WEEK4-RogerThat WP

这周有点寒碜，只有三道题

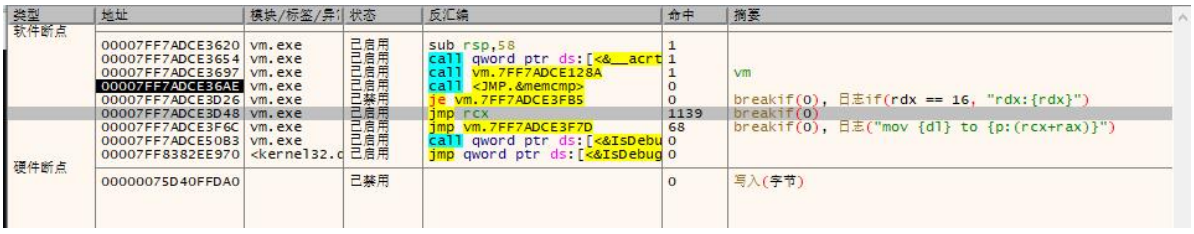
Re

1.vm

这道题是一道虚拟机相关的题目，因此分许需要花不少时间，在没做出来之前也慢慢试着写点东西，希望也能有所收获。

首先寻找到main函数

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  FILE *v3; // rax
  int v4; // eax
  const char *v5; // rcx
  char Buffer[40]; // [rsp+20h] [rbp-38h] BYREF

  puts("Welcome to ovm++!");
  j_inilization();
  puts("Input your flag: ");
  v3 = _acrt_iob_func(0);
  fgets(Buffer, 35, v3);
  j_printf_("Your flag is: %s\n", Buffer);
  puts("VM started successfully!");
  j_vm_function((__int64)&qword_7FF7ADCEE378, (__int64)Buffer);
  v4 = memcmp(Buffer, &unk_7FF7ADCEBD68, 0x22ui64);
  v5 = "nop";
  if ( !v4 )
    v5 = "good";
  puts(v5);
  return 0;
}
```




```
#dialog
mov 83 to 000000B1524FF781
mov 46 to 000000B1524FF780
mov 23 to 000000B1524FF77F
mov 1 to 000000B1524FF77E
mov EC to 000000B1524FF77D
#.....
mov 90 to 000000B1524FF764
mov 75 to 000000B1524FF763
```

```
mov 5A to 000000B1524FF762
mov 39 to 000000B1524FF761
mov E9 to 000000B1524FF760
#一次循环加密
mov 9 to 000000B1524FF781
mov 2C to 000000B1524FF780
mov 69 to 000000B1524FF77F
mov A7 to 000000B1524FF77E
mov F2 to 000000B1524FF77D
#...
mov 3B to 000000B1524FF763
mov 80 to 000000B1524FF762
mov BF to 000000B1524FF761
mov CF to 000000B1524FF760
#第二次加密
```

大致可以得出是加密了两遍，把某一个值计算得到后移到对应位置，这个位置从最后一位到第一位再从最后一位到第一位，两次加密，得到结果

再进一步看看每个位置的结果是结果什么得到的

```
#第一次循环下
rdx:A
rdx:8
rdx:C
rdx:B
rdx:D
rdx:12
rdx:14
rdx:D
rdx:2//different
rdx:13
rdx:7
rdx:15
rdx:C
rdx:11/different
rdx:9
rdx:16
#第二次循环下
rdx:A
rdx:8
rdx:C
rdx:B
rdx:D
rdx:12
rdx:14
rdx:D
rdx:6 //different
rdx:13
rdx:7
rdx:15
rdx:C
rdx:4 //different
rdx:9
rdx:16
```

由此可见A,8,C,B,D,12,14,D,2,13,7,15,C,11,9,16;这些 OPCODE 组成了一个加密的循环，而第一个嘛我也不清楚，总之它的结果是'{'--->?应该也不用太在意

通过对比两次输入的rdx流，发现这个关于每个字符是独立加密的，也就是缺少"扩散"的一种加密方法，虽然这样我更好做题目了，hhha

整理加密的伪代码之后得到如下结果

```
struct VM{
        uint64_t *alloc_mem_pt;
        uint64_t *buffer_pt;
        uint8_t a;
        uint8_t x1;
        uint8_t x2;
        uint8_t x3;
        uint8_t x4;
        uint8_t b;
}vm;

//A:
    vm.x3 = v8 + 1;
    *(_BYTE *)(vm.x3 + pt_of_memory) = vm.x1;
    goto LABEL_32;
//8:
    function();
    vm.x3 = v8 + 1;
    *(_BYTE *)(vm.x3 + pt_of_memory) = v2;

//。。。。。。。。。。。。
//。。。。。。。。。。。。
//我的愚昧的伪代码，留在这警示一下我自己 /吐血.jpg
    result = *a1;
    v7 = *a1 + 8 * v18;
    if ( !*(_BYTE *)(v7 + 4) )
        continue;

void function(){
    v11 = v5++;
    vm.x4 = v5;
    v2 = *(_BYTE *)(*a1 + 8 * v11);
}
```

接下来思路就简单了，要么静态分析，要么动态调试一波，总之为了理解这个程序流不择手段2333
其实也可以分析汇编，不过估计伪代码应该比汇编简单一些

大意了，看出第一个循环是从后往前异或，没想到第二个循环变了.

这里吐槽一下我自己吧，伪代码已经那么诡异了，我为什么还要把伪代码修改成我自己的伪代码.....直接看ida的伪代码看熟悉了应该就可以了吧，如果真的要写成我自己的伪代码的话，估计得涉及到很多高级的重构方法了，要是真的能那么做的话，我也可以写个IDA了hhhha，所以结论就是我应该做不到

看出第一个阶段是异或，第二个阶段是减去莫个值，因此我们dump下来这个值就好了

上脚本

```
encrypted
=b'\xCF\xBF\x80\x3B\xF6\xAF\x7E\x02\x24\xED\x70\x3A\xF4\xEB\x7A\x4A\xE7\xF7\xA2\
x67\x17\xF0\xC6\x76\x36\xE8\xAD\x82\x2E\xDB\xB7\x4F\xE6\x09'
xor_key
=b'\xFE\x21\x44\x67\x8A\xAD\xD0\xF3\x16\x39\x5C\x7F\xA2\xC5\xE8\x0B\x2E\x51\x74\
x97\xBA\xDD\x00\x23\x46\x69\x8C\xAF\xD2\xF5\x18\x3B\x5E\x81'
decrease_key=b'\x7A\x1A\xBA\x5A\xFA\x9A\x3A\xDA\x7A\x1A\xBA\x5A\xFA\x9A\x3A\xDA\
x7A\x1A\xBA\x5A\xFA\x9A\x3A\xDA\x7A\x1A\xBA\x5A\xFA\x9A\x3A\xDA\x7A\x1A'
decrypted   =''
for i in range(34):
    temp_num = int(encrypted[i])+int(decrease_key[33-i])
    temp_num &= 0xFF
    temp_num ^= int(xor_key[33-i])
    decrypted += chr(temp_num)

