VMP学习笔记之壳基础（一）

【文章标题】: Vmp1.21学习笔记
【文章作者】: 黑手_鱼
【软件名称】: Vmp1.21
【下载地址】: 自己搜索下载
【加壳方式】: UPX 0.89.6 - 1.02 / 1.05 - 2.90 (Delphi)
【编写语言】: Borland Delphi 4.0 - 5.0
【操作平台】: win7 32位
【作者声明】: 以看雪作者waiWH的VMP还原系列为原型逆向分析

2019年08月08日 21:24:05

章节目录:

第一章内容:
主题：壳的基本操作
1、读取PE结构信息
2、增加区段
3、根据加密等级选择不同的框架

第二章内容:
主题：Opcode快速入门
1、了解Opcode解析过程
2、辅助第三章解析Opcode引擎而编写的
3、无脑查表就对了

第三章内容:
主题：反汇编引擎框架学习
1、看懂第二章就看得懂第三章
2、无脑查表就对了

第四章内容（已完成）:
主题：壳的初始化与Handle块优化
学到的东西:
1、去掉无用的Handle块（不重要）
2、指令的等级变换
3、部分指令变形
4、汇编的多变性
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

第五章内容:
主题：壳的重定位修复

第六章内容:
主题：壳的伪代码生成与排序等等
1、构造ESI指令的基本套路
2、ESI伪代码加密
3、总结加壳流程

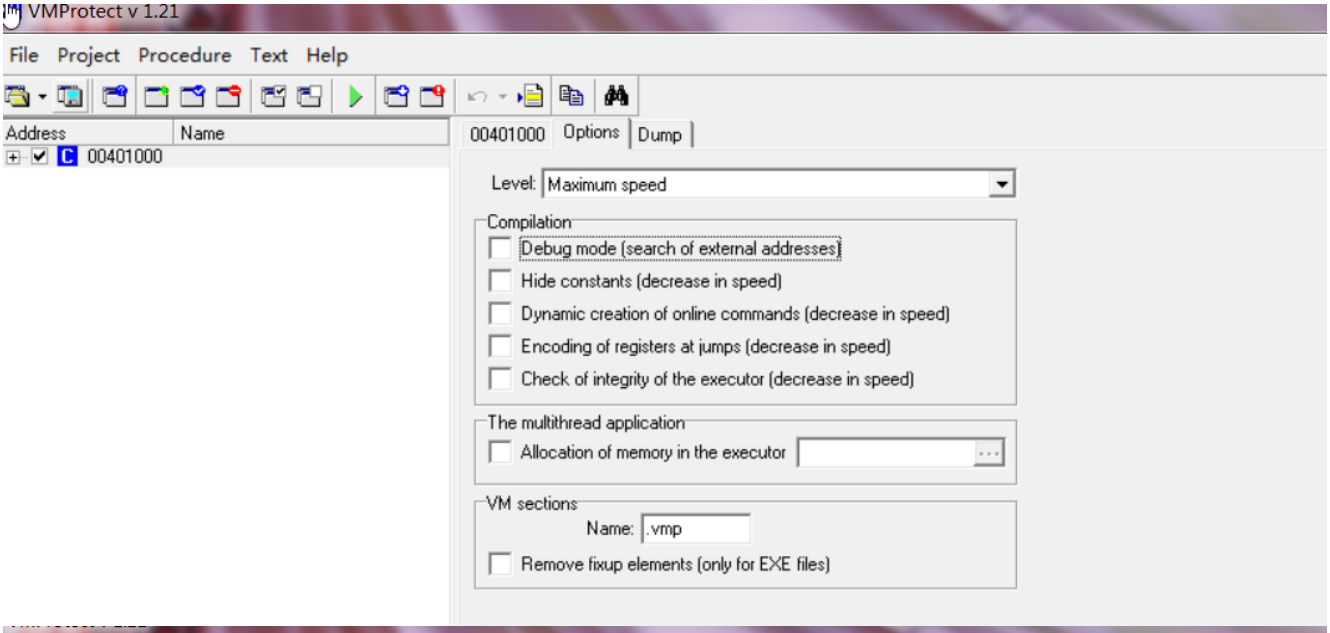
第七章内容:
主题：万用门介绍
1、NOR实现逻辑运算
2、cmp实现（sub）
3、jxx实现

