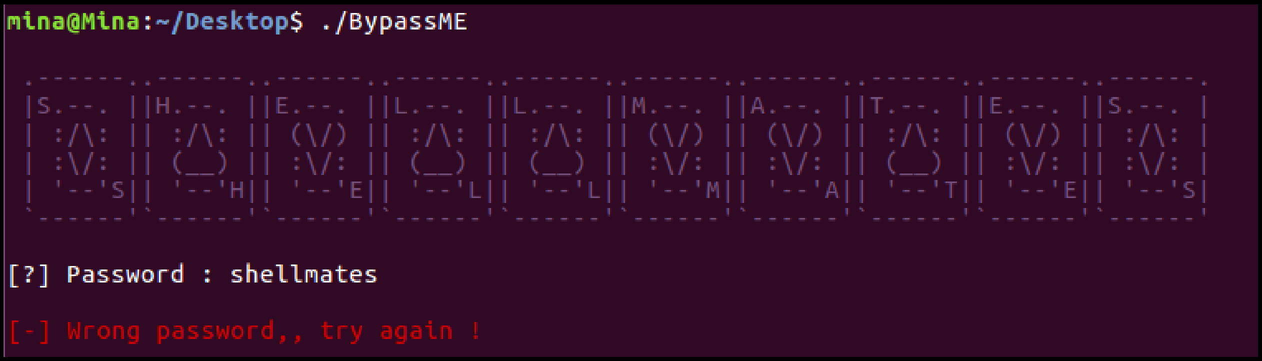
First, let’s look at the program’s strings (strings BypassME):



```
mina@mina:~/Bureau$ strings ./BypassME
/lib/ld-linux.so.2
#u[Y
libc.so.6
_IO_stdin_used
stdin
printf
strlen
getline
__cxa_finalize
__libc_start_main
GLIBC_2.1.3
GLIBC_2.0
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
UWVS
[^ ]
shellmates
0]<8A
```

[2] Non-stripped binaries have debugging information built into it (symbol table…) so we can find the functions names and other information. Whereas, stripped binaries remove this debugging information from the executable for example instead of finding the function’s name we’ll find its address.

We can notice the string “shellmates”, is it the password we’re looking for?

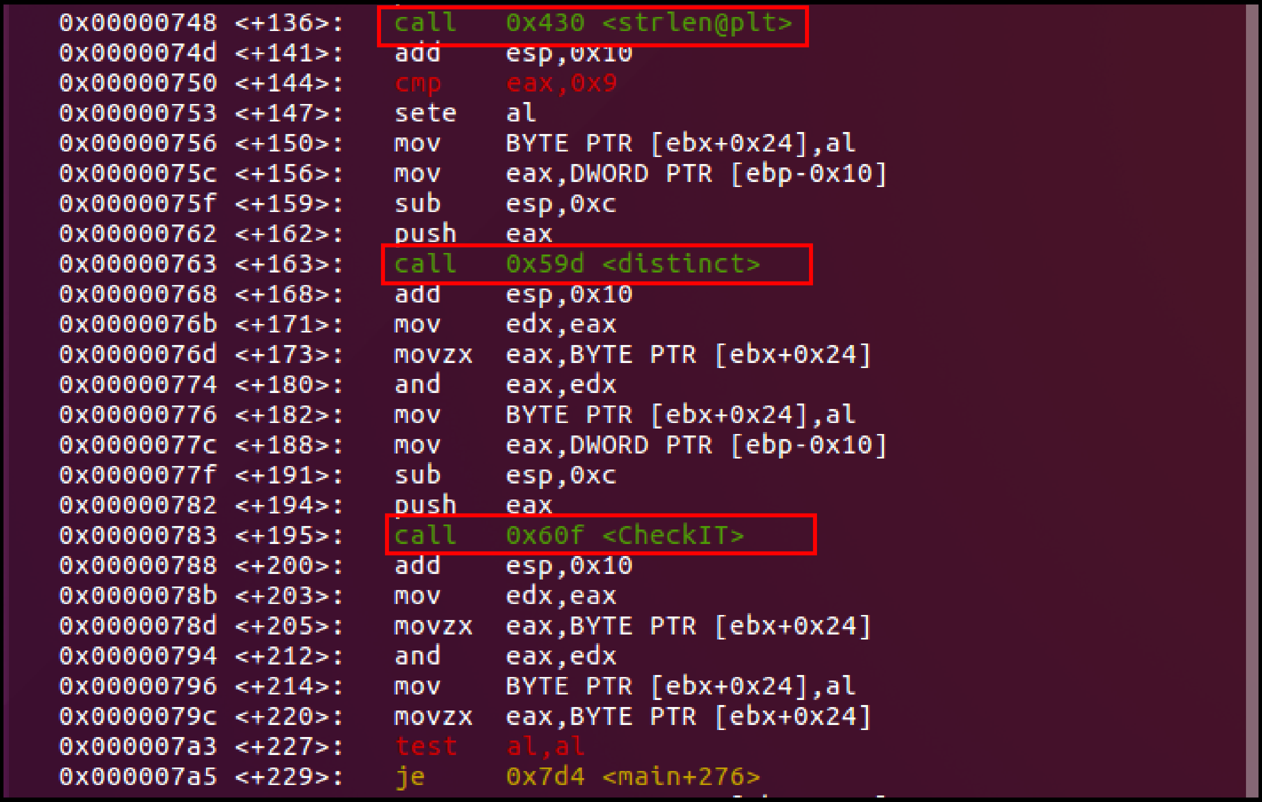


```
mina@Mina:~/Desktop$ ./BypassME

 .------..------..------..------..------..------..------..------..------..------.
 |S.--. ||H.--. ||E.--. ||L.--. ||L.--. ||M.--. ||A.--. ||T.--. ||E.--. ||S.--. |
 | :/\: || :/\: || (\/) || :/\: || :/\: || (\/) || (\/) || :/\: || (\/) || :/\: |
 | :\/: || (__) || :\/: || (__) || (__) || :\/: || :\/: || (__) || :\/: || :\/: |
 | '--'S|| '--'H|| '--'E|| '--'L|| '--'L|| '--'M|| '--'A|| '--'T|| '--'E|| '--'S|
 `------'`------'`------'`------'`------'`------'`------'`------'`------'`------'

[?] Password : shellmates

[-] Wrong password,, try again !
```

