****

**分析报告**

|  |  |
| --- | --- |
| **样本名** | **winmine.exe** |
| **班级** | **软安41期** |
| **作者** | **张海龙** |
| **时间** | **2020年11月30日** |
| **平台** | **Win10** |

15PB信息安全研究院(协议分析报告)

目录

**[1． 样本概况 3](#_Toc24589)**

[1.1 应用程序信息 3](#_Toc13489)

[1.2 分析环境及工具 系统环境:win10 64位 3](#_Toc3050)

[1.3 分析目标 3](#_Toc19205)

**[2． 具体分析过程 3](#_Toc13009)**

**[3.展示效果 17](#_Toc29126)**

**[参考文献 18](#_Toc24785)**

**[[1] 15PB视频课程-逆向工程实战之扫雷分析 18](#_Toc8988)**

**[致 谢 19](#_Toc13407)**

# **样本概况**

## 1.1 应用程序信息

应用程序名称：winmine.exe

MD5值：16A4FD569A3EB5CEBEB3DA99EF1D17E1

SHA1值:31A1A89BA067EA95F117754818429D6D8E8E59CF

1.2 分析环境及工具系统环境:win10 64位

工具:OllyDebug、Dbgview、SPY++、Die

## 1.3 分析目标

1、窗口识别雷区

2、F5一键扫雷

# 具体分析过程

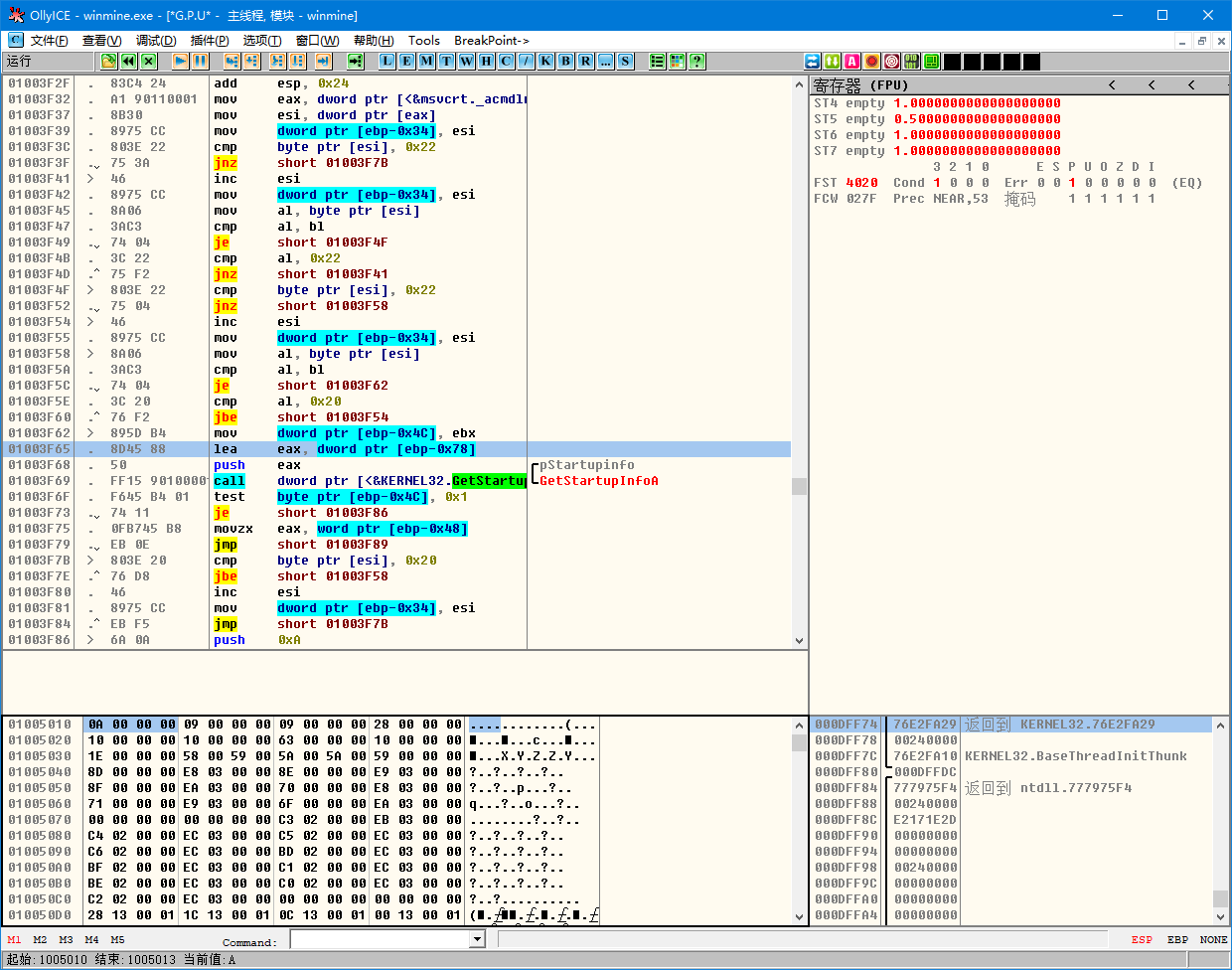
目标查找扫雷的地图高度：

OD附加winmine.exe，F9直接跑起来

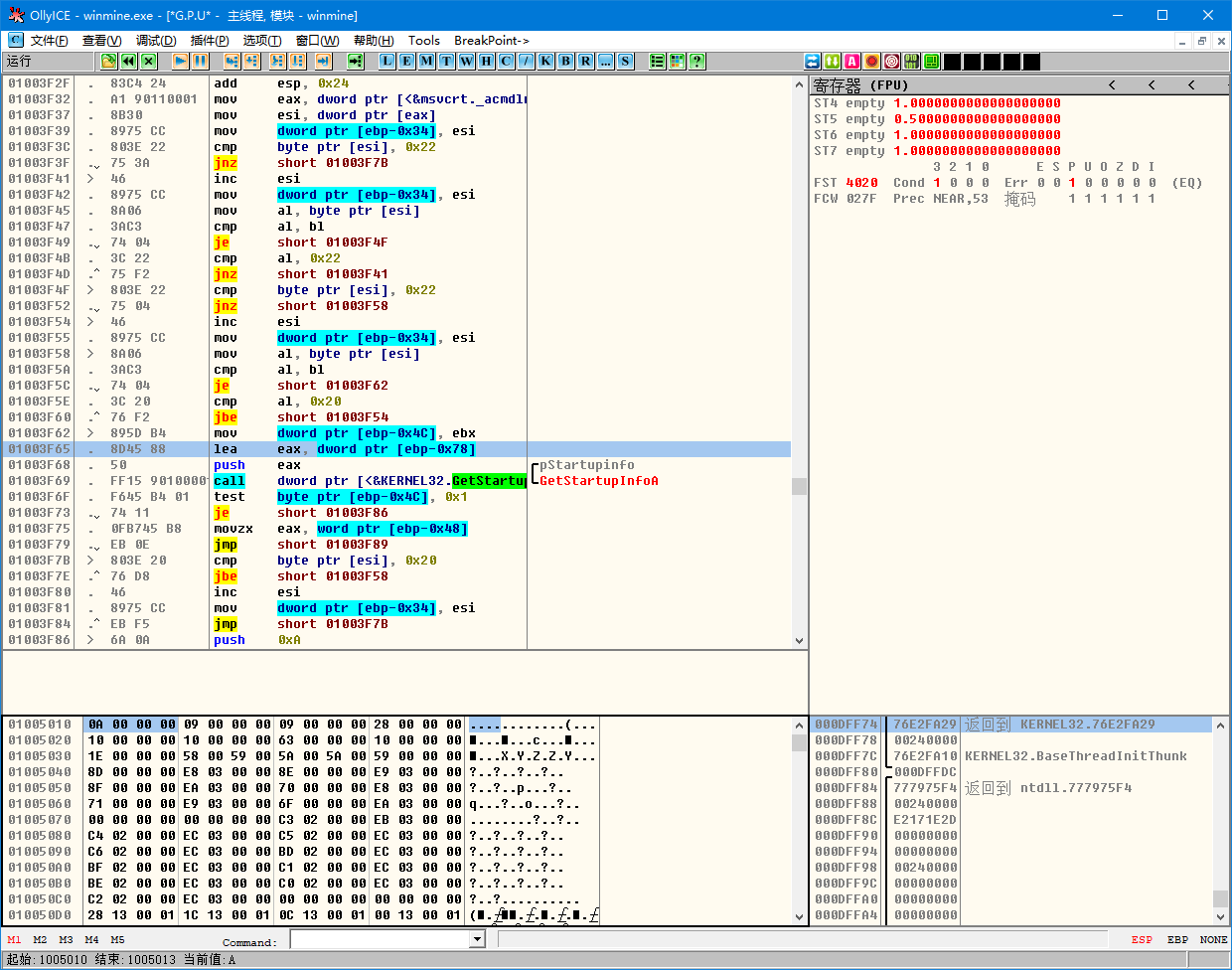
把数据窗口堆的开始。

设置自定义高度为10-->0xA

使用OD的Ctrl+B进行搜索数据----0xA

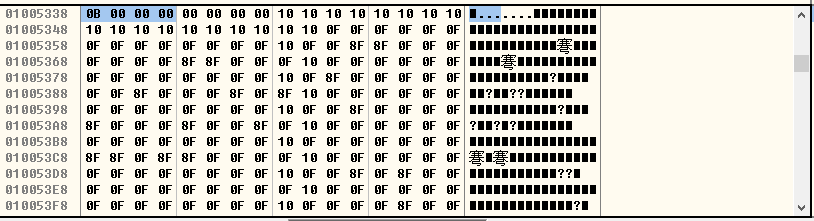


找到了一个数据我们并不确定是不是我们想要找的数据，先验证一下，修改地图的高度为11--->0xB,然后查看内存中的0A是否发生改变。

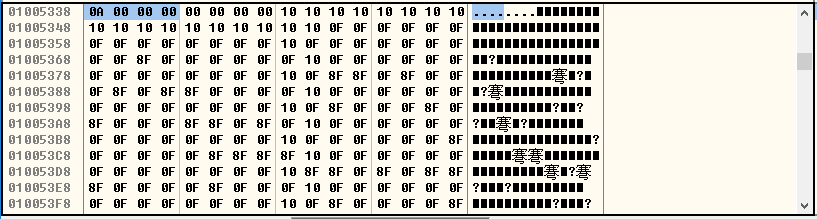


很好，数据并没有发生改变，然后接着搜索0xB,搜索到之后接着做与刚才相同的操作。

