VMP学习笔记之壳的重定位修复（五）

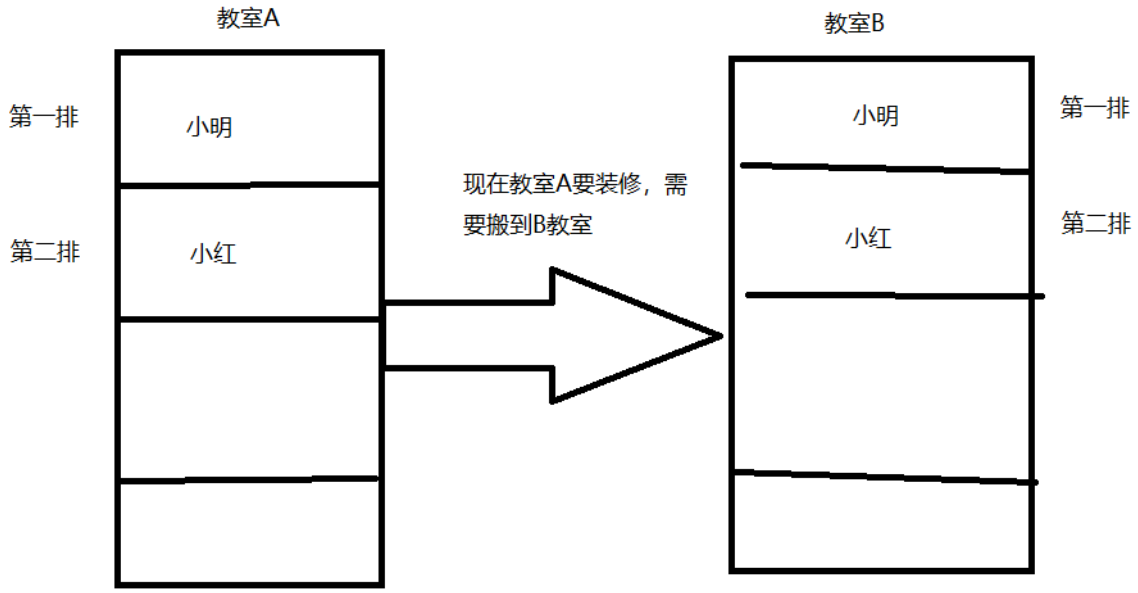
理论知识：

1、为什么需要重定位

因为特殊情况这块地址不给用了你要挪到另外一个地方，你本来是在哪里的挪到另外一个地方你还是在哪里

如下图所示：

小明在教室A是第一排，搬到教室B小明还是第一排



2、待修复数据如下：

push 0xFACE0002 --------->修复后是：0

mov edi,0xFACE0003 --------->修复后是：新区段首地址VMcontext

jmp dword ptr ds:[eax*4+0x474FCF] --------->修复后是：Handle块

jmp short 00474984 --------->修复后是：VMDispatcher

总结：

1、这些地址直接使用肯定会报错，因为这些地址是基于加壳机的，并非我们被加壳程序的。

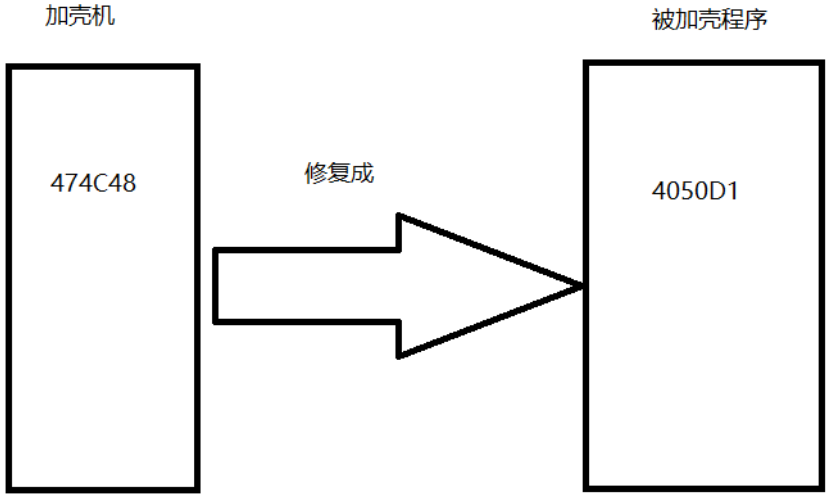
2、我们要将所有地址修复成基本我们被加壳程序的

3、我们要修改两处：struct_DisassemblyFunction->LODWORD_VMP_Address跟struct_DisassemblyFunction->ReadHexAddress