Trying “shellmates” didn’t work, it’s not our password ☹ (but we can be sure that our program use it somehow) … ok move on.

Resolution:

Let’s have a deep look on our program using gdb peda and disassemble the main (pd main):



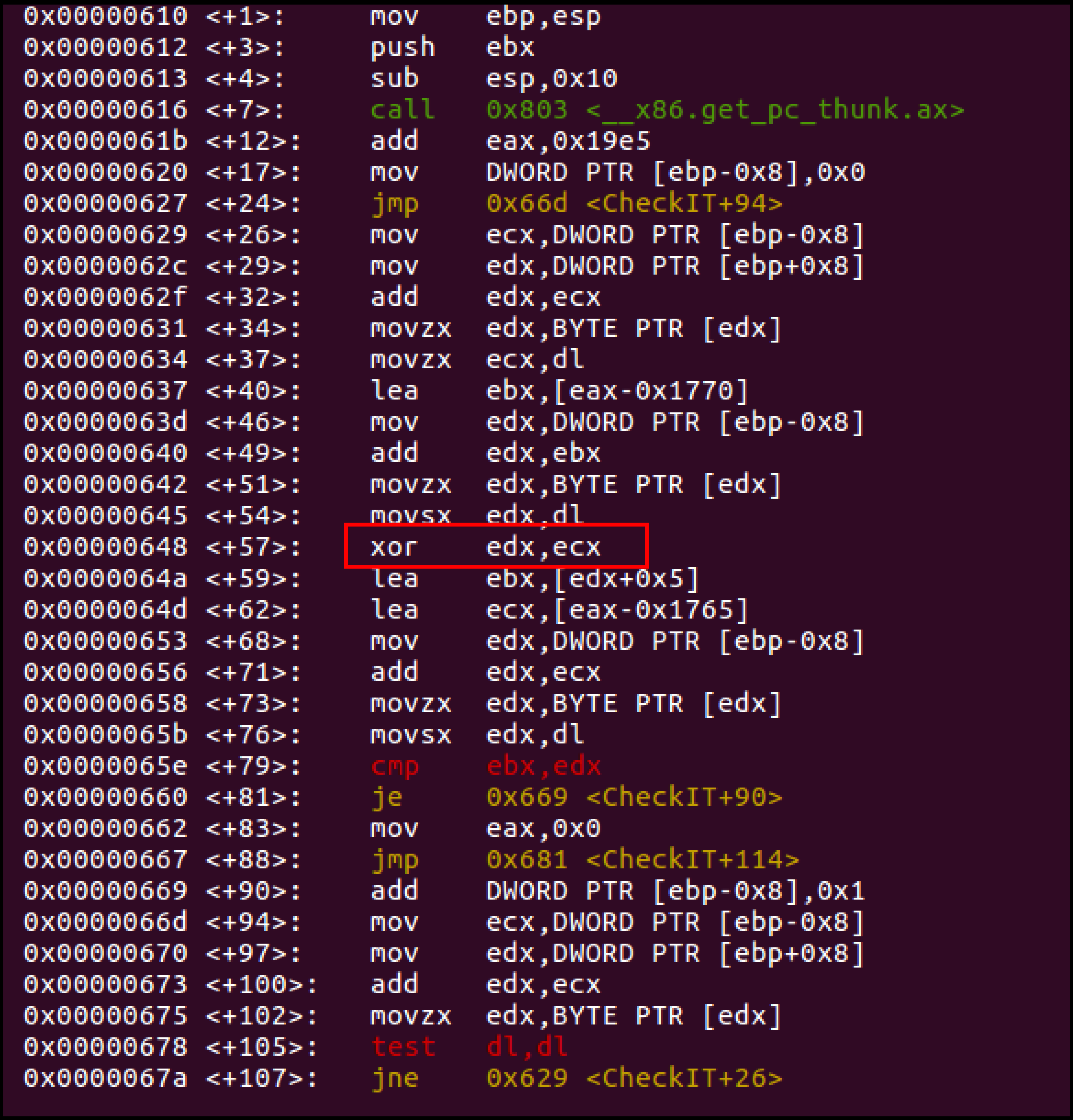
```
   0x00000748 <+136>:	call   0x430 <strlen@plt>
   0x0000074d <+141>:	add    esp,0x10
   0x00000750 <+144>:	cmp    eax,0x9
   0x00000753 <+147>:	sete   al
   0x00000756 <+150>:	mov    BYTE PTR [ebx+0x24],al
   0x0000075c <+156>:	mov    eax,DWORD PTR [ebp-0x10]
   0x0000075f <+159>:	sub    esp,0xc
   0x00000762 <+162>:	push   eax
   0x00000763 <+163>:	call   0x59d <distinct>
   0x00000768 <+168>:	add    esp,0x10
   0x0000076b <+171>:	mov    edx,eax
   0x0000076d <+173>:	movzx  eax,BYTE PTR [ebx+0x24]
   0x00000774 <+180>:	and    eax,edx
   0x00000776 <+182>:	mov    BYTE PTR [ebx+0x24],al
   0x0000077c <+188>:	mov    eax,DWORD PTR [ebp-0x10]
   0x0000077f <+191>:	sub    esp,0xc
   0x00000782 <+194>:	push   eax
   0x00000783 <+195>:	call   0x60f <CheckIT>
   0x00000788 <+200>:	add    esp,0x10
   0x0000078b <+203>:	mov    edx,eax
   0x0000078d <+205>:	movzx  eax,BYTE PTR [ebx+0x24]
   0x00000794 <+212>:	and    eax,edx
   0x00000796 <+214>:	mov    BYTE PTR [ebx+0x24],al
   0x0000079c <+220>:	movzx  eax,BYTE PTR [ebx+0x24]
   0x000007a3 <+227>:	test   al,al
   0x000007a5 <+229>:	je     0x7d4 <main+276>
```