print(decrypted)
#decrypted = "hgame{wOw!itS_CpP_wItH_little_vM!}"
```

看来还有好多可以琢磨的了，怪不得这道题有人做这么快。这个虚拟机的指令真的不是简单的指令，我本来以为是相当于push,pop这种，最后想一下应该是相当于一个函数了，然后又因为一些交叉的变量看起来比较复杂，这点看来很值得思考了。

x64日志dump挺好用的，https://help.x64dbg.com/en/latest/introduction/Formatting.html

2.nllvm

虽说和 llvm 和 ollvm 有一定关系，但是这个混淆只是单纯的增长，程序流并没有混淆

根据程序流，或者根据加密过程中 **sbox** ，可以判断出是 aes ，但是是不是 aes 变种呢，由于我对aes-256 cbc 有所误解，以为key的长度不能是 64 字节，导致我只好手撸这道题，不过想来，如果这样子的话我以后就能对付变种了，虽然。。。建立在我被自己坑的基础上。

不过由于是数论的东西，我也要mark一下。在 GF(2^8) 域上，加法和乘法不是我们一般的域的加法，乘法则需要左移一位，然后再判断正负,乘以 0x1b 之类，正如下面的 Multiply 宏定义。而 xtime 是用于乘法，详细的见https://www.cnblogs.com/Jeely/p/11724506.html，其实手撸 aes 我觉得还是挺有价值的。因为我大一下 hgame 中也做过一道，那道我是用代码辅助手算得到的 flag ，痛苦得很。。。不过感觉数论好重要啊。。。可能我是一个密码手 2333

如果要复现的话一种可能就是用宏函数，在没有什么优化的情况下应该是能让函数如此之长，还有一种就是使用一些混淆的工具，如 ollvm 这种，这种应该就要调试分析了。

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#define Multiply(x, y)                                \
      (  ((y & 1) * x) ^                              \
      ((y>>1 & 1) * xtime(x)) ^                       \
      ((y>>2 & 1) * xtime(xtime(x))) ^                \
      ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^         \
      ((y>>4 & 1) * xtime(xtime(xtime(xtime(x))))))   \

static const uint8_t rsbox[256] = {
        0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e,
0x81, 0xf3, 0xd7, 0xfb,
        0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44,
0xc4, 0xde, 0xe9, 0xcb,
```

```
        0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b,
0x42, 0xfa, 0xc3, 0x4e,
        0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49,
0x6d, 0x8b, 0xd1, 0x25,
        0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc,
0x5d, 0x65, 0xb6, 0x92,
        0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57,
0xa7, 0x8d, 0x9d, 0x84,
        0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05,
0xb8, 0xb3, 0x45, 0x06,
        0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03,
0x01, 0x13, 0x8a, 0x6b,
        0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce,
0xf0, 0xb4, 0xe6, 0x73,
        0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8,
0x1c, 0x75, 0xdf, 0x6e,
        0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e,
0xaa, 0x18, 0xbe, 0x1b,
        0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe,
0x78, 0xcd, 0x5a, 0xf4,
        0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59,
0x27, 0x80, 0xec, 0x5f,
        0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f,
0x93, 0xc9, 0x9c, 0xef,
        0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c,
0x83, 0x53, 0x99, 0x61,
        0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63,
0x55, 0x21, 0x0c, 0x7d };

static const uint8_t const_key[257] =
"\x43\x72\x79\x70\x74\x6F\x46\x41\x49\x4C\x55\x52\x45\x66\x6F\x72"

"\x52\x53\x41\x32\x30\x34\x38\x4B\x65\x79\x21\x21\x21\x21\x21\x21"

"\xBF\x8F\x84\x8D\xCB\xE0\xC2\xCC\x82\xAC\x97\x9E\xC7\xCA\xF8\xEC"

"\x94\x27\x00\xFC\xA4\x13\x38\xB7\xC1\x6A\x19\x96\xE0\x4B\x38\xB7"

"\x0E\x88\x2D\x6C\xC5\x68\xEF\xA0\x47\xC4\x78\x3E\x80\x0E\x80\xD2"

"\x59\x8C\xCD\x49\xFD\x9F\xF5\xFE\x3C\xF5\xEC\x68\xDC\xBE\xD4\xDF"

"\xA4\xC0\xB3\xEA\x61\xA8\x5C\x4A\x26\x6C\x24\x74\xA6\x62\xA4\xA6"

"\x7D\x26\x84\x6D\x80\xB9\x71\x93\xBC\x4C\x9D\xFB\x60\xF2\x49\x24"

"\x25\xFB\x85\x3A\x44\x53\xD9\x70\x62\x3F\xFD\x04\xC4\x5D\x59\xA2"

"\x61\x6A\x4F\x57\xE1\xD3\x3E\xC4\x5D\x9F\xA3\x3F\x3D\x6D\xEA\x1B"

"\x09\x7C\x2A\x1D\x4D\x2F\xF3\x6D\x2F\x10\x0E\x69\xEB\x4D\x57\xCB"

"\x88\x89\x14\x48\x69\x5A\x2A\x8C\x34\xC5\x89\xB3\x09\xA8\x63\xA8"

"\xEB\x87\xE8\x1C\xA6\xA8\x1B\x71\x89\xB8\x15\x18\x62\xF5\x42\xD3"

"\x22\x6F\x38\x2E\x4B\x35\x12\xA2\x7F\xF0\x9B\x11\x76\x58\xF8\xB9"
```

```
"\xC1\xC6\xBE\x24\x67\x6E\xA5\x55\xEE\xD6\xB0\x4D\x8C\x23\xF2\x9E"

"\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0A\x0B\x0C\x0D\x0E\x0F";
static const uint8_t const_iv[17] =
"\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0A\x0B\x0C\x0D\x0E\x0F";
static const uint8_t const_data[65] =
"\x91\xB3\xC1\xEB\x14\x5D\xD5\xCE\x3A\x1D\x30\xE4\x70\x6C\x6B\xD7"

 "\x69\x78\x79\x02\xA3\xA5\xDF\x1B\xFD\x1C\x02\x89\x14\x20\x7A\xFD"

 "\x24\x52\xF8\xA9\xF9\xF1\x6B\x1C\x0F\x5D\x50\x5B\xEC\x42\xD1\x8C"

 "\xB8\x12\xCF\x2C\xA9\x69\x31\x46\xFD\x9B\xEA\xDE\xC8\xBF\x94\x69";
//
"\x92\x16\x48\xBB\x9F\x81\xE0\x47\x7D\xEB\x40\x70\x07\x9E\x2A\x87"
//
"\x87\xC4\x00\xCB\x2F\xC2\x09\x4C\x9E\x4B\x72\xC6\x84\xDE\x54\x7E"
//
"\xAB\x5A\xB8\x46\x0A\xBD\x04\x1E\xFE\x62\x0A\xCC\xFB\xA0\x99\xA5"
//
"\xA2\x83\xF4\x83\x64\x6C\xC1\xE5\xD2\x37\x9E\xC9\x9D\xF0\x28\x6C";


static void InvMixRows(uint8_t *state);
static uint8_t xtime(uint8_t x);
static void xorKey(uint8_t *state,uint8_t *key);
static void invColShift(uint8_t*state);
static void invSbox(uint8_t* state);

int main() {
    uint8_t iv[16],key[256],data[64];
    memcpy(iv,const_iv,16);
    memcpy(key,const_key,256);
    memcpy(data,const_data,64);



    for(int i = 0;i < 4;i++){
        for(int j = 0;j < 14;j++){
            xorKey(data+(3-i)*16,key+(14-j)*16);
            if(j != 0)
                InvMixRows(data+(3-i)*16);
            invColShift(data+(3-i)*16);
            invSbox(data+(3-i)*16);
        }
        xorKey(data+(3-i)*16,key);
        if(2-i>=0)
            xorKey(data+(3-i)*16,data+(2-i)*16);
        else
            xorKey(data+(3-i)*16,iv);
    }
    for(int i = 0;i < 64; i++){
        printf("%c",*(data+i));
    }

//    uint8_t test[4][4] = {0};
//    test[0][0] = 0xF1^0x72^0xEB^xtime(0x77^0xF1);
```

