VMP学习笔记之Handle块优化与壳模板初始化（四）

参考资料：

本文大量内容抄袭看雪作者：waiWH的VMP系列

1、名称：**谈谈vmp的还原(2)**

网址：https://bbs.pediy.com/thread-225278.htm

2、名称：**汇编指令之OpCode快速入门**

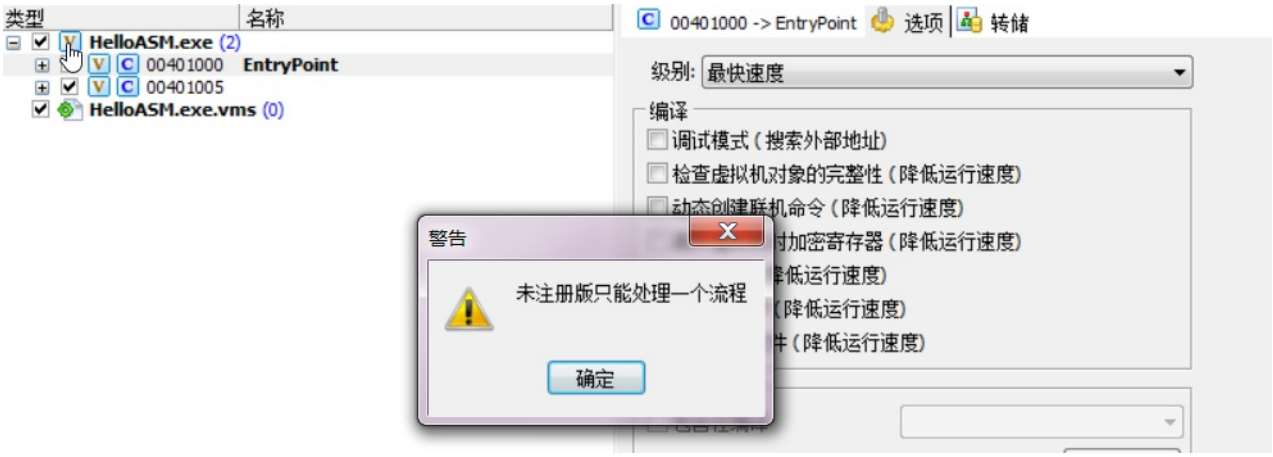
网址：https://bbs.pediy.com/thread-113402.htm

3、名称：**X86指令编码内幕 --- 指令 Opcode 码**

网址：https://blog.csdn.net/xfcyhuang/article/details/6230542

说明：

1、流程 == 加密函数个数（一般demo版本只允许加密一个流程）



2、用户代码跟壳自身代码（模板）都会用Vmp_AllDisassembly函数解析

```
00000000 ; 所有保存的数据：
00000000 ; 1、保存PE信息New节区Voffset跟开启保标志
00000000 ; 2、所有解析后的OPcode
00000000 ; 3、PE基本信息
00000000 struct_VmpAllData struc ; (sizeof=0x74, mappedto_281)
00000000 This            dd ?
00000004 _prev_node      dd ?                    ; 这是最顶层的结构体，所以它没有上一层
00000008 struc_UserVmpOpcode dd ?                ; 保存用户要加密程序的信息
0000000C field_C         dd ?
00000010 struct_PEInformation dd ?               ; 保存PE基本信息的结构体,被加壳的程序信息
00000014 struct_VmpPEInformation dd ?            ; 保存PE基本信息的结构体,加壳机的程序信息(这个结构体分析有问题的，不要参考我的)
00000018 NewSectionVOffset dd ?                  ; 新区段的Voffset（加载到内存后的地址）
0000001C field_1C        dd ?                    ; 兼容64位
00000020 VmpStubStart    dd ?                    ; VMP壳的起始地址
00000024 field_24        dd ?                    ; 兼容64位
00000028 NumberInc       dd ?                    ; 进度条来的，无视
0000002C field_2C        dd ?
00000030 field_30        db ?
00000031 field_31        db ?
00000032 field_32        dw ?                    ; 不知道干嘛的，有判断sub_4A3304
00000034 field_34        dd ?
00000038 Flag            dd ?                    ; 标志位默认就是-1？？？？？
0000003C field_3C        dd ?
00000040 field_40        dd ?
00000044 field_44        dd ?
00000048 ProtectOptions  db ?                    ; 保存开启保护功能标志位
00000049 field_49        db ?
0000004A field_4A        db ?
0000004B field_4B        db ?
0000004C field_4C        dd ?
00000050 struct_VmpOpcode dd ?                   ; 指向struct_VmpOpcode结构体  主要保存壳自身的信息
00000054 field_54        dd ?
00000058 field_58        dd ?
```

2、1 保存用户代码结构体

它们0x8跟0x10都是指向相同的结构体

```
00000000
00000000 ; 1、保存解析后的Opcode信息
00000000 ; 2、保存加密地址与加密结束地址
00000000 struc_UserVmpPEInformation struc ; (sizeof=0x90, mappedto_326)
00000000 This            dd ?
00000004 _prev_node      dd ?                    ; 指向struct_VmpAllData结构体
00000008 struc_UserVmpDisassemblerOpcode dd ?    ; 保存用户要VmpOpcode解析信息了
0000000C field_C         dd ?                    ; 标志位???????
00000010 struc_UserVmpSpecialDisassemblerOpcode dd ? ; 专门保存用户Vmcode解析信息里包含：call jmp的指令
00000014 field_14        dd ?
00000018 UserVmpStartAddr1 dd ?                  ; 用户Vmp加密起始地址
```

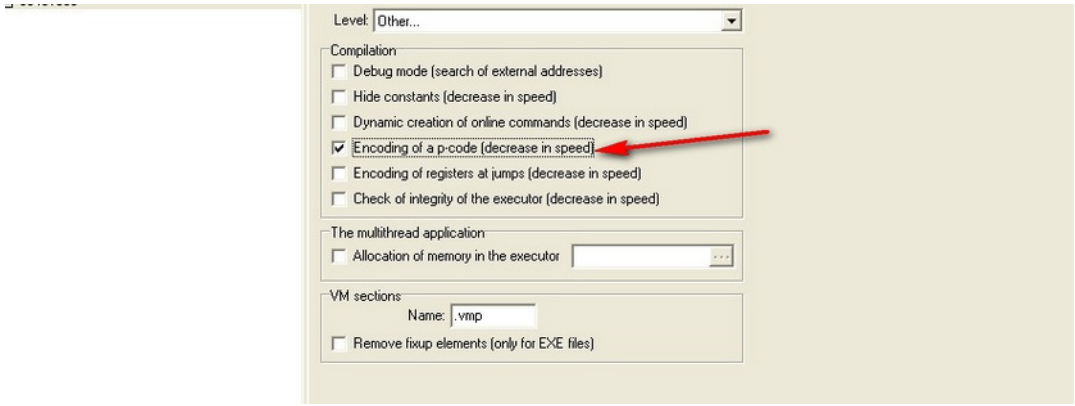
2、2 保存壳自身模板结构体

它们0x8跟0x10都是指向相同的结构体

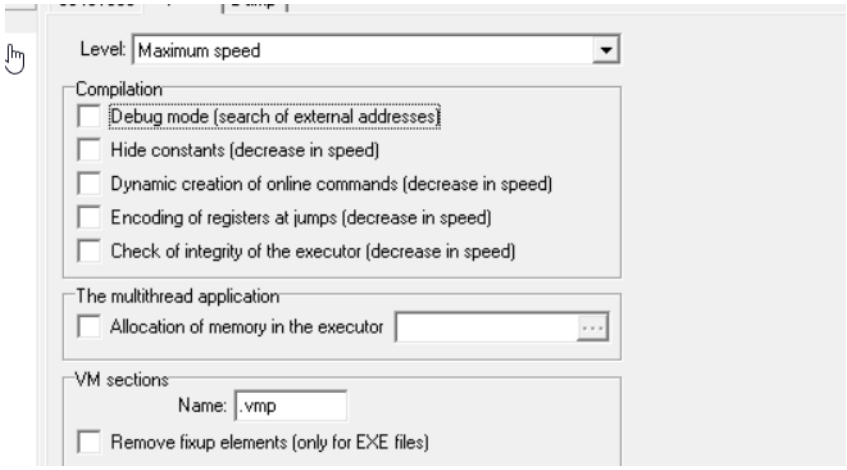
```
0000000 ; ---------------------------------------------------------------------------
0000000
0000000 ; 保存内容：
0000000 ; 1、解析后Opcode信息
0000000 ; 2、作者设计VmpHandle开始和结束地址
0000000 ; 3、壳的入口
0000000 struct_VmpOpcode struc ; (sizeof=0x3DC, align=0x4, mappedto_291)
0000000 This            dd ?
0000004 _prev_node      dd ?                    ; 指向一个链表结构体：指向struct_VmpAllData
0000008 struc_SaveAllDisasmFunData dd ?         ; 保存所有解析后的Opcode函数信息
000000C Magic           dd ?                    ; 数组排列乱序：使用就是1，未使用就是0。
0000010 struc_SavePartDisasmFunData1 dd ?       ; 保存特殊解析后的Opcode信息,专门JmpAddrHandle函数
0000014 gap14           dd ?                    ; 兼容64位
0000018 VmpStubStart    dd ?                    ; Vmp壳入口
```

3、我比较了下1.10跟1.21，发现1.21自带Encoding of a p-code保护

1.10

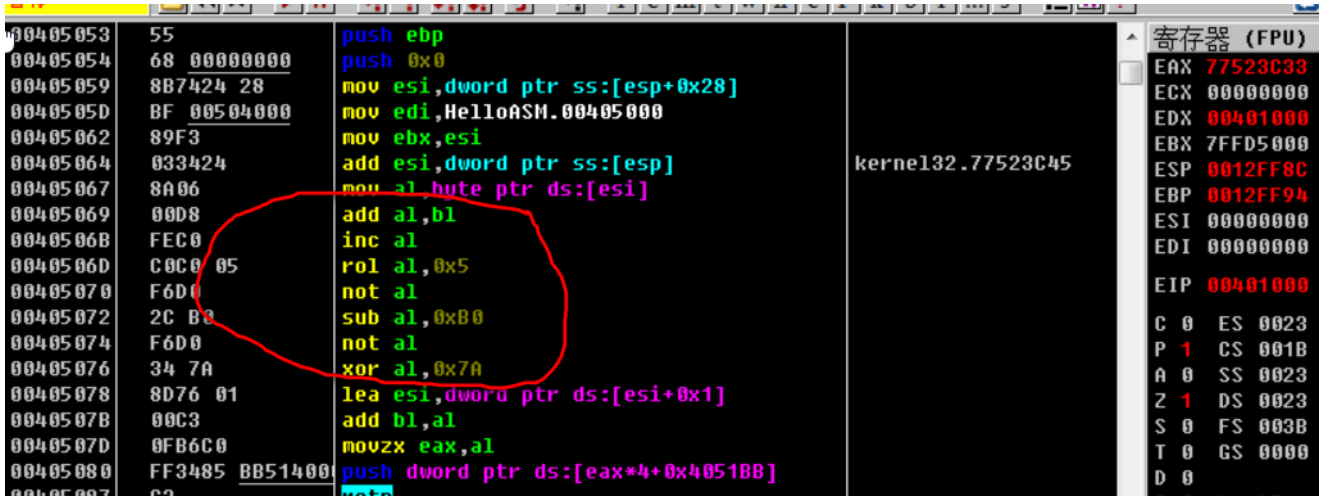


1.21



未加密前的一般长这样子（1.10未启用加密的）：

加密后（1.21自带加密）：



主要干了什么？

1、初始化壳模板：指令变形、等价替换

例如：

jmp = push + retn 或则 lea + jmp

lods byte ptr ds:[esi] = mov al,[esi] + inc esi 或则 mov al,[esi] + add esi,1

等等

2、优化Handle块代码，将不使用的直接删除

ESIResults[X] == 0表示不使用，这种就会优化

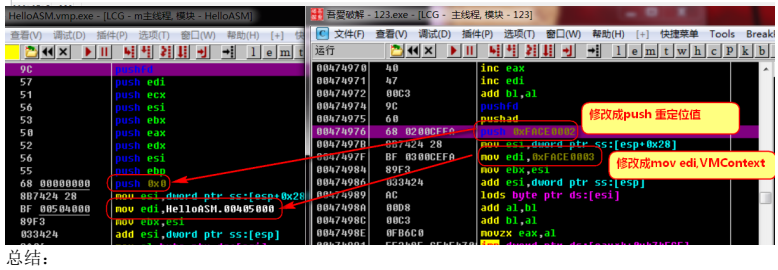
ESIResults[X] == 1表示使用

3、找出填充虚拟机上下文的两个Handle块

正文：

1、找出壳模板push 0xFACE0002与mov edi,0xFACE0003

```
v10 = 0;
do
{
  v11 = (struct_DisassemblyFunction *)GetItem_7((int)v7, v10, v9);// 循环获取DisassemblyFunction函数
  v12 = v11;
  if ( v11->VMOpcode != 0xC || **(_BYTE **)(*(char **)&v11->ReadHexAddress + 3) != 0xE9u )// 1、判断替换后的opcode!=0XC  E9 E8即是jmp 近 短之类的
                                          // 2、读取到的Opcode为召唤= E8(CALL    跳转   短)
  {
    i = 0;
    while ( 1 )                           // 遍历Vmp_Disassembly函数填充内容（第N组），一共有N组
    {
      v224 = &v12->First + i;
      if ( *((_BYTE *)&v12->First.ModRM_mod__Or__Size + 0x17 * i) & 2 )
      {
        v227 = v12->VMOpcode == 0x8 || v224->Mod_Rm ? (unsigned int)sub_A09CC0((int)v12, 2, v224->ReadHexLen) : v224->Displacement;// VmpOpcode==0x8较律，特定call或则jmp
        if ( (v227 & 0xFFFFFF00) == 0xFACE0000 )// 作者设计的handle里面有两个符合条件的分别是，
                                          // push xxxxx       优化后 push  重定位值
                                          // mov edi,xxxxx    优化后 mov edi,VMContext
          break;
      }
      if ( ++i == 3 )
        goto LABEL_15;
    }
    v223 = sub_A9FEC0(*(_DWORD *)&v7[9].gapA[4]);
    *(_BYTE *)(v223 + 0) = v227 & 7;
    *(_DWORD *)(v223 + 0xC) = GetItem_7((int)v7, v10, v13);
    *(_DWORD *)(v223 + 0x10) = i;
  }
  else
  {
    **(_BYTE **)(*(char **)&v11->ReadHexAddress + 3) = 0xE9u;
    __linkproc__ SysArraySetLength(*(_DWORD *)&v11->ReadHexLen + 3);
    *(_DWORD *)&v12->ReadHexLen += 3;
  }
LABEL_15:
  ++v10;
  --v225;
}
while ( v225 );                           // 循环遍历找到符合条件的Opcode，并修改它的值。
```



总结：

1、循环遍历作者设计的Handle块找到符合条件的例如：

规律if(v227 & 0xFFFFFF00) == 0xFACE0000

执行前	转换	执行后
push 0xFACE0002	------->	Push 重定位值
mov edi,0xFACE0003	------->	Mov edi,VMContext

2、所使用的结构体如下：

```
00000000 ; 结构体：1、修复重定位地址(push XXXX) 2、edi（VMcontext）指向地址修复
00000000 struct_Edi_Addr struc ; (sizeof=0x14, align=0x4, mappedto_286)
00000000 This            dd ?
00000004 _prev_node      dd ?                    ; 指向struct_VmpOpcodePY_3D4结构
00000008 Constant_Byte   db ?                    ; 保存作者设计Handle里面的有特殊含义常量(0XFACEXXXX)最低字节，例如push 0xFACE0002的02
00000009                 db ? ; undefined
0000000A                 db ? ; undefined
0000000B                 db ? ; undefined
0000000C struct_DisassemblyFunction dd ?         ; 保存找到的结构内容（struct_DisassemblyFunction）
00000010 Number          dd ?                    ; struct_DisassemblyFunction里面有三组相同的结构体：分别是First、Second、Third，判断到底是第几个
00000014 struct_Edi_Addr ends
00000014
```

