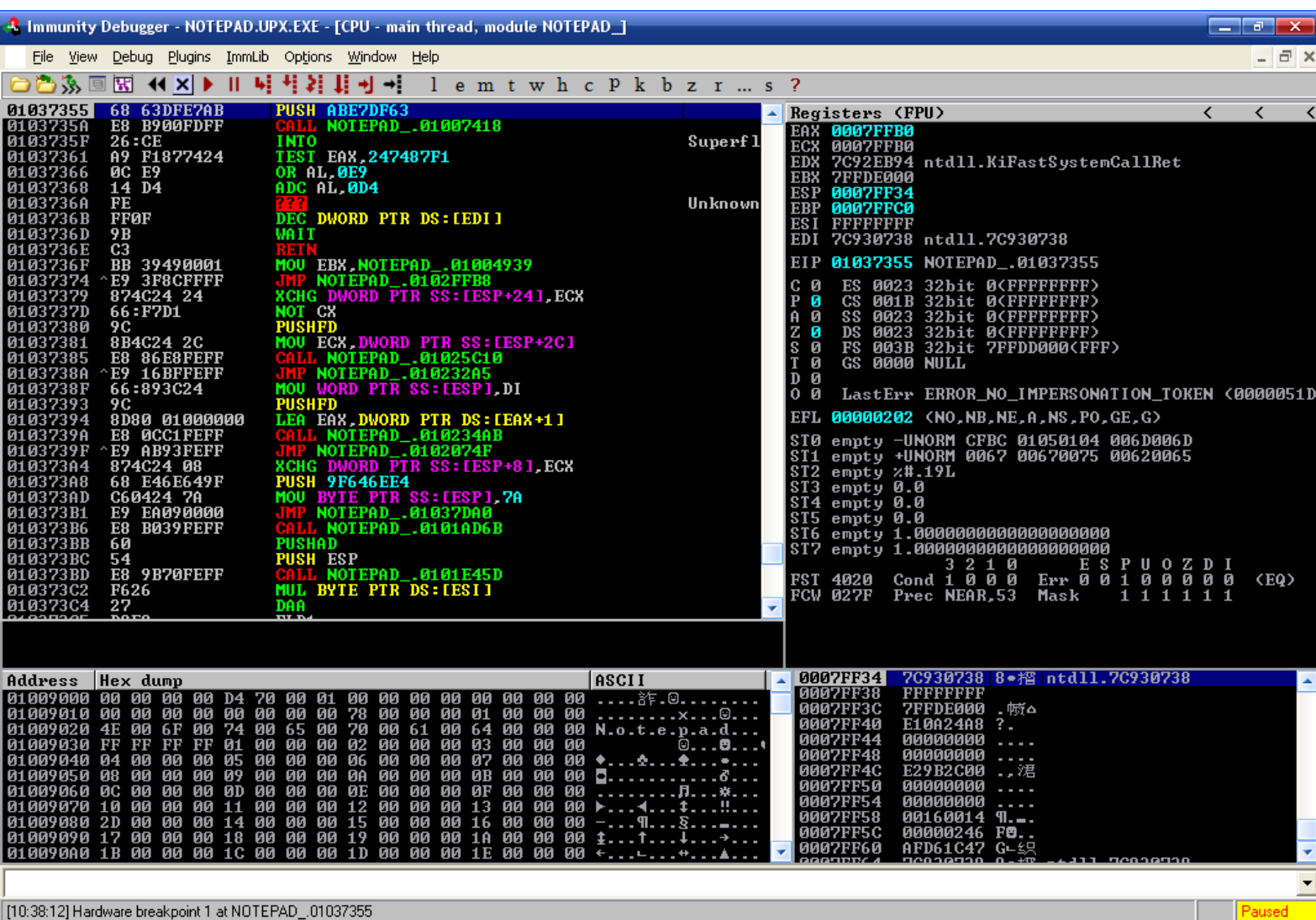


## VMPProtect 逆向分析

## 一. VMP 的入口

```
01037355 68 63DFE7AB    PUSH ABE7DF63
0103735A E8 B900FDFF    CALL NOTEPAD_.01007418
```



VM 开始, 这里是对应源程序的入口, 一开始 vmp 会压入一个 key, 这个 key 是用来计算函数 VM 开始执行的地址.在 2.0 版本后, 这里做了改变,估计主要是因为 Nooby 公开了猜函数的方法。

目前 2.04 的入口如下

```
00401001 > $-E9 DA5A0000    JMP test2_vm.00406AE0    #入口变成了一条 JMP 指令
00406AE0  > 68 38321EFA    PUSH FA1E3238
00406AE5  . FF3424        PUSH DWORD PTR SS:[ESP]
00406AE8  . 60            PUSHAD
00406AE9  . 9C            PUSHFD
00406AEA  . C74424 28 60A1>MOV DWORD PTR SS:[ESP+28],202A160    #这个才是真正的 key
```

```
00406AF2 . 68 45E388B4    PUSH B488E345
00406AF7 . C60424 FD      MOV BYTE PTR SS:[ESP],0FD
00406AFB . 881C24         MOV BYTE PTR SS:[ESP],BL
00406AFE . C64424 04 6D   MOV BYTE PTR SS:[ESP+4],6D
00406B03 . C74424 28 6661>MOV DWORD PTR SS:[ESP+28],D5D26166    #模拟旧版 call 的返回地址,无用
00406B0B . 68 A0913B18    PUSH 183B91A0
00406B10 . FF3424         PUSH DWORD PTR SS:[ESP]
00406B13 . 8D6424 30      LEA ESP,DWORD PTR SS:[ESP+30]
00406B17 . ^E9 86DCFFFF   JMP test2_vm.004047A2
```

这里可以看出新版的變化，首先是抹除了 68 ?? ?? ?? ?? E8 這樣的特征碼，其次是不再使用 push 來寫入 key，上面那兩個 push 都是花指令來的，是用來迷惑使用 push 定位的。

無論新版還是舊版，那個 call 的返回地址都是沒有用的，因為 vmp 從來不會返回，因此新版里使用 JMP 時，丟失返回地址對 vmp 一點影響都沒有。

接着虚拟机入口，开始初始化

```
0103735A Main    CALL NOTEPAD_.01007418                ; ESP=0007FF2C
01007418 Main    JMP NOTEPAD_.010011DA
010011DA Main    PUSHFD                                ; ESP=0007FF28
010011DB Main    PUSHFD                                ; ESP=0007FF24
010011DC Main    MOV DWORD PTR SS:[ESP+4],EBX
010011E0 Main    PUSH DWORD PTR SS:[ESP]                ; ESP=0007FF20
010011E3 Main    MOV DWORD PTR SS:[ESP+4],EBP
010011E7 Main    PUSH B553C5B8                          ; ESP=0007FF1C
010011EC Main    PUSH EDX                              ; ESP=0007FF18
010011ED Main    PUSHFD                                ; ESP=0007FF14
010011EE Main    PUSHFD                                ; ESP=0007FF10
010011EF Main    LEA ESP,DWORD PTR SS:[ESP+14]          ; ESP=0007FF24
010011F3 Main    JMP NOTEPAD_.01007A58
01007A58 Main    CALL NOTEPAD_.01013AA8                ; ESP=0007FF20
01013AA8 Main    MOVSX BP,CL                           ; EBP=0007FFB0
01013AAC Main    XCHG DWORD PTR SS:[ESP],ECX           ; ECX=01007A5D
01013AAF Main    PUSHAD                                ; ESP=0007FF00
01013AB0 Main    MOV DWORD PTR SS:[ESP+1C],EDX
01013AB4 Main    PUSHFD                                ; ESP=0007FEFC
01013AB5 Main    XCHG DWORD PTR SS:[ESP+1C],ESI        ; ESI=01007A5D
01013AB9 Main    CALL NOTEPAD_.01007C4F                ; ESP=0007FEF8
01007C4F Main    JMP NOTEPAD_.01013255
01013255 Main    PUSHFD                                ; ESP=0007FEF4
01013256 Main    MOV DWORD PTR SS:[ESP+20],EAX
0101325A Main    MOVSX SI,DL                           ; ESI=0100FF94
0101325E Main    MOV DWORD PTR SS:[ESP+1C],EDX
01013262 Main    MOVSX EBP,DL                          ; EBP=FFFFFFFF94
01013265 Main    BSWAP SI                              ; ESI=01000000
```

```
01013268 Main    PUSHAD                                ; ESP=0007FED4
01013269 Main    PUSH AFD61C47                        ; ESP=0007FED0
0101326E Main    XCHG DWORD PTR SS:[ESP+3C],EDI      ; EDI=0007FF20
01013272 Main    SETO BH                             ; EBX=7FFD0000
01013275 Main    JMP NOTEPAD_.01013E2F
01013E2F Main    PUSHFD                             ; ESP=0007FECC
01013E30 Main    POP DWORD PTR SS:[ESP+38]          ; ESP=0007FED0
```

保存现场，ESP 不用保存～～

为什么不用保存呢，因为在 vmp 的架构里，ESP 是直接使用的，vmp 在任何时候都可以获得真正的 ESP，同时 vmp 把内部的堆栈和真正的堆栈进行叠加，从而增加追踪堆栈的难度，这里可以想像成不使用 EBP 的函数。

```
0007FF08 00000202 80.. ntdll.7C930738
0007FF0C 7C930738 80.. ntdll.7C930738
0007FF10 7C92EB94 80.. ntdll.KiFastSystemCallRe
0007FF14 0007FFB0 ?..
0007FF18 FFFFFFFF
0007FF1C 7C92EB94 80.. ntdll.KiFastSystemCallRe
0007FF20 0007FFB0 ?..
0007FF24 0007FFC0 ?..
0007FF28 7FFDA000 .狙△
0007FF2C 0103735F _s♥@ RETURN to NOTEPAD_.01037
0007FF30 ABE7DF63 c喂
0007FF34 7C930738 80.. ntdll.7C930738
0007FF38 FFFFFFFF
0007FF3C 7FFDA000 .狙△
```

现在堆栈等于进行了 pushad, pushfd 但 esp 位置随机用一个通用寄存器替代了。

```
01013E34 Main    LEA ESP,DWORD PTR SS:[ESP+38]      ; ESP=0007FF08
01013E38 Main    JNO NOTEPAD_.010143BC
010143BC Main    MOV ESI,ESI
010143BE Main    MOVZX SI,CL                        ; ESI=0100005D
010143C2 Main    MOV DI,SP                          ; EDI=0007FF08
010143C5 Main    PUSHAD                             ; ESP=0007FEE8
010143C6 Main    PUSH DWORD PTR DS:[01013D2C]      ; ESP=0007FEE4
010143CC Main    POP DWORD PTR SS:[ESP+1C]          ; ESP=0007FEE8
010143D0 Main    TEST CL,0C2
010143D3 Main    CMC                                ; FL=C
010143D4 Main    CLC                                ; FL=0
```

注意,这个是很重要的数据,在虚拟机随机算法解密时用,后面会有详细的介绍

开始解码虚拟机 EIP

```
010143D5 Main    MOV DWORD PTR SS:[ESP+18],0
```

这里虚拟机堆栈的顶端数据,作用是初始 EIP 的一个偏移和控制虚拟代码的解密偏移,这个数值一直是常量,我猜测这个变量 VMP 作者是为了提供一个统一接口,因为 VMP\_JMP 指令采用的不是统一的接口(会在堆栈上遗留跳转偏移作为校验,因此这里模拟跳转后的偏移)

```
010143DD Main    SAL BP,CL                          ; FL=PAZ, EBP=FFFF0000
```

```
010143E0 Main  MOV ESI,DWORD PTR SS:[ESP+48]          ; ESI=ABE7DF63    这个就是 call 之前压入堆栈的 key
010143E4 Main  AND BP,3E0A                            ; FL=PZ
010143E9 Main  XOR ESI,B28A20E3                      ; FL=0, ESI=196DFF80
010143EF Main  RCL BP,CL
010143F2 Main  SHR BX,CL                            ; FL=PAZ
010143F5 Main  NEG ESI                              ; FL=CS, ESI=E6920080
010143F7 Main  CMC                            ; FL=S
010143F8 Main  JMP NOTEPAD_.01013135
01013135 Main  ROR ESI,0F                        ; ESI=0101CD24
```

ESI 就是 VM 的起始地址，也相当于 VM 里面的 EIP

```
01013138 Main  XCHG EBP,EBX                        ; EBX=FFFFF000, EBP=7FFD0000
0101313A Main  CMP DI,3E6E                        ; FL=PAS
0101313F Main  SHLD BP,DI,CL                    ; FL=PA, EBP=7FFD1FE1
01013143 Main  LEA EBP,DWORD PTR SS:[ESP+18] ; EBP=0007FF00    建立虚拟机堆栈
01013147 Main  XCHG DI,BX                        ; EBX=FFFFFFF08, EDI=00070000
0101314A Main  BT DX,AX
0101314E Main  MOVSX ECX,BL                        ; ECX=00000008
01013151 Main  BSWAP EDI                        ; EDI=00000700
01013153 Main  SUB ESP,0A8                        ; FL=0, ESP=0007FE40
01013159 Main  SHL DI,CL                        ; FL=CPAZO, EDI=00000000
0101315C Main  SAL EBX,CL                        ; FL=CPAS, EBX=FFFFF0800
0101315E Main  PUSHFD                        ; ESP=0007FE3C
0101315F Main  LEA EDI,DWORD PTR SS:[ESP+4] ; EDI=0007FE40
```

