VMP学习笔记之反汇编引擎学习（三）

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(1)**

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、将struct_VmFunctionAddr结构体称为需要二次解析的称为：特殊Opcode（SetDisassemblyFunction_Address函数填充）

2、将struct_DisassemblyFunction结构体称为：基础Opcode（Vmp_Disassembly填充）

3、壳模板代码和用户加密代码都是调用Vmp_AllDisassembly函数解析，只是保存的位置不一样而已

4、注意struct_DisassemblyFunction是按顺序存放的

例如：

push 1

push 2

push 3

那么push 1 肯定存放在Address[0]

那么push 2 肯定存放在Address[1]

那么push 3 肯定存放在Address[2]

正文：

1、Vmp_AllDisassembly框架详解

```
                if ( !sub_4902C0((int)&savedregs) )
                  goto LABEL_183;
              }
              DisassemblyFunction = (struct_DisassemblyFunction *)(*(int (__cdecl **)(int, int))(v167->This// 每次new出来一个新地址，用来保存Vmp_Disassembly的内容，这个地址保存在((a
                                                                                                  + 0x14))(
                                                                  v146[3],
                                                                  v146[4]);
              Vmp_Disassembly((int)DisassemblyFunction, (int)a1a, (int)&StartAddress, 0);// 里面是反汇编引擎,解析Opcode指令，并将数据保存起来
              if ( a1a->Magic == 2 )
              {
                v146[2] = (int)&savedregs;
                sub_491C0C(
                  DisassemblyFunction->LODWORD_VMP_Address,
                  DisassemblyFunction->HIDWORD_VMP_Address,
                  (int)&savedregs);
              }
              if ( !DisassemblyFunction->Lock_Prefix )
                break;
              SetDisassemblyFunction_Address((int)DisassemblyFunction, 0xD, 0, 0, 0, 0);
            }
            NewOpcode = DisassemblyFunction->VMOpcode;// 取出替换后的Opcode（将系统的Opcode转换成作者自己定义的）
            if ( NewOpcode >= 0x4F )
              break;
            switch ( NewOpcode )              // 部分Opcode需要额外二次处理
            {
              case 2:
              case 4:
              case 5:
              case 6:
              case 7:
              case 0xD:
```

总结：

1、核心部分在于Vmp_Disassembly函数，里面就是解析Opcode指令

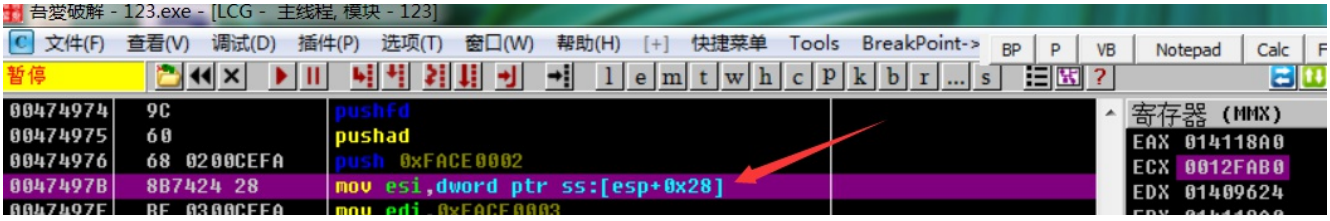
2、部分Opcode需要二次处理

3、解析壳自身代码跟用户Opcode都是调用这个函数

2、Vmp_Disassembly解析Opcode函数分析

0、随便拿条Opcode实例说明：

举例Opcode：0047497B　8B7424 28　　mov esi,dword ptr ss:[esp+0x28]



1、Legacy Prefix（可选）	无
2、Opcode（必须有）	0x8B
3、ModRM（可选）	0x74
4、SIB（可选）	0x24
5、Displacement（可选）	0x28
6、Immediate（可选）	无

1、读取主操作码或则前缀，因为Prefix 与 Opcode 共同占用这个空间

由于 x86/x64 是 CISC 架构，指令不定长。解码器解码的唯一途径就是按指令编码的序列进行解码，关键是第 1 字节是什么？如：遇到 66h，它就是 prefix，遇到 89h，它就是 Opcode。

```
v533 = (struct_DisassemblyFunction *)a1;
v509 = &savedregs;
v508 = &loc_487B83;
v507 = __readfsdword(0);
__writefsdword(0, (unsigned int)&v507);
*(_DWORD *)(a1 + 0x10) = *(_DWORD *)a3;
*(_DWORD *)(a1 + 0x14) = *(_DWORD *)(a3 + 4);
operand_size_override_prefix = 0;             // 66H一操作数大小重载前缀,也可被用作某些指令的强制性前缀.
address_size_override_preifx = 0;             // 67H --- 改变操作数地址模式
segment_override_prefix = 0;                  // segment override prefix: 改变 memory 操作数段选择子
v533->Magic = GetSize_0(v532);                // 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
if ( a4 )
{
  v6 = DateTimeToStr_4(*(_DWORD *)(v532 + 0x38), v533->LODWORD_VMP_Address, v533->HIDWORD_VMP_Address);
  if ( v6 > -1 )
  {
    v7 = *(_DWORD *)(v532 + 0x38);
    v512 = *(_DWORD *)(TCollection::GetItem_3(v5, v6) + 0x18);
    v513 = 11;
    Format(0, (int)&v514);
    __linkproc__ LStrAsg(v8, v514);
    v9 = *(_DWORD *)(v532 + 0x38);
    if ( *(_BYTE *)(TCollection::GetItem_3(v10, v6) + 0x1C) == 2 )
      v533->byte9C = 8;
    else
      v533->byte9C = 2;
    v533->word86 |= 0x200u;
  }
}
while ( 2 )
{
  if ( !v533->VMOpcode )
  {
    v11 = v533;
    v533->LODWORD_VMP_Address1 = *(_DWORD *)v4;// 作者设计的handle当前EIP or 用户Opcode当前EIP
    v11->HIDWORD_VMP_Address1 = *(_DWORD *)(v4 + 4);
    _OpcodeHex = ReadHex_Byte_1_ByReadFile((_DWORD *)v4, (int)&savedregs);// 读出作者设计handle块，每次一个字节。最开始读取的肯定是 前缀 or 主操作码
    v13 = _OpcodeHex;
    switch ( _OpcodeHex )
    {
      case 0u:                                //  ADD r/m8,r8  Add r8 to r/m8
      case 1u:
00081E29 555
```

1、1：GetSize_0函数

函数作用：用来区别读取字节长度

```
// 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
char __usercall GetSize_0@<al>(int a1@<eax>)
{
  char result; // al@2

  if ( *(_BYTE *)(a1 + 9) == 2 )
    result = 3;
  else
    result = 2;                             // //没有去试验ROM映像的文件，这里姑且都认为只返回2
  return result;
}
```

哪里找到赋值的?

