看雪ctf部分题解

原创

合天网安实验室 于 2018-07-18 20:08:00 发布 393 收藏

点击上方蓝色字体，关注我们

15

首先申明，我在遇到疑难点时参考了部分wp（咱还是菜啊）。

第一题没啥好说的，直接拖ida以十六进制输出就行了

```
static void Main(string[] args)
{
    int[] a = new int[] { 0x43, 0x72, 0x61, 0x63 ,0x6b ,0x4d, 0x65, 0x4a ,0x75, 0x73, 0x74 ,0x46 ,0x6f, 0x72 ,0x46, 0x75 , 0x6e };

    for(int i = 0; i < 17; i++)
    {
      // int d = 16 * ((i  % 3)+1);
       //Console.Write(d + " ");
       //int c = a[i] ;
      // Console.Write(c + " ");
       int e =Convert.ToInt32(a[i]);
       Console.Write(Convert.ToChar(e));
       //Console.Write(Convert.ToChar(c));

    }
    Console.
}
```

C:\Program Files\dotnet\dotnet.exe
CrackMeJustForFun

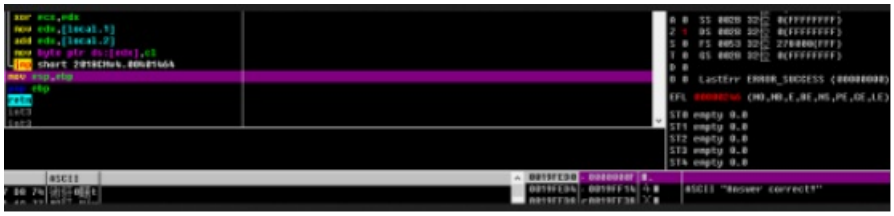
第二题名字叫数据结构，那肯定涉及到部分数据结构的知识，无壳无反调，发现有aslr，为了方便动静结合，用ffi去掉aslr（自我感觉这个习惯良好（滑稽）），把dll标和基础重定位目录归0就行了。

直接拖入ida，看下字符串，发现没有成功或者失败的提示，并且在一些提示信息中可以百度到数据结构是字典树

然后去看下main函数，可以看到一些call，但是不知道是做什么的，去动调看下就知道了

```
13
14  sub_401170("%s", input, 32);
15  sub_401360(&v5);
16  sub_401200(&v9);
17  if ( input_check(input, (int)&v9) )
18  {
19    if ( &input[strlen(input) + 1] - &input[1] == 22 )
20      check(input, (int)&v5, (int)&v9);
21    else
22      sub_4011D0((int)&v9);
23    v9 = 0;
24    v10 = 0;
25    v11 = 0;
26    v12 = 0;
27    v5 = 0;
28    v6 = 0;
29    v7 = 0;
30    v8 = 0;
31    system("pause");
32    result = 0;
33  }
34  else
35  {
36    v5 = 0;
37    v6 = 0;
38    v7 = 0;
39    v8 = 0;
40    v9 = 0;
41    v10 = 0;
42    v11 = 0;
43    v12 = 0;
44    result = 0;
45  }
46  return result;
47}
```

使用od调试观察返回值，可以看到经过401360和401200这两个函数之后栈中出现了成功和失败的提示字眼，可以认定是这两个函数是对success和fail进行了解码



然后继续跟，在对长度与22进行比较之后，传入了核心的check函数，构造22位假码输入，先静态看下check方法了解下大概流程，可以发现这里把假码分成了8租，分别传入403AB0和402B40进行操作，动调跟下确定下分组