这里是设置虚拟机的堆栈。

注意！现在 EDI 的值是 0007FE40

虚拟机申请的内存为 0007FF34-0007FE40=0xF4

这个版本里前面 0x40 是作为通用寄存器使用的，其他作为缓存堆栈，其算法会在下面看到。

缓存包括保存真正的寄存器 0x24，和一个返回地址，一个 key，一个全局控制变量

```
01013163 Main  PUSH EDX                        ; ESP=0007FE38
01013164 Main  JMP NOTEPAD_.01008294
01008294 Main  BSF CX,AX                        ; ECX=00000004
01008298 Main  MOV EBX,ESI                      ; EBX=0101CD24
0100829A Main  SAR AL,CL                        ; FL=AS, EAX=0007FFFB
0100829C Main  JMP NOTEPAD_.01013C7A
01013C7A Main  ADD ESI,DWORD PTR SS:[EBP]        ; FL=P    加上初始偏移,入口时这个应该为 0
01013C7D Main  PUSHAD                        ; ESP=0007FE18
01013C7E Main  SHRD CX,AX,3                        ; FL=CPAO, ECX=00006000
01013C83 Main  NOT CH                        ; ECX=00009F00
```

```
; FL=CPAS, EAX=0007FFFF
```

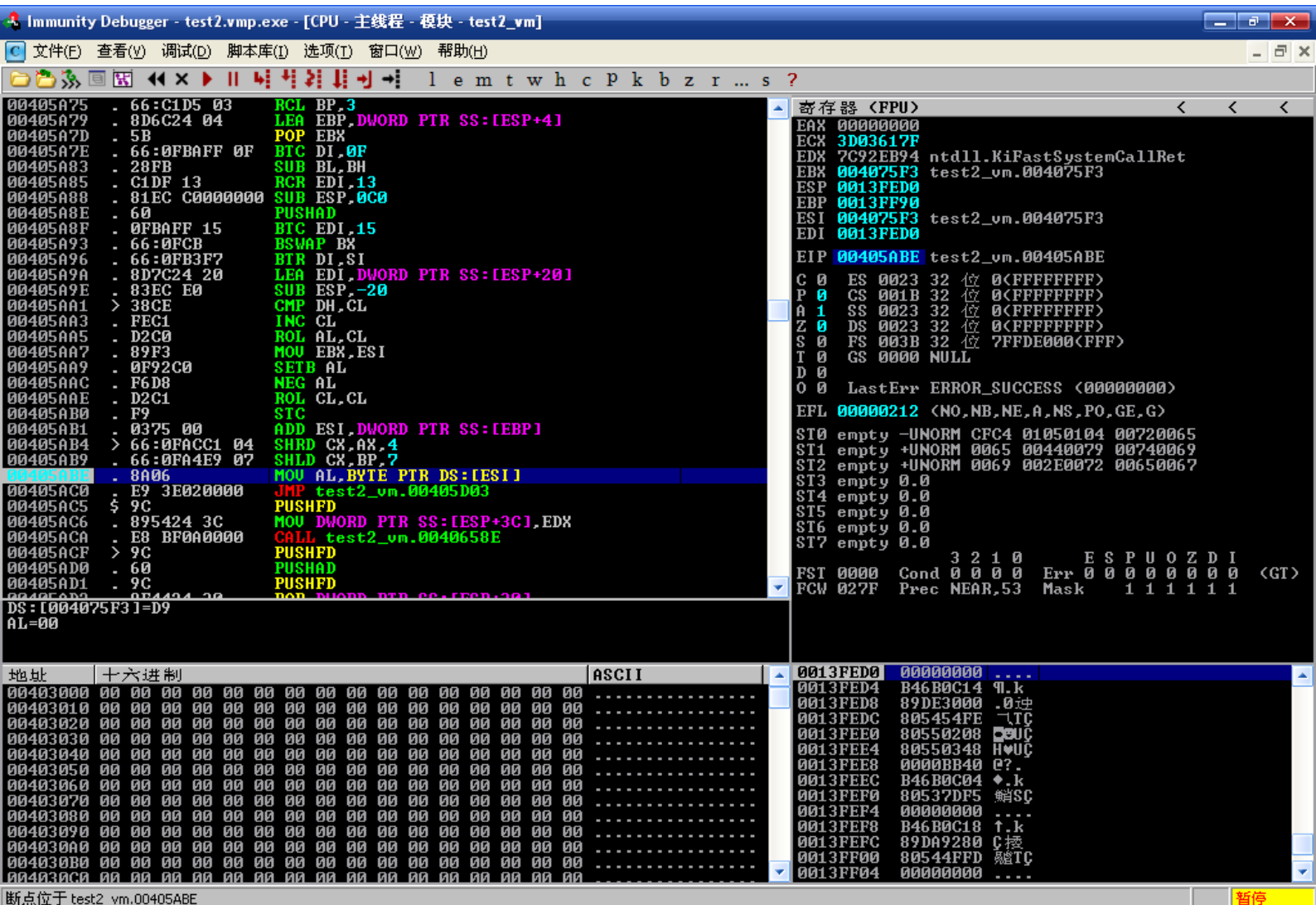
入口到这里也就完成了。

陷阱入口平时并不会被执行,估计是用来 anti 那些暴力猜测函数入口的追踪者的

入口如下:

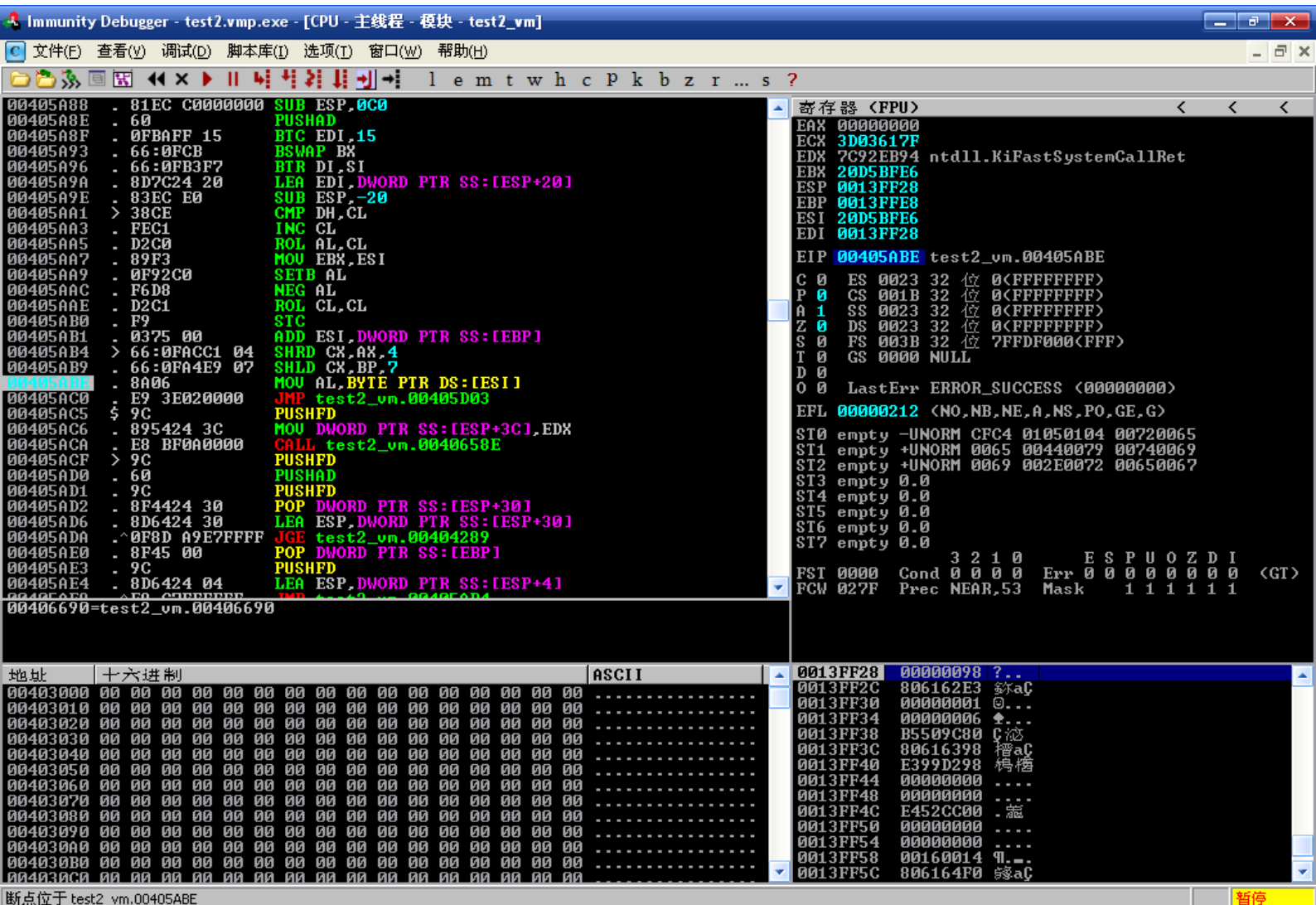


### 正常入口时



陷阱入口如下,可以看出,陷阱破坏了虚拟机的 eip,校验值和堆栈,破坏的值每次也不同,主要是因为入口没有写入正确的 key.





## 三. 基本架构

### VMP 寄存器的基本架构

**Esi** 保存 VM.eip

**Edi** 指向虚拟机通用寄存器的地址,看到[Edi+XXX]这样的东西都是虚拟机在操作内部变量

**Ebp** 指向虚拟机堆栈的栈顶,因为虚拟机的指令全部都是函数,因此要输入参数才能执行。

**Eax** 计算将要执行的指令序号和选择虚拟机通用寄存器,指令执行时用作中间变量.其它时候用作花指令。

**Edx** 执行时通常用作中间变量.其它时候用作花指令。

**Ebx** bl 用来保存校验码。

**Ecx** shld,shrd 时用作中间变量,其它大部分时间用来做花指令

**Esp** 指向垃圾,这个寄存器在虚拟机里是用来迷惑追踪者的。

寄存器的使用从我接触 VMP 到目前的 2.04 都一直没有变过,我试着理解一下 VMP 的作者为什么要这么做。

首先, X86 的寄存器对于特定的指令有着特别的作用, 例如 mul 的 eax, div 的 edx, AAA、RDTSC 等很多指令都需要用到指定的寄存器。

假设用 EAX 来保存 VM.eip 的话, 那么 EAX 的数据就不能随意丢失, 不能用作花指令, 而且执行像 AAM, shld, mul 等这些使用 EAX 寄存器的指令时, 也要将 EAX 保存, 所有与 EAX 相关的指令基本上都不能用来做花指令。而像现在使用 ESI 的话, 则只需要在 rep 这些指令时才需要保存, 保存的次数和影响都比较少, 目前的这个组合已经很优化了。

其实在这一层, VM 受硬件结构影响得比较大, 因此这一层主要的作用是将汇编指令转换成虚拟机的接口, 稳定比一切都重要, 强度并不是太重要, VMP 内部全部都使用了随机寄存器, 但是这里却一直都没有改变。

还有一种架构就是所有的数据都是由虚拟机内部的通用寄存器来保存, VMP 并不是这种架构, 这里不作讨论。

ESI 在 VMP 里代表的是虚拟机当前的 EIP, 为了防止静态扫描, ESI 分成了升序和降序两种, 并且全部都带一个偏移。

```
01013C88 Main      MOV AL,BYTE PTR DS:[ESI-1]          ; EAX=0007FF82
```

这里的 1 是可变的, 每次都不一样。

```
010073D4 Main      SUB ESI,1                          ; FL=0, ESI=0101CD23
```

这种就是升序, ESI 是往上走的, 同样, SUB 指令也会有 LEA 等形式。

EDI 指向的是 VMP 内部通用寄存器, 使用 EDI 时都是作为基址加上 EAX 的索引使用的, VMP 存取内部数据时都会使用像

```
MOV DWORD PTR DS:[EDI+EAX],EDX
```

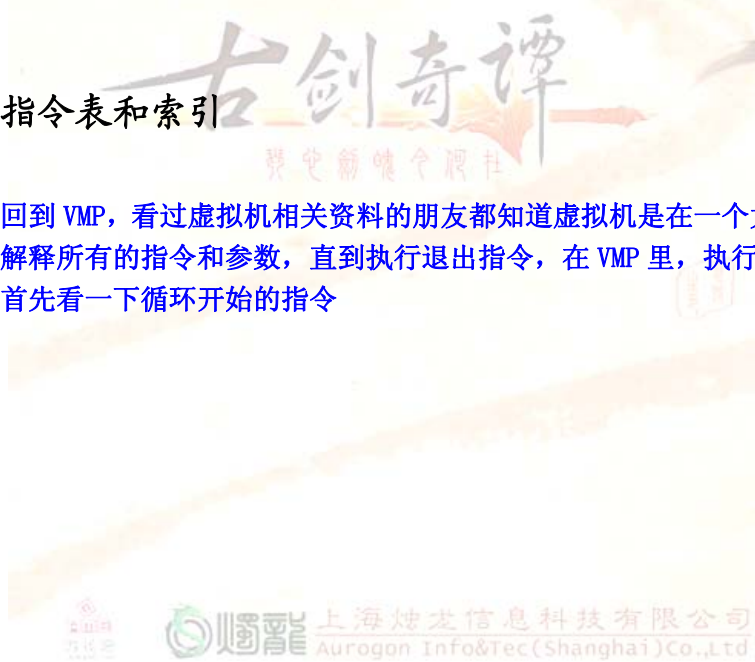
这样的指令。

EBP 指向的是虚拟机的栈顶, VMP 是基于堆栈的虚拟机, VMP 内部的堆栈是由真正的堆栈加上临时的数据组成, 指令使用的数据也是通过堆栈来存放。执行一个指令前, VMP 都会将该指令的参数放到堆栈上, 监视[EBP]会发现一些好玩的东西。

## 指令表和索引

回到 VMP, 看过虚拟机相关资料的朋友都知道虚拟机是在一个大的循环里面执行的, 现在我们就在这个大循环的起点, 这个循环将解释所有的指令和参数, 直到执行退出指令, 在 VMP 里, 执行退出指令并不代表退出虚拟机, 这个我们在后面就可以看到。

