

VMP学习笔记之万用门（七）

前言：

自己学习VMP的经验分享给大家，有错误请指出。

参考资料：

1、名称：**破解vmp程序的关键点**

网址：<https://bbs.pediy.com/thread-82618.htm>

2、名称：**谈谈vmp的爆破**

网址：<https://bbs.pediy.com/thread-224732.htm>

3、名称：**一个VMP1.20程序的伪指令总结**

网址：<https://bbs.pediy.com/thread-54535.htm>

正文：

1、万用门实现逻辑指令

理论知识：

vmp里面只有1个逻辑运算指令 not_not_and 设这条指令为P

$P(a,b) = \sim a \ \& \ \sim b$

这条指令的神奇之处就是能模拟 not and or xor 4条常规的逻辑运算指令
怕忘记了，直接给出公式，后面的数字指需要几次P运算

$\text{not}(a) = P(a,a) \ 1$

$\text{and}(a,b) = P(P(a,a),P(b,b)) \ 3$

$\text{or}(a,b) = P(P(a,b),P(a,b)) \ 2$

$\text{xor}(a,b) = P(P(P(a,a),P(b,b)),P(a,b)) \ 5$

2、模拟CMP减法指令

理论知识：

参考S大的图

原式: `cmp A,B`

`cmp` 可看成 `sub`
 $\Rightarrow A-B = -(-A+B)$
 $= \text{vm_not}(\text{vm_not}(A) \text{ vm_add } B)$

观念:

1. `vm_not` 是用 `vm_nor` 实现的, 而 `vm_nor` 是用汇编 `AND` 指令实现的
2. 汇编 `AND` 指令会影响 `O'S'Z'P'C` (`AND`指令会将`O'C`清为0)
3. 汇编 `SUB'AND` 这二指令所影响的标志位是一样的, 皆影响 `O'S'Z'A'P'C`

解决方案:

思考一下 `-A+B` 跟原式 `A-B` 有什么共同点及不同点呢?

答案是 `S'Z` 不同, 其它 `A'C'O'P` 皆同.

(`A-B` 跟 `A+B` 怎可能 `S'Z` 会一样呢? 对不对)

所以 `S'Z` 不要取, 其它全取, 这就是为何要 `AND 815` 的原因,

`O SZ A P C` \leftarrow `ADD` 指令会影响到的标志位
遮罩 `815 = 1000 0001 0101`

最后的 `vm_not`:

我们知道 `vm_not` 是由汇编的 `AND` 所实现的, 而 `AND` 指令会影响 `O'S'Z'P'C`

思考一下..汇编 `SUB A,B` 的 `S'Z` 会与 `AND A,B` 的 `S'Z` 一样, 所以

`O SZ P C` \leftarrow `AND` 指令会影响到的标志位 (`O'C`清为0)
遮罩 `7EA = 0111 1110 1010`
`DIT SZx x 1`
即可取到 `S'Z` (于 `CMP` 指令, 我们并不关心 `D'I'T`)

二数相加就实现出 `cmp` 后的标志位了

总结:

通过上述我们可以将操作步骤分为两个部分:

1、模拟`sub`减法指令

$\sim(\sim a+b)$

2、模拟`Eflag`标志位

`int Eflag1 = (~a+b)`

`int Eflag2 = ~(~a+b)`

`int Eflag = (Eflag1 and 0x815) + (Eflag2 and ~0x815)`

通过试验证明公式:

