bugku 不好用的CE WriteUp

转载

baoju9513 于 2019-08-14 18:32:39 发布 221 收藏
原文链接：http://blog.51cto.com/13992485/2429569

不好用的CE

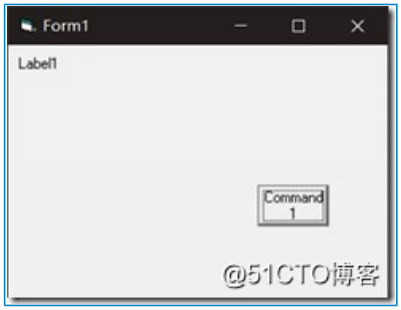
这题有好多种解法，我会一个个解释。

只用OD

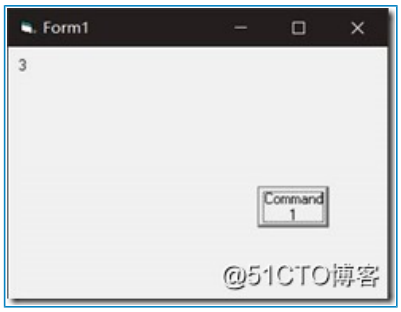
只用CE

CE+OD

下载文件



点两下试试

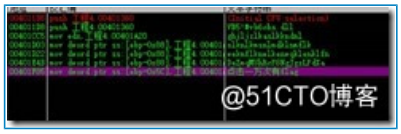


无壳

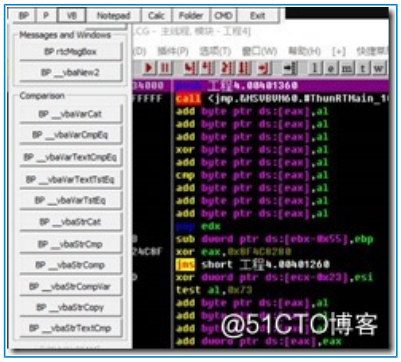
1.只用OD

只用OD我只想出两种办法，虽然只是下断点的方式不同，但也代表了不同的思路。

一、 第一种是最直接的也是最笨的，在搜索字符串里的所有内容都下断点，这里幸好搜索的字符串不是很多，而且flag凑巧是直接存储在内存里的，所以可以使用。若没这么幸运的话就只能在提示的字符串“点击一万次有flag”处下断点，一点点的往上翻代码了。