首先看一下循环开始的指令



01013C7A	0375 00	ADD ESI,DWORD PTR SS:[EBP]
01013C7D	60	PUSHAD
01013C7E	66:0FACC1 03	SHRD CX,AX,3
01013C83	F6D5	NOT CH
01013C85	C0F8 04	SAR AL,4
01013C88	8A46 FF	MOV AL,BYTE PTR DS:[ESI-1]
01013C8B	83EC D8	SUB ESP,-28
01013C8E	0F8C 0FF5FFFF	JL NOTEPAD_.010131A3
01013C94	66:C1D9 09	RCR CX,9
01013C98	66:0FB6C9	MOVZX CX,CL
01013C9C	10E9	ADC CL,CH
01013C9E	00D8	ADD AL,BL
01013CA0	80ED 7A	SUB CH,7A
01013CA3	F6D0	NOT AL
01013CA5	66:0FA4F9 0E	SHLD CX,DI,0E
01013CAA	E9 EE36FFFF	JMP NOTEPAD_.0100739D
01013CAF	F6D7	NOT BH
01013CB1	0F9FC3	SETG BL
01013CB4	894D 04	MOV DWORD PTR SS:[EBP+4],ECX
01013CB7	893C24	MOV DWORD PTR SS:[ESP],EDI
01013CBA	0FBED8	MOVSX EBX,AL
01013CBD	66:F7D3	NOT BX
01013CC0	8D1CDD B78887FF	LEA EBX,DWORD PTR DS:[EBX*8+FF87]
01013CC7	8955 00	MOV DWORD PTR SS:[EBP],EDX
01013CCA	0FBED9	MOVSX EBX,CL
01013CCD	0F9EC7	SETLE BH
01013CD0	8D1CE5 BE9E75DB	LEA EBX,DWORD PTR DS:[DB759EBE]
01013CD7	0F9AC3	SETPE BL
01013CDA	8B5C24 30	MOV EBX,DWORD PTR SS:[ESP+30]
01013CDE	66:897424 04	MOV WORD PTR SS:[ESP+4],SI
01013CE3	8B6424 04	MOV BYTE PTR SS:[ESP+4],AH
01013CE7	C60424 58	MOV BYTE PTR SS:[ESP],58
01013CEB	C64424 04 4E	MOV BYTE PTR SS:[ESP+4],4E

**Registers <FPU>**  
EAX 0007FFFF  
ECX 00009F00  
EDX 7C92EB94 ntdll.KiFastSystemCallRet  
EBX 0101CD24 NOTEPAD\_.0101CD24  
ESP 0007FE18  
EBP 0007FF00  
ESI 0101CD24 NOTEPAD\_.0101CD24  
EDI 0007FE40  
EIP 01013C88 NOTEPAD\_.01013C88  
C 1 ES 0023 32bit 0<FFFFFFFF>  
P 1 CS 001B 32bit 0<FFFFFFFF>  
A 1 SS 0023 32bit 0<FFFFFFFF>  
Z 0 DS 0023 32bit 0<FFFFFFFF>  
S 1 FS 003B 32bit 7FFDD000<FFF>  
T 0 GS 0000 NULL  
D 0  
O 0 LastErr ERROR\_NO\_IMPERSONATION\_TOKEN  
EFL 00000297 <NO,B,NE,BE,S,PE,L,LE>  
ST0 empty -UNORM CFBC 01050104 006D006D  
ST1 empty +UNORM 0067 00670075 00620065  
ST2 empty x#.19L  
ST3 empty 0.0  
ST4 empty 0.0  
ST5 empty 0.0  
ST6 empty 1.00000000000000000000  
ST7 empty 1.00000000000000000000  
3 2 1 0 E S P U O Z D  
FST 4020 Cond 1 0 0 0 Err 0 0 1 0 0 0 0  
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1

其中 ESI 和 EBX 是一样的，都是指向 opcode 的第一个地址

01013C88 Main	MOV AL,BYTE PTR DS:[ESI-1]	; EAX=0007FF82
01013C8B Main	SUB ESP,-28	; FL=C, ESP=0007FE40
01013C8E Main	JL NOTEPAD_.010131A3	
01013C94 Main	RCR CX,9	; ECX=000000CF
01013C98 Main	MOVZX CX,CL	
01013C9C Main	ADC CL,CH	; FL=AS, ECX=000000D0
01013C9E Main	ADD AL,BL	; FL=PS, EAX=0007FFA6
01013CA0 Main	SUB CH,7A	; FL=CAS, ECX=000086D0
01013CA3 Main	NOT AL	; EAX=0007FF59
01013CA5 Main	SHLD CX,DI,0E	; FL=PA, ECX=00003F90
01013CAA Main	JMP NOTEPAD_.0100739D	
0100739D Main	NEG AL	; FL=CAS, EAX=0007FFA7
0100739F Main	JE NOTEPAD_.01007BEA	
010073A5 Main	CMC	; FL=AS
010073A6 Main	ROL AL,5	; FL=ASO, EAX=0007FFF4
010073A9 Main	SAR CL,CL	; FL=CPAS, ECX=00003FFF
010073AB Main	BSR ECX,EDI	; ECX=00000012
010073AE Main	ADD BL,AL	; FL=CP, EBX=0101CD18 修正校验码

```
010073B0 Main    MOVZX CX,BL                      ; ECX=00000018
010073B4 Main    NEG ECX                        ; FL=CPAS, ECX=FFFFFFE8
010073B6 Main    MOVZX EAX,AL                  ; EAX=000000F4
```

这里的 EAX 已经解码完了，EAX 只是作为一个索引，在指令表里查找指令，这里有效的数据只有 AL，因此最多只有 256 个指令

```
010073B9 Main    PUSH ESP                      ; ESP=0007FE3C
010073BA Main    BSR ECX,EBX                      ; ECX=00000018
010073BD Main    MOV ECX,DWORD PTR DS:[EAX*4+10135A0] ; ECX=7B760001
```

这里的 10135A0 就是指令表，这个是硬编码，很容易就可以看出来，而 EAX 的作用和 handle 很像，因此网上有很多人都叫这个为 handle，指令表是这一层的 BOSS，和 IAT 一样，这里是重点的防御对象。

#### 首先看解码指令执行地址

```
010073C4 Main    PUSHFD                      ; ESP=0007FE38
010073C5 Main    CLC                      ; FL=PAS
010073C6 Main    BSWAP ECX                  ; ECX=0100767B
010073C8 Main    PUSHAD                      ; ESP=0007FE18
010073C9 Main    BT BP,7
010073CE Main    BT DX,9                      ; FL=CPAS
010073D3 Main    PUSHFD                      ; ESP=0007FE14
010073D4 Main    SUB ESI,1                  ; FL=0, ESI=0101CD23
010073D7 Main    PUSHFD                      ; ESP=0007FE10
010073D8 Main    ADD ECX,0                  ; FL=P
010073DE Main    LEA ESP,DWORD PTR SS:[ESP+30] ; ESP=0007FE40
010073E2 Main    JNB NOTEPAD_.010081BA
010081BA Main    JMP NOTEPAD_.01013FAB
01013FAB Main    PUSH 91B24AC9                ; ESP=0007FE3C
01013FB0 Main    MOV DWORD PTR SS:[ESP],ECX
01013FB3 Main    PUSHAD                      ; ESP=0007FE1C
01013FB4 Main    MOV BYTE PTR SS:[ESP+4],BL
01013FB8 Main    MOV BYTE PTR SS:[ESP],3A
01013FBC Main    PUSHAD                      ; ESP=0007FDFC
01013FBD Main    PUSH DWORD PTR SS:[ESP+40]          ; ESP=0007FDF8
01013FC1 Main    RETN 44                      ; ESP=0007FE40
```

ret 后就跳到指令去执行了。

#### 下面分析指令的执行

```
0100767B Main    PUSH 39F97BCD                ; ESP=0007FE3C
01007680 Main    LEA ESP,DWORD PTR SS:[ESP+4]          ; ESP=0007FE40
```

01007684 Main JNO NOTEPAD\_.01007A7F  
01007A7F Main CALL NOTEPAD\_.01001149 ; ESP=0007FE3C  
01001149 Main SETNS DL ; EDX=7C92EB01  
0100114C Main CALL NOTEPAD\_.01014019 ; ESP=0007FE38  
01014019 Main BTC DX,7 ; EDX=7C92EB81  
0101401E Main ROR AL,6 ; FL=CP, EAX=000000D3

解码虚拟机寄存器

01014021 Main SETL DL ; EDX=7C92EB00  
01014024 Main PUSH 1AC4BDE0 ; ESP=0007FE34  
01014029 Main BSWAP DX ; EDX=7C920000  
0101402C Main BSF DX,BX ; EDX=7C920003  
01014030 Main DEC AL ; FL=CPS, EAX=000000D2  
01014032 Main BTR DX,SI ; FL=PS  
01014036 Main RCL DL,CL  
01014038 Main ROR AL,2 ; FL=CPSO, EAX=000000B4  
0101403B Main ROL DL,CL ; FL=PS, EDX=7C920018  
0101403D Main AND AL,3C ; FL=0, EAX=00000034  
01014040 Main CALL NOTEPAD\_.0100763F ; ESP=0007FE30  
0100763F Main CLC  
01007640 Main MOV EDX,DWORD PTR SS:[EBP] ; EDX=00000000

取出指令操作数

01007643 Main PUSH DWORD PTR SS:[ESP+4] ; ESP=0007FE2C  
01007647 Main STC ; FL=C  
01007648 Main TEST ESP,C889EC02 ; FL=P  
0100764E Main ADD EBP,4 ; FL=0, EBP=0007FF04

修正堆栈

01007651 Main PUSH DWORD PTR SS:[ESP+4] ; ESP=0007FE28  
01007655 Main PUSHAD ; ESP=0007FE08  
01007656 Main MOV BYTE PTR SS:[ESP+4],0BD  
0100765B Main MOV DWORD PTR DS:[EDI+EAX],EDX

真正执行指令

0100765E Main PUSH DWORD PTR SS:[ESP+10] ; ESP=0007FE04  
01007662 Main LEA ESP,DWORD PTR SS:[ESP+3C] ; ESP=0007FE40  
01007666 Main JMP NOTEPAD\_.010074A9

可以很容易看出来, 这条就是 mov reg, const 指令了

至于 const 是什么东西, 就要看前面的指令把什么压入堆栈了, 有可能是常量, 也有可能是地址(当前面的指令模拟 lea 的时候)

接下来就开始指令识别了, 写个脚本来分析一下指令表

Table address : 0x10135A0

01008480

['1A', '49', '6F', 'B5', 'D5']

01013A6B

['5', 'D', '85', '89', '9E', 'AC', 'B4']

01013C04

['40', '44', '48', '4C', '50', '54', '58', '5C', '60', '64', '68', '6C', '70', '74', '78', '7C']

01013405

['B1', 'B6', 'CB']

01001306

['79', 'A4']

01001188

['0', '38', '5B', 'F9']

0101408C

['2', '8', 'E', '2C', '42', '71', '8E']

0101310E

['39', '83', 'A7', 'BB', 'C5']

0101420F

['4', '69', 'AA', 'DD']

01013514

['2A', '4E', '66', '8B', 'A3', 'AF', 'CE']

01007C16

['2E', '36', '4D', '7B']

01001017

['7A']

01007B84

['87']

01013E1A

['17', '18', '5E', 'AD']

0101411B

['41', 'ED']

0101349C

['13', '5D', '77', '95', 'A8', 'DF', 'F2']

0100755A

['10', '1D', '2F', '93', 'A5', 'CD', 'D3']

01013045

['73', 'BD', 'DA', 'EE']

0101301F

['34']

01007B0F

['29', '5A', 'B3', 'FA']

01007CA2

['6', '47', '9F', 'D2']

01007D24

['F', '20', 'B7', 'EA', 'F1']



010141E8  
[ '27', '45', '84' ]

01013E9D  
[ '94', 'E3', 'EF' ]

01013F88  
[ '7E', 'B0' ]

01013DB7  
[ '7', '76', '7D', '8F', '9B', 'BE' ]

01008320  
[ '5F', '86', '99', 'FE' ]

01013FC4  
[ 'A', '35', '72' ]

01014045  
[ '26', '2D', 'BA', 'F5' ]

01007546  
[ '15', 'B2', 'B8', 'E9' ]

01007455  
[ '21', '32', '62', 'C7', 'CF' ]

01001064  
[ '4F', '67', 'EB' ]

0101425A  
[ 'C', '3A', '82', 'A6', 'D6', 'D7' ]

01013C5B  
[ '14', '8C', 'BC', 'E7', 'FD' ]

01007E5C  
[ '7F', '8D', 'A9', 'AB', 'B9', 'F7' ]

01007C64  
[ '12', '80', 'C2', 'E2' ]

010139E5  
[ '1B', '1F', '2B', '9D', 'A2', 'D9', 'DE' ]

010010E8  
[ '19', '6D', '75' ]

010081EB  
[ '11', '61', 'C6' ]

0100106D  
[ 'B', '1C', '28', '8A', '97', '98', 'E1', 'FF' ]

010132FD  
[ '25', '46', '51', '56', '90', 'CA', 'FB' ]

010131F1  
[ '1', '30', '33' ]

010081F2  
[ '3B', '52', '63', 'E6', 'F3' ]

01007F74  
[ '31', '3C', '43', '4A', '53', 'A0' ]

01008276  
[ '37', '65', '9A', 'AE', 'F6' ]