Hmmmm, interesting ☺ …we can notice that our program calls three functions:

- **strlen**: used to get a string’s length. We are comparing the length of our input with 9, so the password should certainly have the length 9.

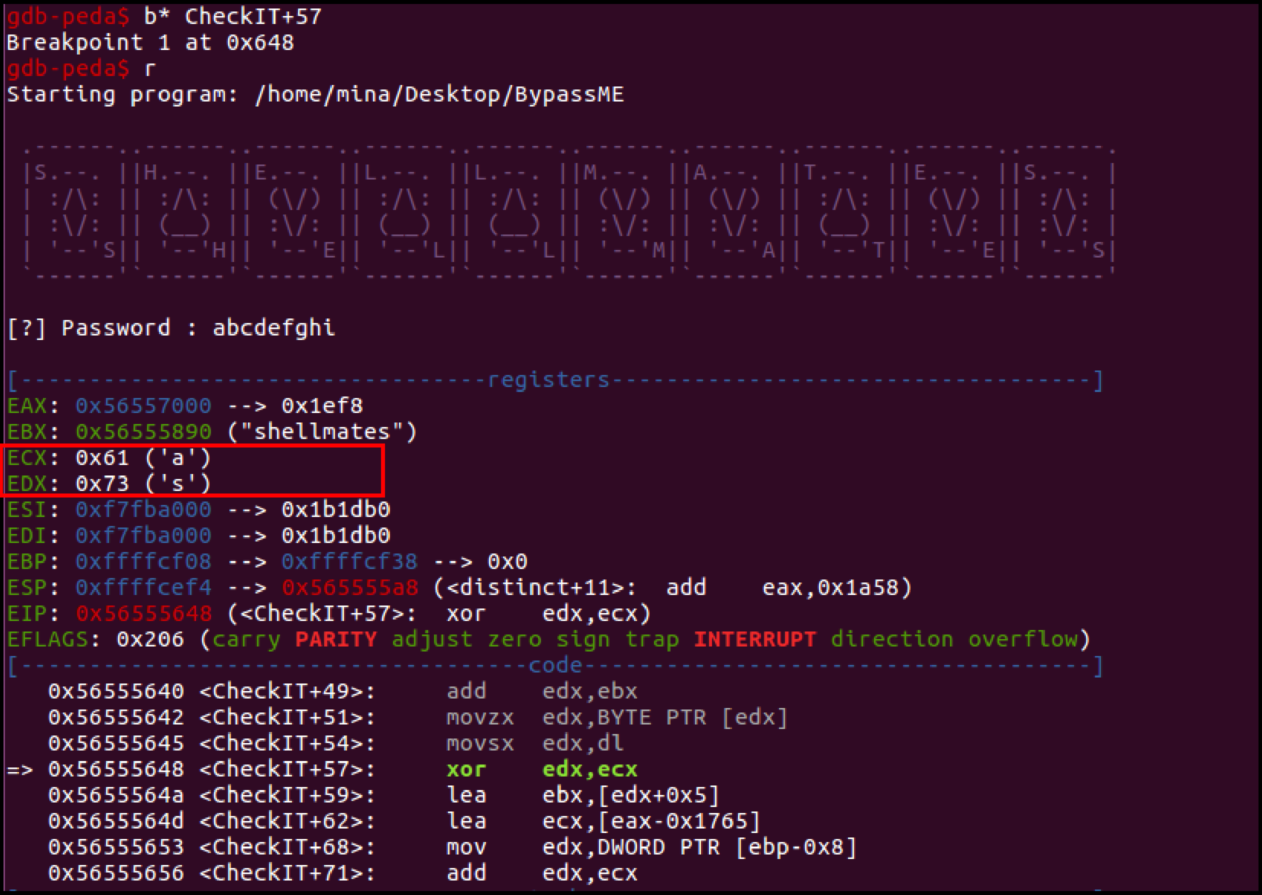
- **distinct**: seems like it checks whether the characters of our input are distinct.