Vmp_ReadPEInformation函数，我整个文件导了个遍都只发现赋值为1，没有2?

```
(*(void (__fastcall **)(signed int, int))(*(_DWORD *)v5 + 4))(2, (int)&v4->pNtHeader_OptionalHeader.Magic);
if ( v4->pNtHeader_OptionalHeader.Magic == 0x10B )// IMAGE_OPTIONAL_HEADER: +18h    WORD    Magic;         // 标志字, ROM 映像 (0107h) ,普通可执
{
  v4->Magic = 1;
  v12 = v4->pDosHeader_e_lfanew;
  TStream::SetPosition_SetFilePointer(v44, (LONG)v45, (PLONG)&v46->This, v47);
  (*(void (__fastcall **)(signed int, int))(*(_DWORD *)v5 + 4))(
    4,
    (int)&v4->pNtHeader_OptionalHeader.SizeOfInitializedData);// SizeOfInitializedData DWORD? ;0020h所有含已初始化数据的节的总大小
  v13 = v4->pDosHeader_e_lfanew;
  TStream::SetPosition_SetFilePointer(v44, (LONG)v45, (PLONG)&v46->This, v47);
  (*(void (__fastcall **)(signed int, int))(*(_DWORD *)v5 + 4))(
    4,
    (int)&v4->pNtHeader_OptionalHeader.AddressOfEntryPoint);// AddressOfEntryPoint DWORD ? ;0028h 程序执行入口RVA
  v14 = v4->pDosHeader_e_lfanew;
  TStream::SetPosition_SetFilePointer(v44, (LONG)v45, (PLONG)&v46->This, v47);
  if ( v4->Magic == 2 )
  {
```

1、2：三个比较重要的变量（legacy prefix 的作用）

```
UPX0:00481DDE                   ; 546:   v529 = 0;
UPX0:00481DDE 094 C6 45 F5 00                   mov     [ebp+var_B], 0
UPX0:00481DE2                   ; 547:   v531 = 0;
UPX0:00481DE2 094 C6 45 F7 00                   mov     [ebp+var_9], 0
UPX0:00481DE6                   ; 548:   v530 = 0;
UPX0:00481DE6 094 C6 45 F6 00                   mov     [ebp+var_A], 0
```

1、它们在哪里赋值?

v529 赋值的地方:

```
case 0x66u:                     // 指令前缀：66H—操作数大小重载前缀.也可被用作某些指令的强制性前缀.
    v529 = 1;
```

v531 赋值的地方:

```
case 0x67u:                     // 指令前缀：67H—地址尺寸重载前缀
    v531 = 1;
```

v530 赋值的地方：（大概猜测是Rex前缀，因为没有Magic=2）

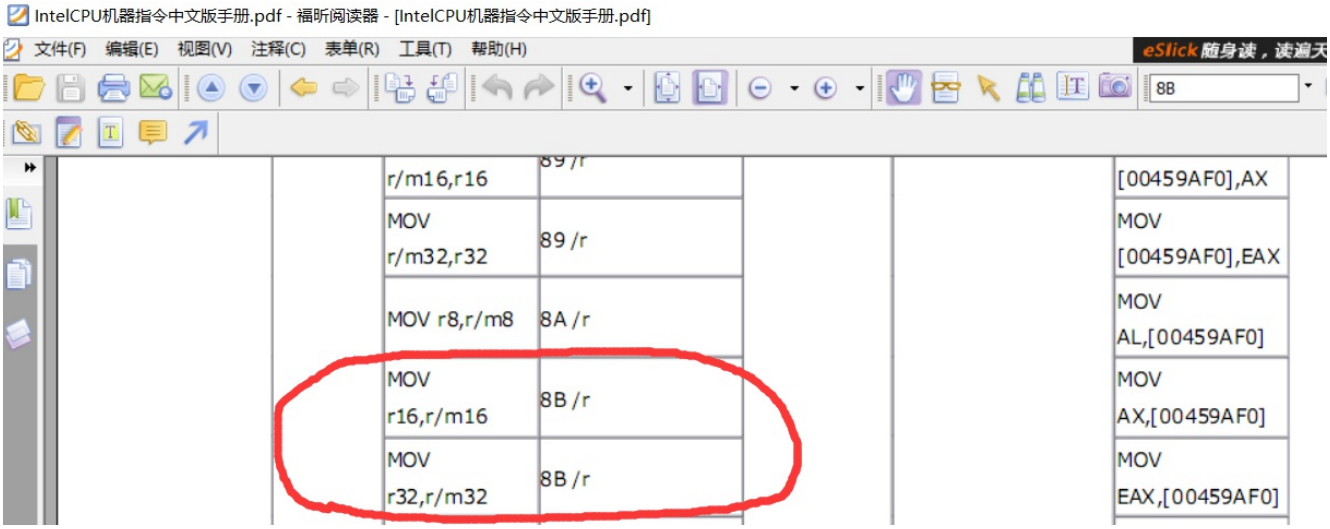
REX前缀是16个编码操作码的集合，包含40H到4FH。这些操作码在IA-32模式和兼容模式中代表有意义的指令。在64位模式中，相同的操作码则代表REX前缀，不再当做单独的指令看待。

```
        case 0x40u:
        case 0x41u:
        case 0x42u:
        case 0x43u:
        case 0x44u:
        case 0x45u:
        case 0x46u:
        case 0x47u:
          if ( v533->Magic == 3 )
          {
            v530 = _OpcodeHex;
          }
          else
          {
            v533->VMOpcode = 41;
            v392 = GetSize(1, (int)&savedregs);
            sub_481A10(v13 & 7, v392, 2, 1, (int)&savedregs);
          }
          continue;
        case 0x48u:
        case 0x49u:
        case 0x4Au:
        case 0x4Bu:
        case 0x4Cu:
        case 0x4Du:
        case 0x4Eu:
        case 0x4Fu:
          if ( v533->Magic == 3 )
          {
            v530 = _OpcodeHex;
          }
```

2、根据switch执行不同的流程解析Opcode

```
case 0x88u:                             // Mov Eb,Gb
case 0x89u:                             // Mov Ev,Gv
case 0x8Au:                             // Mov Gb,Eb
case 0x8Bu:                             // Mov Gv,Ev
  v533->VMOpcode = 3;
  _About_Type = GetSize(_OpcodeHex, (int)&savedregs);
  Register_Or_Memory = (v13 & 2) == 2;  // 区分Mov Gv,Ev或则Mov Ev,Gv
  ModRM = ReadHex_Byte_1_ByReadFile((_DWORD *)v4, (int)&savedregs);
  v426 = v425;
  ModRM_1 = ModRM;
  if ( Register_Or_Memory )
  {
    LOBYTE(v426) = 2;
    Decode_SetReg(ModRM, _About_Type, v426, (int)&savedregs);
    Decode_ModRM(ModRM_1, _About_Type, (_DWORD *)v4, 0, 2, (int)&savedregs);
  }
  else
  {
    Decode_ModRM(ModRM, _About_Type, (_DWORD *)v4, 0, 2, (int)&savedregs);
    v429 = v428;
    LOBYTE(v429) = 2;
    Decode_SetReg(ModRM_1, _About_Type, v429, (int)&savedregs);
  }
  continue;
```

2、1：通过手册我们得知8B对应的是MOV r32,r/m32（**Gv, Ev**）



2、2： Register_Or_Memory = (v13 & 2) == 2这句代码是什么意思？（只针对我举例的，这里只是说明如何找）

我们翻看手册发现了规律是判断目标操作数：G是寄存器或则E是寄存器或者内存操作数

Table A-2. One-Byte Opcodes, Low Nibble 8–Fh

Nibble[1]	8	9	A	B	C	D	E	F
0	OR Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH CS[3]	2-byte opcodes
1	SBB Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH DS[3]	POP DS[3]
2	SUB Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	seg CS[6]	DAS[3]
3	CMP Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	seg DS[6]	AAS[3]
4	DEC[5] eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	POP rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSH Iz	IMUL Gv, Ev, Iz	PUSH Ib	IMUL Gv, Ev, Ib	INSB Yb, DX	INSW/D Yz, DX	OUTSB DX, Xb	OUTSW/D DX, Xz
7	JS Jb	JNS Jb	JP Jb	JNP Jb	JL Jb	JNL Jb	JLE Jb	JNLE Jb
8	MOV Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	Mw/Rv, Sw	LEA Gv, M	MOV Sw, Ew	Group 1a[2] Ev
9	CBW, CWDE, CDQE	CWD, CDQ, CQO	CALL[3] Ap	WAIT FWAIT	PUSHF/D/Q Fv	POPF/D/Q Fv	SAHF	LAHF