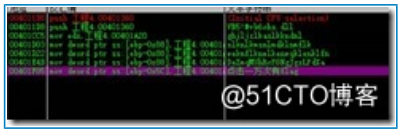


二、 我们知道了这个程序使用VB写的，且会弹出一个对话框，对话框在VB理常用的函数为rtcMsgBox，可以用ODB的插件来自动下断点，

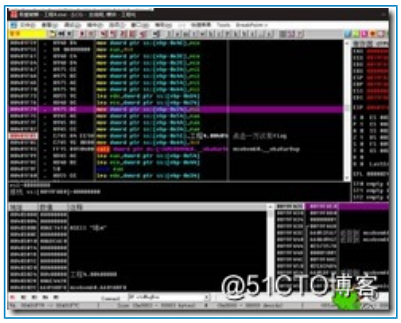


只用OD第一种办法：在每一处字符串下断点，这里我们就只在可疑字符串下断点了

即第3、4、5、6、7行

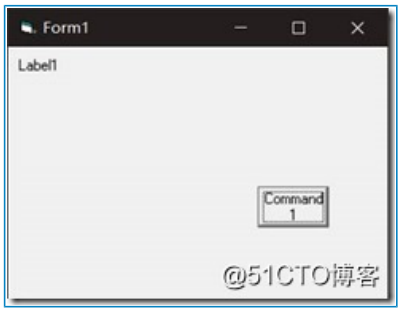


首先断在了最后一条，

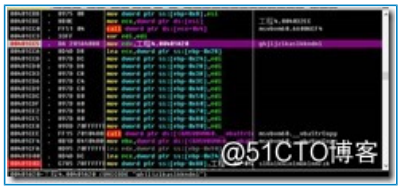


看起来是在初始化变量，F9继续运行

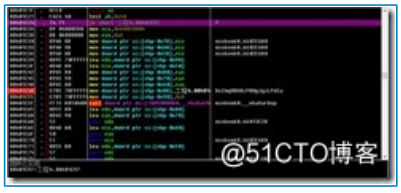
跳出对话框，单击确定继续程序



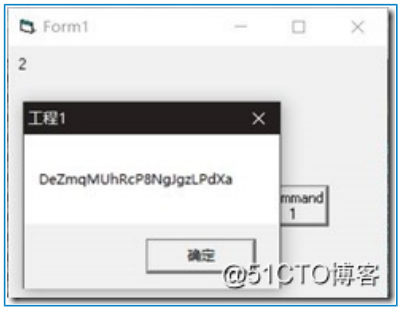
点击后触发断点，



这三个可疑的字符串在一起，但是其上并没有大跳转，甚至根本没有跳转，那最后一个可疑字符串就更可疑了，继续F9运行程序，程序并没有断在最后一个字符串处，说明这个字符串很有可能就是达到条件(点击一万次)后才会出现的字符串，



这个字符串上面还有一个大跳转，那这个字符串就很有可能是flag了，或者flag相关的字符串，我们把这个大跳转nop掉，看看会出现什么



DeZmqMUhRcP8NgJgzLPdXa

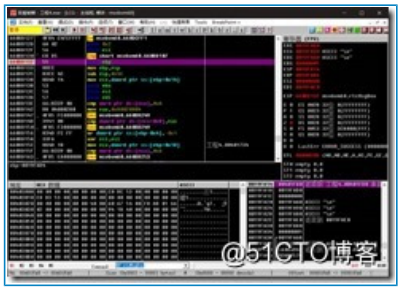
这题最坑的地方也就是这个字符串就出现了，之前做这个题也是断在这里，直到最近出了Writeup才知道，这个看起来像base64的字符串其实是他的远亲，base58

我们都知道base64的范围是 数字(10)+大小写字母(26*2=52)+两个特殊字符(+，/)

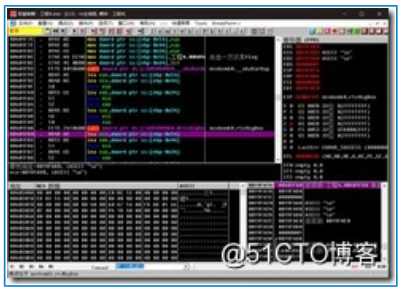
而base58是剔除了容易被人误识别的数字0，L的小写，i的大写和o的大写，还有两个特殊字符(+，/)

得到flag

只用OD第二种方法，利用OD的插件，在rtcMsgBox下断点，F9运行程序，被断下



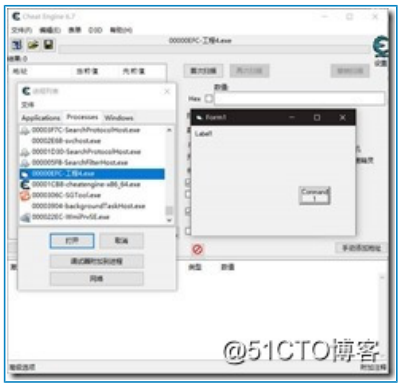
这个地方已经不属于程序的领空了，这里是VB调用的库的领空，在这个位置我们在栈里可以找到程序调用函数的地址，在其上回车以回到程序领空



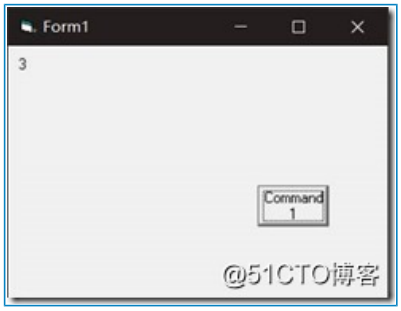
然后我们就可以苦哈哈地慢慢往上翻代码了，这个下断点的方式适合在没有明确的提示字符串的时候使用，在有提示字符串的时候还是用字符串来查找比较方便。

