6

Lab

Buffer Overflow Attack (Buffer Bomb) Part 2

Thực hành Lập trình Hệ thống

Lưu hành nội bộ

A. TỔNG QUAN

A.1 Mục tiêu

Trong bài lab này, sinh viên sẽ vận dụng những kiến thức về cơ chế của stack trong bộ xử lý IA32, nhận biết code có lỗ hổng buffer overflow trong một file thực thi 32-bit x86 để khai thác lỗ hổng này, từ đó làm thay đổi cách hoạt động của chương trình theo một số mục đích nhất định.

A.2 Môi trường

- IDA Pro trên Windows.
- Môi trường Linux trên máy ảo.
- Các file source của bài lab:
  1. **bufbomb**: file thực thi Linux 32-bit chứa lỗ hổng buffer overflow cần khai thác.
  2. **makecookie**, **hex2raw**: một số file hỗ trợ.

A.3 Liên quan

- Có kiến thức về cách mà hệ thống phân vùng bộ nhớ.
- Có kỹ năng sử dụng một số công cụ debug như **IDA, gdb**.
- Kiến thức về remote debugger trên desktop khi sử dụng IDA

B. NHẮC LẠI VỀ BUFFER BOMB LAB

B.1 Chương trình bufbomb

bufbomb là file thực thi dạng command line có lỗ hổng buffer overflow, nhận tham số đầu vào là một chuỗi. Khi chạy, **bufbomb** đi kèm nhiều option như sau:

-u userid Thực thi **bufbomb** của một user nhất định. Khi thực hiện bài lab ***luôn phải cung cấp tham số*** này vì bufbomb có tạo 1 giá trị cookie dựa trên userid, và ở 1 số level, cookie này sẽ được dùng để đánh giá solution đúng hay sai.

-h In danh sách các option có thể dùng với bufbomb.

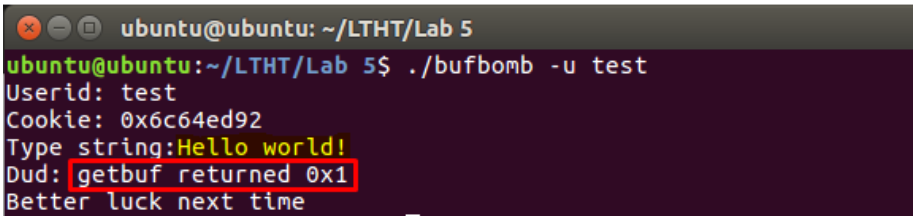
Trong hoạt động của **bufbomb** nhận một chuỗi đầu vào với hàm **getbuf** như sau:

```
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  }
```

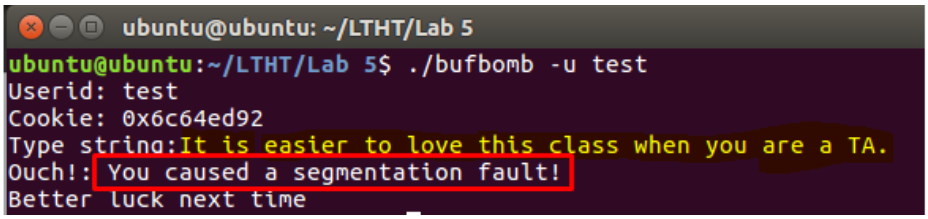
Hàm **Gets** giống với thư viện hàm chuẩn **gets** – đọc một chuỗi đầu vào và lưu nó ở một vị trí đích xác định. Trong đoạn code phía trên, có thể thấy vị trí lưu này là một mảng buf có không gian được cấp vừa đủ cho 32 ký tự.

Vấn đề ở đây là, khi lưu chuỗi, hàm **Gets** không có cơ chế 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 vào vị trí đích đó, có thể làm tràn ra khỏi vùng nhớ được cấp trước đó

Với **bufbomb**, trong trường hợp nhập vào một chuỗi có độ dài không vượt quá 31 ký tự, **getbuf** hoạt động bình thường và sẽ trả về 1, như ví dụ thực thi ở dưới:



Tuy nhiên, thử nhập một chuỗi dài hơn 31 ký tự, có thể xảy ra lỗi:



Khi tràn bộ nhớ thường khiến chương trình bị gián đoạn, dẫn đến lỗi truy xuất bộ nhớ.