010140F7	['3', '23', '55', '6B', 'A1', 'C3', 'DB']
01013EF8	['4B', '81', '88', 'C9']
01013CF9	['9', '3D', '3E', '59', '92', 'E5']
01007A7A	['24', '3F', '57', '91', '96', 'C1']
0100767B	['C0', 'C4', 'C8', 'CC', 'D0', 'D4', 'D8', 'DC', 'E0', 'E4', 'E8', 'EC', 'F0', 'F4', 'F8', 'FC']
0100807C	['16', '22', '6A', '9C', 'D1']
010073FD	['1E', '6E', 'BF']

-----

这里可以看出一些什么东西呢，首先，可以看到有很多 handle 都对应同一个指令，这个也是虚拟机常见的技术，我个人把它叫做残缺指令表。

上面的指令都是这个函数用到的指令，除了输入和输出的指令需要带寄存器索引作为参数，其它的指令正常情况下每条指令应该只对应一个 handle，这里多个 handle 对应一条指令是因为 VMP 把所有没有用到的指令全部去掉，改成使用的指令，目的是为了不让追踪者通过一个指令表而获得所有的指令特征，写出通用的追踪算法。在获得 bin 后，我们可以通过测试每一条指令而追踪出所有 VMP 真正的 handle。

第二种技术我们需要加密两个相同的程序才能看出来，我个人把它叫做随机指令表，这里的思想就是每次都使用不同的地址、不同的解密算法、不同的指令表、不同的 handle 来代表同一个指令。这是一种非常好的思想，可以很好地抵御人肉。

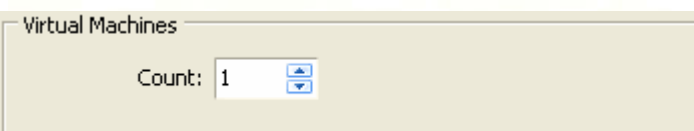
同样是加记事本

0106DE4E Main	MOV AL,BYTE PTR DS:[ESI]	; EAX=0000006E
0106DE50 Main	RCL CL,CL	; FL=CO, ECX=0106ED65
0106DE52 Main	OR CL,0	; FL=P
0106DE55 Main	BT SP,DI	
0106DE59 Main	XOR AL,BL	; FL=PS, EAX=00000099 //校验也是不一样的
0106DE5B Main	SHR CH,3	; FL=CPA, ECX=01061D65
0106DE5E Main	BSWAP ECX	; ECX=651D0601
0106DE60 Main	XOR AL,23	; FL=S, EAX=000000BA
0106DE62 Main	NOT CH	; ECX=651DF901
0106DE64 Main	ROL AL,6	; FL=SO, EAX=000000AE
0106DE67 Main	CALL 1_UPX.0106D700	; ESP=0007FECC
0106D700 Main	SUB AL,7F	; FL=AO, EAX=0000002F
0106D702 Main	SHL CX,0A	; FL=PA, ECX=651D0400
0106D706 Main	ADD ESI,1	; FL=0, ESI=01064D6E
0106D709 Main	SHL ECX,9	; FL=PA, ECX=3A080000
0106D70C Main	BTC ECX,ECX	; ECX=3A080001

```
0106D70F Main    SHLD CX,CX,CL                ; FL=A, ECX=3A080002
0106D713 Main    XOR BL,AL                    ; FL=PS, EBX=01064DD8    //校验
0106D715 Main    LEA ECX,DWORD PTR DS:[EDI*8+CAC6417C]    ; ECX=CB0637FC
0106D71C Main    MOVZX EAX,AL
0106D71F Main    SHLD CX,SP,0B                ; FL=CPAS, ECX=CB06E7F6
0106D724 Main    MOV ECX,DWORD PTR DS:[EAX*4+106D747]    ; ECX=D9AFD098
```

这个就是最基本的随机算法了。详细地下面再讲。

VMP 的指令表还有一种防御的技术，在 VMP 加密的时候有一个选项。



从字面上看，这里写的是虚拟机个数，这个个数和我们硬件上的 CPU 是不同的，这里的个数指的是执行环境，执行环境包括什么呢，就是不同的指令表，不同的解密算法，不同的寄存器(不知道这里是不是某大牛讲的轮转).....还有很多，总之所有随机的东西都会重新初始化。

我们来看个例子，这里我加密的个数为 2，也就是有两张指令表

## 两个指令表解密算法的比较

4067EC

Opcode instruction :0x004065A5      MOV AL,BYTE PTR DS:[ESI]

Opcode start :    0x004079D7  
                 sub const :      0

-----  
Get table :        0x00406CF7      MOV ECX,DWORD PTR DS:[EAX\*4+4067EC]

Table address : 0x004067EC  
-----

decode :

-----  
0x004065a5    MOV AL,BYTE PTR DS:[ESI]  
0x004065b2    XOR AL,BL  
0x004065bc    NEG AL  
0x004065cc    ADD AL,0B0  
0x00404be5    ROR AL,6  
0x00404bed    XOR BL,AL  
0x00404bf6    MOVZX EAX,AL  
0x00406cf7    MOV ECX,DWORD PTR DS:[EAX\*4+4067EC]  
0x00405a8d    INC ECX  
0x00405a96    ADD ECX,0

0x004048bf      INC ESI

-----

404DA9

Opcode instruction :0x00407420      MOV AL,BYTE PTR DS:[ESI-1]

Opcode start :    0x004079D7

sub const :      1

-----

Get table :      0x00407476      MOV ECX,DWORD PTR DS:[EAX\*4+404DA9]

Table address : 0x00404DA9

-----

decode :

-----

0x00407420      MOV AL,BYTE PTR DS:[ESI-1]

0x0040742b      XOR AL,BL

0x00407438      INC AL

0x00407441      NOT AL

0x0040744c      SUB AL,0B2

0x0040745c      XOR BL,AL

0x00407462      MOVZX EAX,AL

0x00407469      ADD ESI,-1

0x00407476      MOV ECX,DWORD PTR DS:[EAX\*4+404DA9]

0x0040670e      ROR ECX,0A

0x00406673      ADD ECX,0

-----

VMP 是怎么将两张指令表链接起来的呢，答案就是靠 VMP\_JMP 指令，这个指令是 VMP 里唯一跨虚拟机的指令，在入口时我们已经看过下半部分，现在看一下完整的。

004065A5    .    8A06      MOV AL,BYTE PTR DS:[ESI]

004065A7    .    66:C1C9 04    ROR CX,4

004065AB    .    66:0FA3DA    BT DX,BX

004065AF    .    F5      CMC

004065B0    .    F6DD      NEG CH

004065B2    .    30D8      XOR AL,BL

004065B4    .    0FABE9      BTS ECX,EBP

004065B7    .    B9 17472410    MOV ECX,10244717

004065BC    .    F6D8      NEG AL

004065BE    .    66:81F1 D0A7    XOR CX,0A7D0

004065C3    .    C0D5 04      RCL CH,4

004065C6    .    18F1      SBB CL,DH

```
004065C8 . 0FBAE6 1C BT ESI,1C
004065CC . 04 B0 ADD AL,0B0
004065CE . 66:0FBECB MOVSX CX,BL
004065D2 . 60 PUSHAD
004065D3 . F9 STC
004065D4 . E8 0CE6FFFF CALL test2_vm.00404BE5
00404BE5 $ C0C8 06 ROR AL,6
00404BE8 . 66:0FBEC8 MOVSX CX,AL
00404BEC . F8 CLC
00404BED . 30C3 XOR BL,AL
00404BEF . 8D8E C6F97A78 LEA ECX,DWORD PTR DS:[ESI+787AF9C6]
00404BF5 . 59 POP ECX
00404BF6 . 0FB6C0 MOVZX EAX,AL
00404BF9 . C64424 04 FD MOV BYTE PTR SS:[ESP+4],0FD
00404BFE . E8 F0200000 CALL test2_vm.00406CF3
00406CF3 $ 66:0FB6CA MOVZX CX,DL
00406CF7 . 8B0C85 EC6740>MOV ECX,DWORD PTR DS:[EAX*4+4067EC]
00406CFE . ^ E9 49E5FFFF JMP test2_vm.0040524C
0040524C > /E9 3A080000 JMP test2_vm.00405A8B
00405A8B > \9C PUSHFD
00405A8C . 9C PUSHFD
00405A8D . 41 INC ECX
00405A8E . 50 PUSH EAX
00405A8F . 53 PUSH EBX
00405A90 . F6C5 1E TEST CH,1E
00405A93 . 0FA3ED BT EBP,EBP
00405A96 . 81C1 00000000 ADD ECX,0
00405A9C . E8 1CEEFFFF CALL test2_vm.004048BD
004048BD $ 9C PUSHFD
004048BE . 9C PUSHFD
004048BF . 46 INC ESI
004048C0 . E9 92110000 JMP test2_vm.00405A57
00405A57 > \FF3424 PUSH DWORD PTR SS:[ESP]
00405A5A . E9 BC1A0000 JMP test2_vm.0040751B
0040751B > \9C PUSHFD
0040751C . 894C24 44 MOV DWORD PTR SS:[ESP+44],ECX
00407520 . 68 F4A508B7 PUSH B708A5F4
00407525 . FF7424 14 PUSH DWORD PTR SS:[ESP+14]
00407529 . 882C24 MOV BYTE PTR SS:[ESP],CH
0040752C . 881C24 MOV BYTE PTR SS:[ESP],BL
0040752F . FF7424 4C PUSH DWORD PTR SS:[ESP+4C]
00407533 . C2 5000 RETN 50
```

#注意指令表

接着执行 `vmp_jump`

```
00405310 . 66:D3C6 ROL SI,CL
00405313 . 0FBAEE 04 BTS ESI,4
00405317 . 66:21C6 AND SI,AX
0040531A . 8B75 00 MOV ESI,DWORD PTR SS:[EBP]
0040531D . F8 CLC
0040531E . 60 PUSHAD
0040531F . 83C5 04 ADD EBP,4
00405322 . 880C24 MOV BYTE PTR SS:[ESP],CL
00405325 . 51 PUSH ECX
00405326 . 9C PUSHFD
00405327 . 8D6424 28 LEA ESP,DWORD PTR SS:[ESP+28]
0040532B . E9 D2200000 JMP test2_vm.00407402
00407402 > \66:F7DB NEG BX
00407405 . 66:0FABD1 BTS CX,DX
00407409 . FEC7 INC BH
0040740B . 89F3 MOV EBX,ESI #初始化新的校验值
0040740D . 0FABF9 BTS ECX,EDI
00407410 . 86E8 XCHG AL,CH
00407412 . 88E0 MOV AL,AH
00407414 . 0375 00 ADD ESI,DWORD PTR SS:[EBP]
```

到这里就切换到另外一个指令表了。

古剑奇谭

```
00407417 > 0FA4D9 12 SHLD ECX,EBX,12
0040741B . |66:0FBAF1 04 BTR CX,4
00407420 . |8A46 FF MOV AL,BYTE PTR DS:[ESI-1]
00407423 . |66:39D7 CMP DI,DX
00407426 . |66:D3F9 SAR CX,CL
00407429 . |F6D5 NOT CH
0040742B . |30D8 XOR AL,BL
0040742D . |50 PUSH EAX
0040742E . |66:0FB6CB MOVZX CX,BL
00407432 . |0F90C5 SETO CH
00407435 . |8A2C24 MOV CH,BYTE PTR SS:[ESP]
00407438 . |FEC0 INC AL
0040743A . |80C1 12 ADD CL,12
0040743D . |9C PUSHFD
0040743E . |0F9AC1 SETPE CL
00407441 . |F6D0 NOT AL
00407443 . |66:19E9 SBB CX,BP
00407446 . |8D8E E342BB3E LEA ECX,DWORD PTR DS:[ESI+3EBB42E3]
0040744C . |2C B2 SUB AL,0B2
0040744E . |80E1 8A AND CL,8A
00407451 . |66:F7D1 NOT CX
00407454 . |68 8ECADC25 PUSH 25DCCA8E
```

```
00407459 . |66:39FE      CMP SI,DI
0040745C . |30C3        XOR BL,AL
0040745E . |30E1        XOR CL,AH
00407460 . |08FD        OR CH,BH
00407462 . |0FB6C0      MOVZX EAX,AL
00407465 . |66:0FB6C8   MOVZX CX,AL
00407469 . |83C6 FF     ADD ESI,-1
0040746C . |66:0FB6CB   MOVZX CX,BL
00407470 . |9C          PUSHFD
00407471 . |66:0FBAE9 0E BTS CX,0E
00407476 . |8B0C85 A94D40>MOV ECX,DWORD PTR DS:[EAX*4+404DA9]
0040747D . |66:0FBAE4 03 BT SP,3
00407482 . |E8 81F2FFFF CALL test2_vm.00406708
```

#这里就是另外一个指令表

通过这条指令,就可以把多个虚拟机串起来,不断地切换指令表和算法。

## 原子指令

上面我们看到了每个 **handle** 都对应一条指令,那么这些都是些什么指令呢,早期的虚拟机(个人认为),那时的 **handle** 和 **X86** 汇编指令是相对应的,例如 **E9** 的 **handle** 就是 **JMP** 这样子,现在应该已经看不到这种东西了。同样,如果你拿 **VMP** 的指令和 **X86** 汇编指令比较的话,应该没有一条是相同的,为什么呢,因为 **VMP** 把所有的 **X86** 汇编指令全部都拆散了,每条指令都只是其中汇编的一部分,要把这些指令在进行组合才能形成真正的 **X86** 汇编指令。这些指令是 **VMP** 最小的执行单位,我个人叫原子指令。指令的设计对 **VMP** 这种用来保护的虚拟机是很重要的,个人认为原子指令的条数越少那么强度就越高,因为通过追踪指令的特征来识别真正的指令就越困难。