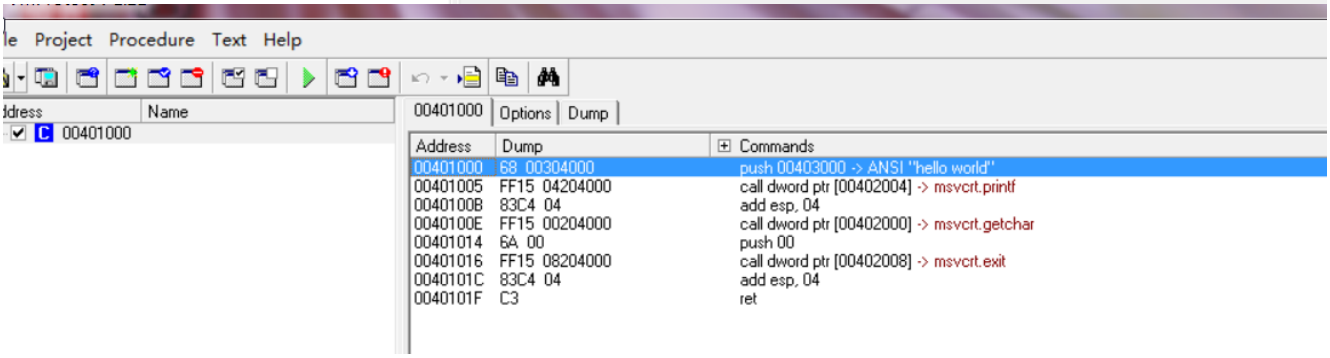
第八章内容:
主题：Vmp壳的实现或则去混淆插件（未完成）

说明:

1、加壳机的壳是秒杀壳，自行百度
2、HelloASM.exe是测试demo

3、保护全关





正文：

0、基础知识之加壳基本套路：

1、读取PE信息

2、添加区段

3、修复重定位

4、获取壳需要使用的API（PEB那一套）

1、读取PE基本信息

IDA View-A | Pseudocode-A | Structures

```
70  unknown_libname_2062(2, (_BYTE *)&v48 + 1);
71  __linkproc__ LStrCopy(&v45);
72  __linkproc__ LStrCmp(v6, &IMAGE_DOS_SIGNATURE);// 判断MZ头合法性
73  if ( v7 )
74  {
75    sub_4131E4(v36, (LONG)v37, v38, v39);
76    (*(void (__fastcall **)(signed int, int))(*(_DWORD *)v5 + 4))(2, (int)&v4->pDosHeader_e_lfanew);// 获取到pDosHeader->e_lfanew内容
77    v8 = v4->pDosHeader_e_lfanew;
78    sub_4131E4(v40, v41, v42, (DWORD)v43);
79    (*(void (__fastcall **)(signed int, int))(*(_DWORD *)v5 + 4))(4, (int)&v4->pNtHeader_Signature);// 获取到PE头
80    v43 = (int *)&v43;
81    unknown_libname_2062(4, &v4->pNtHeader_Signature);
82    __linkproc__ LStrCopy(v43);
83    __linkproc__ LStrCmp(v9, &IMAGE_NT_SIGNATURE);
84    if ( v7 )                                   // 判断PE头合法性
85    {
86      (*(void (__fastcall **)(signed int, int))(*(_DWORD *)v5 + 4))(2, (int)&v4->pNtHeader_FileHeader.Machine);
87      (*(void (__fastcall **)(signed int, int))(*(_DWORD *)v5 + 4))(2, (int)&v4->pNtHeader_FileHeader.NumberOfSections);
88      v10 = v4->pDosHeader_e_lfanew;
89      sub_4131E4(v44, (LONG)v45, (PLONG)&v46->This, v47);
90      (*(void (__fastcall **)(signed int, int))(*(_DWORD *)v5 + 4))(
91        4,
92        (int)&v4->pNtHeader_FileHeader.PointerToSymbolTable);
93      (*(void (__fastcall **)(signed int, int))(*(_DWORD *)v5 + 4))(4, (int)&v4->pNtHeader_FileHeader.NumberOfSymbols);// NumberOfSymbols DWORD ? ; 0010h - 符号表中的符号数量（用于调试
94      v11 = v4->pDosHeader_e_lfanew;
95      sub_4131E4(v44, (LONG)v45, (PLONG)&v46->This, v47);
96      (*(void (__fastcall **)(signed int, int))(*(_DWORD *)v5 + 4))(2, (int)&v4->pNtHeader_FileHeader.Characteristics);
97      (*(void (__fastcall **)(signed int, int))(*(_DWORD *)v5 + 4))(2, (int)&v4->pNtHeader_OptionalHeader.Magic);
98      if ( v4->pNtHeader_OptionalHeader.Magic == 0x10B )// IMAGE_OPTIONAL_HEADER; +18h    WORD    Magic;          // 标志字，ROM 映像（0107h）,普通可执行文件（010Bh）
99      {
100       v4->Magic = 1;
101       v12 = v4->pDosHeader_e_lfanew;
102       sub_4131E4(v44, (LONG)v45, (PLONG)&v46->This, v47);
103       (*(void (__fastcall **)(signed int, int))(*(_DWORD *)v5 + 4))(
104         4,
105         (int)&v4->pNtHeader_OptionalHeader.SizeOfInitializedData);// SizeOfInitializedData DWORD? ;0020h所有含已初始化数据的节的总大小
106       v13 = v4->pDosHeader_e_lfanew;
107       sub_4131E4(v44, (LONG)v45, (PLONG)&v46->This, v47);
108       (*(void (__fastcall **)(signed int, int))(*(_DWORD *)v5 + 4))(
109         4,
```