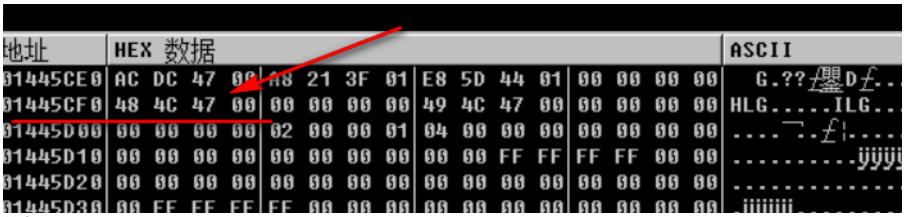
4、基本要修复都是有：Displacement、Immediate这些

3、如图所示

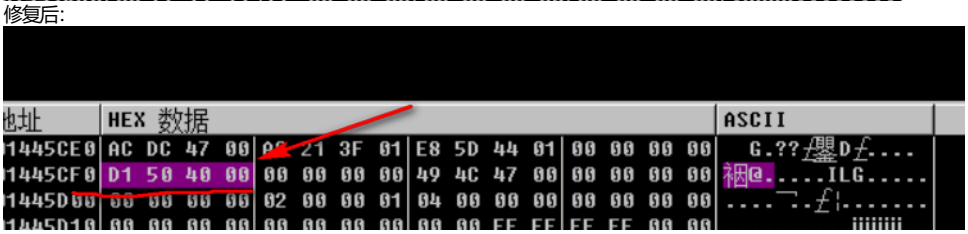
第一种：修复struct_DisassemblyFunction->LODWORD_VMP_Address



修复前：



修复后:



第二种：修复struct_DisassemblyFunction->ReadHexAddress

典型例子1：

00474991　FF2485 CF4F4700 jmp dword ptr ds:[eax*4+0x474FCF]这一句不修复肯定执行会错误

我们要修复这一句的Displacement，改成一个被加壳程序存在的地址就Ok了

例如：jmp dword ptr ds:[eax*4+0x4051BB]

典型例子2：

Jmp VMDispatcher

公式：jmp后的偏移量=目标地址-原来地址-5　　5是指令长度

正文：

1、筛选Esi_Addr[X]结构数据保存到ruct_VmpOpcodePY_50结构

将struct_DisassemblyFunction所在的数组下标保存到struc_69结构里

```
*(_DWORD *)(TList::Add(v3->struct_VmpOpcodePY_50) + 8) = 0;
v5 = 0;
do                                      // 循环作用：1、将Esi_Addr[X]的struct_DisassemblyFunction添加到保存起来
{
  v6 = *(_DWORD *)&v3->Esi_Addr[4 * v5];      // 里面保存的都是经过筛选的struct_DisassemblyFunction结构
  if ( v6 )
  {
    ArrayNumber = TList::IndexOf(v6, v4);     // 返回第几项找到的，返回该struct_DisassemblyFunction结构在struc_SaveAllDisasmFunData->ArrayNumber
    if ( DateTimeToStr_5(
           (struct_VmpOpcode *)v3->struct_VmpOpcodePY_50,
           ArrayNumber,
           (unsigned __int64)ArrayNumber >> 0x20) == -1 )// 列表中找不到就添加
      *(_DWORD *)(TList::Add(v3->struct_VmpOpcodePY_50) + 8) = ArrayNumber;// 保存数组下标
  }
  ++v5;
}
while ( v5 != 0xCC );                           // Esi大小
```

挑选jmp ret指令保存起来

```
ArrayNumber_1 = TCollection::GetCount_0(v3->struct_VmpOpcodePY_50) - 1;
if ( ArrayNumber_1 >= 0 )
{
  ArrayNumber_2 = ArrayNumber_1 + 1;
  v10 = 0;
  do
  {
    v83 = (struc_69 *)TCollection::GetItem_7(v3->struct_VmpOpcodePY_50, v10, v9);
    v11 = v83->ArrayNumber;
    v12 = TCollection::GetCount_0((int)v3);   // struc_SaveAllDisasmFunData->ArrayNumber（基本版）
    if ( v12 - 1 >= v11 )                     // 判断数组下标合法性
    {
      v13 = v12 - 1 - v11 + 1;
      while ( 1 )
      {
        VMOpcode = *(_WORD *)(GetItem_7((int)v3, v11, v9) + 0x24) - 8;
        v15 = VMOpcode < 2u;
        v16 = VMOpcode - 2;
        if ( v15 || v16 == 2 )                // VMOpcode:8 9 =ret C=jmp
          break;
        ++v11;
        if ( !--v13 )
          goto LABEL_14;
      }
      v83->FilterArrayNumber = v11;           // 符合条件VMOpcode保存起来
    }
LABEL_14:
    ++v10;
    --ArrayNumber_2;
  }
  while ( ArrayNumber_2 );                    // 找到struct_DisassemblyFunction->VMOpcode值是：8 9 =ret C=jmp，然后将该结构所在的数组下标保存起来
}
```

