pwnable.kr Toddler's bottle 풀이

취약점분석 박 지희

**fd**

파일을 실행하면 argv[1]에 지정된 fd에서 32byte를 읽는다. fd를 지정하는 방식이

int fd = atoi( argv[1] ) - 0x1234; 인데, 0x1234를 인자로 넘겨주면 fd = 0이기 때문에 stdin에서 입력을 받게 된다. 따라서

fd@ubuntu:~$ ./fd 4660

LETMEWIN

good job :)

mommy! I think I know what a file descriptor is!!

이렇게 입력하면 답을 얻을 수 있다.

**collision**

스스로 구현한 hash function에서 collision을 내는 문제이다. 20byte를 입력해 hashcode = 0x21DD09EC가 되는 string을 찾으면 되는데, hash function이 20byte를 네 바이트씩 끊어 더하는 단순한 방식이다. 따라서 다섯 숫자를 더해 0x21dd09ec가 되는 숫자를 입력하면 된다. 나의 경우에는

0x1dd905e8 +0x01010101 + 0x01010101 + 0x01010101 + 0x01010101 을 입력하였다.

string으로 저장할 때는 한 바이트씩 저장되지만 int로 읽을때는 리틀 엔디안 방식으로 뒤부터 4바이트씩 끊어 읽기 때문에

col@ubuntu:~$ ./col `python -c 'print "₩xe8₩x05₩xd9₩x1d"+"₩x01"*16'`

daddy! I just managed to create a hash collision :)

위와 같이 입력하면 답이 나온다.

**bof**

gets 함수에서 overflow가 발생하는데, 이 버그를 이용해 함수인자의 값을 변조하려고 시도했다. 먼저 오버플로우 방어가 되어 있는지 보기 위해 'a'*33을 입력하였더니, 스택 변조가 감지되었다는 말이 나오고 종료되었다. stack smash protector가 걸려 있는 것을 확인하고, canary의 크기를 넘어서 오버플로우를 일으킬 때까지 입력값의 길이를 늘렸다. canary 값 확인은 리턴 시 이루어져 중간에 실행되는 코드에서의 값은 변조가 가능하기 때문에, 52바이트를 a로 채운 후 나머지 4바이트를 '₩xbe₩xba₩xfe₩xca'로 채워 key값을 변조했다.

**flag**

바이너리 파일을 ida를 이용해 분석하려고 했지만, 함수들의 위치가 이상하게 떨어져 있고 string들이 보이지 않아 제대로 풀리지 않았다. 그러던 중 upx 패킹이 되어 있다는 사실을 알고 언패킹한 후, 다시 분석하였더니

I will malloc() and strcpy the flag there. take it. 라고 출력한 후 두 함수를 실행하는 것을 알았다.

첫 번째 함수는 malloc이었고, 두 번째 함수는 이름이 나와있지 않았지만 64 bit의 strcpy일 것이다. 따라서 디어셈블리 화면에서 strcpy의 call에서 사용되는 레지스터에 저장되는 것들을 찾아본 결과 rdx에 flag가 저장되는 것을 확인했다.

**passcode**

main 함수에서 welcome();과 login();을 순차적으로 실행하는데, 여기서 welcome()에서 쓰이던 stack이 login에서도 그대로 쓰이게 된다. (ebp와 esp와 stack canary의 크기가 다 일정하다고 가정하면)

이 프로그램에 있는 버그 두 가지는 첫 번째로, int passcode1, int passcode2를 초기화하지 않은 것이고 (scanf로 받을 때는 상관 없을 수도 있지만 이 문제 같은 실수에서는 초기화를 하지 않은 게 문제가 된다), 두 번째로 scanf에서 int의 주소를 인자로 준 것이 아니라 int값을 인자로 준 것이다.

따라서 name을 입력할 때 원하는 주소를 쓴 후, scanf 실행에서 원하는 값을 쓰면 arbitrary memory write가 가능하게 되는데, 그걸 이용해 원하는 함수의 got 값을 system("/bin/cat flag")가 실행되는 주소로 바꿔주면 답이 나오게 된다. exit의 got 주소 0x0804a018에 내부 코드의 주소 0x80485e3으로 덮어써 exit이 실행될 때 답이 나오도록 했다.

