VMSweeper 分析

一. 基本使用

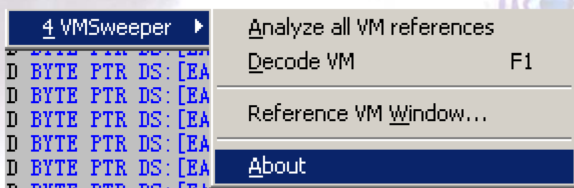
最近,有一位牛人发布了一个比较有意思的插件,我们一起看一下其中一些好玩的东西。既然有大牛放血，那我就顺便给没有追踪过 **VM** 的同学扫扫盲。（本文对应插件更新至 **VMSweeper1.4 beta 8**）

由于我并没有源码，很多东西都是靠猜了，如果有什么地方说的不对，请各位多多指教。

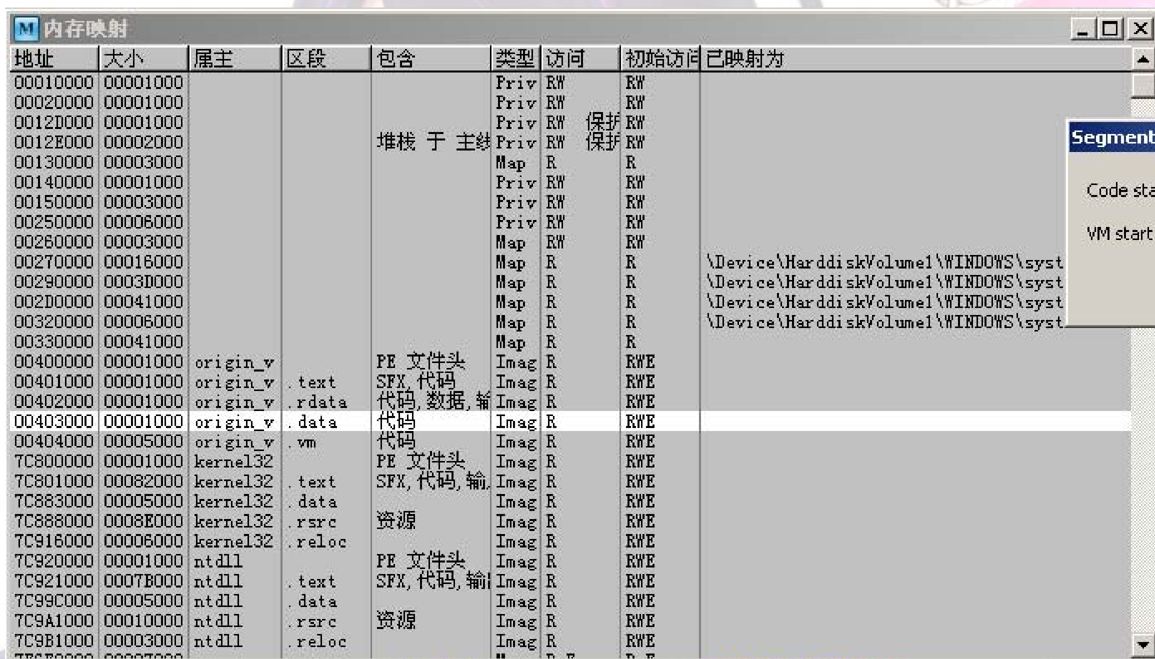
这个插件的下载地址是 **http://forum.exetools.com/showthread.php?t=13084**

这个插件有些东西还没有完善,因此经常会出错,我运行了几个程序,都没有一个能正常运行完的,不过这并不阻碍我们了解程序的大体思想。在看这篇无聊的文章之前首先你要明白 **VMProtect** 的基本架构，因为追踪和还原 **VM** 有一部分是和 **VM** 的架构相关的。

