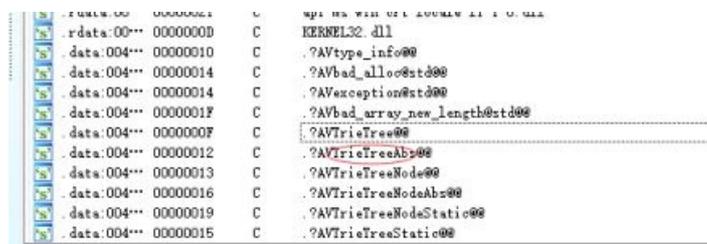


直接拖入ida，看下字符串，发现没有成功或者失败的提示，并且在一些提示信息中可以百度到数据结构是字典树



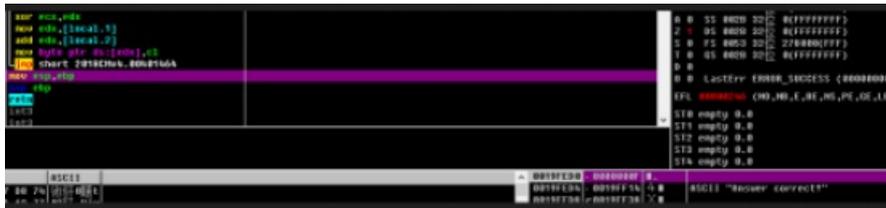
然后去看下main函数，可以看到一些call，但是不知道是做什么的，去动调看下就知道了

```

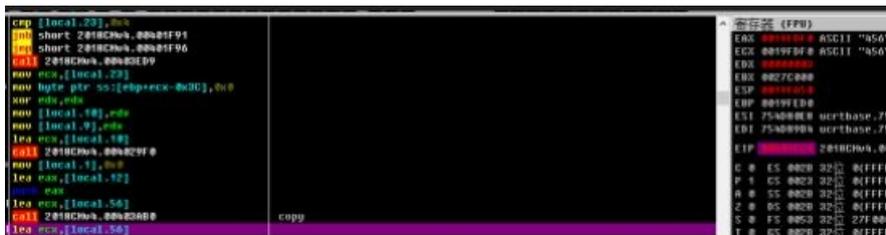
13
14 sub_401170("%s", input, 32);
15 sub_401360(&v5);
16 sub_401200(&v9);
17 if ( input_check(input, (int)&v9) )
18 {
19     if ( &input[strlen(input) + 1] - &input[1] == 22 )
20         check(input, (int)&v5, (int)&v9);
21     else
22         sub_4011D0((int)&v9);
23     v9 = 0;
24     v10 = 0;
25     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组，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]
10    int *v10; // [esp+98h] [ebp-Ch]
11    int *v11; // [esp+9Ch] [ebp-8h]
12    int *v12; // [esp+A0h] [ebp-4h]
13
14    v12 = this;
15    if ( sub_403D70((const char *)this + 4, a2 + 4) )
16        return 0;
17    v7 = v12[66];
18    v6 = *(__DWORD *)(a2 + 264);
19    if ( v7 != v6 )
20        return 0;
21    if ( v12[67] != *(__DWORD *)(a2 + 268) )
22        return 0;
23    v10 = v12 + 34;
24    v11 = v12 + 34;
25    v5 = &v12[v12[66] + 34];
26    while ( v11 != v5 )
27    {
28        v9 = *v11;
29        v3 = (const char *)sub_403670(v9, &v4);
30        v8 = (int *)sub_403830((int *)a2, v3);
31        if ( !v8 )
32            return 0;
33        if ( sub_403610(v8, v9) ) // 递归调用403730进行字符比较
34            return 0;
35        ++v11;
36    }
37    return 1;
38 }