修改地图的高度，查看内存中数据是否发生改变。



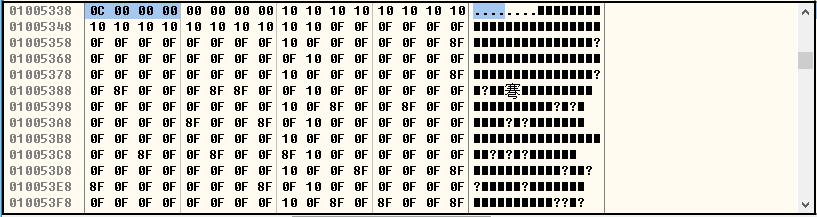
尝试修改数据为10-->0xA，观察内存情况



数据发生改变，恭喜找到了。

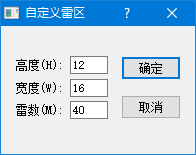
多次测试。

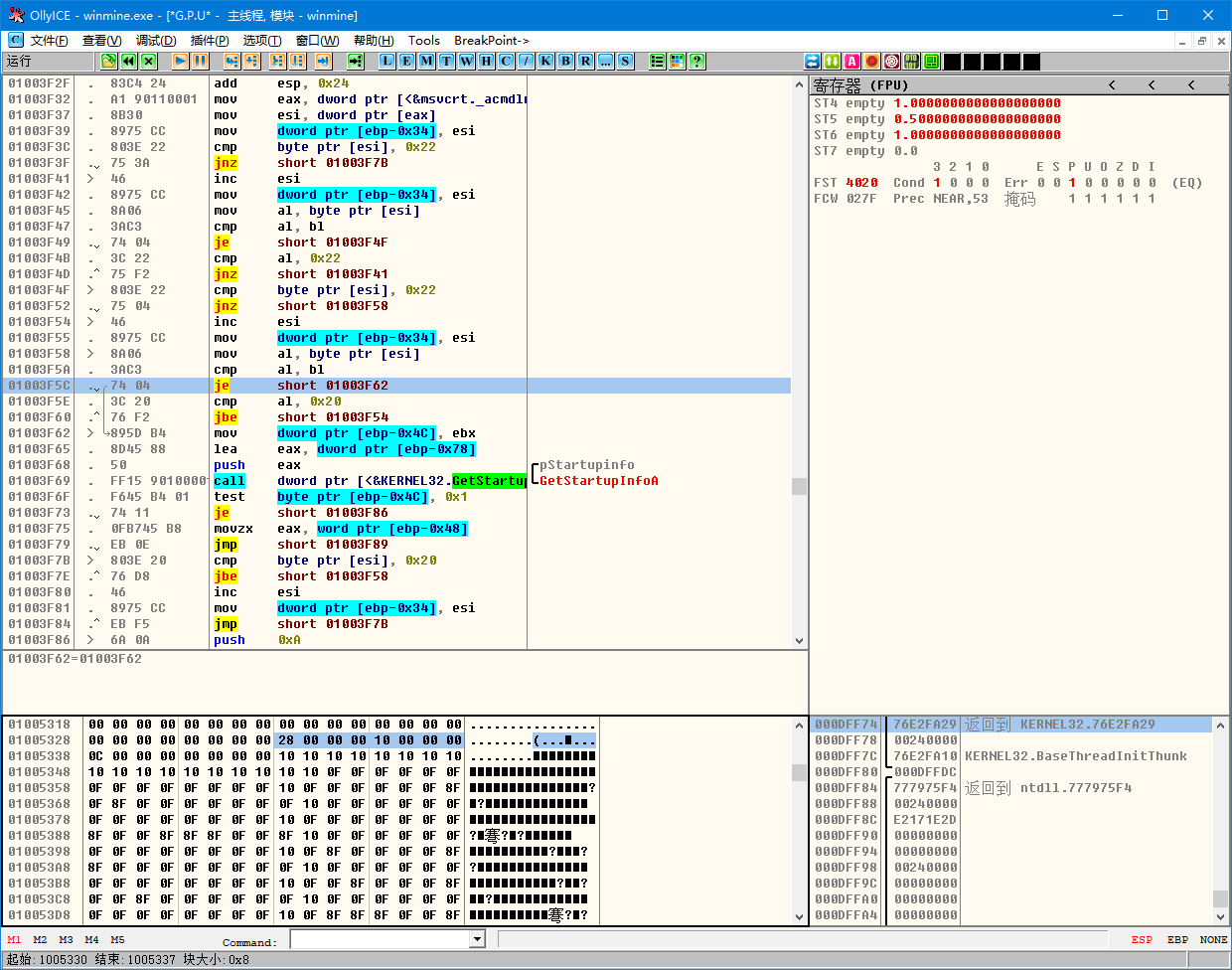
12--->0xC



内存中的数据也是跟着变，说明找到了。

上下查看是否有宽度和雷区的数据



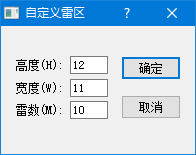


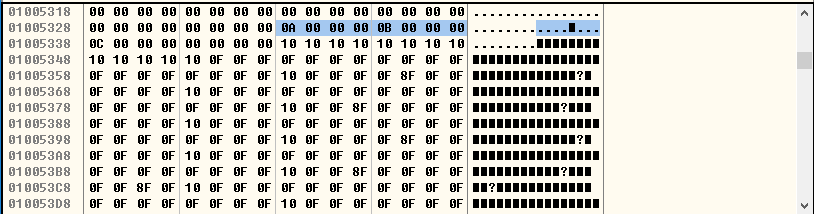
0x28刚好是雷数，0x10刚好是宽度。

尝试修改，查看内存是否发生改变。

雷数 = 10--->0xA

宽度 = 11--->0xB





内存中数据也跟着改变，说明找到了宽度和雷数

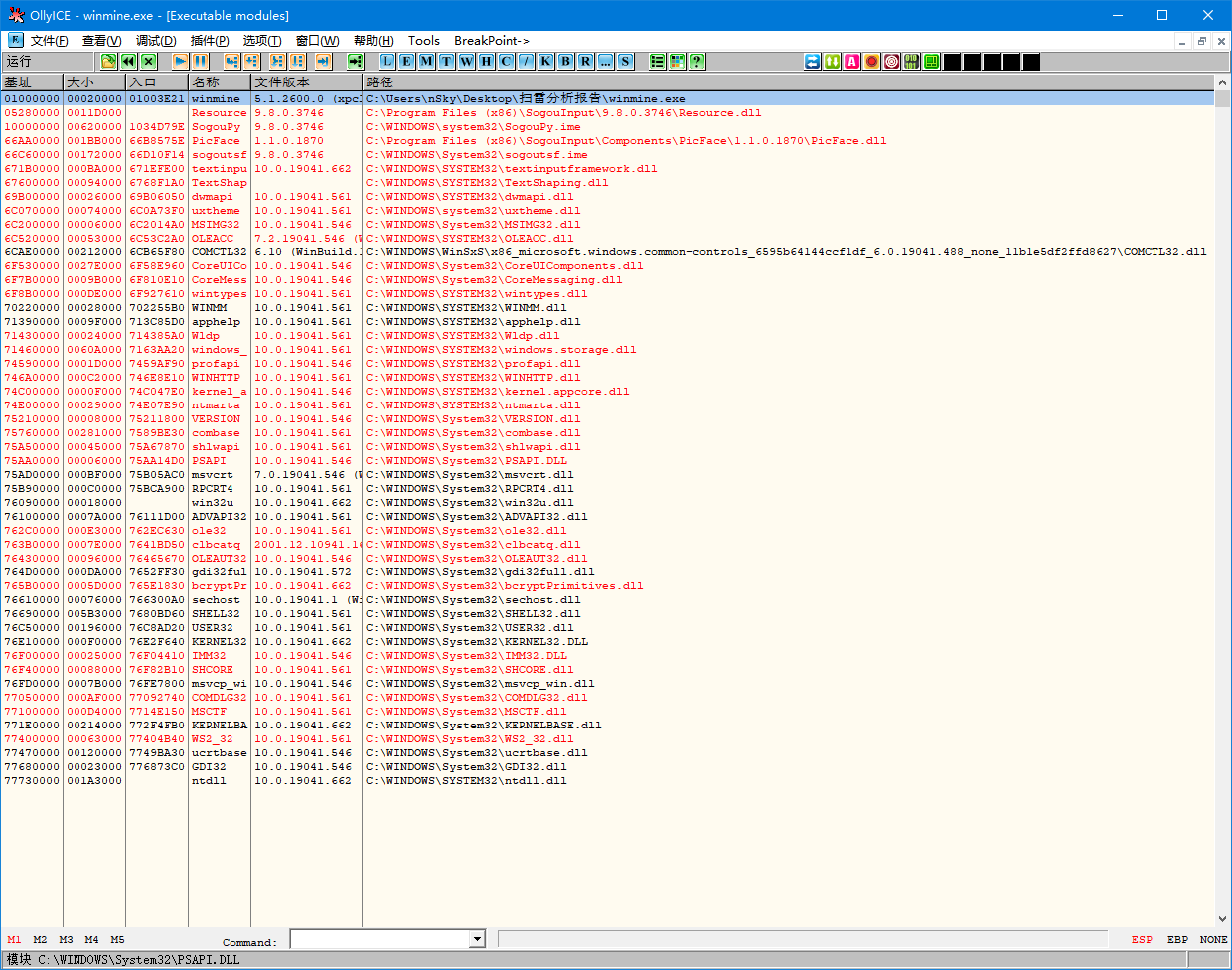
记录地址。

g\_pHight = 0x1005338;

g\_pWigth = 0x1005334;

g\_pMineCount = 0x105330;