安装后打开一个 **VMP** 加密的程序，然后点击 **VMSweeper**



首先要选 **Analyze all VM references**



接着填写相关的参数,这里的 **Code start** 和 **Code end** 指定代码段的起始和结束

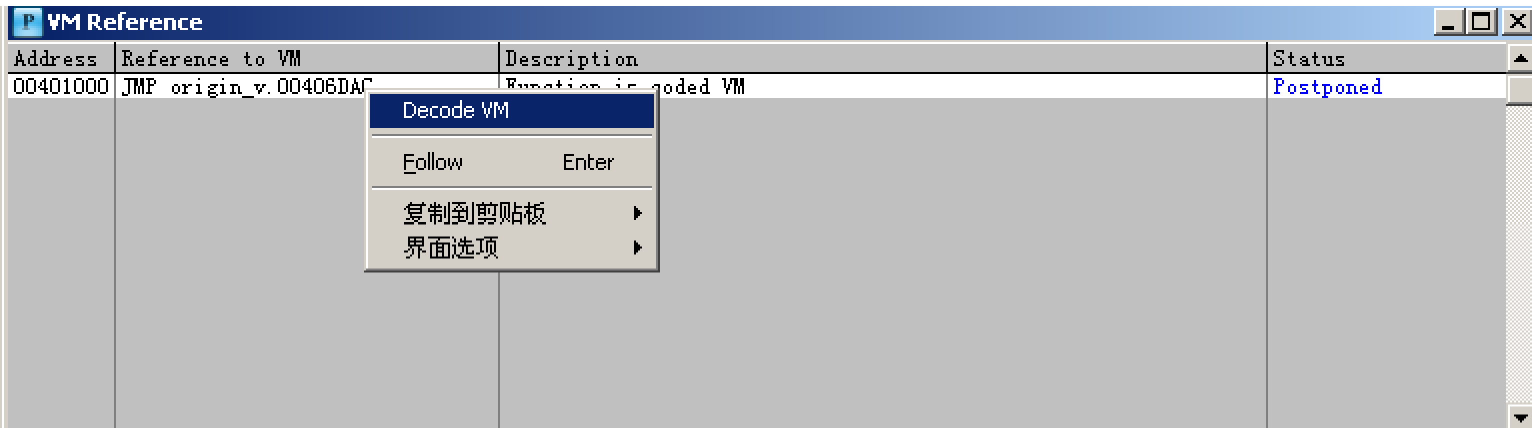
VM start 和 **VM end** 指定 **vm** 段的起始和结束,还有一个要注意的是要把区段名改为 **VM**。

为什么要填这些参数呢，我在《**VMProtect** 逆向分析》里讲过了，因为入口的指令特征被抹除掉了，目前比较好的识别入口算法就是查找 **Code** 段的所有 **JMP** 指令，然后检查是否 **JMP** 到 **VM** 段内，再做一些特征的校验(陷阱校验)。

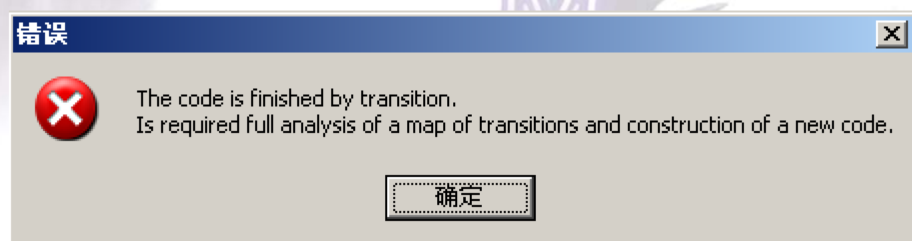
还有一个就是判断 **VM** 内部数据和退出 **VM** 的识别，这些都需要 **VM** 段的界限。

填写好了之后，会列出 **VM** 的入口

新版的 **VMSweeper** 已经自动获取段界限，不用手动输入了。



接着就可以 **Decode VM** 了，这时看到在 **VM** 的入口下了一个断点，当断点命中的时候按 **F1** 进行分析。
进行分析后弹出一个框后要求重新打开程序，这里正常的话应该是重启，然后进行 **patch**。



三. 初始化状态

重新打开程序，我们来看一下刚刚 **VMSweeper** 做了些什么。
打开 **OD** 的目录，你会发现多了一个以程序名称命名的目录，打开它，会看到一些文件。

404BDD.trc
405D1E.log
405D1E.trc
401000.log
401000.trc
Hnd_00401000.map
Mem_00401000.map
PiCode0_00401000.map
PiCode0_00401000.pmb
PiCode1_00401000.map
PiCode1_00401000.pmb
Refs_00401000.map
Regs_00405D1E.map
Regs_00405F91.map
Reloc0_00401000.map
Reloc1_00401000.map
Rvm_00401000.map
Stack_00405D1E.map
Stack_00405F91.map
Trans0_00401000.map
Trans1_00401000.map
Zones_00401000.map

这些文件都是干什么的呢，我们一个一个来看。

Regs_xxxxxxx

Stack_xxxxxxx

很明显,文件名已经提示是寄存器和堆栈了,那么到底是什么寄存器和堆栈呢,我没有插件的源代码,就只能猜了,如果没有猜错的话这个应该是保存插件内部虚拟机的状态。

Regs_00405D1E.map:

eax: eax << (ecx << ecx)