随便改个demo去加密看看VMP是如何实现的



假设: `eax = 77523C33 ebx = 7FFD3000`

模拟出正确结果: `eax-ebx=F7550C33`

模拟出正确标志: `Eflag287`

2、1 模拟减法指令

公式: $\sim(\sim a+b)$

假设: `eax = 0x77523C33 ebx = 0x7FFD3000`

模拟出正确: `eax - ebx = 0xF7550C33`

0~2 行模拟出 $\sim a$ `a = 0x88ADC3CC`

3 行 $\sim(A+B)$ $(\sim A+B) = 0x08AB03CC$ `Eflag1 = 0x207`

4 行保存`Eflag1`

5~6 行取出 $(\sim A+B)$ 的结果

7 行 $\sim(\sim a+b)$ $\sim(\sim a+b) = 0xF7550C33$ `Eflag2 = 0x286`

8 行保存Eflag2

剩下的cmp跟sub唯一区别: sub保留结果、cmp不保留结果

```
case 0x34: // sub
case 0x64: // cmp
    v11 = a1a->First.About_Lval_Byte_Word_Dword;
    Ump_UseDisamaStruct(1, a5[0], v11, (int)&savedregs); // UMP_PUSH_CONTEXT 先取ebx
    sub_497788(0, (int)&savedregs); // 里面就是实现了~a(not(a)) = P(a,a) 1)
    Ump_SetEsiStruct(a1a, 4, 0, -1, 0, 1i64, v11); // UMP_ADDF
    Ump_SetEsiStruct(a1a, 2, 2, -1, 0, 0x10i64, a1a->Magic); // VM_PUSH_CONTEXT 保存Eflag1
    Ump_SetEsiStruct(a1a, 1, 2, -1, 0, 4i64, a1a->Magic); // VM_PUSH_ESP
    Ump_SetEsiStruct(a1a, 1, 3, -1, 0, 3i64, v11); // VM_MOV_ESA_TO_B
    Ump_SetEsiStruct(a1a, 0x8, 0, -1, 0, 1i64, v11); // VM_NOR
    Ump_SetEsiStruct(a1a, 2, 2, -1, 0, 0x13i64, a1a->Magic); // VM_PUSH_CONTEXT 保存Eflag2
    if (a1a->UMOpcode == 0x64) // cmp指令不保存结果
        Ump_SetEsiStruct(a1a, 2, 2, -1, 0, 0x14i64, v11);
    else // sub指令保存结果
        Ump_UseDisamaStruct(0, 1, v11, (int)&savedregs);
    GetEFlag(0x815u, (int)&savedregs); // 计算EFlag的地方
    break;
```

总结:

我们得到两个关键的数据分别是:

int Eflag1 = (~a+b) 207

int Eflag2 = ~(~a+b) 286

2、2 模拟Eflag标志位

公式:

int Eflag1 = (~a+b)

int Eflag2 = ~(~a+b)

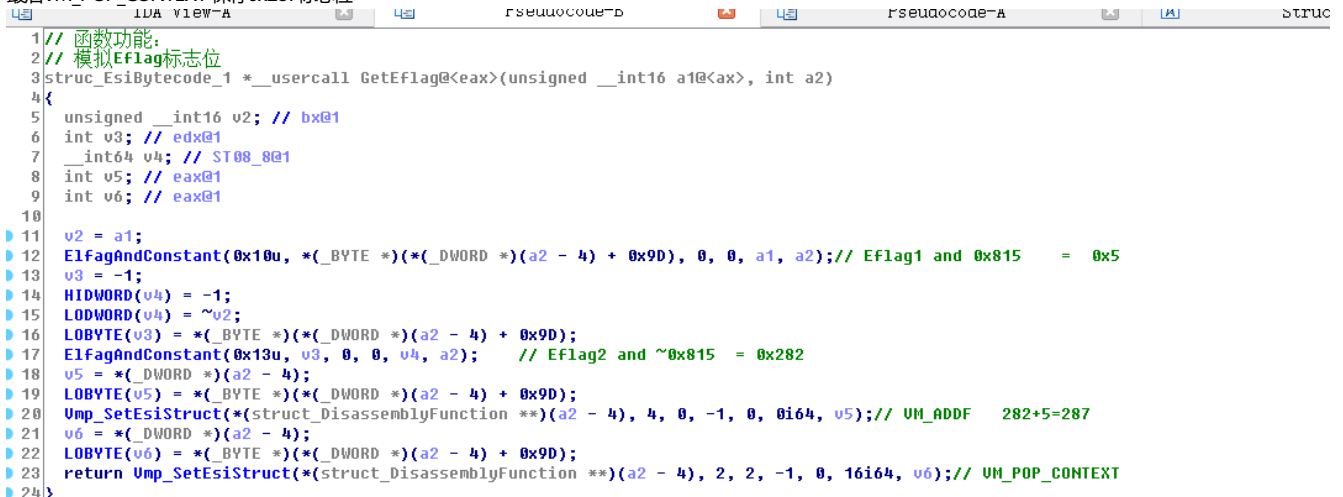
int Eflag = (Eflag1 and 0x815) + (Eflag2 and ~0x815)

