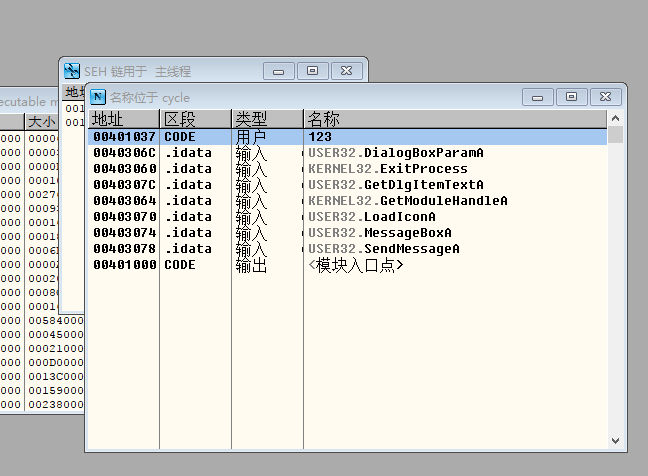
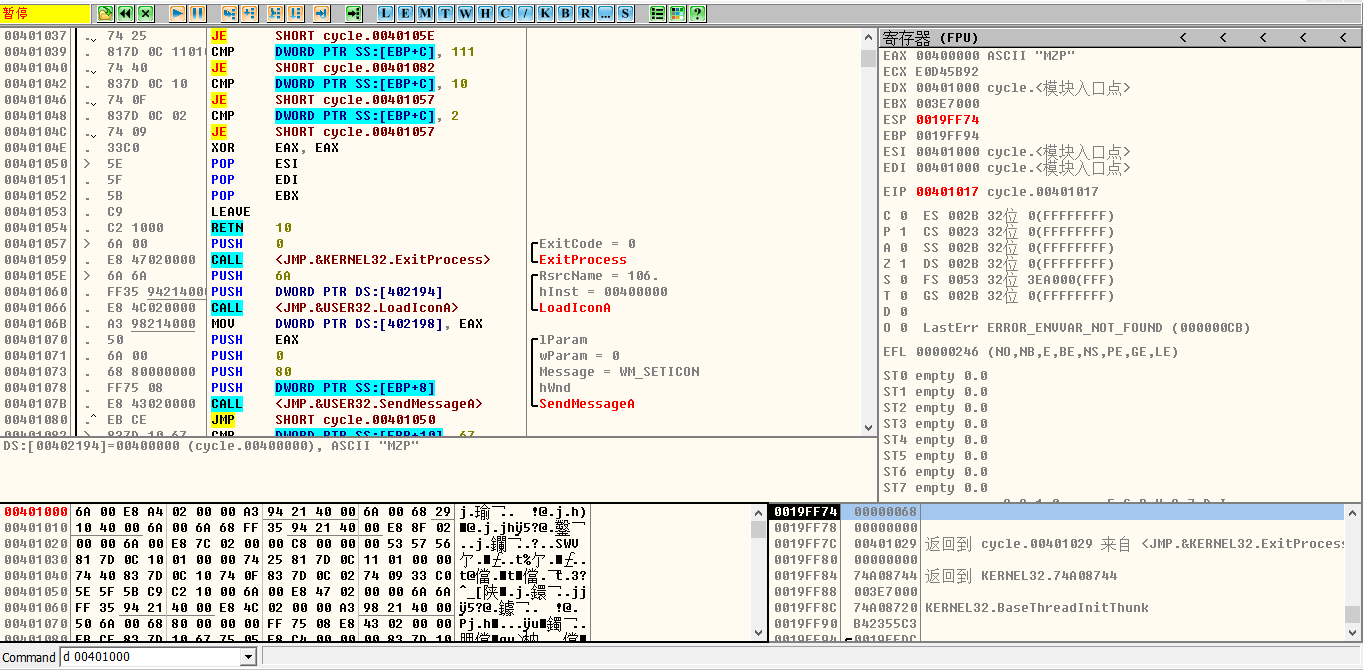
Ollydbg学习笔记 2

1. 查看输入表/输出表的快捷键是 ctrl+N



1. Ollydbg窗口中使用 d 0x12345678会在数据窗口跳转到指定的地址处，反汇编窗口处不会跳转。此外，命令行，可以实现图形界面的所有功能，如下断点，查看堆栈窗口等，因为ollydbg是可以编程的，也就说它提供一些api，命令行是直接调用api，而图形窗口是将 数据做了一些包装，转送给api殊途同归，有些像win32 提供编程 命令行 cmd和 gui界面的区别



1. 去对话框入口处,SHIFT+F4,在"条件"里输入 [ESP+8]==0111 && [ESP+C]==0422 的含义是

[esp+8] 需要做地址解析，解析地址后的数据是16位的，说明默认解析的地址是解析2个byte，解析后的返回值放置到al中。如果需要解析4字节 需要使用 dword ptr [esp+8]，这样就会解析一个32为的数据了，32位的返回值放置到eax中。

