VMP学习笔记之ESI伪代码生成与加密（六）

第六章

主题：VMP学习笔记之ESI伪代码生成与加密

1、构造ESI指令的基本套路

2、ESI伪代码加密（保持一致性，前面对Add系列构造出解密代码，所以这里要反向加密）

3、总结加壳流程

学习到的知识：

1、知道Handle块生成的套路即可（核心）

2、Esi伪代码加密不需要深入了解，为了完整性我才写下去（无用）

基础知识：

1、熟悉壳的都会发现壳入口都是：

pushad

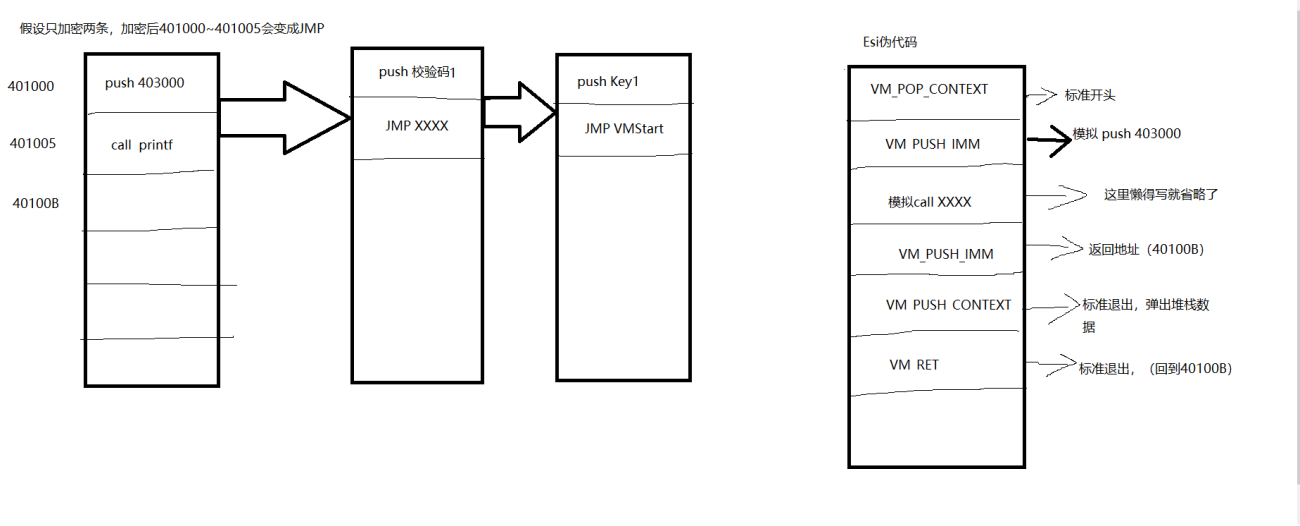
pushfd

XXXX

popad

popfd

前后基本是固定的套路，主要是中间真正模拟的代码不同



2、如何看Vmp_SetEsiStruct参数对应哪条Handle块

ESI_Matching_Array每一组是8个字节

ESI_Matching_Array[2~3]：VMopcode（助记符）

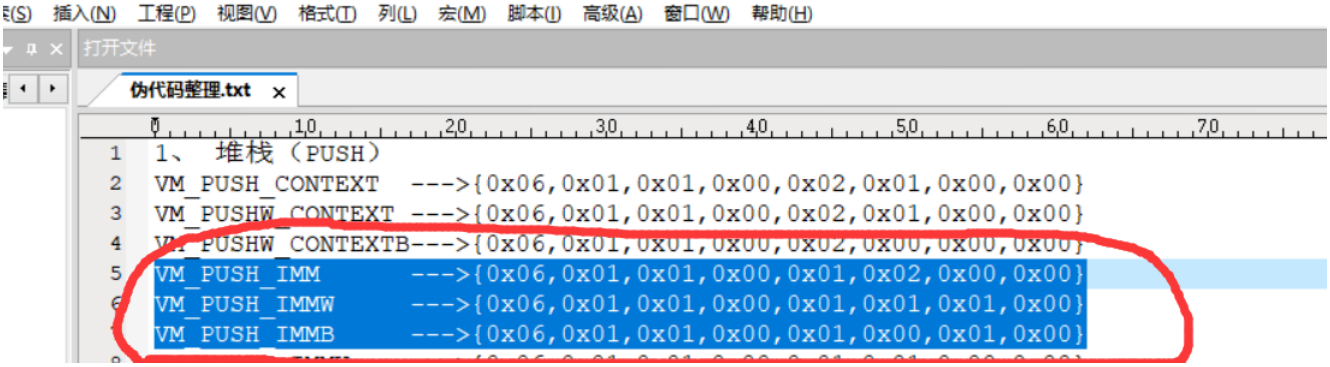
ESI_Matching_Array[4]：　寻址方式

ESI_Matching_Array[5]：　大小

```
209          a6 = ~a6;
210        }
211        if ( v6->Tag > -1 || v6->Mod_Rm || v27 & 4 )
212          a5 |= 8u;
213        LOBYTE(v5) = Size_1;
214        LODWORD(v5) = Vmp_SetEsiStruct(*(struct_DisassemblyFunction **)(a4 - 4), 1, 1, v6->Tag, a5, a6, v5);
```

