1 Buffer Overflow

Câu 1: Trình bày khái niệm lỗ hổng buffer overflow; chỉ ra ví dụ về lỗ hổng và khai thác lỗ hổng này?

- Khái niệm:

Lỗ hổng tràn bộ đệm (Buffer Overflow) là lỗ hổng trong lập trình, cho phép dữ liệu được ghi vào một buffer có thể tràn ra ngoài buffer đó, ghi đè lên dữ liệu khác và dẫn tới hoạt động bất thường của chương trình.

- Ví dụ:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   void vun(){
5       char cmd[16] = "echo Bye!";
6       char buff[16];
7       read(0, buff, 20);
8       system(cmd);
9   }
10
11  int main (){
12      vun();
13      return 0;
14  }
```

  - Chương trình sử dụng hàm read cho phép đọc vào 20 bytes trong khi biến buff chỉ được khai báo 16 bytes.
  - Tận dụng 4 bytes tràn để gọi shell vơi hàm system.
- Khai thác:

  - Ta có thể sử dụng “sh” để system(“sh”) vào shell.
  - Sau khi ghi đè biến buff với 16 bytes, ta sẽ ghi “sh” vào biến cmd.

Câu 2: Trình bài khái niệm lỗ hổng buffer overflow, trình bày cơ chế khai thác lỗ hổng tràn bộ đệm để ghi đè địa chỉ trả về nhằm chuyển hướng thực thi tới 1 hàm tùy ý?

- Khái niệm:

Lỗ hổng tràn bộ đệm (Buffer Overflow) là lỗ hổng trong lập trình, cho phép dữ liệu được ghi vào một buffer có thể tràn ra ngoài buffer đó, ghi đè lên dữ liệu khác và dẫn tới hoạt động bất thường của chương trình.

- Cơ chế khai thác:

Về bản chất, kẻ tấn công chọn một địa chỉ trong chương trình nơi chứa các đoạn mã đang thực thi và ghi đè địa chỉ mới lên địa chỉ trả về. Địa chỉ ghi lên phụ thuộc chủ ý của kẻ tấn công, nhưng thường hướng việc thực thi tới hai dạng:

- Việc thực thi có thể được chuyển hướng đến vùng mã của chương trình đang chạy hoặc tới một vài đoạn mã trong thư viện chia sẻ có chứa những thành phần hữu dụng như hàm system() trong hệ thống UNIX libc nơi có thể chạy các đoạn lệnh trong shell.
- Việc thực thi có thể chuyển hướng tới một vùng nhớ chứa các dữ liệu mà kẻ tấn công kiểm soát, chẳng hạn như một biến toàn cục, vị trí ngăn xếp, hoặc một vùng nhớ tĩnh. Trong trường hợp này, kẻ tấn công điền thêm vào các vị trí trả về

2 Format String

Câu 1: Trình bày khái niệm lỗ hổng Format String; chỉ ra ví dụ về lỗ hổng và khai thác lỗ hổng này?

- Khái niệm:

Lỗi **format string** là lỗi xuất hiện khi dữ liệu người dùng được được sử dụng làm format string (hoặc được đưa vào format string) trong các hàm thuộc họ printf.

- printf, fprintf, sprintf, snprintf
- vprintf, vfprintf, vsprintf, vsnprintf

- Ví dụ:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int cat_flag(){
5       system("cat flag");
6       while (1<2){}
7   }
8
9   void vun(){
10      char buff[1024];
11      read(0, buff, 1000);
12      printf(buff);
13      exit(0);
14  }
15
16  int main (){
17      vun();
18      return 0;
19  }
```

Hàm main gọi hàm vun(), tại đây ta thấy ngay dòng 12 printf đang không có format string. Nên ta sẽ sử dụng lỗi này để khai thác.

- Khai thác:

- Sử dụng gdb và đặt breakpoint sau hàm prinft để xem và quan sát thấy địa chỉ hàm read, printf đã trỏ tới thư viện, hay địa chỉ 2 hàm này đã được resolve (vì đã được gọi). Còn địa chỉ của 2 hàm system và exit do chưa được gọi nên địa chỉ vẫn trỏ tới plt, ta cũng quan sát thấy địa chỉ cat_flag và got.exit chỉ khác nhau 2 byte cuối.
- Như vậy ta có thể ghi đè 2 byte cuối của hàm exit trở thành cat_flag trước khi chương trình gọi exit().

Câu 2: Trình bài khái niệm lỗ hổng chuỗi định dạng (format string); trình bày cơ chế khai thác lỗ hổng chuỗi định dạng để ghi một giá trị tùy ý vào một địa chỉ tùy ý

- Khái niệm:

Lỗi **format string** là lỗi xuất hiện khi dữ liệu người dùng được được sử dụng làm format string (hoặc được đưa vào format string) trong các hàm thuộc họ printf.