这里使用[esp+8] [esp+c]的含义是取参数的作用，这个方法之前调用的时候，之前可能调用其它的方法，传递方法的可能性有以下几种

1. 调用方法前很久，就将此方法的参数压栈

这样做的好处是 可以倒出寄存器供其他寄存器使用。

这样做的坏处是 占用的堆栈空间，和汇编代码可能有增长。

因为需要在栈中存更多的参数，和使用更多的代码维护栈逻辑。

1. 在方法调用前，才将压入栈。

这样做的好处是 减少栈控件的使用和维护栈控件的汇编代码

这样做的坏处是 没有合理的使用栈的存储结构，将方法合理压入栈中，比如说方法的参数可能很早就计算得出，但是方法有些参数是在方法前才能计算得出，这样就需要缓存，很早得出的参数。

缓存的方法有两种

1. 在数据区，使用一部分系统分配的数据空间缓存很早的参数
2. 将参数存入栈中。在方法的使用时，提前出栈到可用的寄存器。然后依据方法的压栈规则存入栈中。

一部分提前压栈，一部分临时压栈

这样做的好处是

1. 合理的利用栈的特性，堆栈平衡。
2. 减少了维护栈控件的维护代码

这样做的坏处是

1. 由于参数的计算顺序不固定，导致应该最后压栈的参数，在很早的时候被计算出。所以如果采用这种方法则会增加栈维护的代码
2. 相对于临时压阵的方法1，增加了栈空间的大小，但是，方法1是占用了数据区的空间，所以综合来看没什么区别。可能由于栈是顺序存储的，而数据区的存储方式不确定，会有一些区别

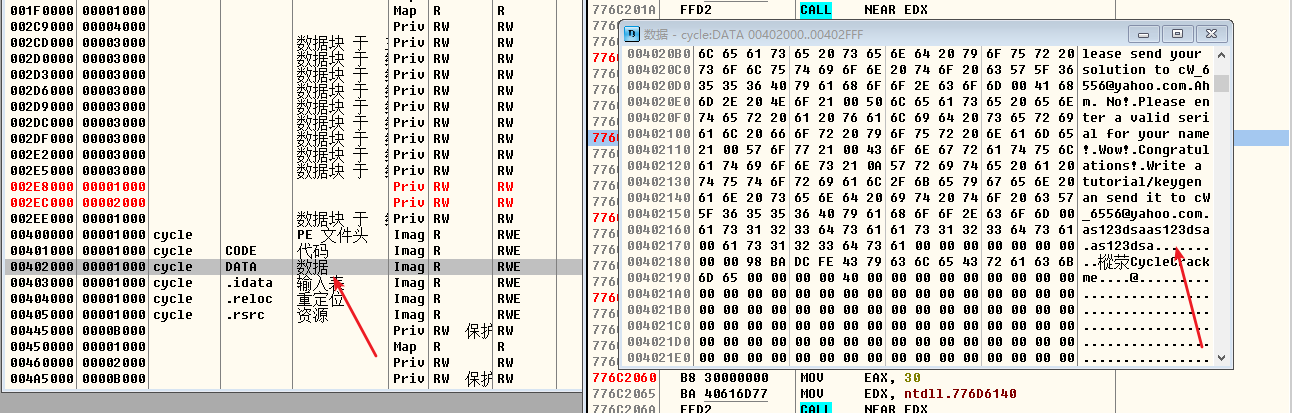
总的来看，当实际存储的参数的时候，应该是 根据不同的方法，在编译器编译的阶段，合理的使用三种方式进行堆栈分配。

关于ollydbg中内存查找数据的操作步骤

1. 使用F9将程序运行起来
2. 打开注册界面，输入一些数据
3. 在ollydbg中使用快捷键 alt+m，打开内存窗口
4. 在内存窗口，右键 查找 输入二进制字符串 ，输入对应的ascii和hex码

注意实现

1. 输入的特征数据最好是ascii码。如果输入中文，ascii码无法解析，但是可以使用hex进行 查找。
2. 当在程序没有运行的时候打开内存窗口的时候，f9，可以看到在程序运行起来的时候。内存窗口中生成了一些新的区段。
3. 当调试内存的时候



关于调试程序是f12 和f9的使用

1. 当程序运行后，使用F12让程序陷入中断，ollydbg则会调到这个窗口调用的dll中，多半是系统领空，但是也有可能是用户领空，因为用户可以自己实现响应的代码。