ecx: (ecx << ecx) - 1

edx: edx

ebx: 0x00407C27

esp: 0x0012FEE4

ebp: 0x0012FFA4

esi: [0x0012FFA4] + 0x00407C27

edi: 0x0012FEE4

efl: efl

首先看地址 **00405D1E**

```
Immunity Debugger - test2.vmp.exe
文件(F) 查看(V) 调试(D) 脚本库(I) 选项(T) 窗口(W) 帮助(H)
CPU - 主线程 - 模块 - test2_vm

00405D1E  > 66:0FBBF1        BTC CX,SI
00405D22  . 0F94C0           SETE AL
00405D25  . 8A46 FF          MOV AL,BYTE PTR DS:[ESI-1]
00405D28  . D2E5             SHL CH,CL
00405D2A  . 00D8             ADD AL,BL
00405D2C  . 66:0FACD9 02     SHRD CX,BX,2
00405D31  . C0C1 07          ROL CL,7
00405D34  . D2E1             SHL CL,CL
00405D36  . 04 91            ADD AL,91
00405D38  .^E9 0DECFFFF      JMP test2_vm.0040494A
00405D3D  $ 895424 48        MOV DWORD PTR SS:[ESP+48],EDX
00405D41  . E9 210B0000      JMP test2_vm.00406867
00405D46  > F9               STC
00405D47  . 66:D3E0          SHL AX,CL
00405D4A  . 9C               PUSHFD
00405D4B  . 66:8945 04       MOV WORD PTR SS:[EBP+4],AX
00405D4F  .^E9 52FEFFFF      JMP test2_vm.00405BA6
00405D54  $ 89F6             MOV ESI,ESI
00405D56  . 9C               PUSHFD
00405D57  . 8F4424 30        POP DWORD PTR SS:[ESP+30]
00405D5B  . F8               CLC
00405D5C  . 66:D3D6          RCL SI,CL
00405D5F  . C1C6 0F          ROL ESI,0F
00405D62  . F9               STC
00405D63  . FF35 6B614000    PUSH DWORD PTR DS:[40616B]
00405D69  . 8F4424 2C        POP DWORD PTR SS:[ESP+2C]
00405D6D  . 66:89EE          MOV SI,BP
00405D70  . C74424 28 0000(  MOV DWORD PTR SS:[ESP+28],0
00405D78  . 66:D3DE          RCR SI,CL
00405D7B  . 8B7424 58        MOV ESI,DWORD PTR SS:[ESP+58]
00405D7F  .^E9 4AF9FFFF      JMP test2_vm.004056CE
00405D84  $ 9C               PUSHFD
00405D85  . 8F4424 34        POP DWORD PTR SS:[ESP+34]
00405D89  . 66:0FB6F1        MOVZX SI,CL
00405D8D  . 5E               POP ESI
00405D8E  . E8 76090000      CALL test2_vm.00406709
SI=7C27
CX=FEFF
跳转来自 00404192, 0040430B, 00404437, 00404510, 00404551, 004047A4, 00404

寄存器 (FPU)
EAX 00000000
ECX 0013FEFF
EDX 7C92E514 ntdll.KiFastSystemCallRet
EBX 00407C27 test2_vm.00407C27
ESP 0013FED0 ASCII "Mm  "
EBP 0013FF90
ESI 00407C27 test2_vm.00407C27
EDI 0013FED0 ASCII "Mm  "
EIP 00405D1E test2_vm.00405D1E
C 0  ES 0023 32 位 0(FFFFFFFF)
P 1  CS 001B 32 位 0(FFFFFFFF)
A 0  SS 0023 32 位 0(FFFFFFFF)
Z 0  DS 0023 32 位 0(FFFFFFFF)
S 0  FS 003B 32 位 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00000206 (NO,NB,NE,A,NS,PE,GE,G)
ST0 empty -UNORM CFC4 01050104 0049005C
ST1 empty +UNORM 0075 00620065 00440079
ST2 empty +UNORM 0069 006E0069 002E0072
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1

地址      十六进制                                         ASCII
00403000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

0013FED0  20206D4D  Mm
0013FED4  00000400
0013FED8  AE9FAC18
0013FEDC  8B893000
0013FEE0  8054CBE1
0013FEE4  8055A388
0013FEE8  8055A4C8
0013FEEC  0000BB40
0013FEF0  AE9FAC08
0013FEF4  8053C675
0013FEF8  00000000  ....
0013FEFC  AE9FAC1C
0013FF00  8B859280
0013FF04  8054C6ED
0013FF08  00000000  ....
暂停
```

这里记录的主要有用的是 **ESI(VMP** 目前版本的 **EIP)**的初始化值。

这里记录 **EAX** 和 **ECX** 不知道是不是作者搞错了其它的 **VM**，**VMP** 在入口 **EAX** 和 **ECX** 并没有什么用，这里还有一个比较有用的是 **EBX** 的校验算法，不过这里并没有记录。这里应该追踪的是 **ADD ESI,DWORD PTR SS:[EBP]**指令。

接下来看

Stack_00405D1E.map:

………

0012FF94: empty
0012FF98: empty
0012FF9C: empty
0012FFA0: empty
0012FFA4: 0
0012FFA8: [0x0040616B]
0012FFAC: efl
0012FFB0: ebp
0012FFB4: ecx
0012FFB8: ebx
0012FFBC: esi
0012FFC0: edx
0012FFC4: edi
0012FFC8: ebx
0012FFCC: eax
0012FFD0: 0x991AD4CC
0012FFD4: 0x0101A0F0
0012FFD8: var0
0012FFDC: var1
0012FFE0: var2
0012FFE4: var3
0012FFE8: var4
0012FFEC: var5
0012FFF0: var6
0012FFF4: var7
0012FFF8: var8
0012FFFC: var9

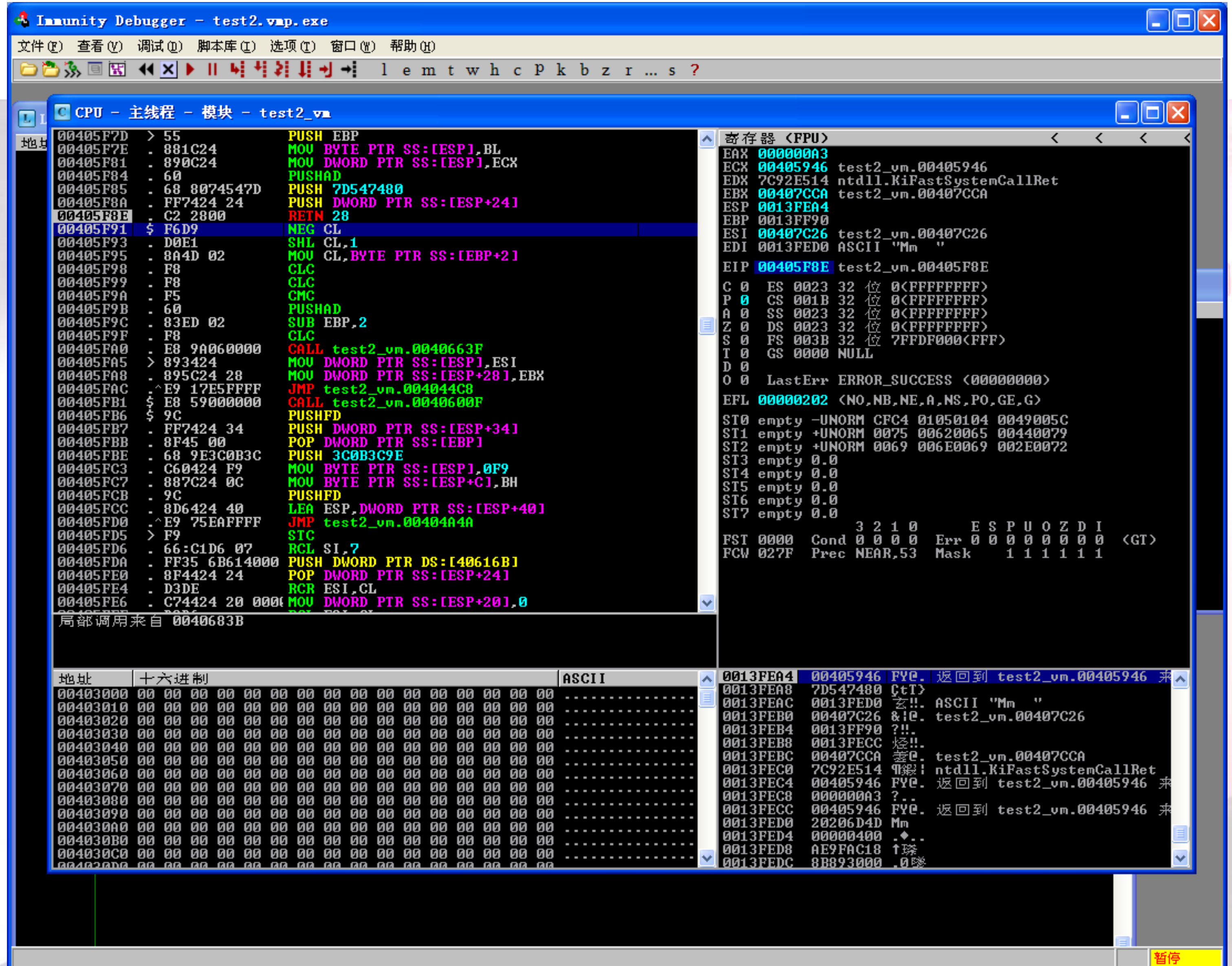
很明显，这里是虚拟机对堆栈的标签。这里主要追踪的是入口数据。

上面还有一个问题，就是堆栈的地址不对，导致 **ESP**，**EBP**，**EDI** 的数值错误，在这个测试程序里没有什么影响，因为 **VM** 的代码没有访问参数，真正的堆栈是这样的。

```
0013FFA0    0013FFB0   ?‼ .
0013FFA4    7FFDD000   .旋
0013FFA8    FFFFFFFF
0013FFAC    7C92E514   ¶鍜|    ntdll.KiFastSystemCallRet
0013FFB0    7C930228   (┐揩    ntdll.7C930228
0013FFB4    7FFDD000   .旋
0013FFB8    00000000   ....
```