OD查看模块winmine.exe的加载基址

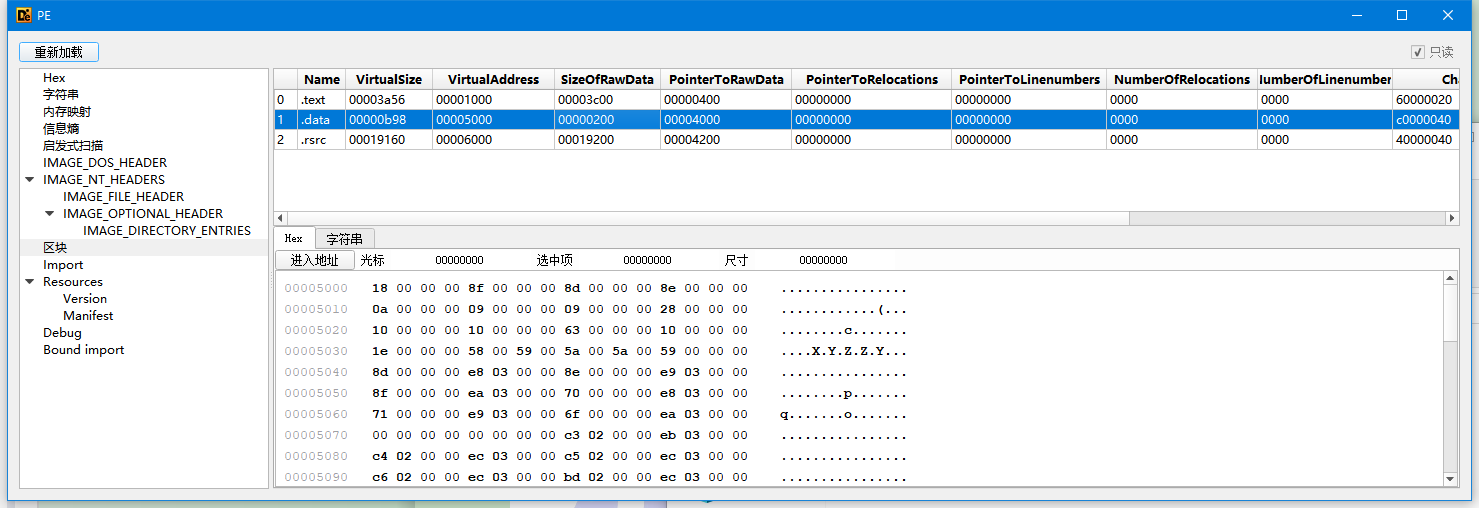


0x1000000

用刚刚的地址去掉这个0x1000000

发现大小都在5330-5340

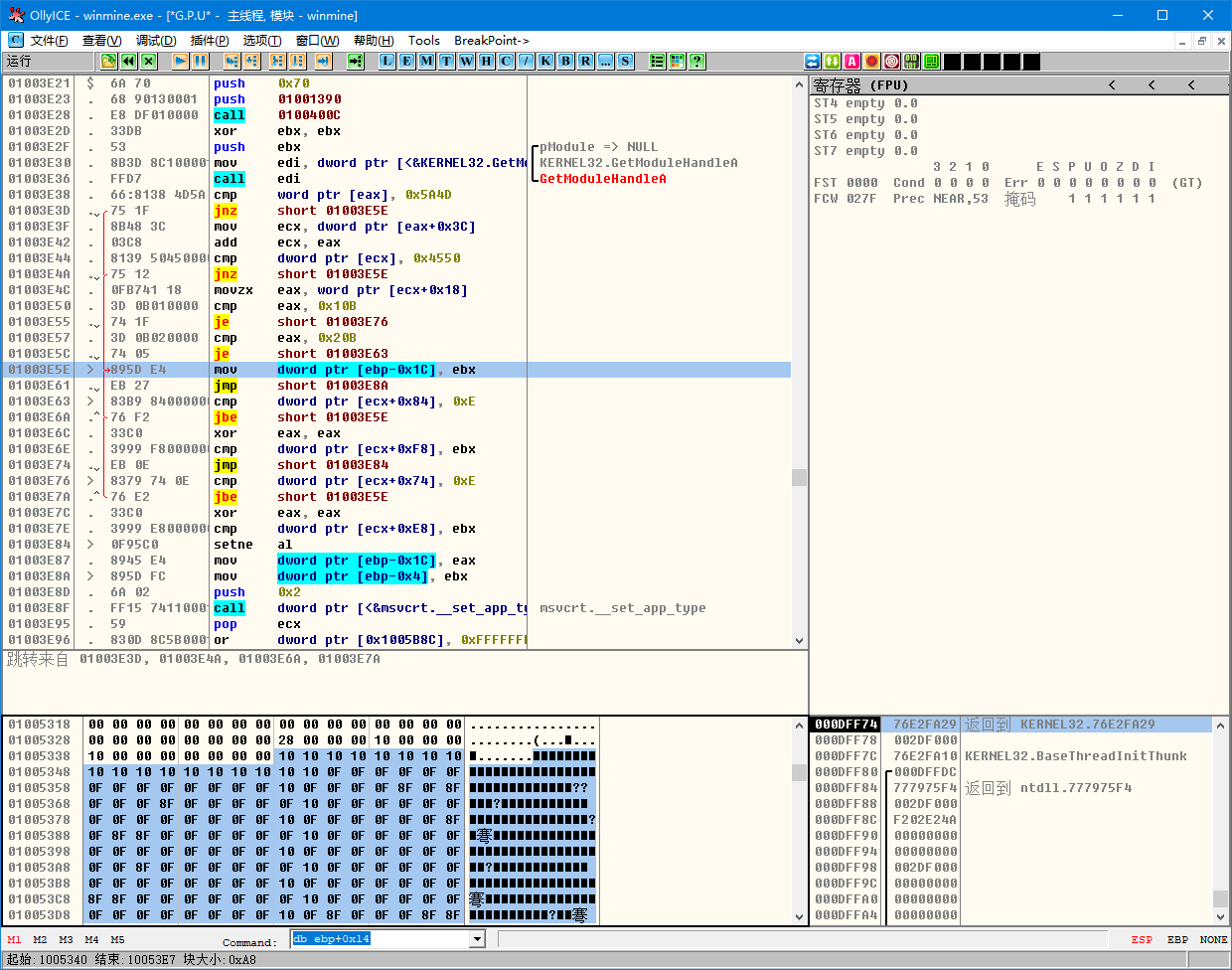
使用DIE查看区段



他们都落在了.data段说明是全局数据。

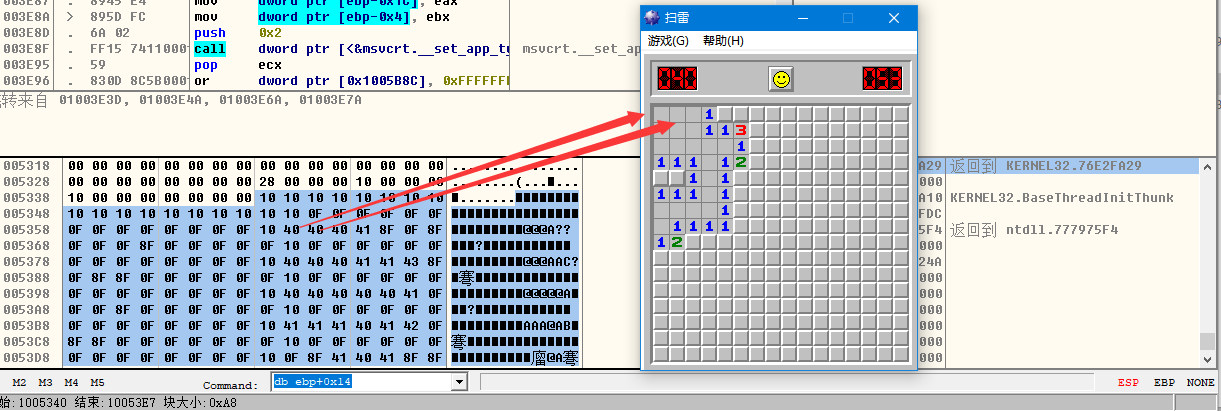
地址是不变的。

继续观察数据



发现被选中的一块很像数组。

点击扫点开一个地图区域观察内存中很想数组的区域

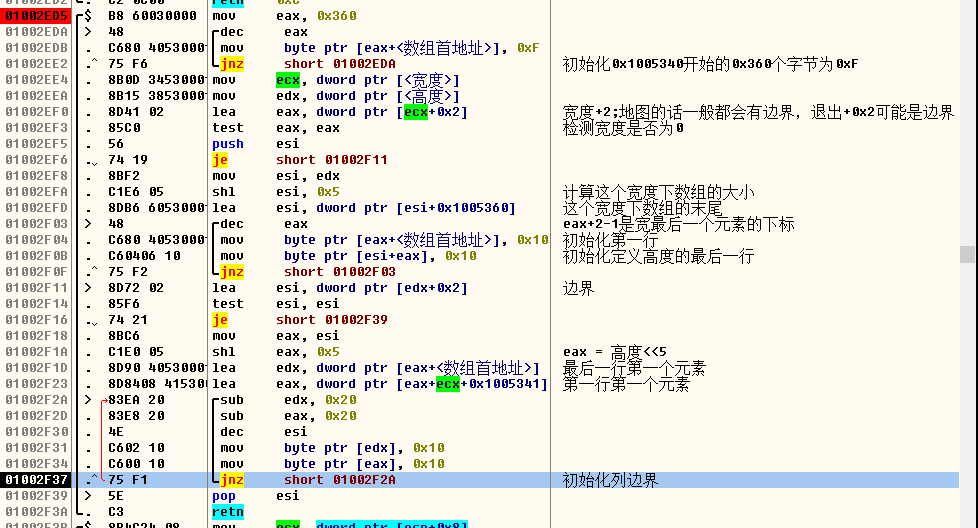
发现其中数据跟着改变了，推测是数组。