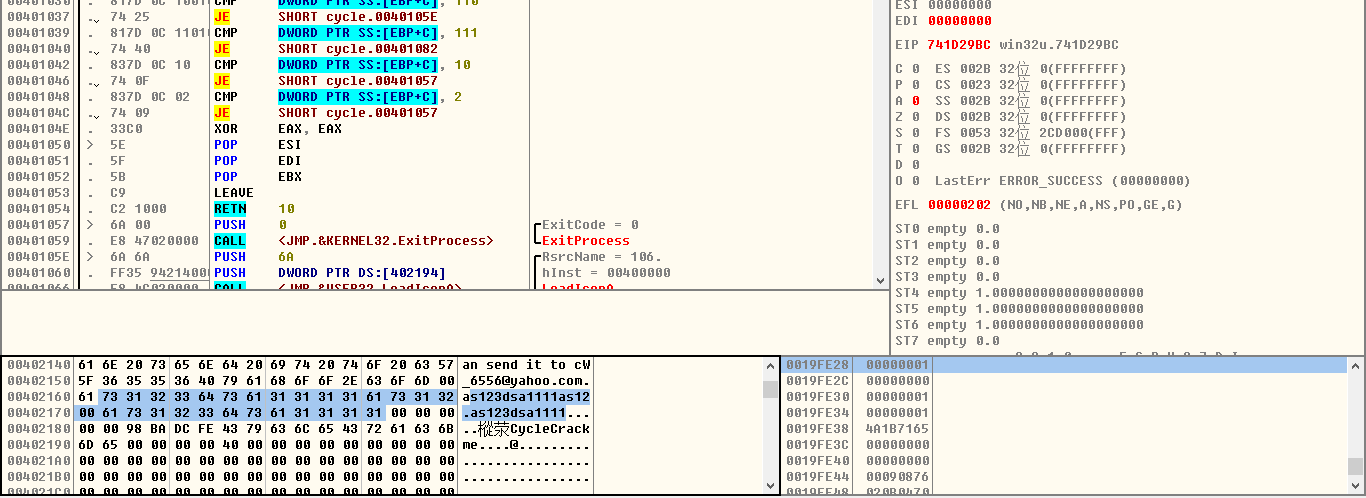
程序此时的状态是虽然仍是图形界面，但是无法点击，无法关闭

1. 当程序陷入中断，之后可以使用F9，恢复程序运行，

程序此时的状态是程序正常了。

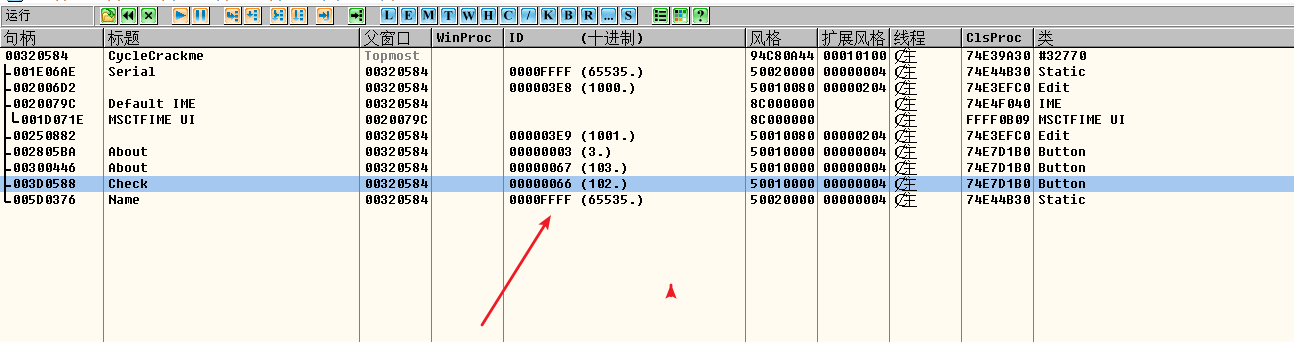
注意的是如果识图调试系统代码，程序可能仍然陷入中断，解决的方法是 按住F9,直到ollydbg左上角出现运行提示，这是程序就正常了。此时反汇编窗口仍然处于系统领空

1. 如果想要回到用户领空，操作步骤是 右键 查看 模块 用户程序的模块如 cycle的字样，此时的程序仍然正常运行，输出的测试数据仍然可以正常更新。



关于ollydbg消息断点和run trace 的体会

1. 关于控件模块的id查看



注意的是这里的id是16进制的，而使用id的时候也是16进制的，

1. 右键 分析 假定参数的使用



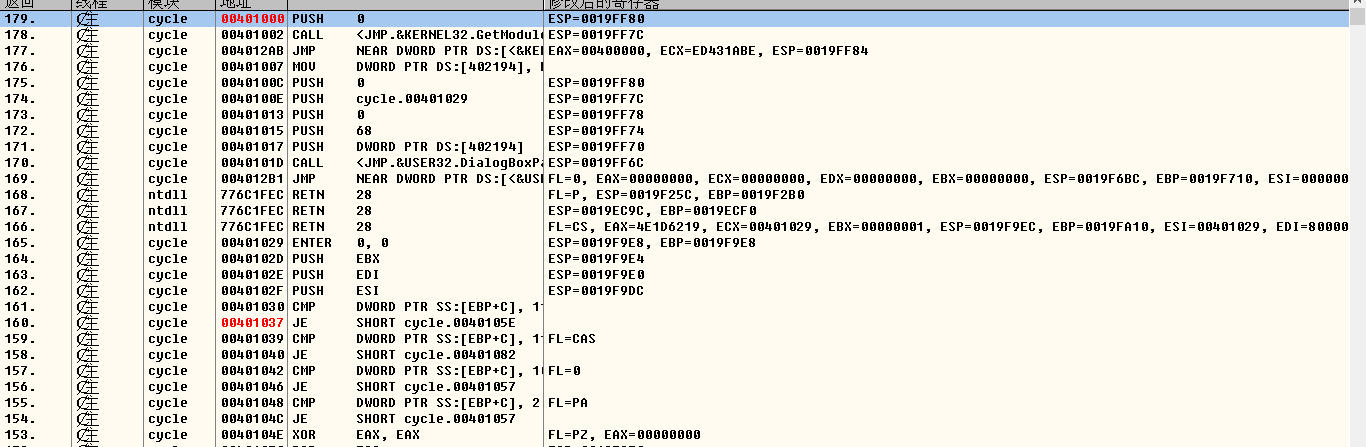
使用假定参数的好处是，可以在使用消息断点的时候。记录参数是使用查看记录的参数的快捷键是 alt+l log data 窗口