0013FFBC　991AD4CC　淘→
这里估计是 VMSweeper 内部的 VM 指令模拟错误。这个问题好久都没有修正-_-

接下来看 Regs_00405F91.map 和 Regs_00405D1E.map
地址 00405F91



指令执行的开始(00405F91)，这里是追踪 VMP 必须记录的地方。很明显这里也是追踪 RETN XX 这条指令
这里记录些什么呢。

Regs_00405F91.map:
eax: 0xA3
软星科技(北京)有限公司 版权所有
ecx: [404E69] ^ 0x3FE1E344
edx: edx
ebx: 0x00407CCA
esp: 0x0012FEE4
ebp: 0x0012FFA4
esi: 0x00407C26
edi: 0x0012FEE4
efl: efl

这里记录的是解密 handle 地址算法，这里的 EAX 就是 handle。这个算法是必须获得的，因为要用这个算法来获得指令表保存的真正地址。

在 VMP 里，没有使用的指令是不会出现在指令表的，VMP 会用使用指令的副本进行填充，在现阶段，我们无法获知 VMP 将会调用哪些指令，VMSweeper 对所有的 handle 都进行了分析，这个架构对目前的版本的 VMP 是没有问题的，但是对某些用陷阱地址填充的 VM 或者未来版本 VMP 也使用陷阱地址的话，就考验 VMSweeper 的容错和启发式了，个人觉得运行时分析对付这些小花招会好一点。

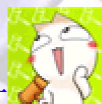
其实用 VM 来追踪 VM 并不是什么新鲜的东西，有时甚至会用一个以上的 VM 来进行追踪，保存 VM 某个时间的现场和记录运行轨迹也是必须的，这样可以方便调试。

接下来 405D1E.trc 和 404BDD.trc，这两个文件保存了 VMSweeper 内部虚拟机的执行轨迹，这里也没有什么讲的

Compile ASSIGN (0, 2)
Compile ASSIGN (3, 4)
Compile ASSIGN (5, 8)
Compile ASSIGN (9, 10)
Compile ASSIGN (11, 13)
Compile ASSIGN (14, 15)
Compile ASSIGN (16, 19)
Compile ASSIGN (20, 21)
Compile ASSIGN (22, 23)

ASSIGN 带两个参数，第一个是汇编代码的起始位置（总的大小），第二个是执行轨迹前面的索引。

接着看一个好东西，405D1E.log，405D1E 是指令表的地址，这里记录的是 VMSweeper 指令收缩和模板生成流程。
第一个模板对应 405D1E.trc 的指令轨迹，生成的是函数地址解码模板，其它的是函数模板对应 404BDD.trc。

这里的信息太多了（不知道 VMSweeper 后面的版本还会不会明文），足够逆出整个 VMSweeper 的扫描架构了。
这里说几句废话，给没有写过解码器的童鞋扫扫盲，据本菜鸟所知，目前常用的扫描方法有三种。

一种是最常见的特征码扫描，这里的特征码主要指的是机器码，这种扫描速度快，精确度高，但是对 VM 的话基本没有用了，原因是 VM 每次生成的机器码都不一样。
建议在写解码器时要尽可能远离机器码，看看 VMP 的入口就知道了，任何的机器码都有可能在下一个版本被抹掉。

第二种是局部特征码，也就是使用模板，个人认为 VMSweeper 和 VMP 都使用这种方法，一部分的代码在编译时动态生成通过逆出 VMP 的模板来生成相同的代码进行扫描。