下一个内存写入断点查看是谁在初始化的时候修改了他。

每一次新游戏都会初始化一下这个数组。猜想。

重新开始游戏。。果然断下来了。

我们大致看一下代码做了什么事情



应该是做初始化工作的。

这段代码走完之后发现内存中数据的确被初始化了。

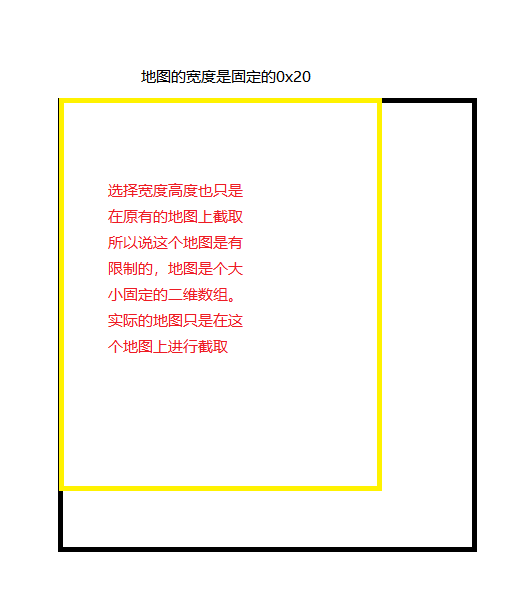


内存每行0x10我们设置宽为0x14因为有边界为2

高尽量少，方便观察，经过测试，高最低是9，就设置9



但是好像发现每行中间又多了一行，经过多伦数据测试，发现并不是多的一行，而是，数组的最大宽度为0x20。剩余数据是为了能控制地图可大可小的兼容性而设定的。



我们可以使用一下方法遍历数组：

typedef int (Base\*)[0x20];

