xctf刷题随笔

原创

Lu1u~ 于 2021-07-26 20:43:59 发布 62 收藏 1

XCTF-RE-hackme and BABYRE

- hackme
- BABYRE
- debug

hackme

1、查壳

64位elf文件，拖入IDA中打开

2、IDA分析

函数窗口发现全是sub_开头，没有解析出函数名。

试着shift+f12查看字符串 发现give me the password字符串，根据调用，跟进函数。

```
14  int i; // [rsp+BCh] [rbp-4h]
15
16  printf((__int64)"Give me the password: ");
17  scanf((__int64)"%s", v7);
18  for ( i = 0; v7[i]; ++i )
19    ;
20  length = i == 22;                           // 输入长度为22
21  v16 = 10;
22  do
23  {
24    v13 = (int)sub_406D90() % 22;             // 需要了解v13的变化
25    v15 = 0;
26    v12 = byte_6B4270[v13];
27    v11 = v7[v13];                            // flag有22位 而只是循环了10次  但是每次异或的v15都与下标有着密切的关系 v15与v13+1有关
28                                              // 所以可以试着循环22次 每次异或 发现结果就是flag
29    v10 = v13 + 1;
30    v14 = 0;
31    while ( v14 < v10 )
32    {
33      ++v14;
34      v15 = 1828812941 * v15 + 12345;
35    }
36    v9 = v15 ^ v11;
37    if ( v12 != ((unsigned __int8)v15 ^ v11) )
38      length = 0;
39    --v16;
40  }
41  while ( v16 );
42  if ( length )
43    v8 = printf((__int64)"Congras\n");
44  else
45    v8 = printf((__int64)"Oh no!\n");
46  return 0LL;
00000F8E sub_400F8E:36 (400F8E)
```

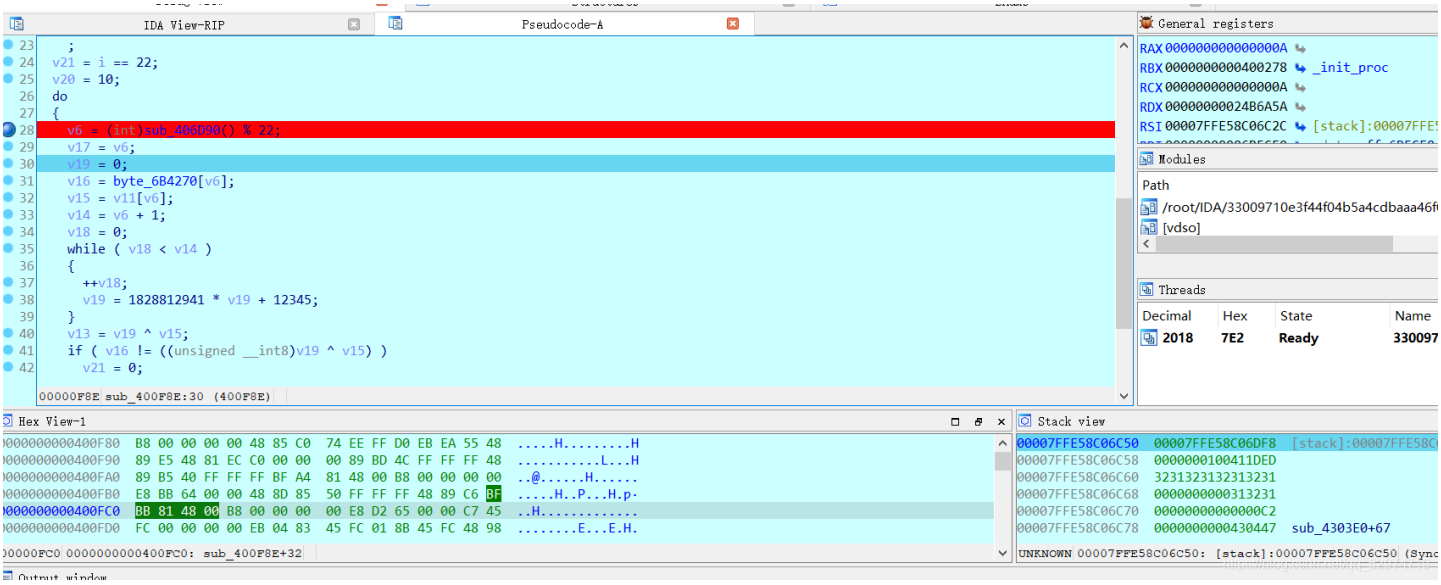
分析函数逻辑，得出v7是我们的输入，即flag并且长度为22位，设定v6等于10，进行do while循环，可知主要操作是v15和v11异或后与同下标的v12进行比较，而v15的生成算法与v13即当前下标有关，所以如果能知道v13的变化就能逆向出输入了。跟进sub_406d90()函数，逻辑很复杂，尝试动调出v3；

```
__int64 sub_440240()
{
  int v2[3]; // [rsp+Ch] [rbp-Ch] BYREF
```

```
  _ESI = 1;
  if ( !dword_6B7E1C )
  {
    __asm { cmpxchg cs:dword_6B7484, esi }
    if ( !dword_6B7E1C )
      goto LABEL_5;
    goto LABEL_10;
  }
  if ( _InterlockedCompareExchange(&dword_6B7484, 1, 0) )
LABEL_10:
    sub_432E10(&dword_6B7484, 1LL);
LABEL_5:
  sub_4404A0(&off_6B5CE0, v2);
  if ( dword_6B7E1C )
  {
    if ( !_InterlockedDecrement(&dword_6B7484) )
      return v2[0];
    goto LABEL_11;
  }
  if ( --dword_6B7484 )
LABEL_11:
    sub_432E40(&dword_6B7484);
```