位数

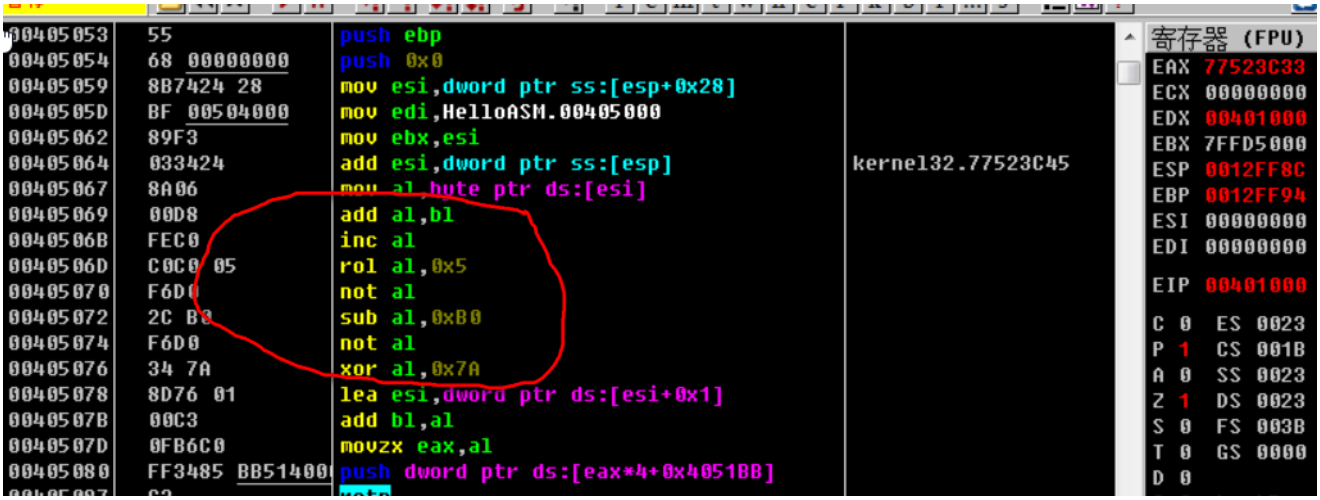
\资料\VMP主程序1.21\伪代码整理.txt]



3、针对ADD系列提供对称的加密流程

加壳后动态解密

加壳前我们应该将数据加密一遍了



正文：

1、ESI伪代码构造的框架简单介绍

按照框架逻辑分为以下三步：

1、处理特殊Opcode （不重要，为了完整性）

2、构造Esi伪代码 （核心，看完这一步即可）

3、加密Esi伪代码与拼接跳转地址 （不重要，为了完整性才写下去）

```
ArrayNumber_1 = TCollection::GetCount_0((int)v1);// 获取需要Vm的流程个数（加密了多少个函数），一般demo版本只能选择一个流程加密
if ( ArrayNumber_1 - 1 >= 0 )
{
  ArrayNumber = ArrayNumber_1;
  v73 = 0;
  do
  {
    v51 = TList::Get(v1->struct_VmpAllDataPY_64, v73);
    v53 = (struc_UserVmpPEInformation *)(*(int (__fastcall **)(int, int))(v1->This + 0x10))(v1->This, v51);// 获取struc_UserVmpPEInformation结构（有多少
    if ( v53->Executable )              // 判断文件属于PE文件还是ELF文件：0=错误 1=PE 2=ELF
    {
      if ( var8 < 1 )                   // 处理特殊指令Opcode例如：call jmp
      {
        if ( !Vmp_DisposeUserSpecialOpcode(v53, v52) )
          goto LABEL_72;
      }
      else if ( var8 == 1 )             // 构造ESI伪代码
      {
        if ( !(unsigned __int8)(*(int (__fastcall **)(int, int))(v53->This + 0x1C))(
                                v53->This,
                                v1->struct_VmpAllDataPY_60) )
          goto LABEL_72;
      }
      else if ( var8 == 2 )             // ESI伪代码加密
      {
        v56 = (int *)v1->field_24;
        (*(void (__fastcall **)(int, int, int, int *))(v53->This + 0x20))(
          v53->This,
          v1->struct_VmpAllDataPY_60,
          v1->VmpStubStart,
          v56);
      }
    }
    ++v73;
    --ArrayNumber;
  }
  while ( ArrayNumber );
```