**random**

random값을 맞추는 문제인데, libc에서 random은 seed에 따라 생성되는 난수 수열이다. seed가

없으면 seed가 1로 고정되어 계속 같은 수열이 나오기 때문에, c 코드를 만들어 rand()를 실행해 보면 그 값을 알 수 있다.

**input**

여러 가지 input을 주면 답을 출력하는 문제이다. 첫 번째로 argument를 주는 방법은 shell에서 띄어쓰기로 구분하여 argument를 주는 방법도 있고, 프로그래밍 언어에서 exec 계열 함수를 사용할 때 배열로 주는 방법도 있다. ₩x00을 전하기 위해서는 쉘로는 전할 수 없기 때문에 c 파일을 이용하는 것을 선택했다. argv['A'] 는 argv[65]와 같은 방식으로 처리된다.

두 번째로는 fd 0과 2에서 입력을 받는다. 0에서는 standard input이 기본으로 지정되어 있기 때문에 입력할 수 있지만, 2에서는 standard error가 기본으로 지정되어 있기 때문에 원하는 문자를 출력할 수 없다. 따라서 standard in과 standard error의 fd 할당을 해제(close)하고 새로운 파일을 열어 fd를 할당했다.

세 번째로 환경변수에 대해 나오는데, 쉘에서 export 명령을 사용하는 것과 c 파일에 arguemnt로 주는 방법이 있다. c에서는 execve 함수를 이용해 환경변수를 정할 수 있다.

네 번째로는 파일을 불러와 읽는다. 간단히 파이썬에서 "₩x0a"라는 이름을 가진 파일을 생성해 원하는 문자를 입력했다.

다섯 번째로는 네트워크 통신으로 문자를 읽는데, argv[67]에 저장된 포트로 연결한다. 따라서 포트를 50485으로 설정한 뒤 파이썬 소켓 통신을 통해 값을 전달했다.

```
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
int main(){
        const char * name = "/home/input/input";
        char *const argv[]={"/home/input/input", "", "", "", "", "", "", "",
"", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "",
"", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "",
"", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "",
"", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "",
"\x00", "\x20\x0a\x0d", "50485", "", "", "", "", "", "", "", "", "", "", "",
"", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "",
"", "", NULL};
        char* const envp[] =
{"PATH=/usr/bin:/bin:/home/input:/tmp/inputtemp","\xde\xad\xbe\xef=\xca\xfe
\xba\xbe",NULL};
        //argv[0] = "/home/input/input";
        //argv['A']="\x00";     //argv['B']="\x20\x0a\x0d";
        printf("%d ", sizeof(argv));
        close(0);
        close(2);
        int a0 = open("/tmp/inputtemp/to0",O_RDWR);
        int a2 = open("/tmp/inputtemp/to2",O_RDWR);
        //printf("%d %d\n",a0,a2);
        execve("/home/input/input" , argv, envp);       return 0;
        }
}
```

위와 같이 c 파일을 만들고 실행했는데, 답이 나오지 않아 error를 확인했다.(to2 파일에서)

flag 파일을 찾지 못했다고 해 symbolic link를 걸어주어 답을 확인했다.

**leg**

leg의 소스 부분을 보면, key1()과 key2()와 key3()를 더한 값과 input을 비교하게 된다. 어셈블리에서 그 내용을 확인할 때, 함수가 끝날 때 r0에 그 값을 저장하고, 메인에서는 그 값을 r3에 넣고 더해 최종적으로 r2가 key1()+key2()+key3()이 된다. 따라서 각 함수에서 저장되는 r0를 분석하면 세 수를 더한 값을 알 수 있게 된다. key1()에서는 단순히 mov r3,pc를 하게 되는데, 이것으로 인해 현재의 위치 + 8의 값을 r3로 대입하게 된다.

key2()에서는 더 복잡해지는데, 먼저 bx를 통해 arm mode를 thumb mode로 바꾸게 되고 그 후에 pc값을 r3에 저장하면 현재 위치 + 4의 값이 r3로 들어가게 되는데 거기에 다시 4를 더하게 된다.

key3()은 lr을 r3에 넣게 되는데 그 값은 함수 호출이 끝난 뒤 다음 갈 곳의 주소이다.

따라서 세 수를 모두 더하면 0x00008d80+0x0008ce4+0x00008d0c = 108400이 나와, 플래그를 얻을 수 있다.