```

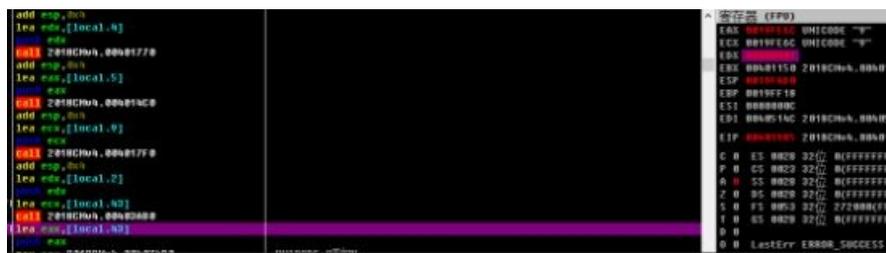
可以发现是一个递归比较的过程，类似于树的遍历，确保v39和目标生成树一摸一样，通过ida的交叉引用看下哪里初始化了407E48这个目标字典树结构

```

1 void *__thiscall sub_401900(void *this)
2 {
3   char v2; // [esp+0h] [ebp-448h]
4   char v3; // [esp+84h] [ebp-3C4h]
5   char v4; // [esp+108h] [ebp-340h]
6   char v5; // [esp+18Ch] [ebp-2BCh]
7   char v6; // [esp+210h] [ebp-238h]
8   char v7; // [esp+294h] [ebp-1B4h]
9   char v8; // [esp+318h] [ebp-130h]
10  char v9; // [esp+39Ch] [ebp-ACh]
11  void *v10; // [esp+420h] [ebp-28h]
12  char v11; // [esp+424h] [ebp-24h]
13  char v12; // [esp+428h] [ebp-20h]
14  char v13; // [esp+42Ch] [ebp-1Ch]
15  char v14; // [esp+430h] [ebp-18h]
16  char v15; // [esp+434h] [ebp-14h]
17  char v16; // [esp+438h] [ebp-10h]
18  char v17; // [esp+43Ch] [ebp-Ch]
19  char Src; // [esp+440h] [ebp-8h]
20
21  v10 = this;
22  sub_4015D0(&v12);
23  sub_401540(&v13);
24  sub_4016E0(&v17);
25  sub_401660(&v14);
26  sub_401880(&Src);
27  sub_401770(&v16);
28  sub_4014C0(&v15);
29  sub_4017F0(&v11);
30  sub_403AB0(&v9, &Src); // unicode 9
31  sub_403620((int)&tree9, &v9);
32  sub_403AB0(&v8, &v17); // M
33  sub_403620((int)&M, &v8);
34  sub_403AB0(&v7, &v16); // k
35  sub_403620((int)&k, &v7);
36  sub_403AB0(&v6, &v15); // c
37  sub_403620((int)&c, &v6);
38  sub_403AB0(&v5, &v14); // 7
39  sub_403620((int)&tree7, &v5);
40  sub_403AB0(&v4, &v13); // t
41  sub_403620((int)&t, &v4);
42  sub_403AB0(&v3, &v11); // kx
43  sub_403620((int)&kx, &v3);
44  sub_403AB0(&v2, &v12); // f
45  sub_403620((int)&f, &v2);
46  sub_403940(&M, (int)&k);
47  sub_403940(&t, (int)&tree9);
48  sub_403940(&tree7, (int)&M);
49  sub_403940(&t, (int)&f);
50  sub_403940(&c, (int)&tree7);
51  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）

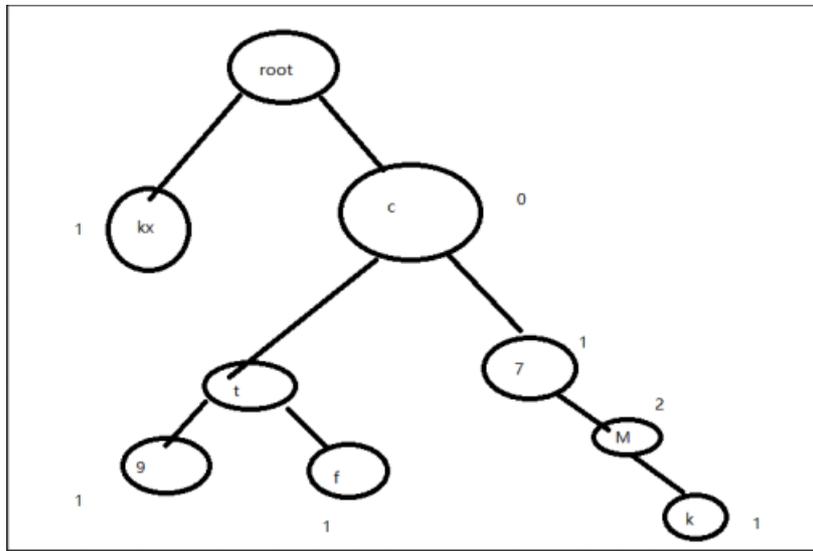


```

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

```

无标题 - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
kx
ctf
ct9
c7Mk
c7M *2
c7
ct
  