2、Vmp_DisposeUserSpecialOpcode函数分析

3、Vmp_CreateEsiBytecode函数分析

3、1 构造出基本框架的Handle块

```
v4 = 0;
ArrayNumber = TCollection::GetCount_0((int)v3) - 1;// 得到用户要加密的代码行数
if ( ArrayNumber >= 0 )                         // 将每一条Opcode汇编指令生成对应的Handle框架，保存在struct_DisassemblyFunction->struc_EsiBytecode
{
  ArrayNumber_1 = ArrayNumber + 1;
  Number = 0;
  do
  {
    v8 = (struct_DisassemblyFunction *)GetItem_7((int)v3, Number, v6);
    if ( CheckVMStart_VMEnd1(v3, *(_QWORD *)&v8->LODWORD_VMP_Address) )// 判断地址合法性
      break;
    (*(void (__cdecl **)(struct_VmpAllDataPY_60 *))(*(_DWORD *)v3->_prev_node + 0x18))(a1a);// 进度条的，无视  界面的东西
    if ( !Number                                // 判断是不是第一次进入
      || *(_BYTE *)(GetItem_7((int)v3, Number - 1, v9) + 0x84) & 0x20// 获取上一组struct_DisassemblyFunction结构
      || *(_WORD *)(GetItem_7((int)v3, Number - 1, v9) + 0x24) == 0x23
      || v8->Flag & 3 )
    {
      if ( v8->VMOpcode != 0x23 )               // 解析跳转表jmp dword ptr ds:[eax*4+JumpAddr]Opcode的时候VMOpcode就是0x23
      {
        Vmp_CreateVMContextHandle(v8, 0x200); // 构造VMContext的命令
        if ( v8->Flag & 2 && v8->field_A4 )
        {
          v10 = v3->_prev_node;
          if ( *(_BYTE *)(v10 + 0x48) & 8 )
          {
            LOBYTE(v10) = v8->Magic;
            v11 = v10;
            v12 = __linkproc__ RandInt();
            Vmp_SetEsiStruct(v8, 1, 1, -1, 0x121, v12, v11);
          }
          else
          {
            LOBYTE(v10) = v8->Magic;
            Vmp_SetEsiStruct(v8, 1, 2, -1, 0x20, 20i64, v10);
          }
          sub_493374((int)v8, 0, 0);
          v8->Flag &= 0xFFDFu;
          v4 = v4->ArrayNumber - Number;
        }
      }
      if ( !Vmp_UserOpcodeDisassembly(v8, v9, (struc_AllBytecode *)v3) )// 将Opcode生成对应的Handle块
      {
        v70 = 0;
        break;
      }
      v15 = Vmp_GetNextAddressStart(v8);        // 获取下一条指令地址
      if ( v15 == *(_QWORD *)&v3->UserVmpFunctionEndAddr
        || (v16 = Vmp_GetNextAddressStart(v8), CheckVMStart_VMEnd1(v3, v16)) )
      {
        if ( v8->VMOpcode != 0x23 && !(v8->Flag & 0x40) )
        {
          LOBYTE(v15) = v8->Magic;
          v17 = v15;
          v18 = Vmp_GetNextAddressStart(v8);
          Vmp_SetEsiStruct(v8, 1, 1, -1, 10, v18, v17);
          sub_493374((int)v8, 0, 9);
        }
        v4->NextArrayNumber = Number;
        v8->struc_AllBytecode = (int)v4;
        if ( v8->VMOpcode != 0x23
          && !(v8->Flag & 0x20)
          && Number < TCollection::GetCount_0((int)v3)// 获取用户需要加密的Opcode总个数
          && *(_WORD *)(GetItem_7((int)v3, Number + 1, v6) + 0x84) & 3 )// 获取下一组struct_DisassemblyFunction结构
        {
          v19 = v3->_prev_node;
          if ( *(_BYTE *)(v19 + 0x48) & 8 )
          {
            LOBYTE(v19) = v8->Magic;
            v20 = v19;
            v21 = __linkproc__ RandInt();
            Vmp_SetEsiStruct(v8, 1, 1, -1, 0x121, v21, v20);
          }
          else
          {
            {

0008CEC2 156
```

3、2 模拟出实际代替指令，以Push 403000为例

1、执行Vmp_CreateVM_POP_Context函数，构造出VM_POP_CONTEXT框架

我们发现以0x12（PUSH KEY）开始，0x14（push 0）结尾。其中过滤掉ESP

```
int v6; // edi@11
int v7; // eax@12
signed int v8; // esi@12
__int16 a5; // [sp+0h] [bp-18h]@1
__int16 a5a; // [sp+0h] [bp-18h]@3
struc_PushRegister *v12; // [sp+4h] [bp-14h]@9

v2 = a1;
a5 = a2 | 0x20;
v3 = TCollection::GetCount_0((int)a1);
if ( !v3 )                                      // 判断是否第一次，（TList没有元素就是第一次了）
  v2->Flag |= 0x10u;
LOBYTE(v3) = v2->Magic;
Vmp_SetEsiStruct(v2, 2, 2, -1, a5, 0x12i64, v3);// VM_POP_CONTEXT
a5a = a5 & 0xFFFB;
if ( HIBYTE(a5a) & 2 )
{
  v4 = (struc_77 *)v2->field_A4;
}
else if ( HIBYTE(a5a) & 4 )
{
  v4 = (struc_77 *)v2->struct_DisassemblyFunctionPY_A8;
}
else
{
  v4 = 0;
}
if ( v4 )
{
  v4 = sub_49FE40(v4);
  v12 = (struc_PushRegister *)v4->struc_PushRegister;
}
else
{
  v12 = *(struc_PushRegister **)(*(_DWORD *)(*(_DWORD *)(v2->_prev_node + 4) + 0x50) + 0x3D8);// 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
}
v5 = v12;
v6 = v12->AddrNumber - 1;
if ( v6 >= 0 )
{
  do
  {
```

```
  v5 = v12;
  v6 = v12->AddrNumber - 1;
  if ( v6 >= 0 )
  {
    do
    {
      v7 = TList::Get((int)v12, v6);
      v8 = v7;
      if ( v7 == 4 )                            // Esp
        v8 = 0x13;
      if ( v4 )
        sub_493568(v2, a5a & 0xFFFD | 0x40, *(&v4->EncodingOfRegAtJumpsPY_08 + v8), 1);
      LOBYTE(v7) = v2->Magic;
      v5 = (struc_PushRegister *)Vmp_SetEsiStruct(v2, 2, 2, -1, a5a, v8, v7);// VM_POP_CONTEXT
      --v6;
    }
    while ( v6 != -1 );
  }
  LOBYTE(v5) = v2->Magic;
  return Vmp_SetEsiStruct(v2, 2, 2, -1, a5a, 0x14i64, (int)v5);// VM_POP_CONTEXT
}
```