struc_69结构定义如下:

```
00000000 struc_69        struc ; (sizeof=0x34, mappedto_353)
00000000 This            dd ?
00000004 _prev_node      dd ?                   ; 指向struct_VmpOpcodePY_50
00000008 ArrayNumber     dd ?                   ; 返回第几项找到该struct_DisassemblyFunction结构在struc_SaveAllDisasmFunData->ArrayNumber
0000000C FilterArrayNumber dd ?                 ; 找到struct_DisassemblyFunction->VMOpcode值是：8 9 =ret C=jmp，然后将该结构所在的数组下标保存起来
00000010 field_10        dd ?
00000014 field_14        dd ?
00000018 field_18        dd ?
0000001C field_1C        dd ?
00000020 field_20        dd ?
00000024 field_24        dd ?
00000028 field_28        dd ?
0000002C field_2C        dd ?
00000030 field_30        dd ?
00000034 struc_69        ends
00000034
```

总结:

1、struc_69结构

struc_69->ArrayNumber　　== Handle块头

struc_69->FilterArrayNumber == Handle块结束地址，也就是jmp或则ret

2、找到第一组Handle块在数组的元素下标，作为结尾标识

符合条件的是：解析JmpAddr跳转表，填充的VMOpcode都是0x23

```
  v17 = TCollection::GetCount_0((int)v3) - 1;   // struc_SaveAllDisasmFunData->ArrayNumber（基本版）
if ( v17 >= 0 )                                 // 功能：1、找到解析JmpAddr跳转表的解析结构，并找到该结构在数组的第几个下标，取出并保存起来
{
  ArrayNumber_2a = v17 + 1;
  v19 = 0;
  while ( 1 )
  {
    v20 = (struct_DisassemblyFunction *)GetItem_7((int)v3, v19, v18);
    v21 = v20;
    v22 = (struct_VmFunctionAddr *)v20->struct_VmFunctionAddr;
    if ( v22 )
    {
      if ( v22->CheckDisassemblyFunction && *(_WORD *)(v22->CheckDisassemblyFunction + 0x24) == 0x23 )// 解析JmpAddr跳转表，填充的VMOpcode都是0x23
        break;
    }
    ++v19;
    if ( !--ArrayNumber_2a )
      goto LABEL_22;
  }
  v23 = (struc_69 *)TList::Add(v3->struct_VmpOpcodePY_50);
  v23->ArrayNumber = TList::IndexOf(*(_DWORD *)(v21->struct_VmFunctionAddr + 0x1C), v24);// 该struc_SaveAllDisasmFunData->ArrayNumber（基本版）在数组第几项
  v23->FilterArrayNumber = TCollection::GetCount_0((int)v3) - 1;// 保存struc_SaveAllDisasmFunData->ArrayNumber（基本版）
}
LABEL_22:
  v25 = TCollection::GetCount_0(v3->struct_VmpOpcodePY_50) - 1;
```

然后对struct_VmpOpcodePY_50结构进行数组乱序

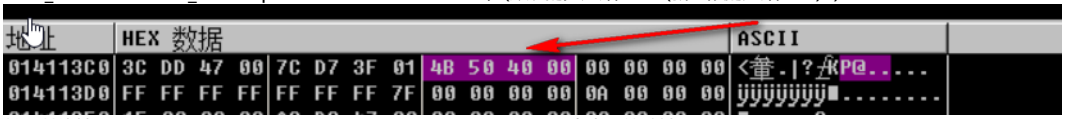
```
  }
LABEL_22:
  v25 = TCollection::GetCount_0(v3->struct_VmpOpcodePY_50) - 1;
  if ( v25 >= 0 )
  {
    ArrayNumber_2b = v25 + 1;
    v26 = 0;
    do
    {
      v27 = v3->struct_VmpOpcodePY_50;
      TCollection::GetCount_0(v3->struct_VmpOpcodePY_50);
      v28 = __linkproc__ RandInt();             // 以ArrayNumber作为随机数种子
      TList::Exchange(v27, v26++, v28);
      --ArrayNumber_2b;
    }
    while ( ArrayNumber_2b );                   // 乱序数组排列顺序
  }
```