值得注意的是 如果不小心 点击了分析 假定参数下面的 从选定内容中删除分析，重新载入程序是没有 用的，要右键，分析 分析代码即可解决

此外，只用在用户领空才能分析代码。在系统领空不能分析代码。

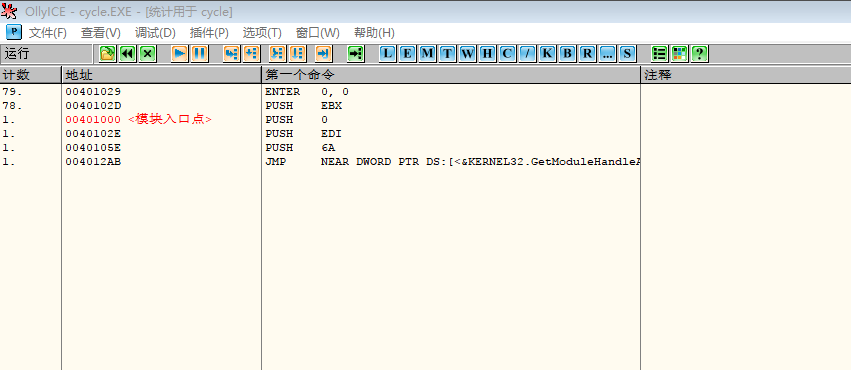
1. 关于run trace 记录的经验

Run trace使用



查看run trace跟踪可以点击 …这个按钮，从图中可以看到，run trace的作用是记录程序的执行完每一条汇编代码后，寄存器的变化，和执行汇编代码所在的模块，还可以记录所在的线程，由于编码编码格式的原因，导致了线程那列是乱码。后边的寄存器修改后的数值方便分析真假注册所造成修改的不同。

当点击统计模块的时候，可以查看当前模块中，所执行的汇编语句情况



第一列是执行次数，当校验程序做真假注册码判断的时候，通常是if结构，所以一般关键跳转的执行次数是一个，但是如果程序做了多次校验。那么这个关键跳转可能出现多次，此外，如果程序使用2种校验机制，那么可能存在多个关键跳转。