Base g\_Base = (Base)0x1005340;

PDWORD g\_pHight = (PDWORD)0x1005338;

PDWORD g\_pWigth = (PDWORD)0x1005334;

PDWORD g\_pMineCount = (PDWORD)0x105330;

for (size\_t y = 1; y < (\*g\_pHight) + 1; y++)

{

for (size\_t x = 1; x < (\*g\_pWidth) + 1; x++)

{

BYTE bCode = g\_pBase[y][x];

CString strTmp;

strTmp.Format(\_T("%02X "), bCode);

StrDbg += strTmp;

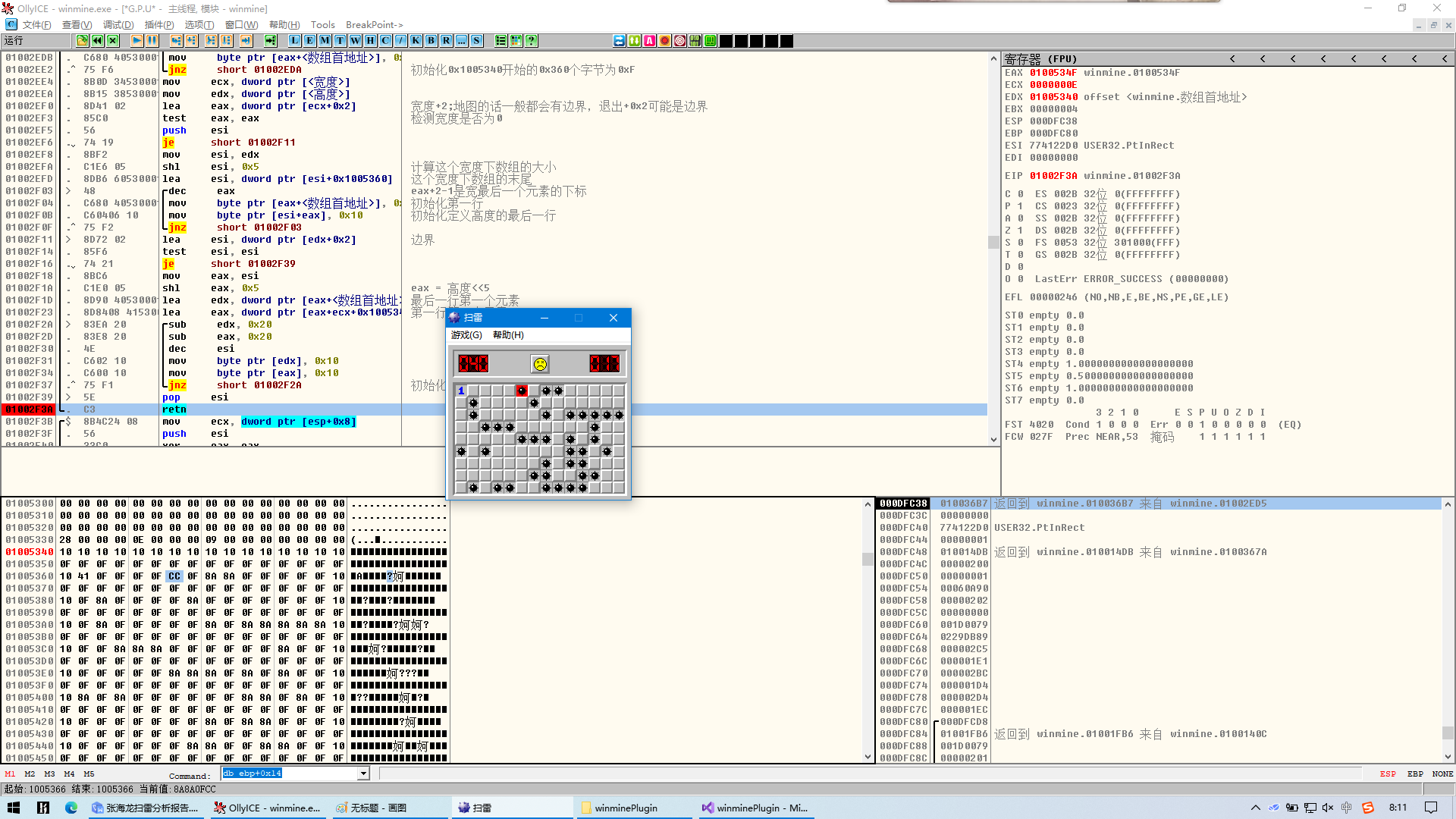
}

::OutputDebugString(StrDbg);