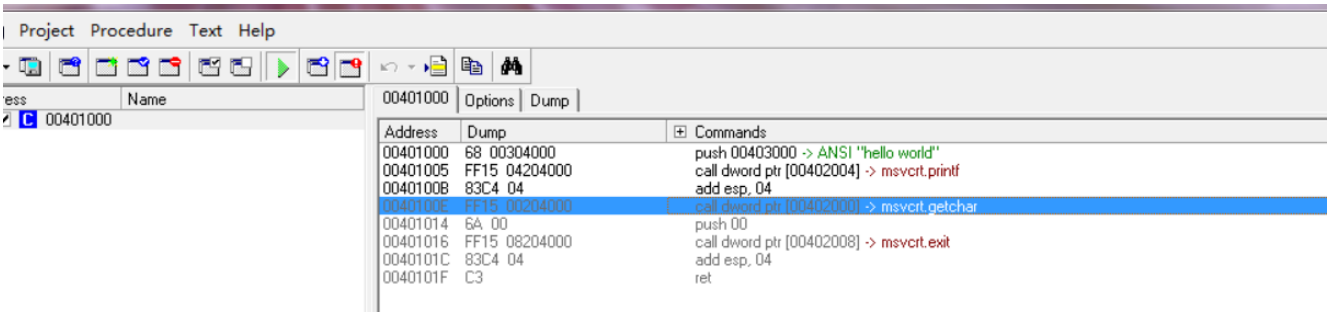
定义的结构如下：

```
00000000 ; 保存要用户加密exe的信息、包括路径、PE结构基本内容
00000000 struct_PEInformation struc ; (sizeof=0x84, align=0x4, mappedto_298)
00000000 This            dd ?
00000004 _prev_node      dd ?                    ; 指向前一个链表结构体，struct_VmpAllData
00000008 Executable      db ?                    ; 判断文件属于PE文件还是ELF文件，0=错误 1=PE 2=ELF
00000009 Magic           db ?                    ; 0=16 1=32 2=64位
00000009                                         ; 根据PE扩展头Maic标志位判断：1、win32普通可执行文件
0000000A gapA            db ?
0000000B                 db ? ; undefined
0000000C pOpenFileName   dd ?                    ; 这是一个指针，指向保存打开要加密exe的全路径
00000010 struc_FileInformation dd ?              ; 指向一个链表结构体，保存打开加密文件的句柄
00000014 Import_NumberOfSections dd ?            ; 第几个区段找到的导入表的
00000018 pNtHeader_Signature dd ?                ; PE头标识
0000001C pNtHeader_FileHeader.Machine dw ?       ; Machine WORD ? ; 0004h - 运行平台
0000001E pNtHeader_OptionalHeader.Subsystem dw ? ; Subsystem WORD ? ;005ch 文件的子系统
00000020 pNtHeader_OptionalHeader.BaseOfDataOrImageBase dd ? ; 根据Magic值作用域不同
00000020                                         ; Magic == 2
00000020                                         ; BaseOfData DWORD ? ;0030h 数据的节的起始RVA
00000020                                         ; Magic == 1
00000020                                         ; ImageBase DWORD ? ;0034h 程序的建议装载地址
00000024 pNtHeader_OptionalHeader.ImageBase dd ? ; 根据Magic值作用域不同
00000024                                         ; Magic == 2
00000024                                         ; ImageBase DWORD ? ;0034h 程序的建议装载地址
00000024                                         ; Magic == 1
00000024                                         ; 不使用，赋值为0
00000028 pNtHeader_OptionalHeader.AddressOfEntryPoint dd ? ; AddressOfEntryPoint DWORD ? ; 0028h 程序执行入口RVA
0000002C pNtHeader_FileHeader.NumberOfSections dw ? ; NumberOfSections WORD ? ; 0006h - 文件的节数目
0000002E SectionSize     dw ?                    ; 区段大小（32位下都是0x28个字节为一组）
00000030 struc_PEInformation_PY30 dd ?           ; 指向一个链表结构体：保存节区信息
00000034 struc_PEInformation_PY34 dd ?           ; 指向一个链表结构体，保存导入表信息
00000038 struc_PEInformation_PY38 dd ?           ; 指向一个链表结构体：保存导出表信息备份
0000003C gap3C           dd ?
00000040 struc_PEInformation_PY40 dd ?           ; 指向一个链表结构体，保存数据目录表信息
00000044 pNtHeader_OptionalHeader.SizeOfImage dd ? ; SizeOfImage DWORD ? ;0050h 内存中整个PE映像尺寸
00000048 pNtHeader_OptionalHeader.SizeOfHeaders dd ? ; SizeOfHeaders DWORD ? ;0054h 所有头＋节表的大小
0000004C pNtHeader_OptionalHeader.DllCharacteristics dd ? ; DllCharacteristics WORD ? ;005eh 总是0
00000050 pNtHeader_OptionalHeader.FileAlignment dd ? ; FileAlignment DWORD ? ;003ch 文件中的节的对齐粒度
00000054 pNtHeader_OptionalHeader.SectionAlignment dd ? ; SectionAlignment DWORD ? ;0038h 内存中的节的对齐粒度
00000058 pDosHeader_e_lfanew dd ?                ; pDosHeader->e_lfanew
0000005C pNtHeader_OptionalHeader.CheckSum dd ?  ; 0x40 DWORD CheckSum;           //PE文件CRC校验和，判断文件是否被修改
00000060 pNtHeader_FileHeader.PointerToSymbolTable dd ? ; PointerToSymbolTable DWORD ? ; 000ch - 指向符号表（用于调试）
00000064 pNtHeader_FileHeader.NumberOfSymbols dd ? ; NumberOfSymbols DWORD ? ; 0010h - 符号表中的符号数量（用于调试）
00000068 pNtHeader_FileHeader.Characteristics dw ? ; Characteristics WORD ? ; 0016h - 文件属性
0000006A pNtHeader_OptionalHeader.Magic dw ?     ; Magic WORD ? ;0018h 107h=ROM Image,10Bh=exe Image
0000006C pNtHeader_OptionalHeader.SizeOfInitializedData dd ? ; SizeOfInitializedData DWORD? ;0020h所有含已初始化数据的节
00000070 pNtHeader_OptionalHeader.BaseOfData1 dd ? ; 根据Magic值作用域不同
00000070                                         ; Magic == 1
00000070                                         ; BaseOfData DWORD ? ;0030h 数据的节的起始RVA
00000070                                         ; Magic == 2
00000070                                         ; 不使用
00000074 struc_PEInformation_PY74 dd ?           ; 指向一个链表结构体：保存Vmp中带包含地址的指令
00000078 Relocation_NumberOfSections dd ?        ; 重定位相关：找不到返回-1，找到了返回重定位在第几个区段
0000007C dword7C         dd ?                    ; 指向一个链表结构体：保存资源表信息
00000080 struc_PEInformation_PY80 dd ?           ; 指向一个链表结构体：保存导出表信息
00000084 struct_PEInformation ends
```

