6.1

Lab

Buffer Overflow Attack
Simple Buffer

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.

A.2 Môi trường

- Môi trường: IDA Pro trên Windows kết hợp với máy ảo Linux để thực thi file.
- Các file source của bài lab:
  - **simple-buffer**: file thực thi Linux 32-bit chứa lỗ hổng buffer overflow cần khai thác.
  - **hex2raw**: một số file hỗ trợ (32 bit).

A.3 Liên quan

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

- Có kiến thức về cách phân vùng bộ nhớ, gọi hàm trong hệ thống.
- 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. Khai thác lỗ hổng buffer overflow

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.

B.1 Chương trình simple-buffer

simple-buffer là file thực thi 32 bit có lỗ hổng buffer overflow. Chương trình này chạy dưới dạng command line và yêu cầu nhập một chuỗi. Khi chạy, **simple-buffer** đi kèm nhiều option như sau:

<MSSV> MSSV của sinh viên.

<ten> Tên sinh viên, lưu ý chỉ dùng tên, không dùng phần lót và họ. Tối đa 7 ký tự.

Lỗ hổng buffer overflow tồn tại trong 1 hàm **smash_my_buffer** được định nghĩa như sau:

```
1   char student_name[8];    # global variable saving the given name of student
2   int student_id;          # global variable saving the int value of given MSSV
3   void smash_my_buffer()
4   {
5         unsigned int var = 0x12ABCDEF;       #default value
6         ...
7         char my_name[8] = "student";         #default value
8         ...
9         int another_var = 0x10;              #default value
10        ...
11        char buf[28];
12        gets(buf);
13        if (strcmp(my_name,"student") || var != 0x12ABCDEF || another_var !=
0x10){
14              printf("You changed my local variables.\n");
15              if (strcmp(my_name, student_name) == 0)
16                    printf("[+] Level 1: DONE. ...");
17              if (another_var == 0x322D)
18                    printf("[+] Level 2: DONE. ... ");
19              if (var == student_id)
20                    printf("[+] Level 3: DONE. ...");
21        }
22        else
23              printf("Try again to change my local variables\n");
24  }
```

Hàm **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 12 ký tự.

Tuy nhiên, 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 nhận hết tất cả ký tự người dùng nhập (kết thúc bằng enter) và sao chép toàn bộ vào vị trí đích đã chỉ định, do đó dữ liệu nhập vào có trường hợp sẽ vượt khỏi vùng nhớ được cấp trước đó. Do vậy, chuỗi **buf** có thể được tận dụng để khai thác chương trình tùy theo độ dài của nó.

Trong bài thực hành này, đối tượng cần khai thác chủ yếu là **smash_my_buffer** và stack của nó. **Nhiệm vụ của sinh viên là truyền vào cho chương trình simple-buffer (hay cho smash_my_buffer) các chuỗi có *độ dài và nội dung* phù hợp. Ta gọi đó là những chuỗi "exploit" – khai thác.**

B.2 Một số file hỗ trợ

Bên cạnh file chính là **simple-buffer,** có một số file được cung cấp thêm nhằm hỗ trợ quá trình thực hiện bài thực hành:

- **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 để có thể truyền làm input cho file **simple-buffer**. **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
```

B.3 Một số lưu ý

- Các lưu ý khi tạo các byte của chuỗi exploit – chuỗi input cho simple-buffer:
  - Chuỗi exploit ***không được*** chứa byte hexan **0x0A** 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**.

C. Các bước khai thác file simple-buffer

Bài lab gồm 3 level với mức độ từ dễ đến khó tăng dần, tập trung khai thác lỗ hổng buffer overflow trong **simple-buffer** ở hàm **gets**. Sinh viên thực hiện theo các bước sau:

Bước 1. Phân tích file simple-buffer và Xác định chuỗi exploit cho từng level

Có 2 bước cần thực hiện để xác định chuỗi exploit:

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 giá trị biến cục bộ,…

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 mục đích, ví dụ thay đổi giá trị vùng nhớ (có địa chỉ cao hơn) nằm gần vị trí lưu chuỗi.

Bước 2. Thực hiện truyền chuỗi exploit vào simple-buffer

Sinh viên viết các chuỗi exploit dưới dạng các byte và sử dụng **hex2raw** để truyền cho **simple-buffer**. Giả sử chuỗi exploit dưới dạng 24 cặp số hexan cách nhau bằng khoảng trắng trong file **exploit.txt** như bên dưới, kết quả sẽ có 24 byte được truyền.

```
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 **simple-buffer** bằng cách lệnh sau:

```
$ ./hex2raw < exploit.txt | ./simple-buffer 2252xxxx name
```

D. NỘI DUNG THỰC HÀNH

Nội dung thực hành tập trung khai thác file **simple-buffer** với 3 level: truyền các chuỗi exploit để ***ghi đè giá trị của một số ô nhớ gần vị trí của chuỗi buffer trong stack***.