还有一种就是追踪数据流，这种方法最理想的状态是不依赖任何的特征码。

我们来一个 VMSweeper 指令模板
VMP_CRC:
这个指令的识别在 VMP 里应该是最难的了，底层的解码器做得怎么样就看这条指令了。
轨迹如下

bbs.pediy.com
bbs.unpack.cn
作者：aalss2

```
Instr: 0 parsing - 0x0040504D: movzx cx, bl (aux:0x1808 insn:0x00)
Instr: 1 parsing - 0x00405051: neg dx (aux:0x1808 insn:0x00)
Instr: 2 parsing - 0x00405054: mov edx, dword ptr ss:[ebp] (aux:0x1C08 insn:0x00)
Instr: 3 parsing - 0x00405057: lea ecx, dword ptr ds:0x3E5FB70A[ecx*4] (aux:0x1828 insn:0x00)
Instr: 4 parsing - 0x0040505E: add ebp, 4 (aux:0x1808 insn:0x00)
Instr: 5 parsing - 0x00405061: shrd ecx, ebx, cl (aux:0x1808 insn:0x00)
Instr: 6 parsing - 0x00405064: inc cl (aux:0x1808 insn:0x00)
Instr: 7 parsing - 0x00405066: sub eax, eax (aux:0x1808 insn:0x00)
Instr: 8 parsing - 0x00405068: shr cl, cl (aux:0x1808 insn:0x00)
Instr: 9 parsing - 0x004042C1: sub cl, 0D0h (aux:0x1808 insn:0x00)
Instr: 10 parsing - 0x004042C4: mov ecx, eax (aux:0x1808 insn:0x00)
Instr: 11 parsing - 0x004042C6: push 004042CBh (aux:0x1808 insn:0x00)
Instr: 12 parsing - 0x00405575: mov byte ptr ss:[esp], ch (aux:0x1C08 insn:0x00)
Instr: 13 parsing - 0x00405578: shl eax, 7 (aux:0x1808 insn:0x00)
Instr: 14 parsing - 0x0040557B: lea esp, dword ptr ss:[esp + 4] (aux:0x1C28 insn:0x00)
Instr: 15 parsing - 0x00405585: shr ecx, 19h (aux:0x1808 insn:0x00)
Instr: 16 parsing - 0x00405588: cmc (aux:0x1808 insn:0x00)
Instr: 17 parsing - 0x00405589: cmc (aux:0x1808 insn:0x00)
Instr: 18 parsing - 0x0040558A: pushad (aux:0x1808 insn:0x00)
Instr: 19 parsing - 0x0040558B: push 00405590h (aux:0x1808 insn:0x00)
Instr: 20 parsing - 0x00405E3F: or eax, ecx (aux:0x1808 insn:0x00)
Instr: 21 parsing - 0x00405E41: push 2E73AA38h (aux:0x1808 insn:0x00)
Instr: 22 parsing - 0x00406661: xor al, byte ptr ds:[edx] (aux:0x1808 insn:0x00)
Instr: 23 parsing - 0x00406663: mov byte ptr ss:[esp], al (aux:0x1C08 insn:0x00)
Instr: 24 parsing - 0x00406666: inc edx (aux:0x1808 insn:0x00)
Instr: 25 parsing - 0x00406667: push 0040666Ch (aux:0x1808 insn:0x00)
Instr: 26 parsing - 0x00406289: push 0040628Eh (aux:0x1808 insn:0x00)
Instr: 27 parsing - 0x0040614D: dec dword ptr ss:[ebp] (aux:0x1C08 insn:0x00)
Instr: 28 parsing - 0x00406150: push esp (aux:0x1808 insn:0x00)
Instr: 29 parsing - 0x00406151: mov byte ptr ss:[esp + 4], bh (aux:0x1C28 insn:0x00)
Instr: 30 parsing - 0x00406155: push dword ptr ss:[esp + 4] (aux:0x1C28 insn:0x00)
Instr: 31 parsing - 0x00406159: lea esp, dword ptr ss:[esp + 38h] (aux:0x1C28 insn:0x00)
Instr: 32 parsing - 0x00406163: pushfd (aux:0x1808 insn:0x00)
Instr: 33 parsing - 0x00406164: pushad (aux:0x1808 insn:0x00)
Instr: 34 parsing - 0x00406165: pushfd (aux:0x1808 insn:0x00)
Instr: 35 parsing - 0x00406166: push 0040616Bh (aux:0x1808 insn:0x00)
Instr: 36 parsing - 0x0040453F: mov dword ptr ss:[ebp], eax (aux:0x1C08 insn:0x00)
Instr: 37 parsing - 0x00404542: pushfd (aux:0x1808 insn:0x00)
Instr: 38 parsing - 0x00404543: mov byte ptr ss:[esp + 8], 1Eh (aux:0x1C28 insn:0x00)
Instr: 39 parsing - 0x00404548: mov word ptr ss:[esp + 10h], sp (aux:0x1428 insn:0x00)
Instr: 40 parsing - 0x0040454D: lea esp, dword ptr ss:[esp + 30h] (aux:0x1C28 insn:0x00)
```

收缩后形成一个匹配模板

```
0040504D: edx = [ebp]
00405057: ebp += 4
00405061: cmd -= cmd
00405068: ecx = cmd
004042C6: cmd <<= 7
0040557B: ecx >>= 0x00000019
00405588: cmd |= ecx
00405E41: cmd ^= [edx]
00406667: [ebp] -= 1
00406150: [ebp] = cmd
```