00040240 sub_440240:1 (440240)

注意:这里因为重新载入一下，下面图片的v6即上面所求的下标v13



通过不断F8和多次尝试发现，v13=[0x11,0xa,0x11,0xd,0x1,0xf,0x0,0x6,0x3,0x1]

分析得知，该程序只是对输入的0、1、3、6、10、13、15、17进行检测，这样的话动调check就可能出错，不是flag也会出现congratulation！！！

换个思路，加密是与位置下标相关的v15和 flag中的值异或==v12，反解一下即可，加密就是一个异或。

```
a=[0x5F, 0xF2, 0x5E, 0x8B, 0x4E, 0x0E, 0xA3, 0xAA, 0xC7, 0x93,
  0x81, 0x3D, 0x5F, 0x74, 0xA3, 0x09, 0x91, 0x2B, 0x49, 0x28,
  0x93, 0x67]
print(len(a))


def kk(n):
    v10=n+1
    v14=0
    v15=0
    while v14 < v10:
        v14=v14+1
        v15 = 1828812941 * v15 + 12345
    return v15
flag=''
for i in range(len(a)):
    flag+=chr((kk(i)^a[i])&0xff)  #防止超出限制  注意优先级
print(flag)

m=['a']*22  #试一试刚刚自己的动调出的下标
m[0]=flag[0]
m[1]=flag[1]
m[3]=flag[3]
m[6]=flag[6]
m[10]=flag[10]
m[13]=flag[13]
m[15]=flag[15]
m[17]=flag[17]
for i in m:
    print(i,end='')
```