D.1 Level 1

Hàm **smash_my_buffer()** có 1 biến cục bộ là chuỗi **my_name**, mặc định là **"student"**

```
1  void smash_my_buffer()
2  {
3        ...
4        char my_name[8] = "student";        #default value
5        ...
6        char buf[28];
7        gets(buf);
```

```
8         if (strcmp(my_name,"student") || var != 0x12ABCDEF || another_var != 0x10){
9               printf("You changed my local variables.\n");
10              if (strcmp(my_name, student_name) == 0)
11                    printf("[+] Level 1: DONE. ...");
12        ...
13  }
```

Yêu cầu: khai thác lỗ hổng buffer overflow trong simple-buffer tại hàm **smash_my_buffer**, sao cho sau khi nhập xong chuỗi, biến **my_name** được đổi thành tên của SV đã nhập khi thực thi file simple-buffer.

Gợi ý cách giải:

Xác định mục tiêu: Chuỗi **buf** và **my_name** đều là biến cục bộ nên sẽ nằm trong stack frame của hàm **smash_my_buffer** (thấp hơn vị trí trỏ đến bởi %ebp của hàm). Nếu **my_name** nằm trên vùng nhớ cao hơn **buf** thì có thể dùng **buf** để ghi đè **my_name**.

Nhiệm vụ của sinh viên là kiểm tra và nhập chuỗi input sao cho sau khi lưu chuỗi **buf**, biến **my_name** sẽ bị ghi đè thành giá trị là chuỗi tên của sinh viên.

Bước 1: Phân tích file simple-buffer và Xác định độ dài input

Mở file **simple-buffer** với các disassembler để quan sát mã assembly của nó. Hướng dẫn này sử dụng IDA Pro. Đối tượng cần xem xét là **smash_my_buffer**, có mã assembly ở địa chỉ **0x0804864A**:

```
.text:0804864A smash_my_buffer proc near               ; CODE XREF: main+51↓p
.text:0804864A
.text:0804864A s               = byte ptr -48h
.text:0804864A s1              = byte ptr -2Ch
.text:0804864A var_28          = dword ptr -28h
.text:0804864A var_24          = dword ptr -24h
.text:0804864A var_20          = dword ptr -20h
.text:0804864A var_1C          = dword ptr -1Ch
.text:0804864A var_18          = dword ptr -18h
.text:0804864A var_14          = dword ptr -14h
.text:0804864A var_10          = dword ptr -10h
.text:0804864A var_C           = dword ptr -0Ch
.text:0804864A var_8           = dword ptr -8
.text:0804864A var_4           = dword ptr -4
.text:0804864A
.text:0804864A                 push    ebp
.text:0804864B                 mov     ebp, esp                                  1
.text:0804864D                 sub     esp, 48h
.text:08048650                 mov     [ebp+var_4], 0
.text:08048657                 mov     [ebp+var_8], 12ABCDEFh
.text:0804865E                 mov     [ebp+var_C], 0
.text:08048665                 mov     [ebp+var_10], 0
.text:0804866C                 mov     dword ptr [ebp+s1], 64757473h
.text:08048673                 mov     [ebp+var_28], 746E65h
.text:0804867A                 mov     [ebp+var_18], 0                           2
.text:08048681                 mov     [ebp+var_14], 0
.text:08048688                 mov     [ebp+var_1C], 10h
.text:0804868F                 mov     [ebp+var_20], 0
.text:08048696                 mov     [ebp+var_24], 0
.text:0804869D                 lea     eax, [ebp+s]
.text:080486A0                 push    eax             ; s                       3
.text:080486A1                 call    _gets
.text:080486A6                 add     esp, 4
```

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

(1) 3 dòng code đầu của **smash_my_buffer** lưu lại **%ebp** của hàm mẹ (**main**) và gán giá trị mới cho **%ebp** để trỏ đến stack frame mới của nó.
Tạo 1 không gian trong stack frame bằng cách trừ **%esp** xuống **0x48 = 72 bytes**.

(2) Gán 1 số giá trị cho các biến cục bộ. Dựa vào code C ta có thể dự đoán dòng mov dữ liệu 0x12ABCDEF vào vị trí **%ebp + var_8 = %ebp - 0x8 = %ebp – 8** là lệnh gán giá trị cho biến **var**. Như vậy biến **var** nằm ở vị trí **%ebp - 8.** Tương tự có thể suy ra vị trí biến **my_name** và **another_var**.

