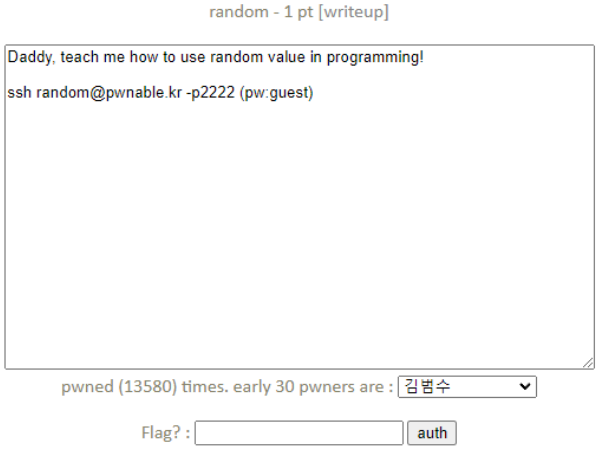
Pwnable.kr
- random-

ssh random@pwnable.kr -p2222
pw : guest

0. 문제 살펴보기



프로그래밍에서의 랜덤 값을 이용하는 법에 대하여 얘기를 하고 있다.

1. SSH 접속 및 살펴보기

SSH를 이용해 상단에 표기해놓은 주소와 포트 번호로 접속한다.

디렉토리의 파일들을 살펴보자 C 코드 파일이 존재하여 확인해보니 위와 같은 코드를 알 수 있었다.
랜덤 값을 생성하고 키 값을 입력 받아 “키와 랜덤의 XOR”의 값이 0xdeadbeef 와 같을 때 flag 파일을 볼 수 있는 것 같다.
하지만, 이 코드에는 취약점이 하나 존재한다. 랜덤 값 생성을 위해 rand()라는 함수를 사용하였는데, 이는 정해진 규칙에 따라 값이 생성된다는 것이다.
만약, 완전히 랜덤한 값을 생성하기 위해서는 랜덤 seed 값을 이용하여 생성해야 한다. 보통 이 값은 시간 값을 이용하며, srand() 함수를 이용하여 생성한다. srand(time(NULL))과 같이 말이다.
우리는 이 점을 이용하여 공략하면 될 것 같다.

2. 랜덤 값 확인해보기

실제로 같은 규칙에 의하여 랜덤 값이 생성되는지 살펴볼 것이다.

```
random@pwnable:~$ gdb -q random
Reading symbols from random...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas main
Dump of assembler code for function main:
   0x00000000004005f4 <+0>:     push   rbp
   0x00000000004005f5 <+1>:     mov    rbp,rsp
   0x00000000004005f8 <+4>:     sub    rsp,0x10
   0x00000000004005fc <+8>:     mov    eax,0x0
   0x0000000000400601 <+13>:    call   0x400500 <rand@plt>
   0x0000000000400606 <+18>:    mov    DWORD PTR [rbp-0x4],eax
   0x0000000000400609 <+21>:    mov    DWORD PTR [rbp-0x8],0x0
   0x0000000000400610 <+28>:    mov    eax,0x400760
   0x0000000000400615 <+33>:    lea    rdx,[rbp-0x8]
   0x0000000000400619 <+37>:    mov    rsi,rdx
   0x000000000040061c <+40>:    mov    rdi,rax
   0x000000000040061f <+43>:    mov    eax,0x0
   0x0000000000400624 <+48>:    call   0x4004f0 <__isoc99_scanf@plt>
   0x0000000000400629 <+53>:    mov    eax,DWORD PTR [rbp-0x8]
   0x000000000040062c <+56>:    xor    eax,DWORD PTR [rbp-0x4]
   0x000000000040062f <+59>:    cmp    eax,0xdeadbeef
   0x0000000000400634 <+64>:    jne    0x400656 <main+98>
   0x0000000000400636 <+66>:    mov    edi,0x400763
   0x000000000040063b <+71>:    call   0x4004c0 <puts@plt>
   0x0000000000400640 <+76>:    mov    edi,0x400769
   0x0000000000400645 <+81>:    mov    eax,0x0
   0x000000000040064a <+86>:    call   0x4004d0 <system@plt>
   0x000000000040064f <+91>:    mov    eax,0x0
   0x0000000000400654 <+96>:    jmp    0x400665 <main+113>
   0x0000000000400656 <+98>:    mov    edi,0x400778
   0x000000000040065b <+103>:   call   0x4004c0 <puts@plt>
   0x0000000000400660 <+108>:   mov    eax,0x0
   0x0000000000400665 <+113>:   leave
   0x0000000000400666 <+114>:   ret
End of assembler dump.
```

먼저, gdb 를 이용하여 main 함수의 어셈블리어를 살펴보았다.
<+18>과 <+56> 부분을 보아 rbp-0x4 가 random 변수의 위치인 것 같다.
랜덤 값이 생성된 후에 브레이크 포인트를 설정하여 프로그램 실행 시마다 값을 확인할 것이다.

```
(gdb) b *0x400619
Breakpoint 1 at 0x400619
```

▲ 브레이크 포인트 설정

```
(gdb) r
Starting program: /home/random/random

Breakpoint 1, 0x0000000000400619 in main ()
(gdb) x/x $rbp-0x4
0x7ffea95f760c: 0x6b8b4567
```

▲ 첫 번째 실행

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/random/random

Breakpoint 1, 0x0000000000400619 in main ()
(gdb) x/x $rbp-0x4
0x7fff9dab2a5c: 0x6b8b4567
```

▲ 두 번째 실행

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/random/random

Breakpoint 1, 0x0000000000400619 in main ()
(gdb) x/x $rbp-0x4
0x7ffcbc9657ec: 0x6b8b4567
```

▲ 세 번째 실행

세 번의 실행을 거쳤지만, random 값은 모두 0x6b8b4567 로 동일한 모습을 볼 수 있었다.
랜덤 값을 생성하였지만, 랜덤한 수가 아닌 것이다.

3. 공격

우리는 key와 random의 XOR 값을 0xdeadbeef로 만들어야 한다.
그렇다면 key를 어떠한 값으로 입력해야 할까?
XOR 연산은 비트가 같을 경우 0, 다를 경우 1을 출력한다.
따라서, random과 0xdeadbeef의 XOR 연산 값이 key와 같다면, key와 random의 XOR 연산 값이 0xdeadbeef가 될 것이다.
0x6b8b4567 ^ 0xdeadbeef = 0xb526fb88

key의 값은 현재 정수형으로 입력 받고 있으므로 이를 정수로 바꿔보면 3,039,230,856이다.
이를 random 프로그램 생성 후 입력해보았다.

```
random@pwnable:~$ ./random
3039230856
Good!
```

flag 파일이 정상적으로 읽혀 우리가 찾고자 했던 flag를 얻었다.