Trong bài thực hành này, đối tượng cần khai thác chủ yếu là **getbuf** và stack của nó.

B.2 Một số file hỗ trợ

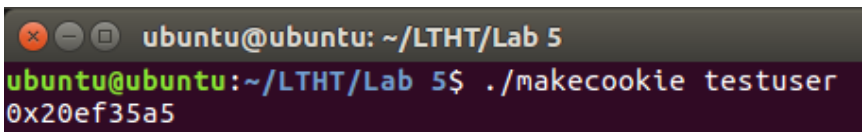
Bên cạnh file chính là **bufbomb,** thư mục source của Buffer Bomb lab gồm một số file hỗ trợ quá trình thực hiện bài thực hành:

- **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. Cookie này cần được dùng trong một số level của bài lab.

Cookie có thể được tạo như sau:

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



- **hex2raw** (dành cho hệ thống **64 bit**)

*Lưu ý: file này chỉ thực thi được trên hệ thống Linux **64-bit**.*

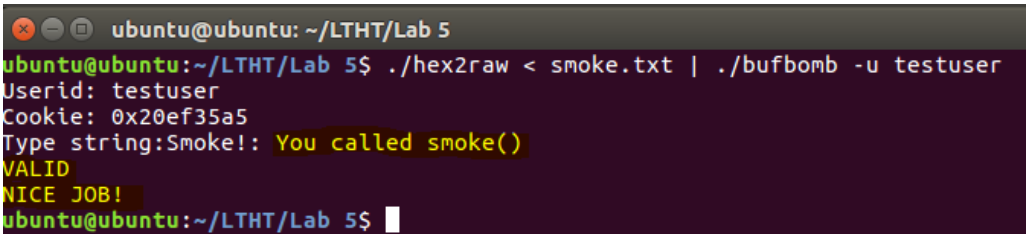
hex2raw sẽ giúp chuyển những byte giá trị không tuân theo bảng mã ASCII (các byte không gõ được từ bàn phím) sang chuỗi có thể truyền làm input cho file **bufbomb**. **hex2raw** nhận đầu vào là chuỗi dạng hexan, mỗi byte được biểu diễn bởi ***2 số hexan*** và các byte cách nhau bởi khoảng trắng (khoảng trống hoặc xuống dòng). Ví dụ chuỗi các byte: **00 0C 12 3B 4C**

Cách dùng: soạn sẵn giá trị của các byte trong một file text với đúng định dạng yêu cầu sau đó truyền vào cho **hex2raw** bằng lệnh sau:

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

B.3 Một số lưu ý

- Các lưu ý khi tạo các byte của chuỗi exploit:
  1. Chuỗi exploit ***không được*** chứa byte hexan **0A** ở bất kỳ 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.
  2. **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.
  3. Để tạo ra một word **0xDEADBEEF**, cần truyền **EF BE AD DE** (đổi vị trí các byte) cho **hex2raw**.
- Khi sinh viên đã giải quyết đúng một trong các mức độ, ví dụ level 0 sẽ có thông báo:



C. Các bước khai thác file bufbomb

Bước 1. Chọn userid và tạo cookie tương ứng.

Chọn 1 userid tuỳ ý và sử dụng **makecookie** được cung cấp để tạo cookie tương ứng.

Bước 2. Xác định chuỗi exploit cho từng level

2.1 Xác định độ dài chuỗi exploit: dựa vào kích thước buffer và khoảng cách đến vùng muốn ghi đè.

2.2 Xác định nội dung chuỗi exploit: thêm những byte có nội dung tuỳ theo yêu cầu của level.

Bước 3. Thực hiện truyền chuỗi exploit cho bufbomb.

Sử dụng file **hex2raw** (với hệ thống 64-bit) hoặc code **python** (với hệ thống 32-bit) để tạo chuỗi exploit và truyền vào cho bufbomb.