}

下面来验证一下地图的标志代表什么。

开始游戏并观察地图。



分别点开地图上的0F和8F

0F点开后不是雷，8F点开后就炸开了。说明8F是雷。

下面就开始模拟鼠标点击来做个一键秒杀。

我们找一下扫雷的回调函数。下一个鼠标点击的消息断点。

Spy++扫雷，查看回调函数。



这里的窗口进程即为窗口的回调函数。

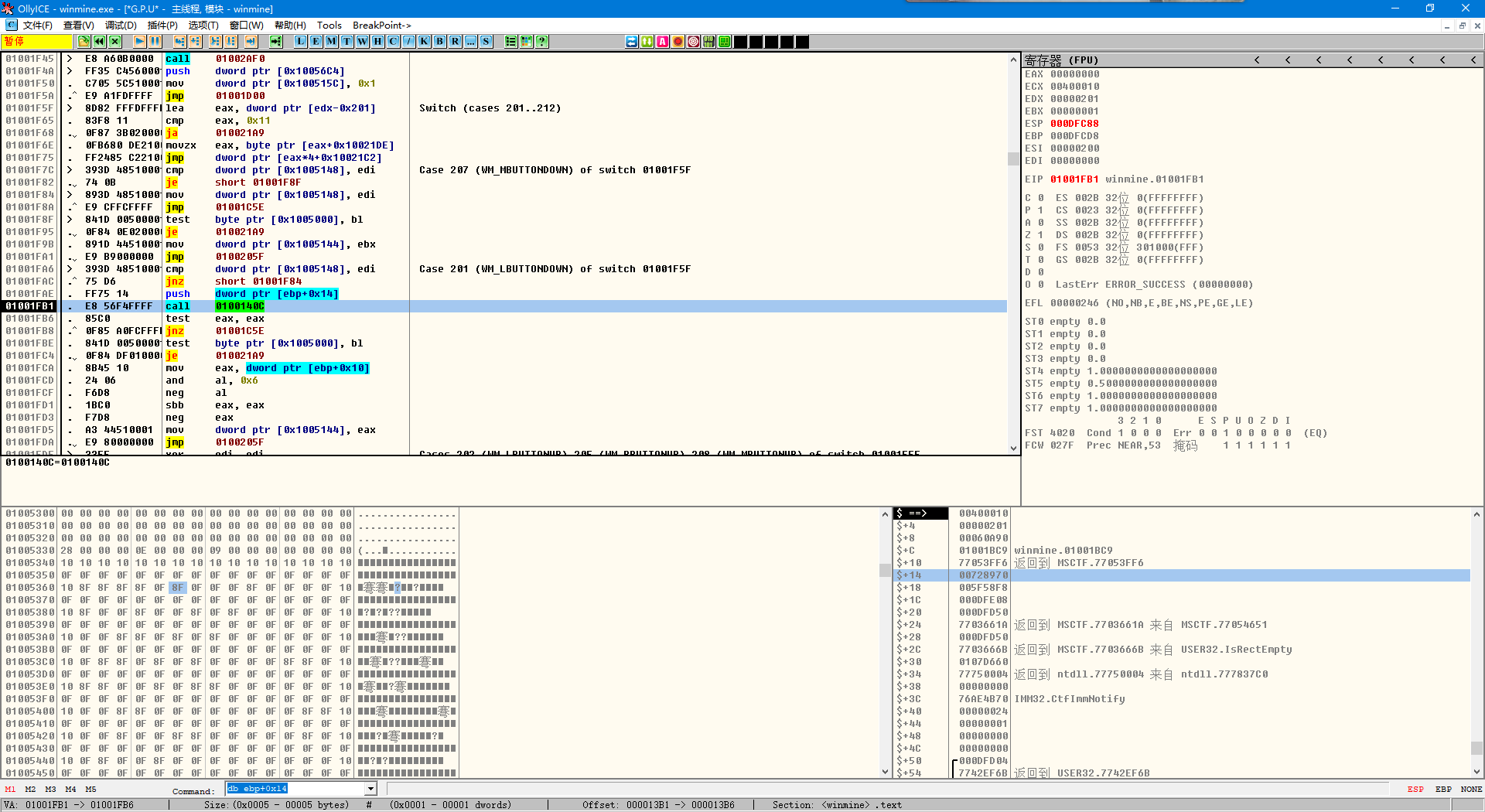
我们在OD中直到这个位置，分析---->假定参数WinProc

下个消息断点WM\_LBUTTONDOWN

点击翻开扫雷地图，断点断下，我们跟踪断点。观察是否哪里读取了我们的点击坐标。

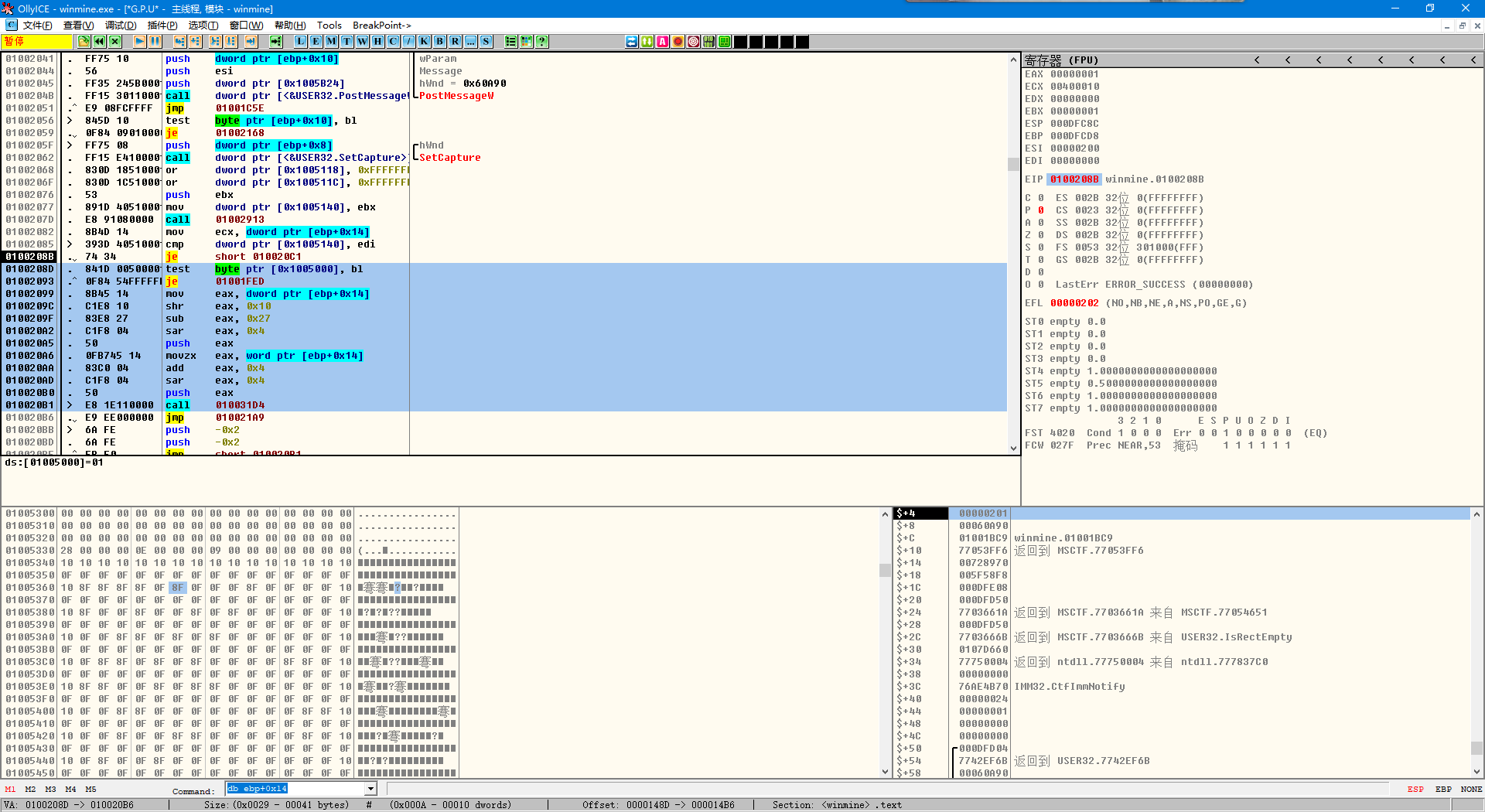
并且把他转换成了数组的下标。（因为只有把坐标转换成数组的下标才能正常访问数组元素。）