2、根据pNtHeader_OptionalHeader.Magic筛选ESI_Matching_Array数组

```
Type = *(_BYTE *)(a2a + 9);
Numberume = 0xCC;
v15 = (const signed __int32 *)ESI_Matching_Array;// ESI_Matching_Array每一组是8个字节，一共有0Xcc组，也就是总长度是0x660=8*0xCC
v181 = (int *)ESIResults;
do
{
  v219 = v15;
  v16 = Type < 7u;
  if ( Type <= 7u )
    v16 = _bittest(v15, Type & 0x7F);         // 判断*(byte*)(ESI_Matching_Array+0) bt Type，如果成立v17=0，跳过下面*(byte*)(ESI_Matching_Array+1)的判断
  v17 = v16 && *((_BYTE *)v219 + 1);          // 判断*(byte*)(ESI_Matching_Array+1) == 0，如果成立v17=0，否则=1
  *(_BYTE *)v181 = v17;                       // 保存结果，后面会使用
  v181 = (int *)((char *)v181 + 1);           // 偏移到下一个字节
  v15 += 2;                                   // 8个一组
  --Numberume;
}
while ( Numberume );
```

首先我们得到的信息有：

1、ESI_Matching_Array每一组是8个字节，一共有0Xcc组，也就是总长度是0x660=8*0xCC

目前已知：

ESI_Matching_Array[0] == 与Magic有关

ESI_Matching_Array[1] == ？？

ESI_Matching_Array[2] == VMOpcode
ESI_Matching_Array[3] == VMOpcode
ESI_Matching_Array[4] == 模式
ESI_Matching_Array[5] == Size
ESI_Matching_Array[6] == ？？
ESI_Matching_Array[7] == 未使用，都是0

2、v184数组保存所有结果0或则1

3、第一个判断：v16 = _bittest(v15, Type & 0x7F)成立条件

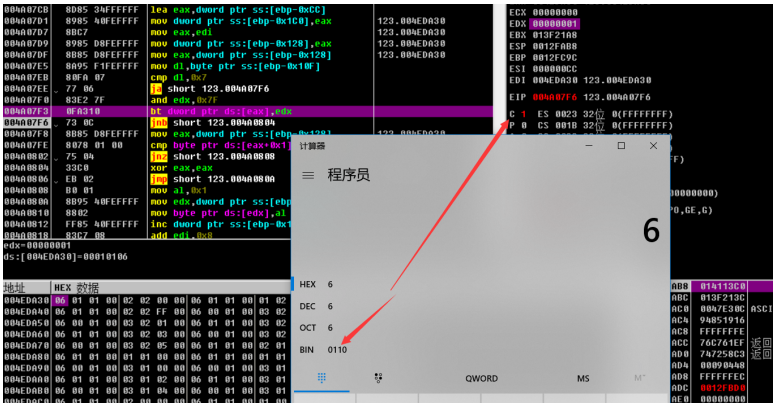
我们查询汇编指令得知BT是位测试

目前只讨论win32PE结构，Type都是=1

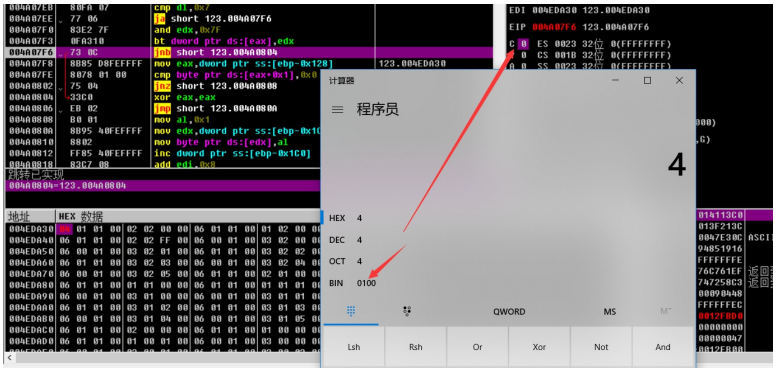
我们发现*(byte*)(ESI_Matching_Array+0)的值只有6(0110)或则4(0100)

```
04E0h: 06 00 61 01 00 02 00 00 06 00 6A 01 00 02 00 00  ..a.......j.....
04F0h: 06 00 6C 01 00 02 00 00 06 00 6E 01 00 02 00 00  ..l.......n.....
0500h: 06 00 60 01 00 02 00 00 06 00 62 01 00 02 00 00  ..`.......b.....
0510h: 06 00 5F 01 00 02 00 00 06 00 59 01 00 02 00 00  .._.......Y.....
0520h: 06 00 5C 01 00 02 00 00 06 00 13 00 00 01 00 00  ..\.............
0530h: 06 00 13 00 00 02 00 00 06 00 14 00 00 01 00 00  ................
0540h: 06 00 14 00 00 02 00 00 06 00 14 00 00 03 00 00  ................
0550h: 06 00 1B 00 00 01 00 00 06 00 1B 00 00 02 00 00  ................
0560h: 04 01 01 00 02 03 00 00 04 01 01 00 01 03 00 00  ................
0570h: 04 01 01 00 01 00 02 00 04 01 01 00 01 01 02 00  ................
0580h: 04 01 01 00 01 02 02 00 04 01 01 00 02 03 FF 00  ..............ÿ.
0590h: 04 00 01 00 03 03 00 00 04 00 01 00 03 03 01 00  ................
05A0h: 04 01 01 00 03 03 02 00 04 01 01 00 03 03 03 00  ................
05B0h: 04 00 01 00 03 03 04 00 04 00 01 00 03 03 05 00  ................
05C0h: 04 01 02 00 02 03 00 00 04 01 02 00 02 03 FF 00  ..............ÿ.
05D0h: 04 00 02 00 03 03 00 00 04 00 02 00 03 03 01 00  ................
05E0h: 04 01 02 00 03 03 02 00 04 01 02 00 03 03 03 00  ................
05F0h: 04 00 02 00 03 03 04 00 04 00 02 00 03 03 05 00  ................
0600h: 04 01 04 00 00 03 01 00 04 01 0A 00 00 03 01 00  ................
0610h: 04 01 3E 00 00 03 01 00 04 01 3D 00 00 03 01 00  ..>.......=.....
0620h: 04 01 46 00 00 03 01 00 04 01 45 00 00 03 01 00  ..F.......E.....
0630h: 04 00 5E 00 00 03 01 00 04 00 60 00 00 03 01 00  ..^.......`.....
0640h: 04 00 61 00 00 03 01 00 04 00 5F 00 00 03 01 00  ..a......._.....
0650h: 04 00 01 00 05 02 08 00 04 00 02 00 05 02 08 00  ................
0660h:
```

6=0110 bt 1 第一位的值给CF，就是1，不成立继续执行第二个判断



4=0100 bt 1 第一位的值给CF，就是0，成立v17=0，不继续执行第二个判断



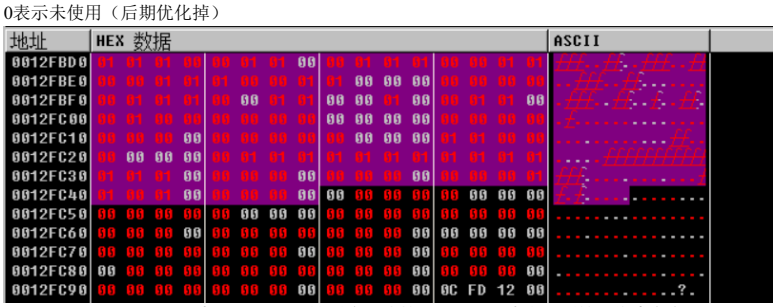
4、总结：

1、v184保存结果的什么时候使用、区别这个有什么意义？1跟0有什么区别？

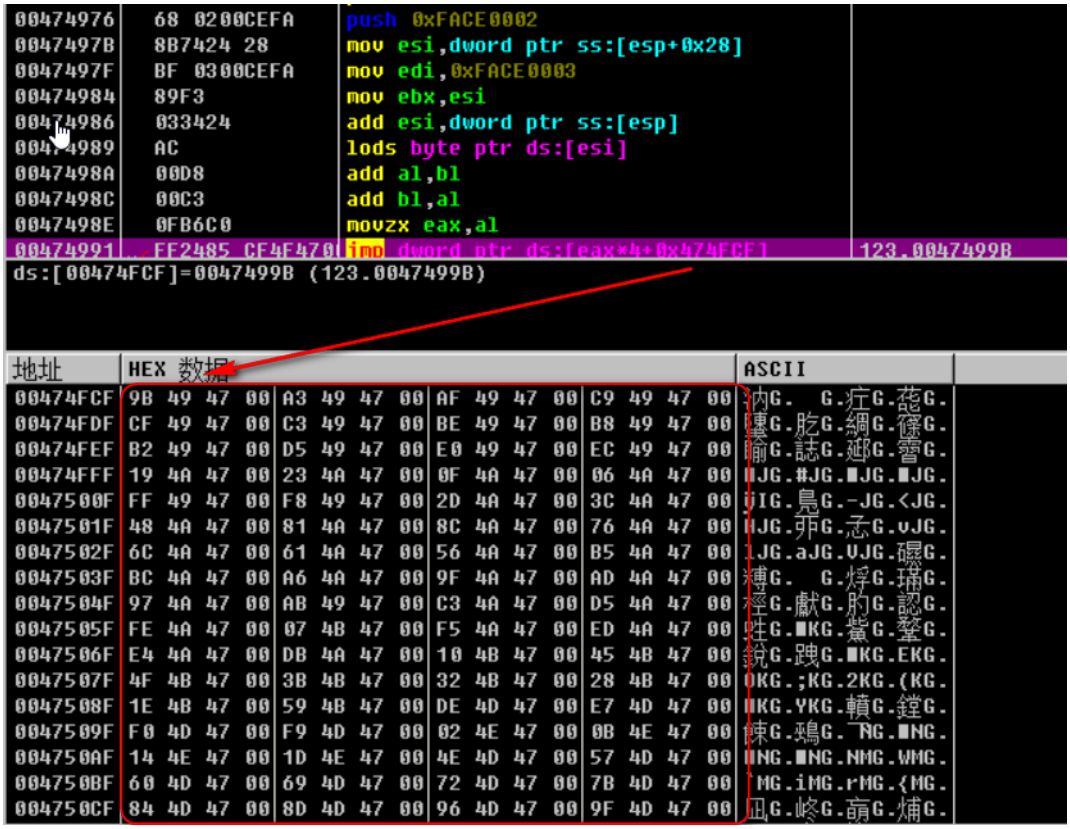
答：

1表示使用

0表示未使用（后期优化掉）



2、ESI_Matching_Array与VmHandle块对应（我整理了一份，未必全对的只作为参考）



3、判断用户解析Opcode有没有需要特殊处理的指令

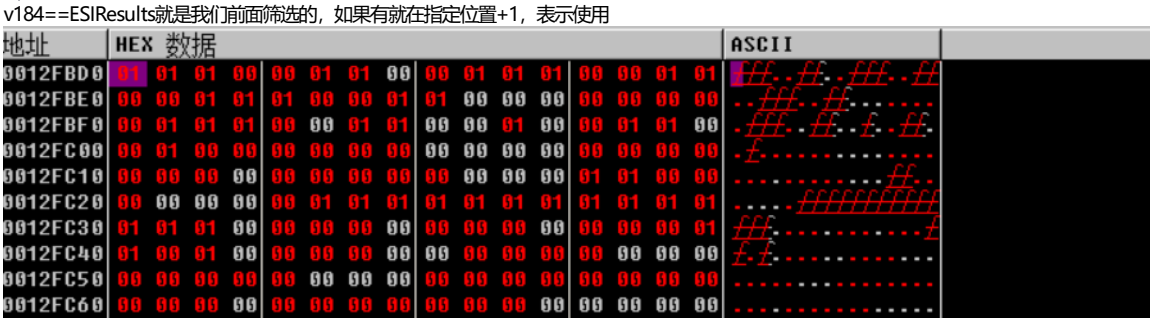
这些都是一些不常用的指令，如果存在就在ESIResults[X]=1，表示使用

```
v18 = TCollection::GetCount_0((int)v218) - 1; // 流程个数
if ( v18 >= 0 )
{
  AddrNumber2 = v18 + 1;
  v19 = 0;
  while ( 1 )
  {
    v217 = (struc_UserVmpPEInformation *)(*(int (__fastcall **)(int, int))(v218->This + 0x10))(v218->This, v19);
    if ( v217->Executable )                 // 可执行文件类型：1=PE 2=ELF
    {
      v20 = TCollection::GetCount_0((int)v217) - 1;// 获取用户Vmp的Opcode总行数（并非我们实际要VM的行数）
      if ( v20 >= 0 )                       // 判断行数是否正确
        break;
    }
LABEL_88:
    ++v19;
    if ( !--AddrNumber2 )
      goto LABEL_89;
  }
  Number = v20 + 1;
  i = 0;
  while ( 1 )                               // 解析用户要加密Opcode信息
  {
    v22 = (struct_DisassemblyFunction *)GetItem_7((int)v217, i, v21);// 软件最开始执行反汇编Opcode时候就已经调用Disassembly解析所有用户要加密的Opcode信息
    v23 = v22;
    if ( !v22->struct_VmFunctionAddr || *(_BYTE *)(v22->struct_VmFunctionAddr + 8) != 0xD )// 目前v22->Address有值的情况是：1、Call
      break;
LABEL_87:
    ++i;
    if ( !--Number )
      goto LABEL_88;
  }
  v24 = v22->VMOpcode;
  if ( v24 <= 0x3A )
  {
    if ( v24 != 0x3A )
    {
      switch ( v24 )
      {
```

假设找到的话执行Vmp_GetVmHandleIndex函数

```
// 函数功能:
// 1、ESI_Matching_Array数组找到符合条件的在ESIResults[X]=1
int __userpurge Vmp_GetVmHandleIndex@<eax>(int Results@<eax>, char a2@<dl>, char a3@<cl>, char a4, int a5)
{
  signed int Numberume; // edx@1
  int *ESI_Matching_Array; // edi@1
  _BYTE *ESIResults; // ecx@1
  char v8; // [sp+Eh] [bp-2h]@1
  char v9; // [sp+Fh] [bp-1h]@1

  v8 = a3;
  v9 = a2;
  Numberume = 0xCC;                             // esi数组个数
  ESI_Matching_Array = ::ESI_Matching_Array;
  ESIResults = (_BYTE *)(a5 - 0xCC);
  while ( (_WORD)Results != *((_WORD *)ESI_Matching_Array + 1)// 找到符合就退出，并在指定ESIResults[X]=1
       || *((_BYTE *)ESI_Matching_Array + 4) != v9
       || *((_BYTE *)ESI_Matching_Array + 5) != v8
       || *((_BYTE *)ESI_Matching_Array + 6) != a4 )
  {
    ++ESIResults;
    ESI_Matching_Array += 2;
    if ( !--Numberume )
      return Results;
  }
  *ESIResults = 1;
  return Results;
}
```

v184==ESIResults就是我们前面筛选的，如果有就在指定位置+1，表示使用

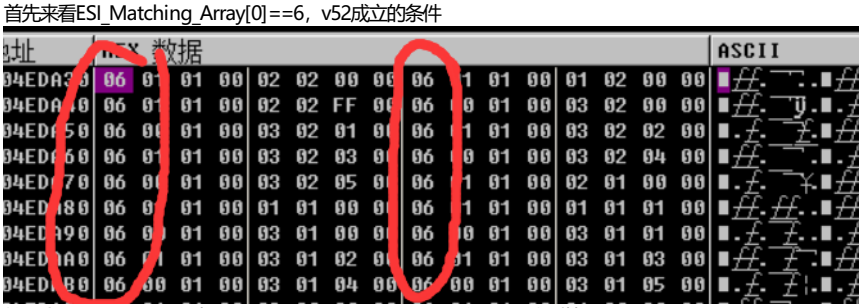


4、将Jmp Handle跟Jmp VMDispatcher分别存储

```
  if ( ArrayNumber >= 0 )
  {                                             // 将Jmp Handle跟Jmp VMDispatcher分别存储
    AddrNumber2 = ArrayNumber + 1;
    v46 = 0;
    do
    {
      v47 = (struct_VmFunctionAddr *)TCollection::GetItem_4(v7->struc_SavePartDisasmFunData1, v46);// 获取特殊解析Opcode信息（JmpAddr）
      v214 = v47;
      LOBYTE(v47) = LOBYTE(v47->Magic) - 5;     // 判断类型1、解析Jmp dword ptr [eax*4+JumpAddr] 值是0xB   2、解析Jmp VMDispatcher              值是0x5
      v16 = (unsigned __int8)v47 < 2u;
      v48 = (_BYTE)v47 - 2;
      if ( v16 )                                // 0~6成立
        goto LABEL_384;
      v49 = v48 - 4;
      if ( !v49 )                               // v49 == 0Xb成立 目前已知：1、解析Jmp dword ptr [eax*4+JumpAddr] 值是0xB
      {
        v52 = Type < 7u;
        if ( Type <= 7u )
          v52 = _bittest((const signed __int32 *)(8 * Number_1 + 0x4EDA30), Type & 0x7F);// 0x4EDA30 = Esi表
        if ( v52 )
        {
          AddrNumber1 = DateTimeToStr_2((int)v7, v214->FunAddr, v214->dword14);// 然后拿v214->FunAddr去struc_SaveAllDisasmFunData数组里面找到符合,并返回找到的数组下标
          v55 = (struct_DisassemblyFunction *)GetItem_7((int)v7, AddrNumber1, This);// 读取N个结构体内容
          if ( ESIResults[Number_1] )           // 前面通过ESI_Matching_Array计算出来的结果
            v214->CheckDisassemblyFunction = (int)v55;// 将找到的DisassemblyFunction结构体保存起来
          *(_DWORD *)&v7->Esi_Addr[4 * Number_1] = v55;// 数组保存起来，Number_1保存数组个数
        }
        ++Number_1;
        goto LABEL_102;
      }
      if ( v49 == 1 )                           // v49 == 0xC 成立
      {
LABEL_384:
        v50 = DateTimeToStr_2((int)v7, v214->FunAddr, v214->dword14);
        v214->CheckDisassemblyFunction = GetItem_7((int)v7, v50, v51);
      }
LABEL_102:
      ++v46;
      --AddrNumber2;
    }
```