**VMP** 大部分的原子指令特征码都是固定的, **VMP** 的强度并不依靠这些指令,除了一些例外的指令。

什么特殊的指令呢,最明显的就是输入和输出的指令。

例如下面这个,这个指令我将它命名为

**VMP\_PUSHC\_IMM32**

也就是向堆栈压入一个通用寄存器

```
0x00404CC1      NOT AL
0x00404CCC      ROR AL,6
0x00404CDD      NEG AL
0x00404CE6      AND AL,3C
0x00404CEF      MOV EDX,DWORD PTR DS:[EAX+EDI]
0x00404CFA      SUB EBP,4
0x00406867      MOV DWORD PTR SS:[EBP],EDX
```

这个指令有什么不同呢,我才重新加密一次

```
0x004051CB      XOR AL,27
0x004051D4      ROR AL,7
```

```
0x004051DE      XOR AL,0A9
0x004051E1      AND AL,3C
0x00405007      MOV EDX,DWORD PTR DS:[EAX+EDI]
0x00405011      SUB EBP,4
0x00404283      MOV DWORD PTR SS:[EBP],EDX
```

先抛开地址不说，红色的指令全部都改变了，这里和指令表解密算法一样，采用了随机算法，但是下面的指令都是一样的，还是因为在这一层，VMP 受到硬件的影响，没有采用随机寄存器，所以这里的比较好追的。  
这些算法在每个虚拟机里都是不一样的，看看多虚拟机的情况

#### VMP\_PUSHC\_IMM8

```
0x004055CA      SUB AL,BL
0x0040730B      INC AL
0x0040647E      NOT AL
0x004075CB      ROL AL,7
0x004075D7      SUB BL,AL
0x004075DA      MOV AL,BYTE PTR DS:[EAX+EDI]
0x004075E2      SUB EBP,4
0x00404D72      MOV WORD PTR SS:[EBP],AX
```

#### VMP\_PUSHC\_IMM8

```
0x004061CB      XOR AL,BL
0x004061E1      INC AL
0x00405501      ROR AL,3
0x00404931      NOT AL
0x00404934      XOR BL,AL
0x0040493E      MOV AL,BYTE PTR DS:[EAX+EDI]
0x0040633F      SUB EBP,4
0x0040634C      MOV WORD PTR SS:[EBP],AX
```

除了算法，指令的 handle 和地址也全部改变

instname : VMP\_PUSHC\_IMM8

0x004055B0

['0x06', '0x29', '0x5A', '0x8C', '0x9B', '0xCA']

-----  
instname : VMP\_PUSHC\_IMM8

0x00405A0F

['0x08', '0x3C', '0x57', '0x62', '0xA3', '0xEB']  
-----

除了随机算法，这里还有一样很重要的东西，这样东西是整个 VMP 加密体系的基础。  
我们来看一下这条指令。

-----  
0100767B

['C0', 'C4', 'C8', 'CC', 'D0', 'D4', 'D8', 'DC', 'E0', 'E4', 'E8', 'EC', 'F0', 'F4', 'F8', 'FC']  
-----

0100767B	68 CD7BF939	PUSH 39F97BCD
01007680	8D6424 04	LEA ESP,DWORD PTR SS:[ESP+4]
01007684	0F81 F5030000	JNO NOTEPAD_.01007A7F
01007A7F	E8 C596FFFF	CALL NOTEPAD_.01001149
01001149	0F99C2	SETNS DL
0100114C	E8 C82E0100	CALL NOTEPAD_.01014019
01014019	66:0FBABA 07	BTC DX,7
0101401E	C0C8 06	ROR AL,6
01014021	0F9CC2	SETL DL
01014024	68 E0BDC41A	PUSH 1AC4BDE0
01014029	66:0FCA	BSWAP DX
0101402C	66:0FBCD3	BSF DX,BX
01014030	FEC8	DEC AL
01014032	66:0FB3F2	BTR DX,SI
01014036	D2D2	RCL DL,CL
01014038	C0C8 02	ROR AL,2
0101403B	D2C2	ROL DL,CL
0101403D	80E0 3C	AND AL,3C
01014040	E8 FA35FFFF	CALL NOTEPAD_.0100763F
0100763F	F8	CLC
01007640	8B55 00	MOV EDX,DWORD PTR SS:[EBP]
01007643	FF7424 04	PUSH DWORD PTR SS:[ESP+4]
01007647	F9	STC
01007648	F7C4 02EC89C8	TEST ESP,C889EC02
0100764E	83C5 04	ADD EBP,4
01007651	FF7424 04	PUSH DWORD PTR SS:[ESP+4]
01007655	60	PUSHAD
01007656	C64424 04 BD	MOV BYTE PTR SS:[ESP+4],0BD
0100765B	891407	MOV DWORD PTR DS:[EDI+EAX],EDX

现在我们假设不知道 AL 的值，我们写个脚本穷举一下。

0x00	['01', '02', '03', '04', '40', '41', '42', '43', '80', '81', '82', '83', 'C0', 'C1', 'C2', 'C3']
0x04	['05', '06', '07', '08', '44', '45', '46', '47', '84', '85', '86', '87', 'C4', 'C5', 'C6', 'C7']
0x08	['09', '0A', '0B', '0C', '48', '49', '4A', '4B', '88', '89', '8A', '8B', 'C8', 'C9', 'CA', 'CB']
0x0C	['0D', '0E', '0F', '10', '4C', '4D', '4E', '4F', '8C', '8D', '8E', '8F', 'CC', 'CD', 'CE', 'CF']
0x10	['11', '12', '13', '14', '50', '51', '52', '53', '90', '91', '92', '93', 'D0', 'D1', 'D2', 'D3']
0x14	['15', '16', '17', '18', '54', '55', '56', '57', '94', '95', '96', '97', 'D4', 'D5', 'D6', 'D7']
0x18	['19', '1A', '1B', '1C', '58', '59', '5A', '5B', '98', '99', '9A', '9B', 'D8', 'D9', 'DA', 'DB']
0x1C	['1D', '1E', '1F', '20', '5C', '5D', '5E', '5F', '9C', '9D', '9E', '9F', 'DC', 'DD', 'DE', 'DF']
0x20	['21', '22', '23', '24', '60', '61', '62', '63', 'A0', 'A1', 'A2', 'A3', 'E0', 'E1', 'E2', 'E3']
0x24	['25', '26', '27', '28', '64', '65', '66', '67', 'A4', 'A5', 'A6', 'A7', 'E4', 'E5', 'E6', 'E7']
0x28	['29', '2A', '2B', '2C', '68', '69', '6A', '6B', 'A8', 'A9', 'AA', 'AB', 'E8', 'E9', 'EA', 'EB']
0x2C	['2D', '2E', '2F', '30', '6C', '6D', '6E', '6F', 'AC', 'AD', 'AE', 'AF', 'EC', 'ED', 'EE', 'EF']
0x30	['31', '32', '33', '34', '70', '71', '72', '73', 'B0', 'B1', 'B2', 'B3', 'F0', 'F1', 'F2', 'F3']
0x34	['35', '36', '37', '38', '74', '75', '76', '77', 'B4', 'B5', 'B6', 'B7', 'F4', 'F5', 'F6', 'F7']
0x38	['39', '3A', '3B', '3C', '78', '79', '7A', '7B', 'B8', 'B9', 'BA', 'BB', 'F8', 'F9', 'FA', 'FB']
0x3C	['00', '3D', '3E', '3F', '7C', '7D', '7E', '7F', 'BC', 'BD', 'BE', 'BF', 'FC', 'FD', 'FE', 'FF']

这个指令我将它命名为 **VMP\_POPC\_IMM32**，也就是从堆栈弹出一个 **DWORD** 到通用寄存器，那么到底是哪个寄存器呢，就要看通过 AL 解密出来的值，因为解密算法里有一句 **AND AL,3C**，所以这个值必定是 0-3C，从上面可以看到，这里解出来的值都是以 **DWORD** 为单位的，每一个 **DWORD** 都代表一个 32 位的通用寄存器（寄存器只是我个人的理解），而每个 **handle** 都对一个寄存器，也就是说指令的寄存器信息已经包含在 **handle** 当中了。这里不知道是不是因为 **handle** 数量问题，**VMP** 在处理 **WORD** 和 **BYTE** 时并没有采用这种架构，而是直接把寄存器硬编码在数据中。

仔细观察一下 **handle**，会发现这里是以 4 递增的，但是这个不能作为判定指令的标准，因为这个 **VMP** 随时会改变的，例如。

0x00404CBC

['C0', 'C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9', 'CA', 'CB', 'CC', 'CD', 'CE', 'FF']

VMP 的原子指令里, 个人认为最复杂的就是 VMP\_CRC 指令了, 这个指令不是用来模拟 X86 汇编, 而是用作校验数据, 这个指令如下

这个指令使用的参数

[EBP]保存校验的地址,[EBP+4]保存校验的大小

```
0007FF20 0101350E 7500 NOTEPAD_.0101350E
0007FF24 00000064 d...
0007FF28 010343C0 繡♥ NOTEPAD_.010343C0
0007FF2C 0101F284 勻 NOTEPAD_.0101F284
0007FF30 00000000 ....
0007FF34 7C930738 8摺 ntdll.7C930738
0007FF38 FFFFFFFF .....
0007FF3C 7FFDA000 .担△
0007FF40 E460BED8 靠
0007FF44 00000000 ....
0007FF48 00000000 .....
0007FF4C E5E96008 珍
```

可以看到,这里校验的是 0x0101350E,大小是 0x64

接下来看代码

```
01008320 8D14BD 04982177 LEA EDX,DWORD PTR DS:[EDI*4+77219804]
01008327 E8 AEFDFFFF CALL NOTEPAD_.010080DA
010080DA 8B55 00 MOV EDX,DWORD PTR SS:[EBP] //取出地址
010080DD 66:0FBBF9 BTC CX,DI
010080E1 FEC1 INC CL
010080E3 83C5 04 ADD EBP,4
010080E6 -E9 7EB00000 JMP NOTEPAD_.01013169
01013169 66:0FACF9 0E SHRD CX,DI,0E
0101316E 66:81D9 AD07 SBB CX,7AD
01013173 66:0FB6CB MOVZX CX,BL
01013177 29C0 SUB EAX,EAX //EAX 初始化为 0
01013179 83C4 04 ADD ESP,4
0101317C 66:F7D1 NOT CX //循环的起始
0101317F 66:21E1 AND CX,SP
01013182 89C1 MOV ECX,EAX
01013184 0F8B C20B0000 JPO NOTEPAD_.01013D4C //这里是假跳转
0101318A C1E0 07 SHL EAX,7
0101318D 0FA3D7 BT EDI,EDX
01013190 81FD A2B39380 CMP EBP,8093B3A2
01013196 9C PUSHFD
01013197 80FA D2 CMP DL,0D2
```

<b>0101319A</b>	<b>C1E9 19</b>	<b>SHR ECX,19</b>	
0101319D	F5	CMC	
0101319E	E8 F9110000	CALL NOTEPAD_.0101439C	
0101439C	80F9 EE	CMP CL,0EE	
<b>0101439F</b>	<b>09C8</b>	<b>OR EAX,ECX</b>	
010143A1	C60424 A3	MOV BYTE PTR SS:[ESP],0A3	
<b>010143A5</b>	<b>3202</b>	<b>XOR AL,BYTE PTR DS:[EDX]</b>	//来了,校验
010143A7	54	PUSH ESP	
<b>010143A8</b>	<b>42</b>	<b>INC EDX</b>	
010143A9	C60424 79	MOV BYTE PTR SS:[ESP],79	
010143AD	C64424 04 67	MOV BYTE PTR SS:[ESP+4],67	
010143B2	60	PUSHAD	
<b>010143B3</b>	<b>FF4D 00</b>	<b>DEC DWORD PTR SS:[EBP]</b>	
010143B6	9C	PUSHFD	
010143B7	E8 4832FFFF	CALL NOTEPAD_.01007604	
01007604	8D6424 34	LEA ESP,DWORD PTR SS:[ESP+34]	
<b>01007608</b>	<b>-0F85 6EBB0000</b>	<b>JNZ NOTEPAD_.0101317C</b>	//标准的循环,这里检测是否跳出循环
0100760E	68 477DEDE7	PUSH E7ED7D47	
01007613	9C	PUSHFD	
01007614	9C	PUSHFD	
<b>01007615</b>	<b>8945 00</b>	<b>MOV DWORD PTR SS:[EBP],EAX</b>	//保存校验值
01007618	882C24	MOV BYTE PTR SS:[ESP],CH	
0100761B	8D6424 0C	LEA ESP,DWORD PTR SS:[ESP+C]	
0100761F	^E9 85FEFFFF	JMP NOTEPAD_.010074A9	

## VMP 的堆栈分配策略

首先我们来看一个指令 **VMP\_PUSH\_DWORD\_CONST**, 这个指令向堆栈压入一个 **DWORD** 常量, 这个常量同样使用了随机算法来加密。

01013514	66:0FB6C3	MOVZX AX,BL	
01013518	66:0FABC0	BTS AX,AX	
0101351C	66:0FA4F0 0D	SHLD AX,SI,0D	
<b>01013521</b>	<b>8B46 FC</b>	<b>MOV EAX,DWORD PTR DS:[ESI-4]</b>	//取出数据
01013524	F8	CLC	
01013525	66:0FA3D0	BT AX,DX	
01013529	F8	CLC	
0101352A	9C	PUSHFD	
<b>0101352B</b>	<b>0FC8</b>	<b>BSWAP EAX</b>	//开始解密
0101352D	83C4 04	ADD ESP,4	
01013530	-0F86 2846FFFF	JBE NOTEPAD_.01007B5E	
01013536	60	PUSHAD	
<b>01013537</b>	<b>01D8</b>	<b>ADD EAX,EBX</b>	
01013539	F8	CLC	
<b>0101353A</b>	<b>C1C8 1D</b>	<b>ROR EAX,1D</b>	