Gv, Ev 表示:

（1）两个 Operands 分别是：目标操作数 Gv，源操作数 Ev 或说：frist operand 是 Gv, second operand 是 Ev

（2）Gv 表示：G 是寄存器操作数，v 是表示操作数大小依赖于指令的 Effective Operand-Size，可以是 16 位，32 位以及 64 位。

（3）Ev 表示：E 是寄存器或者内存操作数，具体要依赖于 ModRM.r/m，操作数大小和 G 一致。

4 个字符便可以很直观的表示出：操作数的个数以及寻址方式，更重要的信息是这个 Opcode 的操作数需要 ModRM 进行寻址。

举例子说明：

0047497B 8B7424 28 mov esi,dword ptr ss:[esp+0x28]
0047497B 8A7424 28 mov dh,byte ptr ss:[esp+0x28]

0047497B 887424 28 mov byte ptr ss:[esp+0x28],dh
0047497B 897424 28 mov dword ptr ss:[esp+0x28],esi

v14 =88 Register_Or_Memory =0
v14 =89 Register_Or_Memory =0
v14 =8a Register_Or_Memory =1
v14 =8b Register_Or_Memory =1

总结：

1、这句代码是判断目标操作数是：G是寄存器或则E是寄存器或者内存操作数
2、Ev是包含不确定性具体要依赖于 ModRM.r/m

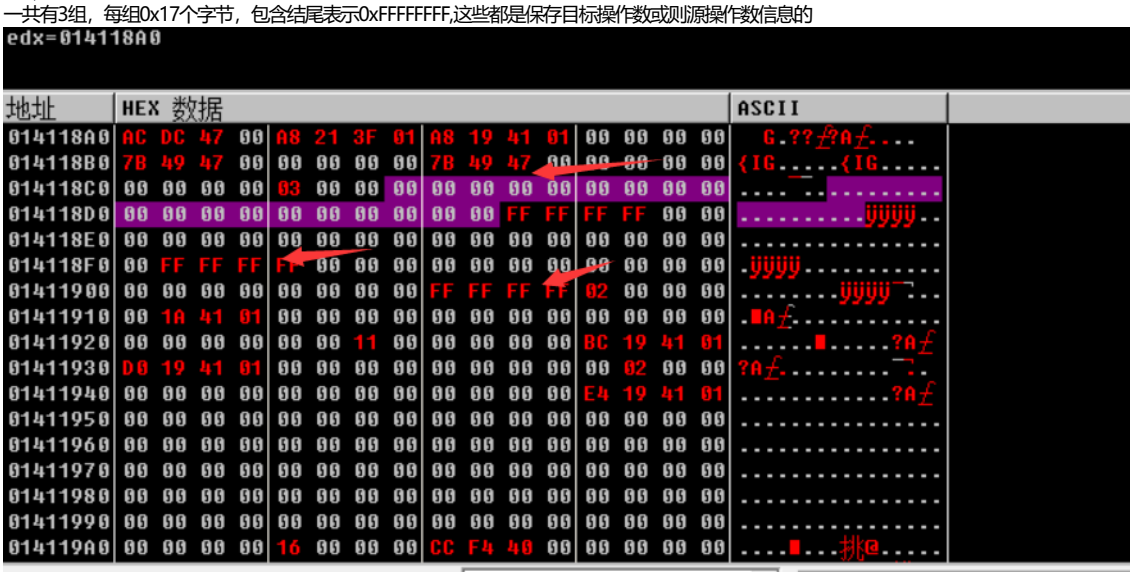
2、3：通过上文描述就可以解释作者为何设计成要区分Register_Or_Memory来区分先执行SetReg跟ModRm
因为假设是Mov Gv,Ev这种类型的：目标操作数是确定Gv，但是源操作数是Ev是包含不确定性具体要依赖于 ModRM.r/m
我们举例的很明显就是MOV r32,r/m32（**Gv, Ev**），目标已知，源带有未知性

```
case 0x88u:                             // Mov Eb,Gb
case 0x89u:                             // Mov Ev,Gv
case 0x8Au:                             // Mov Gb,Eb
case 0x8Bu:                             // Mov Gv,Ev
  v533->VMOpcode = 3;
  _About_Type = GetSize(_OpcodeHex, (int)&savedregs);
  Register_Or_Memory = (v13 & 2) == 2;  // 区分Mov Gv,Ev或则Mov Ev,Gv
  ModRM = ReadHex_Byte_1_ByReadFile((_DWORD *)v4, (int)&savedregs);
  v426 = v425;
  ModRM_1 = ModRM;
  if ( Register_Or_Memory )
  {
    LOBYTE(v426) = 2;
    Decode_SetReg(ModRM, _About_Type, v426, (int)&savedregs);
    Decode_ModRM(ModRM_1, _About_Type, (_DWORD *)v4, 0, 2, (int)&savedregs);
  }
  else
  {
    Decode_ModRM(ModRM, _About_Type, (_DWORD *)v4, 0, 2, (int)&savedregs);
    v429 = v428;
    LOBYTE(v429) = 2;
    Decode_SetReg(ModRM_1, _About_Type, v429, (int)&savedregs);
  }
  continue;
```

2、4: 我们先来分析Decode_SetReg函数

```
1 char __usercall Decode_SetReg@<al>(unsigned __int8 ModRM@<al>, char a2@<dl>, int a3@<ecx>, int a4)
2 {
3   unsigned __int8 v4; // bl@1
4   struct_Disasm *v5; // esi@1
5   char v6; // al@1
6   __int16 v7; // [sp+8h] [bp-8h]@1
7   char v8; // [sp+Eh] [bp-2h]@1
8   unsigned __int8 v9; // [sp+Fh] [bp-1h]@1
9
10  v4 = a3;
11  v8 = a2;
12  v9 = ModRM;
13  v5 = (struct_Disasm *)(*(_DWORD *)(a4 - 4) + 0x17 * sub_4893BC(*(_DWORD *)(a4 - 4)) + 0x27);// 有三个一样的结构体（struct_Disasm），判
14  __linkproc__ SetElem(&v7, v4, 2u);
15  v5->ModRM_mod__Or__Size = v7;                 // 一个大小，由Decode_SetReg参数3决定
16  v5->About_Lval_Byte_Word_Dword = v8;          // GetSize返回值
17  v6 = (v9 >> 3) & 7;                           // reg/opcode域确定寄存器号或者附加的3位操作码.reg/opcode域的用途由主操作码确定.
18  v5->ModRM_Reg_Or_SIB_index = v6;              // 保存ModRm.Reg
19  if ( v4 == 2 )
20  {
21    v6 = a4;
22    if ( *(_BYTE *)(a4 - 0xA) )                 // Rex前缀
23    {
24      v6 = *(_BYTE *)(a4 - 0xA) & 4;
25      if ( v6 == 4 )
26        v5->ModRM_Reg_Or_SIB_index |= 8u;
27    }
28    else if ( !v8 && v5->ModRM_Reg_Or_SIB_index >= 4u )// v8 = 0成立条件：_OpcodeHex&1 == 0
29    {
30      v5->ModRM_Reg_Or_SIB_index &= 3u;
31      v5->ModRM_mod__Or__Size |= 0x100u;
32    }
33  }
34  else if ( v4 == 5 )
35  {
36    v6 = a4;
37    if ( *(_BYTE *)(a4 - 0xA) )                 // Rex前缀
38    {
39      v6 = *(_BYTE *)(a4 - 0xA) & 4;
40      if ( v6 == 4 )
41        v5->ModRM_Reg_Or_SIB_index |= 8u;
42    }
  00081B06 :27
```

