

6

Lab

Buffer Overflow Attack Buffer Bomb

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 để khai thác lỗ hổng này, từ đó làm thay đổi cách hoạt động của chương trình theo một số mục đích nhất định.

Bài thực hành được chia làm 2 phần:

- Phần cơ bản: Level 0 – 1.
- Phần nâng cao: Level 2 – 3.

A.2 Môi trường

- Môi trường debug file thực thi Linux 32-bit:
 - + **Cách 1:** Remote debug từ máy Windows cài IDA Pro sang máy Linux chạy file thực thi (xem hướng dẫn ở Lab 4 hoặc URL đính kèm).
 - + **Cách 2:** Sử dụng GDB trên máy Linux.
- 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 để debug như **IDA, gdb**.

B. NHẮC LẠI KIẾN THỨC VỀ STACK VÀ GỌI HÀM TRONG IA32

B.1 Stack

Stack là một vùng nhớ được quản lý với quy tắc ngăn xếp. Đặc điểm quan trọng của stack cần ghi nhớ:

- Stack hoạt động theo cơ chế First In Last Out.
- Stack phát triển từ vùng nhớ có địa chỉ cao xuống vùng nhớ có địa chỉ thấp.
- Thanh ghi **esp** luôn trỏ đến địa chỉ thấp nhất trong stack (“đỉnh” stack)

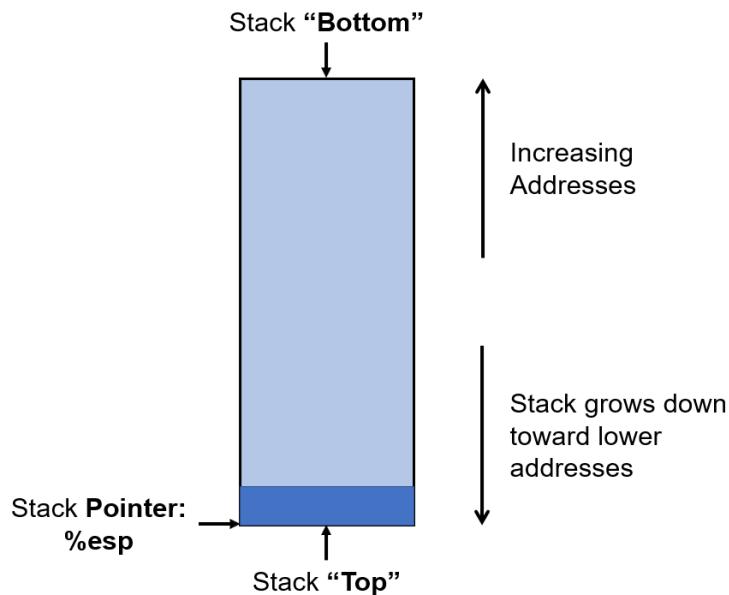
2 hoạt động chính của stack:

- **Push** dữ liệu vào stack: **push src**

esp sẽ trừ xuống 4 bytes để cấp không gian lưu dữ liệu. Giá trị trong **src** sẽ được ghi vào ô nhớ mà **esp** đang trỏ đến. Hoạt động này làm tăng kích thước stack.

- **Pop** dữ liệu ra từ stack: **pop dst**

Đọc 1 giá trị tại địa chỉ mà **esp** đang trỏ đến và ghi vào **dst**. Sau đó tăng giá trị của **esp** lên 4 bytes. Hoạt động này làm giảm kích thước stack.



B.2 Stack trong hỗ trợ gọi và thực thi hàm

- **Stack frame**

Việc gọi hàm làm thay đổi luồng hoạt động giống như câu lệnh jump, tuy nhiên điểm khác biệt là sau khi thực thi xong hàm được gọi (hàm con), cần trả quyền điều khiển chương trình cho hàm gọi (hàm mẹ). Việc này có thể thực hiện được với sự hỗ trợ của stack.

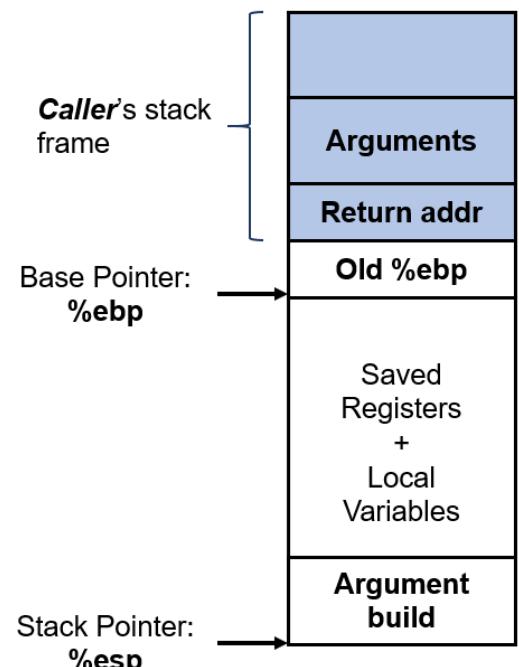
Mỗi hàm (procedure) sẽ có riêng 1 stack frame cho các hoạt động của nó, được định nghĩa là vùng nhớ nằm giữa 2 địa chỉ lưu trong thanh ghi **esp** và **ebp**.

- **Thanh ghi %ebp (base/frame pointer)** trỏ đến **bottom** của stack và là một địa chỉ cố định trong hoạt động của một procedure.
- **Thanh ghi %esp (stack pointer)** lưu trỏ đến **top** (địa chỉ thấp nhất) của stack và có giá trị thay đổi cho mỗi hoạt động thêm hoặc đọc dữ liệu từ stack.

