php逆向ea8.0,DDCTF2019两个逆向writeup

转载

田智广 于 2021-03-25 23:07:21 发布 98 收藏
文章标签： php逆向ea8.0
原创：合天智汇

01Confused

首先参考链接 https://www.52pojie.cn/forum.php?mod=viewthread&tid=860237&page=1

首先分析到这个sub_1000011D0是关键函数是没有什么问题的，直接shift+f12定位DDCTF的字符串就到了这一部分逻辑

```
if ( (unsigned int)sub_1000011D0(*((__int64 *)&v14 + 1)) == 1 )

objc_msgSend(v17, "onSuccess");

else

objc_msgSend(v17, "onFailed");
```

跟进去以后发现

```
__int64 __fastcall sub_1000011D0(__int64 a1)

{

char v2; // [rsp+20h] [rbp-C0h]

__int64 v3; // [rsp+D8h] [rbp-8h]

v3 = a1;

memset(&v2, 0, 0xB8uLL);

sub_100001F60(&v2, a1);

return (unsigned int)sub_100001F00(&v2);

}
```

这个函数首先分配了一个0xb8大小的空间，然后填充为0x00，然后把这个空间传入了一个sub_100001F60函数跟进去

```
_int64 __fastcall sub_100001F60(__int64 a1, __int64 a2)

{

*(_DWORD *)a1 = 0;

*(_DWORD *)(a1 + 4) = 0;

*(_DWORD *)(a1 + 8) = 0;

*(_DWORD *)(a1 + 12) = 0;
```

```
*(_DWORD *)(a1 + 16) = 0;

*(_DWORD *)(a1 + 176) = 0;

*(_BYTE *)(a1 + 32) = 0xF0u;

*(_QWORD *)(a1 + 40) = sub_100001D70;

*(_BYTE *)(a1 + 48) = 0xF1u;

*(_QWORD *)(a1 + 56) = sub_100001A60;

*(_BYTE *)(a1 + 64) = 0xF2u;

*(_QWORD *)(a1 + 72) = sub_100001AA0;

*(_BYTE *)(a1 + 80) = 0xF4u;

*(_QWORD *)(a1 + 88) = sub_100001CB0;

*(_BYTE *)(a1 + 96) = 0xF5u;

*(_QWORD *)(a1 + 104) = sub_100001CF0;

*(_BYTE *)(a1 + 112) = 0xF3u;

*(_QWORD *)(a1 + 120) = sub_100001B70;

*(_BYTE *)(a1 + 128) = 0xF6u;

*(_QWORD *)(a1 + 136) = sub_100001B10;

*(_BYTE *)(a1 + 144) = 0xF7u;

*(_QWORD *)(a1 + 152) = sub_100001D30;

*(_BYTE *)(a1 + 160) = 0xF8u;

*(_QWORD *)(a1 + 168) = sub_100001C60;

qword_100003F58 = malloc(0x400uLL);

return __memcpy_chk((char *)qword_100003F58 + 48, a2, 18LL, -1LL);

}
```

首先先按H把-16这些改成 unsigned_int8类型，看着方便。 这里对照前面那个文章里的看，就是在对vm_cpu结构体进行初始化，前6个四字节的显然是寄存器，后几个是绑定虚拟机字节码和字节码对应的函数 这里可以切换到structures窗口按insert创建一个结构体，这样来更清晰的看代码 首先要对下面初始化的结构体和opcode对应的每个函数简要分析 第一个0xF0虚拟机指令对应的函数

```
_int64 __fastcall sub_100001D70(__int64 a1)

{

__int64 result; // rax

signed int *v2; // [rsp+Ch] [rbp-18h]

v2 = (signed int *)(*(_QWORD *)(a1 + 24) + 2LL);
```

```
switch ( *(unsigned __int8 *)(*(_QWORD *)(a1 + 24) + 1LL) )
{
case 0x10u:
*(_DWORD *)a1 = *v2;
break;
case 0x11u:
*(_DWORD *)(a1 + 4) = *v2;
break;
case 0x12u:
*(_DWORD *)(a1 + 8) = *v2;
break;
case 0x13u:
*(_DWORD *)(a1 + 12) = *v2;
break;
case 0x14u:
*(_DWORD *)a1 = *((char *)qword_100003F58 + *v2);
break;
default:
break;
}
result = a1;
*(_QWORD *)(a1 + 24) += 6LL;
return result;
}
```