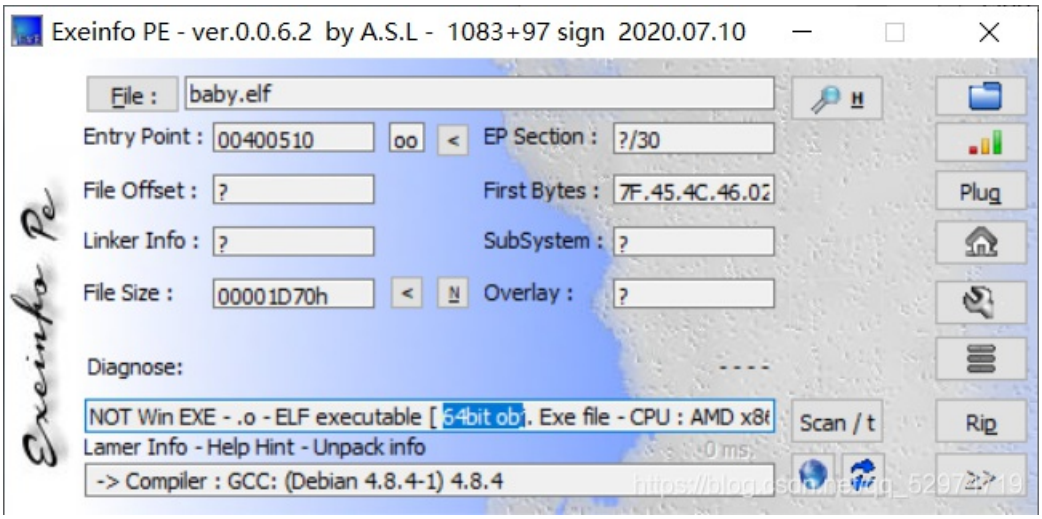
通过第一段的代码，较容易的能拿到flag，同时也尝试了下动调结果。

正如猜想，check函数只是检测了几位而已，一开始还认为是一个随机数检测，那样的话还更可能能保证flag的正确。

总结：不要过于专研某个变量的值，合理的猜测加上对加密的主要操作的简化才能快速拿到flag，本题中的检测也可以理解为是抛砖引玉的过程。

BABYRE

1、查壳

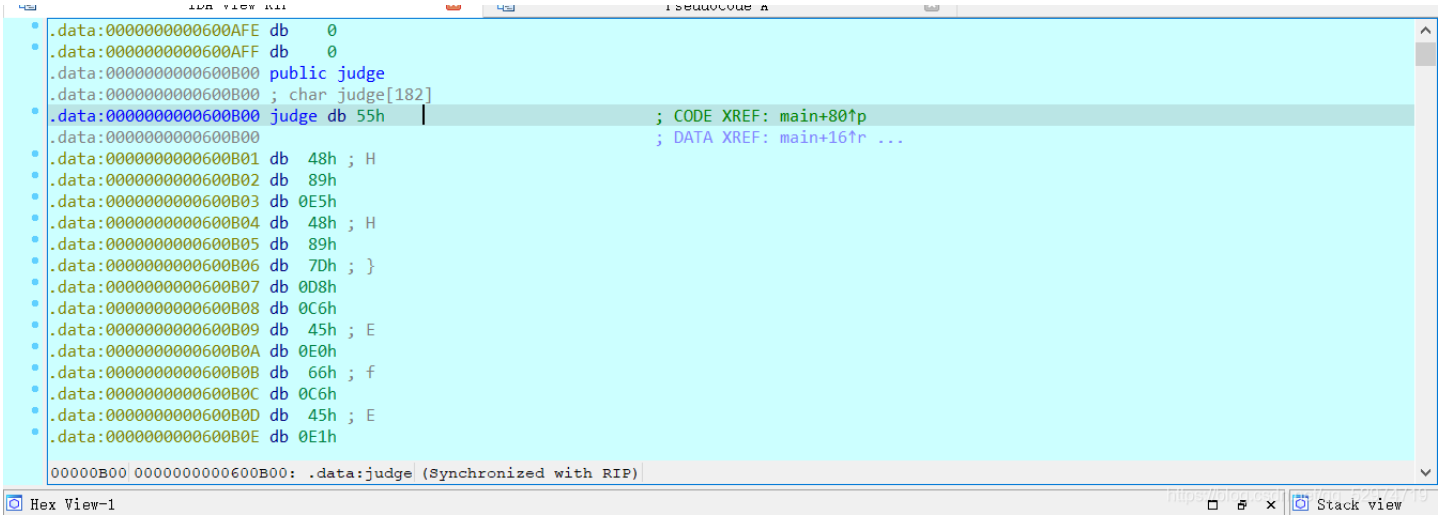


64位elf文件，拖入IDA中静态分析

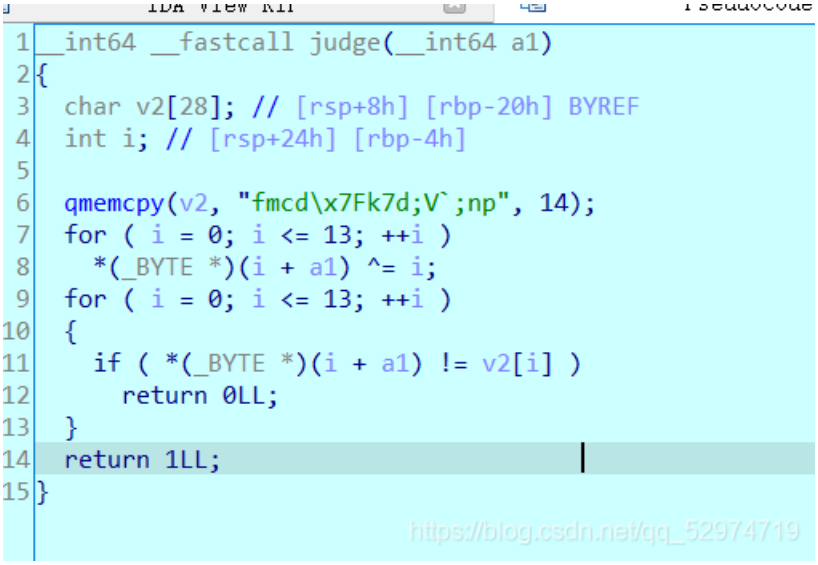
2、IDA静态分析

```
{
  char flag[24]; // [rsp+0h] [rbp-20h] BYREF
  int v5; // [rsp+18h] [rbp-8h]
  int i; // [rsp+1Ch] [rbp-4h]

  for ( i = 0; i <= 181; ++i )
    judge[i] ^= 0xCu;
  printf("Please input flag:");
  __isoc99_scanf("%20s", flag);
  v5 = strlen(flag);
  if ( v5 == 14 && (*(unsigned int (__fastcall **)(char *))judge)(flag) )// judge是个check函数