可以结合ida和动调返回值确定分组，整理下，可以得到如下分组，并且可以发现每一次都是对v39这个变量进行了操作，可以猜测v39是由输入的构成的字典树，然后传入了4030E0和一个未知类型的东西比较，可以看下这里的比较函数

```
 91  v39 = 0;
 92  v40 = 0;
 93  std::_Ref_count_obj<__ExceptionPtr>::_Ref_count_obj<__ExceptionPtr>(&v39);
 94  v52 = 0;
 95  sub_403AB0(&v11, &Src);                       // 输入的是14位（456）
 96  sub_402B40(&v39, (int)&v11);
 97  sub_403AB0(&v10, &v41);                       // 开头两位
 98  sub_402B40(&v39, (int)&v10);
 99  sub_403AB0(&v9, &v47);                        // 10-13位（0123）
100  sub_402B40(&v39, (int)&v9);
101  sub_403AB0(&v8, &v24);                        // 5-7位（567）
102  sub_402B40(&v39, (int)&v8);
103  sub_403AB0(&v7, &v28);                        // 34两位（3 4）
104  sub_402B40(&v39, (int)&v7);
105  sub_403AB0(&v6, &v44);                        // 8 9两位（89）
106  sub_402B40(&v39, (int)&v6);
107  sub_403AB0(&v5, &v35);                        // 17 18 19（7 8 9）
108  sub_402B40(&v39, (int)&v5);
109  sub_403AB0(&v4, &v20);                        // 20 21 22（0 1 2）
110  sub_402B40(&v39, (int)&v4);
111  if ( sub_4030E0(&v39, (int)&dword_407E48) )
112    sub_401B80(&v41, &v44, (int)&Src, (int)&v35, a2, a3);
113  else
```

```
1  char __thiscall sub_403730(int *this, int a2)
2  {
3    const char *v3; // eax
4    char v4; // [esp+0h] [ebp-A4h]
5    int *v5; // [esp+84h] [ebp-20h]
6    int v6; // [esp+88h] [ebp-1Ch]
7    int v7; // [esp+8Ch] [ebp-18h]
8    int *v8; // [esp+90h] [ebp-14h]
9    int v9; // [esp+94h] [ebp-10h]
0    int *v10; // [esp+98h] [ebp-Ch]
1    int *v11; // [esp+9Ch] [ebp-8h]
2    int *v12; // [esp+A0h] [ebp-4h]
3
4    v12 = this;
5    if ( sub_403D70((const char *)this + 4, a2 + 4) )
6      return 0;
7    v7 = v12[66];
8    v6 = *(_DWORD *)(a2 + 264);
9    if ( v7 != v6 )
0      return 0;
1    if ( v12[67] != *(_DWORD *)(a2 + 268) )
2      return 0;
3    v10 = v12 + 34;
4    v11 = v12 + 34;
5    v5 = &v12[v12[66] + 34];
6    while ( v11 != v5 )
7    {
8      v9 = *v11;
9      v3 = (const char *)sub_403670(v9, &v4);
0      v8 = (int *)sub_403830((int *)a2, v3);
1      if ( !v8 )
2        return 0;
3      if ( sub_4036D0(v8, v9) )                  // 递归调用403730进行字符比较
4        return 0;
5      ++v11;
6    }
7    return 1;
8  }
```