3、循环填充struc_SaveAllDisasmFunData->ArrayNumber（基本版）Vmp_Address(0x10)替换成实际壳的真实地址

```
  v29 = TCollection::GetCount_0(v3->struct_VmpOpcodePY_50) - 1;
if ( v29 >= 0 )                                 // 循环填充struc_SaveAllDisasmFunData->ArrayNumber（基本版）Vmp_Address(0x10)替换成实际壳的地址
{
  ArrayNumber_2c = v29 + 1;
  v31 = 0;
  do
  {
    v84 = (struc_69 *)TCollection::GetItem_7(v3->struct_VmpOpcodePY_50, v31, (int)v30);
    ReadHexLen = 0;
    v33 = v84->ArrayNumber;                     // 该handle块首地址
    v34 = v84->FilterArrayNumber;               // 该handle块结束地址
    v35 = __OFSUB__(v34, v33);                  // 结束的 - 起始的 = 获取一共有多少组Opcode
    v36 = v34 - v33;
    if ( !((v36 < 0) ^ v35) )                   // 计算出该handle块指令长度是多少，第一条指令到jmp Dispatcher（jmp Dispatcher不算）为结束
    {
      v37 = v36 + 1;
      do
      {
        ReadHexLen += *(_DWORD *)(GetItem_7((int)v3, v33++, v32) + 0x6C);//  ReadHexLen      dd ?                    ; 读取Opcode长度，每次执行ReadHex函数便会累加
        --v37;
      }
      while ( v37 );                                        |
    }
    UserVmpStartAddr = sub_480BE0(a1a, ReadHexLen, v3->Characteristics & 0x20, 0i64);// 找到有足够空间存放该VmpHandle块的区段，并返回该区段的开始地址
    if ( !v84->ArrayNumber )
      *(_QWORD *)&v3->VmpStubStart = UserVmpStartAddr;// 保存壳的跳到解析器的地址 jmp VstartVM
    v38 = v84->ArrayNumber;
    v39 = v84->FilterArrayNumber;
    v35 = __OFSUB__(v39, v38);                  // 结束的 - 起始的 = 获取一共有多少组Opcode
    v40 = v39 - v38;
    if ( !((v40 < 0) ^ v35) )                   // 将找到的struc_SaveAllDisasmFunData->ArrayNumber（基本版）的Vmp_Address(0x10)替换成前面找到的UserVmpStartAddr(0x8)
    {
      v41 = v40 + 1;
      do
      {
        v30 = (struct_DisassemblyFunction *)GetItem_7((int)v3, v38, (int)v30);
        *(_QWORD *)&v30->LODWORD_VMP_Address = UserVmpStartAddr;
        UserVmpStartAddr += v30->ReadHexLen;
        ++v38;
        --v41;
```

每次执行完毕后sub_480BE0(a1a, ReadHexLen, v3->Characteristics & 0x20, 0i64)函数

struc_59->LODWORD_UserVmpStartAddr+=ReadHexLen;（默认的壳起始OEP（新区段的起始地址））



然后替换掉struct_DisassemblyFunction->LODWORD_VMP_Address为真正的地址

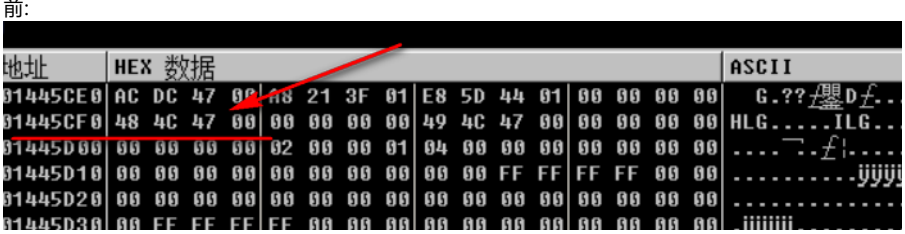
```
    if ( !((v40 < 0) ^ v35) )               // 将找到的struct_SaveAllDisasmFunData->ArrayNumber（基本版）的Vmp_Address(0x10)替换成前面找到的UserVmpStartAddr(0x8)
    {
      v41 = v40 + 1;
      do
      {
        v30 = (struct_DisassemblyFunction *)GetItem_7((int)v3, v38, (int)v30);
        *(_QWORD *)&v30->LODWORD_VMP_Address = UserVmpStartAddr;
        UserVmpStartAddr += v30->ReadHexLen;
        ++v38;
        --v41;
      }
      while ( v41 );                        // Opcode有几行就循环几次
    }
    ++v31;
    --ArrayNumber_2c;
  }
  while ( ArrayNumber_2c );
```

