171204 逆向-JarvisOJ（软件密码破解-3）（3）

原创

奈沙夜影 于 2017-12-04 20:33:05 发布 427 收藏

分类专栏： CTF

本文链接：https://blog.csdn.net/whklhhhh/article/details/78713513

1625-5 王子昂 总结《2017年12月4日》【连续第430天总结】

A. JarvisOJ-Re-软件密码破解-3（3）

B.

立完flag今天就搞定了233

主要依靠http://www.read138.com/archives/041/o2hwtxkv725p2hwk/的讲解

昨天SEH中没找到是因为这里的异常处理是用的VEH，向量化异常处理

VEH有两个新增异常处理结构的API，分别是AddVectoredExceptionHandler和AddVectoredContinueHandler
前者的检查在SEH之前，后者则在SEH之后

而本题中使用的正好就是AddVectoredExceptionHandler

函数过程位于sub_4024F0

并且作者还非常鸡贼的使用了动态获取API，动态解密字符串的方法，从而使得很难静态分析到这个函数调用……另外使用的ntdll中的核心API，而不是Kernel中的封装API-AddVectoredExceptionHandler

```
33  *(_DWORD *)ProcName = 0x9BA9A98C;
34  v11 = 0xA2B9AEA8;
35  v12 = 0x88A9A8BF;
36  v13 = 0xBDA8AEB5;
37  v14 = 0xA4B9u;
38  v15 = 0xA2u;
39  v16 = 0xA3AC85A3;
40  v17 = 0xBFA8A1A9;
41  v18 = 0xCDu;
42  *(_DWORD *)ModuleName = 0xA5C0A5CE;
43  v20 = 0xA5CBA5D7;
44  v21 = 0xA5C9A5C0;
45  v22 = 0xA597A596;
46  v23 = 0xA5A5u;
47  v4 = 0;
48  do
49  {
50    ProcName[v4] ^= 0xCDu;
51    ++v4;
52  }
53  while ( v4 < 28 );
54  v5 = 0;
55  do
56    *((_BYTE *)ModuleName + v5++) ^= 0xA5u;     // 异或解码字符串
57  while ( v5 < 18 );
58  v6 = GetModuleHandleW(ModuleName);
59  v7 = GetProcAddress(v6, ProcName);            // 获取API（ntdll.RtlAddVectoredExceptionHandler）的地址
60  v8 = (void (__stdcall *)(signed int, int (__stdcall *)(int)))v7;
61  if ( v7 )
62  {
63    ((void (__stdcall *)(signed int, int (*)()))v7)(1, sub_401CA0);// 添加两个VEH
64    v8(1, sub_401B80);
65  }
66  return v2;
67}
```


太坑了……除了运行中检查VEH的内存结构以外，就只能对ntdll的API进行下断了
（搜了一圈没发现合适的插件来获取VEH链，以后有机会自己写个）

查找两个异常处理函数，前者直接退出
后者则进行了一个处理和校验

```
1 int __stdcall sub_401B80(int a1)
2 {
3   sub_401970();
4   if ( (unsigned __int8)(byte_571458[3] + byte_571458[2] + byte_571458[1] + byte_571458[0]) == 71
5     && (unsigned __int8)(byte_571458[7] + byte_571458[6] + byte_571458[5]) == 3
6     && (unsigned __int8)byte_571458[0] == (unsigned __int8)byte_571458[1] + 68
7     && (unsigned __int8)byte_571458[1] == (unsigned __int8)byte_571458[2] + 2
8     && (unsigned __int8)byte_571458[2] == (unsigned __int8)byte_571458[3] - 59
9     && (unsigned __int8)byte_571458[6] == (unsigned __int8)byte_571458[4] + 10
10    && (unsigned __int8)byte_571458[6] == (unsigned __int8)byte_571458[7] + 9
11    && (unsigned __int8)byte_571458[4] == (unsigned __int8)byte_571458[5] + 52 )
12  {
13    JUMPOUT(__CS__, 0x1947 + 0x400000);
14  }
15  return 0;
16}
```


流程应该是先进行sub_401B80，如果跳转到0x401947（成功分支）则执行对应嗲马，如果校验未通过则Return 0，系统将执行流转交给下一个异常处理函数sub_401CA0直接退出程序