第一个EflagAndConstant模拟出 Eflag1 and 0x815 = 0x5

第二个EflagAndConstant模拟出 Eflag2 and ~0x815 = 0x282

然后就VM_ADDF = 0x282+0x5= 0x287

最后VM_POP_CONTEXT 保存0x287标志位



```
1 // 函数功能:
2 // 模拟Eflag标志位
3 struc_EsiBytecode_1 * __usercall GetEflag@<eax>(unsigned __int16 a1@<ax>, int a2)
4 {
5     unsigned __int16 v2; // bx@1
6     int v3; // edx@1
7     __int64 v4; // ST08_8@1
8     int v5; // eax@1
9     int v6; // eax@1
10
11     v2 = a1;
12     EflagAndConstant(0x10u, *((_BYTE *)((_DWORD *) (a2 - 4) + 0x9D)), 0, 0, a1, a2); // Eflag1 and 0x815 = 0x5
13     v3 = -1;
14     HIWORD(v4) = -1;
15     LODWORD(v4) = ~v2;
16     LOBYTE(v3) = *((_BYTE *)((_DWORD *) (a2 - 4) + 0x9D));
17     EflagAndConstant(0x13u, v3, 0, 0, v4, a2); // Eflag2 and ~0x815 = 0x282
18     v5 = *((_DWORD *) (a2 - 4));
19     LOBYTE(v5) = *((_BYTE *)((_DWORD *) (a2 - 4) + 0x9D));
20     Ump_SetEsiStruct((struct_DisassemblyFunction **) (a2 - 4), 4, 0, -1, 0, 0i64, v5); // VM_ADDF 282+5=287
21     v6 = *((_DWORD *) (a2 - 4));
22     LOBYTE(v6) = *((_BYTE *)((_DWORD *) (a2 - 4) + 0x9D));
23     return Ump_SetEsiStruct((struct_DisassemblyFunction **) (a2 - 4), 2, 2, -1, 0, 16i64, v6); // VM_POP_CONTEXT
24 }
```