2、执行Vmp_UserOpcodeDisassembly函数，构造Push XXXX指令

首先根据VMopcode选择不同的执行流程

```
  int savedregs; // [sp+1Ch] [bp+0h]@5

  a1a = a1;
  if ( !(a1->word86 & 1) )
    return 1;
  v4 = (struct_VmFunctionAddr *)a1->struct_VmFunctionAddr;
  if ( v4 && LOBYTE(v4->Vm_Mnemonic) == 0xD )
  {
    sub_496A70((int)&savedregs);
  }
  else
  {
    v5 = a1a->VMOpcode;
    v6 = (unsigned __int8)a1a->VMOpcode;
    switch ( v6 )                               // 解析作者定义的OPcode
    {
      case 1:                                   // push
        Vmp_UseDisamaStruct(0, a5[0], (unsigned __int8)a3, (int)&savedregs);
        break;
      case 2:                                   // pop
        Vmp_UseDisamaStruct(0, 1, (unsigned __int8)a3, (int)&savedregs);
        break;
      case 3:                                   // mov and
        Vmp_UseDisamaStruct(1, a5[0], (unsigned __int8)a3, (int)&savedregs);
        Vmp_UseDisamaStruct(0, 1, (unsigned __int8)a3, (int)&savedregs);
        break;
      case 7:                                   // Lea
        Vmp_UseDisamaStruct(1, 0x20, (unsigned __int8)a3, (int)&savedregs);
        Vmp_UseDisamaStruct(0, 1, (unsigned __int8)a3, (int)&savedregs);
        sub_4954E4(a1a->Magic, a1a->First.About_Lval_Byte_Word_Dword, (int)&savedregs);
        break;
      case 0x5C:
        sub_497788(0, (int)&savedregs);
        Vmp_UseDisamaStruct(0, 1, (unsigned __int8)a3, (int)&savedregs);
        break;
      case 0x5D:
        v7 = a1a->First.About_Lval_Byte_Word_Dword;
        Vmp_UseDisamaStruct(0, a5[0], v7, (int)&savedregs);
        Vmp_SetEsiStruct(a1a, 1, 1, -1, 0, -1i64, v7);
        Vmp_SetEsiStruct(a1a, 4, 0, -1, 0, 1i64, v7);
        Vmp_SetEsiStruct(a1a, 2, 2, -1, 0, 16i64, a1a->Magic);
```

根据类型区别读取不同的数据

```
            LODWORD(v5) = *(_DWORD *)(a4 - 4);
            if ( *(_DWORD *)(v5 + 0x88) )
            {
              LODWORD(v5) = *(_DWORD *)(*(_DWORD *)(a4 - 4) + 0x88);
              if ( *(_BYTE *)(v5 + 8) )
                a5 = 7;
            }
          }
        }
        if ( !(v6->ModRM_mod__Or__Size & 8) && v27 & 2 )
        {
          a5 |= 0x10u;
          v5 = ~a6;
          a6 = ~a6;
        }
        if ( v6->Tag > -1 || v6->Mod_Rm || v27 & 4 )
          a5 |= 8u;
        LOBYTE(v5) = Size_1;
        LODWORD(v5) = Vmp_SetEsiStruct(*(struct_DisassemblyFunction **)(a4 - 4), 1, 1, v6->Tag, a5, a6, v5);
      }
      if ( v6->ModRM_mod__Or__Size & 4 )
      {
        if ( v6->ModRM_mod__Or__Size & 8 )
        {
          v12 = v6->SIB_scale;
          if ( v12 )
            Vmp_SetEsiStruct(*(struct_DisassemblyFunction **)(a4 - 4), 1, 1, -1, 0, (unsigned __int8)v12, 1);
          v13 = sub_4954E4(v6->About_RegType_8_16_32, *(_BYTE *)(*(_DWORD *)(a4 - 4) + 0x9D), a4);
          LOBYTE(v13) = v6->About_RegType_8_16_32;
          LODWORD(v5) = Vmp_SetEsiStruct(
                          *(struct_DisassemblyFunction **)(a4 - 4),
                          1,
                          2,
                          -1,
```