CE+OD

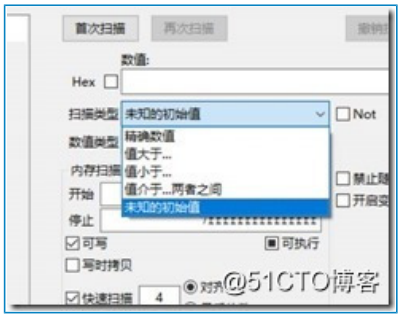
打开程序，CE附加程序，



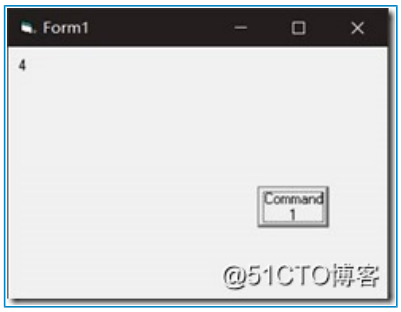
这里我们不知道这个变化的数字的类型，虽然看起来很像整型



所以我们设置扫描类型为未知的初始值，点击首次扫描

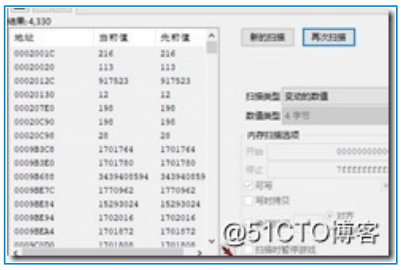


然后点击按钮，变化一下数值



再用CE搜索变化的数值

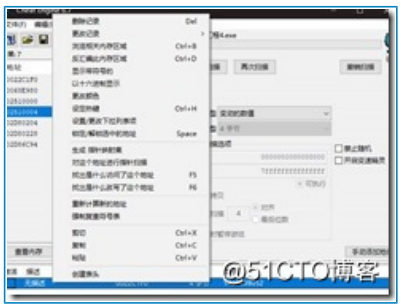
点击再次扫描



这样太慢了，我们可以用变动的数值和未变动的数值切换来不断搜索

最后剩了八个结果实在分辨不出来了

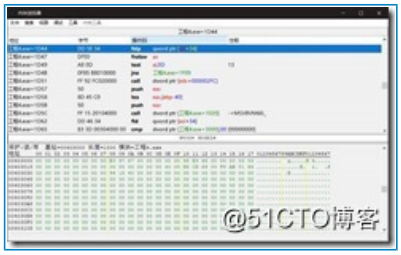
不过这就够了，我们也不需要知道那么细致，随便选一个，双击，拉到下面的界面里，右键他选择 找出是什么改写了这个地址



注意像这样的，地址特别大的，一定不是程序的代码，这个是程序调用的库的地址

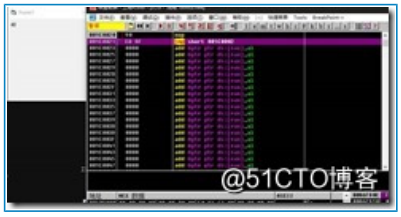
像这样40打头的才是程序的代码，具体要看程序的PE头里定义的基地址，一般为400000。

然后我们就可以记住这个地址，用OD打开程序，到这个地址看看，CE也可以看，但是很多操作不方便

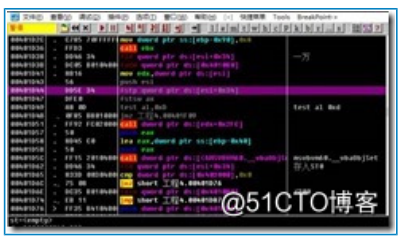


毕竟不是专门用来调试程序的应用。

我们用OD附加到进程上，



ctrl+g 到401D44看看



距离我们第一次找到的关键跳转也很近，

这之间有大量的棕色的浮点数运算，而关键跳转之后再无浮点运算，所以这可能就是算法部分，这次我们仔细分析下算法部分，

这里可以说是算法部分最重要的四条代码了，从0x4010A8存储的10000就能看出来，在我解释浮点助记符之前，我要先解释一下浮点运算：

在包含浮点运算的处理器里，有8个寄存器，分别是ST0-ST7，他们通过浮点助记符来进行浮点运算，他们的使用方法与栈很类似，存储的顺序从ST0开始到ST7，常用的浮点助记符有：