一共有3组，每组0x17个字节，包含结尾表示0xFFFFFFFF,这些都是保存目标操作数或则源操作数信息的



v6 = (v9 >> 3) & 7;

首先v9=0x74，继续我们的查表

--	位	描述
ModRM.mod	[7:6]	提供寻址模式: 11 = register !11 = memory
ModRM.reg	[5:3]	提供 register 或者对 Opcode 进行补充
ModRM.r/m	[2:0]	提供 register 或者 memory 依赖于 ModRM.mod

转换成二进制如下：

0x74=01 110 100

结构	描述	内容
ModRM.mod	寻址模式	01
ModRM.reg	寄存器	110
ModRM.r/m	寄存器或则地址	100

很明显v6=ModRM.reg（Esi）

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r)	AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)	AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)	MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)	XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
/digit (Opcode)	0	1	2	3	4	5	6	7
REG =	000	001	010	011	100	101	110	111

2、5：分析Decode_ModRM结构

1、首先解析ModRm判断寻址模式

```
18  char _About_Type; // [sp+1Bh] [bp-1h]@1
19
20  v21 = a3;
21  _About_Type = _About_Type_1;
22  ModRM = a1;
23  v7 = sub_4893BC(*(_DWORD *)(a6 - 4));         // 找到一个组未使用的数组（一共有三组）
24  if ( (ModRM & 0xC0u) < 1 )                    // ModRM.mod[7:6] 提供寻址模式: 11 = register  !11 = memory
25  {                                             // 00 [base] 提供 [base] 形式的 memory 寻址
26    if ( (*(_BYTE *)(a6 - 9) || (ModRM & 7) != 5) && (!*(_BYTE *)(a6 - 9) || (ModRM & 7) != 6) )// ModRM & 7 = ModRM.r/m [2:0] 提供 register 或者 memory 依赖于 ModRM.mod
27    {
28      ModRM.mod = 0xC;
29    }
30    else
31    {
32      ModRM.mod = 0xA;                          // 当Mod.Rm: 101（5）==disp32、110（6）==[ESI]才会执行到这里
33      DisplacementLen = IsDefaultOperandSize(a6);
34    }
35  }
36  else if ( (ModRM & 0xC0) == 0x40 )            // 01 [base + disp8] 提供 [base + disp8] 形式的 memory 寻址
37  {
38    ModRM.mod = 0xE;
39    DisplacementLen = 0;
40  }
41  else if ( (ModRM & 0xC0) == 0x80u )           // 10 [base + disp32] 提供 [base + disp32] 形式的 memory 寻址
42  {
43    ModRM.mod = 0xE;
44    DisplacementLen = IsDefaultOperandSize(a6);
45  }
46  else                                          // 11 register 提供 register 寻址。
47  {
48    __linkproc__ SetElem(&v10, a5, 2u);
49    ModRM.mod = v10;
50  }
51  v9 = (struct_Disasm *)(*(_DWORD *)(a6 - 4) + 0x17 * v7 + 0x27);
```

首先先将ModRM转换下

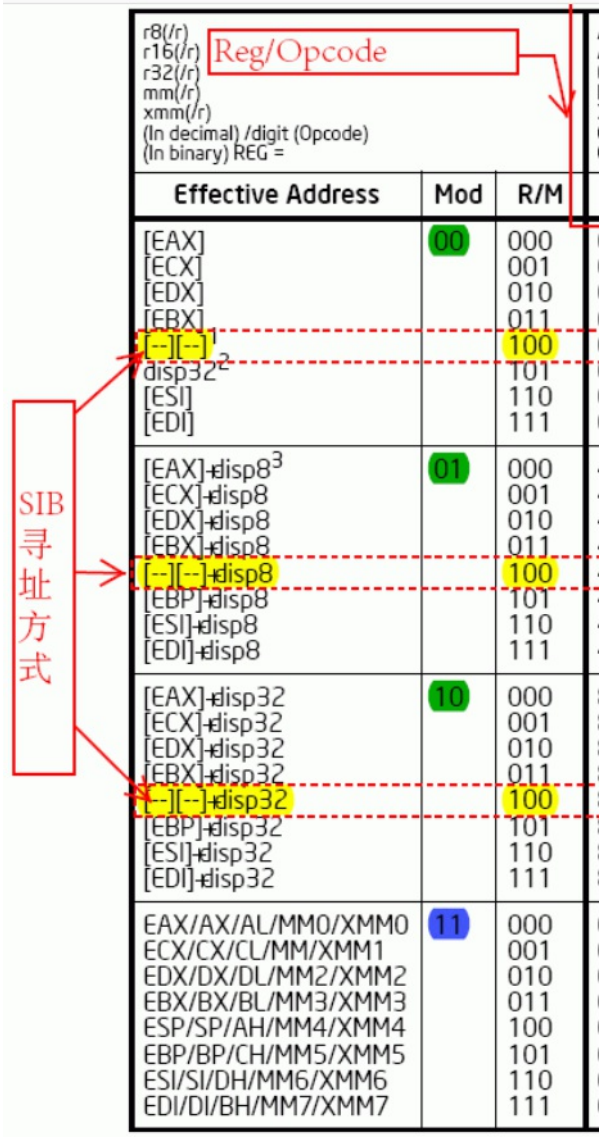
--	位	描述
ModRM.mod	[7:6]	提供寻址模式: 11 = register !11 = memory
ModRM.reg	[5:3]	提供 register 或者对 Opcode 进行补充
ModRM.r/m	[2:0]	提供 register 或者 memory 依赖于 ModRM.mod

转换成二进制如下：

0x74=01 110 100

结构	描述	内容
ModRM.mod	寻址模式	01
ModRM.reg	寄存器	110
ModRM.r/m	寄存器或则地址	100

0x40=01,0x80=10,0xC0=11以此类推



2、判断是否需要SIB寻址方式

R/M==8（100），只有ModRM.mod寻址模式是11（寄存器是不带SIB的）如上图所示

1、第一种这里是ModRM.mod 提供寻址模式: 11 = register（寄存器）

直接保存ModRM.r/m

```
  }
  else                                  // 这里是ModRM.mod 提供寻址模式: 11 = register（寄存器）
  {
    v9->ModRM_Reg_Or_SIB_index_Or_ModRM_rm = ModRM & 7;// ModRM.r/m 寄存器
    if ( a5 == 2 )
    {
      LOBYTE(v10) = a6;
      if ( *(_BYTE *)(a6 - 0xA) )
      {
        LOBYTE(v10) = *(_BYTE *)(a6 - 0xA) & 1;
        if ( (_BYTE)v10 == 1 )
          v9->ModRM_Reg_Or_SIB_index_Or_ModRM_rm |= 8u;
      }
      else if ( !_About_Type && v9->ModRM_Reg_Or_SIB_index_Or_ModRM_rm >= 4u )
      {
        v9->ModRM_Reg_Or_SIB_index_Or_ModRM_rm &= 3u;
        v9->ModRM_mod__Or__Size |= 0x100u;
      }
    }
  }
  return v10;
```

2、第二种情况存在SIB寻址方式

根据上文找到的地址发现是[--][--]",表示有SIB表

SIB结构如下：

--	位	描述
SIB.scale	[7:6]	提供 scale: 00 = 1, 01 = 2, 10 = 4, 11 = 8
SIB.index	[5:3]	提供 index 寄存器
SIB.base	[2:0]	提供 base 寄存器