2、获取到壳要的各种API（这里没怎么看）

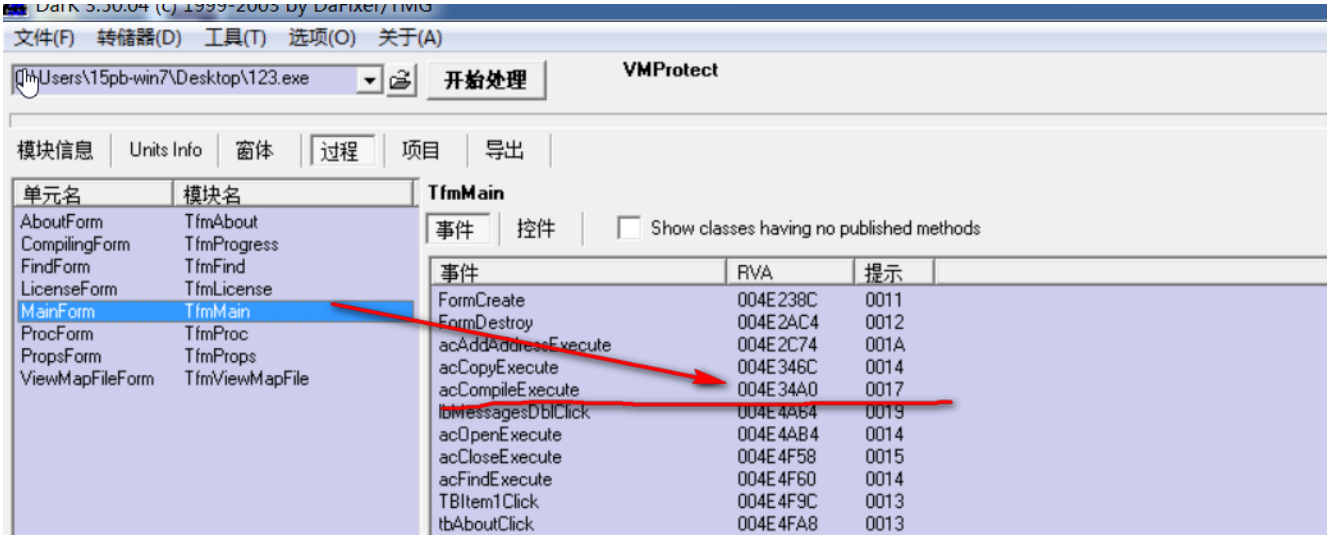
```
98          {
99            v13 = TCollection::GetItem_8(v4[0xC], v12, v11);// 得到节区
00            if ( (*(int (**)(void))(*(_DWORD *)v13 + 0xC))() & 4 )// 得到节区属性
01              break;
02            ++v12;
03            if ( !--v50 )
04              goto LABEL_14;
05          }
06          v57 = v12;
07        }
08LABEL_14:
09        if ( v57 == -1 )
10        {
11          CreateMessageDialog(dword_4ECD74[230], (int)&dword_4ECD00 + 1, 4);
12        }
13        else
14        {
15          sub_4A3240(v64, v11);
16          *(_DWORD *)(v64 + 16) = v4;
17          if ( *((_BYTE *)v4 + 8) == 1 )
18          {
19            v14 = sub_47F7B8(v4[0xD], (int)"kernel32.dll", (int)"GetCurrentThreadId");
20            *(_DWORD *)(v64 + 0x38) = v14;
21            if ( v14 == -1 )
22              *(_DWORD *)(v64 + 0x38) = sub_47F7B8(v4[13], (int)"ntoskrnl.exe", (int)"KeGetCurrentThread");
23          }
24          v63 = 1;
25        }
26      }
27      else
28      {
29        (*(void (__fastcall **)(_DWORD, int *))(*v4 + 16))(*v4, &v39);
```