4、1 Jmp Handle处理方法

首先来看ESI_Matching_Array[0]==6，v52成立的条件



然后拿v214->FunAddr去struc_SaveAllDisasmFunData数组里面找到符合

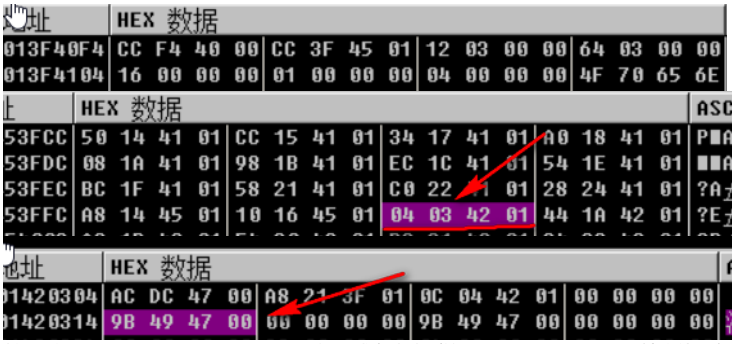
```
AddrNumber1 = DateTimeToStr_2((int)v7, v214->FunAddr, v214->dword14);// 然后拿v214->FunAddr去struc_SaveAllDisasmFunData数组里面找到符合,并返回找到的数组下标
v55 = (struct_DisassemblyFunction *)GetItem_7((int)v7, AddrNumber1, This);// 读取N个结构体内容
```

DateTimeToStr_2函数查找过程，查找到返回该数组下标，然后用GetItem_7读取出该数组：

条件是：v214->FunAddr == struct_DisassemblyFunction->LODWORD_VMP_Address

基础版（struc_SaveAllDisasmFunData）：

```
00000
00000 ; 保存内容：
00000 ; 1、解析后Opcode信息
00000 ; 2、作者设计VmpHandle开始和结束地址
00000 ; 3、壳的入口
00000 struc_VmpOpcode struc ; (sizeof=0x3DC, align=0x4, mappedto_291)
00000 This            dd ?
00004 _prev_node      dd ?                    ; 指向一个链表结构体：指向struc_VmpAllData
00008 struc_SaveAllDisasmFunData dd ?         ; 保存所有解析后的Opcode函数信息
0000C Magic           dd ?                    ; 数组排列乱序：使用就是1，未使用就是0。
00010 struc_SavePartDisasmFunData1 dd ?       ; 保存特殊解析后的Opcode信息,专门JmpAddrHandle函数
00014 gap14           dd ?                    ; 兼容64位
00018 VmpStubStart    dd ?                    ; Vmp壳入口
0001C dword1C         dd ?                    ; 兼容64位
```



然后判断ESIResults[Number_1]==1，如果成立将找到的DisassemblyFunction结构体保存起来

```
if ( ESIResults[Number_1] )           // 前面通过ESI_Matching_Array计算出来的结果
  v214->CheckDisassemblyFunction = (int)v55;// 将找到的DisassemblyFunction结构体保存起来
```

最后将该找到的DisassemblyFunction结构保存到v7->Esi_Addr[4 * Number_1]

```
(_DWORD *)&v7->Esi_Addr[4 * Number_1] = v55;// 数组保存起来，Number_1保存数组个数
```

4、2 Jmp VMDispatcher处理方法

1、直接去struc_SaveAllDisasmFunData数组查找，找到直接保存

2、与Jmp Handle相比少了ESIResults[Number_1]过滤跟将结果保存到*(_DWORD *)&v7->Esi_Addr[4 * Number_1]

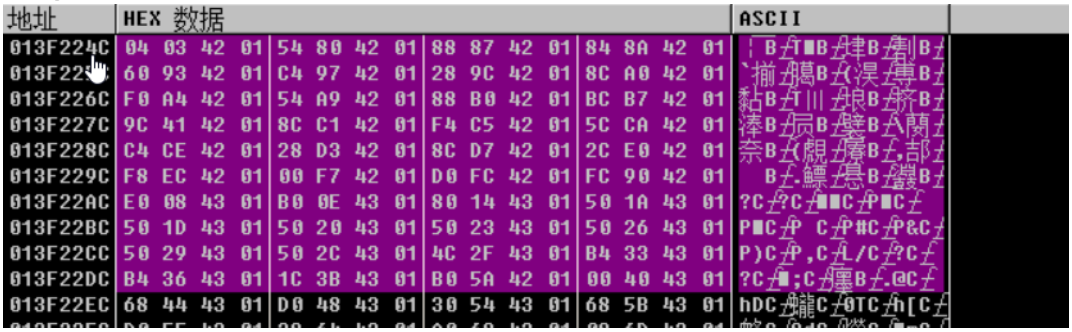
```
LABEL_384:
      v50 = DateTimeToStr_2((int)v7, v214->FunAddr, v214->dword14);
      v214->CheckDisassemblyFunction = GetItem_7((int)v7, v50, v51);
```

4、3 执行完毕结果

IDA定义的结构体：

```
00000074 field_74        dd ?
00000078 field_78        dd ?
0000007C field_7C        dd ?
00000080 struc_VmpOpcodePY_80 dd ?            ; 具体不知道干嘛的,里面都是通过随机数填充字段跟Esi有关
00000084 struc_VmpOpcodePY_84 dd ?            ; 与上面同理使用的结构体与80相同，4个一组84~90
00000088 struc_VmpOpcodePY_88 dd ?
0000008C struc_VmpOpcodePY_8C dd ?
00000090 struc_VmpOpcodePY_90 dd ?
00000094 struc_VmpOpcodePY_94 dd ?            ; 与上面同理使用的结构体与80相同，4个一组94~A0
00000098 struc_VmpOpcodePY_98 dd ?
0000009C struc_VmpOpcodePY_9C dd ?
000000A0 struc_VmpOpcodePY_A0 dd ?
000000A4 Esi_Addr        db 816 dup(?)        ; 保存的都是struc_DisassemblyFunction结构
000003D4 struc_VmpOpcodePY_3D4 dd ?           ; 该结构体保存数据分别是：1、修复重定位地址(push XXXX<-push 0xFACE0002) 2、edi（VMcontext<-mov edi,0xFACE0003）指向地址修复
000003D8 struc_PushRegister dd ?              ; 主要是保存push XX（寄存器环境）对应的数字，后面会进行乱序操作
000003DC struc_VmpOpcode ends
000003DC
```

OD显示：



4、4 Jmp VMDispatcher与Jmp Handle的含义是什么意思？

如图所示：

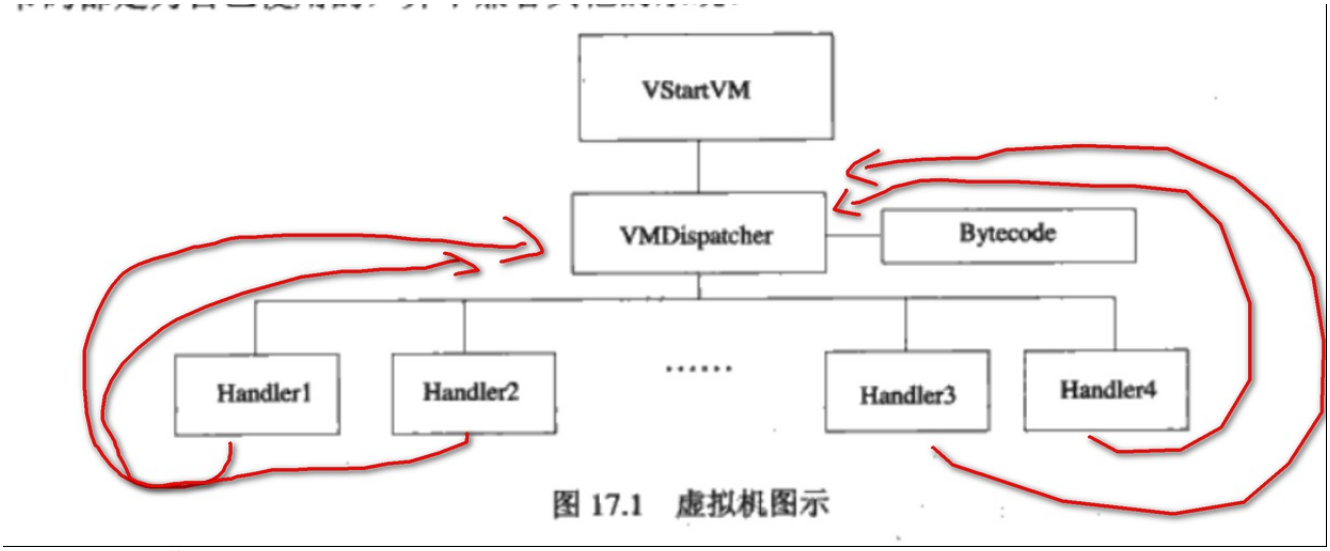


图 17.1 虚拟机图示

Jmp VMDispatcher就是:

注意看jmp short 00474989这一句，每个Handle块执行完毕都是跳回到VMDispatcher进行下一轮字节解析

```
00474986   033424          add esi,dword ptr ss:[esp]
00474989   AC              lods byte ptr ds:[esi]
0047498A   00D8            add al,bl
0047498C   00C3            add bl,al
0047498E   0FB6C0          movzx eax,al
00474991   FF2485 CF4F470  jmp dword ptr ds:[eax*4+0x474FCF]
00474998   5E              pop esi                          123.0047499B
00474999   EB E9           jmp short 123.00474984
0047499B   80E0 3C         and al,0x3C
0047499E   FF3407          push dword ptr ds:[edi+eax]
004749A1   EB E6           jmp short 123.00474989
```

Jmp Handle就是:

VmpHandle块,每个Handle对应不同的功能

```
地址       HEX 数据                                          ASCII
00474FCF   9B 49 47 00 A3 49 47 00 AF 49 47 00 C9 49 47 00   纳G. G.疟G
00474FDF   CF 49 47 00 C3 49 47 00 BE 49 47 00 B8 49 47 00   镭G.肮G.綢G
00474FEF   B2 49 47 00 D5 49 47 00 E0 49 47 00 EC 49 47 00   瞼G.誌G.邾G
00474FFF   19 4A 47 00 23 4A 47 00 0F 4A 47 00 06 4A 47 00   ■JG.#JG.■JG
0047500F   FF 49 47 00 F8 49 47 00 2D 4A 47 00 3C 4A 47 00   ÿIG.鳥G.-JG
0047501F   48 4A 47 00 81 4A 47 00 8C 4A 47 00 76 4A 47 00   HJG.弗G.丞G
0047502F   6C 4A 47 00 61 4A 47 00 56 4A 47 00 B5 4A 47 00   lJG.aJG.VJG
0047503F   BC 4A 47 00 A6 4A 47 00 9F 4A 47 00 AD 4A 47 00   糠G. G.焊G
0047504F   97 4A 47 00 AB 49 47 00 C3 4A 47 00 D5 4A 47 00   桎G.獻G.肭G
0047505F   FE 4A 47 00 07 4B 47 00 F5 4A 47 00 ED 4A 47 00   蛀G.■KG.鷺G
0047506F   E4 4A 47 00 DB 4A 47 00 10 4B 47 00 45 4B 47 00   銳G.踄G.■KG
```

5、根据前面符合Jmp Handle满足条件的Number_1作为循环因子

```
v230 = GetSize_V(a2a);
Size = (unsigned __int8)Global_Size[(unsigned __int8)v230];// 获取大小 分别对应1 2 4 8
v57 = Number_1;                         // 经过前面筛选Number_1=0XCC，一般HandleX与ESI_Matching_Array都是一一对应，大小都是0xCC
if ( Number_1 <= 0xFF )                 // 获取所有解析后Jmp dword ptr [eax*4+JumpAddr] 的个数(也就是HandleX)
{
  do
  {
    v58 = (struct_DisassemblyFunction *)(*(int (**)(void))(v7->This + 0x14))();// New空间，内容继续保存在struc_SavePartDisasmFunData+4
    v58->VMOpcode = 0x23;
    v58->Magic = v230;
    __linkproc__ DynArraySetLength(&v58->ReadHexAddress, (int)dword_47CE4C, 1, Size);
    v58->ReadHexLen = Size;
    v213 = &v58->First.About_Lval_Byte_Word_Dword;
    v58->First.ModRM_mod__Or__Size = 2;
    *v213 = v230;
    v213[0xF] = v230;                   // Lavl_Btye_Word_Dword db ?              ; 根据DisplacementLen判断读几个字节，GetSize的返回值
    SetDisassemblyFunction_Address((int)v58, 0xB, 0, 0, 0, 0);
    ++v57;
  }
  while ( v57 != 0x100 );               // 强行扩充到0xFF大小，不足的new struc_SavePartDisasmFunData和struct_VmFunctionAddr结构
}
```

1、经过前面筛选Number_1=0XCC，一般HandleX与ESI_Matching_Array都是一一对应，大小都是0xCC

```
Numberume = 0xCC;
v15 = (const signed __int32 *)ESI_Matching_Array;// ESI_Matching_Array每一组是8个字节，一共有0Xcc组，也就是总长度是0x660=8*0xCC
```

2、只是设置的基本的Mod信息跟VmpOpcode=0x23

3、new出来的struct_VmFunctionAddr结构只是设置了助记符=0xB

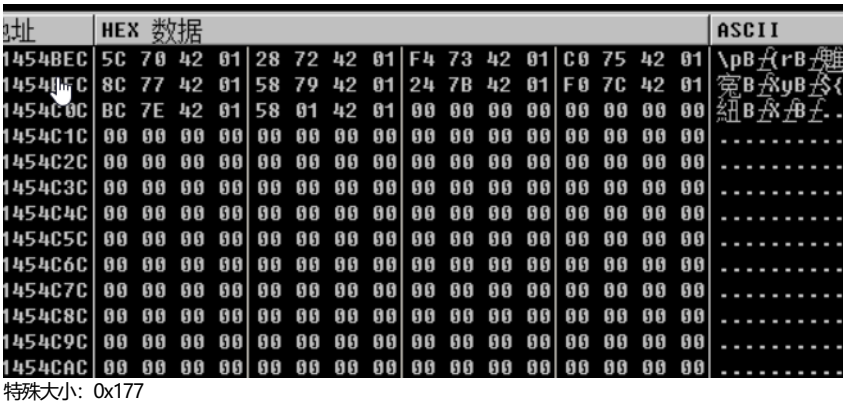
4、强行扩充到0xFF大小，不足的new struc_SavePartDisasmFunData和struct_VmFunctionAddr结构，具体作用不明