可以发现是一个递归比较的过程，类似于树的遍历，确保v39和目标生成树一摸一样，通过ida的交叉引用看下哪里初始化了407E48这个目标字典树结构

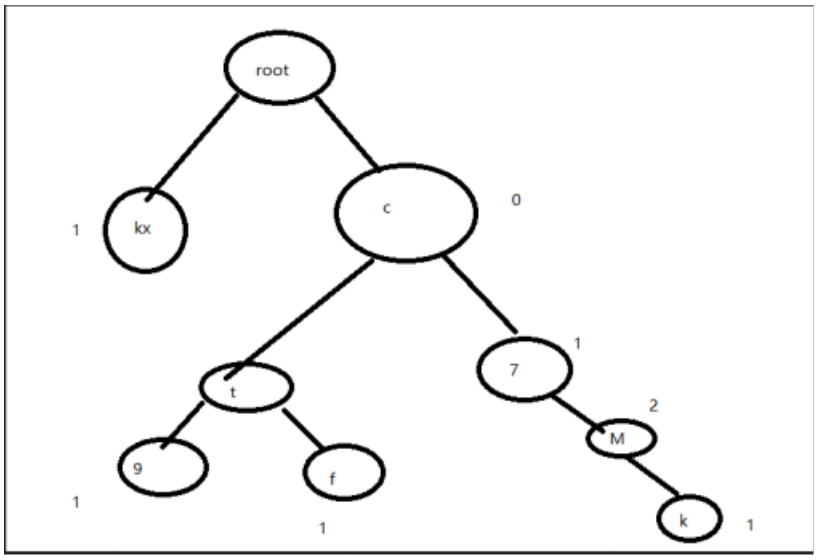
```
void *__thiscall sub_401900(void *this)
{
  char v2; // [esp+0h] [ebp-448h]
  char v3; // [esp+84h] [ebp-3C4h]
  char v4; // [esp+108h] [ebp-340h]
  char v5; // [esp+18Ch] [ebp-2BCh]
  char v6; // [esp+210h] [ebp-238h]
  char v7; // [esp+294h] [ebp-1B4h]
  char v8; // [esp+318h] [ebp-130h]
  char v9; // [esp+39Ch] [ebp-ACh]
  void *v10; // [esp+420h] [ebp-28h]
  char v11; // [esp+424h] [ebp-24h]
  char v12; // [esp+428h] [ebp-20h]
  char v13; // [esp+42Ch] [ebp-1Ch]
  char v14; // [esp+430h] [ebp-18h]
  char v15; // [esp+434h] [ebp-14h]
  char v16; // [esp+438h] [ebp-10h]
  char v17; // [esp+43Ch] [ebp-Ch]
  char Src; // [esp+440h] [ebp-8h]

  v10 = this;
  sub_4015D0(&v12);
  sub_401540(&v13);
  sub_4016E0(&v17);
  sub_401660(&v14);
  sub_401880(&Src);
  sub_401770(&v16);
  sub_4014C0(&v15);
  sub_4017F0(&v11);
  sub_403AB0(&v9, &Src);                    // unicode 9
  sub_403620((int)&tree9, &v9);
  sub_403AB0(&v8, &v17);                    // M
  sub_403620((int)&M, &v8);
  sub_403AB0(&v7, &v16);                    // k
  sub_403620((int)&k, &v7);
  sub_403AB0(&v6, &v15);                    // c
  sub_403620((int)&c, &v6);
  sub_403AB0(&v5, &v14);                    // 7
  sub_403620((int)&tree7, &v5);
  sub_403AB0(&v4, &v13);                    // t
  sub_403620((int)&t, &v4);
  sub_403AB0(&v3, &v11);                    // kx
  sub_403620((int)&kx, &v3);
  sub_403AB0(&v2, &v12);                    // f
  sub_403620((int)&f, &v2);
  sub_403940(&M, (int)&k);
  sub_403940(&t, (int)&tree9);
  sub_403940(&tree7, (int)&M);
  sub_403940(&t, (int)&f);
  sub_403940(&c, (int)&tree7);
  sub_403940(&root, (int)&kx);
```

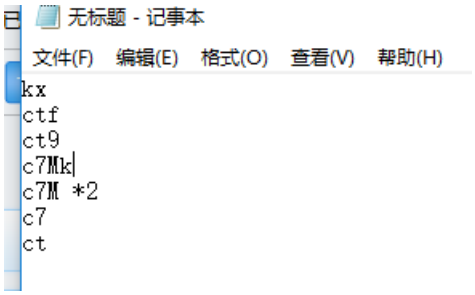
去od对401900下断点，动调跟一下看看各个结点以及他们之间的关系，首先是确定结点，通过观察每一次经过403AB0这个函数的返回值来确定（无法直接观察到结果时数据窗口中跟随）

```
sub_403AB0(&v9, &Src);                    // unicode 9
sub_403620((int)&tree9, &v9);
sub_403AB0(&v8, &v17);                    // M
sub_403620((int)&M, &v8);
sub_403AB0(&v7, &v16);                    // k
sub_403620((int)&k, &v7);
sub_403AB0(&v6, &v15);                    // c
sub_403620((int)&c, &v6);
sub_403AB0(&v5, &v14);                    // 7
sub_403620((int)&tree7, &v5);
sub_403AB0(&v4, &v13);                    // t
sub_403620((int)&t, &v4);
sub_403AB0(&v3, &v11);                    // kx
sub_403620((int)&kx, &v3);
sub_403AB0(&v2, &v12);                    // f
```