(3) Truyền tham số cần thiết để gọi **gets**. Ta có **gets** chỉ nhận 1 tham số đầu vào là vị trí lưu chuỗi là chuỗi buf. Mặt khác, trước khi gọi hàm thì địa chỉ ở vị trí **%ebp + s,** tức là vị trí **%ebp - 0x48** = **%ebp – 72** được đưa vào stack, ta có thể kết luận vị trí **%ebp – 72** này chính là vị trí lưu chuỗi nhập vào.

Yêu cầu E1.1. Sinh viên vẽ stack của hàm **smash_my_buffer** với mô tả như trên để xác định vị trí của chuỗi **buf** sẽ lưu chuỗi được nhập?

Cần thể hiện rõ trong stack các vị trí: return address của getbuf, vị trí của buf và các biến cục bộ khác.

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

Yêu cầu E1.2. Xác định **chuỗi exploit** nhằm ghi đè lên biến cục bộ **my_name**:

- Chuỗi exploit cần có **kích thước bao nhiêu bytes** để ghi đè được biến cục bộ trên?
- **Các bytes ghi đè lên my_name nằm ở vị trí nào** chuỗi exploit?

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

Bước 2: Xác định nội dung cần nhập

Ta xác định **giá trị cần ghi đè** lên biến my_name là gì. Do yêu cầu đổi thành tên của sinh viên (giống tên đã nhập khi thực thi), sinh viên cần chuẩn bị chuỗi tên đúng để ghi đè. Ví dụ: chuỗi tên 'name' gồm các ký tự n, a, m, e hoặc mã ASCII tương ứng.

Lưu ý: chuỗi trong hệ thống sẽ kết thúc bằng byte NULL (0x00). Để xác định vị trí chuỗi kết thúc, nên thêm byte 0x00 ở cuối chuỗi.

Yêu cầu E1.3. 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 chuỗi tên sinh viên vào các vị trí bytes đã xác định sẽ ghi đè lên biến cục bộ **my_name**.
- Chú ý các byte còn lại có thể tùy ý nhưng phải khác byte **0x0A**.
- Tham khảo cách tạo chuỗi ở ***Phần C – Bước 2***.

Bước 3. Truyền chuỗi exploit vào simple-buffer

Yêu cầu E1.4. Thực hiện truyền chuỗi exploit cho **simple-buffer** và báo cáo kết quả.

Ví dụ kết quả ghi đè thành công biến my_name:

```
ubuntu@ubuntu: ~/LTHT/Lab 6/demo
ubuntu@ubuntu:~/LTHT/Lab 6/demo$ ./hex2raw < byte_code | ./simple-buffer 22520260 name
You changed my local variables.
[+] Level 1: DONE. You've changed name to name
[-] Level 2: FAILED. Current another_var is 0x10
[-] Level 3: FAILED. Current var is 313249263
--------
Your finish time: Wed Dec 11 22:00:06 2024
ubuntu@ubuntu:~/LTHT/Lab 6/demo$
```

D.2 Level 2

Tương tự như level 1, ở level 2 tiếp tục khai thác lỗ hổng buffer overflow ở smash_my_buffer để thay đổi biến cục bộ **another_var** thành giá trị **0x322D**. Lưu ý: Byte ordering trong Linux. *Chú ý vẫn phải đảm bảo ghi đè thành công Level 1.*

Kết quả:

```
ubuntu@ubuntu: ~/LTHT/Lab 6/demo
ubuntu@ubuntu:~/LTHT/Lab 6/demo$ ./hex2raw < byte_code | ./simple-buffer 22520260 name
You changed my local variables.
[+] Level 1: DONE. You've changed name to name
[+] Level 2: DONE. You've changed another_var to 0x322d
[-] Level 3: FAILED. Current var is 313249024
--------
Your finish time: Mon Dec 16 21:50:49 2024
ubuntu@ubuntu:~/LTHT/Lab 6/demo$
```

D.3 Level 3

Khai thác tiếp hàm **smash_my_buffer** để ghi đè biến **var** thành giá trị số nguyên của MSSV. Lưu ý: Byte ordering trong Linux, *vẫn phải đảm bảo ghi đè thành công Level 1 và 2.*

Kết quả:

```
ubuntu@ubuntu: ~/LTHT/Lab 6/demo
ubuntu@ubuntu:~/LTHT/Lab 6/demo$ ./hex2raw < byte_code | ./simple-buffer 22520260 name
You changed my local variables.
[+] Level 1: DONE. You've changed name to name
[+] Level 2: DONE. You've changed another_var to 0x322d
[+] Level 3: DONE. You've changed var to 22520260
--------
Your finish time: Mon Dec 16 21:51:34 2024
ubuntu@ubuntu:~/LTHT/Lab 6/demo$
```

E. YÊU CẦU & ĐÁNH GIÁ

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

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

[Mã lớp] Lab6-MSSV_Hoten.pdf

Ví dụ: *Lab6-23520yyy_NguyenVanA.pdf*

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