5、扩充前后对比

未扩充前:

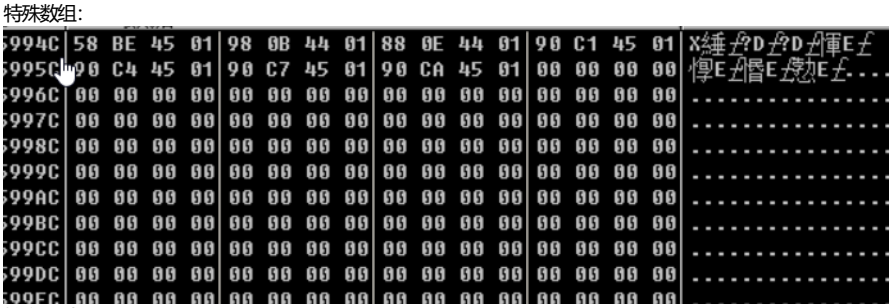
基础大小：0x312

基础数组:



特殊大小：0x177

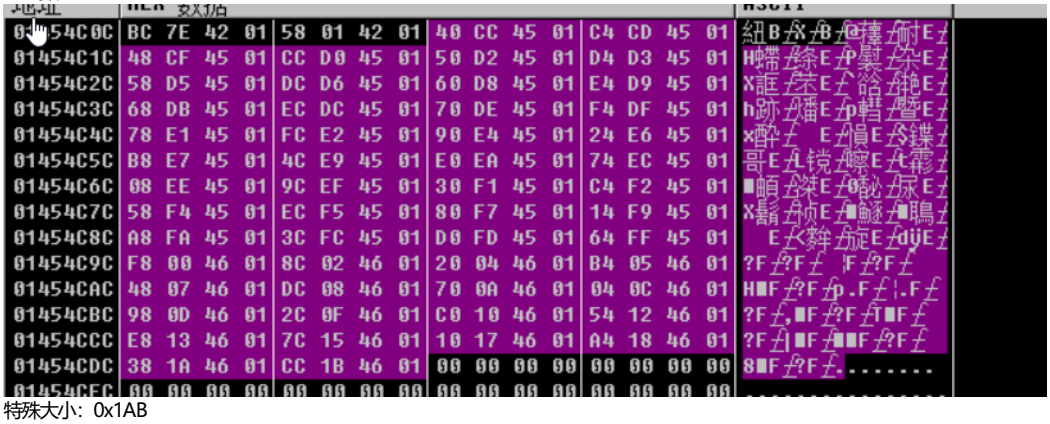
特殊数组：



扩充后(多了0x34个组)：

基础大小：0x346

基础数组：



特殊大小：0x1AB

特殊数组：

6、将不符合条件的struc_SaveAllDisasmFunData和struc_SavePartDisasmFunData1从数组中删除

```
  v181 = (int *)ESIResults;
  do                                    // 将不符合条件的struc_SaveAllDisasmFunData和struc_SavePartDisasmFunData1从数组中删除
  {
    if ( !*(_BYTE *)v181 && *(_DWORD *)&v7->Esi_Addr[4 * v59] )// 专门找EsiResults[i]==0的,EsiResults[i]与v7->Esi_Addr[4 * v59]一一对应
    {
      for ( i = TList::IndexOf(*(_DWORD *)&v7->Esi_Addr[4 * v59], v56);// i = 查找EsiREsults对应的地址在struc_SaveAllDisasmFunData->ArrayAddress第几组
            ;
            (*(void (__fastcall **)(_DWORD, int))(v7->This + 0xC))(v7->This, i) )// 从数组中删除掉不符合条件的地址信息(struc_SaveAllDisasmFunData和struc_SavePartDisasmFunData1)
      {
        VmpOpcode = *(_WORD *)(GetItem_7((int)v7, i, v60) + 0x24) - 8;
        v16 = VmpOpcode < 2u;
        v62 = VmpOpcode - 2;
        if ( v16 || v62 == 2 )          // 判断是否符合条件:8~9和0xC
                                        // 1、VmOpcode==0xC  jmp r/m32(16) FF/4   jmp m16:16(32) FF/5
          break;
      }
      (*(void (__fastcall **)(_DWORD, int))(v7->This + 0xC))(v7->This, i);// 将数组里面的地址删除
      *(_DWORD *)&v7->Esi_Addr[4 * v59] = 0;    // 针对前面保存的特定JmpAddr地址清零
    }
    ++v59;
    v181 = (int *)((char *)v181 + 1);
  }
  while ( v59 != 0xCC );                // Esi最大不超过0xCC, 删除的基本都是EsiResults[i]==0的对应的XXX
```

1、专门找ESIResults[X] == 0的

2、ESIResults[X]与v7->Esi_Addr[4 * X]——对应
3、找到VmpOpcode值是：0~9、0xC则退出，符合条件的基本上是：Jmp VMDispatcher找到后把该数组元素删除
4、清零v7->Esi_Addr[4 * X] = 0
5、看了一圈基本上是把整个HandleX解析信息的都删除，jmp XXXX标志结束
6、未被删除的

```
地址      HEX 数据                                          ASCII
013F224C  04 03 42 01 54 80 42 01 88 87 42 01 00 00 00 00  ♦♥B☺T€B☺ê‡B☺....
013F225C  00 00 00 00 C4 97 42 01 28 9C 42 01 00 00 00 00  ....─ùB☺(£B☺....
013F226C  00 00 00 00 54 A9 42 01 88 B0 42 01 BC B7 42 01  ....T⌐B☺ê░B☺╝╖B☺
013F227C  00 00 00 00 00 00 00 00 F4 C5 42 01 5C CA 42 01  ........⌠┼B☺\╩B☺
013F228C  00 00 00 00 00 00 00 00 8C D7 42 01 2C E0 42 01  ........î╫B☺,αB☺
013F229C  F8 EC 42 01 00 00 00 00 00 00 00 00 FC 90 42 01  °∞B☺........ⁿÉB☺
013F22AC  E0 08 43 01 00 00 00 00 00 00 00 00 00 00 00 00  α◘C☺............
013F22BC  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
013F22CC  00 00 00 00 50 2C 43 01 4C 2F 43 01 B4 33 43 01  ....P,C☺L/C☺┤3C☺
013F22DC  00 00 00 00 00 00 00 00 B0 5A 42 01 00 40 43 01  ........░ZB☺.@C☺
013F22EC  00 00 00 00 00 00 00 00 30 54 43 01 00 00 00 00  ........0TC☺....
013F22FC  00 00 00 00 38 64 43 01 A0 68 43 01 00 00 00 00  ....8dC☺áhC☺....
013F230C  00 00 00 00 D8 75 43 01 00 00 00 00 00 00 00 00  ....╪uC☺........
013F231C  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
013F232C  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
013F233C  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
013F234C  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

总结：
1、ESIResults[X]与v7->Esi_Addr[4 * X]——对应
2、ESIResults[X]==0，那么取对应的v7->Esi_Addr[4 * X]数组内容（struc_SaveAllDisasmFunData结构体）
3、struc_SaveAllDisasmFunData与struc_SavePartDisasmFunData1数组里删除该HandleX信息
4、判断到jmp XXXX为结束点，也就是整个Handle解析的信息都清除掉

```
004749B2    58              pop eax                                 014113C0
004749B3    65:FF30         push dword ptr gs:[eax]
004749B6  ^ EB D1           jmp short 123.00474989
```

5、ESIResults[X]==0就是不使用的了
疑问：
1、这个不删除会如何？删除会如何？后面待分晓

7、随机数填充struct_VmpOpcodePY_80结构

```
if ( *(_BYTE *)(v7->_prev_node + 0x48) & 8 )  // 保护等于，默认为8
{
  (*(void (__fastcall **)(_DWORD, signed int))(*(_DWORD *)v7->struct_VmpOpcodePY_80 + 0x10))(
    *(_DWORD *)v7->struct_VmpOpcodePY_80,
    3);                                 // 随机创建X个结构体struc_47
  v230 = 0;
  do
  {
    v63 = (_DWORD *)*(&v7->struct_VmpOpcodePY_84 + (unsigned __int8)v230);
    (*(void (__fastcall **)(_DWORD, signed int))(*v63 + 0x10))(*v63, 3);
    v64 = (_DWORD *)*(&v7->struct_VmpOpcodePY_94 + (unsigned __int8)v230);
    (*(void (__fastcall **)(_DWORD, signed int))(*v64 + 0x10))(*v64, 3);
    ++v230;
  }
  while ( v230 != 4 );                  // 与上面同理，通过随机数填充结构体
  Number_1 = -1;
  v181 = &dword_4EDA2C;
```

sub_49FB90函数分析：

```
// 函数功能：
// 1、word_4EE0EC[__linkproc__ RandInt()]随机数0~3之间取下标值，填充到struct_VmpOpcodePY_80
// 2、随机填充与Encoding of a p-code加密有关，1.21版本自带
int __usercall sub_49FB90@<eax>(struct_VmpOpcodePY_80 *a1@<eax>, int Constant@<edx>)
{
  int v2; // esi@1
  struct_VmpOpcodePY_80 *v3; // ebx@1
  int This; // ecx@1

  v2 = Constant;
  v3 = a1;
  a1->RandomWord_4EE0EC = word_4EE0EC[__linkproc__ RandInt()];
  return sub_49F958(v3, v2, This);
}
```

```
000000 struct_VmpOpcodePY_80 struc ; (sizeof=0x1C, mappedto_330)
000000 This            dd ?
000004 _prev_node      dd ?                    ; 返回上一层结构体：struct_VmpOpcode
000008 struc_46        dd ?
00000C field_C         dd ?
000010 Size            dd ?                    ; 来源未知
000014 RandomWord_4EE0EC dw ?                  ; 从word_4EE0EC[__linkproc__ RandInt()]随机数0~3之间取下标值
000014                                         ; word_4EE0EC[0]=add 4
000014                                         ; word_4EE0EC[1]=sub 34
000014                                         ; word_4EE0EC[2]=xor 5
000016 field_16        db ?
000017 field_17        db ?
000018 field_18        dd ?
00001C struct_VmpOpcodePY_80 ends
00001C
```

sub_49F958函数分析：

```
  v3 = a1;
  TlistArrayClear((int)a1, This);
  v4 = 0;
  while ( 1 )
  {
    while ( 1 )
    {
      while ( 1 )
      {
        do
          v5 = word_4EE0D8[__linkproc__ RandInt()];// 随机数下标0~A范围取内容,内容分别是：04、05、34、43、44、5c、29、2A、5D、31
        while ( v5 == v4 );                     // 取值!=0就结束循环
        v12 = 0i64;                             // 清零
        if ( (signed int)(unsigned __int16)v5 > 0x31 )
          break;                                // 符合条件：word_4EE0D8[2][3][4][5][8]
        if ( v5 == 0x31 )                       // !=word_4EE0D8[9]
        {
          if ( LOBYTE(v3->Size) )
            goto LABEL_25;
        }
        else
        {
          if ( v5 == 4 )                        // word_4EE0D8[0]
            goto LABEL_12;
          if ( v5 == 5 )                        // word_4EE0D8[1]
          {
            LODWORD(v7) = __linkproc__ RandInt();
            v12 = Dword_Extension_Qword(v7);
            goto LABEL_25;
          }
          if ( (unsigned int)(unsigned __int16)v5 - 0x29 >= 2 )
            goto LABEL_25;
          if ( v4 != 4 && (unsigned __int16)(v4 - 0x29) >= 2u && v4 != 52 )
          {
            v12 = 1i64;
            goto LABEL_25;
          }
        }
      }
      if ( v5 == 0x34 )                         // word_4EE0D8[2]
        break;
      }
     }
     if ( v5 == 0x34 )                              // word_4EE0D8[2]
       break;
     if ( (unsigned int)(unsigned __int16)v5 - 0x43 >= 2 )
       goto LABEL_25;
     if ( (unsigned __int16)(v4 - 0x43) >= 2u )
     {
       v8 = (unsigned __int8)Global_Size[LOBYTE(v3->Size)];
       v12 = __linkproc__ RandInt() + 1;
       goto LABEL_25;
     }
    }
LABEL_12:
    if ( v4 != 4 && (unsigned __int16)(v4 - 0x29) >= 2u && v4 != 52 )
    {
      LODWORD(v6) = __linkproc__ RandInt();
      v12 = Dword_Extension_Qword(v6);
LABEL_25:
      v4 = v5;
      v9 = sub_49F944((int)v3);                 // ADD添加
      v9->RandomWord_4EE0D8 = v5;
      Move((char *)&v12, (char *)v9->AddrRandomBuff, (unsigned __int8)Global_Size[LOBYTE(v9->Size)]);
      result = TCollection::GetCount_0((int)v3);// 获取数组个数
      if ( result >= 0x64 )
        return result;
      if ( TCollection::GetCount_0((int)v3) > Constant_1 )// Add添加的空间是否 > 传进来的常量
      {
        result = __linkproc__ RandInt() - 1;
        if ( !result )
          return result;
      }
    }
  }
```

1、通过随机数取word_4EE0D8数组的下标，符合条件的跳到赋值的地方

2、退出条件是：要Add添加几组元素由Constant（参数2）决定，外加一句RandInt（1），百分之50%几率再来一次

```
    return result;
  if ( TCollection::GetCount_0((int)v3) > Constant_1 )// Add添加的个数是否 > 传进来的常量
  {
    result = __linkproc__ RandInt() - 1;
    if ( !result )
      return result;
  }
```

3、它们使用的结构如下：

```
00000000 struc_46        struc ; (sizeof=0x14, mappedto_331)
00000000 This            dd ?
00000004 ArrayAddress    dd ?                    ; struc_47
00000008 AddrNumber      dd ?                    ; 个数
0000000C Magic           dd ?                    ; 标志位，使用了就是0 （待定）   ChenckAddrNumber_NewOrFree
00000010 field_10        dd ?
00000014 struc_46        ends
00000014
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 struc_47        struc ; (sizeof=0x30, mappedto_332)
00000000 This            dd ?
00000004 _prev_node      dd ?
00000008 RandomWord_4EE0D8 dw ?                  ; 1、word_4EE0D8[__linkproc__ RandInt()];// 随机数下标0~A范围取内容
00000008                                         ; 2、
00000008                                         ; 0x4=add          需要AddrRandomBuff作为目标操作数
00000008                                         ; 0x5=xor          需要AddrRandomBuff作为目标操作数
00000008                                         ; 0x34=sub         需要AddrRandomBuff作为目标操作数
00000008                                         ; 0x43=rol         循环左移，需要AddrRandomBuff作为目标操作数
00000008                                         ; 0x44=ror         逻辑右移，需要AddrRandomBuff作为目标操作数
00000008                                         ; 0x5C=not         单操作数
00000008                                         ; 0x29=inc         单操作数
00000008                                         ; 0x2A=dec         单操作数
00000008                                         ; 0x5D=neg         单操作数
00000008                                         ; 0x31=bswap       单操作数
0000000A Size            dw ?                    ; 配合byte_4EDA20[LOBYTE(v9->Size)]使用
0000000C field_C         dd ?
00000010 AddrRandomBuff  dd 8 dup(?)             ; 1、根据RandomWord_4EE0D8结果计算不同的值，大小是byte_4EDA20[LOBYTE(v9->Size)] 1、2、4、8
00000010                                         ; 2、作用是跟后面进行add sub xor rol rot进行运算
00000030 struc_47        ends
00000030
```

总结:

0、变形总结对照

RandomWord_4EE0EC是对add al,bl的变形

RandomWord_4EE0D8是对add bl,al的变形