    puts("Right!");
  else
    puts("Wrong!");
  return 0;
}
```


函数逻辑非常简单，输入是flag，要求flag长度为14并且通过judge函数，刚开始judge是一个数组，数组与0xc进行异或，之后在judge前有_fasrcall，查阅资料知是一种函数调用协议，那么异或后judge存储的就是一个函数的数据，所以IDA动调一下 P 看一下Judge函数即可。



IDA远程调试断下来后，点击Judge跟进，在judge开始地址处 按 C 键转换为c代码，之后按 P 键转换为函数，之后便可F5查看c的伪代码



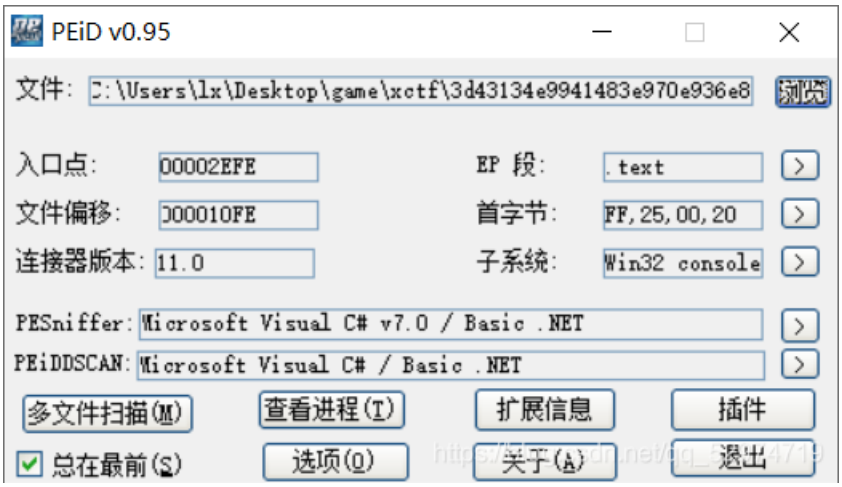
```
__int64 __fastcall judge(__int64 a1)
{
  char v2[28]; // [rsp+8h] [rbp-20h] BYREF
  int i; // [rsp+24h] [rbp-4h]

  qmemcpy(v2, "fmcd\x7Fk7d;V`;np", 14);
  for ( i = 0; i <= 13; ++i )
    *(_BYTE *)(i + a1) ^= i;
  for ( i = 0; i <= 13; ++i )
  {
    if ( *(_BYTE *)(i + a1) != v2[i] )
      return 0LL;
  }
  return 1LL;
}
```