此外，run trace 有效应该输入完注册码，点击了注册按钮，之后的查看模块统计，关键跳转可能是call，可能是cmp。

Run trace查看窗口会在隐藏到后台是自动清除当前run trace的操作。如果想要保留应该关闭run trace

关闭run trace 的操作是 菜单栏 调试 关闭run 跟踪

**#include <stdio.h>**

**void f1(void) { printf("a"); };**

**void f2(void) { printf("b"); };**

**void f3(void) { printf("c"); };**

**void (\*f[3])() = { f1,f2,f3 };**

**void main(void) {**

**int i,j,k;**

**for (i=0; i<100; i++) {**

**for (j=0; j<1000000; j++) ;  // Long code**

**k=i/33;**

**if (k>3) continue;**

**f[k]();                      // Here error (when i==99)!**

**};**

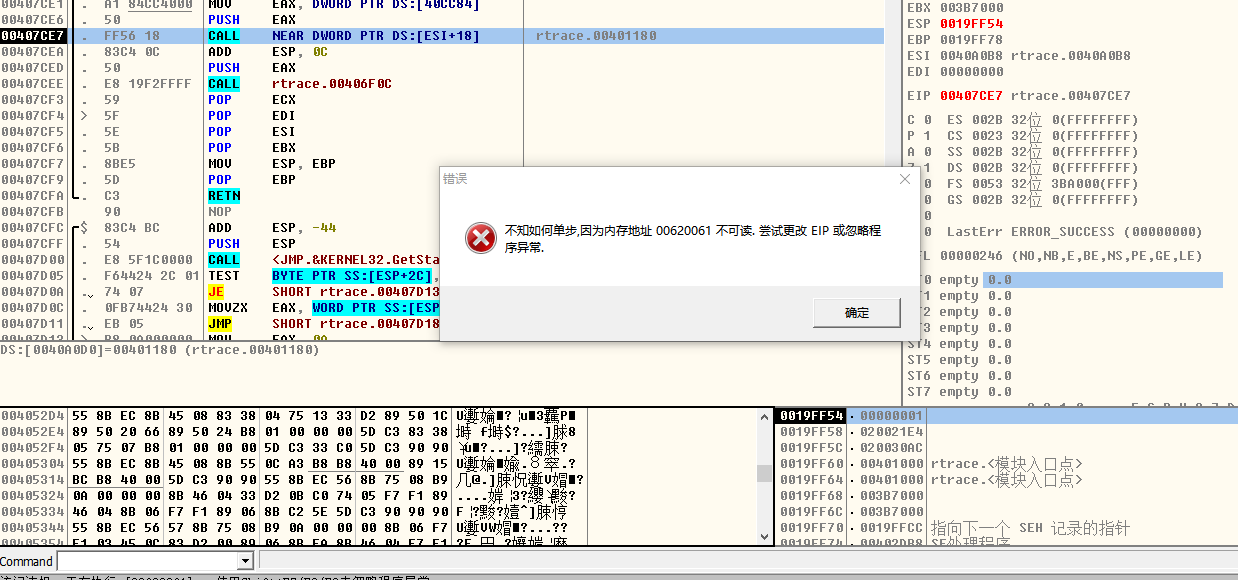
**printf("\n");**

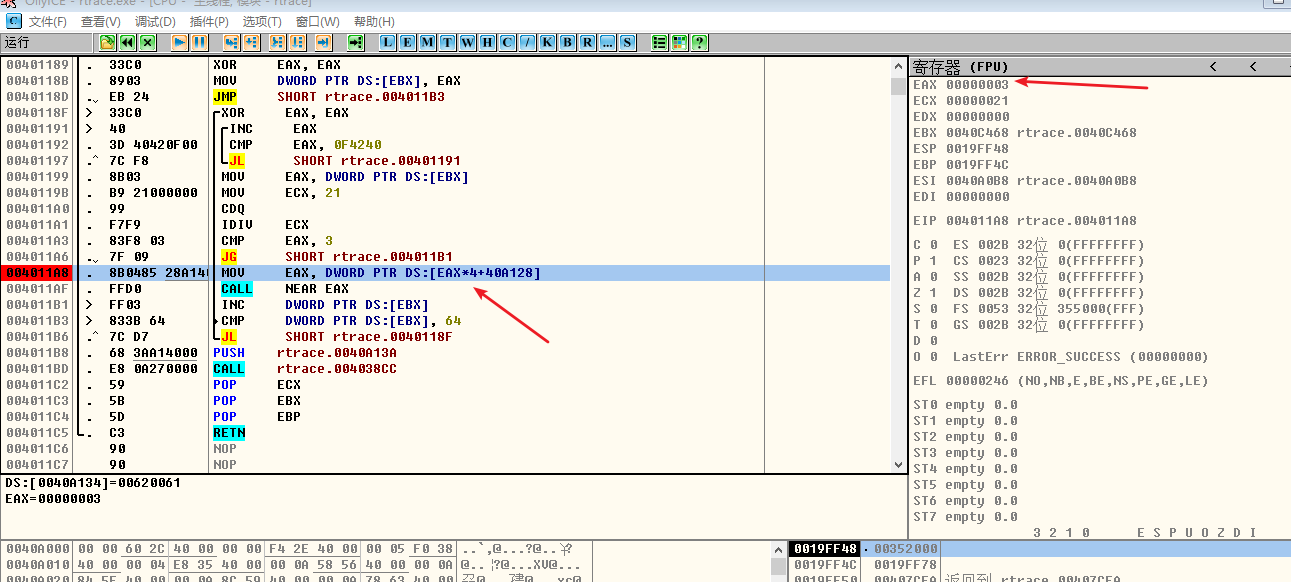