```
00474974   9C              pushfd
00474975   60              pushad
00474976   68 0200CEFA     push 0xFACE0002
0047497B   8B7424 28       mov esi,dword ptr ss:[esp+0x28]
0047497F   BF 0300CEFA     mov edi,0xFACE0003
00474984   89F3            mov ebx,esi
00474986   033424          add esi,dword ptr ss:[esp]
00474989   AC              lods byte ptr ds:[esi]
0047498A   00D8            add al,bl
0047498C   00C3            add bl,al
```

1、填充这些数据到底怎么使用?

2、struc_47数据使用

我们发现执行完毕后一共有6组

```
地址       HEX 数据                                          ASCII
01461E00   60 1D 46 01  90 1D 46 01  AC 1D 46 01  C8 1D 46 01  `■F╩?F╩?F╩?F╩
01461E10   E4 1D 46 01  24 1E 46 01  00 00 00 00  00 00 00 00  ?F╩$■F╩........
```

第一组:

struc_47->RandomWord_4EE0D8=0x29 ->inc

struc_47->AddrRandomBuff=0x1

第二组:

struc_47->RandomWord_4EE0D8=0x43 ->rol

struc_47->AddrRandomBuff=0x5

第三组:

struc_47->RandomWord_4EE0D8=0x5C ->not

struc_47->AddrRandomBuff=0x5

第四组:

struc_47->RandomWord_4EE0D8=0x34 ->sub

struc_47->AddrRandomBuff=0xB0

第五组:

struc_47->RandomWord_4EE0D8=0x5C ->not

struc_47->AddrRandomBuff=0x0

第六组:

struc_47->RandomWord_4EE0D8=0x05 ->xor

struc_47->AddrRandomBuff=0x7A

刚好对应以下6句，因为1、3、5是单操作数所以struc_47->AddrRandomBuff不使用

```
00405062   89F3            mov ebx,esi
00405064   033424          add esi,dword ptr ss:[esp]
00405067   8A06            mov al,byte ptr ds:[esi]
00405069   00D8            add al,bl
0040506B   FEC0            inc al
0040506D   C0C0 05         rol al,0x5
00405070   F6D0            not al
00405072   2C B0           sub al,0xB0
00405074   F6D0            not al
00405076   34 7A           xor al,0x7A
00405078   8D76 01         lea esi,dword ptr ds:[esi+0x1]
0040507B   00C3            add bl,al
0040507D   0FB6C0          movzx eax,al
00405080   FF3485 BB514000 push dword ptr ds:[eax*4+0x4051BB]
```

3、struct_VmpOpcodePY_80->RandomWord_4EE0EC使用

第一种RandomWord_4EE0EC=0x4，注意看405069跟40507B这两句是add

```
00405067   8A06          mov al,byte ptr ds:[esi]
00405069   00D8          add al,bl
0040506B   FEC0          inc al
0040506D   C0C0 05       rol al,0x5
00405070   F6D0          not al
00405072   2C B0         sub al,0xB0
00405074   F6D0          not al
00405076   34 7A         xor al,0x7A
00405078   8D76 01       lea esi,dword ptr ds:[esi+0x1]
0040507B   00C3          add bl,al
0040507D   0FB6C0        movzx eax,al
00405080   FF3485 BB51400( push dword ptr ds:[eax*4+0x4051BB]
```

第二种RandomWord_4EE0EC=0x34，注意看405069跟40507B这两句是sub

```
00405064   033424        add esi,dword ptr ss:[esp]      kernel32.77523C45
00405067   8A06          mov al,byte ptr ds:[esi]
00405069   28D8          sub al,bl
0040506B   FEC0          inc al
0040506D   C0C0 05       rol al,0x5
00405070   F6D0          not al
00405072   2C B0         sub al,0xB0
00405074   F6D0          not al
00405076   34 7A         xor al,0x7A
00405078   8D76 01       lea esi,dword ptr ds:[esi+0x1]
0040507B   28C3          sub bl,al
0040507D   0FB6C0        movzx eax,al
00405080   FF3485 BB51400( push dword ptr ds:[eax*4+0x4051BB]
```

RandomWord_4EE0EC=0x34

第三种RandomWord_4EE0EC=0x5，注意看405069跟40507B这两句是xor

```
00405062   89F3          mov ebx,esi
00405064   033424        add esi,dword ptr ss:[esp]      kernel32.77523C45
00405067   8A06          mov al,byte ptr ds:[esi]
00405069   30D8          xor al,bl
0040506B   FEC0          inc al
0040506D   C0C0 05       rol al,0x5
00405070   F6D0          not al
00405072   2C B0         sub al,0xB0
00405074   F6D0          not al
00405076   34 7A         xor al,0x7A
00405078   8D76 01       lea esi,dword ptr ds:[esi+0x1]
0040507B   30C3          xor bl,al
0040507D   0FB6C0        movzx eax,al
00405080   FF3485 BB51400( push dword ptr ds:[eax*4+0x4051BB]
00405087   C3            retn
```

8、使用struct_VmpOpcodePY_80~A0结构

```
  v181 = &dword_4EDA2C;
  while ( Number_1 != -1 )                      // 遍历struct_DisassemblyFunction结构体来匹配信息，结果保存在struc_SaveAllDisasmFunData结构
  {
    if ( *(_DWORD *)&v7->Esi_Addr[4 * Number_1] )
    {
      i = TList::IndexOf(*(_DWORD *)&v7->Esi_Addr[4 * Number_1], v65);
LABEL_122:
      v66 = i;
      ArrayNumber_1 = TCollection::GetCount_0((int)v7) - 1;// struc_SaveAllDisasmFunData->ArrayNumber（基础版）
      v68 = __OFSUB__(ArrayNumber_1, v66);
      v69 = ArrayNumber_1 - v66;
      if ( !((v69 < 0) ^ v68) )
      {
        AddrNumber2 = v69 + 1;
        do
        {
          v70 = (struct_DisassemblyFunction *)GetItem_7((int)v7, v66, v65);
          v71 = v70;
          LOWORD(v70) = v70->VMOpcode - 8;
          v16 = (unsigned __int16)v70 < 2u;
          v72 = (_WORD)v70 - 2;
          if ( v16 || v72 == 2 )
            break;                              // VMOpcode:8 9 =ret C=jmp符合条件
          if ( v71->VMOpcode == 4               // Add系列
            && 4 == v71->First.ModRM_mod__Or__Size// 11 register 提供 register 寻址。
            && !v71->First.ModRM_Reg_Or_SIB_index_Or_ModRM_rm
            && 4 == v71->Second.ModRM_mod__Or__Size
            && v71->Second.ModRM_Reg_Or_SIB_index_Or_ModRM_rm == 3 )// 符合条件的例如：add al,bl
          {
            v230 = v71->First.About_Lval_Byte_Word_Dword;
            if ( i )
            {
              if ( *(_BYTE *)v181 == 2 )
                v227 = (struct_VmpOpcodePY_80 *)*(&v7->struct_VmpOpcodePY_84 + *((_BYTE *)v181 + 1));// 前面随机数填充的字段
              else
                v227 = (struct_VmpOpcodePY_80 *)*(&v7->struct_VmpOpcodePY_94 + (unsigned __int8)v230);// 前面随机数填充的字段
            }
            else                                // i=0情况：第1次或则IndexOf返回0
            {
              v227 = (struct_VmpOpcodePY_80 *)v7->struct_VmpOpcodePY_80;// 前面随机数填充的字段
```

1、目前发现符合if条件的只有register寻址方式的并且是add aXX,BXX这种，每次都是两条组合出现

```
0047498A   00D8          add al,bl
0047498C   00C3          add bl,al
0047498E   EB E0         jmp short 123.00474970
00474990   AD            lods dword ptr ds:[esi]
00474991   01D8          add eax,ebx
00474993   01C3          add ebx,eax
00474995   50            push eax
00474996   EB DE         jmp short 123.00474976
```

2、通过来区分到底取struct_VmpOpcodePY_80、struct_VmpOpcodePY_84~90、struct_VmpOpcodePY_94~A0其中一组

3、判断v227->RandomWord_4EE0EC! =4

```
RandomNumber = v227->RandomWord_4EE0EC;// 4、34、5三个值其中一个
if ( RandomNumber != 4 )          // RandomWord_4EE0EC[1]、[2]情况
{
```

第一种：RandomWord_4EE0EC！=4执行流程

```
RandomNumber = v227->RandomWord_4EE0EC;// 4、34、5三个值其中一个
if ( RandomNumber != 4 )          // RandomWord_4EE0EC[1]、[2]情况
{
  v71->VMOpcode = RandomNumber;
  SetDisasm(v71, v65);
  v73 = TCollection::GetCount_0((int)v7) - 1;
  v68 = __OFSUB__(v73, v66 + 1);
  v75 = v73 - (v66 + 1);
  if ( !((v75 < 0) ^ v68) )
  {
    Number = v75 + 1;
    Size = v66 + 1;
    while ( 1 )                   // 遍历找到数组里面VmpOpcode==4的，就只有add才符合
    {
      v76 = (struct_DisassemblyFunction *)GetItem_7((int)v7, Size, v74);
      if ( v76->VMOpcode == 4 )   // add系列，找到就退出
        break;
      ++Size;
      if ( !--Number )
        goto LABEL_142;
    }
    v76->VMOpcode = RandomNumber;
    SetDisasm(v76, v74);
  }
}
```

第二种：RandomWord_4EE0EC==4执行流程

```
{
  a1a = (struct_DisassemblyFunction *)(*(int (**)(void))(v7->This + 0x14))();// Add添加
  v78 = TCollection::GetCount_0((int)v7);// 获取struc_SaveAllDisasmFunData.AddrNumber数组个数
  TList::Move(v7, v78 - 1, v66 + Size + 1);
  a1a->Magic = v71->Magic;
  v80 = (struc_47 *)TCollection::GetItem_9(v79, Size);
  a1a->VMOpcode = v80->RandomWord_4EE0D8;// word_4EE0D8[__linkproc__ RandInt()];// 随机数下标0~A范围取内容: dw 4, 5, 34h, 43h, 44h, 5Ch, 29h, 2Ah, 5Dh, 31h
  if ( a1a->VMOpcode != 0x31 || v230 != 1 )
  {
    v210 = &a1a->First.About_Lval_Byte_Word_Dword;
    a1a->First.ModRM_mod__Or__Size = 4;
    *v210 = v230;
    v210[3] = 0;                  // First.ModRM_Reg_Or_SIB_index
    if ( (unsigned __int16)(a1a->VMOpcode - 4) < 2u// 1、VmpOpcode==4 add系列  VmpOpcde==5 xor系列
      || a1a->VMOpcode == 0x34  // sub系列
      || (unsigned __int16)(a1a->VMOpcode - 0x43) < 2u )// VmpOpcode==43(rol)、44(rot)
    {                             // 这里面都是双操作数的
      v209 = &a1a->Second.About_Lval_Byte_Word_Dword;
      a1a->Second.ModRM_mod__Or__Size = 2;
      if ( (unsigned __int16)(a1a->VMOpcode - 0x43) >= 2u )// 43h(rol), 44h(rot)需要两个操作数
      {
        *v209 = v230;
        v209[0xF] = v230;
      }
      else                        // 4(add)、5(xor)、34(sub)系列
      {
        *v209 = 0;
        v209[0xF] = 0;
      }
      v82 = TCollection::GetItem_9(v81, Size);
      v83 = v209;
      *(_DWORD *)(v209 + 7) = *(_DWORD *)(v82 + 0x10);
      v81 = *(_DWORD *)(v82 + 0x14);
      *(_DWORD *)(v83 + 0xB) = v81;
    }
  }
  else                            // 0x31=bswap            单操作数
  {
    a1a->VMOpcode = 0x38;
    v212 = &a1a->First.About_Lval_Byte_Word_Dword;
    a1a->First.ModRM_mod__Or__Size = 4;
    *v212 = 0:
      }
      else                              // 0x31=bswap              单操作数
      {
        a1a->VMOpcode = 0x38;
        v212 = &a1a->First.About_Lval_Byte_Word_Dword;
        a1a->First.ModRM_mod__Or__Size = 4;
        *v212 = 0;
        v212[3] = 0;
        v211 = &a1a->Second.About_Lval_Byte_Word_Dword;
        a1a->Second.ModRM_mod__Or__Size = 0x104;
        *v211 = 0;
        v211[3] = 0;
      }
      SetDisasm(a1a, v81);          // VmpOpcode==5Ch(not), 29h(inc), 2Ah(dec), 5Dh(neg)单操作数的
      ++Size;
      --Number;
    }
    while ( Number );
  }
  break;
}
++v66;
--AddrNumber2;
}
while ( AddrNumber2 );
```

1、注意v277的值是struct_VmpOpcodePY_80~A0其中一组内容

2、根据随机值来执行不同的流程填充struct_DisassemblyFunction结构

3、针对第一种RandomWord_4EE0EC！=4执行流程主要是修改add aXX,BXX变成xor或则sub

```
00405064   033424          add esi,dword ptr ss:[esp]          kernel32.77523C45
00405067   8A06            mov al,byte ptr ds:[esi]
00405069   28D8            sub al,bl
0040506B   FEC0            inc al
0040506D   C0C0 05         rol al,0x5
00405070   F6D0            not al
00405072   2C B0           sub al,0xB0
00405074   F6D0            not al
00405076   34 7A           xor al,0x7A
00405078   8D76 01         lea esi,dword ptr ds:[esi+0x1]
0040507B   28C3            sub bl,al
0040507D   0FB6C0          movzx eax,al
00405080   FF3485 BB51400  push dword ptr ds:[eax*4+0x4051BB]
```

RandomWord_4EE0EC=0x34

4、针对第二种RandomWord_4EE0EC==4将第二句add bXX,aXX进行变形

```
00405062   89F3            mov ebx,esi                         HelloASM.<ModuleEntryP
00405064   033424          add esi,dword ptr ss:[esp]          kernel32.74558484
00405067   8A06            mov al,byte ptr ds:[esi]
00405069   00D8            add al,bl
0040506B   FEC0            inc al
0040506D   C0C0 05         rol al,0x5
00405070   F6D0            not al
00405072   2C B0           sub al,0xB0
00405074   F6D0            not al
00405076   34 7A           xor al,0x7A
00405078   8D76 01         lea esi,dword ptr ds:[esi+0x1]
0040507B   00C3            add bl,al
0040507D   0FB6C0          movzx eax,al
00405080   FF3485 BB51400  push dword ptr ds:[eax*4+0x4051BB]
```

8、1 SetDisassemblyFunction函数分析

如果说Vmp_Disassembly函数是将Opcode解析

那么SetDisassemblyFunction就是将解析后的Opcode再重新组装回去

```
result = a1;
a1->ReadHexLen = 0;
__linkproc__ DynArrayClear(&result->ReadHexAddress, (int)dword_47CE4C);
v2 = 0;
v3 = struct_Disasm;
do
{
  v4 = &result->First.About_Lval_Byte_Word_Dword + 0x17 * v2;
  qmemcpy(v3, v4, 0x14u);
  v4 += 0x14;
  v5 = (int)(v3 + 0x14);
  *(_WORD *)v5 = *(_WORD *)v4;
  *(_BYTE *)(v5 + 2) = v4[2];
  ++v2;
  v3 += 0x17;
}
while ( v2 != 3 );                              // 拷贝struct_Disasm一共3组，每组长度0x17个字节
v47 = 0;
if ( (_WORD)xdom::_18137 != *(_WORD *)&struct_Disasm[1] )// 判断寻址方式ModRM_mod__Or__Size
{
  if ( struct_Disasm[0] == 1 )                  // About_Lval_Byte_Word_Dword
  {
    Vmp_memcpy_ReadHexAddress(0, 0x66, 0, (int)&savedregs);
  }
  else if ( struct_Disasm[0] == 3
         && (unsigned __int16)(result->VMOpcode - 1) >= 2u
         && (unsigned __int16)(result->VMOpcode - 0xB) >= 2u
         && result->VMOpcode != 0x23
         && (unsigned __int16)(result->VMOpcode - 0x39) >= 2u
         && 0x20 != *(_WORD *)&struct_Disasm[1]
         && 0x20 != *(_WORD *)&struct_Disasm[0x18]
         && 0x40 != *(_WORD *)&struct_Disasm[1]
         && 0x40 != *(_WORD *)&struct_Disasm[0x18] )
  {
    v47 = 8;
  }
}
v6 = 0;
```