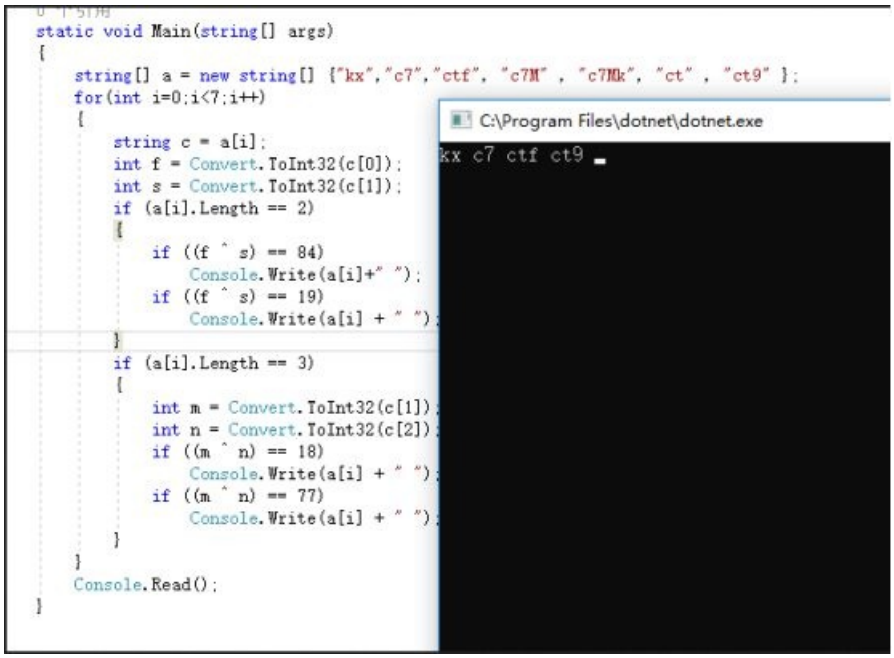
之后就是进行设置字数以及限制多解的操作，403940这个函数就是把第一个参数设置为第二个参数的孩子，并且之后给每个结点加了次数统计，对变量进行重命名之后可以清晰的画出树的结构



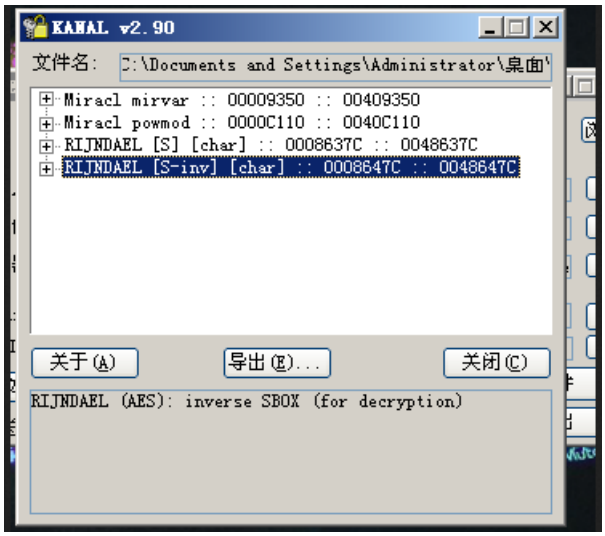
由于字典树是以前缀作为结点的，可以得到单词表



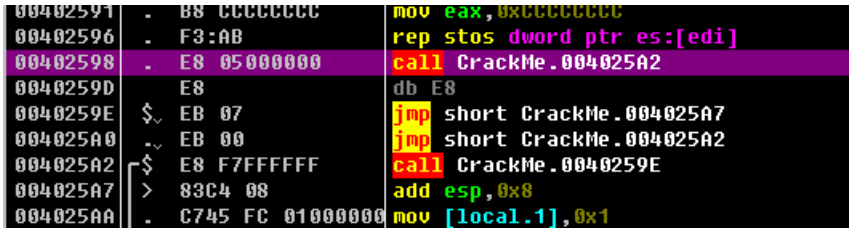
下面就要确定输入的顺序了，与输入的八组单词进行对比，通过单词的长度可以确定部分顺序，至于长度一样的给出了第二个check，写脚本爆破下

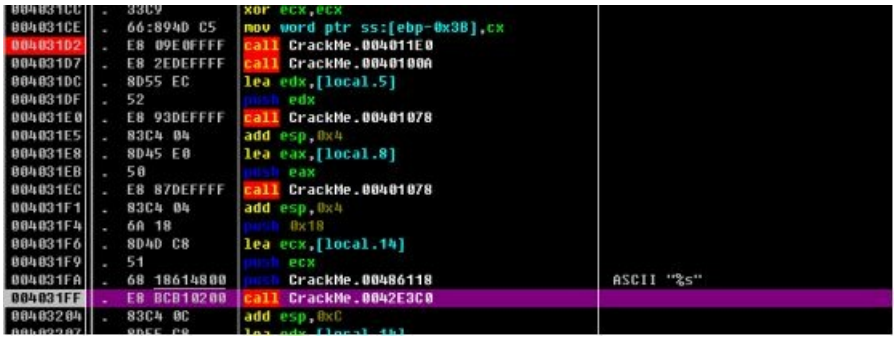


可以确定输入的序列，本题就到此结束了，遗憾就是没有在ida里新建结构体还原下字典树，再看下第四题（咱不会pwn），无壳，放入peid查看下，根据名字猜测里面用了部分加密算法，于是使用peid的密码学插件看下，查出了四个东西，