细看第一个EflagAndConstant函数 (就是模拟and指令) 第二个同理

VM_PUSH_CONTEXT 取 Eflag1 = 0x207

VM_PUSH_CONTEXT 取 Eflag1 = 0x207

VM_NOR ~Eflag1 = 0xFFFFFFFFFFDF8

VM_PUSH_IMM ~815 = 0x7EA

VM_NOR ~Eflag1 and ~0x7EA = 207 and 815

转了一圈发现是模拟and(a,b) = P(P(a,a),P(b,b)) 3

那么第一个 (Eflag1 and 0x815) = 0x207 and 0x815 = 0x7

```

1 struct_EsiBytecode_1 * __userpurge ElfAgAndConstant@<eax>(unsigned __int8 a1@<a1>, int a2@<edx>, unsigned __int8 a3@<c1>, char a4, __int64 a5, int a6)
2 {
3     int v6; // ebx@1
4     unsigned __int8 v8; // [sp+6h] [bp-2h]@1
5     unsigned __int8 v9; // [sp+7h] [bp-1h]@1
6
7     v8 = a3;
8     v6 = a2;
9     v9 = a1;
10    if ( a4 )
11    {
12        Ump_SetEsiStruct(*(struct_DisassemblyFunction **)(a6 - 4), 1, 8, -1, 0, a1, a2);
13        Ump_SetEsiStruct(*(struct_DisassemblyFunction **)(a6 - 4), 1, 8, -1, 0, v9, v6);
14    }
15    else
16    {
17        Ump_SetEsiStruct(*(struct_DisassemblyFunction **)(a6 - 4), 1, 2, -1, 0, a1, a2);
18        Ump_SetEsiStruct(*(struct_DisassemblyFunction **)(a6 - 4), 1, 2, -1, 0, v9, v6);
19    }
20    Ump_SetEsiStruct(*(struct_DisassemblyFunction **)(a6 - 4), 0x0, 0, -1, 0, 0i64, v6);
21    Ump_SetEsiStruct(*(struct_DisassemblyFunction **)(a6 - 4), 1, 1, -1, 2, ~a5, v6);
22    return Ump_SetEsiStruct(*(struct_DisassemblyFunction **)(a6 - 4), 0x0, 0, -1, 0, v8, v6);
23 }

```

2、3 题外话

感谢群里小零大佬提供0x811与0x815等价

811其实就是

OF, AF, CF

而

PF, ZF, SF不管是add还是sub, or, and计算出来都是一样的

通过nor计算出PF, ZF, SF和

add出来的标志位合并一下就好了

3、模拟JXX指令

理论知识:

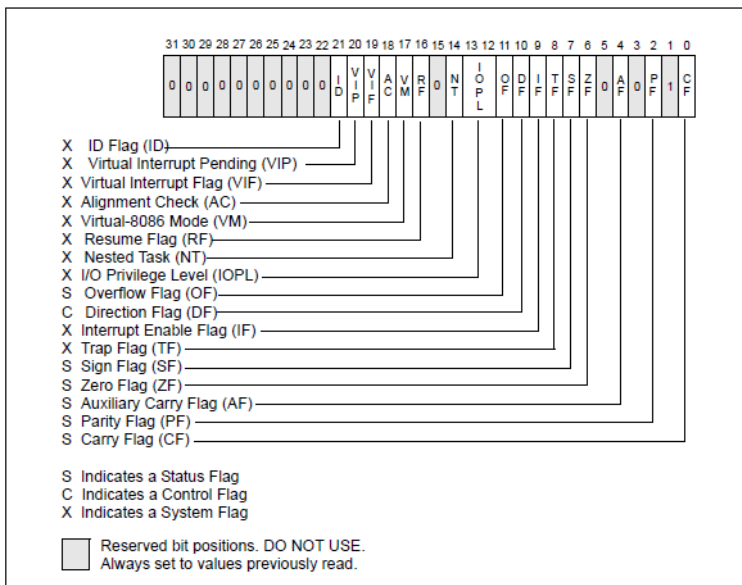


Figure 3-8. EFLAGS Register

EFLAGS中的状态标志

EFLAGS的状态标志代表什么意思呢？它们代表的是算术指令（arithmetic instruction）的结果状态，算术指令就是大家熟悉的加、减、乘、除，ADD, SUB, MUL和DIV，当然还有很多其他指令暗含有这些基本的算术指令，比如cmp指令就暗含有sub操作，因此cmp也会影响状态标志。

typedef struct _EFLAGS

```

{
    unsigned CF: 1; // 进位或错位
    unsigned Reser1: 1; // 对Dr0保存的地址启用 全局断点
    unsigned PF: 1; // 计算结果低位包含偶数个1时 此标志位1
    unsigned Reser2: 1; // 对Dr0保存的地址启用 全局断点
    unsigned AF: 1; // 辅助进位标志 当位3处 有进位或结尾时 该标志为1
    unsigned Reser3: 1; // 保留
    unsigned ZF: 1; // 计算结果为0时 此标志位1
    unsigned SF: 1; // 符号标志 计算结果为负时 该标志位1
    unsigned TF: 1; // 陷阱标志 此标志为1时 CPU每次仅会执行一条指令
    unsigned IF: 1; // 中断标志 为0时禁止响应（屏蔽中断） 为1回复
    unsigned DF: 1; // 方向标志
    unsigned OF: 1; // 溢出标志 计算结果超过表达范围为1 否则为0
    unsigned IOPL: 2; // 用于标明当前任务的I/O特权级