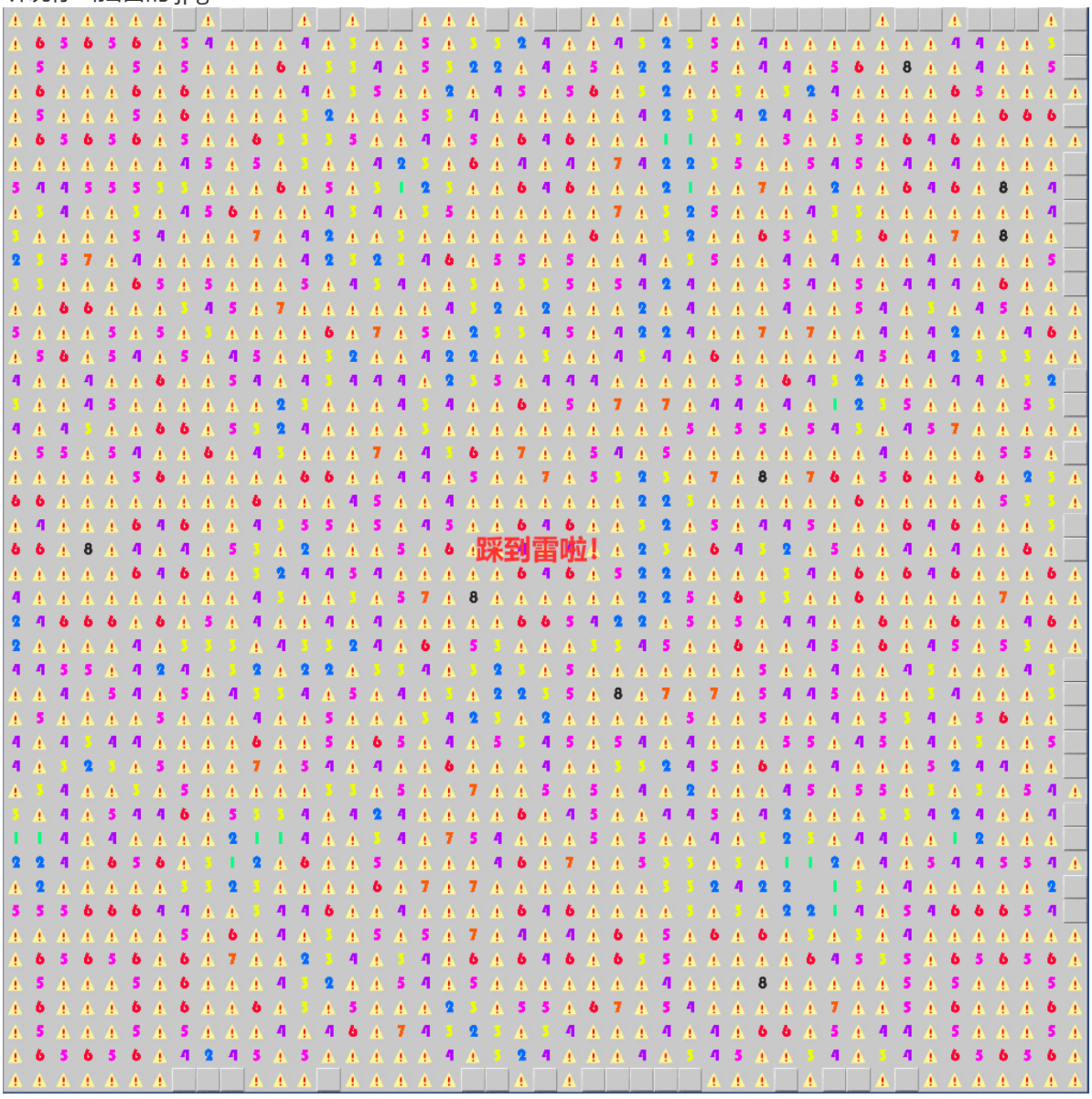
```
//    test[0][1] = 0x77^0x72^0xEB^xtime(0xF1^0x72);
//    test[0][2] = 0x77^0xF1^0xEB^xtime(0x72^0xEB);
//    test[0][3] = 0x77^0xF1^0x72^xtime(0xEB^0x77);
//    InvMixRows((uint8_t*)test);


    return 0;
}

static void xorKey(uint8_t *state,uint8_t *key){
    for(int i = 0;i < 16;i++){
        *(state+i) ^= *(key+i);
    }
}
static void invColShift(uint8_t*state){
    uint8_t a   = *(state+1);
    uint8_t b1  = *(state+2);
    uint8_t b2  = *(state+6);
    uint8_t c   = *(state+3);
    *(state+1)  = *(state+13);
    *(state+13) = *(state+9);
    *(state+9)  = *(state+5);
    *(state+5)  = a;
    *(state+2)  = *(state+10);
    *(state+10) = b1;
    *(state+6)  = *(state+14);
    *(state+14) = b2;
    *(state+3)  = *(state+7);
    *(state+7)  = *(state+11);
    *(state+11) = *(state+15);
    *(state+15) = c;
}
static void invSbox(uint8_t* state){
    for(int i = 0;i < 16;i++){
        *(state+i) = rsbox[*(state+i)];
    }
}
static void InvMixRows(uint8_t *state)
{
    int i;
    int8_t a, b, c, d;
    for (i = 0; i < 4; ++i)
    {
        a = (__int8)*(state + 4 * i);
        b = (__int8)*(state + 4 * i + 1);
        c = (__int8)*(state + 4 * i + 2);
        d = (__int8)*(state + 4 * i + 3);

        *(state + 4 * i)     = Multiply(a, 0x0e) ^ Multiply(b, 0x0b) ^
Multiply(c, 0x0d) ^ Multiply(d, 0x09);
        *(state + 4 * i + 1) = Multiply(a, 0x09) ^ Multiply(b, 0x0e) ^
Multiply(c, 0x0b) ^ Multiply(d, 0x0d);
        *(state + 4 * i + 2) = Multiply(a, 0x0d) ^ Multiply(b, 0x09) ^
Multiply(c, 0x0e) ^ Multiply(d, 0x0b);
        *(state + 4 * i + 3) = Multiply(a, 0x0b) ^ Multiply(b, 0x0d) ^
Multiply(c, 0x09) ^ Multiply(d, 0x0e);
```

```
    }
}
static uint8_t xtime(uint8_t x)
{
    return ((x<<1) ^ (((x>>7) & 1) * 0x1b));
}
//hgame{cOsm0s_is_still_fight1ng_and_NEVER_GIVE_UP_OOoOoOoOOOoOOo}
```