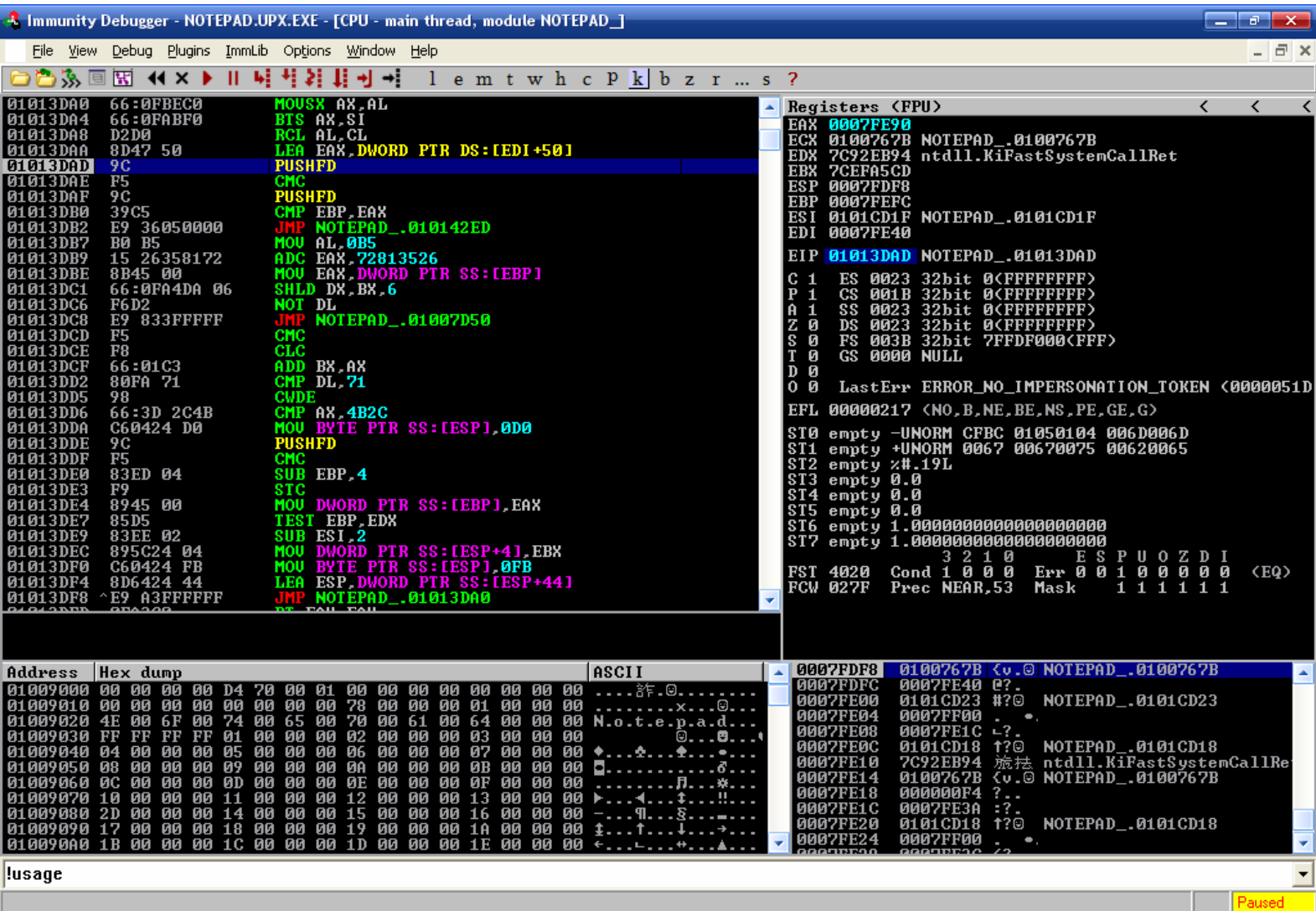
0101353D	60	PUSHAD	
0101353E	80FD FF	CMP CH,0FF	
01013541	F7D8	NEG EAX	
01013543	83C4 40	ADD ESP,40	
01013546	-0F86 E64DFFFF	JBE NOTEPAD_.01008332	
0101354C	66:0FBAE4 03	BT SP,3	
01013551	F5	CMC	
01013552	84FB	TEST BL,BH	
01013554	83EE 04	SUB ESI,4	//修正 eip
01013557	80FE 6B	CMP DH,6B	
0101355A	F8	CLC	
0101355B	0FC8	BSWAP EAX	
0101355D	80FA 76	CMP DL,76	
01013560	01C3	ADD EBX,EAX	//修正校验字
01013562	F5	CMC	
01013563	83ED 04	SUB EBP,4	
01013566	E8 AB45FFFF	CALL NOTEPAD_.01007B16	
01007B16	8945 00	MOV DWORD PTR SS:[EBP],EAX	//压入堆栈

到这里,指令的功能就已经实现了,接下来 VMP 就要进行堆栈的检查

01007B19	9C	PUSHFD	
01007B1A	68 52527461	PUSH 61745252	
01007B1F	8D6424 0C	LEA ESP,DWORD PTR SS:[ESP+C]	
01007B23	-E9 78C20000	JMP NOTEPAD_.01013DA0	
01013DA0	66:0FBEC0	MOVSX AX,AL	
01013DA4	66:0FABF0	BTS AX,SI	
01013DA8	D2D0	RCL AL,CL	
01013DAA	8D47 50	LEA EAX,DWORD PTR DS:[EDI+50]	//注意这里
01013DAD	9C	PUSHFD	
01013DAE	F5	CMC	
01013DAF	9C	PUSHFD	
01013DB0	39C5	CMPEBP,EAX	
01013DB2	E9 36050000	JMP NOTEPAD_.010142ED	
010142ED	60	PUSHAD	
010142EE	8D6424 28	LEA ESP,DWORD PTR SS:[ESP+28]	
010142F2	-0F87 B131FFFF	JA NOTEPAD_.010074A9	//这里跳走则代表堆栈正常,不需要重新分配

这里首先取出 EDI+50 的地址,这个地址作为一条警戒线,这 0x50 堆栈里有 0x40 是寄存器,而 VMP 目前使用最多堆栈的是 VMP\_CPUID 指令,一共消耗 0x10,因此 0x50 是一个安全的数值.

然后比较 EBP 是否已经超过了警戒线,超过了则要重新分配堆栈,现在寄存器的状态如下



现在堆栈的状态是

ESP=0007FDF8

中间保存的都是一些垃圾

EDI=0007FE40 //虚拟机分配内存的顶部

这段区域保存的是虚拟机的寄存器,大小是 0x40

EAX=0007FE90 //作为一条警戒线

EBP=0007FEFC //指向虚拟机堆栈的栈顶

ebp=真正的 esp+虚拟机运行时堆栈的数据

返回的地址 //当计算 EIP 后,这两个是没有用的,返回时不会用这个地址

计算 VMEip 的 key

把 EBP 修改成 0007FE80, 使 VMP 强行分配堆栈

接下来开始调整堆栈

010142F8	80DE 47	SBB DH,47
010142FB	37	AAA
010142FC	C1E8 10	SHR EAX,10
010142FF	89E2	MOV EDX,ESP
01014301	66:D3E8	SHR AX,CL
01014304	B1 3A	MOV CL,3A
01014306	66:0FADE9	SHRD CX,BP,CL
0101430A	66:81C1 17E4	ADD CX,0E417
0101430F	8D4F 40	LEA ECX,DWORD PTR DS:[EDI+40]
01014312	66:0FBAEF 02	BTS DI,2
01014317	0FBCC3	BSF EAX,EBX
0101431A	0F91C4	SETNO AH
0101431D	8D0465 364C3D34	LEA EAX,DWORD PTR DS:[343D4C36]
01014324	29D1	SUB ECX,EDX

这里计算出寄存器加垃圾的大小,等一下这部分要一起往上移,如果不加上垃圾,那么在执行 RETN XXX 的时候就会破坏堆栈平衡

01014326	F5	CMC
01014327	27	DAA
01014328	F5	CMC
01014329	8D45 80	LEA EAX,DWORD PTR SS:[EBP-80]

把堆栈的警戒线提高 0x80

0101432C	66:89D7	MOV DI,DX
0101432F	80E0 FC	AND AL,0FC
01014332	66:0FACCF 0D	SHRD DI,CX,0D
01014337	29C8	SUB EAX,ECX

加上垃圾和寄存器的新位置

01014339	66:0FB6FB	MOVZX DI,BL
0101433D	9C	PUSHFD

0101433E	89C4	MOV ESP,EAX
----------	------	-------------

修正堆栈

```
01014340  89E7          MOV EDI,ESP
01014342  9C            PUSHFD
01014343  66:89D7       MOV DI,DX
01014346  0FB6FA        MOVZX EDI,DL
01014349  8DBE 6DDD45D6 LEA EDI,DWORD PTR DS:[ESI+D645DD6D]
0101434F  56            PUSH ESI
```

保存 VMEip

```
01014350  FD            STD
01014351  66:BE AC93    MOV SI,93AC
01014355  FC            CLD
01014356  89D6          MOV ESI,EDX
```

mov esi, esp

```
01014358  FD            STD
01014359  FD            STD
0101435A  F7D7          NOT EDI
0101435C  8D7C08 C0      LEA EDI,DWORD PTR DS:[EAX+ECX-40]
```

计算新的 EDI,这时的 EDI 指向通用寄存器的底部。

```
01014360  FC            CLD
01014361  FD            STD
01014362  68 0C69E750   PUSH 50E7690C
01014367  873C24        XCHG DWORD PTR SS:[ESP],EDI
0101436A  E8 D136FFFF   CALL NOTEPAD_.01007A40
```

这时寄存器的值如下

EAX 0007FD78

ECX 00000088

EDX 0007FDF8

EBX 7CEFA5CD

ESP 0007FD68

EBP 0007FE80

ESI 0007FDF8

EDI 50E7690C

EIP 01007A40 NOTEPAD\_.01007A40

0007FD68 0101436F oC r RETURN to NOTEPAD\_.0101436F from NOTEPAD\_.01007A40

0007FD6C 0007FDC0 例•.

0007FD70 0101CD1F ? r NOTEPAD\_.0101CD1F

其中 EAX 保存修正后的 ESP

ECX 通用寄存器加垃圾的大小

[ESP+4]保存准备放新寄存器的位置

[ESP+8]保存的是 VMEip

EDX 保存旧的 ESP

EBP 还是指向虚拟机堆栈数据

ESI 保存旧的 ESP

这里保存 VMEip 的作用是为了下面的 rep movsb 指令

```
01007A40  66:0FB6F8      MOVZX DI,AL
01007A44  89C7           MOV EDI,EAX
01007A46  -E9 95B90000    JMP NOTEPAD_.010133E0
010133E0  FC            CLD
010133E1  891424         MOV DWORD PTR SS:[ESP],EDX
010133E4  F3:A4         REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
```

开始复制数据,这时的 ECX= 00000088, 将通用寄存器和垃圾一起进行移动

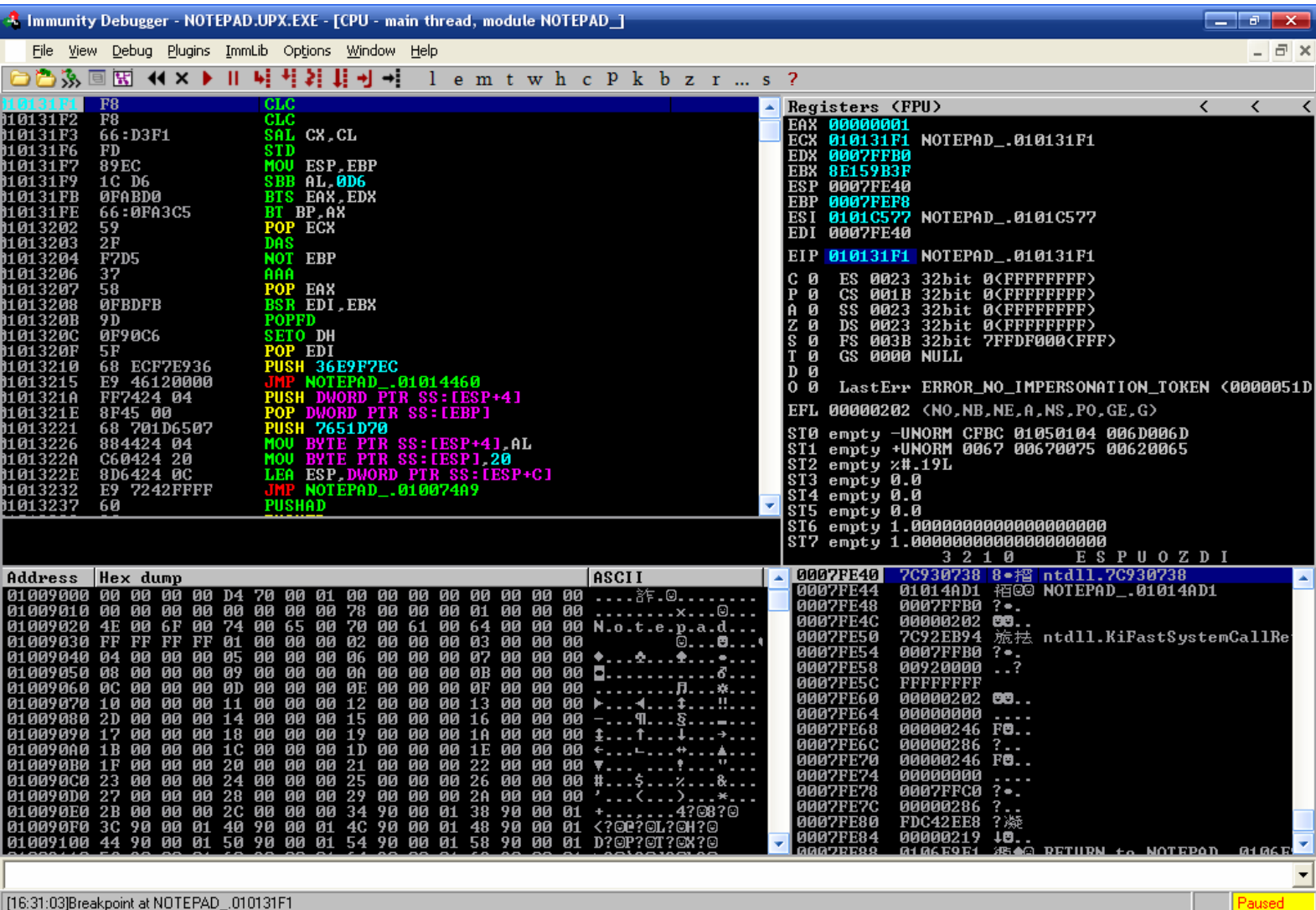
```
010133E6  66:0FBBD7      BTC DI,DX
010133EA  8B7C24 04      MOV EDI,DWORD PTR SS:[ESP+4]
010133EE  0FBAEE 04      BTS ESI,4
010133F2  68 F2593916    PUSH 163959F2
010133F7  8B7424 0C      MOV ESI,DWORD PTR SS:[ESP+C]
010133FB  66:0FBAE2 09   BT DX,9
01013400  E9 690B0000    JMP NOTEPAD_.01013F6E
01013F6E  FF7424 10      PUSH DWORD PTR SS:[ESP+10]
01013F72  9D            POPFD
01013F73  C70424 205015AF MOV DWORD PTR SS:[ESP],AF155020
01013F7A  884424 04      MOV BYTE PTR SS:[ESP+4],AL
01013F7E  60            PUSHAD
01013F7F  8D6424 34      LEA ESP,DWORD PTR SS:[ESP+34]
01013F83  -E9 2135FFFF    JMP NOTEPAD_.010074A9
```