-**CheckIt**: we have no idea how it works so we should take a deeper look (pd checkIt).



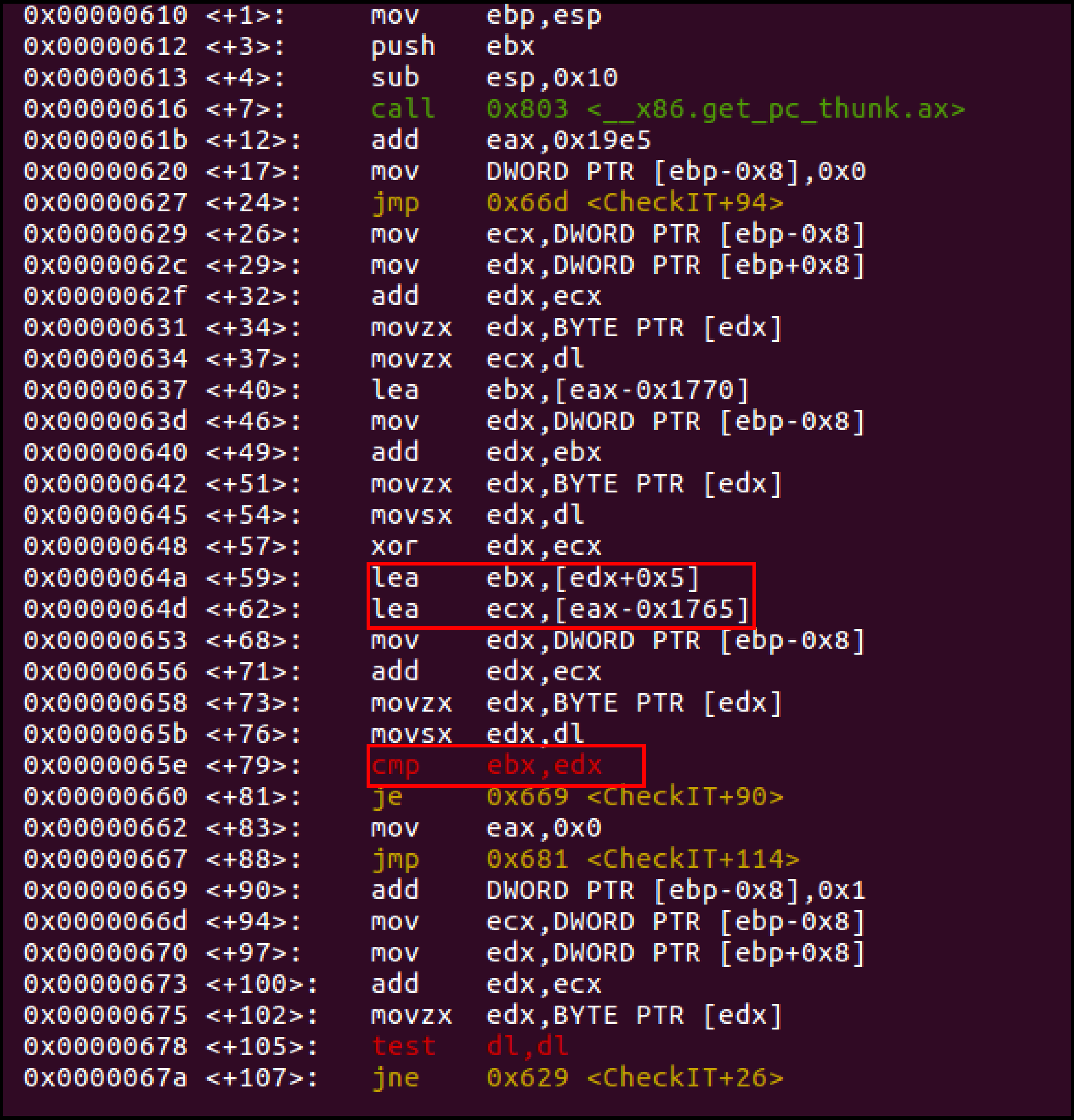
```
 0x00000610 <+1>:     mov    ebp,esp
 0x00000612 <+3>:     push   ebx
 0x00000613 <+4>:     sub    esp,0x10
 0x00000616 <+7>:     call   0x803 <__x86.get_pc_thunk.ax>
 0x0000061b <+12>:    add    eax,0x19e5
 0x00000620 <+17>:    mov    DWORD PTR [ebp-0x8],0x0
 0x00000627 <+24>:    jmp    0x66d <CheckIT+94>
 0x00000629 <+26>:    mov    ecx,DWORD PTR [ebp-0x8]
 0x0000062c <+29>:    mov    edx,DWORD PTR [ebp+0x8]
 0x0000062f <+32>:    add    edx,ecx
 0x00000631 <+34>:    movzx  edx,BYTE PTR [edx]
 0x00000634 <+37>:    movzx  ecx,dl
 0x00000637 <+40>:    lea    ebx,[eax-0x1770]
 0x0000063d <+46>:    mov    edx,DWORD PTR [ebp-0x8]
 0x00000640 <+49>:    add    edx,ebx
 0x00000642 <+51>:    movzx  edx,BYTE PTR [edx]
 0x00000645 <+54>:    movsx  edx,dl