```

```
unsigned NT : 1; // 任务嵌套标志
unsigned Reserve4 : 1; // 对Dr0保存的地址启用 全局断点
unsigned RF : 1; // 调试异常相应标志位 为1禁止相应指令断点异常
unsigned VM : 1; // 为1时启用虚拟8086模式
unsigned AC : 1; // 内存对齐检查标志
unsigned VIF : 1; // 虚拟中断标志
unsigned VIP : 1; // 虚拟中断标志
unsigned ID : 1; // cpuid检查标志
unsigned Reserve5 : 1; // 保留
}EFLAGS, *PEFLAGS;
```

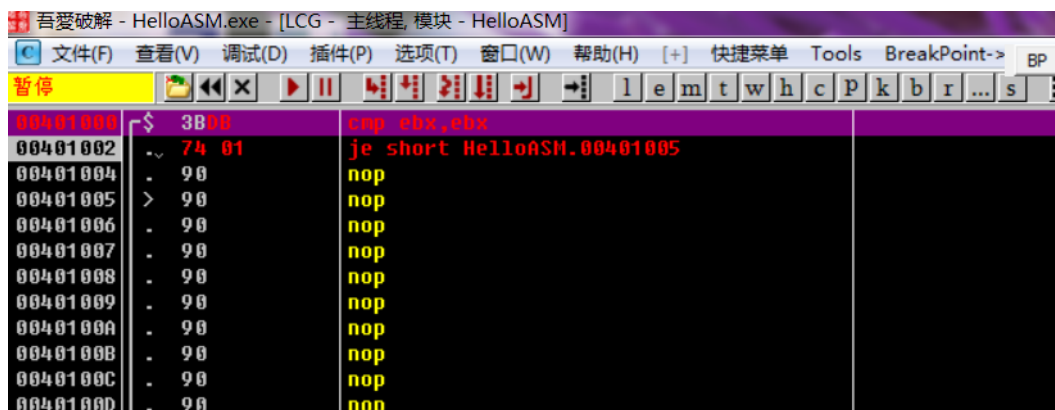
关系运算和条件跳转的对应

表 4-1 条件跳转指令表

指令助记符	检查标记位	说 明
JZ	ZF == 1	等于 0 则跳转
JE	ZF == 1	相等则跳转
JNZ	ZF == 0	不等于 0 则跳转
JNE	ZF == 0	不相等则跳转
JS	SF == 1	符号为负则跳转
JNS	SF == 0	符号为正则跳转
JP/JPE	PF == 1	“1” 的个数为偶数则跳转
JNP/JPO	PF == 0	“1” 的个数为奇数则跳转
JO	OF == 1	溢出则跳转
JNO	OF == 0	无溢出则跳转
JC	CF == 1	进位则跳转
JB	CF == 1	小于则跳转
JNAE	CF == 1	不大于等于则跳转
JNC	CF == 0	无进位则跳转

指令助记符	检查标记位	说 明
JNB	CF == 0	不小于则跳转
JAE	CF == 0	大于则跳转
JBE	CF == 1 或 ZF == 1	小于等于则跳转
JNA	CF == 1 或 ZF == 1	不大于则跳转
JNBE	CF == 0 或 ZF == 0	不小于等于则跳转
JA	CF == 0 或 ZF == 0	大于则跳转
JL	SF != OF	小于则跳转
JNGE	SF != OF	不大于等于则跳转
JNL	SF == OF	不小于则跳转
JGE	SF == OF	不大于等于则跳转
JLE	ZF != OF 或 ZF == 1	小于等于则跳转
JNG	ZF != OF 或 ZF == 1	不大于则跳转
JNLE	SF ==OF 且 ZF == 0	不小于等于则跳转
JG	SF ==OF 且 ZF == 0	大于则跳转