```
{
  if ( *(_BYTE *)(v7 + 1) & 8 )
  {
    if ( *(_BYTE *)(v7 + 2) & 2 )
    {
      if ( *(_BYTE *)(v7 + 4) > 7u )
      {
        *(_BYTE *)(v7 + 4) &= 7u;
        v47 |= 1u;
      }
      if ( *(_BYTE *)(v7 + 1) & 4 && *(_BYTE *)(v7 + 3) > 7u )
      {
        *(_BYTE *)(v7 + 3) &= 7u;
        v47 |= 2u;
      }
    }
    else if ( *(_BYTE *)(v7 + 1) & 4 && *(_BYTE *)(v7 + 3) > 7u )
    {
      *(_BYTE *)(v7 + 3) &= 7u;
      v47 |= 1u;
    }
  }
  else if ( *(_BYTE *)(v7 + 2) & 1 )
  {
    *(_WORD *)(v7 + 1) &= 0xFEFFu;
    *(_BYTE *)(v7 + 3) |= 4u;
  }
  else if ( (*(_BYTE *)(v7 + 1) & 4 || *(_BYTE *)(v7 + 1) & 0x20) && *(_BYTE *)(v7 + 3) > 7u )
  {
    *(_BYTE *)(v7 + 3) &= 7u;
    if ( v6 >= 2 || *(_WORD *)&struct_Disasm[23 * (1 - v6) + 1] & 0x2C )
      v47 |= 4u;
    else
      v47 |= 1u;
  }
  ++v6;
  v7 += 0x17;
}
while ( v6 != 3 );                              // 一共3组循环3次
if ( v47 )
```

判断是否存在前缀

```
  }
  while ( v6 != 3 );                            // 一共3组循环3次
  if ( v47 )
    Vmp_memcpy_ReadHexAddress(0, v47 | 0x40, 0, (int)&savedregs);
  v8 = 3;
  v9 = struct_Disasm;
  while ( !(v9[1] & 8) )
  {
    v9 += 0x17;
    if ( !--v8 )
      goto LABEL_50;
  }
  switch ( result->Segment_Override_Prefix )    // 段重载前缀
  {
    case 1:                                     // 26 --- ES register =1
      Vmp_memcpy_ReadHexAddress(0, 0x26, 0, (int)&savedregs);
      break;
    case 3:                                     // 36 --- SS register =3
      Vmp_memcpy_ReadHexAddress(0, 0x36, 0, (int)&savedregs);
      break;
    case 2:                                     // 2E --- CS register =2
      Vmp_memcpy_ReadHexAddress(0, 0x2E, 0, (int)&savedregs);
      break;
    case 4:                                     // 3E --- DS register =4
      Vmp_memcpy_ReadHexAddress(0, 0x3E, 0, (int)&savedregs);
      break;
    case 5:                                     // 64 --- FS register =5
      Vmp_memcpy_ReadHexAddress(0, 0x64, 0, (int)&savedregs);
      break;
    case 6:                                     // 65 --- GS register =6
      Vmp_memcpy_ReadHexAddress(0, 0x65, 0, (int)&savedregs);
      break;
    default:
      break;
  }
```

根据前面Opcode选择读取对应主操作码，假设该Opcode操作码需要依赖Mod寻址就执行sub_49DFD0

```
  v10 = result->VMOpcode;
  VMOpcode = result->VMOpcode;
  if ( VMOpcode > 0x31 )
  {
    if ( VMOpcode >= 0x3D )
    {
      if ( VMOpcode >= 0x5C )
      {
        v22 = VMOpcode - 0x5C;
        v13 = v22 < 2;
        v23 = v22 - 2;
        if ( v13 )
        {
          Vmp_memcpy_ReadHexAddress(0, (struct_Disasm[0] != 0) | 0xF6, 0, (int)&savedregs);
          if ( result->VMOpcode == 0x5C )       // 0x5C=not                  单操作数
            v46 = 0x10;
          else
            v46 = 0x18;
          sub_49DFD0(0, v46, (int)&savedregs);
        }
        else
        {
          v24 = v23 - 8;
          if ( v24 )
          {
            if ( v24 == 1 )
              Vmp_memcpy_ReadHexAddress(0, 0x61, 0, (int)&savedregs);
          }
          else
          {
            Vmp_memcpy_ReadHexAddress(0, 0x60, 0, (int)&savedregs);
          }
        }
      }
      else
```

根据ModRm_Mod寻址方式判断，从而构造不同的指令

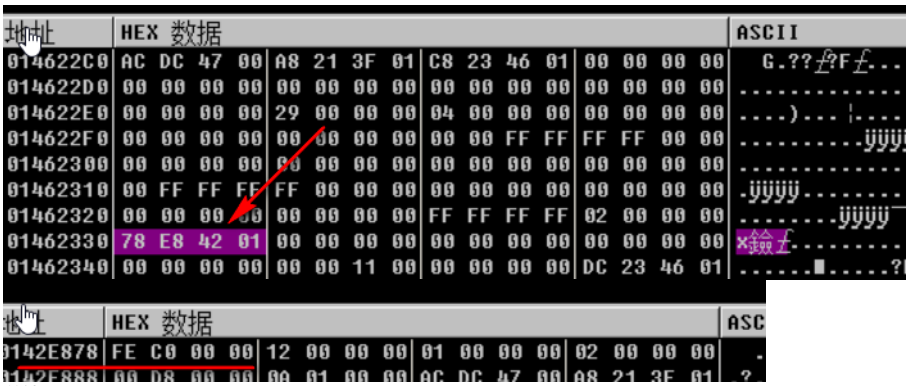
```
// 函数作用：
// 1、如果要执行这个函数说明：该Opcode需要ModRM进行补充的
signed __int16 __usercall sub_49DFD0@<ax>(int a1@<eax>, char a2@<dl>, int a3)
{
  char v3; // bl@1
  int v4; // edi@1
  struct_Disasm *v5; // esi@1
  char v6; // bl@9
  signed __int16 result; // ax@13
  char a3a; // [sp+Ch] [bp-18h]@9
  char v9; // [sp+10h] [bp-14h]@9

  v3 = a2;
  v4 = a1;
  v5 = (struct_Disasm *)(a3 + 0x17 * a1 - 0x49);
  if ( v5->ModRM_mod__Or__Size & 8 )            // 提供寻址模式: 11 = register  !11 = memory
  {
    if ( v5->ModRM_mod__Or__Size & 2 )
    {
      if ( v5->ModRM_mod__Or__Size & 0x204 )
      {
        if ( v5->Lavl_Btye_Word_Dword )
          v3 = a2 | 0x80;
        else
          v3 = a2 | 0x40;
      }
      else
      {
        v3 = a2 | 5;
      }
    }
    if ( HIBYTE(v5->ModRM_mod__Or__Size) & 2 )
    {
      Vmp_memcpy_ReadHexAddress(0, v3 | 4, 0, a3);
      v6 = v5->SIB_base | (v5->SIB_scale << 6);
      Vmp_GetDiSasmData(*(struct_DisassemblyFunction **)(a3 - 4), v4, &a3a);
      if ( v9 == 4 )
      {
        v6 |= 0x20u;
      }
```

举例子说明：

VmpOpcode=0x29

```
    if ( v16 )
    {
      if ( (unsigned int)(v16 - 2) < 2 )
      {
        if ( 4 != *(_WORD *)&struct_Disasm[1] || result->Magic == 3 || result->Magic != struct_Disasm[0] )
        {
          Vmp_memcpy_ReadHexAddress(0, (struct_Disasm[0] != 0) | 0xFE, 0, (int)&savedregs);// 注意：FE是byte  FF是dword
          if ( result->VMOpcode == 0x29 )
            v46 = 0;
          else
            v46 = 8;
          sub_49DFD0(0, v46, (int)&savedregs);
        }
        else
        {
          if ( result->VMOpcode == 0x29 )
            v46 = 0x40;
          else
            v46 = 0x48;
          Vmp_memcpy_ReadHexAddress(0, struct_Disasm[3] | v46, 0, (int)&savedregs);
        }
      }
```

执行完毕后，指令就构造完毕：FE C0 对应汇编代码：inc al



总结:

1、设置struct_DisassemblyFunction的内容

2、用struct_DisassemblyFunction 提供的Opcode信息还原回一条完整的汇编指令

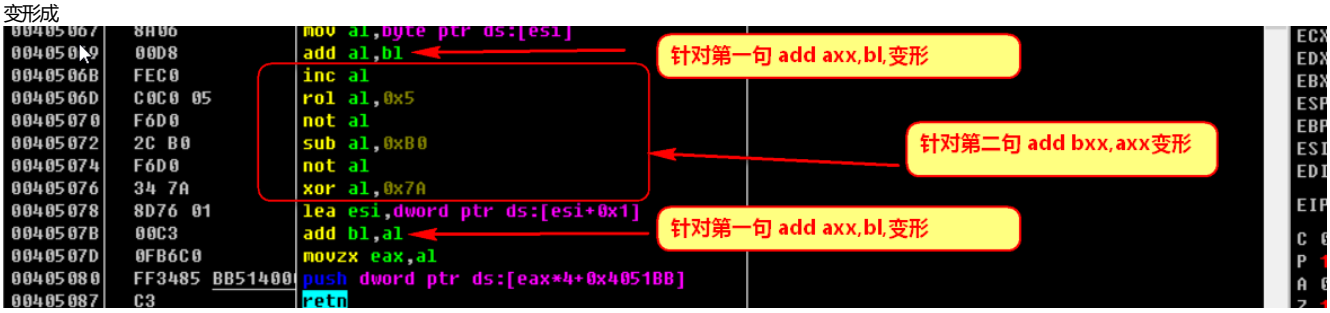
8、2 总结:

0、第一次执行才使用struct_VmpOpcodePY_80，非第一次都是使用struct_VmpOpcodePY_84~90或则struct_VmpOpcodePY_94~A0

1、针对壳模板的add指令进行修改变形处理

符合条件的如下:

变形成

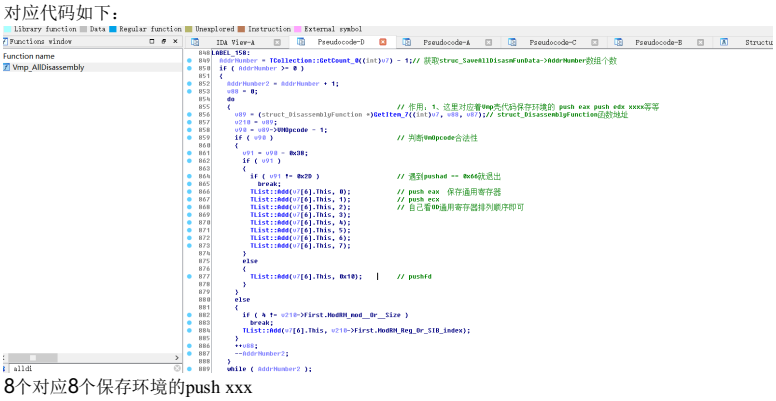


9、保存寄存器环境的代码，注意后面会随机乱序的

原因:

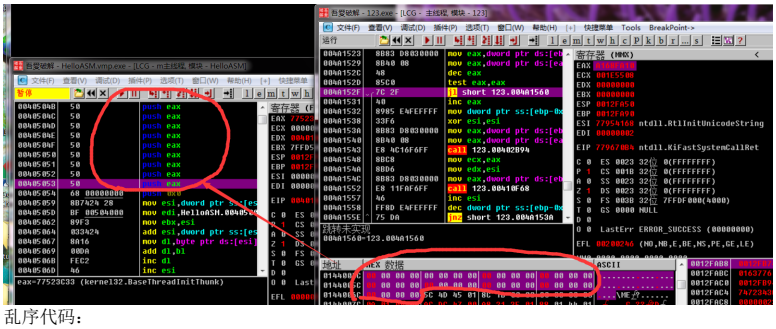
为什么壳起始代码 push环境每次都是乱序的？如何实现的？

对应代码如下:



8个对应8个保存环境的push xxx

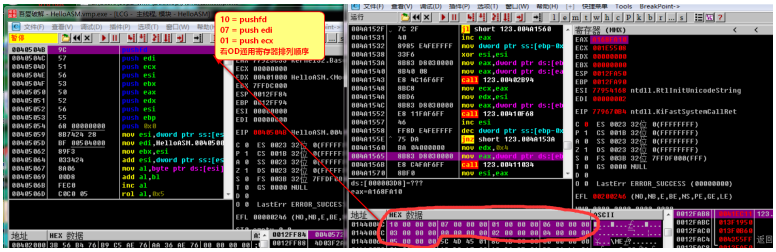
笔者为了方便测试所以全写成0，可见0 == push eax跟OD通用寄存器对应顺序一样



乱序代码：

只要打乱数组存放顺序就可以实现乱序了

```
  }
  if ( *(_DWORD *)(v7[6].This + 8) - 1 >= 0 )
  {
    AddrNumber2 = *(_DWORD *)(v7[6].This + 8);  // 获取数组大小（push XXX  寄存器环境的9行代码）
    v92 = 0;
    do
    {
      Max = *(_DWORD *)(v7[6].This + 8);
      v94 = __linkproc__ RandInt();             // 随机数最大值就是数组个数
      TList::Exchange(v7[6].This, v92++, v94);  // 交换
      --AddrNumber2;
    }
    while ( AddrNumber2 );                      // 打乱数组保存顺序，乱序保存寄存器环境的9行代码
  }
  v95 = TList::IndexOf(v7[6].This, 4);
```



它们保存在：

struct_VmpOpcode->struc_PushRegister指向的结构体

10、找到lods byte ptr ds:[esi]并保存起来

```
  v237 = 0;
  AddrNumber_2 = TCollection::GetCount_0((int)v7) - 1;// 获取struc_SaveAllDisasmFunData->AddrNumber个数（基础版）
  if ( AddrNumber_2 >= 0 )
  {
    AddrNumber2 = AddrNumber_2 + 1;
    v103 = 0;
    while ( *(_WORD *)(GetItem_7((int)v7, v103, v102) + 0x24) != 0x36 )// 判断VMOpcode == lods byte ptr ds:[esi](0x36)
    {
      ++v103;
      if ( !--AddrNumber2 )
        goto LABEL_186;
    }
    v237 = (struct_DisassemblyFunction *)GetItem_7((int)v7, v103, v102);// v239指向找到的struct_DisassemblyFunction结构体
  }
LABEL_186:
```

10、1 构造出PushRegister那几条指令

1、将不符合条件的全部删除，直到找到push 0xFACE0002这条为止

2、因为Vmp保存寄存器环境代码是随机性的，原始壳模板的是固定的所以要替换掉

```
do
{
  if ( Number_1 == -1 )                         // 默认第一次执行，替换壳模板pushfd pushad代码
  {
    v104 = 0;
    v233 = 0;
    v105 = TCollection::GetCount_0((int)v7) - 1;// struc_SaveAllDisasmFunData->AddrNumber(基本版)
    if ( v105 >= 0 )                            // 删除掉不符合条件的数组内容
    {
      AddrNumber2 = v105 + 1;
      v107 = 0;
      do                                        // 查找push 0xFACE0002 这一句重定位Opcode，因为pushfd pushad后面必然是push 0xFACE0002
      {
        v104 = v107;
        while ( 1 )
        {
          v108 = *(_WORD *)(GetItem_7((int)v7, v107, v106) + 0x24) - 0x39;// 过滤掉pushfd
          if ( v108 )
          {
            if ( v108 != 0x2D                   // 过滤掉pushad
              && (*(_WORD *)(GetItem_7((int)v7, v107, v109) + 0x24) != 1// VmOpcode == push XXXXX(1)
               || 4 != *(_WORD *)(GetItem_7((int)v7, v107, v106) + 0x28)) )
            {
              break;                            // 找到就退出
            }
          }
          v233 = 1;
          (*(void (__fastcall **)(_DWORD, int))(v7->This + 0xC))(v7->This, v107);// Delete(Index: Integer);    过程。删除列表中对应索引的项目。
        }
        if ( v233 )
          break;
        ++v107;
        --AddrNumber2;
      }
      while ( AddrNumber2 );
    }
```

3、因为 pushfd pushad模板后面必然是push 0xFACE0002

3、根据寄存器不同而设置不同的VmpOpcode，进行构造填充struct_DisassemblyFunction结构

4、返回：lods byte ptr ds:[esi]在数组第几个元素