3.A 5 Second Challenge

il2cpp逆向题目，没想到我技能树点歪了

先上图：
听说你叫扫雷的.jpg

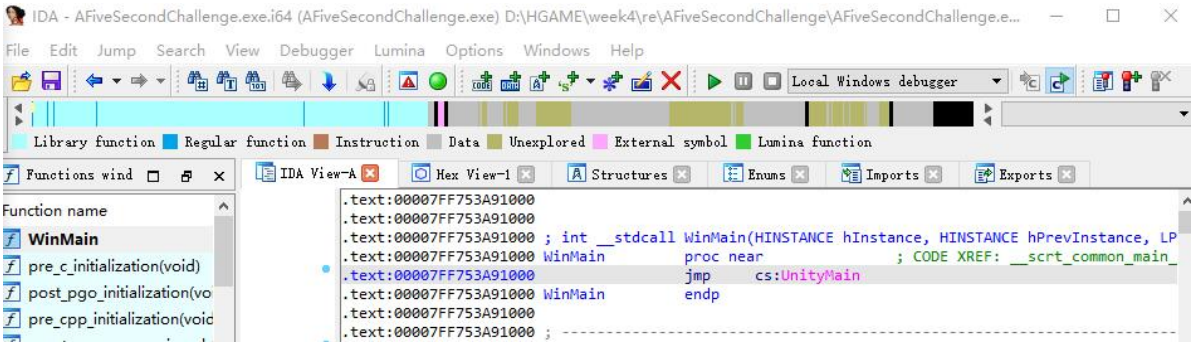


其实我玩扫雷很菜，但是炸弹炸不死我 2333
为了破解这个东西，我干了好多活，最累的其实是点开 💣，再其次是画二维码。

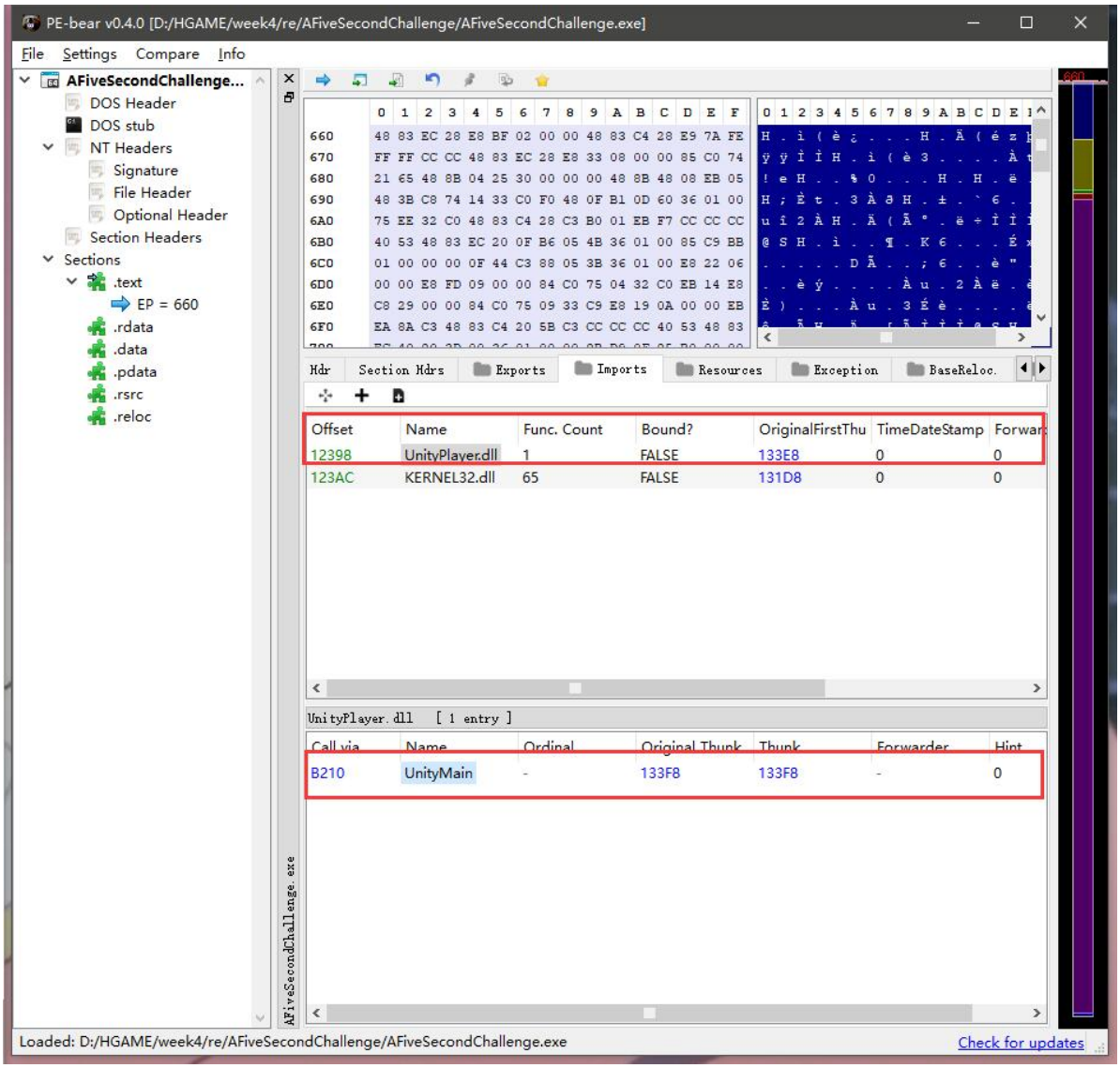
那我现在从头开始讲。

由于开始时对unity的一些文件配置完全不懂，导致我看的晕头转向。不过看了别人逆向的一些实例，我大概理解了文件结构。那我写长一点，从启动函数开始。

众所周知为了玩游戏，点开了游戏，但是游戏 .exe 里面其实并没有游戏的逻辑

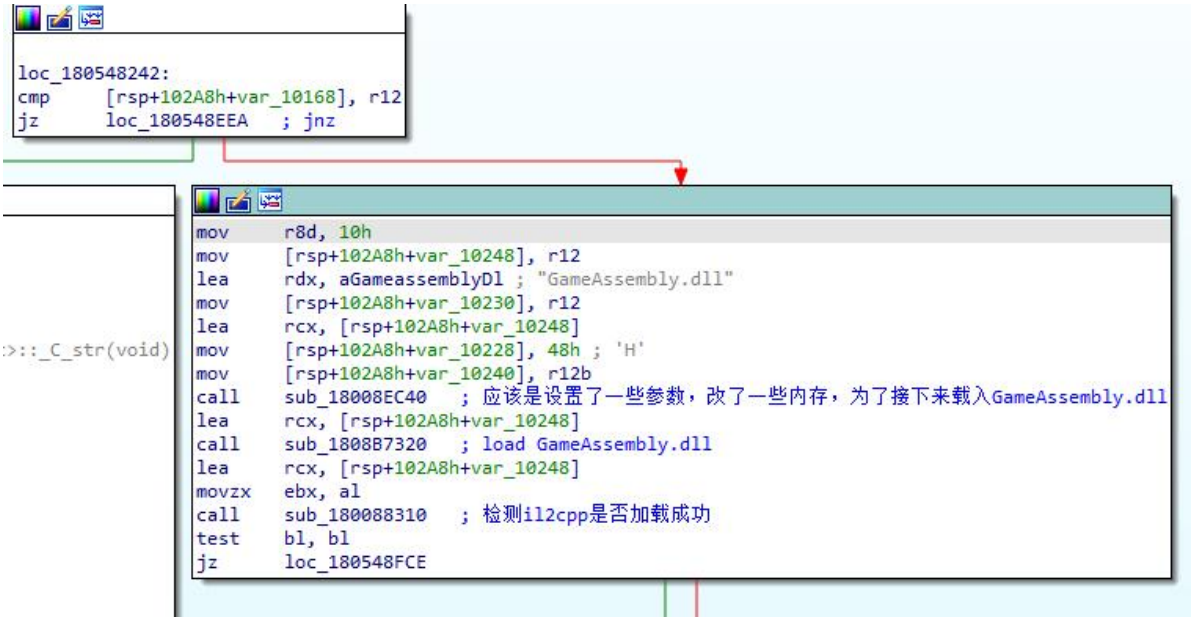


直接跳转到了 UnityMain 函数中，而这个函数在别的 dll 中，我们可以打开 PE-bear 其实也可以自己对着节区去看，不过算起来比较复杂，有现成的工具用一下也挺好



然后我们就开始对 UnityPlayer 释放魔法（动态调试），发现 UnityPlayer 其实是通过 CreatWindowExW 创建句柄，传递给 showWindow，最终打开 unity 窗口。其中 sub0x180547E70 可以称为 Load_Game_function （我自己这么定义的 233 ）

其中 sub_180088310 函数常用于检测或者加载是否加载成功一些文件，比如 app.info , config , Play.log 等文件，但是我们仍然找不到游戏逻辑，继续追踪发现在



此处加载了 GameAssembly.dll 加载成功后会 继续加载一些文件比如 $(game)/game_data/il2cpp_data/etc , 再后来也会加载一些 creen 相关的模块， 不顾哦这个我也不懂，过了一段函数，会加载一堆 dll， 最后就是打开窗口 showWindow . 再进一步就可以调试 GameAssembly.dll 结合网上所说，GameAssembly.dll 中包含着游戏的逻辑，我对他就更加期待了起来。

打开会发现很多名为 il2cpp_XXX_XXX_XXX 的函数，查阅一些别人的博客，再加自己的动态调试，加看 IL 源码， 我也稍微理解了一些il2cpp_xxx 函数的作用， 因为是从 C# -> CPP -> PE 他面向对象的方法，思想也就变成了 il2cpp_xxx 的一些函数，而在 il2cpp 源码中比如这一段

```
// System.Boolean AFiveSecondChallenge.BombChecker::CheckIfExpired()
IL2CPP_EXTERN_C IL2CPP_METHOD_ATTR bool
BombChecker_CheckIfExpired_m11965D79AB8C709FB9F999ACFFA108438A43652E (const
RuntimeMethod* method)
{
    static bool s_Il2CppMethodInitialized;
    if (!s_Il2CppMethodInitialized)
    {
        il2cpp_codegen_initialize_method
(BombChecker_CheckIfExpired_m11965D79AB8C709FB9F999ACFFA108438A43652E_MetadataUs
ageId);
        s_Il2CppMethodInitialized = true;
    }
    {

IL2CPP_RUNTIME_CLASS_INIT(BombChecker_t6F78547653A30303197DE752136BFF107B035342_
il2cpp_TypeInfo_var);
        int64_t L_0 =
((BombChecker_t6F78547653A30303197DE752136BFF107B035342_StaticFields*)il2cpp_cod
egen_static_fields_for(BombChecker_t6F78547653A30303197DE752136BFF107B035342_il2
cpp_TypeInfo_var))->get_firstQueryTime_4();
        if (!L_0)
        {
            goto IL_001b;
        }
    }
    {
```

```
IL2CPP_RUNTIME_CLASS_INIT(BombChecker_t6F78547653A30303197DE752136BFF107B035342_
il2cpp_TypeInfo_var);
        int64_t L_1 =
BombChecker_GetNowUnixTime_m02658B00EEC2CD986B3894FB392B39BB186716A8(/*hidden
argument*/NULL);
        int64_t L_2 =
((BombChecker_t6F78547653A30303197DE752136BFF107B035342_StaticFields*)il2cpp_cod
egen_static_fields_for(BombChecker_t6F78547653A30303197DE752136BFF107B035342_il2
cpp_TypeInfo_var))->get_firstQueryTime_4();
        int32_t L_3 =
((BombChecker_t6F78547653A30303197DE752136BFF107B035342_StaticFields*)il2cpp_cod
egen_static_fields_for(BombChecker_t6F78547653A30303197DE752136BFF107B035342_il2
cpp_TypeInfo_var))->get_expireIn_5();
        return (bool)((((int64_t)((int64_t)il2cpp_codegen_subtract((int64_t)L_1,
(int64_t)L_2))) > ((int64_t)(((int64_t)((int64_t)L_3)))))? 1 : 0);
    }

IL_001b:
    {
        return (bool)0;
    }
}
```

每次调用一些函数之前一般会使用 il2cpp_runtime_class_init （猜测是创建对象），然后使用 il2cpp_substract_xxx 等函数实现对对象的操作，非常奇妙的感觉，由此可以看出 **只要我们在 il2cpp_runtime_class_init 下断点，我们就能知道 CheckIfExpired，通过交叉引用能找到。**

上面所说不失为一种方法，但是事实是，即使是调试到 只有点击地雷才会 碰到断点的时候，通过条件断点，日志，我们能发现 调用了 15次 il2cpp_runtime_class_init 而从一开始运行的话调用了将近 50 次 所以通过这个方法 我们往往会偏离目标，而且找到函数之后 我们还要根据函数的逻辑去比对 il2cpp 源码 实在是离谱。由此 我们引入老大哥。 ***global-metadata.dat***