*** Primitive Template ***
{ NN_mov, dt_dword, "edx", "[ebp]", R_SS };
{ NN_add, dt_dword, "ebp", "4", R_NONE };
{ NN_sub, dt_dword, "cmd", "cmd", R_NONE };
{ NN_mov, dt_dword, "ecx", "cmd", R_NONE };
{ NN_shl, dt_dword, "cmd", "7", R_NONE };
{ NN_shr, dt_dword, "ecx", "0x00000019", R_NONE };
{ NN_or, dt_dword, "cmd", "ecx", R_NONE };
{ NN_xor, dt_byte, "cmd", "[edx]", R_DS };
{ NN_sub, dt_dword, "[ebp]", "1", R_SS };
{ NN_mov, dt_dword, "[ebp]", "cmd", R_SS };
这个应该就是内部定义的匹配模板了，参数也很简单。

VM primitive 0B – Crc

VMSweeper 将一些特殊值进行了定义，个人理解为可能考虑这些值在以后的版本会改变。

cmd ： 动态解码的数值
reg_cmd ： VMP 内部寄存器数值
efl ： eflags

还有一些有趣的寄存器大小转换，这里就不说了。

三. 伪代码追踪

在识别所有的 handle 后，VMSweeper 开始追踪伪代码，其轨迹记录在(入口 eip).trc 里，这里的测试程序是 401000.trc
这里的花指令识别和指令收缩和底层的并没有什么不同
这个文件上半部分记录的是入口的轨迹，从
***** Start Virtual Machine *****
开始记录 VMP 伪代码的轨迹 log 窗口显示的数据

```
00407C20  A9  pop  dword ptr [reg_34]  ;00000206  ;efl
00407C1F  A9  pop  dword ptr [reg_34]  ;203D9563  ;__$ret
00407C1E  B1  pop  dword ptr [reg_4]   ;00000246  ;iEFL
00407C1D  A5  pop  dword ptr [reg_2C]  ;0012FFF0  ;iEBP
00407C1C  BB  pop  dword ptr [reg_10]  ;0012FFB0  ;iECX
00407C1B  A7  pop  dword ptr [reg_28]  ;7FFD9000  ;iEBX
00407C1A  BF  pop  dword ptr [reg_18]  ;FFFFFFFF  ;iESI
00407C19  B7  pop  dword ptr [reg_8]   ;7C92EB94  ;iEDX
00407C18  AF  pop  dword ptr [reg_38]  ;7C930738  ;iEDI
00407C17  A1  pop  dword ptr [reg_24]  ;7FFD9000  ;iEBX
00407C16  AD  pop  dword ptr [reg_3C]  ;00000000  ;iEAX
00407C15  BD  pop  dword ptr [reg_1C]  ;991AD4CC  ;0x991AD4CC
00407C14  B9  pop  dword ptr [reg_14]  ;0101A0F0  ;0x0101A0F0
……
```
再往下看

***** Stop Virtual Machine *****

Instr: 0 parsing - 0x00140040: push 0101A0F0h (aux:0x1808 insn:0x00)
Instr: 1 parsing - 0x00140045: push 991AD4CCh (aux:0x1808 insn:0x00)
Instr: 2 parsing - 0x0014004A: push eax (aux:0x1808 insn:0x00)
Instr: 3 parsing - 0x0014004B: push ebx (aux:0x1808 insn:0x00)
Instr: 4 parsing - 0x0014004C: push edi (aux:0x1808 insn:0x00)
Instr: 5 parsing - 0x0014004D: push edx (aux:0x1808 insn:0x00)
Instr: 6 parsing - 0x0014004E: push esi (aux:0x1808 insn:0x00)
……

这里记录的是伪代码的轨迹，再下来是花指令识别的轨迹。
然后就不正常了

Can't devirtualize line - svm_34 = and ~svm_22, 0x00000010
Can't devirtualize line - rvm_14 = shr svm_34, 0x04
Can't devirtualize line - svm_45 = and rvm_3C, 0x0F
Not equal variable (0x0F - 00000001) in line - svm_45 = and rvm_3C, 0x0F
……

这里先不管它了……

我们接下来看 401000.log。
VMSweeper 在运行时申请了一块内存，我这里的是

内存映射，条目 5
地址=00140000
大小=00040000 (262144.)
属主= 00140000 (自身)
区段=
类型=Priv 00021040
访问=RWE
初始访问=RWE