**mistake**

위 프로그램에서는 /home/mistake/password에서 string을 읽어온 후 1과 xor했을 때 같은 값이면

key를 보여주는 프로그램이다. 하지만 실행해 보았을 때 입력을 두번 하게 되어 다시 살펴보게 되었는데, fd를 지정하는 부분에서 실수가 존재한다. 등호와 부등호 중에는 부등호가 더 우선순위가 높기 때문에,

fd = open("/home/mistake/password",O_RONLY,0400) < 0

에서는 먼저 open을 실행한 값(아마 3 정도) 가 나오게 되고, 그러면 fd = 3 < 0 이 되어 fd = 0이 된다. 결국 read(fd)를 실행했을 때, stdin에서 읽은 값이 들어가는 것이다.

이것을 이용하여 bbbbbbbbbb을 입력하고 다시 ccccccccc를 입력하면 문제가 풀린다.

**shellshock**

bash –c 'echo shock_me'를 실행하는 단순한 프로그램이다. 하지만 bash 쉘에 취약점이 있다고 하여 shellshock 취약점에 대해 검색해 보니 bash shell에서 parsing 문제로 인해 일어나는 취약점이었다. 관련된 취약점은 CVE-2014-6271과 CVE-2014-7169였는데, CVE-2014-7169를 이용해 cat flag를 실행하였다.

env X='() { (a)=>₩' bash –c "echo date"

를 실행하면 date라는 문자가 출력되야 하는게 정상이지만 date를 실행한 결과를 echo에 저장하게 된다. 이를 shellshock에 echo shock_me가 있다는 것에 응용하여, shock_me 에 cat /home/shellshock/flag를 저장한다. 그 후 env X='() { (a)=>₩' /home/shellshock/shellshock를 실행하면 echo라는 파일에 flag가 저장되어 있는 것을 볼 수 있다.

**coin1**

N개의 coin들 중에 불량품을 C번만에 찾아내는 문제이다. n개의 코인을 뽑았을 때 10*n의 weight가 나오면 모두 정상품이고, 10*n-1개가 나오면 하나의 불량품이 존재하는 것으로 추측할 수 있다. 따라서, 임의의 집합을 잡았을 때 불량품 index가 있는지 없는지를 알아낼 수 있다.

이 문제에 대한 간단한 풀이는 binary search를 이용한 풀이이다. 불량품이 존재할 수 있는 범위를 반으로 줄여 나가면서 불량품이 있는 곳을 찾는 것으로 2^C>N이라면 항상 결정할 수 있다. 따라서 파이썬으로 binary search를 구현하여 문제를 100번 풀었다.

**blackjack**

블랙잭 프로그램이다. 베팅을 한 뒤 블랙잭을 이기면 베팅 한 만큼 돈이 더해지고, 지면 그만큼 빼지는 프로그램이다. 여기서 베팅 금액을 정할 때 문제가 있는데,

```
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
```

이 부분이다. bet이 cash보다 많을 때는 다시 베팅을 하게 하는데, 이 때 다시 한번 cash보다 많은 값을 입력하면 다시 검사하지 않고 베팅이 된다. 이 방법을 이용해서 한번에 많은 금액을 얻을 수 있다.

또 다른 풀이는 bet에 음수 값을 입력하는 것이다. 0보다 작은 값을 입력했을 때 if문으로 처리해 주지 않으므로 -100000같은 값으로 베팅을 한 후 지면 돈의 값이 올라간다.

**lotto**

6개의 값을 입력한 후 count값이 6이 되면 flag를 출력해 주는 문제이다. 이 문제에서 잘못 코딩된 점은, 같은 값을 입력했을 때 그것을 처리해주지 못한다는 점이다.

입력이 1 1 1 1 1 1 이고,

랜덤값이 1 2 3 4 5 6 일때,

count를 세는 코드는 그 값을 6으로 계산한다. 이런 것을 제대로 처리하기 위해서는 입력 중 같은 숫자가 있을 때 다시 입력을 받도록 해야 한다. 이것을 이용해 1/45의 확률로 답을 출력할 수 있다.

lotto@ubuntu:~$ (python -c 'print "1"';python -c 'print "₩x01₩x01₩x01₩x01₩x01₩x01"';python -c 'print "3"';)|./lotto

를 여러 번 실행하여 답을 얻었다.