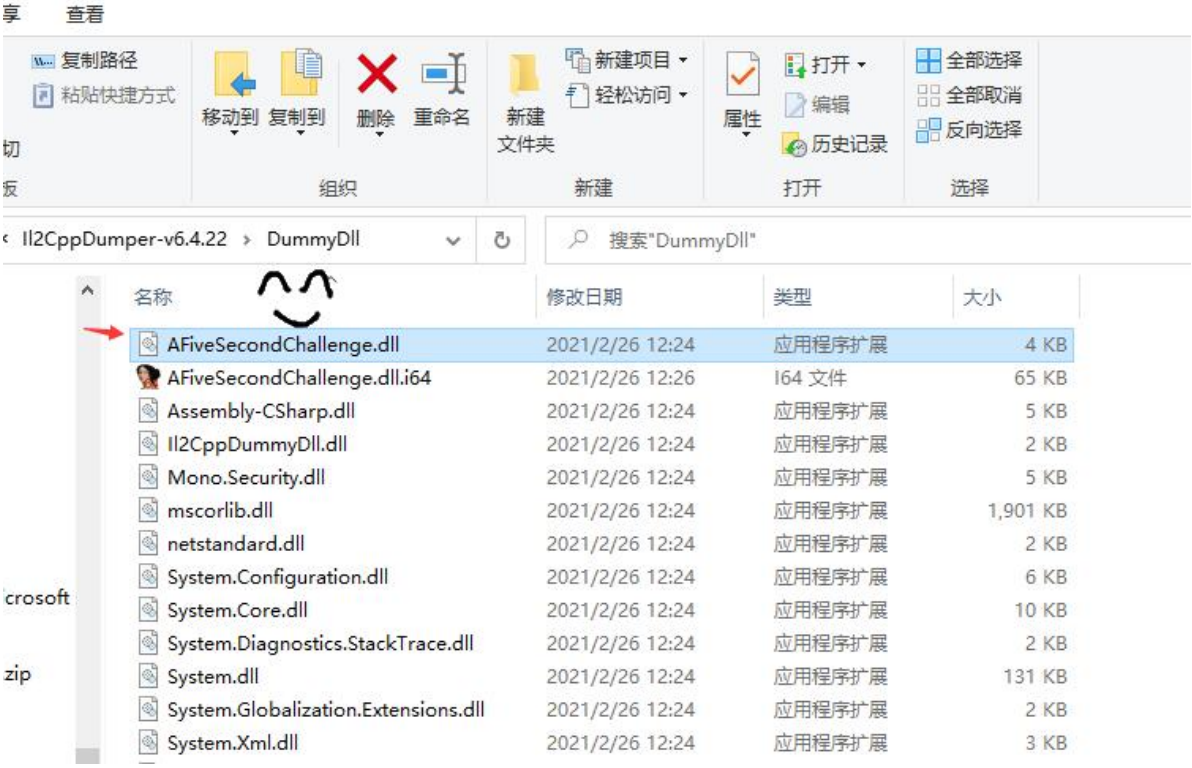
这个文件记载了一些被调用的字符串，还有函数名以及 偏移，这个文件有一个结构体，其实可以mark一下

```
struct Il2CppGlobalMetadataHeader
{
    int32_t sanity;
    int32_t version;
    int32_t stringLiteralOffset; // string data for managed code
    int32_t stringLiteralCount;
    int32_t stringLiteralDataOffset;
    int32_t stringLiteralDataCount;
    int32_t stringOffset; // string data for metadata
    int32_t stringCount;
    int32_t eventsOffset; // Il2CppEventDefinition
    int32_t eventsCount;
    int32_t propertiesOffset; // Il2CppPropertyDefinition
    int32_t propertiesCount;
    int32_t methodsOffset; // Il2CppMethodDefinition
    int32_t methodsCount;
    ...
}
```

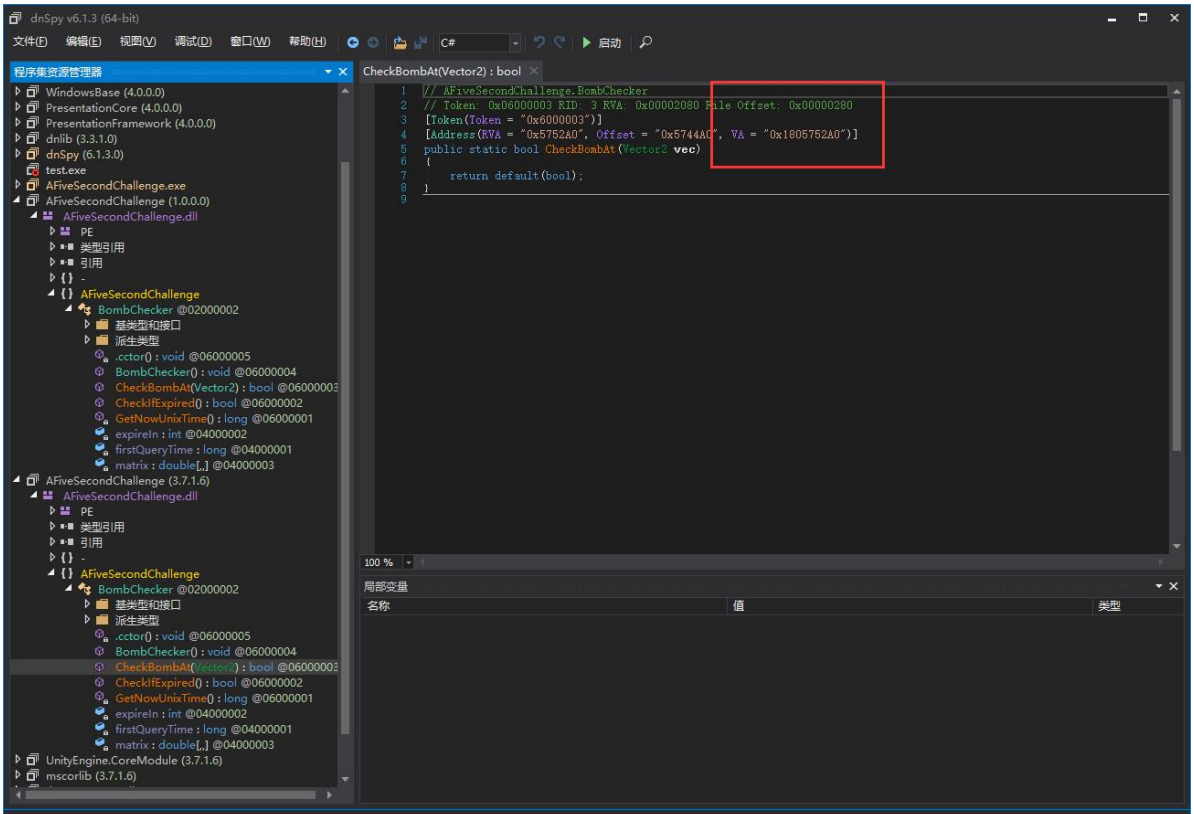
然后通过调试得知，估计这个 Metadata 文件是在 UnityPlayer.dll 中载入的，反之我们也能从中读取到一些东西，甚至网上已经有了可以 dump Metadata 的工具 Il2cppDumper
hint 2: il2cpp 中间文件（源码）已经直接给了，就不太需要用 il2cppdumper 这类工具去死怼 GameAssembly.dll 了 囧

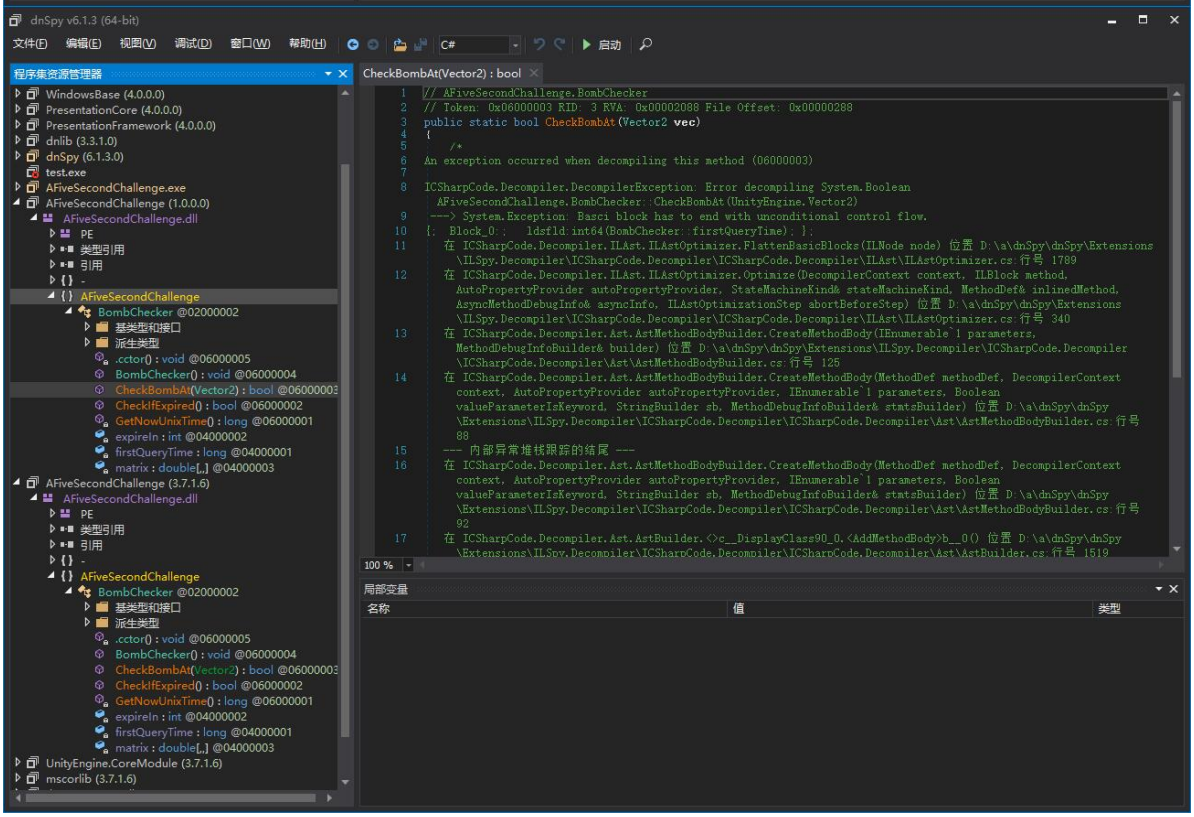
其实我不知道魔法少女为什么会给这个hint，估计是担心我们白怼 GameAssembly.dll ，其实正常的思路应该是去解密那个地雷矩阵，不过我一脑子扎进去就没停下来，最后造成这个结局 233。（其实 IL2CPP 读起来有点反人类，括号太多了，看着就烦...）其实这里有一个小 Tip 待会儿再讲