```
    AddrNumber2 = *(_DWORD *)(v7->struc_PushRegister + 8);
    v111 = 0;
    do                                  // 将构造好的struct_DisassemblyFunction数据保存到struc_SaveAllDisasmFunData->Address数组里面，保存构造好通用寄存器代码
    {
      v238 = TList::Get(v7->struc_PushRegister, v111);
      if ( !pushXX_Or_pushad_Flag || !v238 || v238 == 0x10 )
      {
        v207 = (struct_DisassemblyFunction *)(*(int (**)(void))(v7->This + 0x14))();// Add
        v207->Magic = v230;
        if ( v238 == 0x10 )             // pushfd == 0x10
        {
          v207->VMOpcode = 0x39;
          v207->First.About_Lval_Byte_Word_Dword = v207->Magic;
        }
        else if ( !pushXX_Or_pushad_Flag || v238 )// push 通用寄存器
        {
          if ( v238 == 4 )              // push esp (4)
            v238 = __linkproc__ RandInt();  // 随机数范围：0~8
          v207->VMOpcode = 1;
          v206 = &v207->First.About_Lval_Byte_Word_Dword;
          v207->First.ModRM_mod__Or__Size = 4;
          *v206 = v238;
          v206[3] = v238;
        }
        else
        {
          v207->VMOpcode = 0x66;
          v207->First.About_Lval_Byte_Word_Dword = v230;
        }
        SetDisassemblyFunction(v207, v112);
        v113 = TCollection::GetCount_0((int)v7);
        TList::Move(v7, v113 - 1, v104++);  // 将最后一个（当前new出来的）移动到前面
      }
      ++v111;
      --AddrNumber2;
    }
    while ( AddrNumber2 );
  }
  Size = TList::IndexOf((int)v237, v110);   // 返回：lods byte ptr ds:[esi]在数组第几个元素
```

OD最终效果图如下：

```
0040504B   9C              pushfd
0040504C   57              push edi                    HelloASM.<ModuleEntryPoint>
0040504D   51              push ecx                    HelloASM.<ModuleEntryPoint>
0040504E   56              push esi                    HelloASM.<ModuleEntryPoint>
0040504F   53              push ebx
00405050   50              push eax
00405051   52              push edx                    HelloASM.<ModuleEntryPoint>
00405052   56              push esi                    HelloASM.<ModuleEntryPoint>
00405053   55              push ebp
```

OD数组struc_SaveAllDisasmFunData->ArrayAddress排列顺序如下：

```
004A185D   8BCF            mov ecx,edi
004A185F   8BC3            mov eax,ebx
004A1861   E8 8281FEFF     call 123.004899E8
004A1866   47              inc edi
004A1867   46              inc esi
004A1868   FF8D E4FEFFFF   dec dword ptr ss:[ebp-0x11C]
004A186E   0F85 C7FEFFFF   jnz 123.004A173B
004A1874   8B85 08FFFFFF   mov eax,dword ptr ss:[ebp-0xF8]
004A187A   E8 C579FEFF     call 123.00489244
004A187F   8985 18FFFFFF   mov dword ptr ss:[ebp-0xE8],eax
004A1885   E9 BE020000     jmp 123.004A1B48
004A188A   8B85 14FFFFFF   mov eax,dword ptr ss:[ebp-0xEC]
004A1890   83BC83 A4000000 cmp dword ptr ds:[ebx+eax*4+0xA4],0x0
004A1898   0F84 E10D0000   je 123.004A267F
004A189E   8B85 40FEFFFF   mov eax,dword ptr ss:[ebp-0x1C0]   123.004EDA2A
004A18A4   66:8B00         mov ax,word ptr ds:[eax]
004A18A7   83C0 F8         add eax,
堆栈 ss:[0012FBA4]=01411E54
eax=01453FCC

EBP 0012FC9C
ESI 00000009
EDI 00000009
EIP 004A1874 1
C 0  ES 0023 3
P 1  CS 001B 3
A 0  SS 0023 3
Z 1  DS 0023 3
S 0  FS 003B 3
T 0  GS 0000 N
D 0
O 0  LastErr E
EFL 00200246 (
MM0 0000 0000
MM1 0000 0000
MM2 0000 0000
MM3 0000 0000
MM4 0000 0000

那9条保存寄存器环境的指令

地址      HEX 数据                                          ASCII
01453FCC  80 00 44 01 E8 01 44 01 50 03 44 01 B8 04 44 01   ■.D♂?D♂P ♥D♂?D♂
01453FDC  20 06 44 01 88 07 44 01 F0 08 44 01 58 0A 44 01    ♠D♂?D♂?D♂X.D♂
01453FEC  C0 0B 44 01 34 17 41 01 A0 18 41 01 08 1A 41 01   ?D♂4↨A♂?A♂■←A♂
01453FFC  98 1B 41 01 EC 1C 41 01 54 1E 41 01 BC 1F 41 01   ?A♂?A♂T▲A♂?A♂
```

10、2 现在该处理lods byte ptr ds:[esi]指令了

0、lods byte ptr ds:[esi]指令介绍：

指令规定源操作数为(DS:SI)，目的操作数隐含为AL（字节）或AX（字）寄存器。三种指令都用于将目的操作数的内容取到AL或AX寄存器，字节还是字操作由寻址方式确定，并根据寻址方式自动修改SI的内容。

一句指令相当于以下两句：

mov al,[esi]

inc esi

1、初始化v245跟v246数组，具体用处待定

```
v126 = 8;
v127 = v245;                            // 都是标志位
v128 = v246;                            // 都是标志位
do
{                                       // 初始化？？？作用待定
  *v127 = -1;
  *v128++ = 1;
  ++v127;
  --v126;
}
while ( v126 );
v246[4] = 0;
v246[6] = 0;
if ( *(_BYTE *)(v7->_prev_node + 0x48) & 8 )// 保护等级默认为8，不成立就错误
  v246[3] = 0;                          // 必须成立
if ( Number_1 > -1 )                    // Number_1默认是-1的(第一次)，判断非第一次进入
{
  v202 = (int *)((char *)v181 - 2);
  if ( (unsigned __int16)(*(_WORD *)v181 - 1) < 2u )
  {
    if ( *((_BYTE *)v202 + 4) == 2 )
      v246[0] = 0;
  }
  else if ( (unsigned __int16)(*(_WORD *)v181 - 0x3D) < 2u )
  {
    v246[1] = 0;
  }
}
```

2、找到处理的地方Vmp == 0x36

3、struct_DisassemblyFunction结构重新赋值

```
if ( v133->VMOpcode == 0x36 )          // losd = VmOpcode0x36
{
  v230 = v133->First.About_Lval_Byte_Word_Dword;
  v229 = v133->Second.About_Lval_Byte_Word_Dword;// 1、win32普通执行文件值=1，返回2，作用是区别长度
  if ( Number_1 != -1 && __linkproc__ RandInt() != 1 )// 判断是否非第一次 && 随机数不等于1
  {
    v231 = v133->Magic;
    if ( v231 == 3 && v230 == 2 )
      v231 = 2;
    if ( v230 == v231 )
      v133->VMOpcode = 3;
    else
      v133->VMOpcode = 0x27;
    v195 = &v133->First.About_Lval_Byte_Word_Dword;
    v133->First.ModRM_mod__Or__Size = 4;
    *v195 = v231;
    v195[3] = 0;
    v194 = &v133->Second.About_Lval_Byte_Word_Dword;
    v133->Second.ModRM_mod__Or__Size = 0xC;
    *v194 = v230;
    v194[0x12] = v229;
    v194[3] = 6;
  }
  else
  {
    v246[0] = Number_1 == -1;           // 判断是否第一次，如果是赋值为1
    v133->VMOpcode = 3;
    v197 = &v133->First.About_Lval_Byte_Word_Dword;
    v133->First.ModRM_mod__Or__Size = 4;
    *v197 = v230;
    v197[3] = 0;
    v196 = &v133->Second.About_Lval_Byte_Word_Dword;
    v133->Second.ModRM_mod__Or__Size = 0xC;
    *v196 = v230;
    v196[0x12] = v229;                  // About_RegType_8_16_32 db ?             ; 读取长度
    v196[3] = 6;
  }
  v232 = 1;
  v145 = TCollection::GetCount_0((int)v7) - 1;
  v68 = __OFSUB__(v145, v129 + 1);
  v147 = v145 - (v129 + 1);
  if ( !((v147 < 0) ^ v68) )
```

4、找到该struct_DisassemblyFunction所在的数组位置

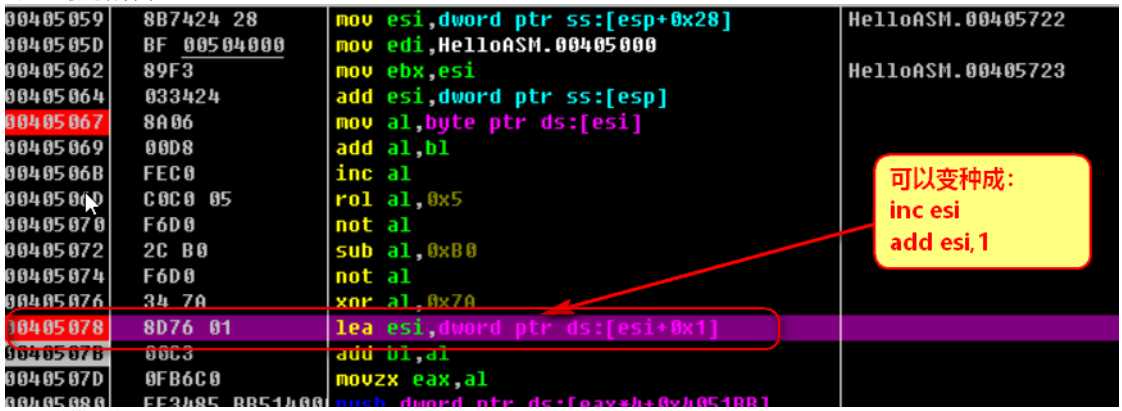
5、并重新new个新的struct_DisassemblyFunction

6、根据随机数构造命令：INC、Add、lea，实际上只要实现esi+1都行

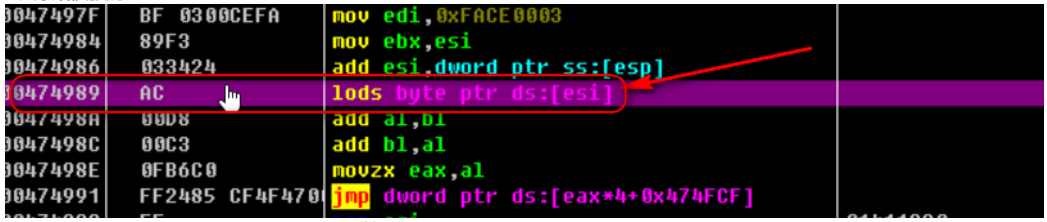
```
    v68 = __OFSUB__(v145, v129 + 1);
    v147 = v145 - (v129 + 1);
    if ( !((v147 < 0) ^ v68) )
    {
      Number = v147 + 1;
      i = v129 + 1;
      do
      {
        v148 = *(_WORD *)(GetItem_7((int)v7, i, v146) + 0x24) - 8;
        v16 = v148 < 2u;
        v149 = v148 - 2;
        if ( v16 )                      // VMOpcode:8 9 =ret C=jmp
          break;
        if ( v149 == 2 )
          break;
        ++i;
        --Number;
      }
      while ( Number );
    }
    v193 = (struct_DisassemblyFunction *)(*(int (**)(void))(v7->This + 0x14))();// Add(Item: Pointer): Integer;    函数。用来向列表中添加指针。
    v192 = &v193->First.About_Lval_Byte_Word_Dword;
    v193->First.ModRM_mod__Or__Size = 4;
    *v192 = v229;
    v192[3] = 6;
    if ( v230 || __linkproc__ RandInt() != 1 )// 随机数范围0~2
    {
      if ( __linkproc__ RandInt() == 1 )
        v193->VMOpcode = 4;             // Add系列 ->add esi,1
      else
        v193->VMOpcode = 7;             // LEA Gv,M ->lea esi,dword ptr ds:[esi+0x1]
      v191 = &v193->Second;
      if ( v193->VMOpcode == 7 )
      {
        v191->ModRM_mod__Or__Size = 0xE;
        v191->About_RegType_8_16_32 = v229;
        v191->ModRM_Reg_Or_SIB_index_Or_ModRM_rm = 6;
      }
      else
      {
        v191->ModRM_mod__Or__Size = 2;
      }
```

```
v193 = (struct_DisassemblyFunction *)(*(int (**)(void))(v7->This + 0x14))();// Add(Item: Pointer): Integer;    函数。用来向列表中添加指针
v192 = &v193->First.About_Lval_Byte_Word_Dword;
v193->First.ModRM_mod__Or__Size = 4;
*v192 = v229;
v192[3] = 6;
if ( v230 || __linkproc__ RandInt() != 1 )// 随机数范围0~2
{
  if ( __linkproc__ RandInt() == 1 )
    v193->VMOpcode = 4;             // Add系列 ->add esi,1
  else
    v193->VMOpcode = 7;             // LEA Gv,M ->lea esi,dword ptr ds:[esi+0x1]
  v191 = &v193->Second;
  if ( v193->VMOpcode == 7 )
  {
    v191->ModRM_mod__Or__Size = 0xE;
    v191->About_RegType_8_16_32 = v229;
    v191->ModRM_Reg_Or_SIB_index_Or_ModRM_rm = 6;
  }
  else
  {
    v191->ModRM_mod__Or__Size = 2;
  }
  v191->About_Lval_Byte_Word_Dword = v229;
  v150 = v191;
  v191->LODWORD_RestHex_Lval_Displacement_Immediate = (unsigned __int8)Global_Size[(unsigned __int8)v230];
  v150->HIDWORD_RestHex_Lval_Displacement_Immediate = 0;
  v191->Lavl_Btye_Word_Dword = 0;
}
else
{
  v193->Magic = v133->Magic;
  v193->VMOpcode = 0x29;            // Inc系类 ->inc esi
}
SetDisassemblyFunction(v193, (int)v150);
v151 = v129 + 1 + __linkproc__ RandInt();
v152 = TCollection::GetCount_0((int)v7);
TList::Move(v7, v152 - 1, v151);
}
```

7、OD最终效果图:



8、原始模板的

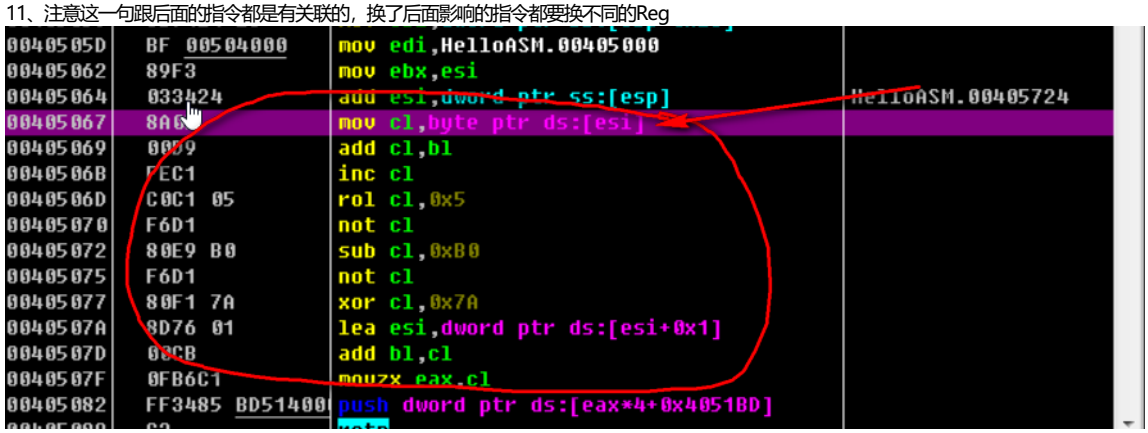


9、前面的构造出了inc esi(add lea)，那么还差一句mov al,[esi]

10、注意v158 = GetRandInt0123((int)&savedregs);这一句是随机获取0~3，也就是Reg：0=al、1=cl、2=dl、3=bl