Stack có thể chứa các tham số của hàm, các biến cục bộ, các dữ liệu cần thiết để khôi phục stack frame của hàm trước đó. Sau đây ta quy ước: Hàm gọi hàm khác (*caller*) là hàm mẹ, hàm được gọi (*callee*) là hàm con.

- **Gọi hàm với lệnh call**

Các bước để gọi một hàm (procedure):



Lab 6 – Buffer Bomb

- B1: Hàm mẹ đưa các tham số cần thiết (nếu có) vào stack trước khi gọi hàm con.
- B2: Thực hiện lệnh gọi hàm **call label**, trong đó label trả đến vị trí của hàm con.

Với câu lệnh **call** này, có 2 công việc được thực hiện:

- (1) Địa chỉ trả về sẽ được push vào stack. Đây là địa chỉ của câu lệnh ngay phía sau câu lệnh **call**, hay là địa chỉ hiện tại lưu trong con trỏ lệnh **%eip**. Địa chỉ này **đóng vai trò quan trọng** trong luồng thực thi chương trình sau khi thực thi xong 1 hàm.
- (2) Nhảy đến địa chỉ ở label.

- **Thực thi hàm**

Khi bắt đầu thực thi hàm con, công việc đầu tiên là lưu lại trạng thái của stack frame của hàm mẹ. Do các hàm đều cần sử dụng chung thanh ghi **%ebp** để định nghĩa stack frame của mình, hàm con cần phải lưu lại **%ebp** của hàm mẹ trước khi sử dụng và khôi phục khi trả về. Trong quá trình thực thi một hàm, **%esp** sẽ được thay đổi để cấp phát hoặc thu hồi vùng nhớ trong stack. Các thanh ghi **%esp** và **%ebp** cũng được dùng trong từng stack frame để truy xuất các tham số và biến của từng hàm.

- **Trả về hàm**

Sau khi hàm con thực thi xong, trước khi trở về hàm mẹ, hàm con cần thu dọn stack của mình và khôi phục một số trạng thái (thanh ghi) đã thay đổi. Các lệnh thường sử dụng:

- **leave**: thu dọn stack và khôi phục **%ebp** của hàm mẹ
- **ret**: pop (lấy) địa chỉ trả về từ stack và nhảy đến đó.

C. BUFFER BOMB LAB

C.1 Chương trình bufbomb

bufbomb là file thực thi dạng command line có lỗ hổng buffer overflow. Chương trình này chạy dưới dạng command line và sẽ 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.
- 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 int getbuf(){
4     char rd_array[RSIZE];    // array with random size
5     char buf[NORMAL_BUFFER_SIZE];
6     Gets(buf);
7     return 1;
8 }
```

Lab 6 – Buffer Bomb

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 kích thước 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, do đó dữ liệu nhập vào có trường hợp sẽ vượt 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:

```
ubuntu@ubuntu:~/LTHT/Lab 5$ ./bufbomb -u test
Userid: test
Cookie: 0x6c64ed92
Type string:Hello world!
Dud: getbuf returned 0x1
Better luck next time
```

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

```
ubuntu@ubuntu:~/LTHT/Lab 5$ ./bufbomb -u test
Userid: test
Cookie: 0x6c64ed92
Type string:It is easier to love this class when you are a TA.
Ouch!: You caused a segmentation fault!
Better luck next time
```

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. **Nhiệm vụ của sinh viên là truyền vào cho chương trình bufbomb (hay cho getbuf) các chuỗi có độ dài và nội dung phù hợp để nó làm một số công việc thú vị. Ta gọi đó là những chuỗi “exploit” – khai thác.**

Lưu ý: mỗi nhóm sinh viên sẽ có riêng 1 phiên bản file bufbomb

C.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>
```

```
ubuntu@ubuntu:~/LTHT/Lab 5$ ./makecookie testuser
0x20ef35a5
```

Lab 6 – Buffer Bomb

- **hex2raw**

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:

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

Hoặc

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

C.3 Một số lưu ý

- Các lưu ý khi tạo các byte của chuỗi exploit – chuỗi input cho bufbomb:
 - 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.
 - 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.
 - Cần để ý đến byte ordering trong Linux là Little Endian khi cần truyền cho hex2raw các giá trị lớn hơn 1 byte. Ví dụ để truyền 1 word 4 bytes **0xDEADBEEF**, cần truyền **EF BE AD DE** (đổi vị trí các byte) cho **hex2raw**.
- Khi thực thi các file **bufbomb**, **hex2raw** hay **makecookie**, nếu gặp lỗi về **Permission denied**, sinh viên cần cấp quyền thực thi các file.

```
ubuntu@ubuntu:~/LTHT/Lab5$ ./bufbomb -u testuser
bash: ./bufbomb: Permission denied
ubuntu@ubuntu:~/LTHT/Lab5$ chmod +x bufbomb
ubuntu@ubuntu:~/LTHT/Lab5$ ./bufbomb -u testuser
Userid: testuser
Cookie: 0x20ef35a5
Type string:hello
Dud: getbuf returned 0x1
Better luck next time
ubuntu@ubuntu:~/LTHT/Lab5$
```

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

```
ubuntu@ubuntu:~/LTHT/Lab 5$ ./hex2raw < smoke.txt | ./bufbomb -u testuser
Userid: testuser
Cookie: 0x20ef35a5
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
ubuntu@ubuntu:~/LTHT/Lab 5$
```

D. Các bước khai thác file bufbomb

Bài lab gồm 4 cấp độ từ 0 – 3 với mức độ từ dễ đến khó tăng dần, tập trung khai thác lỗ hổng buffer overflow có trong **bufbomb** ở hàm **Gets**. Sinh viên thực hiện các bước sau:

Bước 1. Chọn userid và tạo cookie tương ứng

Sinh viên sử dụng một userid trong bài thực hành và khi chạy file **bufbomb** luôn luôn phải truyền vào tham số **-u <userid>**.