转换成2进制如下：

0x24=00 101 100

结构	描述	内容
SIB.scale	提供 index 寄存器乘数因子 scale	00
SIB.index	提供 index 寄存器寻址	101
SIB.base	提供 base 寄存器寻址	100

1、读取SIB字节
2、((unsigned __int8)v10 >> 3) & 7; // SIB.index 提供 index 寄存器寻址
3、v9->SIB_base = SIB & 7; // SIB.base 提供 base 寄存器寻址
4、if (v9->ModRM_Reg_Or_SIB_index_Or_ModRM_rm == 4)// SIB.index 提供 index 寄存器寻址是否是none 4 = (100)
5、假设index寄存器！=4就保存SIB.scale 提供 index 寄存器乘数因子 scale
6、判断SIB.base 提供 base 寄存器寻址是否是[*] 5 = (101)

```
  else if ( (ModRM & 7) == 4 )                // 判断是否存在SIB, 因为ModRM.r/m == 4（100）就表示存在
  {
    LOBYTE(v10) = ReadHex_Byte_1_ByReadFile(v21, a6);
    SIB = v10;
    v11 = (unsigned __int8)v10;
    v9->ModRM_Reg_Or_SIB_index_Or_ModRM_rm = ((unsigned __int8)v10 >> 3) & 7;// SIB.index 提供 index 寄存器寻址
    v9->SIB_base = SIB & 7;                   // SIB.base 提供 base 寄存器寻址
    v9->ModRM_mod__Or__Size |= 0x200u;
    LOBYTE(v10) = a6;
    if ( *(_BYTE *)(a6 - 0xA) )
    {
      if ( (*(_BYTE *)(a6 - 0xA) & 1) == 1 )
        v9->SIB_base |= 8u;
      LOBYTE(v10) = *(_BYTE *)(a6 - 0xA) & 2;
      if ( (_BYTE)v10 == 2 )
        v9->ModRM_Reg_Or_SIB_index_Or_ModRM_rm |= 8u;
    }
    if ( v9->ModRM_Reg_Or_SIB_index_Or_ModRM_rm == 4 )// SIB.index 提供 index 寄存器寻址是否是none 4 = (100)
    {
      v9->ModRM_mod__Or__Size &= 0xFFFBu;
    }
    else
    {
      v10 = v11 >> 6;
      v9->SIB_scale = v11 >> 6;               // SIB.scale 提供 index 寄存器乘数因子 scale
    }
    if ( v9->SIB_base == 5 )                  // SIB.base 提供 base 寄存器寻址是否是[*] 5 = (101)
                                              // 1. “[*]”记号表示:若MOD = 00B表示没有基,且带有一个32位的偏移量；否则表示disp8或disp32 + [EBP].即提供如
                                              // 00  [scaled index] + disp32
                                              // 01  [scaled index] + disp8 + [EBP]
                                              // 10  [scaled index] + disp32 + [EBP]
                                              //
    {
      v9->ModRM_mod__Or__Size |= 2u;
      v12 = ModRM & 0xC0;
      v13 = v12 < 1u;
      v14 = v12 - 1;
      if ( v13 )
      {
        DisplacementLen = 2;
        v9->ModRM_mod__Or__Size &= 0xFDFFu;
      }
00081895 ·134
```

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	SS	Index	Value of SIB Byte (in Hexadecimal)							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	89	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

Base

把每一行看作一组，那么这四个组的编码是一样的，有3组的信息冗余。

最后读取Displacement_Immediate

```
    }
    if ( v9->ModRM_mod__Or__Size & 2 )          // 根据ModRM.mod判断以及读取Displacement的长度
    {
      v9->ReadHexLen = *(_BYTE *)(*(_DWORD *)(a6 - 4) + 0x6C);
      if ( DisplacementLen < 1u )
      {
        v9->LODWORD_RestHex_Lval_Displacement_Immediate = (unsigned __int8)ReadHex_Byte_1_ByReadFile(v21, a6);
        v9->HIDWORD_RestHex_Lval_Displacement_Immediate = 0;
      }
      else if ( DisplacementLen == 1 )
      {
        v9->LODWORD_RestHex_Lval_Displacement_Immediate = (unsigned __int16)ReadHex_Byte_2_ByReadFile(v21, a6);
        v9->HIDWORD_RestHex_Lval_Displacement_Immediate = 0;
      }
      else if ( DisplacementLen == 2 )
      {
        v9->LODWORD_RestHex_Lval_Displacement_Immediate = ReadHex_Byte_4_ByReadFile(v21, a6);
        v9->HIDWORD_RestHex_Lval_Displacement_Immediate = 0;
      }
      else
      {
        LODWORD(v16) = ReadHex_Byte_4_ByReadFile(v21, a6);
        *(_QWORD *)&v9->LODWORD_RestHex_Lval_Displacement_Immediate = Dword_Extension_Qword(v16);
        v9->Mod_Rm = 0xA == v9->ModRM_mod__Or__Size;
      }
      LOBYTE(v10) = DisplacementLen;
      v9->Lavl_Btye_Word_Dword = DisplacementLen;
    }
  }
  else                                          // 这里是ModRM.mod 提供寻址模式: 11 = register（寄存器）
```

2、6 还有一些Opcode需要ModRM进行补充的

单纯的一个FF无法表达它到底是CALL、INC、jmp、push需要ModRm辅助的，具体看ModRm.Reg

```
case 0xFFu:
  v501 = ReadHex_Byte_1_ByReadFile((_DWORD *)v4, (int)&savedregs);
  switch ( ((unsigned int)(unsigned __int8)v501 >> 3) & 7 )// ModRM.reg [5:3] 提供 register 或者对 Opcode 进行补充
  {
    case 0u:                            // INC r/m32(16) FF/0  Mod.reg=0的
      v533->VMOpcode = 0x29;
      _About_Type = GetSize(1, (int)&savedregs);
      break;
    case 1u:                            // DEC r/m32(16) FF/1  Mod.reg=1的
      v533->VMOpcode = 0x2A;
      _About_Type = GetSize(1, (int)&savedregs);
      break;
    case 2u:                            // CALL r/m32(16) FF/2
      v533->VMOpcode = 0xB;
      _About_Type = GetSize_1((int)&savedregs);
      break;
    case 3u:                            // CALL m16:16(32) FF/3
      v533->VMOpcode = 0xB;
      _About_Type = IsDefaultOperandSize((int)&savedregs);
      break;
    case 4u:                            // jmp r/m32(16) FF/4
      v533->VMOpcode = 0xC;
      _About_Type = GetSize_1((int)&savedregs);
      break;
    case 5u:                            // jmp m16:16(32) FF/5
      v533->VMOpcode = 0xC;
      _About_Type = IsDefaultOperandSize((int)&savedregs);
      break;
    case 6u:                            // push r/m32(16) FF/6
      v533->VMOpcode = 1;
      _About_Type = GetSize_1((int)&savedregs);
      break;
    case 7u:                            // 不存在FF/7 错误的opcode，需要重新读取过
      v533->VMOpcode = 0x12A;
      break;
    default:
      break;
  }
  if ( v533->VMOpcode != 0x12A )
    Decode_ModRM(v501, _About_Type, (_DWORD *)v4, 1, 2, (int)&savedregs);
```