```
else if ( v246[v184->ModRM_Reg_Or_SIB_index_Or_ModRM_rm] )
{
  v158 = GetRandInt0123((int)&savedregs);
  v102 = v159;
  v238 = v158;
  v245[v184->ModRM_Reg_Or_SIB_index_Or_ModRM_rm] = v158;
  v239 = v184->ModRM_Reg_Or_SIB_index_Or_ModRM_rm;
  v184->ModRM_Reg_Or_SIB_index_Or_ModRM_rm = v238;
  if ( v239 != v238 )
  {
    v232 = 1;
    if ( Number_1 > -1 && !v184->About_Lval_Byte_Word_Dword && !v239 )
    {
      if ( __linkproc__ RandInt() == 1 )
      {
        v184->About_Lval_Byte_Word_Dword = 1;
        v133->VMOpcode = 0x27;  // mov DH/R14L,lb
      }
      else
      {
        a1a = (struct_DisassemblyFunction *)(*(int (**)(void))(v7->This + 20))();
        v160 = v129 + __linkproc__ RandInt();
        v161 = TCollection::GetCount_0((int)v7);
        TList::Move(v7, v161 - 1, v160);
        a1a->word86 &= 0xFFFEu;
        if ( __linkproc__ RandInt() == 1 )
          a1a->VMOpcode = 5;    // xor
        else
          a1a->VMOpcode = 0x34; // sub系列
        v183 = &a1a->First.About_Lval_Byte_Word_Dword;
        a1a->First.ModRM_mod__Or__Size = 0x104;
        *v183 = 0;
        v183[3] = v238;
        v182 = &a1a->Second.About_Lval_Byte_Word_Dword;
        a1a->Second.ModRM_mod__Or__Size = 0x104;
        *v182 = 0;
        v182[3] = v238;
        SetDisassemblyFunction(a1a, v162);
      }
    }
```

11、注意这一句跟后面的指令都是有关联的，换了后面影响的指令都要换不同的Reg



10、3 处理Jmp Ret指令

1、通过随机数决定jmp ret指令是变换成:

随机数==2

lea exx,dword ptr ds:[eax*4+0x474FCF]

jmp [exx]

随机数==1

push dword ptr ds:[eax*4+0x4051BB]

retn

2、注意v238 = GetRandInt0123((int)&savedregs);这一句，表示它的Mod.Reg寄存器是随机的0~3

```
v116 = v134 < 1;
v135 = v134 - 1;
if ( !v116 )                            // 通过随机数决定是否进行变换，随机数是0就不变换
{
  if ( v135 )                           // 随机数是2
  {
    v238 = GetRandInt0123((int)&savedregs);
    v133->VMOpcode = 7;                 // LEA Gv,M ->
    v137 = v129;
    v138 = v133;
    v139 = &v133->First.About_Lval_Byte_Word_Dword;
    v140 = &v133->Second.About_Lval_Byte_Word_Dword;
    qmemcpy(v140, v139, 0x14u);
    v139 += 0x14;
    v140 += 0x14;
    *(_WORD *)v140 = *(_WORD *)v139;
    v140[2] = v139[2];
    v129 = v137;
    v200 = &v138->First.About_Lval_Byte_Word_Dword;
    v138->First.ModRM_mod__Or__Size = 4;
    *v200 = v138->Magic;
    v200[3] = v238;                     // ModRM_Reg_Or_SIB_index_Or_ModRM_rm
    SetDisassemblyFunction(v138, 0);
    v199 = (struct_DisassemblyFunction *)(*(int (**)(void))(v7->This + 0x14))();// Add添加数组元素
    v199->VMOpcode = 0xC;               // Jmp
    v198 = &v199->First.About_Lval_Byte_Word_Dword;
    v199->First.ModRM_mod__Or__Size = 0xC;
    *v198 = v199->Magic;
    v198[3] = v238;
    SetDisassemblyFunction(v199, v141);
  }
  else                                  // 随机数是1
  {
    v133->VMOpcode = 1;                 // push   ->例子push dword ptr ds:[eax*4+0x4051BB]
    SetDisassemblyFunction(v133, v102);
    v201 = (struct_DisassemblyFunction *)(*(int (**)(void))(v7->This + 20))();
    v201->VMOpcode = 9;                 // retn   ->例子：retn
    SetDisassemblyFunction(v201, v136);
  }
  v142 = TCollection::GetCount_0((int)v7);
  TList::Move(v7, v142 - 1, v129 + 1);
}
```

10、4 处理Handle里面的Vmp_Ret函数

0、跟前面一样，将popad复杂化，变成pop eax、pop ecx等等

1、ESI_Matching_Array[2] == VMOpcode,符合条件的是：Vmp_Ret指令（pop xx popad popfd这种）

,{ 0x06,0x01,0x09,0x00,0x00,0x02,0x00,0x00, }

//00474FCB 58 pop eax; 123.0047499B

//00474FCC 61 popad

//00474FCD 9D popfd

,{ 0x06,0x00,0x08,0x00,0x00,0x02,0x01,0x00, }

//00474FC7 58 pop eax; 123.0047499B

//00474FC8 61 popad

//00474FC9 9D popfd

2、将popad跟popfd删除，直到遍历到ret就退出

```
if ( (unsigned __int16)(*(_WORD *)v181 - 8) < 2u )// ESI_Matching_Array[2]  == VMOpcode,符合条件的是：Vmp_Ret指令（pop xx popad popfd这种）
{
  v114 = 0;
  v233 = 0;
  v115 = TList::IndexOf(*(_DWORD *)&v7->Esi_Addr[4 * Number_1], v102);
  v116 = TCollection::GetCount_0((int)v7) - 1;// 获取特殊版数组元素个数
  v68 = __OFSUB__(v116, v115);
  v118 = v116 - v115;
  if ( !((v118 < 0) ^ v68) )
  {
    AddrNumber2 = v118 + 1;
    do                                  // 直到ret就退出
    {
      v114 = v115;
      while ( 1 )
      {
        v121 = *(_WORD *)(GetItem_7((int)v7, v114, v117) + 0x24) - 0x3A;// 0x3A == popfd 0x3B == sahf
        if ( v121 )
        {
          if ( v121 != 0x2D             // popad == 0x67
            && (*(_WORD *)(GetItem_7((int)v7, v114, v117) + 0x24) != 2
             || 4 != *(_WORD *)(GetItem_7((int)v7, v114, v117) + 0x28)) )
          {
            break;
          }
        }
        if ( v233 )
        {
          (*(void (__fastcall **)(_DWORD, int))(v7->This + 0xC))(v7->This, v114);// 删除数组元素（struc_SaveAllDisasmFunData）
        }
        else
        {
          v205 = (struct_DisassemblyFunction *)GetItem_7((int)v7, v114, v117);
          v119 = __linkproc__ RandInt();
          v205->First.ModRM_Reg_Or_SIB_index_Or_ModRM_rm = v119;// Mod.Reg：0=al、1=cl、2=dl、3=bl
          SetDisassemblyFunction(v205, v120);
          ++v114;
        }
        v233 = 1;
```

```
)A18D3 1035
```

3、将前面v7->struc_PushRegister保存的寄存器递减方式存储，注意去掉Esp寄存器

```
v230 = GetSize_0(a2a);
v122 = *(_DWORD *)(v7->struc_PushRegister + 8) - 1;
if ( v122 >= 0 )
{
  do
  {
    v238 = TList::Get(v7->struc_PushRegister, v122);
    if ( !pushXX_Or_pushad_Flag || v238 == 7 || v238 == 0x10 )// 7 == Edi
    {
      v204 = (struct_DisassemblyFunction *)(*(int (**)(void))(v7->This + 0x14))();// ADD添加数组元素（struc_SaveAllDisasmFunData）
      v204->Magic = v230;
      if ( v238 == 0x10 )               // 10 == pushfd
      {
        v204->VMOpcode = 0x3A;
        v204->First.About_Lval_Byte_Word_Dword = v204->Magic;
      }
      else if ( pushXX_Or_pushad_Flag && v238 == 7 )
      {
        v204->VMOpcode = 0x67;
        v204->First.About_Lval_Byte_Word_Dword = v204->Magic;
      }
      else
      {
        if ( v238 == 4 )                // 4 == Esp需要重新Rand
        {
          do
          {
            v124 = __linkproc__ RandInt();
            v238 = TList::Get(v7->struc_PushRegister, v124);
          }
          while ( v238 == 0x10 );       // 10 == pushfd
        }
        v204->VMOpcode = 2;
        v203 = &v204->First.About_Lval_Byte_Word_Dword;
        v204->First.ModRM_mod__Or__Size = 4;
        *v203 = v230;
        v203[3] = v238;
      }
      SetDisassemblyFunction(v204, v123);
      v125 = TCollection::GetCount_0((int)v7);
      TList::Move(v7, v125 - 1, v114++);
    }
```

4、总结:

原始的:

pop eax

popad

popfd

ret

修改成:

pop eax

pop xx

pop xx

pop xx

xxxxx

ret

11、找到填充虚拟机上下文的Handle块

1、根据GetSize的返回值填充v223数组

```
v167 = 0x100;
v181 = RandIndexArray;
do
{
  *v181 = -1;
  ++v181;
  --v167;
}
while ( v167 );                         // 初始化dword_4EF974全局变量，长度0x100
v168 = 2;
v181 = (int *)v223;
do
{
  *(_BYTE *)v181 = 0;
  v181 = (int *)((char *)v181 + 1);
  --v168;
}
while ( v168 );                         // 初始化
v169 = GetSize_0(a2a);
v230 = v169;
v170 = v169 - 2;
if ( v170 )                             // 根据大小判断
{
  if ( v170 == 1 )
  {
    v223[__linkproc__ RandInt()] = 1;   // 结果随机数填充到数组下标0~1
    Number_1 = 0x18;
  }
  else
  {
    Number_1 = 0;
  }
}
else                                    // 标志字，ROM 映像（0107h），普通可执行文件（010Bh），如果是普通可执行文件结果 *(_BYTE *)(a1 + 9) = 1
{
  v171 = __linkproc__ RandInt();        // 随机数范围:0~3
  v172 = __ROL1__(0x10 << v171, 2);
  v223[__linkproc__ RandInt()] = v172;  // 结果随机数填充到数组下标0~1
  Number_1 = 0x10;
}
```

2、根据大小跟助记符再过滤一遍Handle块，将符合条件的下标保存起来

```
v173 = 0;
v181 = ESI_Matching_Array;
v174 = ESIResults;
do                                      // 找到填充VMcontext的代码，and al,0x3C这种,一共两句push跟popVMcontext环境的代码
{
  result = v181;
  if ( (unsigned __int16)(*((_WORD *)v181 + 1) - 1) < 2u// ESI_Matching_Array[2]  == VMOpcode
    && *((_BYTE *)v181 + 4) == 2
    && *((_BYTE *)v181 + 5) == v230          // ESI_Matching_Array+5  == Size  , 根据大小找不同的
    && !*((_BYTE *)v181 + 6) )
  {
    if ( Number_1 - 1 >= 0 )
    {
      AddrNumber2 = Number_1;
      i = 0;
      do
      {
        RandIndexArray[i++ * (unsigned __int8)Global_Size[(unsigned __int8)v230] | (unsigned __int8)v223[*((_WORD *)result + 1) == 1]] = v173;// 符合条件的下标值保存起来,
        --AddrNumber2;
      }
      while ( AddrNumber2 );
    }
    *v174 = 0;
  }
  ++v173;
  ++v174;
  v181 += 2;                            // 每次+8，偏移下一组
}
while ( v173 != 0xCC );                 // Esi数组大小就是0XCC
```

3、符合条件的有2处（填充虚拟机上下文的Handle块和还原真实堆栈的Handle块）：

{ 0x06,0x01,0x01,0x00,0x02,0x02,0x00,0x00, }

//0047499B　80E0 3C　　and al,0x3C

//0047499E　FF3407　　push dword ptr ds : [edi + eax]

,{ 0x06,0x01,0x02,0x00,0x02,0x02,0x00,0x00, }

//00474AC3　80E0 3C　　and al, 0x3C

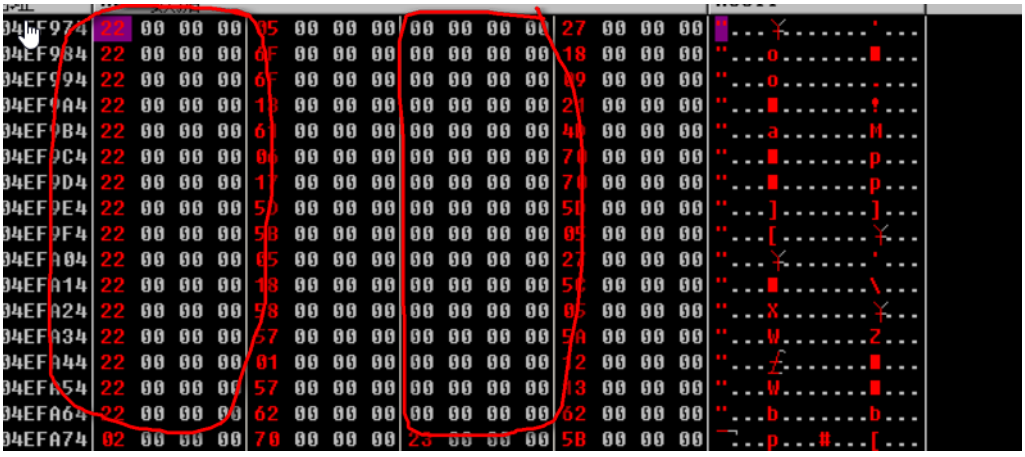
//00474AC6　8F0407　　pop dword ptr ds : [edi + eax]; 123.0047499B

4、未初始化的地方填充随机数

```
while ( v173 != 0xCC );                 // Esi数组大小就是0XCC
v176 = 0;
v177 = ESIResults;                      // 根据ESIResults[X]的值，非0就继续，随机填充RandIndexArray[X]数组，注意这里是填充RandIndexArray[X]===1的位置。不影响前面找到VMContext的
do
{
  if ( *v177 )
  {
    do
      i = __linkproc__ RandInt();       // 随机数范围，0x100
    while ( RandIndexArray[i] != -1 );
    result = (int *)i;
    RandIndexArray[i] = v176;
  }
  ++v176;
  ++v177;
}
while ( v176 != 0xCC );
v178 = 0x100;
v179 = RandIndexArray;                  // 继续随机数填充RandIndexArray[X]数组未赋值的数组下标,将剩下的未初始化的，全部用随机数填充
do
{
  if ( *v179 == -1 )
  {
    do
      i = __linkproc__ RandInt();
    while ( !ESIResults[i] );           // 遇到ESIResults[RandInt]==1的情况下才赋值
    result = (int *)i;
    *v179 = i;
  }
  ++v179;
  --v178;
}
while ( v178 );
return result;
}
```

5、效果图：

实际有用的只有22跟0，其他都是随机数填充的



12、总结

1、其实这部分代码都是针对部分特殊指令进行变形替换

例如：jmp可以变成 jmp+ret

例如：lods byte ptr ds:[esi]可以变成 mov aXX,[ESI] INC esi 等等

2、涉及重定位的代码还是没有修复

例如：

push 0xFACE0002

mov edi,0xFACE0003

jmp dword ptr ds:[eax*4+0x474FCF]

jmp short 00474984

3、找到填充虚拟机上下文的两个Handle块

后续介绍：

1、剩下重定位修复

2、伪代码构造