内存开始的 0x40 作为执行引擎的寄存器，然后将 VMP 伪代码用汇编指令模拟

00140040: [0x00140020] = 0
00140064: svm_1 = add [0x0040616B], 0x203D9563
0014006D: [0x00140034] = iEFL
00140074: [0x00140034] = svm_1
0014007A: [0x00140004] = iEFL
00140080: [0x0014002C] = iEBP
00140086: [0x00140010] = iECX
0014008C: [0x00140028] = iEBX
00140092: [0x00140018] = iESI
00140098: [0x00140008] = iEDX
0014009E: [0x00140038] = iEDI
001400A4: [0x00140024] = iEBX
001400AA: [0x0014003C] = iEAX
001400B0: [0x0014001C] = 0x991AD4CC
001400B6: [0x00140014] = 0x0101A0F0
001400BC: svm_2 = add 0xB1B55788, 0x4E8AF8C5

001400CF: [0x00140014] = iEFL
001400D6: svm_3 = add svm_2, [0x00140020]
001400E0: [0x0014001C] = iEFL
001400E7: svm_4 = add 0x696F2C6D, [svm_3]
001400EE: [0x00140030] = iEFL
001400F5: svm_5 = add 0x0000000A, __$esp
……

Patch 的指令是

```
00140040    68 F0A00101     PUSH 101A0F0                  //模拟入口
00140045    68 CCD41A99     PUSH 991AD4CC
0014004A    50              PUSH EAX
0014004B    53              PUSH EBX
0014004C    57              PUSH EDI
0014004D    52              PUSH EDX
0014004E    56              PUSH ESI
0014004F    53              PUSH EBX
00140050    51              PUSH ECX
00140051    55              PUSH EBP
00140052    9C              PUSHFD
00140053    FF35 6B614000   PUSH DWORD PTR DS:[40616B]
00140059    68 00000000     PUSH 0
0014005E    8F05 20001400   POP DWORD PTR DS:[140020]
00140064    68 63953D20     PUSH 203D9563
00140069    58              POP EAX
0014006A    010424          ADD DWORD PTR SS:[ESP],EAX    //用堆栈来模拟运算
0014006D    9C              PUSHFD
0014006E    8F05 34001400   POP DWORD PTR DS:[140034]
00140074    8F05 34001400   POP DWORD PTR DS:[140034]
0014007A    8F05 04001400   POP DWORD PTR DS:[140004]
00140080    8F05 2C001400   POP DWORD PTR DS:[14002C]
00140086    8F05 10001400   POP DWORD PTR DS:[140010]
0014008C    8F05 28001400   POP DWORD PTR DS:[140028]
00140092    8F05 18001400   POP DWORD PTR DS:[140018]
00140098    8F05 08001400   POP DWORD PTR DS:[140008]
0014009E    8F05 38001400   POP DWORD PTR DS:[140038]
001400A4    8F05 24001400   POP DWORD PTR DS:[140024]
001400AA    8F05 3C001400   POP DWORD PTR DS:[14003C]
001400B0    8F05 1C001400   POP DWORD PTR DS:[14001C]
001400B6    8F05 14001400   POP DWORD PTR DS:[140014]
001400BC    68 6D2C6F69     PUSH 696F2C6D
001400C1    68 8857B5B1     PUSH B1B55788
001400C6    68 C5F88A4E     PUSH 4E8AF8C5
001400CB    58              POP EAX
001400CC    010424          ADD DWORD PTR SS:[ESP],EAX
```

```
001400CF    9C               PUSHFD
001400D0    8F05 14001400    POP DWORD PTR DS:[140014]
001400D6    FF35 20001400    PUSH DWORD PTR DS:[140020]
001400DC    58               POP EAX
001400DD    010424           ADD DWORD PTR SS:[ESP],EAX
001400E0    9C               PUSHFD
001400E1    8F05 1C001400    POP DWORD PTR DS:[14001C]
001400E7    58               POP EAX
001400E8    FF30             PUSH DWORD PTR DS:[EAX]
001400EA    58               POP EAX
001400EB    010424           ADD DWORD PTR SS:[ESP],EAX
001400EE    9C               PUSHFD
001400EF    8F05 30001400    POP DWORD PTR DS:[140030]
001400F5    68 4DF9ADA9      PUSH A9ADF94D
……
```

四. 最后的一点废话

这里猜想作者是想用和底层同样的扫描架构来识别伪代码，这个效果到底怎么样现在还不太清楚，我看了一下，**Patch** 的代码并不完全准确，特别是在跳转方面，**VMSweeper** 并没有 **patch** 完整的伪代码，期待 **VMSweeper** 的改进。最后感谢一下 **VMSweeper** 的作者，放出这个插件让我们可以学习到 **VMP** 的知识。

相关工具

Immunity Debugger 1.5 汉化修正 080417
VMSweeper1.4 beta 8
VMProtect.Ultimate.V2.0.4.4140.Incl.License.Offer.By.1ST
HA_OllyDBG_1.10_second_cao_cong_fix22

软星科技(北京)有限公司 版权所有

一只小菜鸟
2011-03-01