首先可以看到每个函数中一开始都在引用 (_QWORD*)(a1+24)，于是猜测这是指令指针寄存器，在structure窗口中先创建结构体给这个偏移处的qword*取个名字叫myeip

可以看到v2是虚拟机指令指针寄存器指向的地方后面第二个字节的内容，然后有一个switch结构，是在判断虚拟机指令后面第一个字节指定的是哪个虚拟机寄存器，然后将v2赋值到寄存器里，如果是0x14的话就把 *((char*)qword_100003F58+*v2)赋值给第一个寄存器，显然这qword100003F58相当于虚拟机的栈，然后函数负责移动指令指针寄存器，这个函数对应的虚拟机指令占六个字节，所以在最后有 *(_QWORD*)(a1+24)+=6LL;，所以这个函数的功能就是把一个立即数传入虚拟机的一个寄存器中，给他取个名字叫movreg_imm，方便后面查看

然后第二个0xF1对应的函数

```
__int64 __fastcall sub_100001A60(__int64 a1)

{

__int64 result; // rax

result = (unsigned int)(*(_DWORD *)(a1 + 4) ^ *(_DWORD *)a1);

*(_DWORD *)a1 = result;

++*(_QWORD *)(a1 + 24);

return result;

}
```

可以看到他把前两个寄存器里的值进行异或然后又把结果放到了第一个寄存器中，同样的这个虚拟机指令占一个字节，此函数负责将指令指针寄存器加1字节

0xF2对应的函数和0xF6对应的函数要结合来看

```
_int64 __fastcall sub_100001AA0(__int64 a1)

{

__int64 result; // rax

*(_DWORD *)(a1 + 16) = *(_DWORD *)a1 == *((char *)qword_100003F58 + *(unsigned __int8 *)(*(_QWORD *)(a1 + 24) + 1LL));

result = a1;

*(_QWORD *)(a1 + 24) += 2LL;

return result;

}

__int64 __fastcall sub_100001B10(__int64 a1)

{

__int64 result; // rax

if ( *(_DWORD *)(a1 + 16) )

*(_DWORD *)(a1 + 16) = 0;

else

*(_QWORD *)(a1 + 24) += *(unsigned __int8 *)(*(_QWORD *)(a1 + 24) + 1LL);

result = a1;

*(_QWORD *)(a1 + 24) += 2LL;

return result;

}
```

可以看到0xF2中他把

(_DWORD)a1==*((char*)qword_100003F58+*(unsigned__int8*)(*(_QWORD*)(a1+24)+1LL));这个表达式的结果放到了 *(_DWORD*)(a1+16)里，0xF6中他有根据 *(_DWORD*)(a1+16)移动eip指令指针寄存器，这就很明显了，这个寄存器的功能就是标志寄存器，存放对比的结果，然后后面一个函数就是条件跳转指令了

然后0xF4

```
_int64 __fastcall sub_100001CB0(__int64 a1)
{
__int64 result; // rax
result = (unsigned int)(*(_DWORD *)(a1 + 4) + *(_DWORD *)a1);
*(_DWORD *)a1 = result;
++*(_QWORD *)(a1 + 24);
return result;
}
```

很明显的看出这个实在做加法运算，把前两个寄存器的值加起来放到第一个寄存器里，取个名字叫addregimm

然后0xF5

```
_int64 __fastcall sub_100001CF0(__int64 a1)
{
__int64 result; // rax
result = (unsigned int)(*(_DWORD *)a1 - *(_DWORD *)(a1 + 4));
*(_DWORD *)a1 = result;
++*(_QWORD *)(a1 + 24);
return result;
}
```

与上面的函数类似，这个是减法，取个名字叫subregimm

跟进0xF3对应的函数发现只有一个花括号，于是取名为nop

然后0xF7

```
__int64 __fastcall sub_100001D30(__int64 a1)
{
__int64 result; // rax
result = *(unsigned int *)(*(_QWORD *)(a1 + 24) + 1LL);
*(_DWORD *)(a1 + 176) = result;
*(_QWORD *)(a1 + 24) += 5LL;
return result;
```

```
}
```