3、将用户要VM的Opcode进行解析（这个后面有详细说明）



4、如何定位到加密按钮

1、使用DarkDe4.exe



2、分析的设置是保护全关，所以ProtectOptions = 8

```
v68 = a1;
v41 = &savedregs;
v40 = &loc_4E3A06;
__writefsdword(0, (unsigned int)&v39);
ProtectOptions = 8;                     // v1保存开启的多少个（混淆反调试XXX）功能，对应着Options
if ( (unsigned __int8)(*(int (__stdcall **)(unsigned __int32))(**(_DWORD **)(a1 + 0x4B8) + 0xB4))(__readfsdword(0)) )// 是否开启 Debug Mode功能（调试模式）
  ProtectOptions = 9;
if ( (unsigned __int8)(*(int (**)(void))(**(_DWORD **)(v68 + 0x4B0) + 0xB4))() )// 是否开启Hide constants功能(隐藏常量)
  ProtectOptions |= 2u;
if ( (unsigned __int8)(*(int (**)(void))(**(_DWORD **)(v68 + 0x4B4) + 0xB4))() )// 动态创建连接命令
  ProtectOptions |= 4u;
if ( (unsigned __int8)(*(int (**)(void))(**(_DWORD **)(v68 + 0x52C) + 0xB4))() )// 离开虚拟机加密寄存器
  ProtectOptions |= 0x20u;
if ( (unsigned __int8)(*(int (**)(void))(**(_DWORD **)(v68 + 0x4BC) + 0xB4))() )// 检查虚拟机对象的完整性
  ProtectOptions |= 0x10u;
if ( (unsigned __int8)(*(int (**)(void))(**(_DWORD **)(v68 + 0x4D0) + 0xB4))() )// 以此类推
  ProtectOptions |= 0x40u;
if ( (unsigned __int8)(*(int (**)(void))(**(_DWORD **)(v68 + 0x4D8) + 0xB4))() )// 以此类推
{
  v3 = *(_DWORD *)(v68 + 0x4DC);
  TControl::GetText(v2, &v60);
  v5 = sub_47E5C0(v60, v4);
  if ( HIDWORD(v5) )
  {
    if ( SHIDWORD(v5) <= 0 )
      goto LABEL_19;
  }
  else if ( !(_DWORD)v5 )
  {
    goto LABEL_19;
  }
  ProtectOptions |= 0x80u;
}
LABEL_19:
v6 = *(struct_UmpAllData **)(v68 + 0x570);
v6->ProtectOptions = ProtectOptions;          // 保存开启保护功能标志位
v7 = *(_DWORD *)(v68 + 0x4CC);
TControl::GetText(v2, &v59);                  // 保存用户定义的节区名字
__linkproc__ LStrAsg(v8, v59);                // 赋值给v8
```

3、分析关键函数sub_4A3414（最核心的函数）

```
  v6->ProtectOptions = ProtectOptions;          // 保存开启保护功能标志位
  v7 = *(_DWORD *)(v68 + 0x4CC);
  TControl::GetText(v2, &v59);                  // 保存用户定义的节区名字
  __linkproc__ LStrAsg(v8, v59);                // 赋值给v8
  v9 = *(_DWORD *)(v68 + 0x4DC);
  TControl::GetText(v10, &v58);
  *(_QWORD *)&v6->field_40 = sub_47E5C0(v58, v11);// 不知道干嘛的，里面都是字符串操作
  v66 = Vmp_GetCodeLen(*(struct_VmpAllData **)(v68 + 0x570));// 函数：返回要VMP代码的长度
  if ( v66 )                                    // 判断是否是有效长度
  {
    TScreen::SetCursor(0, 0xFFF5, v12);
    v39 = &savedregs;
    v38 = (const CHAR *)&loc_4E368D;
    v37 = (const CHAR *)__readfsdword(0);
    __writefsdword(0, (unsigned int)&v37);
    __linkproc__ LStrCat3(&v57, dword_4ECD74[0xC2], (int)dword_4E3A24);
    sub_4DC750(v57, v66, 0);                    // 不知道干嘛的,尝试NOP掉，不影响实际结果
    v65 = sub_4A3414(*(struct_VmpAllData **)(v68 + 0x570));// 关键
    __writefsdword(0, (unsigned int)v37);
    v39 = (int *)&loc_4E3694;
    sub_4DC804();
    TScreen::SetCursor(0, 0, v13);
    sub_4CEE08(*(_DWORD **)(v68 + 0x4A0));
    (*(void (__cdecl **)(void *, int *))(**(_DWORD **)(v68 + 0x4A0) + 0x378))(v40, v41);
    v14 = TCollection::GetCount_0(*(_DWORD **)(v68 + 0x570)) - 1;
```