上面两个不知道是干嘛的，但是提示说是使用了miracl库中的两个方法，去百度下，miracl是大数运算库中，使用的两个方法中一个是横幂，一个是负责分配大数的内存空间的（从var也可以知道一般用来定义变量之类的），还是不知道是啥算法，不过百度下这两个函数全是rsa的，后面两个就是aes中的s盒和逆s盒啦（看到地址记一下），似乎ida识别出的main函数不是真正的main函数，试图从字符串的交叉引用中找到main函数，嗯，找是找到了，但是没法f5生成伪c代码，决定这回不依赖f5来提高自己的汇编阅读能力，找到输出一行*的地方，在od里下个断点，直接走到那里，此时以及绕过了开头的一些乱跳走到了main的内部，这里再扯几句，整个程序到处都是那种positive sp value的错误，是因为部分call加上一个很近的地址然后add esp抬高栈所导致的，例如下图，有针对这种esp值的错误，可以手动patch掉这些干扰分析的代码，也可以选择alt+k自己调整下sp值修复（我几乎都是第二种方法，因为keypatch还没装好，不是很方便）



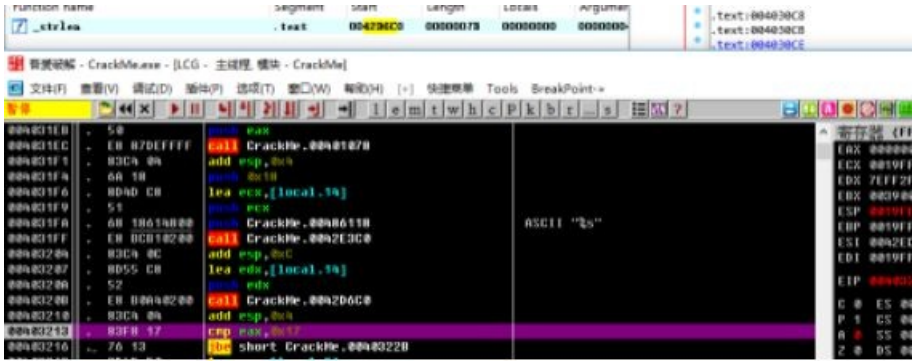


还无意间看到了反调（其实是看到可疑字符串some problem交叉引用找到的），看来调试时得小心点

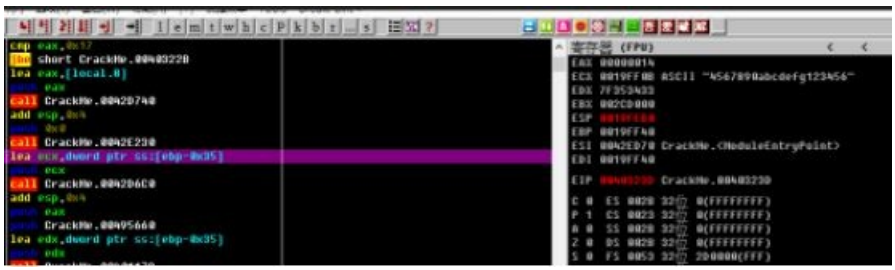
```
int sub_402070()
{
  int result; // eax
  int v1; // [esp+4Ch] [ebp-2Ch]
  int v2; // [esp+50h] [ebp-28h]
  int v3; // [esp+54h] [ebp-24h]
  char v4; // [esp+58h] [ebp-20h]
  char v5; // [esp+5Ch] [ebp-1Ch]
  int v6; // [esp+74h] [ebp-4h]

  sub_401320(&v4);
  v6 = 0;
  dwProcessId = GetCurrentProcessId();
  dword_495650 = sub_40121C(dwProcessId);
  sub_4012DA((int)&v5, dword_495650);
  sub_4012DA((int)&unk_495640, dwProcessId);
  if ( (unsigned __int8)sub_4010FA((int)&v5, "explorer.exe") )
  {
    v3 = 0;
    v6 = -1;
    sub_401249(&v5);
    result = v3;
  }
  else if ( (unsigned __int8)sub_401262(&v5, &unk_495640) )
  {
    if ( !sub_401271(dword_495650) )
      printf("some problem!0.0");
    v2 = 1;
    v6 = -1;
    sub_401249(&v5);
    result = v2;
  }
  else
  {
    v1 = 1;
    v6 = -1;
    sub_401249(&v5);
    result = v1;
  }
  return result;
}
```