2、7 Vmp_Disassembly使用的结构体如下：

所有解析后的数据都保存在struc_SaveAllDisasmFunData结构里面

```
00000000 ; 保存内容：
00000000 ; 1、解析后Opcode信息
00000000 ; 2、作者设计VmpHandle开始和结束地址
00000000 ; 3、壳的入口
00000000 struct_VmpOpcode struc ; (sizeof=0x3DC, align=0x4, mappedto_291)
00000000 This            dd ?
00000004 _prev_node      dd ?                    ; 指向一个链表结构体：指向struct_VmpAllData
00000008 struc_SaveAllDisasmFunData dd ?         ; 保存所有解析后的Opcode函数信息
0000000C Magic           dd ?                    ; 数组排列乱序：使用就是1，未使用就是0。
00000010 struc_SavePartDisasmFunData1 dd ?       ; 保存特殊解析后的Opcode信息,专门JmpAddrHandle函数
00000014 gap14           dd ?                    ; 兼容64位
00000018 VmpStubStart    dd ?                    ; Vmp壳入口
0000001C dword1C         dd ?                    ; 兼容64位
00000020 StartVMHandleAddress dd ?               ; 作者设计VM_Handle开始地址
00000024 gap24           dd ?                    ; 兼容64位
00000028 EndVMHandleAddress dd ?                 ; 作者设计VM_Handle结束地址
0000002C gap2C           dd ?                    ; 兼容64位
00000030 dword30         dd ?
00000034 gap34           dd ?
00000038 struc_CommonPY_38 dd ?                  ; 不知道干嘛的
0000003C Executable      db ?
0000003D Characteristics db ?                    ; 区段保护属性
0000003E field_3E        db ?
0000003F field_3F        db ?

0000 ; （基础版）保存所有解析后的Opcode信息，这些信息都以为数组形式存放
0000 struc_SaveAllDisasmFunData struc ; (sizeof=0x14, mappedto_292)
0000 field_0         dd ?
0004 ArrayAddress    dd ?                        ; 这是一个指针指向一个数组，这个数组保存所有struct_DisassemblyFunction函数地址
0008 ArrayNumber     dd ?                        ; 数组个数
000C field_C         dd ?
0010 field_10        dd ?
0014 struc_SaveAllDisasmFunData ends
0014
```

```
0000000
0000000 ; 保存所有解析Opcode函数的结构体
0000000 struct_DisassemblyFunction struc ; (sizeof=0x128, align=0x2, mappedto_265)
0000000 This            dd ?
0000004 _prev_node      dd ?                    ; 指向struct_VmpOpcode结构体
0000008 struc_EsiBytecode dd ?                  ; 保存伪代码（Esi）
000000C field_C         dd ?
0000010 LODWORD_VMP_Address dd ?                ; 壳的当前EIP
0000014 HIDWORD_VMP_Address dd ?                ; 兼容64位
0000018 LODWORD_VMP_Address1 dd ?               ; 要VM的地址，后面就是VMPHandle块的地址
000001C HIDWORD_VMP_Address1 dd ?               ; 兼容64位
0000020 Lock_Prefix     dw ?                    ; 锁定前缀
0000022 Repeat_Prefix   dw ?                    ; 重复前缀
0000024 VMOpcode        dw ?                    ; 替换后的Opcode，例如你原始的是90 替换成作者设计的66
0000026 Segment_Override_Prefix db ?            ; 段重载前缀:2EH—CS=1、36H—SS=2以此类推一共6个
0000027 First           struct_Disasm ?         ; Vmp_Disassembly函数填充内容（第X组）
000003E Second          struct_Disasm ?
0000055 Third           struct_Disasm ?
000006C ReadHexLen      dd ?                    ; 读取Opcode长度，每次执行ReadHex函数便会累加
0000070 ReadHexAddress  dd ?                    ; 读到的Hex地址，这是一个地址指向的内容是读取到的OPcode
0000074 field_74        dd ?
0000078 LODWORD_PushEsiBytecode dd ?            ; 字节码地址（伪代码）
000007C HIDWORD_PushEsiBytecode dd ?
0000080 field_80        dd ?                    ; 未知
0000084 Flag            dw ?                    ; 1、SetDisassemblyFunction_84_88函数赋值
0000084                                         ; 2、如果是第一行Opcode信息Vmp_DisposeUserSpecialOpcode会执行 | 2操作
0000086 word86          dw ?                    ; 一个标志，具体作用待定----------,默认是0x11
0000088 struct_VmFunctionAddr dd ?              ; 1、解析call
0000088                                         ; 2、解析JmpAddr
000008C struct_DisassemblyFunctionPY_8C dd ?    ; 不知道干嘛的,里面保存部分的struc_80结构
0000090 field_90        dd ?                    ; ???????????
0000094 JmpAddr_HandlBuffer dd ?                ; 所有Vmp_Handle模块，指向保存JmpAddrHandl块地址的字符串例如：dd 0047499B
0000098 JmpAddr         dd ?                    ; 当前解析的地址：CASE <- 00474991  (00474991    FF2485 CF4F4700 jmp dword ptr ds:[eax*4+0x474FCF]
000009C byte9C          db ?                    ; 一个标志，具体作用待定--------
000009D Magic           db ?                    ; 1、win32普通执行文件值=1，返回2，作用是区别长度
000009D                                         ; 2、其他情况未测试出来
000009E field_9E        db ?
000009F                 db ? ; undefined
00000A0 struc_84        dd ?                    ; 指向struc_84结构
00000A4 field_A4        dd ?                    ; 值不知道怎么用法的,SetDisassemblyFunction_A4_A8赋值
00000A8 struct_DisassemblyFunctionPY_A8 dd ?    ; 保存struc_77结构
00000AC struct_DisassemblyFunctionPY_AC dd ?    ; 不知道什么情况下才成立，（未分析）
0 ; ---------------------------------------------------------------------------
0
0 ; 保存要解析Opcode的基本信息
0 struct_Disasm   struc ; (sizeof=0x17, mappedto_278) ; XREF: sub_4A063C/r
0                                       ; struct_DisassemblyFunction/r ...
0 About_Lval_Byte_Word_Dword db ?       ; GetSize的返回值，或则固定值
1 ModRM_mod__Or__Size dw ?              ; XREF: sub_4A063C+599/r
1                                       ; sub_4A063C:loc_4A0C1D/r ... ; 当它是G（寄存器）== 2，否则==1
1                                       ; 1、ModRM.mod 提供寻址模式: 11 = register（寄存器）  11 != memory（地址）
1                                       ; 2、一个大小？？？？，由Decode_SetReg参数3决定
3 ModRM_Reg_Or_SIB_index_Or_ModRM_rm db ? ; XREF: sub_4A063C+5AC/r
3                                       ; sub_4A063C+5C7/r ... ; 当它是G（寄存器）== 2，否则==1
3                                       ; 1、存在SIB时SIB.index 提供 index 寄存器寻址，否则就是ModRM_rm寄存器
3                                       ; 2、保存ModRm.Reg
4 SIB_base        db ?                  ; 存在SIB时SIB.base 提供 base 寄存器寻址,否则就是ModRM_reg寄存器（是否保存ModRM_Reg内容待定）
5 SIB_scale       db ?                  ; SIB.scale 提供 index 寄存器乘数因子 scale
6 ReadHexLen      db ?                  ; 读取Opcode长度，每次执行ReadHex函数便会累加，记住偏移量(可选)以及立即数(可选)长度不计算在内
7 LODWORD_RestHex_Lval_Displacement_Immediate dd ? ; Displacement（可选），如果不存在ModRm结构时候。这里就保存内容
B HIDWORD_RestHex_Lval_Displacement_Immediate dd ? ; 兼容64位?
F Lavl_Btye_Word_Dword db ?             ; 根据DisplacementLen判断读几个字节，GetSize的返回值
0 IsAddressing    db ?                  ; 不知道作用，参数：a4
1 Mod_Rm          db ?                  ; 它成立的条件是：Mod.Rm：101==disp32 or 110==[ESI]
2 About_RegType_8_16_32 db ?            ; 读取长度
3 Tag             dd ?                  ; 结尾标识,FFFF
7 struct_Disasm   ends
7
```