5、sub_4A3414函数分析

1、偏移到新区段的起始地址

```
  LODWORD(v57) = &loc_4A394A;
  v56 = (int *)__readfsdword(0);
  __writefsdword(0, (unsigned int)&v56);
  v74 = 0;
  v2 = TCollection::GetCount_0((int)a1);
  if ( v2 - 1 >= 0 )
  {
    v3 = v2;
    v73 = 0;
    do
    {
      v4 = (struc_UserVmpPEInformation *)(*(int (__fastcall **)(int, int))(v1->This + 0x10))(v1->This, v73);
      if ( !v4->Executable )
        TlistArrayClear(v4->struc_CommonPY_38, v5);
      ++v73;
      --v3;
    }
    while ( v3 );
  }
  v66 = (struct_PEInformation *)v1->struct_PEInformation;
  v73 = TCollection::GetCount_0((int)v66->struc_PEInformation_PY30) + 1;// 获取区段个数
  if ( !(v1->ProtectOptions & 0x40) )           // （勾选的保护等级）
    ++v73;
  FirstSectionAddress = (*(int (__stdcall **)(int *))(v66->This + 0x1C))(v56);// 得到第一个区段
  Rva = v73 * v66->SectionSize + FirstSectionAddress;// 偏移到最后一个区段地址
                                                // 1、公式：区段个数*0x28（每个区段占0x28个字节）+第一块区段首地址
                                                // 2、因为加壳的时候需要添加区段
                                                // 3、NT+PE+节区的总大小
  NumberOfRvaAndSizes = 0;
  v65 = dword_4ED21C;
```

2、判断RVA合法性

```
  v65 = dword_4ED21C;
  do
  {
    v9 = (struc_IMAGE_DATA_DIRECTORY *)TList::Get(v66->struc_PEInformation_PY40, NumberOfRvaAndSizes, v7);
    if ( !Vmp_ChenckDirectoryRva_Size(v9) && v9->VirtualAddress < (unsigned int)Rva && NumberOfRvaAndSizes != 0xB )
    {
      v56 = &v64;                               // 走到这里就报错了，提示XXX目录错误之类的
      v62 = *v65;
      v63 = 0xB;
      v10 = dword_4ECD74[0xDC];
      Format(0, (int)&v64);
      LOBYTE(v11) = 1;
      CreateMessageDialog(v64, v11, 4);
      goto LABEL_72;
    }
    ++NumberOfRvaAndSizes;
    ++v65;
  }
  while ( NumberOfRvaAndSizes != 0xF );         // 主要判断数据目录表的RVA是否小于我们前面找到的Rva，如果是就报错，简单的过滤错误工作
```

2、1 设置起始struct_VmpAllDataPY_60结构，构造好push jmp跳转地址

```
  else
  {                                             // 这部分都是判断地址合法性的（待定）
    v27 = v26;
    v73 = 0;
    while ( 1 )
    {
      v28 = (struc_UserVmpPEInformation *)(*(int (__fastcall **)(int, int))(v1->This + 0x10))(v1->This, v73);// 设置struct_VmpAllDataPY_60结构结构，构造好第一组push+j
      if ( v28->Executable )
      {
        if ( !(unsigned __int8)(*(int (__fastcall **)(int, int))(v28->This + 0x18))(
                                 v28->This,
                                 v1->struct_VmpAllDataPY_60) )// 判断地址合法性操作？？？？？？
          break;
      }
      ++v73;
      if ( !--v27 )
        goto LABEL_34;
    }
  }
```

OD视图:

地址	HEX 数据											
01411378	3C	DD	47	00	7C	D7	3F	01	0A	10	40	00	00	00	00	00
01411388	20	10	40	00	00	00	00	00	E8	BD	40	01	06	00	00	00

最终效果视图:

```
00401000    68 ADF2034D     push 0x4D03F2AD
00401005  .- E9 1B480000    jmp HelloASM.00405825
0040100A  .  1C 00          sbb al,0x0
0040100C  .  6E             outs dx,byte ptr es:[edi]
0040100D  .  EF             out dx,eax
0040100E  .  FF15 0020400(  call dword ptr ds:[<&msvcrt.getchar>]   [getchar
00401014  .  6A 00          push 0x0                                [status = 0x0
00401016  .  FF15 0820400(  call dword ptr ds:[<&msvcrt.exit>]      [exit
```