如何看Vmp_SetEsiStruct参数对应那条Handle块

ESI_Matching_Array每一组是8个字节

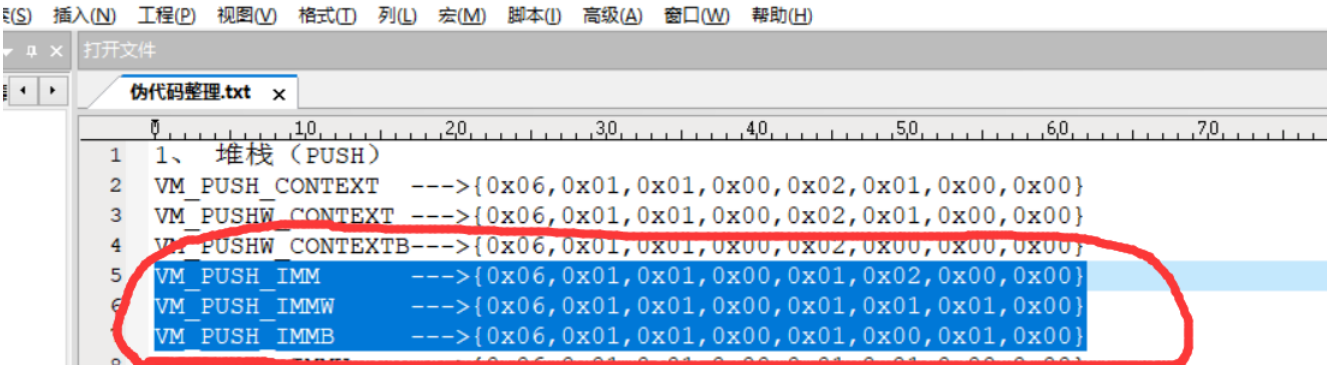
ESI_Matching_Array[2~3]：VMopcode（助记符）

ESI_Matching_Array[4]：　寻址方式
ESI_Matching_Array[5]：　大小
不就是对应我们那一句push 401000吗?

```
209           a6 = ~a6;
210         }
211         if ( v6->Tag > -1 || v6->Mod_Rm || v27 & 4 )
212           a5 |= 8u;
213         LOBYTE(v5) = Size_1;
214         LODWORD(v5) = Vmp_SetEsiStruct(*(struct_DisassemblyFunction **)(a4 - 4), 1, 1, v6->Tag, a5, a6, v5);
```

位数

\资料\VMP主程序1.21\伪代码整理.txt]

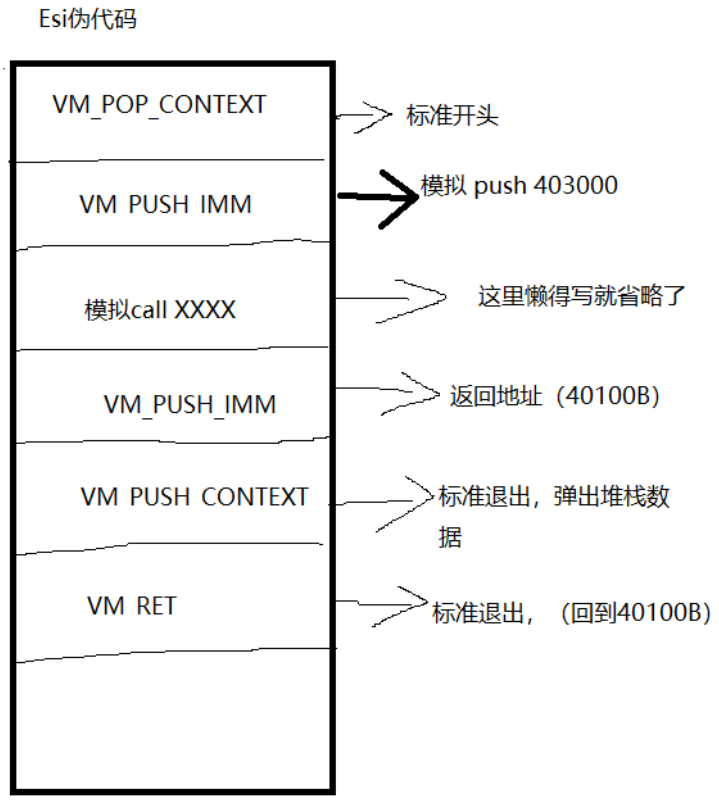


现在我们就已经把push 401000这一条指令模拟成功了
VM_POP_CONTEXT　---->0xB次
VM_PUSH_IMM　　---->push 401000
那么我们还剩下构造退出指令的代码，就是VM_PUSH_CONTEXT加VM_RET