Bắt buộc: userid tạo từ MSSV của các thành viên trong nhóm (sắp xếp theo thứ tự tăng dần)

- Với nhóm 3 thành viên: ghép từ 03 số cuối của MSSV. VD: 21520260, 21520143, 21520221, ta có userid là 260143221.
- Với nhóm 2 thành viên: ghép từ 04 số cuối của MSSV bằng dấu -. VD: 21520260 và 21520143, ta có userid 0260-0143.
- Với nhóm 1 thành viên: dùng MSSV làm userid.

Xem cookie tương ứng với userid bằng cách chạy chương trình **makecookie**, ghi nhớ giá trị này để sử dụng về sau. Giá trị cookie này là cơ sở để đánh giá của một số level.

Bước 2. Phân tích file bufbomb và Xác định chuỗi exploit cho từng level

Bước 2.1. Xác định độ dài chuỗi exploit, phụ thuộc vào:

- Độ dài buffer được cấp phát: chuỗi exploit ít nhất phải có độ dài lớn hơn không gian dành cho buffer để làm tràn được buffer.
- Khoảng cách giữa ô nhớ cần ghi đè so với buffer trong stack: ví dụ ghi đè để thay đổi địa chỉ trả về, giá trị biến,...

Bước 2.2. Xác định nội dung chuỗi exploit

Chuỗi exploit đã xác định được độ dài ở bước trên sẽ được điền nội dung với những giá trị byte phù hợp để thực hiện đúng ý định, có 2 mức độ:

- Thay đổi giá trị vùng nhớ lân cận (level 0, 1).
- Truyền vào và thực hiện một số câu lệnh nhất định (level 2, 3).

Bước 3. Thực hiện truyền chuỗi exploit vào bufbomb

Sinh viên viết các chuỗi exploit dưới dạng các byte và sử dụng **hex2raw** để truyền cho **bufbomb**. Giả sử chuỗi exploit dưới dạng các cặp số hexan cách nhau bằng khoảng trắng trong file **exploit.txt** như bên dưới.

00	01	02	03
00	00	00	00
00	01	02	03
00	00	00	00
00	01	02	03
00	00	00	00

Sinh viên có thể truyền chuỗi raw cho **bufbomb** bằng cách lệnh sau:

```
$ ./hex2raw < exploit.txt | ./bufbomb -u testuser
```

E. CÁC LEVEL CƠ BẢN CỦA BUFFER BOMB

Phần này gồm các level khai thác file **bufbomb** cơ bản 0 – 1: truyền các chuỗi exploit để **ghi đè giá trị của một số ô nhớ gần vị trí của chuỗi buffer trong stack.**

E.1 Level 0

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     val = getbuf();
7
8     /* Check for corrupted stack */
9     if (local != uniqueval()) {
10         printf("Sabotaged!: the stack has been corrupted\n");
11     }
12     else if (val == cookie) {
13         printf("Boom!: getbuf returned 0x%x\n", val);
14         validate(3);
15     } else {
16         printf("Dud: getbuf returned 0x%x\n", val);
17     }
18 }
```

Hàm **test()** đóng vai trò caller, **getbuf()** là callee. Theo đúng logic của chương trình, khi **getbuf** thực thi xong, chương trình quay lại thực thi tiếp các lệnh tiếp theo hàm **test**. Tuy nhiên, nếu khai thác lỗ hổng buffer overflow trong **getbuf()** có thể thay đổi luồng thực thi này.

Cụ thể, trong file **bufbomb** cũng có một hàm **smoke** có code C như bên dưới. Hàm này vốn không được gọi trong quá trình thực thi của **bufbomb**. Sinh viên cần chuyển luồng chương trình đến thực thi đoạn code này.