3、特殊Opcode解析

一般例如jmp call都是需要二次解析的

```
    }
    NewOpcode = DisassemblyFunction->VMOpcode;// 取出替换后的Opcode（将系统的Opcode转换成作者自己定义的）
    if ( NewOpcode >= 0x4F )
      break;
    switch ( NewOpcode )                // 部分Opcode需要额外二次处理
    {
      case 2:
      case 4:
      case 5:
      case 6:
      case 7:
      case 0xD:
      case 0xE:
      case 0xF:
      case 0x10:
      case 0x11:
      case 0x13:
      case 0x14:
```

0、随便拿条命令举例说明:

jmp dword ptr ds:[eax*4+0x474FCF] 这一条命令就是跳到不同的Handle块执行

jmp它Opcode是FF所以对应的VMOpcode=0xC

```
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
0047498E   0FB6C0           movzx eax,al
00474991   FF2485 CF4F470   jmp dword ptr ds:[eax*4+0x474FCF]
```

根据跳转类型判断，判断E8 E9 EA 近 段间 短跳转

例如像翔种：jmp VMDispatcher就会符合条件

```
        goto LABEL_7;
      case 0xC:
        if ( 2 == DisassemblyFunction->First.ModRM_mod__Or__Size )// jmp 近短跳转
        {
          if ( 2 == DisassemblyFunction->Second.ModRM_mod__Or__Size )// jmp 段间跳转
          {
            DisassemblyFunction->word86 |= 0x80u;
          }
          else
          {
            SetDisassemblyFunction_Address(
              (int)DisassemblyFunction,
              5,
              v166,
              0,
              DisassemblyFunction->First.LODWORD_RestHex_Lval_Displacement_Immediate,
              DisassemblyFunction->First.HIDWORD_RestHex_Lval_Displacement_Immediate);
            a2a[0] = DisassemblyFunction->First.LODWORD_RestHex_Lval_Displacement_Immediate;
```

我们这个是else if (DisassemblyFunction->Magic == 2)

```
    else if ( DisassemblyFunction->Magic == 2 )// 区别长度，win32普通执行文件值
    {
      if ( 0xE == (DisassemblyFunction->First.ModRM_mod__Or__Size & 0xE)// [base + dispX] 形式的 memory 寻址
        && DisassemblyFunction->First.SIB_scale == 2 )// 提供 index 寄存器乘数因子 scale [EXX * 4] or none
      {
        v146[2] = (int)&savedregs;
        *(_QWORD *)v146 = *(_QWORD *)&DisassemblyFunction->First.LODWORD_RestHex_Lval_Displacement_Immediate;
        v102 = TCollection::GetCount_0((int)v167);// 获取解析后保存Opcode信息函数个数
        v104 = (struct_DisassemblyFunction *)GetItem_7((int)v167, v102 - 1, v103);// 获取最后一组Opcode函数，也就是当前的
        sub_4918E8(v104, *(__int64 *)v146, v146[2]);// 主要是针对跳转表jmp dword ptr ds:[eax*4+JumpAddr]这类做特殊处理
        SetDisassemblyFunction_Address(
          (int)DisassemblyFunction,
          0xC,
          0,
          0,
          DisassemblyFunction->First.LODWORD_RestHex_Lval_Displacement_Immediate,
          DisassemblyFunction->First.HIDWORD_RestHex_Lval_Displacement_Immediate);
      }
    }
    else if ( DisassemblyFunction->Magic == 3 )
    {
      v106 = TCollection::GetCount_0((int)v167) - 2;
      if ( v106 > 0
        && 4 == DisassemblyFunction->First.ModRM_mod__Or__Size
```

1、首先解析sub_4918E8函数

```
  v15 = &loc_491A53;
  v14 = __readfsdword(0);
  __writefsdword(0, (unsigned int)&v14);
  a3a = GetSize_0(*(_DWORD *)(a3 - 0xC));       // 返回要读取的大小
LABEL_2:
  Size = Global_Size[a3a];                      // 获取要读取的大小
  v22 = 0;
  while ( 1 )                                   // 循环将JmpAddr解析后保存到起来（struc_SavePartDisasmFunData）
  {
    Displacement = a2 + v22;                    // 偏移到下一组
    v6 = sub_480B54(*(_DWORD *)(v3->This + 0x10), 0, a2 + v22, (a2 + (unsigned __int64)v22) >> 0x20);// 判断是否有重复的,有就退出，没就继续执行
    if ( !((unsigned __int8)((v6 + 1 < 0) ^ __OFADD__(1, v6)) | (v6 == -1)) )
      break;
    v7 = DateTimeToStr_4(*(_DWORD *)(*(_DWORD *)v4 + 0x38), Displacement, SHIDWORD(Displacement));
    if ( !((unsigned __int8)((((v7 + 1) & 0x80000000) != 0) ^ __OFADD__(1, v7)) | (v7 == -1)) )
      break;
    v8 = FindAddress(v3->This, Displacement, SHIDWORD(Displacement));// 搜索Displacement地址，找到返回!=0
    if ( !((unsigned __int8)((((v8 + 1) & 0x80000000) != 0) ^ __OFADD__(1, v8)) | (v8 == -1)) )
      break;
    ++v22;                                      // 循环Size次，每次--，struct_Disasm->Displacement每次+1寻找符合条件的
    if ( !--Size )                              // 没有找到就解析JmpAddr
  if ( !--Size )                                // 没有找到就解析JmpAddr
  {
    v9 = (struct_DisassemblyFunction *)(*(int (__cdecl **)(unsigned int, void *, int *))(*(_DWORD *)v3->This + 0x14))(
                                          v14,
                                          v15,
                                          v16);// New出struct_DisassemblyFunction结构体
    Displacement = sub_494F60(v9, (#290 *)*(_DWORD *)v4, a3a, (int)&a2);// 参数1：[out]填充struct_DisassemblyFunction结构体
                                          // 参数2：struct_PEInformation结构体
                                          // 参数3：要读取的大小
                                          // 参数4：跳转表JmpAddr这个地址是一个数组，保存所有Vmp的Handle块地址
    if ( GetCharacteristics(
           *(struc_PEInformation_PY30 **)(*(_DWORD *)v4 + 0x30),
           (int)v9,
           Displacement,
           SHIDWORD(Displacement)) & 4 )
    {
      SetDisassemblyFunction_Address((int)v9, 0x8, 0, 0, Displacement, SHIDWORD(Displacement));
      *(_DWORD *)(v9->struct_VmFunctionAddr + 0x18) = v23;
      v17 = &v23->LODWORD_VMP_Address;
      v18 = 0x10;
      Format(0, (int)&v19);
      __linkproc__ LStrAsg(v12, v19);
      v9->byte9C = 7;
      v9->word86 |= 0x200u;
      goto LABEL_2;
    }
    v11 = TList::IndexOf((int)v9, v10);
    (*(void (__fastcall **)(_DWORD, int))(*(_DWORD *)v3->This + 12))(*(_DWORD *)v3->This, v11);
    break;
  }
}
__writefsdword(0, v14);
v16 = (int *)&loc_491050;
```