VMP 在所有使用堆栈的指令 (也就是执行 SUB EBP,xxxxx) 后都会进行堆栈的检查。

为什么要进行堆栈检查呢,因为 VMP 的堆栈策略是这样的,在进入 VMP 时,作者计算了足够的堆栈,正常的情况下是不用再分配的,但是加密的函数存在 push 指令时就不一样了,在这个时候,VMP 会将所有的暂存数据移出堆栈,放入寄存器,然后将真正的数据压入堆栈,有了这些数据, VMP 开始分配的堆栈则存在溢出的可能,因此 VMP 每个向堆栈写入数据的指令都要进行堆栈检查。

## 三. VMP 的出口

执行出口指令时, 状态如下



EAX 00000001

ECX 010131F1 NOTEPAD\_.010131F1

EDX 0007FFB0

EBX 8E159B3F

ESP 0007FE40

EBP 0007FEF8

ESI 0101C577 NOTEPAD\_.0101C577

EDI 0007FE40

EIP 010131F1 NOTEPAD\_.010131F1

可以看到,现在 ESP=EDI,垃圾已经没有了.

0007FEF8 B0 FF 07 00 00 00 00 00 97 02 00 00 8B 54 02 36 ?•.....?..娘 6

0007FF08 6D 48 01 01 D1 9C 7E 84 A9 81 75 5B 03 63 86 31 mH 亅 裏~劑莖[<sup>L</sup>c?

0007FF18 5A 87 AA 58 1D 65 5E 88 F8 0A 95 1C 61 A3 01 01 Z 嚙 X e^培.?a?亅

0007FF28 C0 43 03 01 84 F2 01 01 00 00 00 00 38 07 93 7C 繡<sup>L</sup> 亅 勻 亅 亅 ....8• 摺

这里的寄存器都和一个随机数进行了异或

010343C0 是真正的返回地址, 0101A361 是中间函数的地址

010131F1	F8	CLC
010131F2	F8	CLC
010131F3	66:D3F1	SAL CX,CL
010131F6	FD	STD
010131F7	89EC	MOV ESP,EBP

虚拟机的寄存器已经没有用了.

010131F9	1C D6	SBB AL,0D6
010131FB	0FABD0	BTS EAX,EDX
010131FE	66:0FA3C5	BT BP,AX
01013202	59	POP ECX
01013203	2F	DAS
01013204	F7D5	NOT EBP
01013206	37	AAA
01013207	58	POPE AX
01013208	0FBDFB	BSR EDI,EBX
0101320B	9D	POPFD
0101320C	0F90C6	SETO DH
0101320F	5F	POP EDI
01013210	68 ECF7E936	PUSH 36E9F7EC
01013215	E9 46120000	JMP NOTEPAD_.01014460
01014460	F6D6	NOT DH
01014462	8B5424 04	MOV EDX,DWORD PTR SS:[ESP+4]
01014466	66:0FBEC3	MOVSX AX,BL
0101446A	98	CWDE
0101446B	8B4424 08	MOV EAX,DWORD PTR SS:[ESP+8]
0101446F	66:87D6	XCHG SI,DX
01014472	8D8B 2F6DD368	LEA ECX,DWORD PTR DS:[EBX+68D36D2F]
01014478	E8 41F6FFFF	CALL NOTEPAD_.01013ABE
01013ABE	8D14C5 D229E1DA	LEA EDX,DWORD PTR DS:[EAX*8+DAE129D2]
01013AC5	8B7424 10	MOV ESI,DWORD PTR SS:[ESP+10]
01013AC9	66:87EA	XCHG DX,BP
01013ACC	66:87CB	XCHG BX,CX
01013ACF	8B5424 14	MOV EDX,DWORD PTR SS:[ESP+14]
01013AD3	68 86DADC76	PUSH 76DCDA86
01013AD8	8D6424 1C	LEA ESP,DWORD PTR SS:[ESP+1C]
01013ADC	0F8A 1248FFFF	JPE NOTEPAD_.010082F4
010082F4	66:0FBECB	MOVSX CX,BL
010082F8	66:0FCB	BSWAP BX
010082FB	59	POP ECX



```
010082FC  ^E9 01FFFFFF    JMP NOTEPAD_.01008202
01008202  F7D3            NOT EBX
01008204  0F9FC3          SETG BL
01008207  5D              POP EBP
01008208  8D9F D8318C31   LEA EBX,DWORD PTR DS:[EDI+318C31D8]
0100820E  E8 11FEFFFF     CALL NOTEPAD_.01008024
01008024  88D7            MOV BH,DL
01008026  8B5C24 04       MOV EBX,DWORD PTR SS:[ESP+4]
0100802A  68 5F883FC1     PUSH C13F885F
0100802F  FF7424 0C       PUSH DWORD PTR SS:[ESP+C]
01008033  C2 1000         RETN 10
```

返回了,在没有加密寄存器的情况下,这里将返回外部,这里返回的是一个中间函数,这个函数的地址是硬编码在 `opcode` 中的,这时返回的是一个中间函数,这个函数将解密寄存器,然后再返回外部

☒ 离开虚拟机时加密寄存器 (降低运行速度)

`vmp` 在加密时对生成了每个寄存器生成一个随机数

例如这里的 `EDI` 是 `0x4A9153B3`

那么将执行

```
XOR EDI,4A9153B3    #这条指令指令是在 vmp 解释成原子指令在解释器里执行的
XOR EDI,4A9153B3    #这条则硬编码在出口的中间函数里
```

这种技术可以防止某些比较有特征的常数一直在虚拟机的通用寄存器里出现,追踪出口则不受影响。

而且 `VMP` 这里有点奇怪,在执行 `ret` 指令退出虚拟机时,并不会加密寄存器,而使用 `call` 等指令退出时,则会加密寄存器,我不清楚 `VMP` 作者这样考虑的原因。

寄存器状态:

`EAX` 847E9CD1

`ECX` 58AA875A

`EDX` 31866303

`EBX` 1C950AF8

`ESP` 0007FF08

`EBP` 885E651D

`ESI` 5B7581A9

`EDI` 3602548B

可以看到,除了 `ESP`,其它的都不是明文

```
0101A361  60              PUSHAD
0101A362  E9 DF650100     JMP NOTEPAD_.01030946
01030946  9C              PUSHFD
01030947  8F4424 1C       POP DWORD PTR SS:[ESP+1C]
```

0103094B 51 PUSH ECX  
0103094C 81F7 B353914A XOR EDI,4A9153B3  
01030952 9C PUSHFD  
01030953 FF3424 PUSH DWORD PTR SS:[ESP]  
01030956 81F1 EA78AD58 XOR ECX,58AD78EA  
0103095C 9C PUSHFD  
0103095D 81F6 567E8AA4 XOR ESI,A48A7E56  
01030963 ^E9 937AFFFF JMP NOTEPAD\_.010283FB  
010283FB F6C2 55 TEST DL,55  
010283FE 35 61637984 XOR EAX,84796361  
01028403 F5 CMC  
01028404 81F3 F80A951C XOR EBX,1C950AF8  
0102840A 66:895424 18 MOV WORD PTR SS:[ESP+18],DX  
0102840F 81F5 DD9A5988 XOR EBP,88599ADD  
01028415 F9 STC  
01028416 81F2 9788144D XOR EDX,4D148897  
到这里全部寄存器都恢复了,ESP 除外

0102841C F8 CLC  
0102841D C60424 27 MOV BYTE PTR SS:[ESP],27  
01028421 817424 2C D10000>XOR DWORD PTR SS:[ESP+2C],0D1  
01028429 66:F7C5 A54C TEST BP,4CA5  
0102842E FF7424 2C PUSH DWORD PTR SS:[ESP+2C]  
01028432 9D POPFD  
01028433 FF7424 04 PUSH DWORD PTR SS:[ESP+4]  
01028437 FF7424 34 PUSH DWORD PTR SS:[ESP+34]  
0102843B C2 3800 RETN 38

出口到这里就结束了。

这一层我所知道的也就分析完了，我个人认为分析每个原子指令是追踪虚拟机所必须的，虚拟机是算法之间的对抗，在人肉一次分析完所有的指令并理解作者的思想后才能继续往上追，写出自动识别指令的算法是必须的，在这一层的指令识别最大难度就是随机指令表和使用随机算法的指令。从我开始跟踪 VMP 开始，这些东西就基本上没有什么变化，感觉上就只是花指令变麻烦了。完成指令识别后我们就可以进入 VMP，看看虚拟机真正的威力。

上海烛龙信息科技有限公司  
Aurogon Info&Tec(Shanghai)Co.,Ltd

## 四. VMP 解释器执行

这里基本上和 X86 的汇编没有什么关系了，这里执行的都是 VMP 的原子指令，寄存器特征也全部抹掉了，里面全部都是随机算法和随机寄存器，到 2.04 版为止，VMP 共使用 16 个 32 位的通用寄存器作为内部数据的存放。在这里要做些什么呢，答案还是识别指令和去花指令，在 VMP 内部识别指令要比上面的难多了，我们一起来看看为什么。

就我而言，我并不关心 VMP 是如何实现 AND、OR、XOR 等指令，第一是因为我水平太差，第二是因为在获得 VMP 的 bin 之后，可以让 VMP 加密相关指令而追出相关的实现方法，关于 NAND 和 NOR 的资料请自行百度或者到左上角的论坛搜索相关资料。

## VMP 简单的执行流程

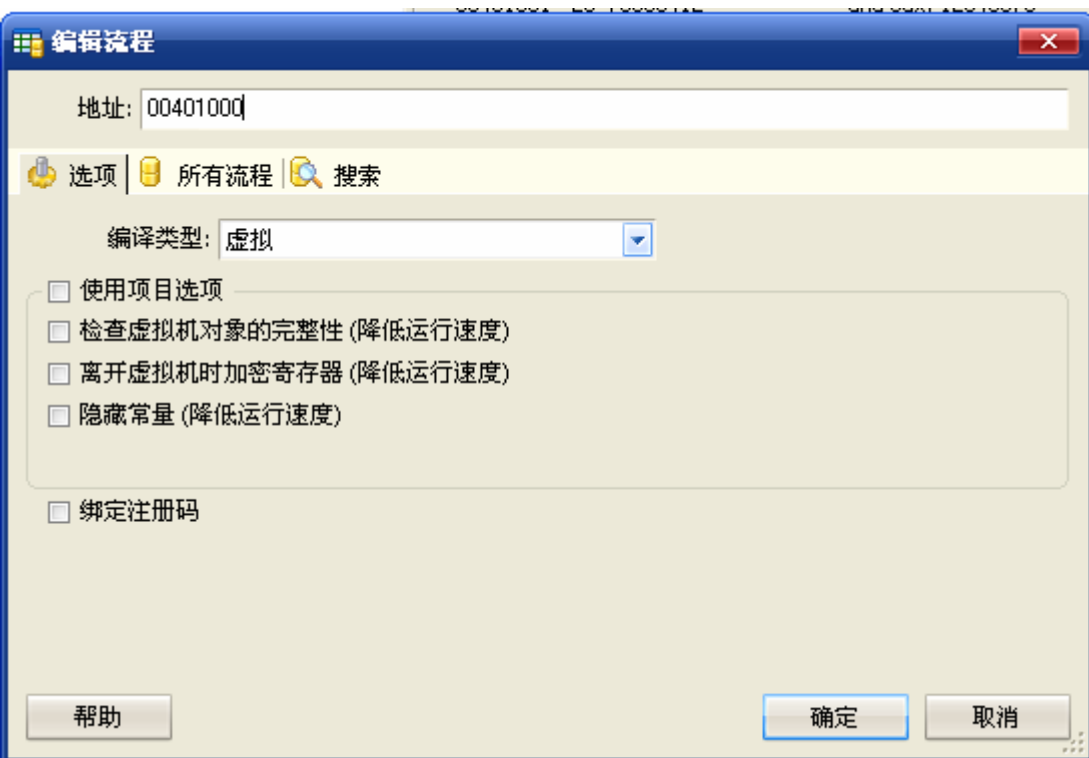
首先以 `and eax,12345678h` 来测试一下 vmp,这里的版本是 2.04