可以看到他把操作数放进了 (_DWORD*)(a1+176)处，先不管他是在干啥，先看0xF8

```
__int64 __fastcall sub_100001C60(__int64 a1)
{
__int64 result; // rax
result = sub_100001B80((unsigned int)(char)*(_DWORD *)a1, 2LL);
*(_DWORD *)a1 = (char)result;
++*(_QWORD *)(a1 + 24);
return result;
}
__int64 __fastcall sub_100001B80(char a1, int a2)
{
bool v3; // [rsp+7h] [rbp-11h]
bool v4; // [rsp+Fh] [rbp-9h]
char v5; // [rsp+17h] [rbp-1h]
v4 = 0;
if ( a1 >= 65 )
v4 = a1 <= 90;
if ( v4 )
{
v5 = (a2 + a1 - 65) % 26 + 65;
}
else
{
v3 = 0;
if ( a1 >= 97 )
v3 = a1 <= 122;
if ( v3 )
v5 = (a2 + a1 - 97) % 26 + 97;
else
v5 = a1;
```

```
}

return (unsigned int)v5;

}
```

可以看到他调用了下级函数sub100001B80，显然这个sub100001B80函数是在对(unsigned int)(char)*(_DWORD *)a1进行移位为2的凯撒加密，首先他通过ascii码的范围判断是大写还是小写，以确保大小写字母都能够通用，然后对其进行凯撒移位

分析完了这几个函数和几个寄存器以后，可以把结构体补充完整了 如下 前四个就是通用用途寄存器，第五个就是标志寄存器，第六个是指令指针寄存器，其余的函数的功能就如其名称所示

```
_int64 __fastcall sub_100001F60(vm_cpu *a1, __int64 a2)

{

a1->vm_r1 = 0;

a1->vm_r2 = 0;

a1->vm_r3 = 0;

a1->vm_r4 = 0;

a1->flag = 0;

a1->myeip = 0;

LOBYTE(a1->opcode_f0) = 0xF0u;

a1->mov_reg_imm = (__int64)mov_reg_imm;

LOBYTE(a1->opcode_f1) = 0xF1u;

a1->xor_r1_r2 = (__int64)xor_r1_r2;

LOBYTE(a1->opcode_f2) = 0xF2u;

a1->cmp_r1_imm = (__int64)cmp_r1_imm;

LOBYTE(a1->opcode_f4) = 0xF4u;

a1->add_r1_r2 = (__int64)add_r1_r2;

LOBYTE(a1->opcode_f5) = 0xF5u;

a1->dec_r1_r2 = (__int64)sub_r1_r2;

LOBYTE(a1->opcode_f3) = 0xF3u;

a1->nop = (__int64)nop;

LOBYTE(a1->opcode_f6) = 0xF6u;

a1->jz_imm = (__int64)jz_imm;

LOBYTE(a1->opcode_f7) = 0xF7u;

a1->mov_buff_imm = (__int64)mov_buff_imm;
```

LOBYTE(a1->opcode_f8) = 0xF8u;

a1->shift_r1_2 = (__int64)shift_r1_2;

qword_100003F58 = malloc(0x400uLL);

return __memcpy_chk((char *)qword_100003F58 + 48, a2, 18LL, -1LL);

}

现在返回去查看sub_100001F00函数里调用的第二个函数

__int64 __fastcall sub_100001F00(vm_cpu *a1)

{

a1->myeip = (__int64)&loc_100001980 + 4;

while (*(unsigned __int8 *)a1->myeip != 0xF3)

sub_100001E50(a1);

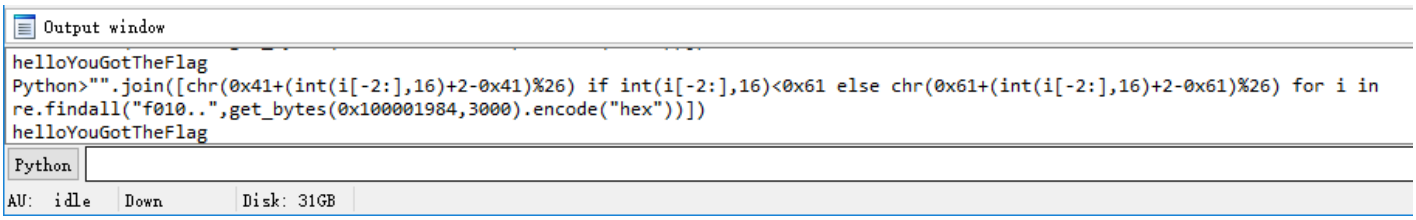
free(qword_100003F58);

return a1->vm_r6;

}

可以看到此函数初始化了myeip指令指针寄存器将其指向 (__int64)&loc_100001980+4的地方 现在vm_cpu结构体搞清楚了，就可以照着结构体翻译 (__int64)&loc_100001980+4处的虚拟机opcode了 这里既然我们已经了解了这些opcode的功能，其实并不需要费力去提取出代码执行或者写脚本一句一句翻译了 他就是把一个立即数装入r1中，然后判断大小写，对其进行凯撒移位，我们直接在idapython里写一句脚本便出来了