继续向下看，首先检查了字符串长度，必须为0x17（判断点在403213）



然后底下的几个call一个个看，首先传入401172把输入的4-23位转换成hex表示，动调可以看出来

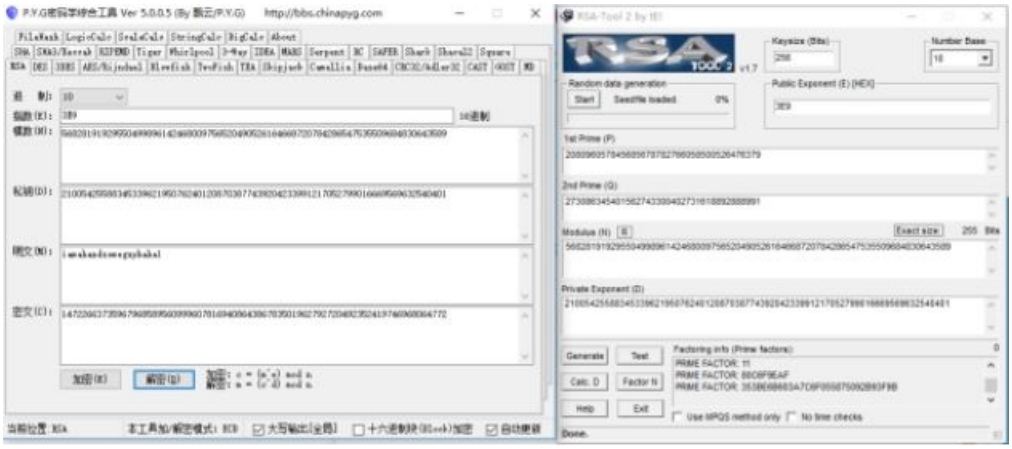


然后传入一个比较复杂的函数分析，之前说到这个程序用到了miracl这个大数运算库，本来想直接导入sig文件看看能不能识别出一些函数,发现ida 7.0并没有自带这个库，需要自己动手做一个，首先去编译一份miracl.lib这个静态库，然后找到flair文件夹先用pcf生成.pat文件，最后再使用sigmake生成了sig文件，嗯，似乎看上去一路顺风，发现。。。识别出的函数屈指可数（估计是自己编译时函数找到不够全），还是选择了用现成的sig文件，发现好用至极，进入那个函数看一下，在进行一些赋值之后，传入了402a3a分析，这个函数用到了许多miracl的函数

mirvar是定义大数变量的，powmod是横幂，具体用法可以在csdn上找到，分析下大概逻辑，已经写在idb里了，想用数学类软件强解，发现出不了结果，在被点拨之后，发现本质上就是rsa的加密函数啊喂喂喂

$$n^e \equiv c \pmod N$$

于是根据rsa的加密函数把几个大数和用mirval定义的几个变量依次对上号，直接用RSATool+pyg密码学工具解决~（真香



在这个函数return 1之后，前面三位还是未知的，继续看下面的call

```
push    offset unk_495000
lea     edx, [ebp-35h]
push    edx
call    sub_401172      ; 输入的4-23转换为十六进制
add     esp, 0Ch
call    sub_40125D      ; 传入一个复杂函数分析
mov     [ebp-4], eax
push    3
lea     eax, [ebp-38h]
push    eax
lea     ecx, [ebp-3Ch]
push    ecx
call    _memcpy
add     esp, 0Ch
lea     edx, [ebp-3Ch]
push    edx
call    sub_40108C      ; 判断是否为数字
add     esp, 4
and     eax, 0FFh
test    eax, eax
jz      short loc_403289
lea     eax, [ebp-3Ch]
push    eax
call    sub_40128F      ; 第二个复杂函数
add     esp, 4
mov     [ebp-8], eax
jmp     short loc_403299
```