> fld 相当于push

> fstp 相当于pop

> fadd 相当于add

> fsub 相当于sub

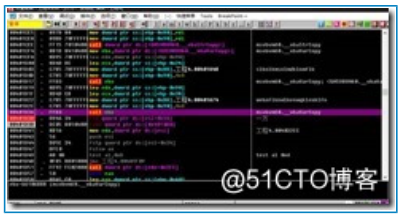
> fdiv 相当于div

> fmul 相当于mul

> fstsw 把状态寄存器存入寄存器里

> fcomp 相当于cmp

再具体点的用法我会在用的时候解释，现在在最开始的浮点运算处下断点



因为代码跨度有点大，我就不一一截图了，只把关键代码写下来

```
fld qword ptr ds:[esi+0x34]
```

把从[esi+0x34]存入ST0

```
fadd qword ptr ds:[0x4010B0]
```

0x4010B0是200.0，即ST0+=200.0

```
fstp qword ptr ds:[esi+0x34]
```

即[esi+0x34] = ST0

```
fstsw ax
```

把状态寄存器存入ax，周围并没有可以影响到状态寄存器的代码，所以忽略就行

```
fld qword ptr ds:[esi+0x34]
```

即ST0=[esi+0x34]

fdiv qword ptr ds:[0x4010B0]

即ST0/=200.0

```
fstp qword ptr ss:[esp]
```

即[esp]=ST0，这里存储的就是实际的点击数了

```
fclex
```

查了一下是叫做浮点检查错误清除，不会影响结果所以忽略

```
fld qword ptr ds:[esi+0x34]
```

即ST0=[esi+0x34]，

```
fdiv qword ptr ds:[0x4010B0]
```

即ST0/=200.0

```
fcomp qword ptr ds:[0x4010A8]
```

即ST0与10000比较

```
fstsw ax
```

把状态寄存器存入ax

```
test ah,0x40
```

比对状态寄存器，

```
je 401e97
```

关键跳转

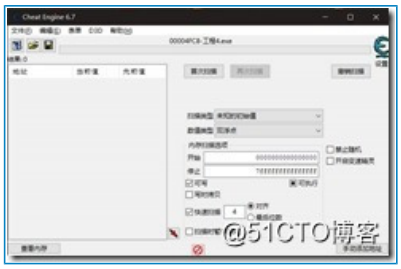
然后怎么改就看个人喜欢了，可以像上次一样直接nop掉关键跳转，也可以修改0x4010B0里的值来达到点一次等于数次的效果，也可以直接修改0x4010A8里的值，让一万次变成1次。flag处理部分不再赘述。

后来我查了一下，test ah,0x40 比对的是状态寄存器的cf寄存器，即进位寄存器，所以他只会在从9999进位到10000时触发，

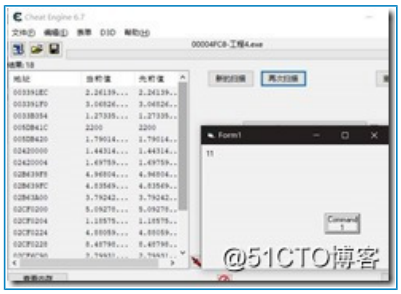
只用CE：

运行程序，用CE附加上

由于我们已经知道了数值的类型为双浮点(双浮点数占八个字节，有效数字16位，之前的200.0可以数一下有效数字就知道了，即使不知道类型为双浮点也可以一个个试，通常数据存储类型只有4字节，单浮点，双浮点类型，偶尔也有单字节的布尔类型)，我们设置扫描类型为未知的初始值，数值类型为双浮点搜索，

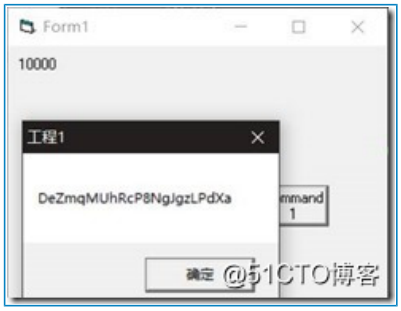


然后用 变动的数值/未变动的数值切换搜索，很快就搜索到了一个很扎眼的数值，除了这个2200都是后面跟了很多个小数的双浮点数，然后用2200/11得到增量200，



双击把他加入下面的界面，设置大小为1999800

然后点击程序的按钮



就从11变成了100000，从而得到flag

当然，如果我们知道了增量为200，也可以直接搜索200*X