"".join([chr(0x41+(int(i[-2:],16)+2-0x41)%26) if int(i[-2:],16)<0x61 else chr(0x61+(int(i[-2:],16)+2-0x61)%26) for i in re.findall("f010..",get_bytes(0x100001984,3000).encode("hex"))])



当然也可以写个脚本把整个虚拟机字节码全部翻译出来，鉴于比赛时的时间限制，如今比赛已停止，可以看一下 首先可以用get_bytes(0x100001984,3000,0).encode("hex")把虚拟机字节码给提取出来，也可以用lazyida插件 然后解析代码如下

import re

loc_100001980="f01066000000f8f230f6c1f01063000000f8f231f6b6f0106a000000f8f232f6abf0106a000000f8f2

loc_100001980=re.findall("..",loc_100001980)

#print(loc_100001980)

a=0

code=[]

```
c=[
["f0",6,lambda x:str("mov r"+x[1]+",0x"+x[2])],
["f1",1,lambda x:str("xor r10,r11")],
["f2",2,lambda x:str("cmp r10,byte ptr ss:[0x"+x[1]+"]")],
["f3",1,lambda x:"nop"],
["f4",1,lambda x:str("add r10,r11")],
["f5",1,lambda x:str("sub r10,r11")],
["f6",2,lambda x:str("jz "+x[1])],
["f7",5,lambda x:str("mov buf,imm")],
["f8",1,lambda x:str("caesar encode r10,2")]
]
while 1:
for i in c:
if(loc_100001980[a] in i):
print(i[2](loc_100001980[a:a+i[1]]))
#print(loc_100001980[a:a+i[1]])
a+=i[1]
break
```

运行后效果如下，极大的增强了可读性

```
mov r10,0x66
caesar encode r10,2
cmp r10,byte ptr ss:[0x30]
jz c1
mov r10,0x63
caesar encode r10,2
cmp r10,byte ptr ss:[0x31]
jz b6
mov r10,0x6a
caesar encode r10,2
cmp r10,byte ptr ss:[0x32]
jz ab
```

```
mov r10,0x6a
caesar encode r10,2
cmp r10,byte ptr ss:[0x33]
jz a0
mov r10,0x6d
caesar encode r10,2
cmp r10,byte ptr ss:[0x34]
jz 95
mov r10,0x57
caesar encode r10,2
cmp r10,byte ptr ss:[0x35]
jz 8a
mov r10,0x6d
caesar encode r10,2
cmp r10,byte ptr ss:[0x36]
jz 7f
mov r10,0x73
caesar encode r10,2
cmp r10,byte ptr ss:[0x37]
jz 74
mov r10,0x45
caesar encode r10,2
cmp r10,byte ptr ss:[0x38]
jz 69
mov r10,0x6d
caesar encode r10,2
cmp r10,byte ptr ss:[0x39]
jz 5e
mov r10,0x72
caesar encode r10,2
cmp r10,byte ptr ss:[0x3a]
```

```
jz 53
mov r10,0x52
caesar encode r10,2
cmp r10,byte ptr ss:[0x3b]
jz 48
mov r10,0x66
caesar encode r10,2
cmp r10,byte ptr ss:[0x3c]
jz 3d
mov r10,0x63
caesar encode r10,2
cmp r10,byte ptr ss:[0x3d]
jz 32
mov r10,0x44
caesar encode r10,2
cmp r10,byte ptr ss:[0x3e]
jz 27
mov r10,0x6a
caesar encode r10,2
cmp r10,byte ptr ss:[0x3f]
jz 1c
mov r10,0x79
caesar encode r10,2
cmp r10,byte ptr ss:[0x40]
jz 11
mov r10,0x65
caesar encode r10,2
cmp r10,byte ptr ss:[0x41]
jz 06
mov buff,imm
nop
```