通过 Il2cppDumper 我们可以得到源码的 dll , 不过并没有具体的内容，大多只是有 偏移地址之类的，不过这样也足够了



再通过 dnspy 打开 这个dll

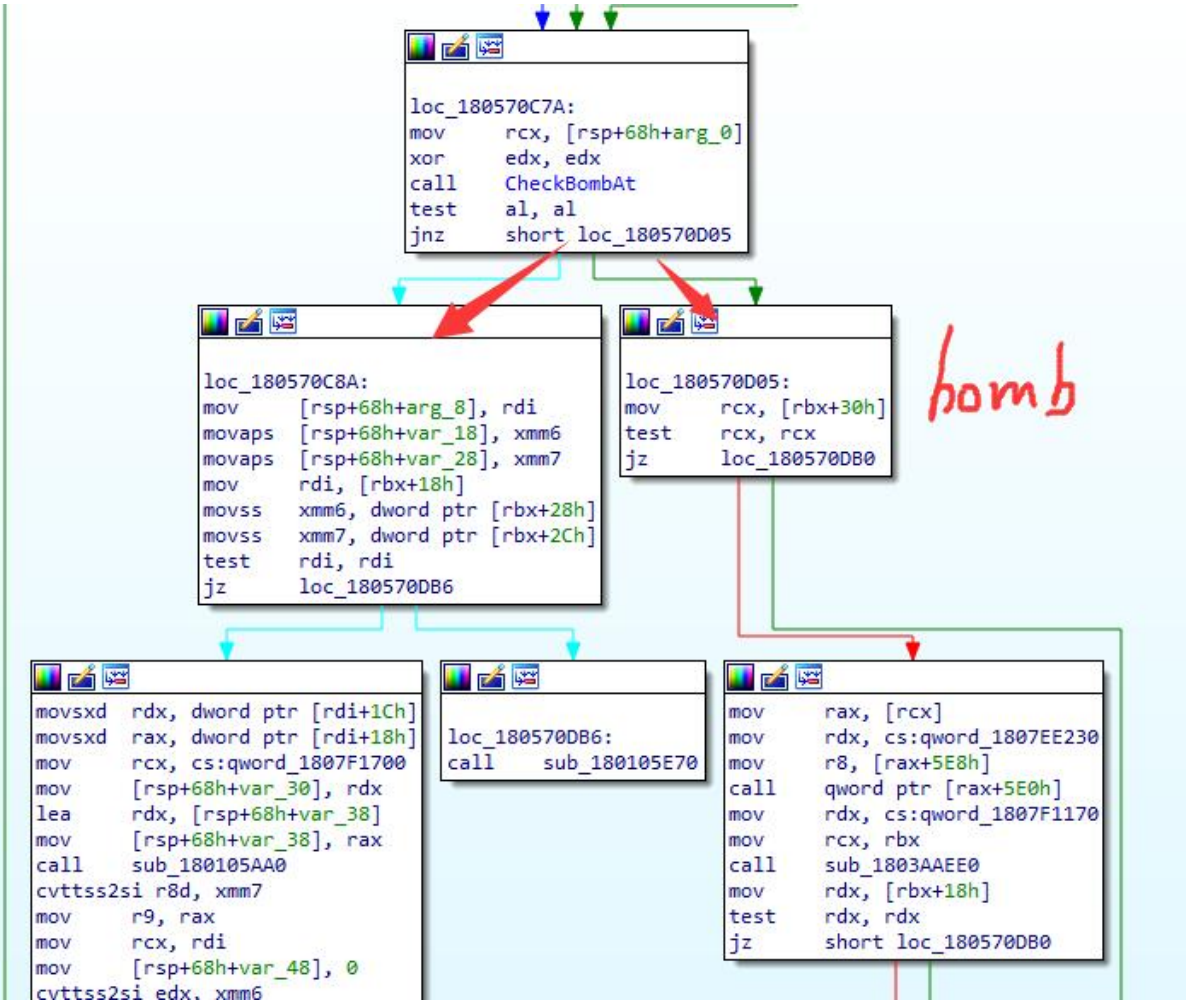




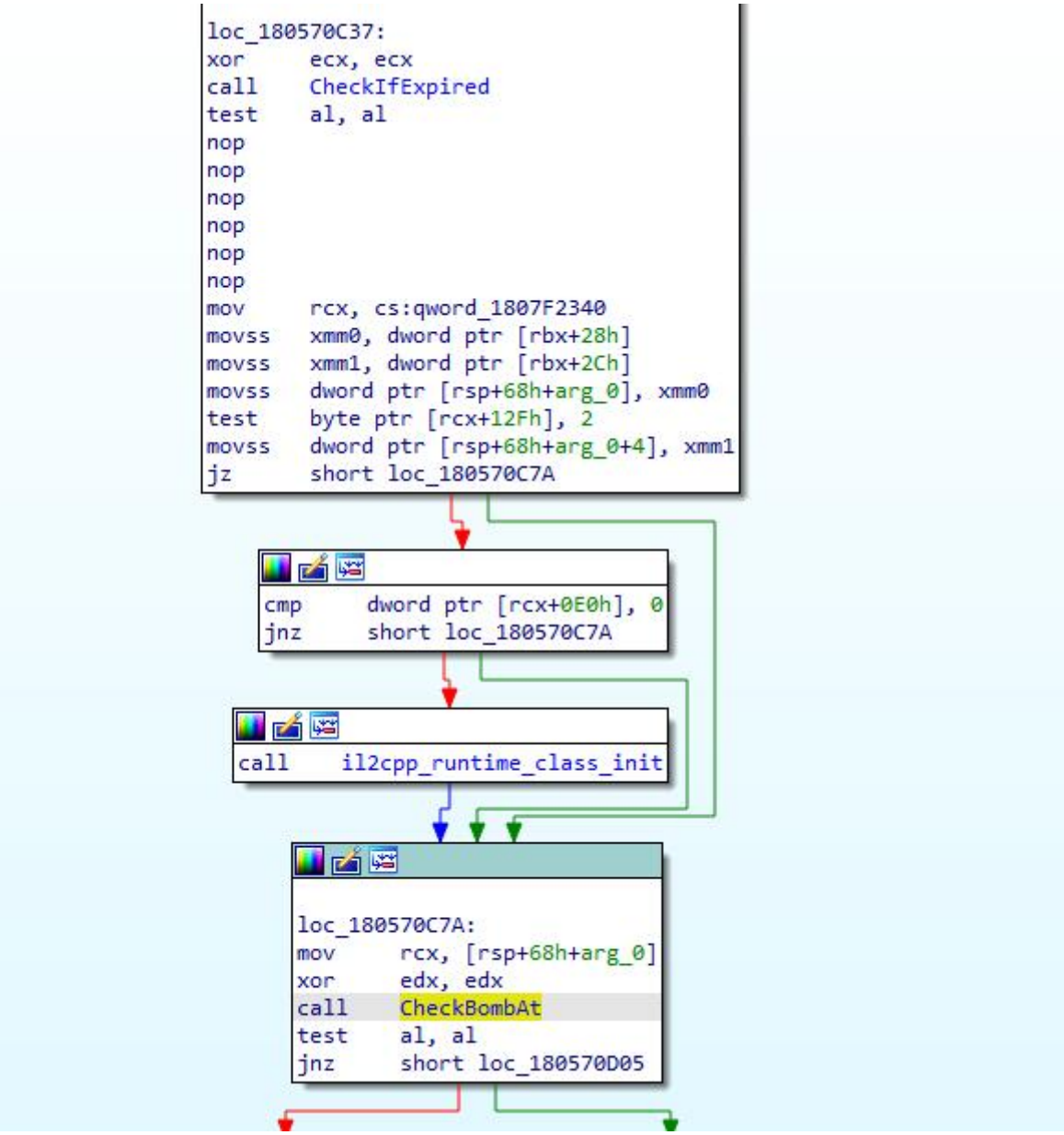
两个对比很明显 我们得到了函数地址， 最后就开始了魔改环节 。

这个环节试错比较多，我也慢慢讲

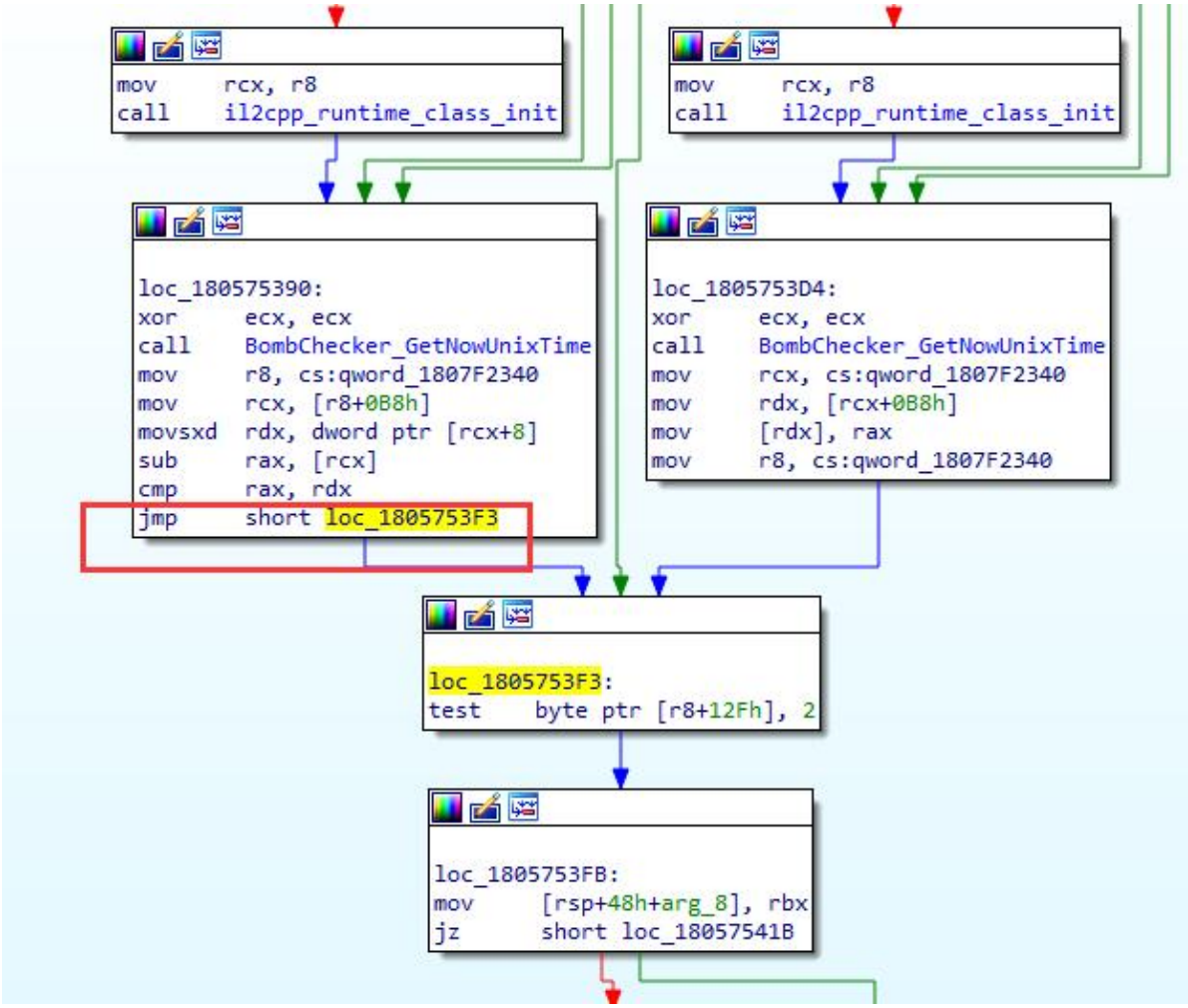