通过试验证明公式:
随便改个demo去加密看看VMP是如何实现的
jxx需要保存了两个信息, 跳转成立 (401005) 与跳转不成立 (401004) 两种



总结:

- 1、jxx我们需要准备两个跳转地址。
- 2、判断Eflag特定位数的值是否为1，然后进行相应的跳转
- 3、我们写的demo中ZF标志位必然是成立的，jz XXX是跳的401005

3、1 实践 (ZF)

- 1、关于jxx处理的函数地址，只关注0x32的jxx系列

```

case 0x32: // JXX系列
case 0x47:
case 0x48:
case 0x49:
case 0x4A: // jcxz
    sub_496044(v5, (int)&savedregs);
    if ( !(a1a->Flag & 8) )
    {
        Ump_CreateVM_PDP_Context(a1a, 0x404);
        v15 = a1a->Magic;
        v16 = Ump_GetNextAddressStart(a1a);
        Ump_SetEsiStruct(a1a, 1, 1, -1, 0xA, v16, v15);
        Ump_CreateVM_PUSH_Context((int)a1a, 0, 9);
    }
    if ( !(a1a->Flag & 4) )
    {
        Ump_CreateVM_PDP_Context(a1a, 0x404);
        Ump_SetEsiStruct(a1a, 1, 1, -1, 14, *(__QWORD *) (a1a->struct_UmFunctionAddr + 16), a1a->Magic);
        Ump_CreateVM_PUSH_Context((int)a1a, 0, 9);
    }
    break;

```

- 2、首先Push两个加密跳转地址，后面根据运算结果来判断使用哪个地址

具体跳转地址A还是地址B取决于前面一句cmp ebx, ebx

```

1 struct_EsiBytecode_1 = __usercall sub_496044@eax(unsigned __int16 a1@eax, int a2)
2 {
3     struct_DisassemblyFunction **v2; // esi@1
4     char v3; // b1@1
5     unsigned __int8 Magic; // a1@1
6     int v5; // edx@13
7     int v6; // eax@13
8     char v7; // al@13
9     char v8; // al@14
10    struct_DisassemblyFunction *v9; // eax@15
11    char v10; // al@15
12    char v11; // al@16
13    int v12; // eax@17
14    char v13; // al@17
15    int v14; // eax@21
16    char v15; // al@21
17    unsigned __int8 v17; // [sp+9h] [bp-3h]@1
18    unsigned __int16 v18; // [sp+Ah] [bp-2h]@1
19    int savedregs; // [sp+Ch] [bp-0h]@13
20
21    v18 = a1;
22    v2 = (struct_DisassemblyFunction **)(a2 - 4);
23    v3 = *(_BYTE *) (a2 - 4) + 0x3E;
24    Magic = *(_BYTE *) (a2 - 4) + 0x9D;
25    v17 = byte_4ED028[Magic];
26    Ump_SetEsiStruct((struct_DisassemblyFunction **)(a2 - 4), 1, 1, -1, 7, 0164, Magic); // VM_PUSH_CONTEXT 跳转地址1
27    Ump_SetEsiStruct((struct_DisassemblyFunction **)(a2 - 4), 1, 1, -1, 7, 0164, *(_BYTE *) (a2 - 4) + 0x9D); // VM_PUSH_CONTEXT 跳转地址2
28    Ump_SetEsiStruct((struct_DisassemblyFunction **)(a2 - 4), 1, 2, -1, 0, 4164, *(_BYTE *) (a2 - 4) + 0x9D); // VM_PUSH_ESP