F8跟踪他的第四个参数ebp+0x1C



这个函数访问了lParam但是进去后发现他只是做了和绘图有关的操作，我们先略过。

继续往下跟踪。



我们发现这里又对这个参数进行了访问，并且做了一下逻辑操作。

01002099 |. 8B45 14 mov eax, dword ptr [ebp+0x14]

0100209C |. C1E8 10 shr eax, 0x10

0100209F |. 83E8 27 sub eax, 0x27

010020A2 |. C1F8 04 sar eax, 0x4

010020A5 |. 50 push eax

010020A6 |. 0FB745 14 movzx eax, word ptr [ebp+0x14]

010020AA |. 83C0 04 add eax, 0x4

010020AD |. C1F8 04 sar eax, 0x4

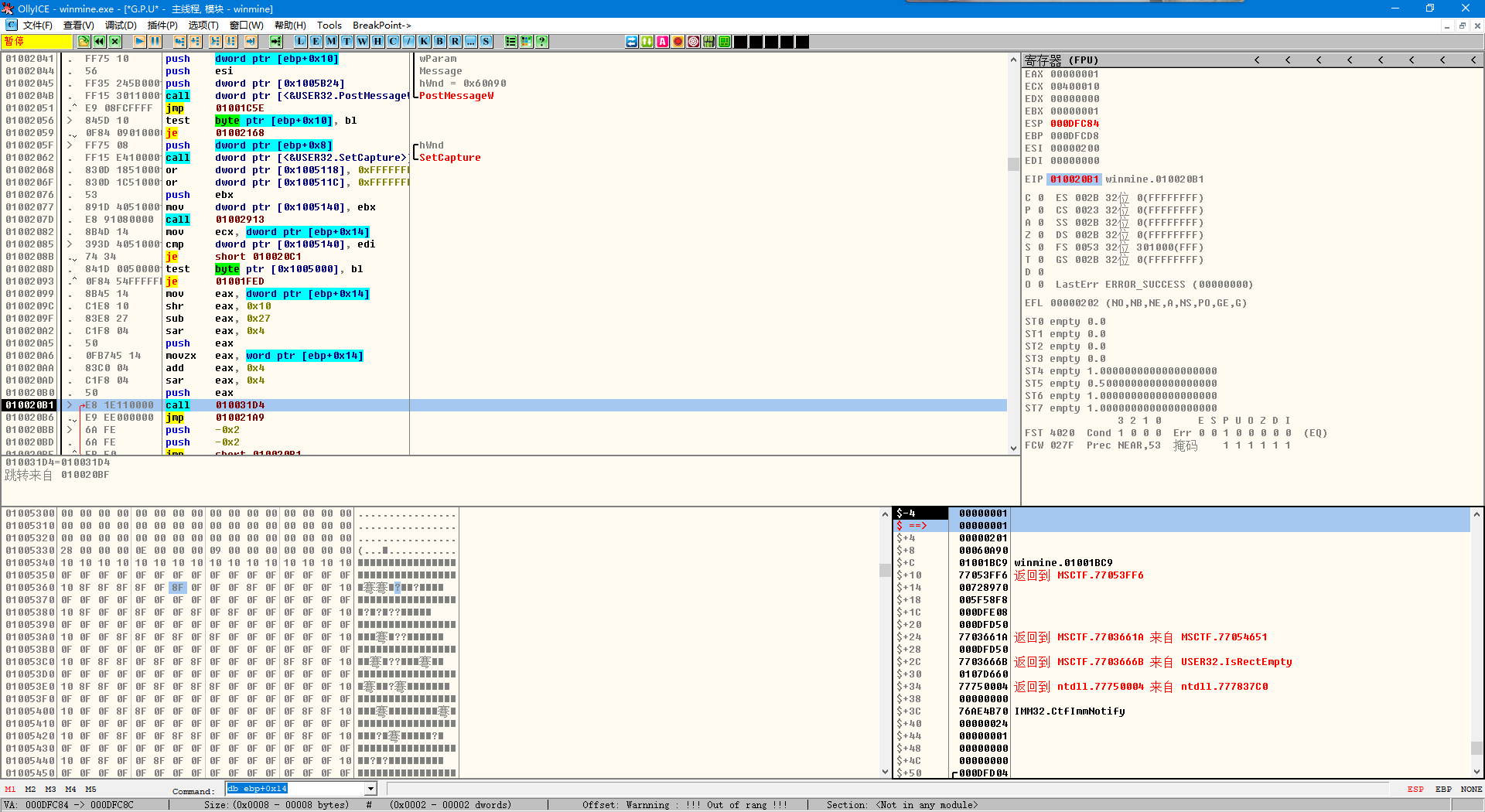
010020B0 |. 50 push eax

010020B1 |> E8 1E110000 call 010031D4

(取出高16位-0x27)>>0x4

(取出低16位+0x4)>>0x4

我们占时不是是干嘛的先走完这几条语句。

观察栈发现这正是我们刚才点击的下标1,1说明这里是计算下标的。

我们就可以利用这个公式来计算下标了。

如果要反推坐标，我们把公式逆转即可。

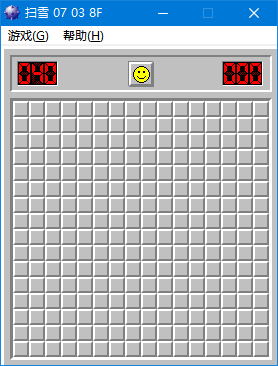
# 展示效果

打开扫雷，把我们编写的DLL注入其中。验证功能是否有效

F5一键秒杀完成



鼠标悬停地图上方查看是否有雷，完成，扫雪代表有雷



参考文献

[1] 15PB视频课程-逆向工程实战之扫雷分析

致 谢

感谢15PB信息安全研究院。