在 CS_Mouse_Brick 函数中我们会发现 是否发现炸弹的两条分路



而在此之前我们会先检测一遍时间，我当场就把他的对我不好的逻辑 给 nop 掉了



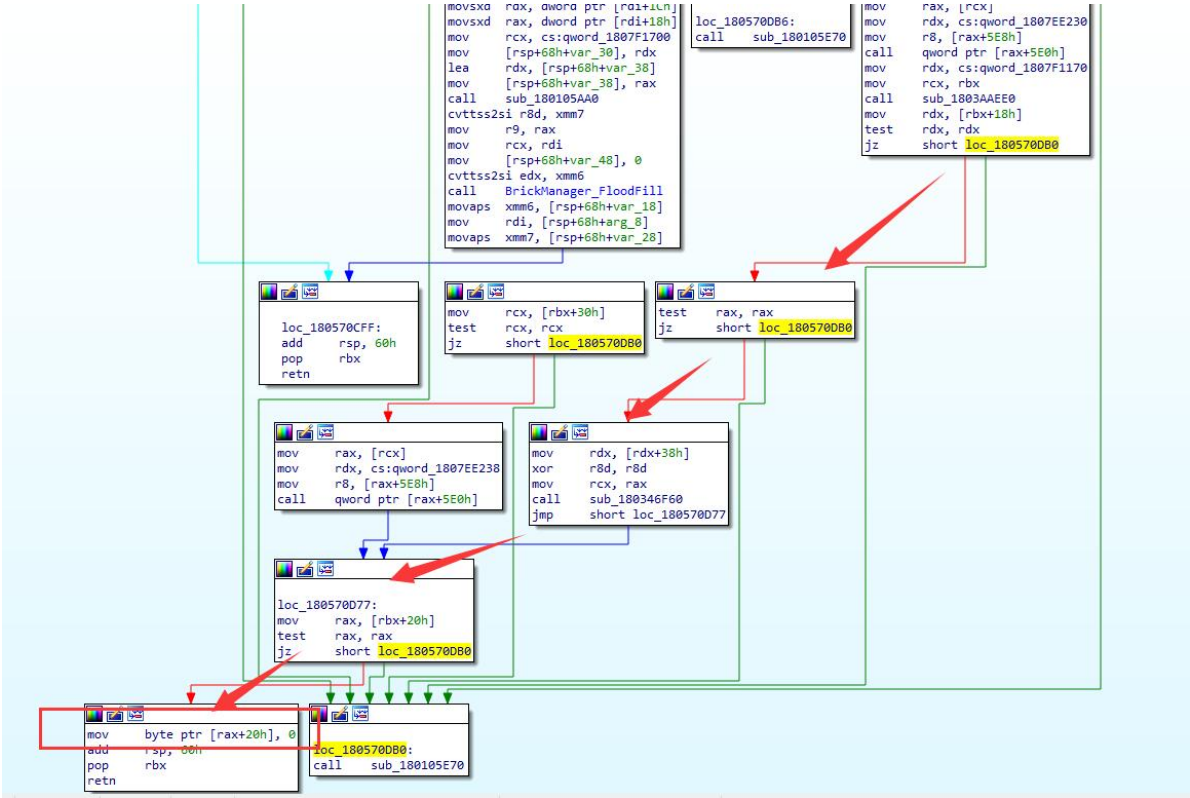
只改了这一处 会发现 时间过了之后，不管点什么 他都是炸弹！ 动调了一下 发现了原因



这一段在 CheckBombAt 函数中，检测到超时之后，这里本来会指向别的地方，最终造成 点啥都是炸弹，我就把这里两个逻辑都汇聚在一起，这样这俩就都是魔法少女了。