 0x00000648 <+57>:    xor    edx,ecx
 0x0000064a <+59>:    lea    ebx,[edx+0x5]
 0x0000064d <+62>:    lea    ecx,[eax-0x1765]
 0x00000653 <+68>:    mov    edx,DWORD PTR [ebp-0x8]
 0x00000656 <+71>:    add    edx,ecx
 0x00000658 <+73>:    movzx  edx,BYTE PTR [edx]
 0x0000065b <+76>:    movsx  edx,dl
 0x0000065e <+79>:    cmp    ebx,edx
 0x00000660 <+81>:    je     0x669 <CheckIT+90>
 0x00000662 <+83>:    mov    eax,0x0
 0x00000667 <+88>:    jmp    0x681 <CheckIT+114>
 0x00000669 <+90>:    add    DWORD PTR [ebp-0x8],0x1
 0x0000066d <+94>:    mov    ecx,DWORD PTR [ebp-0x8]
 0x00000670 <+97>:    mov    edx,DWORD PTR [ebp+0x8]
 0x00000673 <+100>:   add    edx,ecx
 0x00000675 <+102>:   movzx  edx,BYTE PTR [edx]
 0x00000678 <+105>:   test   dl,dl
 0x0000067a <+107>:   jne    0x629 <CheckIT+26>
```

We notice a “XOR” between edx and ecx , we can make a breakpoint at that step to discover what is in edx and ecx:

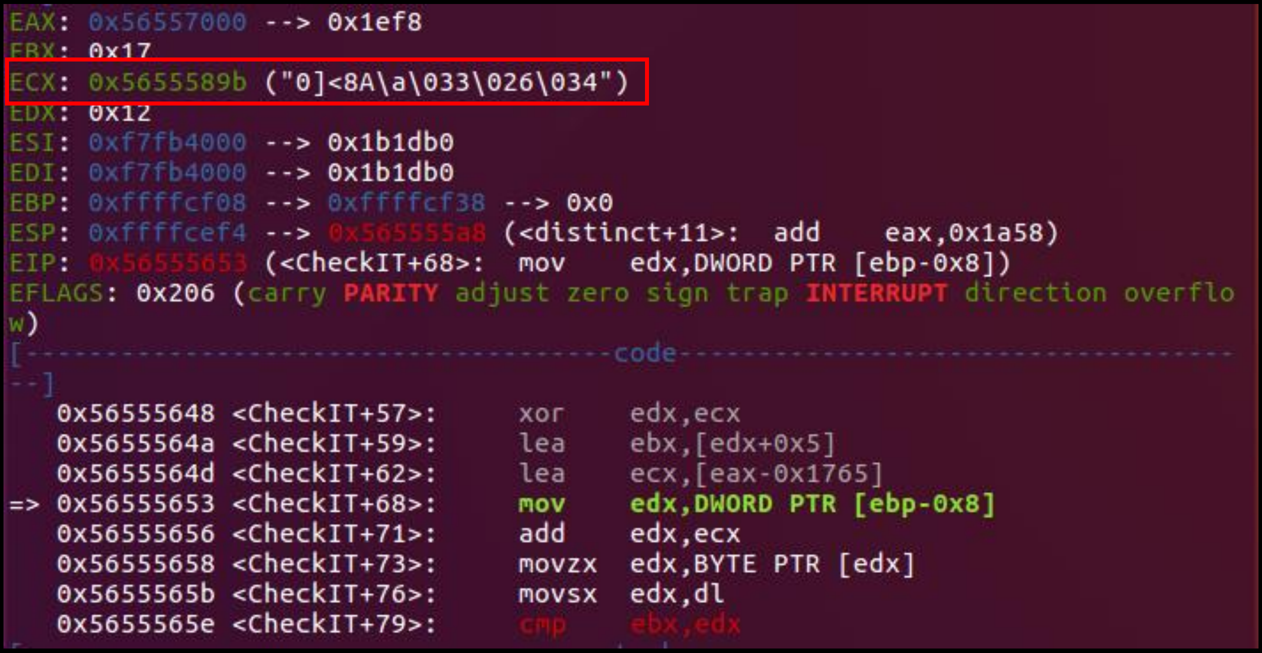


```
gdb-peda$ b* CheckIT+57
Breakpoint 1 at 0x648
gdb-peda$ r
Starting program: /home/mina/Desktop/BypassME

 .------..------..------..------..------..------..------..------..------..------.
 |S.--. ||H.--. ||E.--. ||L.--. ||L.--. ||M.--. ||A.--. ||T.--. ||E.--. ||S.--. |
 | :/\: || :/\: || (\/) || :/\: || :/\: || (\/) || (\/) || :/\: || (\/) || :/\: |
 | :\/: || (__) || :\/: || (__) || (__) || :\/: || :\/: || (__) || :\/: || :\/: |
 | '--'S|| '--'H|| '--'E|| '--'L|| '--'L|| '--'M|| '--'A|| '--'T|| '--'E|| '--'S|
 `------'`------'`------'`------'`------'`------'`------'`------'`------'`------'

[?] Password : abcdefghi

[----------------------------------registers-----------------------------------]
EAX: 0x56557000 --> 0x1ef8
EBX: 0x56555890 ("shellmates")
ECX: 0x61 ('a')
EDX: 0x73 ('s')
ESI: 0xf7fba000 --> 0x1b1db0
EDI: 0xf7fba000 --> 0x1b1db0
EBP: 0xffffcf08 --> 0xffffcf38 --> 0x0
ESP: 0xffffcef4 --> 0x565555a8 (<distinct+11>:  add    eax,0x1a58)
EIP: 0x56555648 (<CheckIT+57>:  xor    edx,ecx)
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x56555640 <CheckIT+49>:     add    edx,ebx
   0x56555642 <CheckIT+51>:     movzx  edx,BYTE PTR [edx]
   0x56555645 <CheckIT+54>:     movsx  edx,dl
=> 0x56555648 <CheckIT+57>:     xor    edx,ecx
   0x5655564a <CheckIT+59>:     lea    ebx,[edx+0x5]
   0x5655564d <CheckIT+62>:     lea    ecx,[eax-0x1765]
   0x56555653 <CheckIT+68>:     mov    edx,DWORD PTR [ebp-0x8]
   0x56555656 <CheckIT+71>:     add    edx,ecx
```

So we're doing a "XOR" between the first character of our input and the first one of the string "shellmates" (we do the same to the rest of the characters) then we add (0x5) and we compare it to a string that we find its address in ecx.

