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(一)

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4次/总4193次

Format String 漏洞介绍/总结(一)
bkbll(bkbll@cnhonker.net)
2003/4/8
其实这篇文章没什么技术含量，format string(格式化字符串)漏洞很久很久就被研究透了，scut的一篇pd文档属于非常详细的介绍/入门级文章，但是全英文以及里面例子有些解释不彻底， 以及有些例子已经不能使用了，所以这里想大致总结一下，并给出实验好的环境和代码.(实验平台：rh8.0,gcc自带版本)
Format string 漏洞一般是由以下几个函数引起的：
o fprintf - 输出到文件句柄
o printf - 输出到终端
o sprintf - 输出到一个字符串
o snprintf -输出指定长度到字符串
o vfprintf - print to a FILE stream from a va_arg structure
o vprintf - prints to 'stdout' from a va_arg structure
o vsprintf - prints to a string from a va_arg structure
o vsnprintf - prints to a string with length checking from a va_arg structure
Relatives:
o setproctitle - set argv[]
o syslog - output to the syslog facility
o 其他比如 err*, verr*, warn*, vwarn*

在使用这些函数的时候，如果指定了format格式的话，是不存在任何问题的，但是如果程序员偷懒，没指定format而直接输出字符串内容的话，就导致格式化字符串漏洞的发生，比如：

```
char *buffer;
……….
printf("%s\n",buffer);
```

这段程序是不会产生字符串格式化的漏洞的，但是下面这个程序：

```
char *buffer;
…….
printf(buffer);
```

如果buffer可以由用户控制的话，就会导致格式化字符串漏洞的发生。
类似还有：

```
syslog (LOG_NOTICE, buf);
fprintf(FILE *stream,buffer);
sprintf(char *string,buffer);
snprintf(char *string,strlen(string),buffer)
vfprintf(File *stream,buffer);
```

等。
1. 漏洞的产生/介绍
我们选用应用最普遍的printf函数来解答format string 漏洞的原理：

我们知道，一个标准正常的printf函数的格式化字符和参数应该是一一对应的，比如：
printf("%s%d%x\n",(char *)string,(int)intnum,(int)hexnum);
有几个%s等，后面就应该有几个参数，这样才可以一一显示该参数的内容，但是，如果有了格式化字符，如果没有跟参数，printf函数会怎么处理的呢？
[bkbll@mobile format]$ cat 6.c
main() { printf("%p %p %p %p %p %p\n"); }
%p表示按指针格式显示结果,我们编译运行下看看:
[bkbll@mobile format]$ gcc -o 6 6.c
[bkbll@mobile format]$ ./6
0x4212a2d0 0xbffffaf8 0x8048246 0x4200aec8 0x4212a2d0 0xbffffb18
显示的是一大堆的内存数据, 我们看看这些数据到底放在哪里的:
[bkbll@mobile format]$ gdb -q 6
(gdb) b main
Breakpoint 1 at 0x804832e
(gdb) r
Starting program: /home/bkbll/format/6

Breakpoint 1, 0x0804832e in main ()
(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/8wx $esp
0xbffffacc: 0x08048345 0x08048394 0x4212a2d0 0xbffffae8
0xbffffadc: 0x08048246 0x4200aec8 0x4212a2d0 0xbffffb08
(gdb)
从这里我们看到, printf的入口在0x42052390, 我们分析一下堆栈数据的结构:
当系统调用某个函数的时候,首先会将函数参数压入堆栈, 最后把函数的返回地址压入堆栈, 上面的0x08048345是函数printf的返回地址, 也就是在main函数里面调用printf函数后下一条要执行的指令.0x08048394存放的是我们给printf的参数:
(gdb) x/s 0x08048394
0x8048394 <_IO_stdin_used+4>: "%p %p %p %p %p %p\n"
由于我们给printf了很多的格式化字符%p,但是又没给上相应的参数, 系统认为紧跟格式化字符串后面的数据即为printf的参数,所以就按照%p的格式打印在了终端上.
如果我给出了足够多的%p, 是否可以一直打印数据到0xbfffffff呢? 答案是肯定的, 这个不段用%p显示内存数据就是在scut的pdf上讲到的stack popup的涵义.
好,我们现在可以显示调用printf函数堆栈以上的内容了, 但是我们可以显示任意内存地址的内容吗? 我们看以下事例:
[bkbll@mobile format]$ cat 7.c
main()
{
char buffer[100]="";
strcpy(buffer,"AAAA%p %p %p %p %p %p\n");
printf(buffer);

}
[bkbll@mobile format]$ gcc -o 7 7.c
[bkbll@mobile format]$ ./7
AAAA0x8048458 0x4200dbb3 0x420069e8 0x41414141 0x25207025 0x70252070
0x41414141就是我们写的AAAA的16进制码, 如果我把显示0x41414141的%p换成%s, 不是就可以显示0x41414141地址的内容呢?
[bkbll@mobile format]$ cat 8.c
main()
{
char buffer[100]="";
strcpy(buffer,"AAAA%p %p %p %s %p %p\n");
printf(buffer);