mov buff,imm

nop

像这种伪代码不能运行，所以说实际比赛中还是分析好了像前一个方法一样，不一句一句翻译，直接写脚本解题比较快

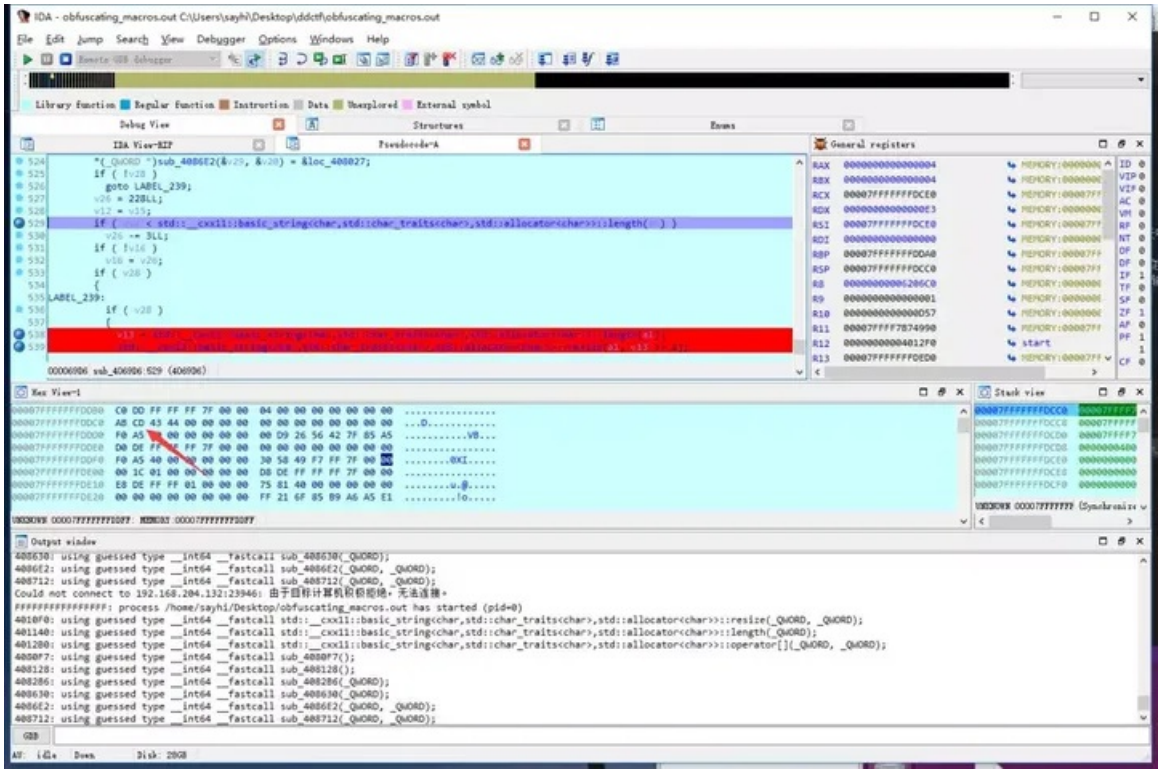
02Obfuscating macros

ubuntu64 + gdbserver + ida64调试环境

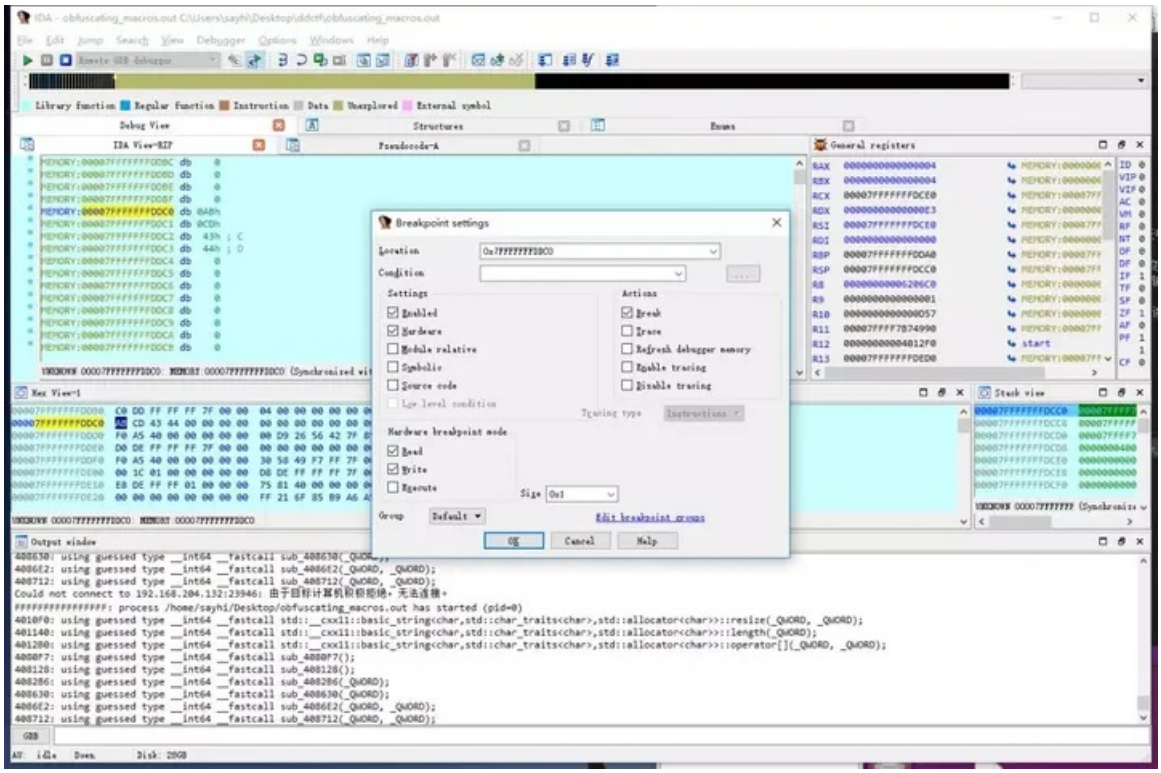
首先main函数里关键的函数只有两个，一个sub4069D6一个sub4013E6，先跟进第一个函数，在所有传入我们输入的变量的地方下断点 然后配置好远程gdb调试器，f9运行起来，随手输入ABCD，ida中触发断点，首先先观察寄存器的值，在堆栈中找到我们输入的字符串