```

1 void smoke()
2 {
3     printf("Smoke!: You called smoke()\n");
4     validate(0);
5     exit(0);
6 }
```

Yêu cầu: khai thác lỗ hổng buffer overflow trong bufbomb tại hàm **getbuf**, sao cho sau khi **getbuf** thực thi xong, chương trình sẽ thực thi đoạn code của hàm **smoke** thay vì quay về thực thi tiếp hàm mẹ là **test**.

Lưu ý: Trong code của hàm **smoke()** có gọi **exit()** để thoát chương trình và không có các hoạt động truy xuất stack frame cũ. Do đó ở level này dù %ebp cũ đang lưu trong stack của **getbuf** có bị ghi đè thì vẫn không xảy ra lỗi.

Gợi ý cách giải:

Xác định mục tiêu: Khi thực thi xong **getbuf**, **bufbomb** dựa vào **địa chỉ trả về** đã lưu trong stack khi gọi hàm này để biết được cần thực thi tiếp câu lệnh nằm ở đâu. Trong trường hợp bình thường, **địa chỉ trả về** trong stack của **getbuf** là địa chỉ của 1 dòng lệnh ở hàm **test**. Địa chỉ này quyết định luồng chạy của chương trình. Do đó, nếu có thể thay đổi địa chỉ trả về trong stack của **getbuf** thành 1 giá trị nào đó theo mong muốn, ta có thể điều khiển chương trình thực thi những đoạn code nhất định.

Nhiệm vụ của sinh viên là nhập chuỗi input sao cho sau khi **getbuf** nhận và lưu chuỗi, **địa chỉ trả về** trong stack của nó cũng bị thay đổi thành **địa chỉ** của hàm **smoke**.

Phân hướng dẫn sử dụng 1 phiên bản bất kỳ để phân tích.

Bước 1: Phân tích file bufbomb và Xác định độ dài input

Mở file **bufbomb** với các disassembler để quan sát mã assembly của nó. Hướng dẫn này sử dụng IDA Pro. Đối tượng cần xem xét là **getbuf**, có mã assembly ở địa chỉ **0x08824158**:

Phân tích mã assembly của **getbuf** ta được: Khi thực hiện lệnh **call getbuf**, **địa chỉ trả về** đã được đẩy vào stack, sau đó chương trình chuyển đến thực thi code của **getbuf**:

- (1) 2 dòng code đầu của **getbuf** lưu lại **%ebp** của hàm mẹ (**test**) và gán giá trị mới cho **%ebp** để trỏ đến stack frame mới của nó.
- (2) Tạo 1 không gian trong stack frame bằng cách trừ **%esp** xuống **0x38 = 56 bytes** và **0xC = 12 bytes**, vậy tổng cộng **68 bytes**.
- (3) Truyền tham số cần thiết để gọi **Gets**. Ta có **Gets** chỉ nhận 1 tham số đầu vào là vị trí lưu chuỗi. Mặt khác, trước khi gọi hàm thì địa chỉ ở vị trí **%ebp + var_34**, tức là vị trí **%ebp - 0x34 = %ebp - 52** (**ebp** trong stack của **getbuf**) được đưa vào stack, ta có thể kết luận vị trí **%ebp - 52** này chính là vị trí lưu chuỗi nhập vào.

Yêu cầu E1.1. Sinh viên vẽ stack của hàm **getbuf()** với mô tả như trên để xác định vị trí của chuỗi **buf** sẽ lưu chuỗi input?

Cần thể hiện rõ trong stack các vị trí: return address của **getbuf**, vị trí của **buf**

Mục tiêu là sẽ ghi đè **địa chỉ trả về** trong stack của **getbuf**. Trong stack vẽ được ở E1.1, quan sát khoảng cách giữa vị trí lưu chuỗi **buf** và vị trí cần ghi đè (**địa chỉ trả về**).

Lab 6 – Buffer Bomb

Yêu cầu E1.2. Xác định các đặc điểm sau của chuỗi exploit nhằm ghi đè lên địa chỉ trả về của hàm **getbuf**:

- Chuỗi exploit cần có **kích thước bao nhiêu bytes** để ghi đè được địa chỉ trả về?
- **4 bytes ghi đè** lên 4 bytes địa chỉ trả về sẽ **nằm ở vị trí nào** trong chuỗi exploit?

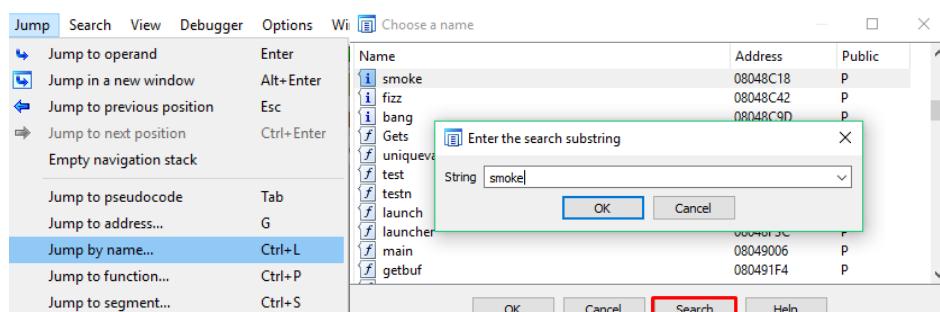
Gợi ý: Chuỗi input khi nhập vào sẽ được ghi vào stack từ địa chỉ thấp đến địa chỉ cao.

Bước 2: Xác định nội dung input

Ta cần xác định **địa chỉ trả về** sẽ được ghi đè 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** trong stack của **getbuf** thành địa chỉ của hàm **smoke**.

Yêu cầu E1.3. Xác định địa chỉ của hàm **smoke** để làm 4 bytes ghi đè lên địa chỉ trả về.

Do trong hoạt động chính **bufbomb** không gọi **smoke** nên IDA Pro không liệt kê **smoke** trong **Functions window**. Để tìm vị trí của **smoke** trong IDA Pro, trên thanh công cụ chọn **Jump → Jump by name...** và dùng chức năng **Search** với keyword là “smoke”.



Khi đó ta sẽ tìm được một label **smoke**. Địa chỉ của hàm **smoke** sẽ có độ dài 4 bytes.

Yêu cầu E1.4. Xây dựng chuỗi exploit với độ dài và nội dung đã xác định trước đó.

Các bước:

- Tạo chuỗi exploit có độ dài phù hợp đã tìm thấy ở **Yêu cầu E1.1**.
- Thực hiện đưa địa chỉ của hàm **smoke** đã tìm thấy vào vị trí 4 bytes đã xác định sẽ ghi đè lên địa chỉ trả về trong stack của hàm **getbuf**. Cần lưu ý đến byte ordering Little Endian. Ví dụ khi cần truyền chuỗi byte ứng với địa chỉ **0xA0B1C2D3**, ta cần viết các byte lần lượt là **D3 C2 B1 A0**.
- Chú ý các byte còn lại có thể tùy ý nhưng phải khác byte **0x0A**.
- Tham khảo cách tạo chuỗi ở **Phần D – Bước 3**.

Bước 3. Truyền chuỗi exploit vào bufbomb

Yêu cầu E1.5. Thực hiện truyền chuỗi exploit cho **bufbomb** và báo cáo kết quả.

Lab 6 – Buffer Bomb

Ví dụ kết quả khai thác thành công:

```
ubuntu@ubuntu:~/LTHT/Lab 5
ubuntu@ubuntu:~/LTHT/Lab 5$ ./hex2raw < smoke.txt | ./bufbomb -u testuser
Userid: testuser
Cookie: 0x20ef35a5
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
ubuntu@ubuntu:~/LTHT/Lab 5$
```

E.2 Level 1

Tương tự như level 0, ở level 1 tiếp tục khai thác lỗ hổng buffer overflow ở getbuf() để chuyển hướng thực thi một hàm khác là **fizz()** (không được gọi khi thực thi **bufbomb**), có code như sau:

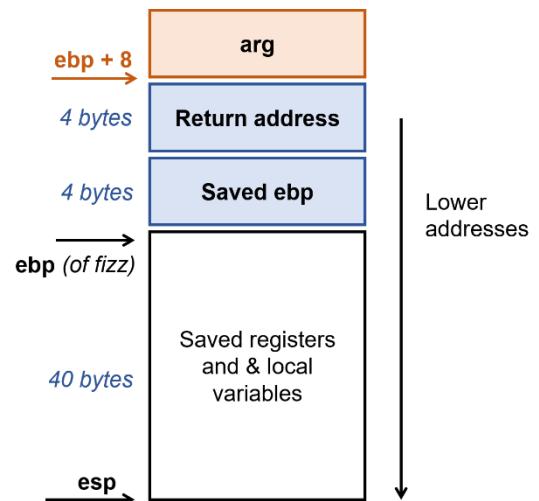
```
1 void fizz(int val)
2 {
3     if (val == cookie) {
4         printf("Fizz!: You called fizz(0x%x)\n", val);
5         validate(1);
6     } else {
7         printf("Misfire: You called fizz(0x%x)\n", val);
8         exit(0);
9     }
10 }
```

Tuy nhiên khác với hàm **smoke()**, hàm **fizz()** này có yêu cầu truyền vào một tham số là **val**. Quan sát mã nguồn ở trên, có thể thấy tham số **val** sẽ được dùng so sánh với **cookie** – giá trị được tạo duy nhất với mỗi userid đã đề cập ở trên. Chỉ khi truyền đúng giá trị cookie làm tham số, level mới được giải thành công.

Yêu cầu: Khai thác lỗ hổng buffer overflow để **bufbomb** thực thi đoạn code của **fizz** thay vì trả về hàm **test**. Đồng thời, truyền giá trị cookie của sinh viên làm tham số của **fizz**. *Hàm fizz cũng gọi exit() để thoát chương trình, nên cũng không cần quan tâm đến %ebp bị ghi đè.*

Gợi ý:

- Do cùng khai thác hàm **getbuf** nên độ dài của buffer được cấp phát trong stack vẫn giống level trước.
- Tham số của 1 hàm nằm ở một số vị trí nhất định như (%ebp + 8), (%ebp + 12)...
Ta có hình bên là stack của **fizz** trong **trường hợp gọi thông thường thông qua lệnh call**. Chuỗi input cần ghi đè giá trị **arg** ở vị trí (%ebp + 8) thành giá trị cookie.



- ➔ Vị trí **arg** trong stack của **fizz** sẽ nằm ở đâu so với vị trí **buf** trong stack của **getbuf**?
- Thứ tự cấp phát/thu hồi stack các hàm nếu chuyển hướng thành công sang **fizz**: (1) cấp phát stack cho test, (2) cấp phát stack cho **getbuf()**, (3) thu hồi stack của **getbuf()**, (4) cấp phát stack cho **fizz()**.
- Lưu ý trong trường hợp bị khai thác, **bufbomb** không thực sự gọi hàm **fizz** qua lệnh **call** mà chỉ nhảy đến thực thi code của hàm. Vậy stack của **fizz** khi đó có gì khác biệt so với hình bên không?

F. CÁC LEVEL NÂNG CAO CỦA BUFFER BOMB LAB

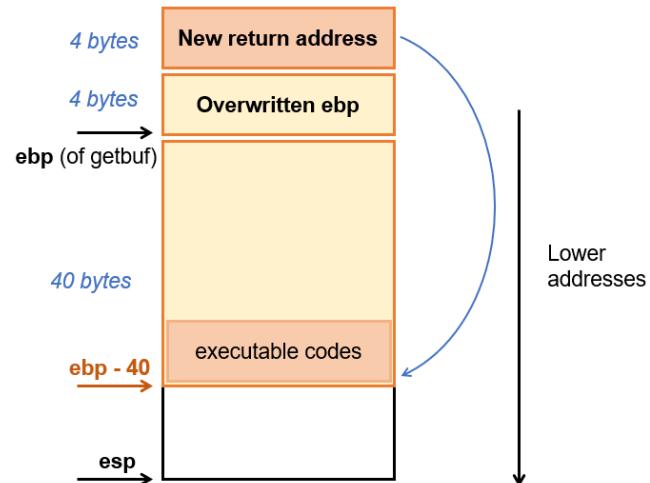
F.1 Hướng dẫn sử dụng buffer overflow để chèn và thực thi mã code

F.1.1 Ý tưởng tấn công

Một dạng phức tạp hơn của tấn công buffer overflow là chuỗi exploit là một chuỗi chứa các byte code có thể thực thi được, được chèn vào stack để thực thi. Khi đó, **chuỗi exploit sẽ ghi đè để thay đổi địa chỉ trả về để trở về vị trí của những byte code này trên stack**. Khi hàm bị khai thác (trong trường hợp này là **getbuf**) thực thi câu lệnh **ret**, chương trình sẽ nhảy đến vị trí lưu byte code thực thi đã chèn vào để thực thi thay vì quay về hàm trước.

Giả sử có stack của hàm **getbuf** cần khai thác như hình. Theo đó, ta cần nhập 1 chuỗi exploit màu cam, trong đó:

- Chuỗi exploit có chứa 1 số byte code thực thi của một số lệnh (phần **executable codes** màu cam đậm trong hình), có độ dài tùy thuộc vào các lệnh mà chúng đại diện.
- Chuỗi đủ dài để ghi đè địa chỉ trả về thành **địa chỉ mới** phù hợp (phần màu cam đậm phía trên) để thực hiện ý đồ dấu mũi tên. Ở đây, **địa chỉ trả về mới** chính là địa chỉ của chuỗi đã nhập, đang chứa executable code cần thực thi.
- Các byte còn lại (màu cam nhạt) có thể tùy ý (khác 0x0A) hoặc tuỳ yêu cầu.



Nếu có thể ghi đè như trên khi tấn công buffer overflow, hàm **getbuf** khi kết thúc sẽ lấy **địa chỉ trả về mới** ra khỏi stack và nhảy đến đó thực thi, tức là nhảy đến thực thi các executable code đang nằm trong chuỗi buf.

F.1.2 Ví dụ khai thác buffer overflow để truyền code thực thi

Theo cách tấn công buffer overflow đã biết, ta cần 1 chuỗi gồm n byte tùy ý + 4 byte để ghi đè lên địa chỉ trả về. Trong n byte đầu, sẽ có 1 số byte chứa các executable code.

- **Tạo mã thực thi**

Tùy vào chức năng mong muốn thực hiện, sinh viên có thể định nghĩa code cần thực thi với các bước sau:

Lab 6 – Buffer Bomb

- Viết code dưới dạng mã assembly trong các file **.s**. Ví dụ ở đây định nghĩa 2 lệnh assembly có chức năng thoát chương trình (system call exit).

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

- Sử dụng 1 số công cụ để tạo các byte code tương ứng (khoanh đỏ).