3、 剩下代码就是构造出结尾指令的
VMP是将两条指令为一组，在最后一组代码后面加上退出指令操作

```
39          break;
40        }
41        v15 = Vmp_GetNextAddressStart(v8);        // 获取下一条指令地址
42        if ( v15 == *(_QWORD *)&v3->UserVmpFunctionEndAddr
43          || (v16 = Vmp_GetNextAddressStart(v8), CheckVMStart_VMEnd1(v3, v16)) )
44        {
45          if ( v8->VMOpcode != 0x23 && !(v8->Flag & 0x40) )
46          {
47            LOBYTE(v15) = v8->Magic;
48            v17 = v15;
49            v18 = Vmp_GetNextAddressStart(v8);
50            Vmp_SetEsiStruct(v8, 1, 1, -1, 10, v18, v17);
51            Vmp_CreateVM_PUSH_Ret_Context((int)v8, 0, 9);
52          }
53        }
54        v4->NextArrayNumber = Number;
55        v8->struc_AllBytecode = (int)v4;
56        if ( v8->VMOpcode != 0x23
57          && !(v8->Flag & 0x20)
58          && Number < TCollection::GetCount_0((int)v3)// 获取用户需要加密的Opcode总个数
59          && *(_WORD *)(GetItem_7((int)v3, Number + 1, v6) + 0x84) & 3 )// 获取下一组struct_DisassemblyFunction结构
60        {
61          v19 = v3->_prev_node;
62          if ( *(_BYTE *)(v19 + 0x48) & 8 )
63          {
64            LOBYTE(v19) = v8->Magic;
65            v20 = v19;
66            v21 = __linkproc__ RandInt();
67            Vmp_SetEsiStruct(v8, 1, 1, -1, 0x121, v21, v20);
68          }
69          else
70          {
71            LOBYTE(v19) = v8->Magic;
72            Vmp_SetEsiStruct(v8, 1, 2, -1, 0x20, 0x14i64, v19);
73          }
74          v8->struct_DisassemblyFunctionPY_A8 = *(_DWORD *)(GetItem_7((int)v3, Number + 1, v22) + 164);
75          Vmp_CreateVM_PUSH_Ret_Context((int)v8, 0x400, 0);
76        }
77        if ( v8->Flag & 0x40 )
78          v4 = 0;
```

执行完毕后如下:



总结:

1、将两组Opcode信息保存为一组的结构是struc_AllBytecode

```
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 struc_AllBytecode struc ; (sizeof=0x3C, mappedto_368)
00000000 This            dd ?
00000004 _prev_node      dd ?                    ; 指向struc_UserVmpPEInformationPY_50
00000008 ArrayNumber     dd ?                    ; 第x组要加密的Opcode
0000000C NextArrayNumber dd ?                    ; 第x组要加密的Opcode下一条(如何是最后一条跟ArrayNumber相同)
00000010 ContextRegBuff  dd 6 dup(?)             ; 保存寄存器信息
00000028 ContextRegNumber dd ?                   ; ContextRegNumber（一般是0x10个）
0000002C Magic           dd ?                    ; Magic
00000030 struc_84PY_30   dd ?                    ; 随机的
00000034 struct_DisassemblyFunction dd ?         ; 独立生成一堆无用的忽悠人的Esi伪代码信息
00000038 field_38        dd ?
0000003C struc_AllBytecode ends
0000003C
```

2、现在ESI伪代码已经构造完成，那么还剩下两个问题：

ESI代码存在哪里?

ESI代码对应的Index是什么?

3、3 针对struc_UserVmpPEInformationPY_50~54进行初始化或则乱序操作

struc_UserVmpPEInformationPY_50 == 2个Opcode为一组

struc_UserVmpPEInformationPY_54 == 保存特殊Opcode的

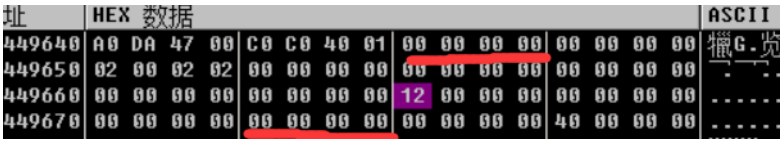
```
if ( v70 )
{
  v23 = TCollection::GetCount_0(v3->struc_UserVmpPEInformationPY_50) - 1;
  if ( v23 >= 0 )
  {
    ArrayNumber_1 = v23 + 1;
    v24 = 0;
    do
    {
      v25 = v3->struc_UserVmpPEInformationPY_50;
      TCollection::GetCount_0(v3->struc_UserVmpPEInformationPY_50);
      v26 = __linkproc__ RandInt();
      TList::Exchange(v25, v24++, v26);
      --ArrayNumber_1;
    }
    while ( ArrayNumber_1 );                    // 打乱数组顺序
  }
  (*(void (**)(void))(*(_DWORD *)v3->struc_UserVmpPEInformationPY_54 + 8))();// 元素清零
  v27 = TCollection::GetCount_0((int)&v3->struc_UserVmpSpecialDisassemblerOpcode->This) - 1;
  if ( v27 >= 0 )                               // 根据v3->struc_UserVmpSpecialDisassemblerOpcode添加对应的数组元素
  {
    ArrayNumber_1 = v27 + 1;
    v28 = 0;
    do
    {
      TList::Add(v3->struc_UserVmpPEInformationPY_54, v28++);
      --ArrayNumber_1;
    }
    while ( ArrayNumber_1 );
  }
  v29 = TCollection::GetCount_0((int)&v3->struc_UserVmpSpecialDisassemblerOpcode->This) - 1;
  if ( v29 >= 0 )
  {
    ArrayNumber_1 = v29 + 1;
    v30 = 0;
    do
    {
      v31 = v3->struc_UserVmpPEInformationPY_54;
      v32 = *(_DWORD *)(v31 + 8);
      v33 = __linkproc__ RandInt();
      TList::Exchange(v31, v30++, v33);
      --ArrayNumber_1;
```