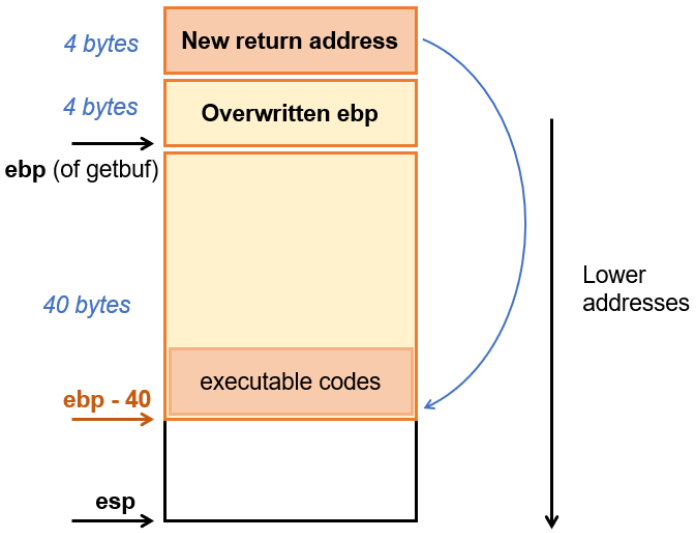
D. CÁC LEVEL NÂNG CAO CỦA BUFFER BOMB LAB

D.1 Sử dụng buffer overflow để chèn và thực thi mã độc

Một dạng phức tạp hơn của tấn công buffer overflow là chuỗi exploit là một chuỗi encode các câu lệnh mã máy có thể thực thi được, được chèn vào stack để thực thi. Khi đó, ***chuỗi exploit sẽ ghi đè địa chỉ trả về ban đầu để trỏ về vị trí 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 nhảy đến vị trí lưu mã thực thi đã chèn vào để thực thi thay vì quay về hàm trước.

Như ở hình bên, do vẫn khai thác hàm **getbuf** như lab trước, chuỗi exploit vẫn có độ dài tối thiểu là 48 bytes, trong đó sinh viên sẽ:

- Tạo và chèn thêm những byte code thực thi của một số lệnh (phần ***executable codes*** màu cam đậm trong hình), có độ dài tuỳ thuộc vào các lệnh mà chúng đại diện.
- Tìm địa chỉ trả về mới phù hợp để ghi đè lên **địa chỉ trả về** (byte 45 đến 48 trong chuỗi exploit – phần màu cam đậm) để thực hiện ý đồ như dấu mũi tên.



- Các byte còn lại (màu cam nhạt) có thể mang giá trị tuỳ ý (khác 0x0A).

- **Tạo mã thực thi**

Để tạo mã thực thi trong chuỗi exploit, sinh viên thực hiện các bước sau:

- Viết code dưới dạng mã assembly trong các file **.s**. Ví dụ:

```
test.s
movl $1, %eax
int $0x80
```

- Chạy các lệnh để tạo các byte code tương ứng (khoanh đỏ) đưa vào chuỗi exploit.

```
$ gcc -m32 -c <file .s đầu vào> -o <file .o đầu ra>
$ objdump -d <file .o>
```

```
ubuntu@ubuntu: ~/LTHT/Lab6
ubuntu@ubuntu:~/LTHT/Lab6$ gcc -m32 -c input.s -o input.o
ubuntu@ubuntu:~/LTHT/Lab6$ objdump -d input.o

input.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   b8 01 00 00 00          mov    $0x1,%eax
   5:   cd 80                   int    $0x80
ubuntu@ubuntu:~/LTHT/Lab6$
```

- **Tìm địa chỉ trả về mới và vị trí mã thực thi**

Để chương trình thực thi được đoạn mã thực thi có trong chuỗi exploit, nên đảm bảo:

(1) **Địa chỉ trả về mới** là ***vị trí lưu*** của chuỗi exploit. Vị trí lưu này chỉ xác định khi chương trình chạy.

(2) Mã thực thi (executable codes) nằm ***ở đầu*** chuỗi exploit.

Các vị trí này có thể tuỳ chỉnh, tuy nhiên **luôn đảm bảo** rằng: **địa chỉ trả về mới *trỏ đúng*** vào **vị trí bắt đầu mã thực thi** trong chuỗi exploit. Nếu không, khi chương trình nhảy đến vị trí những byte không phải mã thực thi, cố gắng thực thi chúng sẽ gây ra lỗi.