那么最后的结果是k 可能出现的结果是 0 1，2 ，3

由于程序做了校验保护

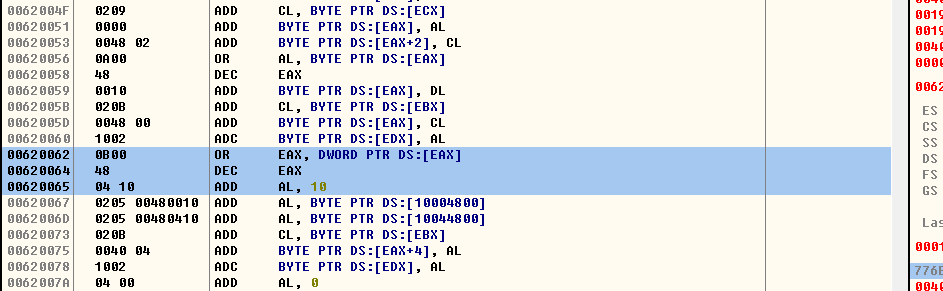
If（k>3） continue; 所以程序不会出现数组访问异常。

利用 run trace 解决异常

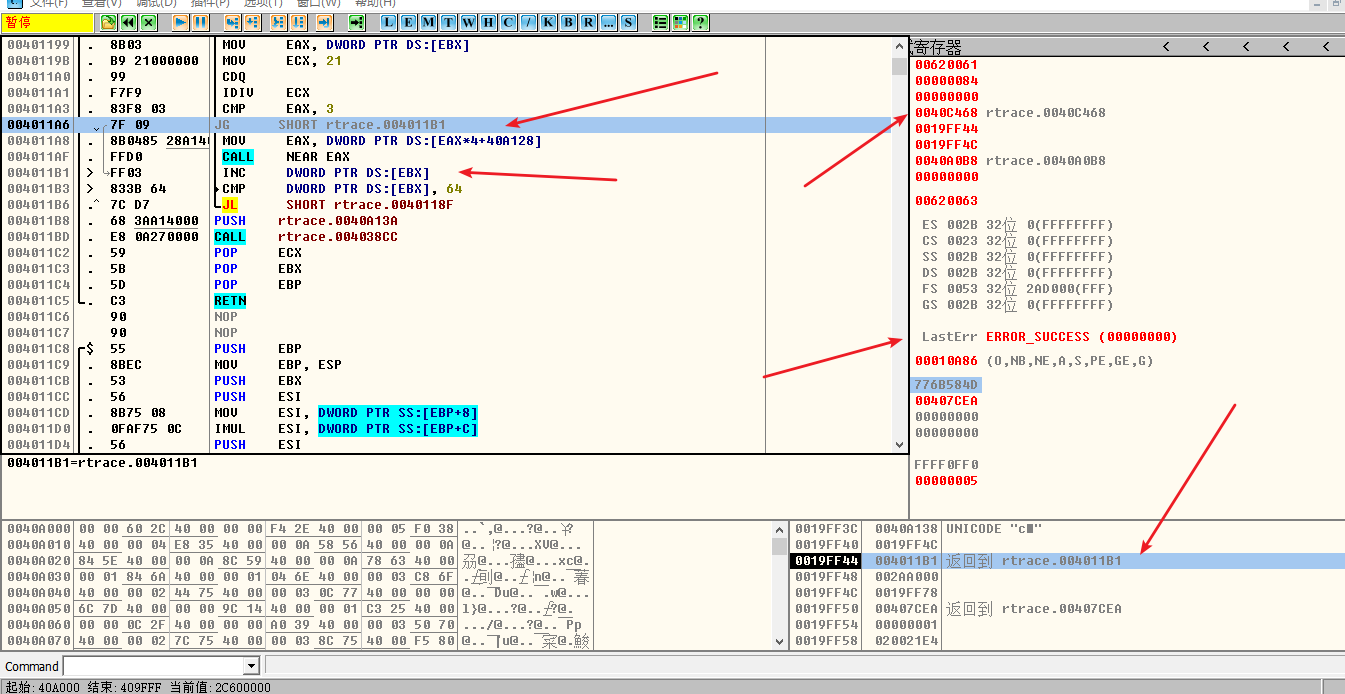




汇编中数组的表示形式，是基址指针+索引指针 基址指针不动，索引指针用eax表示，数组中存的是方法的地址，所以dword的外层还有个中括号表示取地址的值。



异常中断时，汇编窗口的位置



上图说明

使用ollydbg调试程序出现异常的时候，可以从栈中，看到调用方法的返回值，也就是说异常一定是在这个主方法中产生的。这个就是右下角箭头的函数

由于之前采用过run trace方法调试过，所以调试过的位置是灰色的，这就是左边最上方箭头的含义。

左边下边的箭头是一个承上启下的作用，它利用栈空间的保存返回之地址，找到在汇编窗口的对应的位置 对应的查找快捷键是 ctrl+g

也可以在汇编窗口使用右键 转到 表达式 达到同样的效果。

从最左边的转折线可以看到他是从哪个位置跳转过来的，这就引出了左边的第一条箭头

此外，根据异常的提示可知是 访问不到对应的汇编代码，也就是数组越界的异常。但是程序之前是正常的，说明是判断的条件的跳转出现了问题。而这里的判断条件 是cmp eax，3，它的下方就是跳转 这就进一步确定以异常代码的位置。