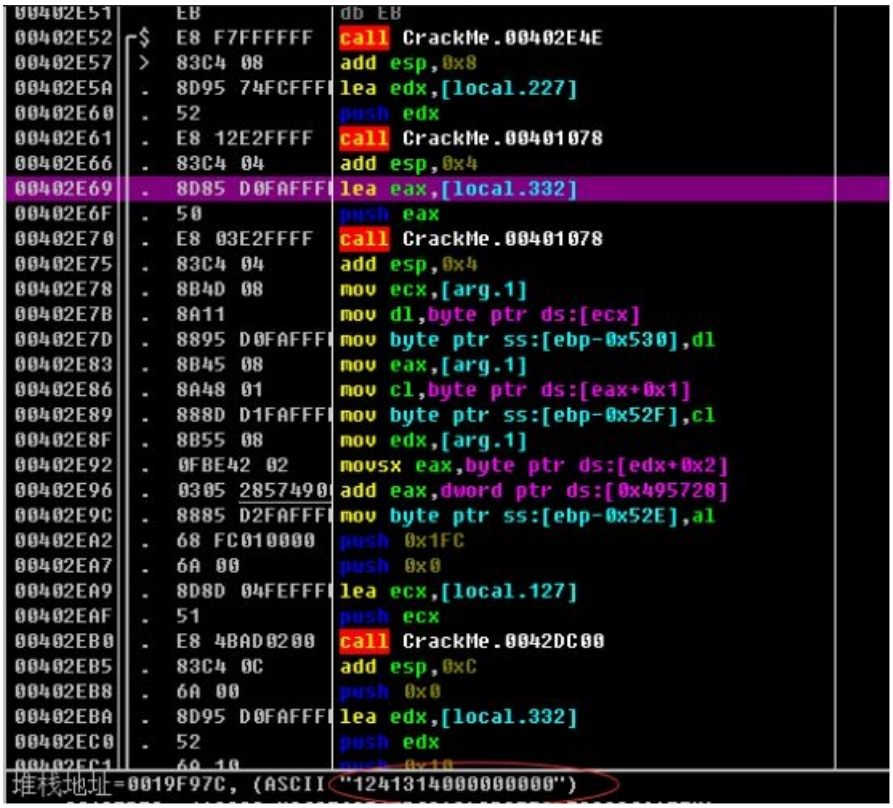
首先判断了前三位是否是数字

```
1  char __cdecl sub_402FC0(int a1)
2  {
3    signed int i; // [esp+4Ch] [ebp-4h]
4
5    for ( i = 0; i < 3; ++i )
6    {
7      if ( !isdigit(*(char *)(i + a1)) )
8        return 0;
9    }
10   return 1;
11 }
```

然后再次传入了复杂函数分析，点进去发现是标准aes加密

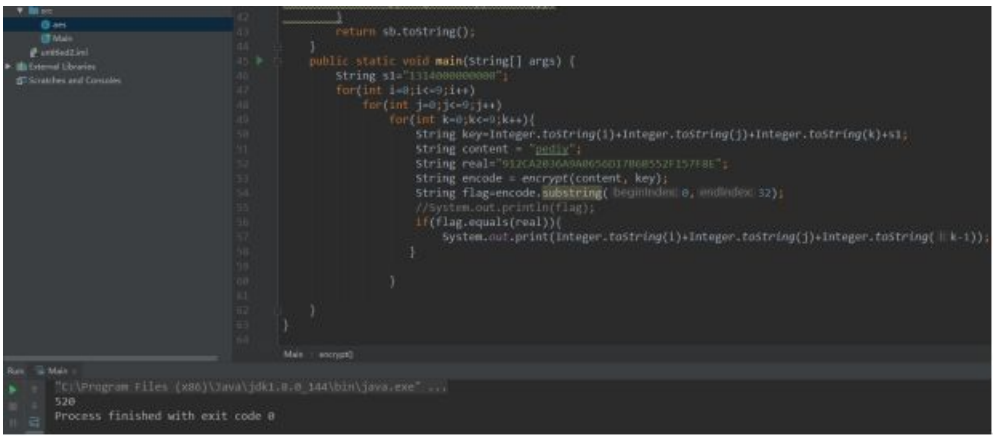
```
sub_402E52      proc near               ; CODE XREF: sub_402D60+E8↑p
                call    near ptr loc_402E4D+1
                add     esp, 8
                lea     edx, [ebp-38Ch]
                push    edx
                call    sub_401078      ; xor_decode
                add     esp, 4
                lea     eax, [ebp-530h]
                push    eax
                call    sub_401078
                add     esp, 4
                mov     ecx, [ebp+8]
                mov     dl, [ecx]
                mov     [ebp-530h], dl
                mov     eax, [ebp+8]
                mov     cl, [eax+1]
                mov     [ebp-52Fh], cl
                mov     edx, [ebp+8]
                movsx   eax, byte ptr [edx+2]
                add     eax, dword_495728
                mov     [ebp-52Eh], al
                push    1FCh            ; size_t
                push    0               ; int
                lea     ecx, [ebp-1FCh]
                push    ecx             ; void *
                call    _memset
                add     esp, 0Ch
                push    0
                lea     edx, [ebp-530h]
                push    edx
                push    10h
                push    0
                lea     eax, [ebp-1FCh]
                push    eax
                call    _aes_init
                add     esp, 14h
                lea     ecx, [ebp-454h]
                push    ecx
                lea     edx, [ebp-1FCh]
                push    edx
                call    _aes_encrypt
                add     esp, 8
                lea     eax, [ebp-454h]
```