然后一路f9观察对我们输入的字符串做了什么

f9几次后会发现他把我们输入的ABCD转换成了0xABCD



继而我们推测下一个函数会对0xABCD进一步处理或者与一些值进行比较，于是删除所有其他的断点，在0xABCD处下内存读写断点，然后f9运行，就触发了硬件断点



第一次 test al,al是检测输入是否为空，再按f9触发第二次断点，会发现如下代码

text:0000000000405FA3 loc_405FA3:

.text:0000000000405FA3 mov rax, [rbp+var_220]

.text:0000000000405FAA lea rdx, [rax+1]

.text:0000000000405FAE mov [rbp+var_220], rdx

.text:0000000000405FB5 movzx edx, byte ptr [rax]

.text:0000000000405FB8 mov rax, [rbp+var_210]

.text:0000000000405FBF movzx eax, byte ptr [rax]

.text:0000000000405FC2 mov ecx, eax

.text:0000000000405FC4 mov eax, edx

.text:0000000000405FC6 sub ecx, eax

.text:0000000000405FC8 mov eax, ecx

.text:0000000000405FCA mov edx, eax

.text:0000000000405FCC mov rax, [rbp+var_210]

.text:0000000000405FD3 mov [rax], dl

.text:0000000000405FD5 mov rax, [rbp+var_280]

.text:0000000000405FDC test rax, rax

.text:0000000000405FDF jnz short loc_

f8到0000000000405FC6处会发现他拿我们的0xAB与一个0x79进行了对比，这样的话我们只需要在sub这里下一个断点，一路f9就好了 他一开始没有验证长度，直接进行了对比，只要对比到不同就退出，因为我们输入了ABCD，观察到他对比一次就会退出了 这样我们就可以每次多输入两个字符，并且这两个字符只能是0到9和A到F，然后取出新的对比的字符再多输入两个字符直到他不在对比就得到flag了

这道题目还是说，如果没有先调试得到的知识背景，用pintool解题也会卡壳的，因为经过我们的分析，发现第一个函数把字符串ABCD转换成了0xABCD，这里四个字节变成了两个字节，如果用pintool解题需要两个字符两个字符得组合得测试，这个题目不是纯正的ollvm混淆的，大体的思路就是污点追踪，把所有的涉及到用户输入的地方都下断点，还有就是内存断点的使用，在必要时刻是很有用的。

03题目链接

链接：https://pan.baidu.com/s/1yx9RlgITi5rjcHGBgZ1DDA 提取码：ql9p

04相关操作学习

CTF-Reverse系列汇总：从最简单的逆向分析开始，循序渐进地讲解关键代码定位、算法分析、手工脱壳、病毒分析等实验，同时详细介绍了常见逆向分析工具的使用方法，点击课程:CTF-Reverse系列汇总(合天网安实验室)开始操作学习！

声明：笔者初衷用于分享与普及网络知识，若读者因此作出任何危害网络安全行为后果自负，与合天智汇及原作者无关，本文为合天原创，如需装载，请注明出处！