jg short rtrace.004011b1

由于上文说到是数组越界的异常，所以也就是等于的时候没有判断好，

所以我的修改代码是

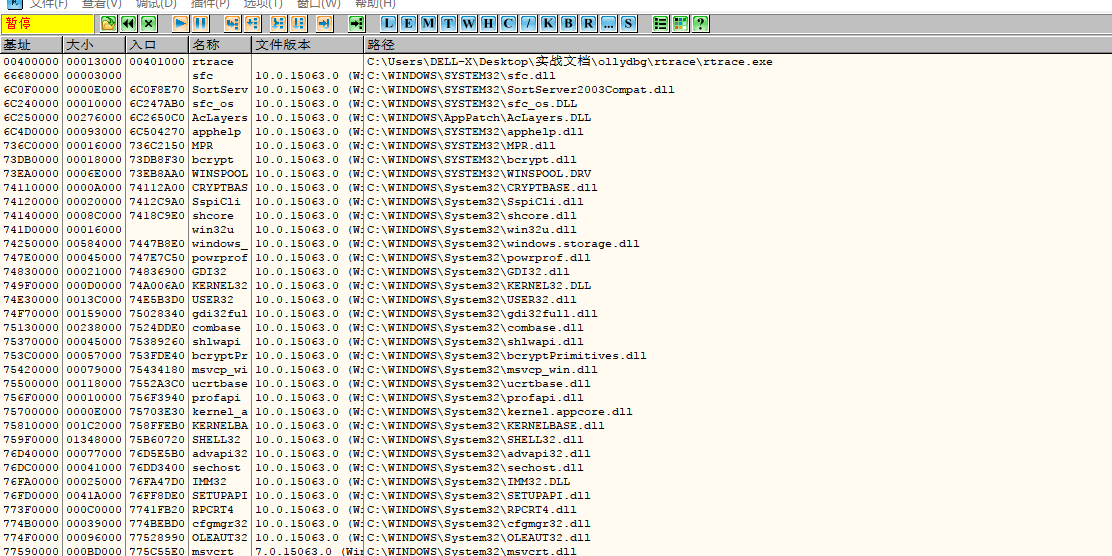
Jge short rtrace.004011b1

之后的操作是邮件，保存修改选中修改的代码，出现一个新的汇编窗口，然后接着右键保存文件。之后运行则正常了。

值得说明的是

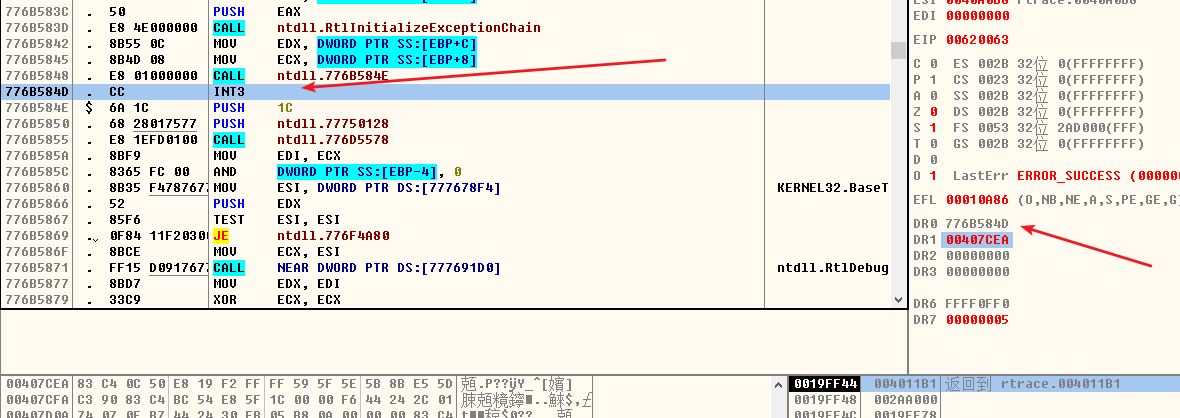
1. 由于程序是cmd debug模式生成的，所以程序一开始汇编并不会直接进入main方法，而会初始化debug hook，在进入main方法之后会初始环境变量，读取系统环境
2. 如果想要统计程序中执行频率的代码，可以使用crtl+f11进行记录执行，记录执行的含义是ollydbg会记录他执行过的汇编代码。

如果程序没有运行完，需要调试的时候，需要使用esc或者是f12中断。之后点击菜单栏 查看 可执行模块 查看对应的模块执行的汇编代码 也可以使用快捷键alt +e 打开。之后点击右键 查看模块统计，可以看到对应模块的汇编代码执行频率。