- printf, fprintf, sprintf, snprintf
- vprintf, vfprintf, vsprintf, vsnprintf

- Cơ chế khai thác:

Để hiểu cơ chế, trước hết ta xét ví dụ:

```
int x=0, y=0;
printf ("12345%n", &x);
printf ("%50%n", &y);
```

- Lệnh printf thứ nhất sẽ in ra 5 ký tự "12345" và lưu vào biến x (lưu vào ô nhớ có địa chỉ là &x) số ký tự đã in ra. Còn lệnh printf thứ hai in ra số 0 và ghi vào y giá trị bằng 50.
- Nếu thay "%s" bằng "%n" thì giá trị của ô nhớ tại địa chỉ 0x08480110 sẽ bị thay đổi. Giá trị mới chính là số ký tự mà lệnh printf đã in ra cho đến trước khi gặp ký hiệu định dạng %n. Bằng cách này, kẻ tấn công có thể ghi được giá trị bất kỳ vào ô nhớ. Nếu thực hiện thành công dạng tấn công này, kẻ tấn công có thể đạt được:
  - ✓ Ghi đè lên các cờ kiểm soát quyền truy cập trong chương trình;
  - ✓ Thay đổi địa chỉ trả về trong ngăn xếp, con trỏ hàm...

Câu 3: Khái niệm lỗ hổng chuỗi định dạng (format string); giải thích cơ chế khai thác lỗ hổng chuỗi định dạng để đọc dữ liệu dạng chuỗi từ một địa chỉ tùy ý.

- Khái niệm:

Lỗi **format string** là lỗi xuất hiện khi dữ liệu người dùng được được sử dụng làm format string (hoặc được đưa vào format string) trong các hàm thuộc họ printf.

  - printf, fprintf, sprintf, snprintf
  - vprintf, vfprintf, vsprintf, vsnprintf

- Cơ chế khai thác:

Để có thể đọc dữ liệu từ một địa chỉ nhớ tùy ý ngoài ngăn xếp thì cần làm được những việc sau:

- Phải có ký hiệu định dạng mà buộc hàm định dạng coi một giá trị trong ngăn xếp là địa chỉ bộ nhớ cần đọc. Đó là ký hiệu định dạng %s.
- Đưa được giá trị địa chỉ vào ngăn xếp. Thông thường, chuỗi định dạng cũng được lưu trong ngăn xếp. Vậy có thể đưa địa chỉ này vào chuỗi định dạng và nghiễm nhiên nó được lưu vào ngăn xếp.

3 Race condition

Câu 1: trình bày khái niệm lỗ hổng trường hợp đua (race condition); chỉ ra ví dụ về lỗ hổng trường hợp đua và khai thác lỗ hổng trường hợp đua.

- Khái niệm:

Trường hợp đua (Race Condition) là lỗ hổng xảy ra khi nhiều tiến trình truy cập và sửa đổi cùng một dữ liệu vào cùng một lúc, và kết quả cuộc việc thực thi phụ thuộc vào thứ tự của việc truy cập. Nếu một chương trình có lỗi này, kẻ tấn công có thể chạy nhiều chương trình song song để "đua" với chương

trình có lỗi, với mục đích thay đổi hoạt động của chương trình ấy. Lỗi này đôi khi còn được gọi là thời điểm kiểm tra/thời điểm sử dụng (TOC/TOU).

- Ví dụ:

```
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>

int main(int argc, char ** argv)
{
FILE *file;
char buffer[256];
if (access (argv[1], R_OK) ==0 ) {
    usleep(1);
    file = fopen(argv [1], "r");
    if(file == NULL)
    {
        goto cleanup;
    }
    fgets (buffer, sizeof(buffer),file);
    fclose(file);
    puts(buffer);
    return 0;
    }
cleanup;
    perror("cannot_open_file");
    return 0;
}
```

```
#gcc -o race race.c
#sudo chown root:root race
#sudo chmod u+s race
```

Tạo thêm 1 tập tin chỉ root đọc được:

```
#echo ‘you win!’ > race.txt
```

```
#chown root:root race.txt
#chmod 600 race.txt
```

Khai thác:

```
#./race race.txt
#sudo ./race race.txt
```

Chương trình ví dụ đọc nội dung của một tập tin có tên và in nội dung tập tin đó ra màn hình. Do chương trình có suid là root nên fopen có thể đọc được bất kì tập tin nào. Access để kiểm tra nếu như là người dùng thường sẽ không đọc được nội dung của các tệp đặc biệt (race.txt)

Nhưng fopen và access thực hiện 2 tác vụ tác rời nhau -> có 1 khoảng thời gian chèn vào giữa access và fopen -> khoảng thời gian này hđh chuyển qua thực thi 1 tiến trình khác -> khai thác được lỗi

- Khắc phục:
  - Để loại bỏ race condition thì cần xác định cửa sổ race (đoạn mã truy nhập tới đối tượng theo cách mở một cửa sổ)

4. Off-by-one

5 Integer Overflow

6 Shellcode