3、内存对齐后的总大小

```
                    v57,
                    HIDWORD(v57));
  GetTotalImageSize = Vmp_GetAlignSize(
                      v32->pNtHeader_OptionalHeader.SizeOfImage,
                      v32->pNtHeader_OptionalHeader.FileAlignment);// GetTotalImageSize = 内存对齐后的总大小
                                                   // v32 + 0x44 =镜像大小（所有节区地址相加以1000对齐）
                                                   // v32 + 0x50 = 对齐的倍数(0x200) 按道理来说应该是按内存对齐0x1000? 这里文件对齐0x200? ? ? ? ?
  v57 = GetTotalImageSize;
  *(_QWORD *)&v40->LODWORD_UserVmpStartAddr = GetTotalImageSize
                                            + *(_QWORD *)&v32->pNtHeader_OptionalHeader.BaseOfDataOrImageBase;
  v40->LODWORD_UserVmpFunctionEndAddr = -1;
  v40->HIDWORD_UserVmpFunctionEndAddr = 0x7FFFFFFF;
  LOBYTE(v40->Characteristics) = 0xA;
  v67 = 0x20;
  v42 = TCollection::GetCount_0((int)v1);
  if ( v42 - 1 >= 0 )                               // 唯一区别，v67的值是否置0
  {
    v44 = v42;
    v73 = 0;
    while ( 1 )
    {
      v45 = (struc_UserVmpPEInformation *)(*(int (__fastcall **)(int, int))(v1->This + 0x10))(v1->This, v73);
      if ( v45->Executable )
      {
        if ( !(v45->Characteristics & 0x20) )// 判断区段保护属性
          break;
      }
      ++v73;
      if ( !--v44 )
        goto LABEL_48;
    }
    v67 = 0;
```

3、1 将内存对齐后的总大小保存到struct_VmpAllDataPY_60结构里，当作壳的OEP也就是Vmp入口

根据大小将地址往后移0x40或则0xC0个字节（这个就是VMCONTEXT的大小）

```
  else
  {
    if ( GetSize_0(v1->struct_PEInformation) == 3 )// 根据IMAGE_OPTIONAL_HEADER：+18h WORD Magic; // 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就是1
      v46 = 0xC0;
    else
      v46 = 0x40;
    if ( v1->Flag > -1 )
      v46 = (v46 + 4) << 8;
    v57 = 0i64;
    *(_QWORD *)&v1->NewSectionVOffset = sub_480BE0(
                                        (struct_VmpAllDataPY_60 *)v1->struct_VmpAllDataPY_60,
                                        v46,
                                        v67 | 8u,
                                        0i64);// 返回新区段(.vmp)的VoffSet
    v73 = v1->Flag > -1;
  }
```

OD视图：

```
地址      HEX 数据
014113B0  3C DD 47 00 7C D7 3F 01 40 50 40 00 00 00 00 00
014113C0  FF FF FF FF FF FF FF 7F 00 00 00 00 0A 00 00 00
```

最终结果：

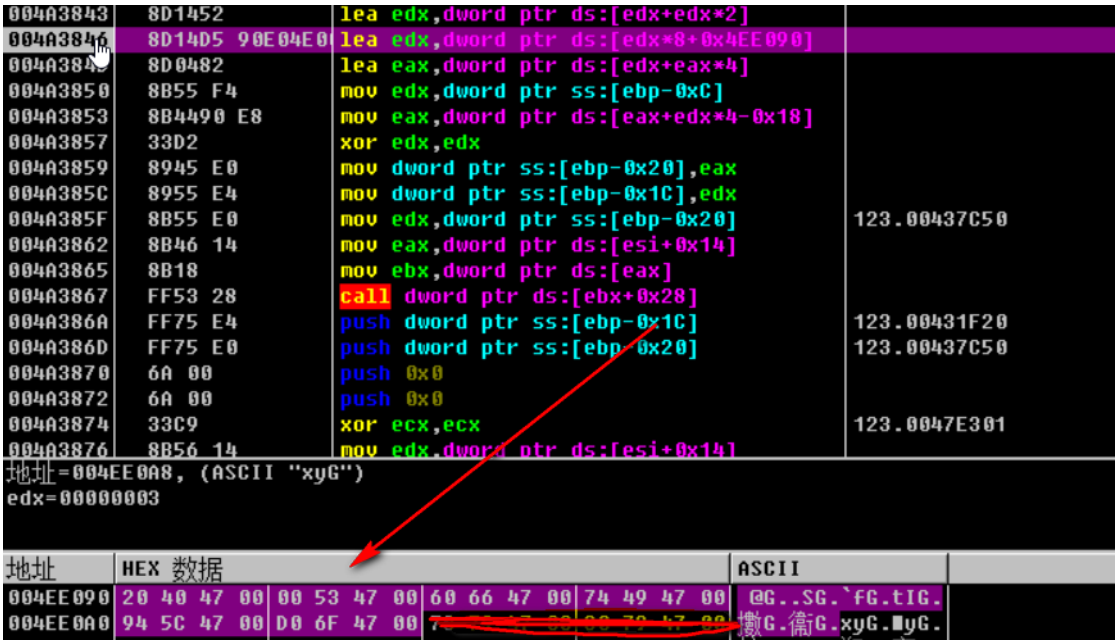
我们发现它是从0x40504B开始，并非0x405040地址开始，这个后续讲解

```
0040503D   3F              aas
0040503E   A0 31586659     mov al,byte ptr ds:[0x59665831]
00405043   36:8808         mov byte ptr ss:[eax],cl
00405046   E9 1C000000     jmp HelloASM.00405067
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
00405054   68 00000000     push 0x0
00405059   8B7424 28       mov esi,dword ptr ss:[esp+0x28]
0040505D   BF 00504000     mov edi,HelloASM.00405000
```