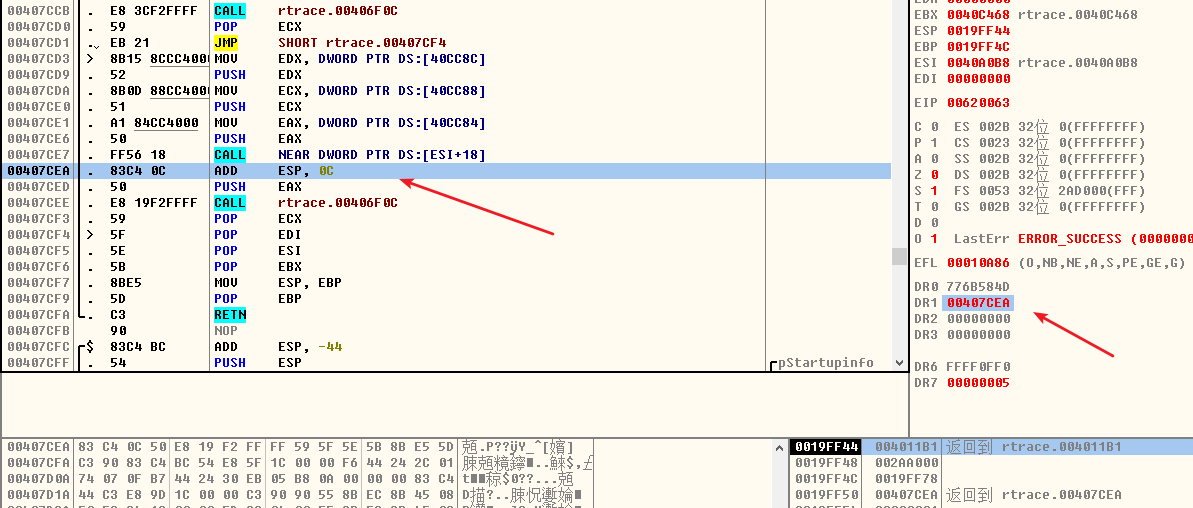


从可执行模块的图上，可以看到对应模块的基址地址，模块的大小，模块的入口点。 和模块的版本和文件的位置，测试环境是win10

Red stone2；



硬件寄存器0 出现int 3中断。



由于出现了异常，所以硬件寄存器1 存的是异常处理窗口对应的代码。

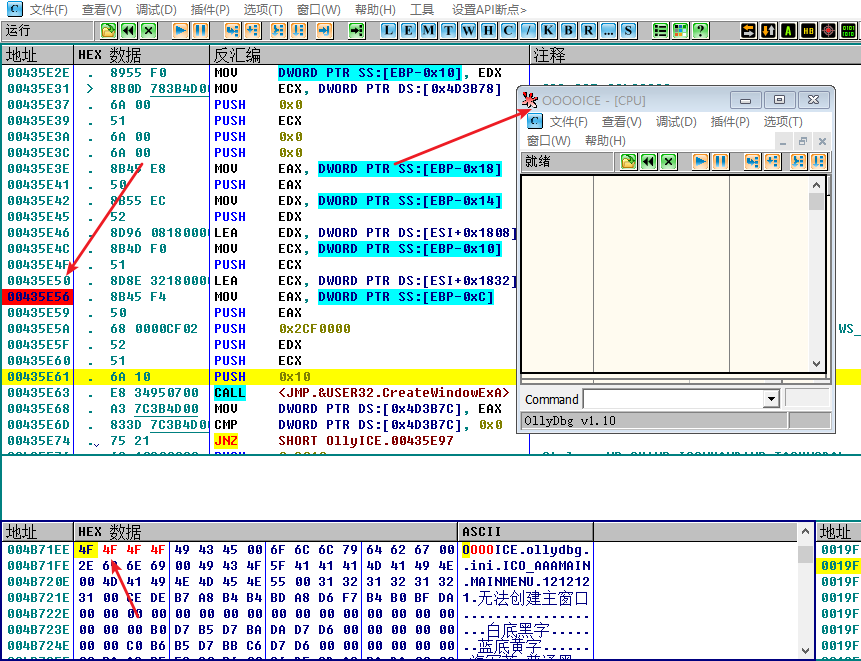
打造自己的ollydbg的方法

1. 通过修改窗口标题和类名可以设置自己的ollydbg。

设置断点的API是 createwindowsexA

使用的16进制指纹是 6A 00 51 6A 00 6A 00

值得注意的是不同的ollydbg的版本createwindowsexA所在的段的位置是不一样的 这里使用的是看雪论坛 1.10汉化版。



此外，

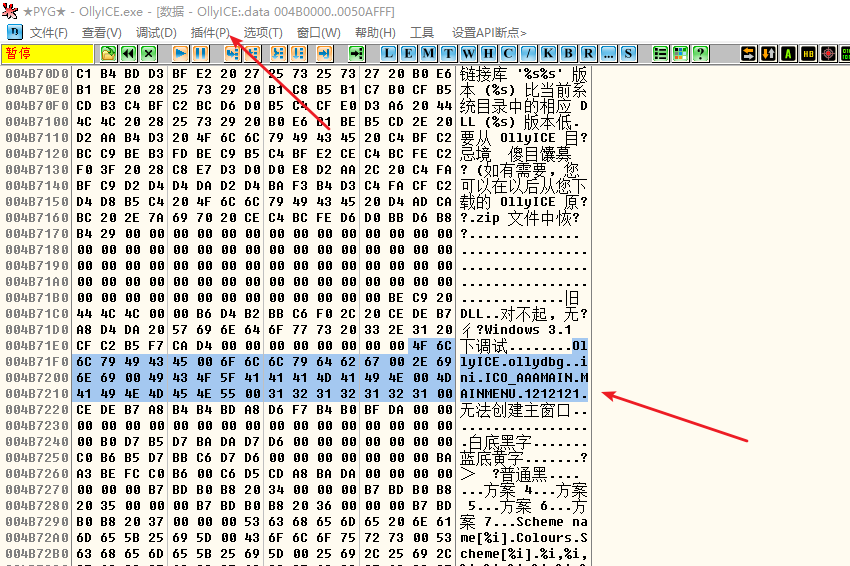
程序地址 00435E37

程序入口 00401000

文件入口 600

计算出在文件中的位置是 35437 435E37-401000+600

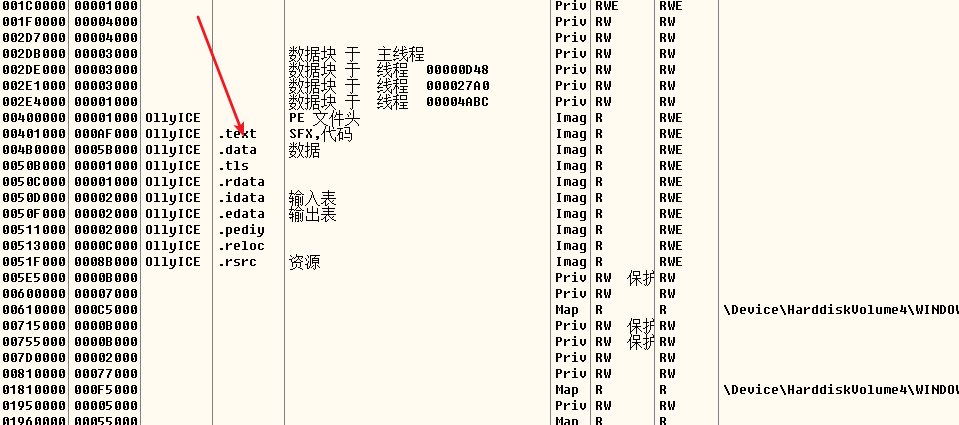
找到的实际位置是 35437



尝试使用内存断点方式，结果设置内存的asm代码，结果无法断下，说明。当前的数据属于data段。

值得注意的是

1. 内存断点的使用是当前调试有效，下一次要重新设置。
2. 删除内存断点的方法是 汇编窗口，或只是数据窗口右键断点 删除内存断点



1. 查看 区块表的窗口是 内存窗口，打开的快捷键是alt+m

Lpvoid类型的使用

Lpvoid的类型是指针类型，但是特殊的是他是没有指定任何基本类型的，也就是说可以转换为任何基本类型的。

示例程序

class CMyClass

{

void Start();

static UINT StartThread(LPVOID lParam);

};

void CMyClass::Start()

{

AfxBeginThread(StartThread, this);

}

UINT CMyClass::StartThread(LPVOID lParam)

{

CMyClass \* pMyClass = (CMyClass\*)lParam;

...

return 0;

}