逻辑很简单就是，flag与下标逐个异或后与v2比较，将v2与下标逐个异或后便得到flag。

总结：本题逻辑其实很简单，主要就是把函数转换为数据之后异或存储的方式值得学习，同时要善于发现一些如_fastcall等一些明显的函数标志。

debug

1、查看程序架构



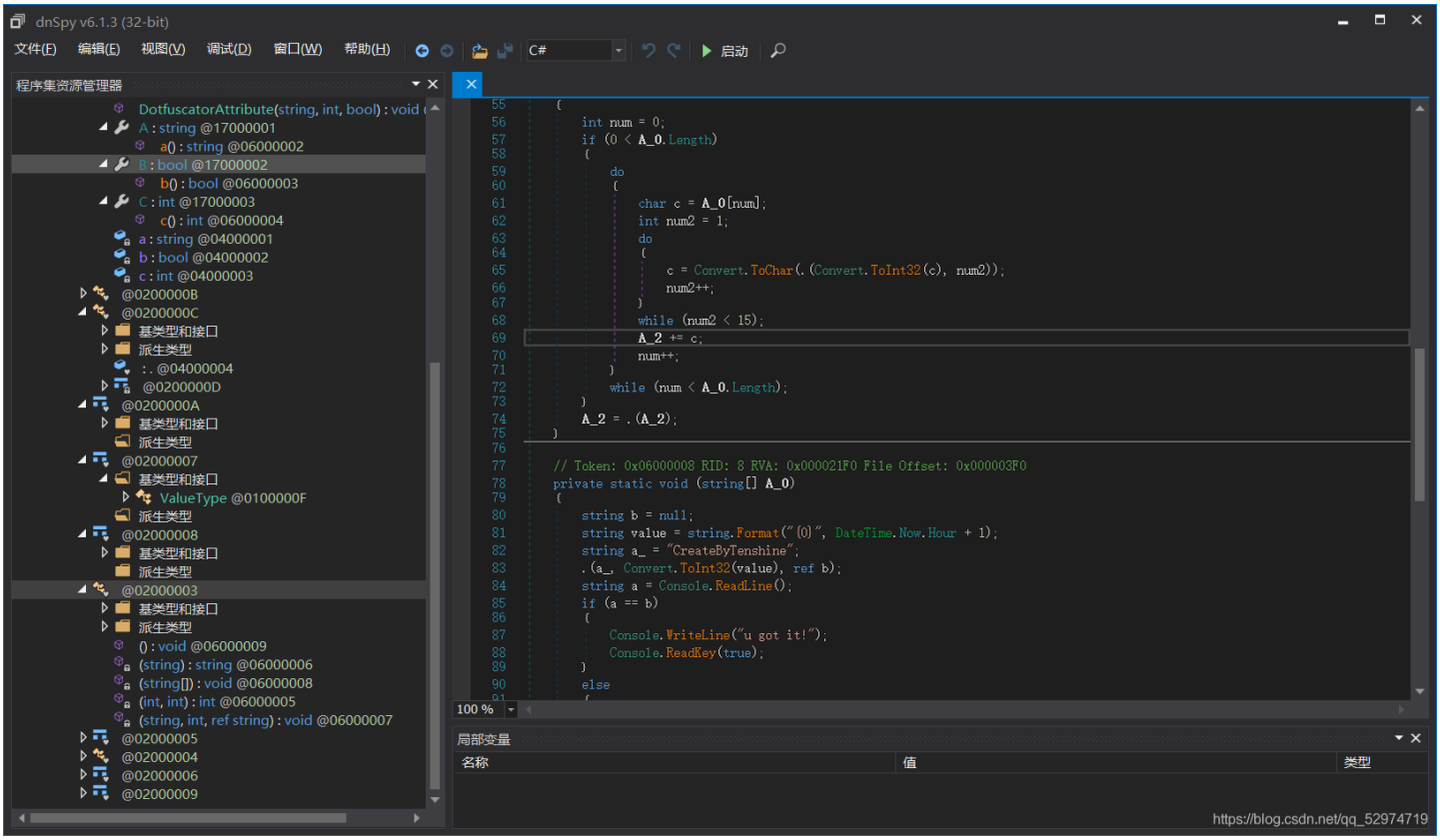
32位PE文件，用c#编写，.NET程序逆向

选择合适的工具，比如.NET reflector，ILSpy 或者 dnSpy

2、动态调试

这里我选择dnSpy，该软件的排版和风格十分美观，并且支持动调，类似IDA。

注意：用dnSpy32打开PE文件



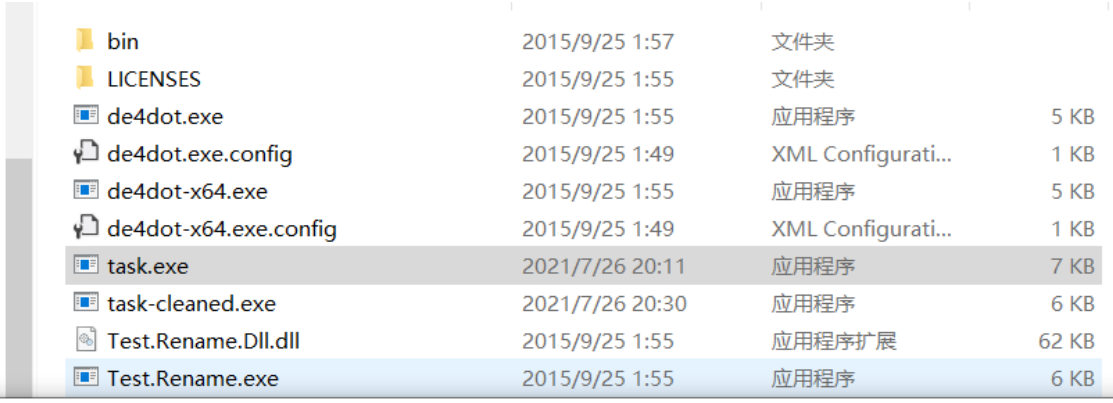
打开后发现并没有彻底解析出函数名称并且感觉变量名称都有些奇怪，猜测可能是.NET混淆。