```

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

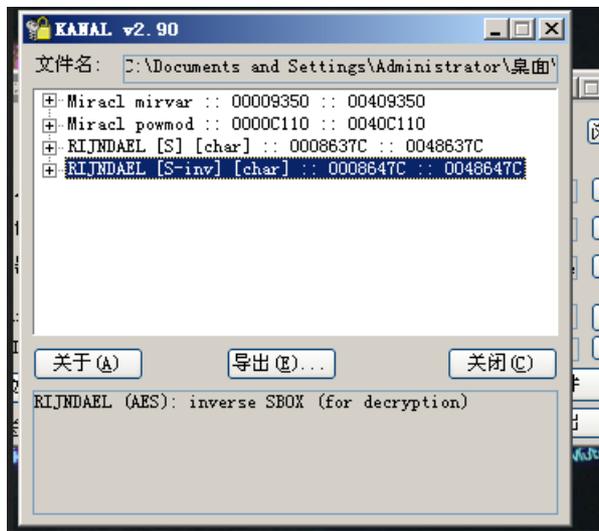
```

0 1 5188
static void Main(string[] args)
{
    string[] a = new string[] { "kx", "c7", "ctf", "c7M", "c7Mk", "ct", "ct9" };
    for (int i = 0; i < 7; i++)
    {
        string c = a[i];
        int f = Convert.ToInt32(c[0]);
        int s = Convert.ToInt32(c[1]);
        if (a[i].Length == 2)
        {
            if ((f ^ s) == 84)
                Console.WriteLine(a[i] + " ");
            if ((f ^ s) == 19)
                Console.WriteLine(a[i] + " ");
        }
        if (a[i].Length == 3)
        {
            int m = Convert.ToInt32(c[1]);
            int n = Convert.ToInt32(c[2]);
            if ((m ^ n) == 18)
                Console.WriteLine(a[i] + " ");
            if ((m ^ n) == 77)
                Console.WriteLine(a[i] + " ");
        }
    }
    Console.ReadLine();
}
  
```

```

C:\Program Files\dotnet\dotnet.exe
kx c7 ctf ct9
  
```

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）

```

00402591 - B8 CCCCCC    mov eax,0xCFFFFFFF
00402596 - F3:AB       rep stos dword ptr es:[edi]
00402598 - E8 05000000 call CrackMe.004025A2
0040259D - E8         db E8
0040259E - EB 07       jmp short CrackMe.004025A7
004025A0 - EB 00       jmp short CrackMe.004025A2
004025A2 - E8 F7FFFFFF call CrackMe.0040259E
004025A7 - 83C4 08     add esp,0x8
004025AA - C745 FC 01000000 mov [local.1],0x1

```

```

00403100 - 33C9       xor ecx,ecx
0040310E - 66:894D C5 mov word ptr ss:[ebp-0x3B],cx
00403102 - E8 09E0FFFF call CrackMe.004011E0
00403107 - E8 2EDEF0FF call CrackMe.0040100A
0040310C - 8D55 EC    lea edx,[local.5]
0040310F - 52        push edx
004031E0 - E8 93DEF0FF call CrackMe.00401078
004031E5 - 83C4 04    add esp,0x4
004031E8 - 8D45 E0    lea eax,[local.8]
004031EB - 50        push eax
004031EC - E8 87DEF0FF call CrackMe.00401078
004031F1 - 83C4 04    add esp,0x4
004031F4 - 6A 18     push 0x18
004031F6 - 8D4D C8    lea ecx,[local.14]
004031F9 - 51        push ecx
004031FA - 68 18614800 push CrackMe.00486118
004031FF - E8 BC810200 call CrackMe.0042E3C0
00403204 - 83C4 0C    add esp,0xC
00403207 - 8D55 C8    lea edx,[local.14]

```

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

```

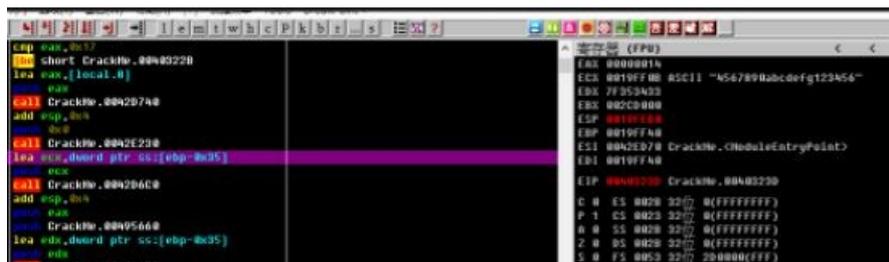
1 int sub_402070()
2 {
3     int result; // eax
4     int v1; // [esp+4Ch] [ebp-2Ch]
5     int v2; // [esp+50h] [ebp-28h]
6     int v3; // [esp+54h] [ebp-24h]
7     char v4; // [esp+58h] [ebp-20h]
8     char v5; // [esp+5Ch] [ebp-1Ch]
9     int v6; // [esp+74h] [ebp-4h]
10
11     sub_401320(&v4);
12     v6 = 0;
13     dwProcessId = GetCurrentProcessId();
14     dword_495650 = sub_40121C(dwProcessId);
15     sub_4012DA((int)&v5, dword_495650);
16     sub_4012DA((int)&unk_495640, dwProcessId);
17     if ( (unsigned __int8)sub_4010FA((int)&v5, "explorer.exe") )
18     {
19         v3 = 0;
20         v6 = -1;
21         sub_401249(&v5);
22         result = v3;
23     }
24     else if ( (unsigned __int8)sub_401262(&v5, &unk_495640) )
25     {
26         if ( !sub_401271(dword_495650) )
27             printf("some problem!0.0");
28         v2 = 1;
29         v6 = -1;
30         sub_401249(&v5);
31         result = v2;
32     }
33     else
34     {
35         v1 = 1;
36         v6 = -1;
37         sub_401249(&v5);
38         result = v1;
39     }
40     return result;
41 }