```

- 3、计算出Eflags右移X位 > Size，我们的例子flags是0x40，Size_1=4,那么v8就是4次了

```

1 }
2 else
3 {
4     while ( (unsigned __int16)v7 > Size_1 ) // 计算出EFlags右移X位 > Size
5     {
6         ++v8;
7         v7 = (unsigned int)(unsigned __int16)v7 >> 1;
8     }
9 }

```

- 4、首先取Eflags标志位，再压入~0x40入栈，最后再进行NOR (~0x40, Eflags) 操作

可以简化成0x40 and ~ Eflags

```

else
{
    while ( (unsigned __int16)EFlags_2 > Size_1 )// 计算出EFlags右移x位 > Size
    {
        ++u8;
        EFlags_2 = (unsigned int)(unsigned __int16)EFlags_2 >> 1;
    }
}
if ( u8 )
    Ump_SetEsiStruct(*u4, 1, 1, -1, 2, abs(u8), 1);// UM_PUSH_IMMW 配合后面的SHR或SHL
if ( u56 )
{
    u9 = *u4;
    LOBYTE(u9) = (*u4)->Magic;
    Ump_SetEsiStruct(*u4, 1, 2, -1, 0, 0x10i64, (int)u9);// UM_PUSH_CONTEXT 取ELFGS标志位
    if ( !u58 )
    {
        u10 = *u4;
        LOBYTE(u10) = (*u4)->Magic;
        Ump_SetEsiStruct(*u4, 1, 2, -1, 0, 0x10i64, (int)u10);// UM_PUSH_CONTEXT 取ELFGS标志位
        u11 = *u4;
        LOBYTE(u11) = (*u4)->Magic;
        Ump_SetEsiStruct(*u4, 0xA, 0, -1, 0, 0i64, (int)u11);// UM_NOR
    }
    u12 = *u4;
    LOBYTE(u12) = (*u4)->Magic;
    HIWORD(u13) = 0xFFFFFFFF;
    LODWORD(u13) = ~EFlags;
    Ump_SetEsiStruct(*u4, 1, 1, 0xFFFFFFFF, 2, u13, (int)u12);// UM_PUSH_IMM
    u14 = *u4;
    LOBYTE(u14) = (*u4)->Magic;
    result = Ump_SetEsiStruct(*u4, 0xA, 0, -1, 0, 0i64, (int)u14);// UM_NOR
}
else
{
    OF_SF = (EFlags & 0x880) == 0x880; // OF与SF置1
    if ( (EFlags & 0x880) == 0x880 )
    {
        LOBYTE(u6) = (*u4)->Magic;
5、根据VM_NOR结果取决使用哪个地址: +0地址1 +4地址2
    }
    if ( u8 )
    {
        if ( u8 <= 0 )
        {
            u54 = *u4;
            LOBYTE(u54) = (*u4)->Magic;
            result = Ump_SetEsiStruct(*u4, 0x3D, 0, -1, 0, 0i64, (int)u54);// UM_SHL_F
        }
        else
        {
            u53 = *u4;
            LOBYTE(u53) = (*u4)->Magic;
            result = Ump_SetEsiStruct(*u4, 0x3E, 0, -1, 0, 0i64, (int)u53);// UM_SHR_F
        }
    }
    return result;
}
}
}
LABEL_22:
Ump_SetEsiStruct(*u2, 4, 0, -1, 0, 0i64, (*u2)->Magic);// UM_ADDF
Ump_SetEsiStruct(*u2, 1, 3, -1, 0, 3i64, (*u2)->Magic);// UM_MOV_XSA_TO_B
Ump_SetEsiStruct(*u2, 2, 2, -1, 0, 0x11i64, (*u2)->Magic);// UM_POP_CONTEXT
Ump_SetEsiStruct(*u2, 2, 2, -1, 0, 0x14i64, (*u2)->Magic);// UM_POP_CONTEXT
Ump_SetEsiStruct(*u2, 2, 2, -1, 0, 0x14i64, (*u2)->Magic);// UM_POP_CONTEXT
Ump_SetEsiStruct(*u2, 1, 2, -1, 0, 0x11i64, (*u2)->Magic);// UM_PUSH_CONTEXT
return Ump_CreateVM_PUSH_Context((int)*u2, 0x400, 0xC);
}
}