Nhiệm vụ của sinh viên là truyền vào các chuỗi exploit có *độ dài và nội dung chứa mã thực thi* phù hợp cho chương trình bufbomb (hay cho getbuf) để nó làm một số công việc thú vị.

D.2 Level 2 - Firecracker

Trong file **bufbomb** có một hàm **bang** (cũng không được gọi trong **bufbomb**) như sau:

```
1  int global_value = 0;
2  void bang(int val)
3  {
4     if (global_value == cookie) {
5        printf("Bang!: You set global_value to 0x%x\n", global_value);
6        validate(2);
7     } else {
8        printf("Misfire: global_value = 0x%x\n", global_value);
9        exit(0);
10    }
11 }
```

Trong hàm này có so sánh giá trị một biến toàn cục **global_value** (được gán ban đầu là 0) với giá trị **cookie**. Khi nào 2 giá trị này bằng nhau thì thành công. Do đó, trước khi gọi **bang**, ta cần chạy một lệnh nào đó để thay đổi được giá trị của **global_value**. Do giá trị **global_value** này không nằm trên stack, nên không thể áp dụng phương pháp ghi đè những vùng nhớ lân cận ở lab trước, cần truyền vào các executable code để thực hiện.

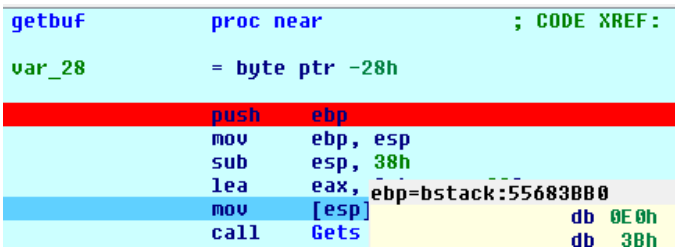
Yêu cầu: Khai thác lỗ hổng buffer overflow để truyền vào **bufbomb** một chuỗi exploit có chứa mã thực thi sao cho:

- Thay đổi được giá trị của **global_value** trước khi gọi hàm **bang**.
- Thực thi hàm **bang** thay vì trở về hàm **test**.

Gợi ý:

- Địa chỉ trả về mới nên là địa chỉ bắt đầu lưu chuỗi exploit trong stack, ở vị trí **ebp – 40 bytes** trong stack của **getbuf.** Để tìm được địa chỉ này, cần debug chương trình với

breakpoint ở **getbuf** để xem giá trị của **ebp** (hoặc **ebp-40**). Ví dụ debug với IDA Pro ta có ở hình dưới, khi chạy đến dòng lệnh màu xanh dương, giá trị của **ebp** = **0x55683BB0**:



- Trong mã thực thi cần thực hiện 2 công việc:
  1. Gán giá trị cookie của userid cho biến toàn cục **global_value**.
  2. Nhảy đến hàm **bang** để thực thi tiếp.
- Một cách để đến thực thi 1 hàm:

```
push <địa chỉ của hàm muốn thực thi>
ret
```

D.3 Level 3 - Dynamite

Ở các level trước, trong quá trình bị tấn công buffer overflow, có 1 số vùng nhớ trên stack bị ghi đè bằng những byte tuỳ ý, trong đó có những vùng nhớ chứa thông tin liên quan đến các trạng thái thanh ghi/bộ nhớ của hàm mẹ (hàm **test**). Việc ghi đè này có thể khiến chương trình không thể quay về hàm ban đầu sau khi bị buffer overflow. Các tấn công ở những level trước chỉ khiến cho chương trình nhảy đến đoạn code của những hàm khác, sau đó thoát chương trình, do đó ảnh hưởng này không rõ rệt. Mục đích của level này là làm cho chương trình sau khi bị khai thác vẫn có thể quay về hàm mẹ ban đầu (hàm **test**). Để làm được điều đó, sinh viên cần biết được các trạng thái thanh ghi/bộ nhớ của hàm mẹ đã bị thay đổi và sau đó khôi phục lại giá trị đúng.