}
[bkbll@mobile format]$ gcc -o 8 8.c ;./8
Segmentation fault
段错误, 我们跟踪一下:
[bkbll@mobile format]$ gdb -q 8
(gdb) r
Starting program: /home/bkbll/format/8

Program received signal SIGSEGV, Segmentation fault.

```
0x4207a4cb in strlen () from /lib/i686/libc.so.6
(gdb) disass $eip $eip+4
Dump of assembler code from 0x4207a4cb to 0x4207a4cf:
0x4207a4cb <strlen+11>: cmp %ch,(%eax)
0x4207a4cd <strlen+13>: je 0x4207a56a <strlen+170>
End of assembler dump.
(gdb) i reg ecx eax
ecx 0x1 1
eax 0x41414141 1094795585
(gdb) x/bx $eax
0x41414141: Cannot access memory at address 0x41414141
```
Oh,因为我们没有权限访问0x41414141这个地址,所以系统提示段错误.
那我们换一个我们可以访问的地址吧:
```
[bkbll@mobile format]$ cat 9.c
main()
{
char buf1[]="hello,world";
char buffer[100]="";
strcpy(buffer,"AAAA%p %p %p %s %p %p\n");
buffer[0]=(int)buf1 & 0xff;
buffer[1]=((int)buf1 >> 8) & 0xff;
buffer[2]=((int)buf1 >> 16) & 0xff;
buffer[3]=((int)buf1 >> 24) & 0xff;
printf(buffer);

}
[bkbll@mobile format]$ gcc -o 9 9.c ; ./9
帴    ?x80484cc (nil) (nil) hello,world 0x25207025 0x70252070
```
我们输出了hello,world字符串, 而这个字符串的地址是我们替换了AAAA的数据得到的.
从上面的例子我们可以看出通过精心构造buffer, 我们可以显示任何地方的数据, 也就是所谓的:read anywhere.
能读数据虽然可以得到很多东西,但结构并不是我们想要的, 我们要可写才可以控制这个程序的流程, 才能运行我们的shellcode.
printf系列函数提供了%n的格式, 用来把显示的数据长度写进一个int型的变量里面, 比如:
```
[bkbll@mobile format]$ cat 10.c
main()
{
int i=0;
printf("before printf,i:%d\n",i);
printf("hello,word\n%n",&i);
printf("after printf,i:%d\n",i);
}
[bkbll@mobile format]$ gcc -o 10 10.c;./10
before printf,i:0
hello,word
after printf,i:11
```
我们把printf的输出长度写到了变量i里面,所以i值变成了11,既然可以写, 那我再试试可不可以写到其他地方,我们试一下写到main的返回地址里面:
```
[bkbll@mobile format]$ gdb -q 10
(gdb) x/i main
0x8048328 <main>: push %ebp
(gdb) b *0x8048328
Breakpoint 1 at 0x8048328
(gdb) r
Starting program: /home/bkbll/format/10

Breakpoint 1, 0x08048328 in main ()
(gdb) x/wx $esp
0xbffffaec: 0x420158d4
```
我们得到了main的返回地址在0xbffffaec处.
Ok, 我们修改一下程序:
```
[bkbll@mobile format]$ cat 11.c
main()
{
int i=0xbffffaec;
printf("hello,word\n%n",i);
}
[bkbll@mobile format]$ gcc -o 11 11.c
[bkbll@mobile format]$ gdb -q 11
(gdb) r
Starting program: /home/bkbll/format/11
hello,word
```

Program received signal SIGSEGV, Segmentation fault.
0x0000000b in ?? ()
(gdb) i reg eip
eip 0xb 0xb
(gdb)
ok,我们已经成功的把数据写到了main返回地址那里, 0x000000b显然是一个不可以执行的地址, 所以会报错.
联想一下,结合前面的read anywhere和这里的写, 我们是否可以动态写数据到任何地址呢?
[bkbll@mobile format]$ cat 12.c
main()
{
int buf1=0xbffffaec;
char buffer[100]="";
strcpy(buffer,"AAAA%p %p %p %n %p %p\n");
buffer[0]=(int)buf1 & 0xff;
buffer[1]=((int)buf1 >> 8) & 0xff;
buffer[2]=((int)buf1 >> 16) & 0xff;
buffer[3]=((int)buf1 >> 24) & 0xff;
printf(buffer);

}
[bkbll@mobile format]$ gcc -o 12 12.c
[bkbll@mobile format]$ gdb -q 12
(gdb) r
Starting program: /home/bkbll/format/12
禧　?x80484b0 (nil) (nil) 0x25207025 0x70252070

Program received signal SIGSEGV, Segmentation fault.
0x0000001a in ?? ()
(gdb) x/wx 0xbffffaec
0xbffffaec: 0x0000001a
(gdb) i reg eip
eip 0x1a 0x1a
我们成功的覆盖了main的返回地址?BR>利用printf的格式化字符串漏洞，我们可以write-anywhere, read anywhere。有了这两个条件后，想执行我们的shellcode还不是简单的事情吗？

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