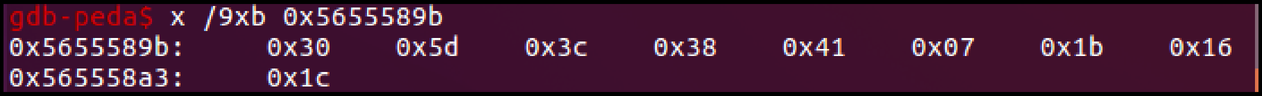


```
   0x00000610 <+1>:     mov    ebp,esp
   0x00000612 <+3>:     push   ebx
   0x00000613 <+4>:     sub    esp,0x10
   0x00000616 <+7>:     call   0x803 <__x86.get_pc_thunk.ax>
   0x0000061b <+12>:    add    eax,0x19e5
   0x00000620 <+17>:    mov    DWORD PTR [ebp-0x8],0x0
   0x00000627 <+24>:    jmp    0x66d <CheckIT+94>
   0x00000629 <+26>:    mov    ecx,DWORD PTR [ebp-0x8]
   0x0000062c <+29>:    mov    edx,DWORD PTR [ebp+0x8]
   0x0000062f <+32>:    add    edx,ecx
   0x00000631 <+34>:    movzx  edx,BYTE PTR [edx]
   0x00000634 <+37>:    movzx  ecx,dl
   0x00000637 <+40>:    lea    ebx,[eax-0x1770]
   0x0000063d <+46>:    mov    edx,DWORD PTR [ebp-0x8]
   0x00000640 <+49>:    add    edx,ebx
   0x00000642 <+51>:    movzx  edx,BYTE PTR [edx]
   0x00000645 <+54>:    movsx  edx,dl
   0x00000648 <+57>:    xor    edx,ecx
   0x0000064a <+59>:    lea    ebx,[edx+0x5]
   0x0000064d <+62>:    lea    ecx,[eax-0x1765]
   0x00000653 <+68>:    mov    edx,DWORD PTR [ebp-0x8]
   0x00000656 <+71>:    add    edx,ecx
   0x00000658 <+73>:    movzx  edx,BYTE PTR [edx]
   0x0000065b <+76>:    movsx  edx,dl
   0x0000065e <+79>:    cmp    ebx,edx
   0x00000660 <+81>:    je     0x669 <CheckIT+90>
   0x00000662 <+83>:    mov    eax,0x0
   0x00000667 <+88>:    jmp    0x681 <CheckIT+114>
   0x00000669 <+90>:    add    DWORD PTR [ebp-0x8],0x1
   0x0000066d <+94>:    mov    ecx,DWORD PTR [ebp-0x8]
   0x00000670 <+97>:    mov    edx,DWORD PTR [ebp+0x8]
   0x00000673 <+100>:   add    edx,ecx
   0x00000675 <+102>:   movzx  edx,BYTE PTR [edx]
   0x00000678 <+105>:   test   dl,dl
   0x0000067a <+107>:   jne    0x629 <CheckIT+26>
```



```
EAX: 0x56557000 --> 0x1ef8
EBX: 0x17
ECX: 0x5655589b ("0]<8A\a\033\026\034")
EDX: 0x12
ESI: 0xf7fb4000 --> 0x1b1db0
EDI: 0xf7fb4000 --> 0x1b1db0
EBP: 0xffffcf08 --> 0xffffcf38 --> 0x0
ESP: 0xffffcef4 --> 0x565555a8 (<distinct+11>:  add    eax,0x1a58)
EIP: 0x56555653 (<CheckIT+68>:  mov    edx,DWORD PTR [ebp-0x8])
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflo
w)
[-----------------------------------code-----------------------------------
--]
   0x56555648 <CheckIT+57>:     xor    edx,ecx
   0x5655564a <CheckIT+59>:     lea    ebx,[edx+0x5]
   0x5655564d <CheckIT+62>:     lea    ecx,[eax-0x1765]
=> 0x56555653 <CheckIT+68>:     mov    edx,DWORD PTR [ebp-0x8]
   0x56555656 <CheckIT+71>:     add    edx,ecx
   0x56555658 <CheckIT+73>:     movzx  edx,BYTE PTR [edx]
   0x5655565b <CheckIT+76>:     movsx  edx,dl
   0x5655565e <CheckIT+79>:     cmp    ebx,edx
```

So to get the adress of that string we keep on executing the command (si) to step an instruction until the one where we load the adress in ecx and then we use use the command (x x/9xb 0x56555889b) to examine the address content (we want to show 9 bytes):



```
gdb-peda$ x /9xb 0x5655589b
0x5655589b:     0x30    0x5d    0x3c    0x38    0x41    0x07    0x1b    0x16
0x565558a3:     0x1c
```

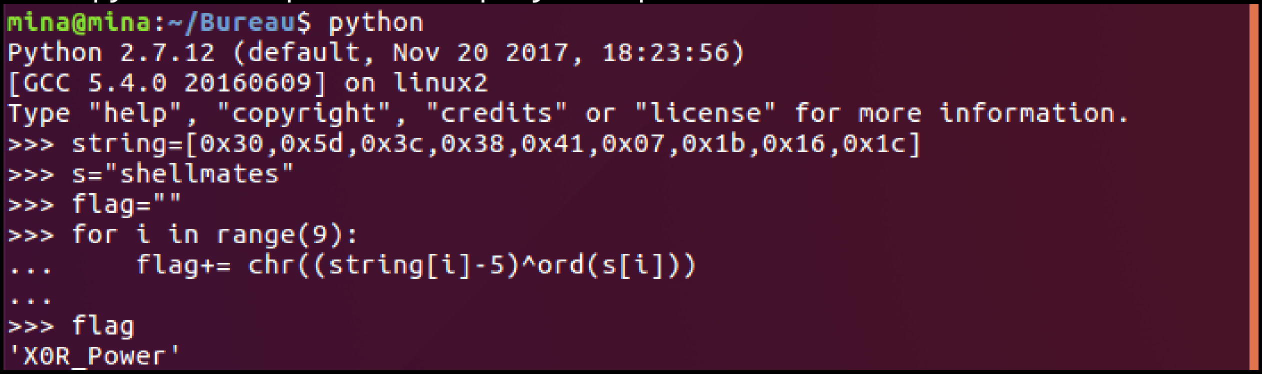
To sum up, what we do in the program is a xor between our input and “shellmates” and we add 5 to result (character by character) then we check whether it is equal to a certain string:

(input XOR ”shellmates”)+5=string

So, to get our password, we need to do the inverse operation:

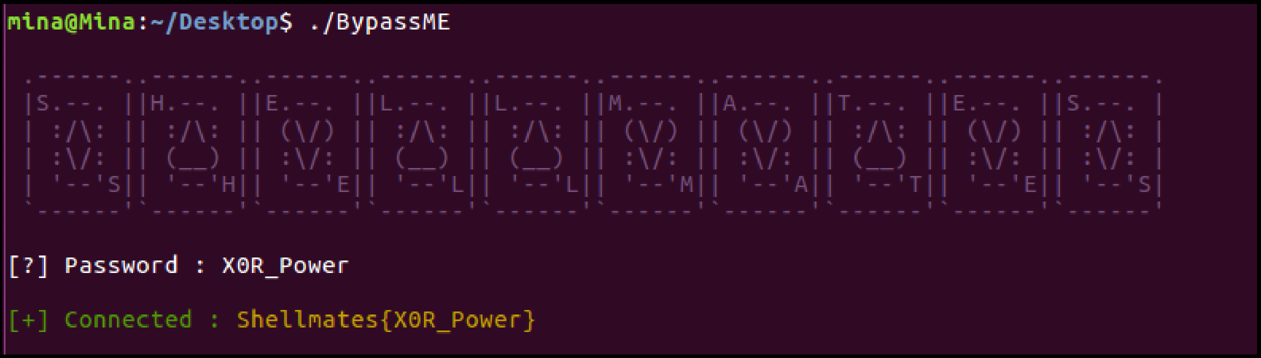
password= (string-5) XOR ”shellmates”:

This python script will display the password:



```
mina@mina:~/Bureau$ python
Python 2.7.12 (default, Nov 20 2017, 18:23:56)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> string=[0x30,0x5d,0x3c,0x38,0x41,0x07,0x1b,0x16,0x1c]
>>> s="shellmates"
>>> flag=""
>>> for i in range(9):
...     flag+= chr((string[i]-5)^ord(s[i]))
...
>>> flag
'XOR_Power'
```

We use it to find the flag:



```
mina@Mina:~/Desktop$ ./BypassME

 .------..------..------..------..------..------..------..------..------..------.
 |S.--. ||H.--. ||E.--. ||L.--. ||L.--. ||M.--. ||A.--. ||T.--. ||E.--. ||S.--. |
 | :/\: || :/\: || (\/) || :/\: || :/\: || (\/) || (\/) || :/\: || (\/) || :/\: |
 | :\/: || (__) || :\/: || (__) || (__) || :\/: || :\/: || (__) || :\/: || :\/: |
 | '--'S|| '--'H|| '--'E|| '--'L|| '--'L|| '--'M|| '--'A|| '--'T|| '--'E|| '--'S|
 `------'`------'`------'`------'`------'`------'`------'`------'`------'`------'

[?] Password : X0R_Power

[+] Connected : Shellmates{X0R_Power}
```

Flag: Shellmates{X0R_Power}

What we learn from this task

We learned through this challenge the different commands that can help us in reversing binaries:

- file filename: to determine the file type[3].
- strings filename: to print the strings of printable characters in files[4]. This command was very useful in this challenge, we got the password easily.
- Debugging a program using **gdb peda**:
    - PEDA (Python Exploit Development Assistance for GDB) enhance the display of gdb: colorize and display disassembly codes, registers, memory information during debugging. It adds commands to support debugging and exploit development too (for a full list of commands use peda help)[5].
    - The command pd (or pdisas) is a gdb disassemble command, the argument can be a function name (if the file is not stripped) like we did in this challenge (pd main, pd CheckIT) or we could rather use an address with the syntax pd address /NN (NN is the number of instructions we won’t to disassemble).
    - The command b* (breakpoint*) is used to make program stop in certain points (breakpoints).
    - The command r (run) is used to start the program being debugged.

[3] man file
[4] man strings
[5] https://github.com/longld/peda

- The command c (continue) is used to continue running the program being debugged after a breakpoint.
- The command si (or step) to execute the next instruction (step one instruction).
- The command x /Nxb address (or examine) to examine memory content: N is number of bytes we won't to show (if we have xb, else it's according to the format of the information), x is for the format, here is for the hexadecimal (there is also o for octal, d for decimal…) , b is for size , here b for byte (there is also w for word, I for instruction…)

We learned in this challenge, too, how to trace a program and move from a function to another until you find what interest you (what help you to get the flag).

Thanks for reading ☺