Kiểu tấn công này cần thực hiện các bước:

1) Đưa được mã thực thi lên stack thông qua input

2) Thay đổi địa chỉ trả về thành địa chỉ bắt đầu của chuỗi exploit chứa mã thực thi

3) Trong đoạn mã thực thi, bên cạnh việc thực hiện một công việc nào đó, cần khôi phục bất kỳ thay đổi nào đã gây ra với stack và trở về đúng hàm mẹ ban đầu.

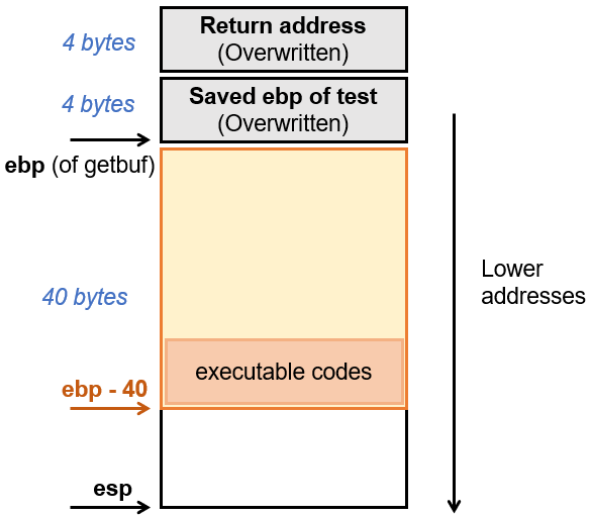
Yêu cầu: Khai thác lỗ hổng buffer overflow để truyền vào một chuỗi exploit chứa mã thực thi sao cho **getbuf** khi thực thi xong sẽ trả về cookie tương ứng với userid cho hàm **test**, thay vì trả về 1. Mã thực thi cần có các lệnh thực hiện các công việc:

- Gán cookie vào giá trị trả về.
- Khôi phục các trạng thái thanh ghi/bộ nhớ bị thay đổi của hàm mẹ (**test**)
- Đẩy địa chỉ trả về đúng vào stack (là 1 câu lệnh cần thực thi tiếp theo của **test**).
- Thực thi câu lệnh **ret** để trở về **test**.

Gợi ý:

- Giá trị trả về của hàm thường lưu trong thanh ghi %eax.
- Quan sát stack trước và sau khi lưu input, không tính địa chỉ trả về bị thay đổi, có ô nhớ lưu **ebp của test** bị ghi đè. Làm cách nào tìm được giá trị ban đầu của nó?
- Có thể khôi phục các giá trị thanh ghi bằng 1 trong 2 cách:
  (1) Ghi đè trực tiếp giá trị lên ô nhớ ở vị trí tương ứng trong chuỗi exploit.
  (2) Khôi phục trong mã thực thi bằng lệnh:

```
movl <giá trị>, <thanh ghi/địa chỉ>
```



E. YÊU CẦU & ĐÁNH GIÁ

E.1 Yêu cầu

Sinh viên thực hành và nộp bài **theo nhóm từ 2 - 3 sinh viên** theo thời gian quy định. Sinh viên nộp cả file báo cáo trình bày các bước thực hiện và exploit code cho từng level.

Lưu ý: báo cáo cần ghi rõ nhóm sinh viên thực hiện.

Toàn bộ project đặt vào 1 file nén (.rar/.zip) với tên theo quy tắc sau:

LabX-MSSV1-MSSV2-MSSV3

Ví dụ: *Lab2-16520xxx-17520yyy-18520zzz*

E.2 Đánh giá kết quả

Tiêu chí	Chuyên cần (A)	Điểm bài lab (B)	Kết quả
Điểm	Có mặt = 10 Vắng = 7	Tối đa = 10 Không nộp bài = 0	**Điểm = (A + B)/2**
Lưu ý	**Vắng + Không nộp bài: 0**		

F. THAM KHẢO

[1] Randal E. Bryant, David R. O'Hallaron (2011). *Computer System: A Programmer's Perspective (CSAPP)*

[2] Hướng dẫn sử dụng công cụ dịch ngược IDA Debugger – phần 1 [Online] https://securitydaily.net/huong-dan-su-dung-cong-cu-dich-nguoc-ma-may-ida-debugger-phan-1/

HẾT