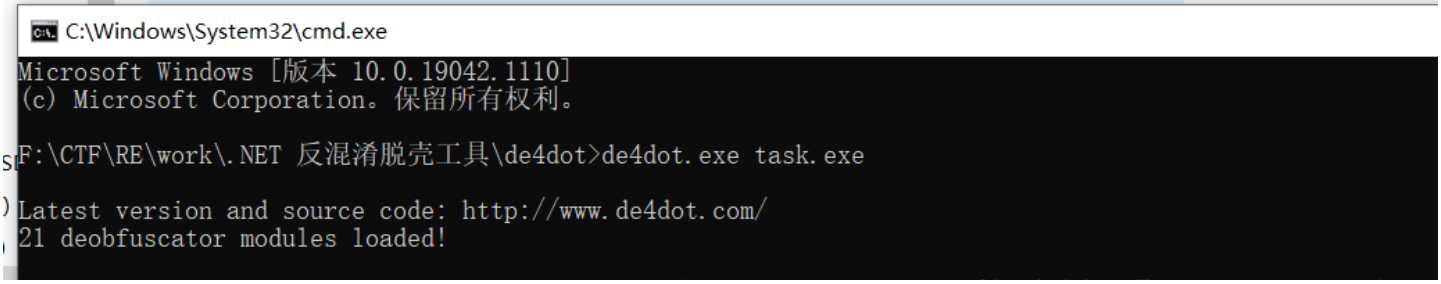
补充：为了保护代码，开发者可以运用混淆技术对代码进行保护。

在混淆中最简单的就为名称混淆，将一些有意义的变量名换成难以理解的a,b,c等或者让函数名称看起来十分的混乱。这和本题的情形有些类似。

破解方法：对于一些简单的名称混淆其实无关紧要，函数的主体内容并没有改变。如果让程序更加可读，可以使用de4dot工具进行.NET反混淆。

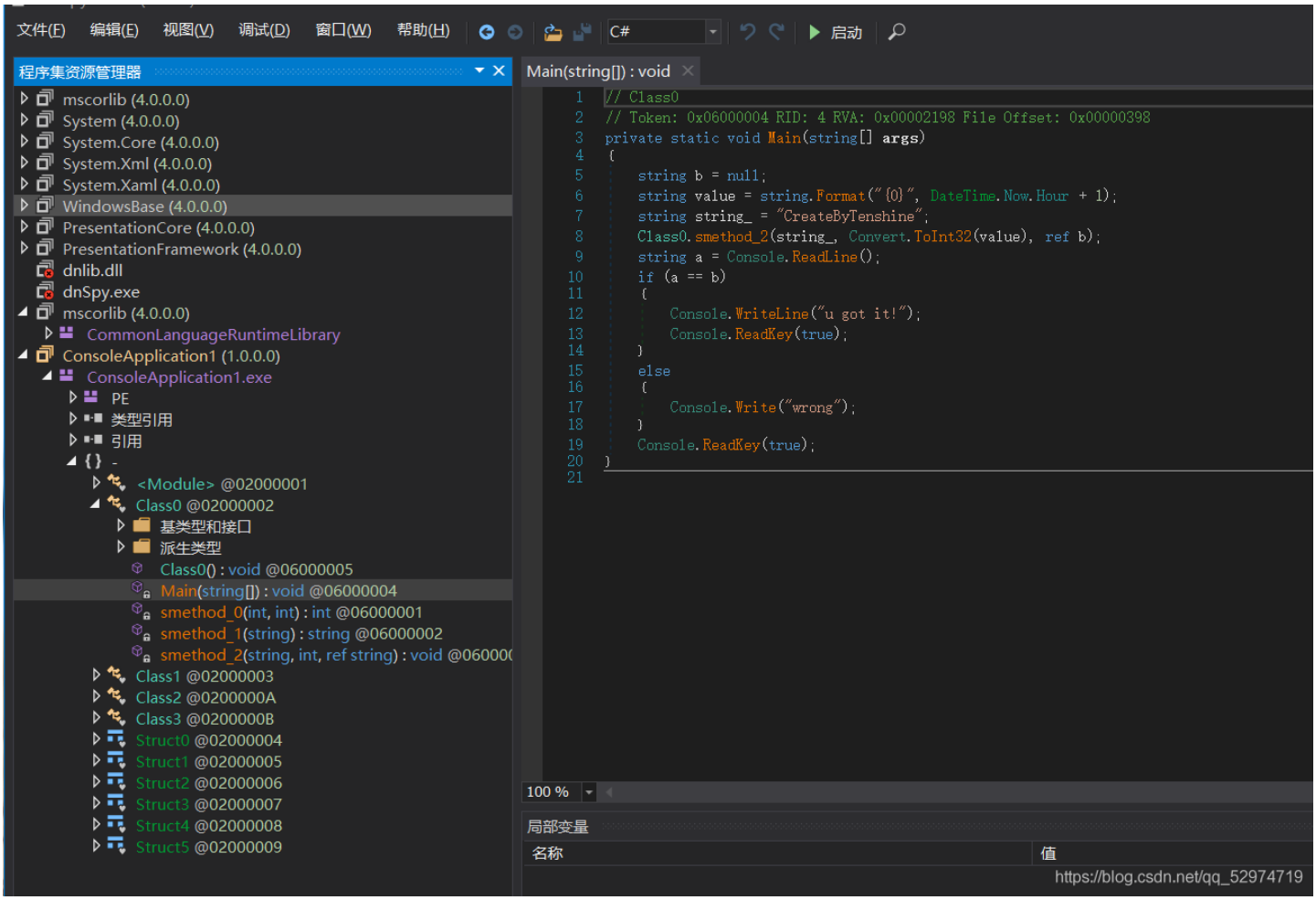
放





```
Detected Dotfuscator 124576:1:1:4.9.6005.29054 (F:\CTF\RE\work\.NET 反混淆脱壳工具\de4dot\task.exe)
Cleaning F:\CTF\RE\work\.NET 反混淆脱壳工具\de4dot\task.exe
Renaming all obfuscated symbols
Saving F:\CTF\RE\work\.NET 反混淆脱壳工具\de4dot\task-cleaned.exe

F:\CTF\RE\work\.NET 反混淆脱壳工具\de4dot>
```