光静态分析还是无法理清函数调用和参数的情况的，这里继续动调看



一开始通过两个401078这两个函数（在之前的rsa校验中也出现过），xor解密处需要用的原密钥以及用于strcmp的密文，在动调中看到是0001314000000000000，可以看出这里取出了前三位之后，每一位进行移动，并给第三位加上了1（可以看到我输入的前三位假码是123，生成的密钥是124131400000000000），用来取代原密钥中的前三位，然后作为密钥，对存储在edi寄存器中的明文“pediy”进行加密，最后进行比较

现在就成了如何爆破出前三位的问题，我还是直接选择用aes解密（这里要注意，源程序编译时aes选择的填充方式是zeropadding以及ecb模式的加密，解密时也要对应相应的模式和填充方式，我这里选择的是用jce库里的函数写，发现并没有提供AES/ECB/ZeroPadding这种方式，需要自己用0补充下，既然自己补全了，程序里面就要选择带有nopadding的模式了,但是也没有提供AES/ECB/NoPadding，只好选择默认的填充方式，补充后密文长度变成了之前的两倍，不过只有前一半才是真正有效的值，直接选取前一半就行了



这里附上代码

```
import javax.crypto.SecretKey;

import javax.crypto.Cipher;

    import javax.crypto.spec.SecretKeySpec;

    public class Main {

    public static String encrypt(String content, String key) {//密文和密钥

        try {

            String ALGO_MODE = "AES" ;//为了补零这里本应选择nopadding

            byte[] enCodeFormat = key.getBytes();

            SecretKey secretKey=new SecretKeySpec(enCodeFormat,"AES");//生成密钥对象

            Cipher cipher = Cipher.getInstance(ALGO_MODE);

            //默认ECB和pkcs5padding

            int blockSize = cipher.getBlockSize();

            //获取加密块大小

            byte[] byteContent = content.getBytes();
```

```
        //获得byte类型的密文
        int plaintextLength = byteContent.length;
        if (plaintextLength % blockSize != 0) {
            plaintextLength = plaintextLength + (blockSize - (plaintextLength % blockSize));
        }
        byte[] plaintext = new byte[plaintextLength];
        System.arraycopy(byteContent, 0, plaintext, 0, byteContent.length);
        //上面4行实现了zeropadding
        //System.out.println(plaintext.toString());
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);
        byte[] result = cipher.doFinal(plaintext);
        return Byte2HexStr(result);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
private static String Byte2HexStr(byte[] result) {//byte类型数组转hex方便操作
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < result.length; i++) {
        String hex = Integer.toHexString((int)result[i] & 0xFF);
        if (hex.length() == 1) {
            hex = '0' + hex;
        }
        sb.append(hex.toUpperCase());
    }
    return sb.toString();
}
public static void main(String[] args) {
    String s1="1314000000000";
    for(int i=0;i<=9;i++)
```

```
        for(int j=0;j<=9;j++)
            for(int k=0;k<=9;k++){
                String key=Integer.toString(i)+Integer.toString(j)+Integer.toString(k)+s1;
                String content = "pediy";
                String real="912CA2036A9A0656D17B6B552F157F8E";
                String encode = encrypt(content, key);
                String flag=encode.substring(0,32);
                //System.out.println(flag);
                if(flag.equals(real)){
                    System.out.print(Integer.toString(i)+Integer.toString(j)+Integer.toString(k-1));
                }
            }
    }
}
```

解密得到520

别忘了投稿哦！

合天公众号开启原创投稿啦！！！

大家有好的技术原创文章。

欢迎投稿至邮箱：edu@heetian.com

合天会根据文章的时效、新颖、文笔、实用等多方面评判给予100元-500元不等的稿费哟。

有才能的你快来投稿吧！

点击了解投稿详情 重金悬赏 | 合天原创投稿等你来！


合天智汇