```
$ as --32 <file .s đầu vào> -o <file .o đầu ra>
$ objdump -d <file .o>
```

```
ubuntu@ubuntu:~/LTHT/Lab6$ as --32 exit.s -o exit.o
ubuntu@ubuntu:~/LTHT/Lab6$ objdump -d exit.o

exit.o:      file format elf32-i386

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

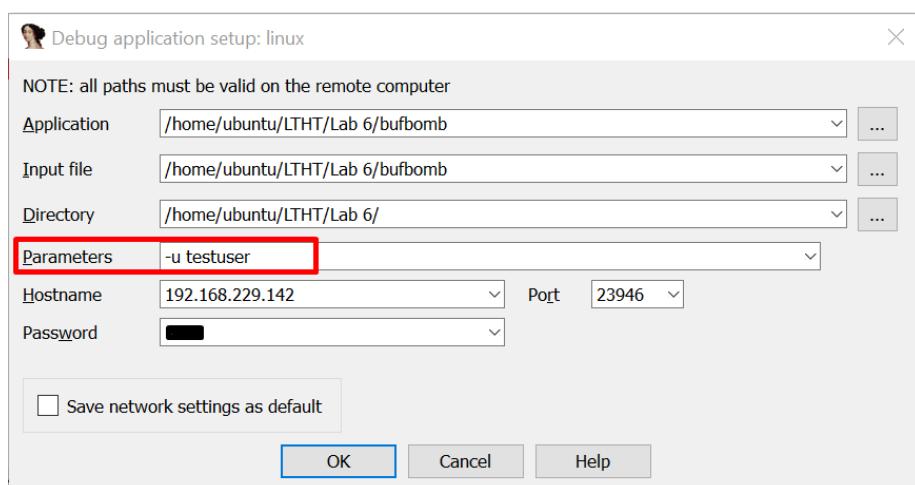
7 bytes khoanh đỏ chính là các executable code sẽ đặt trong chuỗi để truyền vào chương trình. Số byte có thể thay đổi tùy theo các lệnh được viết.

- **Xác định địa chỉ trả về mới**

Địa chỉ trả về mới nên là địa chỉ bắt đầu lưu chuỗi buf trong stack. Địa chỉ này chỉ có giá trị cụ thể trong lúc chương trình chạy, nên cần debug chương trình để xem vị trí chính xác của **buf**.

- + **Cách 1:** Thực hiện remote debug với IDA Pro (bản 32 bit)

Thiết lập cấu hình remote debugger (xem lại Lab 5) phù hợp, lưu ý cần truyền tham số **-u <userid>** khi debug.



Ta cần tìm vị trí lưu của chuỗi **buf** trong stack của **getbuf**, đặt breakpoint tại hàm **getbuf** và tiến hành debug.