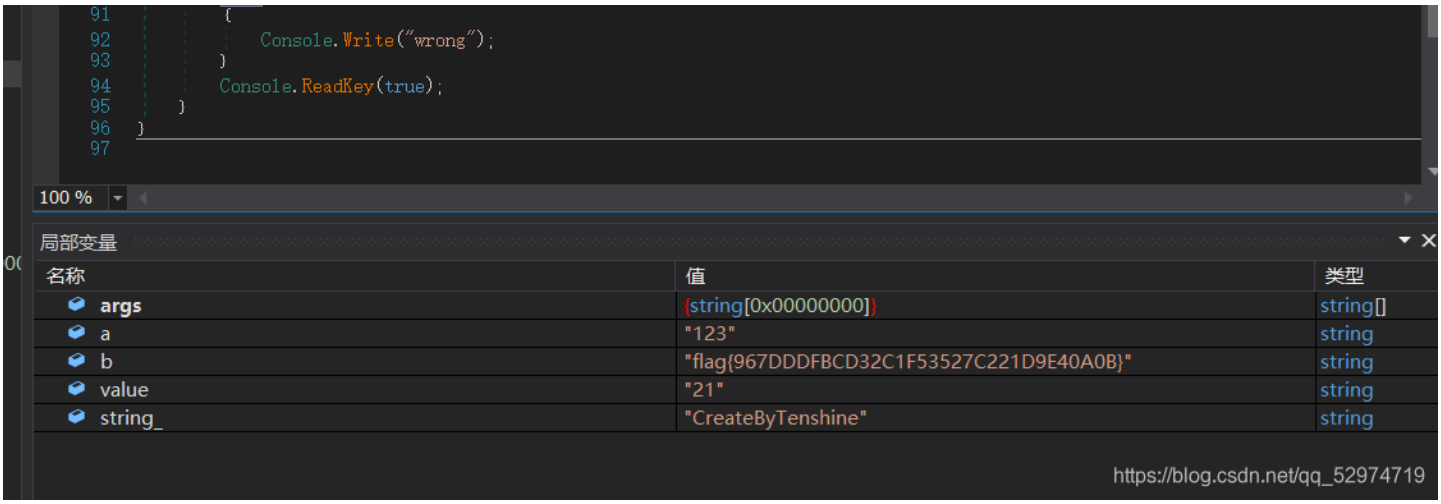

将文件copy到de4dot的目录下，选择合适的版本在cmd执行de4dot.exe + filename 即可生成反混淆后的文件 name-cleaned.exe，之后用dnSpy分析。



打开后发现，main函数和其他一些函数都给分析好了，接着看flag的逻辑。

根据主函数可知 Readline是输入字符串 即a是我们的输入，让a和b进行比较如果相等就成功。

所以主要关心b的生成，观察知，经过一系列的变化，把函数生成的字符串存入b中，静态逻辑看上去很复杂，不过生成b与我们的输入无关，同时结合题目debug，因此需要动调。



在b生成后的某一位置打上断点，运行，随便输入a，断下后在变量窗口便能看到b的内容即为flag。

总结：题目本身逻辑不难，主要涉及了.NET混淆和动态调试，需要用的工具比较多，同时在输入不干扰其他函数运行时动调必将是首选的方法，可以快速获取flag。