最后我们还需修改一处，就是当程序发现你点了雷，他就会终止你接下来的行为。

而这个逻辑是在



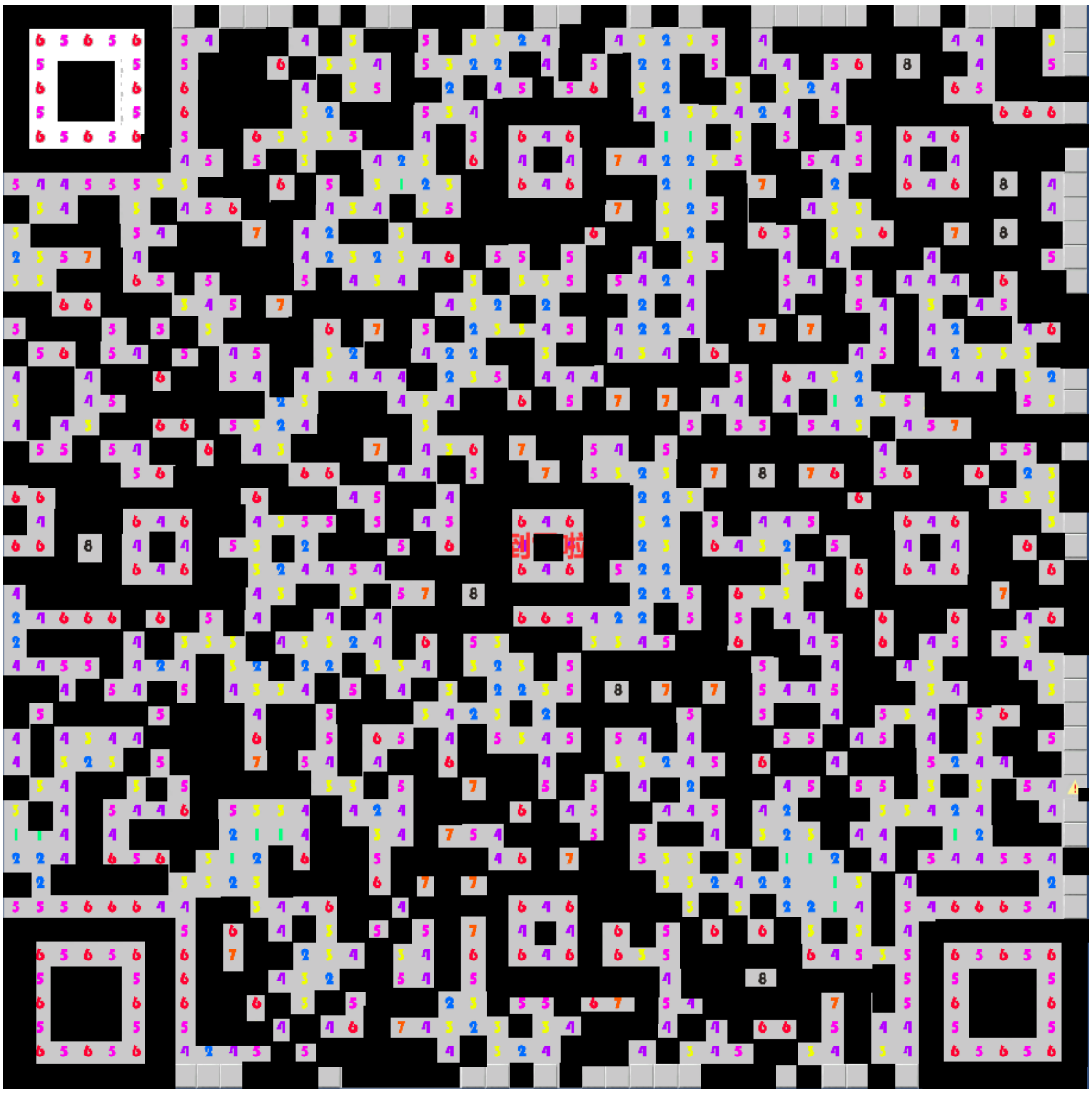
如果要退出他会置 1 那我反过来不让他退出，所以我就置 0；最后就成功地劫持了程序流

话说我刚才说了 （劫持） 。。。我这明明是 （抢劫）

pwn里面改不了程序流，大多通过溢出来劫持，因此称我这个为 （抢劫） 一点不过分。

然后就是手酸的点地雷了 233 ， 再是手酸的绘制 二维码了

最后结果



其实只要缩小一下就能扫出来，还好二维码可以有误差，不然我只好裂开。

手~~~好酸啊，我下次一定去看矩阵。。。

好了，最后我再收回我说的 Tip 我们看回 il2cpp 的源码

```
    {

IL2CPP_RUNTIME_CLASS_INIT(BombChecker_t6F78547653A30303197DE752136BFF107B035342_
il2cpp_TypeInfo_var);
        DoubleU5BU2CU2CU5D_t52D90E91F26D35646AF02F1D1DF3770A40CD9B7D* L_3 =
((BombChecker_t6F78547653A30303197DE752136BFF107B035342_StaticFields*)il2cpp_cod
egen_static_fields_for(BombChecker_t6F78547653A30303197DE752136BFF107B035342_il2
cpp_TypeInfo_var))->get_matrix_6();
        Vector2_tA85D2DD88578276CA8A8796756458277E72D073D  L_4 = ___vec0;
        float L_5 = L_4.get_y_1();
```

```
        Vector2_tA85D2DD88578276CA8A8796756458277E72D073D  L_6 = ___vec0;
        float L_7 = L_6.get_x_0();

NullCheck((DoubleU5BU2CU2CU5D_t52D90E91F26D35646AF02F1D1DF3770A40CD9B7D*)
(DoubleU5BU2CU2CU5D_t52D90E91F26D35646AF02F1D1DF3770A40CD9B7D*)L_3);
        double L_8 =
((DoubleU5BU2CU2CU5D_t52D90E91F26D35646AF02F1D1DF3770A40CD9B7D*)
(DoubleU5BU2CU2CU5D_t52D90E91F26D35646AF02F1D1DF3770A40CD9B7D*)L_3)-
>GetAt((((int32_t)((int32_t)L_5))), ((int32_t)((int32_t)(((int32_t)
((int32_t)L_7)))/(int32_t)3)), 0);
        DoubleU5BU2CU2CU5D_t52D90E91F26D35646AF02F1D1DF3770A40CD9B7D* L_9 =
((BombChecker_t6F78547653A30303197DE752136BFF107B035342_StaticFields*)il2cpp_cod
egen_static_fields_for(BombChecker_t6F78547653A30303197DE752136BFF107B035342_il2
cpp_TypeInfo_var))->get_matrix_6();
        Vector2_tA85D2DD88578276CA8A8796756458277E72D073D  L_10 = ___vec0;
        float L_11 = L_10.get_y_1();
        Vector2_tA85D2DD88578276CA8A8796756458277E72D073D  L_12 = ___vec0;
        float L_13 = L_12.get_x_0();

NullCheck((DoubleU5BU2CU2CU5D_t52D90E91F26D35646AF02F1D1DF3770A40CD9B7D*)
(DoubleU5BU2CU2CU5D_t52D90E91F26D35646AF02F1D1DF3770A40CD9B7D*)L_9);
        double L_14 =
((DoubleU5BU2CU2CU5D_t52D90E91F26D35646AF02F1D1DF3770A40CD9B7D*)
(DoubleU5BU2CU2CU5D_t52D90E91F26D35646AF02F1D1DF3770A40CD9B7D*)L_9)-
>GetAt((((int32_t)((int32_t)L_11))), ((int32_t)((int32_t)(((int32_t)
((int32_t)L_13)))/(int32_t)3)), 1);
        V_0 = L_14;
        DoubleU5BU2CU2CU5D_t52D90E91F26D35646AF02F1D1DF3770A40CD9B7D* L_15 =
((BombChecker_t6F78547653A30303197DE752136BFF107B035342_StaticFields*)il2cpp_cod
egen_static_fields_for(BombChecker_t6F78547653A30303197DE752136BFF107B035342_il2
cpp_TypeInfo_var))->get_matrix_6();
        Vector2_tA85D2DD88578276CA8A8796756458277E72D073D  L_16 = ___vec0;
        float L_17 = L_16.get_y_1();
        Vector2_tA85D2DD88578276CA8A8796756458277E72D073D  L_18 = ___vec0;
        float L_19 = L_18.get_x_0();

NullCheck((DoubleU5BU2CU2CU5D_t52D90E91F26D35646AF02F1D1DF3770A40CD9B7D*)
(DoubleU5BU2CU2CU5D_t52D90E91F26D35646AF02F1D1DF3770A40CD9B7D*)L_15);
        double L_20 =
((DoubleU5BU2CU2CU5D_t52D90E91F26D35646AF02F1D1DF3770A40CD9B7D*)
(DoubleU5BU2CU2CU5D_t52D90E91F26D35646AF02F1D1DF3770A40CD9B7D*)L_15)-
>GetAt((((int32_t)((int32_t)L_17))), ((int32_t)((int32_t)(((int32_t)
((int32_t)L_19)))/(int32_t)3)), 2);
        V_1 = L_20;
        Vector2_tA85D2DD88578276CA8A8796756458277E72D073D  L_21 = ___vec0;
        float L_22 = L_21.get_x_0();
        V_2 = (((double)((double)((float)il2cpp_codegen_subtract((float)
(fmodf(L_22, (3.0f))), (float)(1.0f))))));
        double L_23 = V_2;
        double L_24 = V_2;
        double L_25 = V_0;
        double L_26 = V_2;
        double L_27 = V_1;
        //此注释没什么用，请忽略
        //      L_8*L_23*L_24+L_25*L_26+L_27
        // L_8  = GetAt(y1,x0/3,0)
        // L_23 = x0 fmodf 3 -1
        // L_24 = x0 fmodf 3 -1
```

```
        // L_25 = GetAt(y1,x0/3,1)
        // L_26 = x0 fmodf 3 -1
        // L_27 = GetAt(y1,x0/3,1)
        return (bool)((((double)((double)il2cpp_codegen_add((double)
((double)il2cpp_codegen_add((double)((double)il2cpp_codegen_multiply((double)
((double)il2cpp_codegen_multiply((double)L_8, (double)L_23)), (double)L_24)),
(double)((double)il2cpp_codegen_multiply((double)L_25, (double)L_26)))),
(double)L_27))) > ((double)(0.0)))? 1 : 0);
    }
```

我们可以很快发现这个是判断是否是地雷的源码，但是很明显，这也太难看懂了，因此我建议，以后反调试，只要把 cpp -> c# -> il -> cpp 就好了，保证大部分人看裂开，但是我们其实这里有另外一种思维，源码看不懂，那看逆向 出来的 伪代码 呗。

ida 中可以有

```
return v9 * v11 * v11 + v11 * *(double *)(v2 + 8 * v10 + 40) + *(double *)(v2 +
8 * v10 + 48) > 0.0;
```

我们再结合起来看其实就方便了很多。