Lab 6 – Buffer Bomb

```

.text:805D8478          public getbuf
.text:805D8478 getbuf    proc near             ; CODE XREF: test+Etp
.text:805D8478
.text:805D8478 var_28     = byte ptr -28h
.text:805D8478
.text:805D8478 push    ebp
.text:805D8479 mov     ebp, esp
.text:805D847B sub     esp, 28h

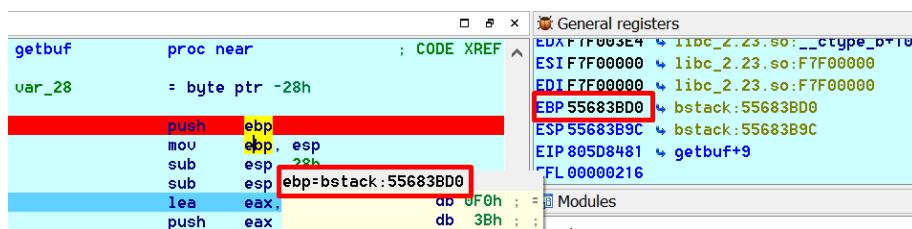
```

```

ubuntu@ubuntu:~/LTHT/Lab 6$ ./linux_server
IDA Linux 32-bit remote debug server(ST) v1.17. Hex-Rays (c) 2004-2014
Listening on port #23946...
=====
[1] Accepting connection from 192.168.229.1...
Userid: testuser
Cookie: 0x20ef35a5

```

Ta cần di chuyển đến dòng lệnh phù hợp và xem giá trị các thanh ghi/ô nhớ khi ở dòng lệnh đó. Ví dụ bên dưới, khi debug đến dòng lệnh màu xanh dương đậm, trỏ trên code assembly hay quan sát ở cửa sổ **General registers** cho ta giá trị của **ebp = 0x55683BD0**.



+ Cách 2: Dùng GDB trên Linux

Sử dụng **gdb** với bufbomb, đặt breakpoint tại hàm **getbuf** và chạy chương trình (lưu ý vẫn cần truyền tham số userid như khi thực thi bình thường).

```
$ gdb bufbomb
(gdb) break getbuf
(gdb) run -u <userid>
```

```

ubuntu@ubuntu:~/LTHT/Lab 6$ gdb bufbomb
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bufbomb... (no debugging symbols found)... done.
(gdb) break getbuf
Breakpoint 1 at 0x805d847e
(gdb) run -u testuser
Starting program: /home/ubuntu/LTHT/Lab 6/bufbomb -u testuser
Userid: testuser
Cookie: 0x20ef35a5

```

Xem mã assembly của getbuf và giá trị của ebp (khi đứng ở lệnh có ký hiệu =>)

```
(gdb) disassemble getbuf
(gdb) info registers ebp
```

Lab 6 – Buffer Bomb

```
(gdb) disassemble getbuf
Dump of assembler code for function getbuf:
0x805d8478 <+0>: push %ebp
0x805d8479 <+1>: mov %esp,%ebp
0x805d847b <+3>: sub $0x28,%esp
=> 0x805d847e <+6>: sub $0xc,%esp
0x805d8481 <+9>: lea -0x28(%ebp),%eax
0x805d8484 <+12>: push %eax
0x805d8485 <+13>: call 0x805d7f28 <Gets>
0x805d848a <+18>: add $0x10,%esp
0x805d848d <+21>: mov $0x1,%eax
0x805d8492 <+26>: leave
0x805d8493 <+27>: ret
End of assembler dump.
(gdb) info registers ebp
ebp          0x55683bd0      0x55683bd0 <_reserved+1039312>
(gdb)
```

Cả 2 cách debug cho giá trị **ebp = 0x55683BD0**. **Sử dụng giá trị ebp này để tính tiếp vị trí cụ thể của chuỗi buf? Địa chỉ cụ thể của chuỗi buf sẽ là địa chỉ trả về mới trong trường hợp tấn công buffer overflow để truyền và thực thi code định nghĩa bên ngoài.**

Ví dụ nếu vị trí chuỗi buf là **ebp – 40**, với kết quả trên ta sẽ có địa chỉ trả về mới là **0x55683BD0 – 0x28 (40) = 0x55683BA8**

- Dựng chuỗi exploit chứa executable code và địa chỉ trả về mới**

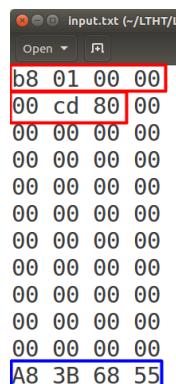
Để chương trình thực thi được các executable code trong chuỗi exploit, nên đảm bảo:

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

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

Các vị trí này có thể tuỳ chỉnh, tuy nhiên **luôn đảm bảo** rằng: **địa chỉ trả về mới trodung vào vị trí bắt đầu của những byte code đầu tiên** 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.

Cụ thể, phần hướng dẫn này sẽ đặt 7 byte executable code đã tạo vào 7 byte đầu trong n byte đầu tiên của chuỗi exploit. Đồng thời, 4 byte ở vị trí tương ứng với địa chỉ trả về là địa chỉ vừa tìm được qua debug.