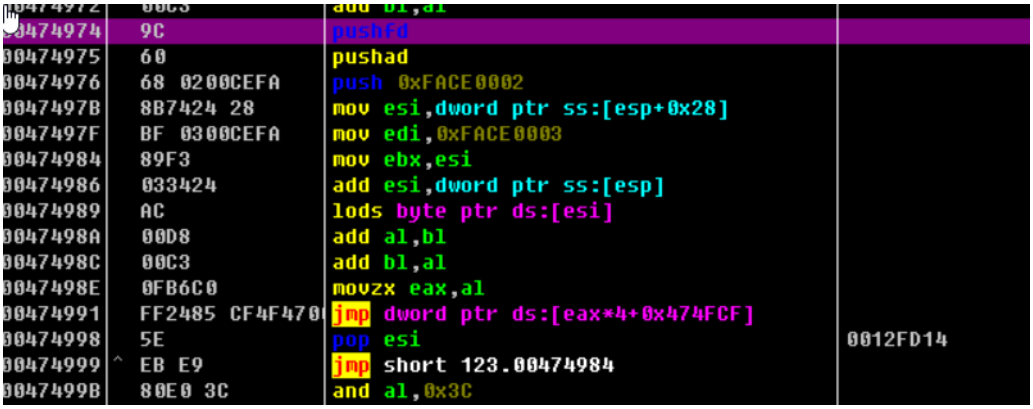
4、根据保护等级选择使用哪个壳模板，并设置区段保护属性

```
{
  if ( GetSize_0(v1->struct_PEInformation) == 3 )// 根据IMAGE_OPTIONAL_HEADER，+18h WORD Magic; // 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就是1
    v46 = 0xC0;
  else
    v46 = 0x40;
  if ( v1->Flag > -1 )
    v46 = (v46 + 4) << 8;
  v57 = 0i64;
  *(_QWORD *)&v1->NewSectionVOffset = sub_480BE0(
                                        (struct_VmpAllDataPY_60 *)v1->struct_VmpAllDataPY_60,
                                        v46,
                                        v67 | 8u,
                                        0i64);// 返回新区段(.vmp)的VoffSet
  v73 = v1->Flag > -1;
}
LOBYTE(v43) = *(_BYTE *)(v1->struct_PEInformation + 9);
v68 = *(void **)(24 * (unsigned __int8)v43 + 0x4EE090 + 12 * ((v1->ProtectOptions & 8) != 0) + 4 * v73 - 0x18);// 0x4EE090:vmp作者设计了handle???，初步判断一共有6组,根据保护等级选
v69 = 0;
(*(void (__fastcall **)(int, int, void *, int))(*(_DWORD *)v1->struct_VmpPEInformation
                                              + 0x28))(// 设置生成Vmp区段并属性
  v43,
  (int)v68,
  v58,
  v59);
v59 = v69;
v58 = v68;                              // 作者设计的VMP handle
v57 = 0i64;
(*(void (__fastcall **)(_DWORD, int))(*(_DWORD *)v1->struct_VmpOpcode + 0x24))(0, v1->struct_VmpPEInformation);// 函数功能：1、分析Opcode  2、构造Esi结构 3、RandIndexArray结构填充
v47 = (struct_VmpOpcode *)v1->struct_VmpOpcode;
v47->Characteristics = v67;
(*(void (__fastcall **)(_DWORD, int))(v47->This + 0x1C))(v47->This, v1->struct_VmpAllDataPY_60);// 函数作用：1、使用RandIndexArray，里面除了保存VMcontext下标，其他都是随机值填充的
v48 = (struct_VmpOpcode *)v1->struct_VmpOpcode;
```

5、壳模板一共有6个



6、我们使用的是474974，到这里这篇章节就完成了



下一篇内容解析Opcode