1、1 首先看一看sub_494F60函数

1、保存VmOpcode信息跟Displacement

2、根据大小读取

```
__writefsdword(0, (unsigned int)&v10);
a1->LODWORD_VMP_Address = *(_DWORD *)Displacement;
a1->HIDWORD_VMP_Address = *(_DWORD *)(Displacement + 4);
a1->LODWORD_VMP_Address1 = *(_DWORD *)Displacement;
a1->HIDWORD_VMP_Address1 = *(_DWORD *)(Displacement + 4);
a1->VMOpcode = 0x23;
if ( v18->Magic == 2 )
  a1->Magic = 3;
else
  a1->Magic = 2;
if ( Size < 1u )
{
  jmpAddr_Handle = (unsigned __int8)Vmp_ReadAddressBuff_8((int)v18, (_DWORD *)Displacement, Size);
}
else
{
  switch ( Size )                           // 根据GetSize返回值判断,最终实际读取的长度
  {
    case 1u:                                // byte
      jmpAddr_Handle = (unsigned __int16)Vmp_ReadAddressBuff_16((int)v18, (_DWORD *)Displacement, 1);
      break;
    case 2u:                                // dword
      jmpAddr_Handle = (unsigned int)Vmp_ReadAddressBuff_32(v18, (_DWORD *)Displacement, 2);// 得到JmpAddr函数跳转表Handle地址，指针指向下一组Handle地址
      v14 = &jmpAddr_Handle;
      v15 = 0x10;
      Format(0, (int)&v16);
      __linkproc__ LStrAsg(v6, v16);
      break;
    case 3u:                                // qdword
      jmpAddr_Handle = Vmp_ReadAddressBuff_64((int)v18, (_DWORD *)Displacement);
      v14 = &jmpAddr_Handle;
      v15 = 0x10;
      Format(0, (int)&v13);
      __linkproc__ LStrAsg(v7, v13);        // 将数据填充到struct_DisassemblyFunction->JmpAddr_HandlBuffer
      break;
  }
```

设置ModRM信息

```
v5->First.Tag = DateTimeToStr_0(
                  *(_DWORD *)(*(_DWORD *)(*(_DWORD *)(v5->_prev_node + 4) + 0x10) + 0x74),
                  v5->LODWORD_VMP_Address,
                  v5->HIDWORD_VMP_Address);
v5->First.ModRM_mod__Or_|_Size = 2;
v5->First.About_Lval_Byte_Word_Dword = v4;
v5->First.Lavl_Btye_Word_Dword = v4;
*(_QWORD *)&v5->First.LODWORD_RestHex_Lval_Displacement_Immediate = jmpAddr_Handle;
v8 = v4 & 0x7F;
__linkproc__ DynArraySetLength(
  &v5->ReadHexAddress,
  (int)dword_47CE4C,
  1,
  v5->ReadHexLen + (unsigned __int8)Global_Size[v8]);
Move((char *)&jmpAddr_Handle, (char *)(v5->ReadHexAddress + v5->ReadHexLen), (unsigned __int8)Global_Size[v8]);// 返回要读取的长度
v5->ReadHexLen += (unsigned __int8)Global_Size[v8];
__writefsdword(0, v10);
v12 = (int *)&loc_495128;
__linkproc__ LStrClr(&v13);
__linkproc__ LStrClr(&v16);
return jmpAddr_Handle;
```

v5->First.ModRM_mod__Or__Size = 2;

jmp dword ptr ds:[eax*4+0x474FCF] 就是这种寻址方式

1、2 SetDisassemblyFunction_Address函数解析

```
// 函数作用：
// 填充DisassemblyFunction->Address的内容它是一个地址，该地址指向一个结构体
__int16 __userpurge SetDisassemblyFunction_Address@<ax>(int a1@<eax>, char a2@<dl>, char a3@<cl>, char a4, int Displacement, int HIDWORD_Displacement)
{
  char v6; // ST0A_1@1
  char v7; // ST0B_1@1
  struct_DisassemblyFunction *v8; // ebx@1
  struct_VmFunctionAddr *v9; // eax@1
  __int16 result; // ax@1

  v6 = a3;
  v7 = a2;
  v8 = (struct_DisassemblyFunction *)a1;
  v9 = (struct_VmFunctionAddr *)(*(int (**)(void))(**(_DWORD **)(*(_DWORD *)(a1 + 4) + 0x10) + 0x10))();// new数组元素
  v8->struct_VmFunctionAddr = (int)v9;
  LOBYTE(v9->Magic) = v7;
  v9->FunAddr = Displacement;
  v9->dword14 = HIDWORD_Displacement;          // 兼容64位
  v9->DisassemblyFunction = v8;
  v9->byte24 = v6;
  v9->byte25 = a4;
  result = v8->VMOpcode - 1;
  if ( v8->VMOpcode == 1 || (result = v8->VMOpcode - 3, v8->VMOpcode == 3) )
    v8->word86 |= 0x200u;
  return result;
}
```

if (v8->VMOpcode == 1 || (result = v8->VMOpcode - 3, v8->VMOpcode == 3))成立条件

v533->VMOpcode=1：

case 0xA8: // Test al,imm8

case 0x16u: // push ss

case 0x1Eu: // push ds

case 0x50~0x57: // 50=push rax/r8 51=push rcx/r9以此类推

case 0x68u: // push Imm32/16

等等

v533->VMOpcode=3：

case 0xC7u: // MOV R/M32,IMM32

case 0x20: // and Eb,Gb

case 0x22: // and Gb,Eb

等等

1、3 SetDisassemblyFunction_Address函数设计到的结构体如下：

```
00 ; 它们都使用这个使用指向的结构体，但是内容不同
00 struct_VmFunctionAddr struc ; (sizeof=0x26, align=0x2, mappedto_294)
00 This            dd ?
04 _prev_node      dd ?
08 Magic           dd ?                    ; 1、解析Jmp dword ptr [eax*4+JumpAddr] 值是0xB
08                                         ; 2、解析Jmp VMDispatcher               值是0x5
08                                         ; 3、解析call                           值是0xA
0C DisassemblyFunction dd ?                ; 保存JmpAddr函数表 或则 Jmp VMDispatcher信息
10 FunAddr         dd ?                    ; JmpAddr（Displacement（可选），如果不存在ModRm结构时候。这里就保存内容）
14 dword14         dd ?                    ; 兼容64位
18 FormerDisassemblyFunction dd ?          ; 1、保存原始的Jmp dword ptr [eax*4+JumpAddr]地址Opcode信息
18                                         ; 2、Jmp VMDispatcher 这里就是0
1C CheckDisassemblyFunction dd ?           ; 1、查找匹配的DisassemblyFunction（根据FunAddr查找）
1C                                         ; 2、根据ESIResults值判断是否保存  1保存 0不保存
20 NextDisassemblyFunction dd ?            ; 保存下一条指令的struct_DisassemblyFunction
24 byte24          db ?                    ; 参数a3的值
25 byte25          db ?                    ; 参数a4的值
26 struct_VmFunctionAddr ends
26
00 ; ---------------------------------------------------------------------------
```

2、执行依次SetDisassemblyFunction_Address标记结尾

注意这里参数2就是：0xC

```
SetDisassemblyFunction_Address(// 标记结尾
    (int)DisassemblyFunction,
    0xC,
    0,
    0,
    DisassemblyFunction->First.LODWORD_RestHex_Lval_Displacement_Immediate,
    DisassemblyFunction->First.HIDWORD_RestHex_Lval_Displacement_Immediate);
}
```

```
048915C=123.0048915C

地址      HEX 数据                                          ASCII
14202D8   E8 DB 47 00 64 32 3F 01 0C 00 00 00 28 24 41 01   构G.d2?.....($A.
14202E8   CF 4F 47 00 00 00 00 00 00 00 00 00 00 00 00 00   蟧G.............
```

4、历史遗留问题

1、保存的所有struct_DisassemblyFunction（基础）、struct_VmFunctionAddr（特殊）该如何使用，这个留到后面揭晓

2、v530 赋值的地方：（大概猜测是Rex前缀，因为没有Magic=2）

3、Magic==2满足条件（这个我没仔细跟）