```

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

```

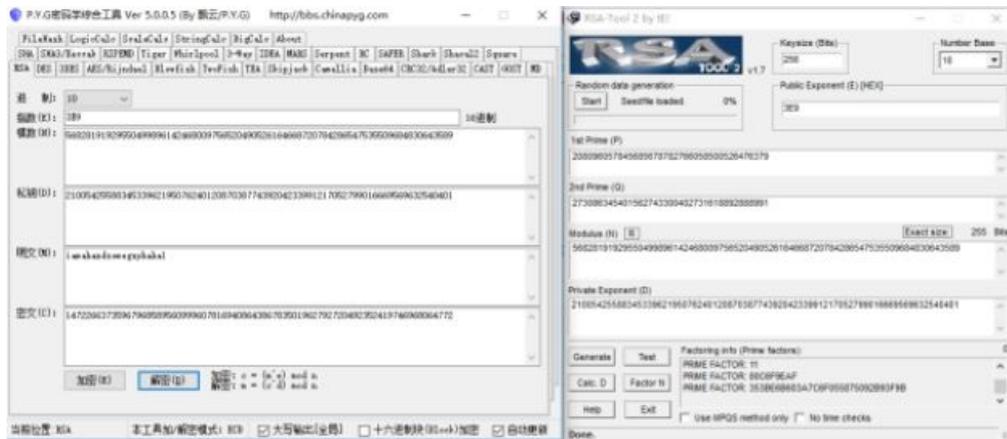
__usercall sub_402A3A@eax(int a1@ebp)
{
    void *v1; // ecx
    int v2; // eax

    {void (*)[void]}(char *)&loc_402A35 + 1();
    sub_401878(v1, v2 - 204);
    sub_401878((void *)(&v1 - 004), v1 - 004); // 生成大数用于比较 ASCII "298C8F7C26FC6A516D97D978F5F0681F534E212505C49A007202084805384"
    *(_DWORD *)(&v1 - 004) = v1 - 4; // 564 = 16; // 7da39de60816477b1af34c8e509dc4291de851f0616d2215579b68b8a185
    *(_DWORD *)(&v1 - 008) = mirvar(0); // 四次大数赋值
    *(_DWORD *)(&v1 - 012) = mirvar(0);
    *(_DWORD *)(&v1 - 020) = mirvar(0);
    *(_DWORD *)(&v1 - 016) = mirvar(0);
    cinstr("(_DWORD *)(&v1 - 020), &v1, 405408); // 下面发现是把输入的a-23位转hex之后转成整数
    cinstr("(_DWORD *)(&v1 - 008), v1 - 004);
    cinstr("(_DWORD *)(&v1 - 012), "3a9");
    if (compare("(_DWORD *)(&v1 - 020), "(_DWORD *)(&v1 - 008)")) != -1 )
        return 0;
    powmod("(_DWORD *)(&v1 - 020), "(_DWORD *)(&v1 - 012), "(_DWORD *)(&v1 - 008), "(_DWORD *)(&v1 - 016)); // a1-016=input[4-23]*3e9 mod 7da39... 转的
    big_to_bytes(0, "(_DWORD *)(&v1 - 016), v1 - 404, 0);
    mirkill("(_DWORD *)(&v1 - 008));
    mirkill("(_DWORD *)(&v1 - 012));
    mirkill("(_DWORD *)(&v1 - 020));
    mirkill("(_DWORD *)(&v1 - 016));
    mirkill("(_DWORD *)(&v1 - 020));
    v2 = strlen(const char *)(&v1 - 404);
    sub_40180F(v1 - 404, v1 - 004, v2); // 以十六进制形式memcpy
    return strcmp(const char *)(&v1 - 204), const char *)(&v1 - 004)) == 0; // 错误的结果心须和208... 转的相等
    // 06486449800970c709485406c70c709110364885238584633581738186080180537299802948105
    // wx"1801 mod 568281912529550498986142468889756528498526164668728784286547535509684838643589
}
    
```