```

6、总结下前面流程

整理下流程（数据参考demo）可以分为三步骤：

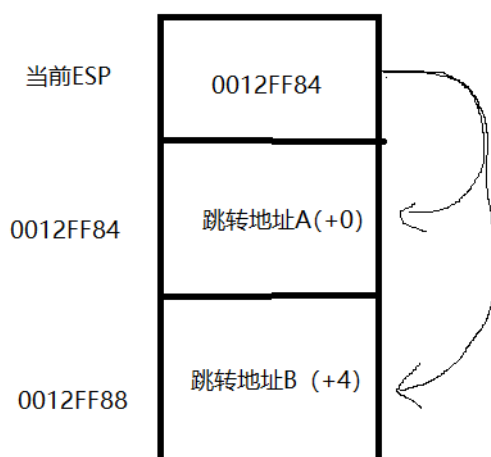
- 1、保存跳转地址A和B
- 2、计算eflags zf的数值，返回对应的结果

公式：

ZF = 0x40 and ~ELFGS

- 3、根据结果ADD进行跳转地址A或B

ADD+0还是ADD+4判断跳转A还是B



伪代码如下:

序号	伪指令	说明
0	VM_PUSH_IMM	加密跳转地址A入栈
1	VM_PUSH_IMM	加密跳转地址B入栈
2	VM_PUSH_ESP	保存当前Esp
3	VM_PUSH_IMMW	配合后面的SHR或SHL, 注意这里是保存2字节
4	JVM_PUSH_CONTEXT	取堆栈中ELFGS标志位, 0x246
5	VM_PUSH_IMM	保存~0x40,后面用来VM_NOR计算
6	VM_NOR	0x40 and ~0x246 = 0
7	VM_SHR	!=是4, ==就是0
8	VM_ADDF	根据VM_SHR返回值决定指向跳转地址A(+0)或则B(+4)
XX	XX	XX

7、OD实战数据

VM_PUSH_IMM 加密地址A入栈(401004)

0012FF88 4C69E901 加密地址A

VM_PUSH_IMM 加密地址B入栈(401005)

0012FF84 4C69F601 加密地址B

0012FF88 4C69E901 加密地址A

VM_PUSH_ESP

0012FF80 0012FF84 保存ESP, 后面+0还是+4来指向加密地址A、B

0012FF84 4C69F601

0012FF88 4C69E901

VM_PUSH_IMMW 配合后面的SHR或SHL

0012FF7E FF840004 注意这个4, 后面与VM_NOR的结果做运算

VM_PUSH_CONTEXT 取ELFGS标志位

0012FF7A 00000246 这里取出了Elfgs的值, 前面cmp ebx,ebx执行完毕的值

PUSH_IMM

0012FF76 FFFFFFFBF 保存~0x40的值, 后面要跟Elfgs运算

VM_NOR

0012FF7A 00000000 0x40 and ~0x246 = 0, 相同0, 不同0x40

VM_SHR 注意这里是区分跳转A和B的关键

0012FF7C 00000000 相同=0, 不同=4

0012FF80 0012FF84

0012FF84 4C69F601

0012FF88 4C69E901

ADD_F 判断+0还是+4来指向加密跳转地址A还是B
0012FF80 0012FF84 根据SHR的值 0(4) + 0x12FF84
0012FF84 4C69F601
0012FF88 4C69E901

VM_MOV_XSA_TO_B 取地址的值
0012FF80 4C69F601
0012FF84 4C69F601
0012FF88 4C69E901

后面的操作就是解密跳转地址，然后设置Esi的值跳到401005执行。懒的写下去了，核心都写完了。