Lab 6 – Buffer Bomb

Trước tiên quan sát trường hợp bình thường, sau khi nhập chuỗi, chương trình đều in ra 1 số thông báo dù có lỗi hay không.

```
ubuntu@ubuntu:~/LTHT/Lab 6/demo$ ./bufbomb -u testuser
Userid: testuser
Cookie: 0x20ef35a5
Type string:hello
Dud: getbuf returned 0x1
Better luck next time
ubuntu@ubuntu:~/LTHT/Lab 6/demo$ ./bufbomb -u testuser
Userid: testuser
Cookie: 0x20ef35a5
Type string:12345678901234567890123456789012345678901234567890654321432
Ouch!: You caused a segmentation fault!
Better luck next time
ubuntu@ubuntu:~/LTHT/Lab 6/demo$
```

Kết quả thực thi với chuỗi exploit chứa executable code: chương trình nhận chuỗi sau đó kết thúc ngay lập tức. Điều này chứng tỏ 2 dòng lệnh assembly định nghĩa system exit đã thực thi thành công.

```
ubuntu@ubuntu:~/LTHT/Lab 6/demo$ ./hex2raw < input.txt | ./bufbomb -u testuser
Userid: testuser
Cookie: 0x20ef35a5
ubuntu@ubuntu:~/LTHT/Lab 6/demo$
```

Sinh viên sử dụng phương pháp tương tự để giải 2 level bên dưới.

F.2 Level 2

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 }
```

Level này yêu cầu sẽ gọi thực thi hàm **bang**. Tuy nhiên, 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 level này mới coi như thành công thành công. Vì giá trị **global_value** này không nằm trên stack (không nằm gần **buf**), nên không thể áp dụng phương pháp ghi đè những vùng nhớ lân cận ở lab trước, thay vào đó cần thực thi một số lệnh gán giá trị. Do đó, ta cần định nghĩa và truyền vào chương trình những executable code để thực thi nhằm đảm bảo trước khi gọi **bang**, ta thay đổi được giá trị của **global_value**.

Yêu cầu: Khai thác lỗ hổng buffer overflow để truyền vào **bufbomb** một chuỗi exploit có chứa executable code gồm 1 số chức năng:

Lab 6 – Buffer Bomb

- Thay đổi giá trị của **global_value** thành giá trị cookie: **global_value** là 1 biến toàn cục, nằm ở 1 ô nhớ có địa chỉ cố định, có thể xem code assembly của hàm **bang** có tham chiếu đến **global_value** để biết địa chỉ này.
- Đi đến thực thi được hàm **bang**.

Gợi ý: Một cách để đến thực thi 1 hàm:

```
push $<địa chỉ của hàm cần thực thi> # push địa chỉ vào stack
ret                                # pop địa chỉ vừa push ra và nhảy đến đó
```

F.3 Level 3

Các tấn công ở những level trước chỉ khiến cho chương trình nhảy đến đoạn code của những hàm khác có hàm 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, thực thi một số hoạt động nhất định vẫn có thể quay về hàm mẹ ban đầu (hàm **test**). 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 tùy 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 test sau khi bị khai thác. 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 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 số công việc nào đó, cần khôi phục lại bất kỳ thay đổi nào đã gây ra với thông tin của hàm mẹ lưu trên stack hàm con và trở về đoạn code trong 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, giá trị trả về sẽ là cookie tương ứng với userid cho hàm **test**, thay vì trả về 1 và sẽ quay về hàm test như bình thường.

Executable code cần có các lệnh thực hiện các công việc:

- Gán cookie vào vị trí lưu 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 câu lệnh cần thực thi tiếp theo của **test**).
- Thực thi câu lệnh **ret** để trở về **test**.

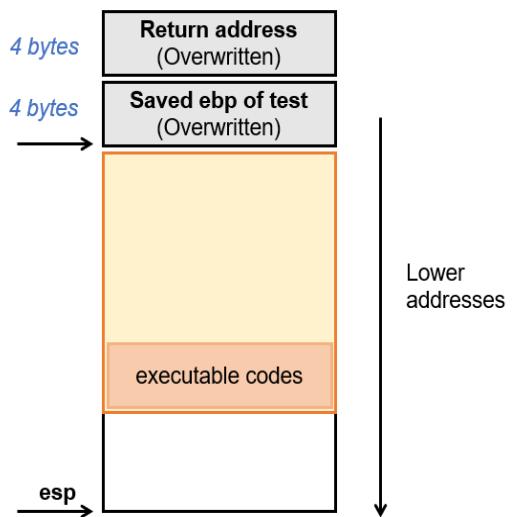
Gợi ý:

- Giá trị trả về của hàm thường lưu trong %eax.

Lab 6 – Buffer Bomb

- 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** cũng bị ghi đè. Làm cách nào để khôi phục lại được giá trị ban đầu của nó trước khi quay về **test**?
- Có thể khôi phục giá trị thanh ghi bằng 1 trong 2 cách:
 - Ghi đè trực tiếp giá trị đúng lên ô nhớ ở vị trí tương ứng trong chuỗi exploit.
 - Khôi phục trong mã thực thi bằng lệnh:

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



Yêu cầu thêm (cộng điểm): Có 1 cách để khôi phục giá trị %ebp cũ của hàm test bằng code thực thi nhưng không cần debug để tìm giá trị chính xác. Sinh viên thử đề xuất phương pháp và thử thực hiện tấn công?

G. YÊU CẦU & ĐÁNH GIÁ

Sinh viên thực hành và nộp bài **theo nhóm** theo thời gian quy định. Sinh viên nộp file **.pdf** chứa hình ảnh chụp màn hình kết quả khai thác file kèm theo chuỗi exploit, trong đó:

- Bắt buộc: Level 0 – 1
- Không bắt buộc: Level 2 - 3

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

File báo cáo .pdf được đặt tên theo quy tắc sau:

Lab6-NhomX_MSSV1-MSSV2-MSSV3.pdf

Ví dụ: *Lab6-Nhom2_2252xxxx-2252zzzz-2252yyyy.pdf*

H. THAM KHẢO

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