总结:

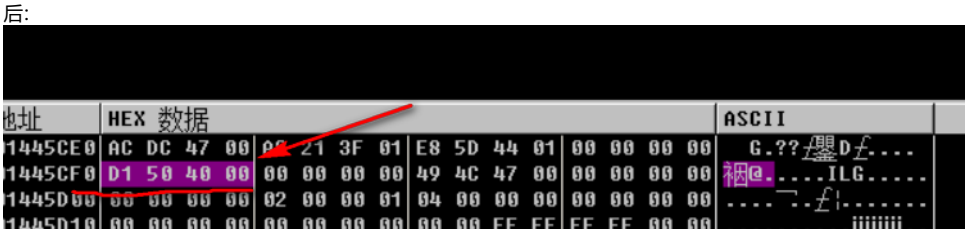
1、前面所有工作都是找到所有使用到的struct_DisassemblyFunction结构，然后对其地址进行替换成真实壳地址

替换前后对比:

前:



后:



4、修复特定值

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

```
TlistArrayClear(v3->struct_VmpOpcodePY_50, (int)v30);// 内容全部释放
FixArray((int)v3);                      // 修复数组排列顺序，从小到大
v42 = TCollection::GetCount_0(v3->struct_VmpOpcodePY_3D4) - 1;// 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,0xFACE
if ( v42 >= 0 )                         // 修复指令重定位信息
{
  ArrayNumber_2d = v42 + 1;
  v44 = 0;
  do
  {
    v45 = v3->struct_VmpOpcodePY_3D4;
    v46 = (struct_Edi_Addr *)TCollection::GetItem_10(v43, v44);
    v48 = (struct_DisassemblyFunction *)v46->struct_DisassemblyFunction;
    v49 = 0x17 * v46->Number;
    v50 = (struct_Disasm *)((char *)&v48->First + v49);
    v51 = v46->Constant_Byte - 1;       // 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
    if ( v51 )                          // 1
    {
      v52 = v51 - 1;
      if ( v52 )                        // 2
      {
        v53 = v52 - 1;
        if ( v53 )                      // 3
        {
          if ( v53 == 1 )               // 4  call dword ptr ds:[0xFACE0004]
          {
            v56 = v3->_prev_node;
            v57 = *(_DWORD *)(*(_DWORD *)(v56 + 0x10) + 0x34);
            v74 = *(_QWORD *)(TCollection::GetItem_1(v47, *(_DWORD *)(v56 + 0x38)) + 8);
            if ( v50->Mod_Rm )
            {
              *(_QWORD *)&v50->LODWORD_RestHex_Lval_Displacement_Immediate = v74 - Vmp_GetNextAddressStart(v48);
            }
            else
            {
              *(_QWORD *)&v50->LODWORD_RestHex_Lval_Displacement_Immediate = v74;
              v50->Tag = -2;
            }
          }
        }
```