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_493000
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+E81p
            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]

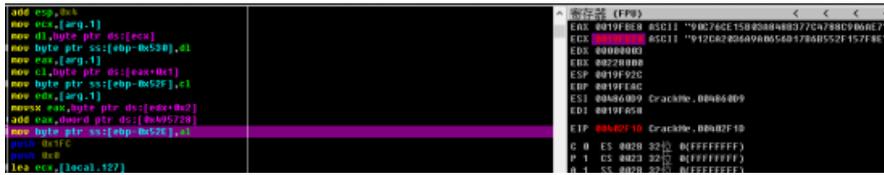
```

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

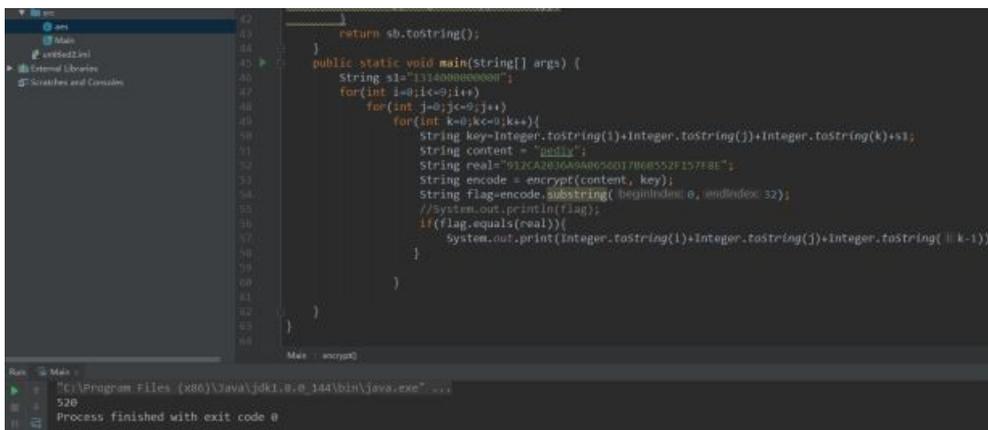
00402E51	EB	db EB
00402E52	\$ E8 F7FFFFFF	call CrackMe.00402E4E
00402E57	> 83C4 08	add esp, 0x8
00402E5A	- 8D95 74FCFFF	lea edx, [local.227]
00402E60	- 52	push edx
00402E61	- E8 12E2FFF	call CrackMe.00401078
00402E66	- 83C4 04	add esp, 0x4
00402E69	- 8D85 D0FAFFF	lea eax, [local.332]
00402E6F	- 50	push eax
00402E70	- E8 03E2FFF	call CrackMe.00401078
00402E75	- 83C4 04	add esp, 0x4
00402E78	- 8B40 08	mov ecx, [arg.1]
00402E7B	- 8A11	mov dl, byte ptr ds:[ecx]
00402E7D	- 8895 D0FAFFF	mov byte ptr ss:[ebp-0x530], dl
00402E83	- 8B45 08	mov eax, [arg.1]
00402E86	- 8A48 01	mov cl, byte ptr ds:[eax+0x1]
00402E89	- 888D D1FAFFF	mov byte ptr ss:[ebp-0x52F], cl
00402E8F	- 8B55 08	mov edx, [arg.1]
00402E92	- 0FB E42 02	movsx eax, byte ptr ds:[edx+0x2]
00402E96	- 0305 2857490	add eax, dword ptr ds:[0x495728]
00402E9C	- 8885 D2FAFFF	mov byte ptr ss:[ebp-0x52E], al
00402EA2	- 68 FC010000	push 0x1FC
00402EA7	- 6A 00	push 0x0
00402EA9	- 8D8D 04FEFFF	lea ecx, [local.127]
00402EAF	- 51	push ecx
00402EB0	- E8 4BAD0200	call CrackMe.0042DC00
00402EB5	- 83C4 0C	add esp, 0xC
00402EB8	- 6A 00	push 0x0
00402EBA	- 8D95 D0FAFFF	lea edx, [local.332]
00402EC0	- 52	push edx
00402EC1	- 60 10	push 0x10

堆栈地址=0019F97C, (ASCII "1241314000000000")

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000131400000000000，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！](#)