3、4 填充补充Esi信息

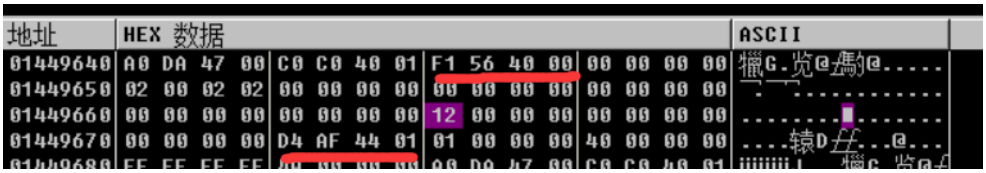
获取该Esi存放的地址和获取该Handle块的Index

```
    ArrayNumber_1 = v46 + 1;
    v48 = 0;
    do
    {
      v79 = (struc_AllBytecode *)TCollection::GetItem_7(v3->struc_UserVmpPEInformationPY_50, v48, v47);
      LOBYTE(v79->Magic) = 1;
      a2a = 0;
      v49 = v79->ArrayNumber;
      v50 = v79->NextArrayNumber;
      v40 = __OFSUB__(v50, v49);
      v51 = v50 - v49;
      if ( !((v51 < 0) ^ v40) )
      {
        v52 = v51 + 1;
        v72 = v79->ArrayNumber;
        do
        {
          (*(void (__cdecl **)(struct_VmpAllDataPY_60 *))(*(_DWORD *)v3->_prev_node + 24))(a1a);
          v54 = GetItem_7((int)v3, v72, v53);
          Vmp_FindAndSetTOStruct_0(v54);        // 获取当前Handle的Index，并填充该Index
          a2a += GetEsiBytecodeLen(v54);        // 计算Esi伪代码长度
          ++v72;
          --v52;
        }
        while ( v52 );
      }
      *(_QWORD *)v76 = SetAddress(a1a, a2a, a3a, 0i64);
      v56 = v79->ArrayNumber;
      v57 = v79->NextArrayNumber;
      v40 = __OFSUB__(v57, v56);
      v58 = v57 - v56;
      if ( !((v58 < 0) ^ v40) )
      {
        v59 = v58 + 1;
        v73 = v79->ArrayNumber;                 // 这里填充struct_DisassemblyFunction+0x78
        do
        {
          v60 = (struct_DisassemblyFunction *)GetItem_7((int)v3, v73, v55);
          Vmp_SetEsiBytecodeAddress(v60, (int)v76);// 设置Esi伪代码存放的地址
          ++v73;
          --v59;
        }
```

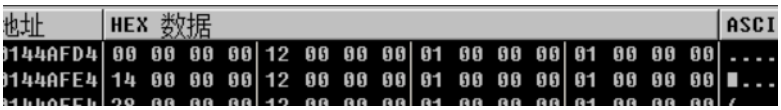
填充前:



填充后:



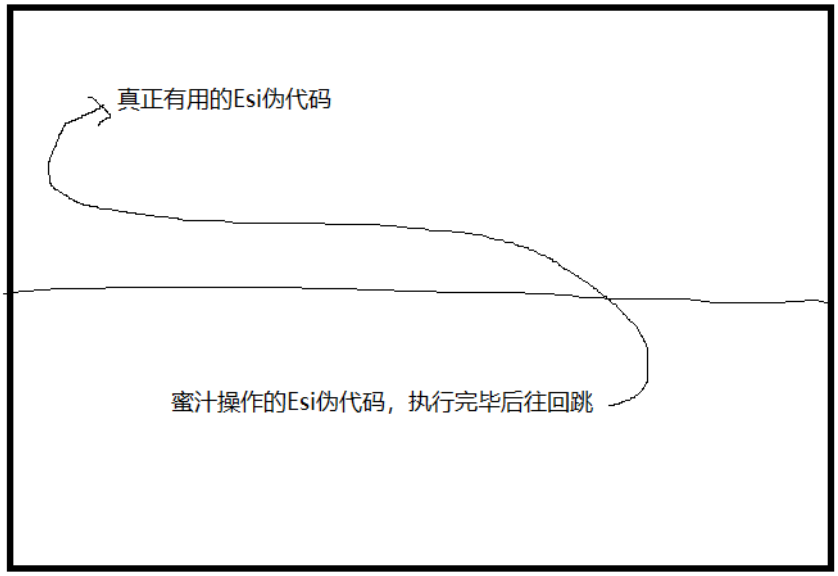
Index是0，这是未加密的



3、5 蜜汁操作，生成一堆无用的垃圾代码

各位真正逆向分析找到Esi往前跳的即可,感觉这堆指令没什么用

```
    }
    LOBYTE(v79->Magic) = 0;
    v61 = (struct_DisassemblyFunction *)v79->struct_DisassemblyFunction;
    Vmp_CreateVM_POP_Context((struct_DisassemblyFunction *)v79->struct_DisassemblyFunction, 0);// 构成出POP_CONTEXT标准开头
    AddEnc(v61, 0x40, v79->struc_84PY_30, 1);// 针对Add系列构造出对称的加密流程
    LOBYTE(v62) = v61->Magic;
    v63 = Vmp_SetEsiStruct(v61, 2, 2, -1, 0, 0x11i64, v62);// VM_POP_CONTEXT
    LOBYTE(v63) = v61->Magic;
    Vmp_SetEsiStruct(v61, 1, 2, -1, 0x20, 0x14i64, (int)v63);// VM_PUSH_CONTEXT
    v64 = Vmp_CreateVM_PUSH_Context((int)v61, 0, 0);
    LOBYTE(v64) = v61->Magic;
    v65 = Vmp_SetEsiStruct(v61, 1, 2, -1, 0x20, 0x11i64, (int)v64);// VM_PUSH_CONTEXT
    LOBYTE(v65) = v61->Magic;
    Vmp_SetEsiStruct(v61, 0xC, 0, -1, 0x20, 0i64, (int)v65);// VM_JMP
    Vmp_FindAndSetTOStruct_0((int)v61);
    v66 = GetEsiBytecodeLen(v79->struct_DisassemblyFunction);
    *(_QWORD *)v76 = SetAddress(a1a, v66, a3a, 0i64);
    Vmp_SetEsiBytecodeAddress((struct_DisassemblyFunction *)v79->struct_DisassemblyFunction, (int)v76);
    ++v48;
    --ArrayNumber_1;
  }
  while ( ArrayNumber_1 );
```



3、6 Vmp_CreateEsiBytecode函数总结

1、其实VMP生成Handle块套路基本都是固定模板
2、假设是mov模拟：push +pop 不就可以了吗
xchg交换：push + pop 或则 xor a,b xor b,a xor a,b
其实自己随便搞也能模拟出来
3、剩下比较复杂的逻辑运算跟模拟cmp指令的我专门写一篇讲解

4、Vmp_EncEsiBytecodeAndCreateJmp函数分析

4、1 构造push 伪指令 + jmp Start指令

```
a3a[1] = v308->Characteristics & 0x20;        // 节区属性
v3 = TCollection::GetCount_0(v308->struc_UserVmpPEInformationPY_50) - 1;
if ( v3 >= 0 )                                // 填充好Vmp跳转指令  push 伪代码  jmp 调度器
                                              // 注意这段循环之处理普通指令类似于：call是进行特殊处理，并不在这里面
{
  ArrayNumber_1 = v3 + 1;
  Counter = 0;
  do
  {
    v279 = (struc_59 *)TCollection::GetItem_7(v308->struc_UserVmpPEInformationPY_50, Counter, v4);
    RandomNumber = __linkproc__ RandInt();
    RandomNumber_1 = RandomNumber + 1;
    *(_QWORD *)(v279->struct_DisassemblyFunction + 0x10) = SetAddress(a1a, RandomNumber + 0xB, word_48FEFC[0], 0i64);
    VMP_Address = *(_QWORD *)(v279->struct_DisassemblyFunction + 0x10);// struct_DisassemblyFunction->VMP_Address
    Buff[0] = *(_DWORD *)(v279->struct_DisassemblyFunction + 0x78);// EsiBytecode（未知只是猜测），只有解析用户需要加密的Opcode代码才会产生
    v300 = (struc_VmpJmpAddress *)(*(int (__cdecl **)(unsigned int, void *, int *))(*(_DWORD *)v308->struc_UserVmpPEInformationPY_48
                                                                                   + 0x10))(// Add 添加数组元素
                                  v220,
                                  v221,
                                  v222);
    Vmp_StructureOpcode(v300, 0, 0x68, 0);    // push
    Vmp_StructureOpcode(v300, 2, Buff[0], 0); // bytecode         ;伪代码地址 ，作为参数
    Vmp_StructureOpcode(v300, 0, 0xE9, 0);    // jmp
    Vmp_StructureOpcode(                      // VstartVM(调度器VStartVM)
      v300,
      2,                                      // 长度
      VstartVM - VMP_Address - 0xA,           // push + jmp =0xA 长度？？？
      (unsigned __int64)(VstartVM - VMP_Address - 0xA) >> 0x20);
    RandomNumber_2 = RandomNumber_1 - 1;
    if ( RandomNumber_2 >= 0 )                // 根据前面随机数为长度（5Max）：指令后面随意填充随机字节
    {
      Counter_1 = RandomNumber_2 + 1;
      do
      {
        v8 = __linkproc__ RandInt();
        Vmp_StructureOpcode(v300, 0, v8, (unsigned __int64)v8 >> 0x20);
        --Counter_1;
      }
      while ( Counter_1 );
    }
    Vmp_SetOpcodeLenOrAddress(v300, (int)&VMP_Address);// 函数功能：设置Opcode长度和填写地址
                                              // 函数返回值：当前Opcode地址+长度=下一条需要填充的地址
0008D400 350
```

5、画图总结所学内容