5、修复所有 JMP VMDispatcher的地址

公式： jmp后的偏移量=目标地址-原来地址-5　　5是指令长度

```
      ++v72;
    }
    else if ( v61->VMOpcode != 0xC || 2 != v61->First.ModRM_mod__Or__Size )// VMpocode == 0xC是jmp系列
    {
      v67 = (struct_VmFunctionAddr *)v61->struct_VmFunctionAddr;// 判断是否是Handle块函数
      if ( v67 && LOBYTE(v67->Vm_Mnemonic) == 0xC && v67->CheckDisassemblyFunction )// 解析JmpAddr跳转表结束后，再生成一组以0xC标记结尾
      {
        v68 = 0;
        while ( !(*((_BYTE *)&v62->First.ModRM_mod__Or__Size + 0x17 * v68) & 2) )
        {
          if ( ++v68 == 3 )
            goto LABEL_71;
        }
        v75 = *(_QWORD *)(*(_DWORD *)(v62->struct_VmFunctionAddr + 0x1C) + 0x10);
        if ( *(&v62->First.Mod_Rm + 0x17 * v68) )
        {
          v69 = Vmp_GetNextAddressStart(v62);
          v59 = (int)(&v62->First.About_Lval_Byte_Word_Dword + 0x17 * v68);
          *(_QWORD *)((char *)&v62->First.LODWORD_RestHex_Lval_Displacement_Immediate + 0x17 * v68) = v75 - v69;
        }
        else
        {
          *(_QWORD *)((char *)&v62->First.LODWORD_RestHex_Lval_Displacement_Immediate + 0x17 * v68) = v75;
          *(int *)((char *)&v62->First.Tag + 0x17 * v68) = -2;
        }
LABEL_71:
        SetDisassemblyFunction(v62, v59);     // 修复重定位 Jmp dword ptr [eax*4+JumpAddr];跳到Handler执行处，由加壳引擎填充
      }
    }
    else
    {
      v65 = *(_QWORD *)(*(_DWORD *)(v61->struct_VmFunctionAddr + 0x1C) + 0x10);
      *(_QWORD *)&v61->First.LODWORD_RestHex_Lval_Displacement_Immediate = v65 - Vmp_GetNextAddressStart(v61);
      SetDisassemblyFunction(v62, v66);       // 修复重定位 Jmp VMDispatcher
    }
    ++v60;
    --ArrayNumber_2e;
  }
  while ( ArrayNumber_2e );
```

6、修复完毕后的样子

```
05053   55              push ebp
05054   68 00000000     push 0x0
05059   8B7424 28       mov esi,dword ptr ss:[esp+0x28]
0505D   BF 00504000     mov edi,HelloASM.00405000
05062   89F3            mov ebx,esi                     HelloASM.<ModuleEntryPo
05064   033424          add esi,dword ptr ss:[esp]      kernel32.77998484
05067   8A06            mov al,byte ptr ds:[esi]
05069   00D8            add al,bl
0506B   FEC0            inc al
0506D   C0C0 05         rol al,0x5
05070   F6D0            not al
05072   2C B0           sub al,0xB0
05074   F6D0            not al
05076   34 7A           xor al,0x7A
05078   8D76 01         lea esi,dword ptr ds:[esi+0x1]
0507B   00C3            add bl,al
0507D   0FB6C0          movzx eax,al
05080   FF3485 BB51400(  push dword ptr ds:[eax*4+0x4051BB]
05087   C3              retn
05088   54              push esp
05089 ^ E9 D9FFFFFF     jmp HelloASM.00405067
0508E   58              pop eax                         kernel32.77998484
0508F   8F00            pop dword ptr ds:[eax]          kernel32.77998484
05091 ^ E9 D1FFFFFF     jmp HelloASM.00405067
05096   58              pop eax                         kernel32.77998484
05097   5A              pop edx                         kernel32.77998484
05098   66:59           pop cx
```

下一篇文章介绍：

1、伪代码构造

2、伪代码加密（前面构成出Add的解密代码，所以这里要对称整出一套加密代码）

```
00000074 field_74        dd ?
00000078 field_78        dd ?
0000007C field_7C        dd ?
00000080 struct_VmpOpcodePY_80 dd ?          ; 随机填充与Encoding of a p-code加密有关
00000084 struct_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 struct_VmpOpcodePY_88 dd ?
0000008C struct_VmpOpcodePY_8C dd ?
00000090 struct_VmpOpcodePY_90 dd ?
00000094 struct_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 struct_VmpOpcodePY_98 dd ?
0000009C struct_VmpOpcodePY_9C dd ?
000000A0 struct_VmpOpcodePY_A0 dd ?
000000A4 Esi_Addr        db 816 dup(?)       ; 保存的都是struct_DisassemblyFunction结构
000003D4 struct_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<-r
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 struct_VmpOpcode ends
000003DC
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 struct_VmpOpcodePY_3D4 struc ; (sizeof=0x14, mappedto_354)
```

正向就是解密，取反就是加密了

0x4=add ---->加密用sub

0x5=xor ---->xor不变
0x34=sub ---->加密用add
0x43=rol 循环左移
0x44=ror 逻辑右移
0x5C=not 单操作数
0x29=inc 单操作数
0x2A=dec 单操作数
0x5D=neg 单操作数
0x31=bswap 单操作数