于是先看校验结果，是个方程组，直接Z3求解就行

而401970中的内容是一个多次查表变换的函数

```
26  v0 = 64;
27  do
28  {
29    v1 = sbox[(unsigned __int8)value[3]];
30    v2 = sbox[(unsigned __int8)sbox[(unsigned __int8)sbox[(unsigned __int8)value[1]]]];
31    v3 = sbox[(unsigned __int8)sbox[(unsigned __int8)sbox[(unsigned __int8)sbox[(unsigned __int8)value[0]]]]];
32    value[2] = sbox[(unsigned __int8)sbox[(unsigned __int8)value[2]]];
33    value[1] = v2;
34    v4 = sbox[(unsigned __int8)value[2]];
35    value[0] = v3;
36    v5 = sbox[(unsigned __int8)value[1]];
37    value[3] = v1;
38    v6 = sbox[(unsigned __int8)v1];
39    v7 = sbox[(unsigned __int8)value[4]];
40    value[3] = v6;
41    value[2] = v4;
42    v8 = sbox[(unsigned __int8)v6];
43    value[1] = v5;
44    v9 = sbox[(unsigned __int8)value[2]];
45    value[4] = v7;
46    v10 = sbox[(unsigned __int8)v7];
47    v11 = sbox[(unsigned __int8)value[5]];
48    value[4] = v10;
49    value[3] = v8;
50    v12 = sbox[(unsigned __int8)v10];
51    value[2] = v9;
52    v13 = sbox[(unsigned __int8)value[3]];
53    value[5] = v11;
54    v14 = sbox[(unsigned __int8)v11];
55    v15 = sbox[(unsigned __int8)value[6]];
56    v16 = (unsigned __int8)sbox[(unsigned __int8)sbox[(unsigned __int8)value[7]]];
57    value[5] = v14;
```


仔细看看就可以发现一共循环了64次，每次循环中进行了4次循环查表，所以实际上就可以直接当做是循环了64*4次查表233这个小坑意义不大

至于表的来源，通过查看表来搜索也行，密码学分析工具也行，可以发现它就是AES算法的S盒
那么求解只需要通过逆S盒（wiki上有）来查表就行了

求解算法过程：
首先通过方程求出解，然后将解查表64*4次即可，输出HEX

Python3：

```
from z3 import *

re_s_table = [0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3, 0xD7,
    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]


def decrypt(c):
    result = b''
```

```
        for i in c:
            a = i // 0x10
            b = i % 0x10
            result += re_s_table[a * 0x10 + b].to_bytes(1, "little")
        return result

    r = [Int("r%d"%i) for i in range(8)]
    s = Solver()
    s.add(((r[3] + r[2] + r[1] + r[0] )%256 == 71, (r[7] + r[6] + r[5] )%256 == 3, (r[0] )%256 == r[1] + 68

    for i in r:
        s.add(i>=0)
        s.add(i<256)

    while(s.check()==sat):
        c = b''
        m = s.model()

        for i in range(8):
            c +=(((m[r[i]].as_long()).to_bytes(1, 'little')))

        # 输出方程的解
        for i in c:
            print(hex(i)[2:].zfill(2).upper(), end='')
        print('\t', end='')

        # 查表
        for i in range(64*4):
            p = decrypt(c)
            c = p

        # 输出flag
        for i in c:
            print(hex(i)[2:].zfill(2).upper(), end='')

        #  排除该解后继续求解
        s.add(r[0] != m[r[0]].as_long())

        print()
```

PS：WriteUp上都只提到了一个解，方程解和flag分别为`7733316C64306E65 C7C536FC625CEFCD`
实际上可通过的flag也确实是这个
但是通过Z3一共可以算出3个方程的解，对应的flag也分别有3个，测试发现它们也都能通过程序的校验，弹出正确的提示

> B77371AC64306E65 0F73974F625CEFCD
> 7733316C64306E65 C7C536FC625CEFCD
> F7B3B1EC64306E65 2DA0F111625CEFCD

其他地方也没有对flag的限定，仅说是16位MD5，但是MD5解不出也不一定代表就不是MD5值对吧……毕竟彩虹表并不能解出所有的MD5ヽ(ﾍ _ ヽ)ノ

这题又学到了不少东西~JarvisOJ大赞(°∀°)ﾉ
（就是WriteUp难找有点麻烦233）

C. 明日计划
JarvisOJ-RE