入口的寄存器状态如下

```
寄存器 (FPU) < < <
EAX 00000000
ECX 0013FFB0
EDX 7C92EB94 ntdll.KiFastSystemCallRet
EBX 7FFD5000
ESP 0013FFC4
EBP 0013FFF0
ESI FFFFFFFF
EDI 7C930738 ntdll.7C930738
EIP 00401001 test2_vm.<模块入口点>
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 1 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 00000246 <NO,NB,E,BE,NS,PE,GE,LE>
ST0 empty -UNORM CFC4 01050104 00720065
ST1 empty +UNORM 0065 00440079 00740069
ST2 empty +UNORM 0069 002E0072 00650067
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
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
```

首先是最快速度的





看一下执行的原子指令,非常简单,就几十条指令

VMP\_POPC\_IMM32 REG\_18

VMP\_PUSH\_DWORD\_CONST 0x5F81466E

ADD\_DWORD 0x5F81466E, 0x0 (0x5F81466E)

#这里加的是虚拟压入的偏移量,在我追踪的时候,这个变量从来没  
#有起过作用,这个 0x5F81466E 也是随机的

VMP\_POPC\_IMM32 REG\_C

VMP\_POPC\_IMM32 REG\_8

VMP\_POPC\_IMM32 REG\_1C

VMP\_POPC\_IMM32 REG\_3C

VMP\_POPC\_IMM32 REG\_28

VMP\_POPC\_IMM32 REG\_14

VMP\_POPC\_IMM32 REG\_0

VMP\_POPC\_IMM32 REG\_2C

VMP\_POPC\_IMM32 REG\_30

VMP\_POPC\_IMM32 REG\_C

VMP\_POPC\_IMM32 REG\_38

VMP\_POPC\_IMM32 REG\_34

VMP\_POPC\_IMM32 REG\_24

这里是 vmp 里面的一条 pushad 指令,保存所有进入 vmp 时的寄存器。

VMP\_PUSH\_DWORD\_CONST 0xEDCBA987

VMP\_PUSHC\_IMM32 REG\_38

VMP\_PUSHC\_IMM32 REG\_38

NAND\_DWORD 0x0,0x0 (0xFFFFFFFF)

很明显这里的 REG\_38 就是 EAX

VMP\_POPC\_IMM32 REG\_4

NAND\_DWORD 0xFFFFFFFF,0xEDCBA987 (0x0)

VMP\_POPC\_IMM32 REG\_34

VMP\_POPC\_IMM32 REG\_4

这里的常量 0xEDCBA987 是 not 0x12345678 ,这里的指令并不是

push 12345678

push 12345678

NAND

而是直接 push EDCBA987 是为了干扰指令的识别,让指令 NAND 条数变得随机

VMP\_PUSHC\_IMM32 REG\_4

VMP\_PUSHC\_IMM32 REG\_C

VMP\_PUSHC\_IMM32 REG\_30

VMP\_PUSHC\_IMM32 REG\_2C

VMP\_PUSHC\_IMM32 REG\_0

VMP\_PUSHC\_IMM32 REG\_14

VMP\_PUSHC\_IMM32 REG\_34

VMP\_PUSHC\_IMM32 REG\_3C

VMP\_PUSHC\_IMM32 REG\_1C

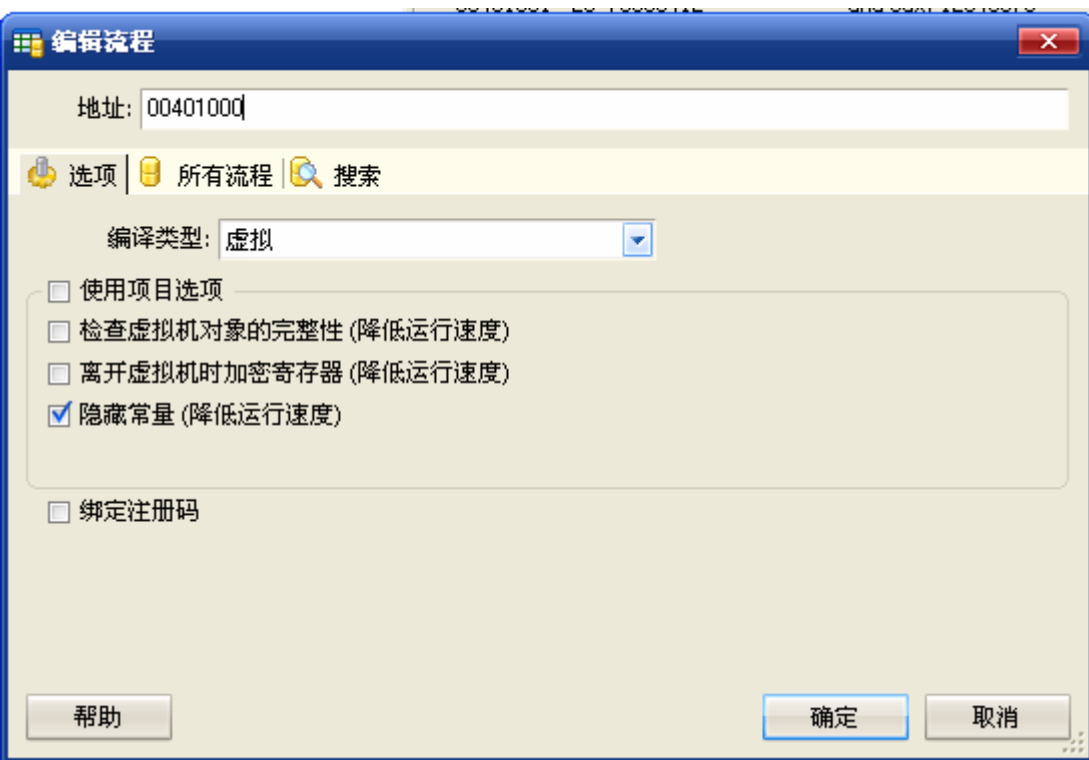
VMP\_PUSHC\_IMM32 REG\_34

VMP\_PUSHC\_IMM32 REG\_10

最后就是 VMP 的 popad,弹出寄存器后就执行 vmp\_exit 退出虚拟机.

接下来,增加难度





选取隐藏常量后会有什么变化呢

首先抛去上面的 `pushad` 和 `popad` and 指令的变化如下

```
VMP_PUSH_DWORD_CONST 0x12345678
VMP_PUSH_DWORD_CONST 0x105470
VMP_PUSHVM_ESP
VMP_MOVIN_WORD_ADDR_SS [0x13FFBC] (0x5470)
VMP_POPC_IMM8 REG_18
NAND_DWORD 0x105470, 0x12345678 (0xEDCBA987)
```

看到 0xEDCBA987 是不是很熟悉呢,这里就是 VMP 的随机算法,从这里可以看出几个有趣的现象.

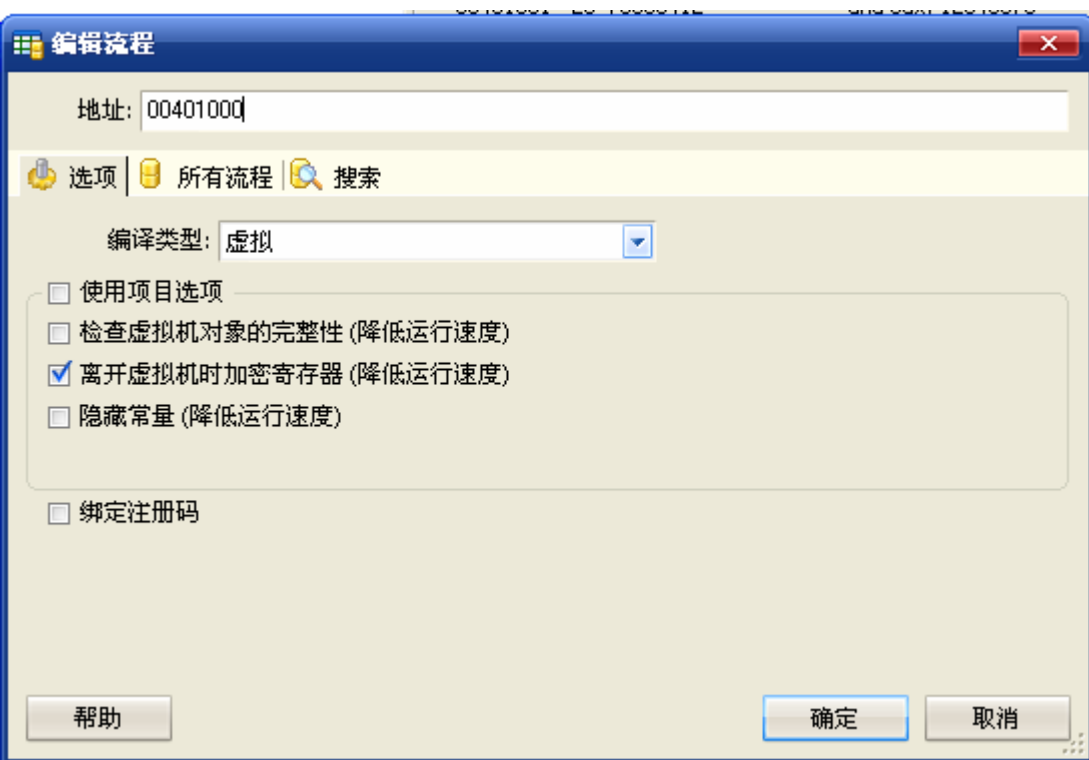
第一,正常来讲这里执行的是 `not 0x12345678`,但是这里第二个常量为 0x105470,这是因为 `vmp` 在加密时生成了一个随机掩码然后将常量和掩码进行与,掩码在加密后丢弃,这样就可以干扰数据的分析,在 `hook` 住 `NAND` 的情况下也无法确定这个是不是 `not` 指令,随机掩码和上面的残缺算法都是为了尽量抹除用 `NAND` 指令模拟其它指令时的特征。

```
VMP_PUSHVM_ESP
VMP_MOVIN_WORD_ADDR_SS [0x13FFBC] (0x5470)
VMP_POPC_IMM8 REG_18
```

这个是非常标准的花指令,请参考花指令分析。

VMP 将所有的常量都使用了这种技术,并且把很多真正的指令都用带常量的模板进行替换,因此选了隐藏常量后(这个选项默认是打勾的),强度会有很大的提高。这里的随机算法是整个 VMP 强度的来源,识别这里的指令要比下层难得多(还是个人认为),在这里,我再也找不到一条相同的指令和一个相同的寄存器了,就算找到一些特征,也不应该用来识别指令,因为下个版本这些特征有可能就挂掉了。

接着看



这里还是以 EAX 为例

NAND\_DWORD 0xFFFFFFFF, 0xEDCBA987 (0x0)

VMP\_POPC\_IMM32 REG\_18

VMP\_POPC\_IMM32 REG\_0

这时的 REG\_0 代表 EAX

VMP\_PUSHC\_IMM32 REG\_0

VMP\_POPC\_IMM32 REG\_1C

移动到 REG\_1C

VMP\_PUSHC\_IMM32 REG\_1C

VMP\_PUSHC\_IMM32 REG\_1C

NAND\_DWORD 0x0, 0x0 (0xFFFFFFFF)

VMP\_POPC\_IMM32 REG\_34

VMP\_PUSH\_DWORD\_CONST 0xFD234D27

NAND\_DWORD 0xFD234D27, 0xFFFFFFFF (0x0)

VMP\_POPC\_IMM32 REG\_8

VMP\_PUSHC\_IMM32 REG\_1C

VMP\_PUSH\_DWORD\_CONST 0x2DCB2D8

NAND\_DWORD 0x2DCB2D8, 0x0 (0xFD234D27)

VMP\_POPC\_IMM32 REG\_20

NAND\_DWORD 0xFD234D27, 0x0 (0x2DCB2D8)

VMP\_POPC\_IMM32 REG\_24

很明显这里执行了 xor eax, 0x2DCB2D8

再看出口

地址	HEX 数据	反汇编	注释
004070FE	9C	PUSHFD	
004070FF	8F0424	POP DWORD PTR SS:[ESP]	
00407102	F8	CLC	
00407103	81F7 E2C1BEE6	XOR EDI, E6BEC1E2	
00407109	F8	CLC	
0040710A	66:0FBAE2 04	BT DX, 4	
0040710F	81F3 13CC9822	XOR EBX, 2298CC13	
00407115	66:0FBAE2 05	BT DX, 5	
0040711A	813424 85080000	XOR DWORD PTR SS:[ESP], 885	
00407121	38E5	CMP CH, AH	
00407123	66:0FBAE7 0B	BT DI, 0B	
00407128	81F5 644CD676	XOR EBP, 76D64C64	
0040712E	84D4	TEST AH, DL	
00407130	81F6 F991BB5C	XOR ESI, 5CBB91F9	
00407136	F9	STC	
00407137	35 D8B2DC02	XOR EAX, 2DCB2D8	
0040713C	84FF	TEST BH, BH	
0040713E	F9	STC	
0040713F	81F2 36E3240F	XOR EDX, 0F24E336	
00407145	9C	PUSHFD	
00407146	9C	PUSHFD	
00407147	81F1 3D624F39	XOR ECX, 394F623D	
0040714D	F5	CMC	
0040714E	FF7424 08	PUSH DWORD PTR SS:[ESP+8]	
00407152	9D	POPPD	
00407153	68 D86D099B	PUSH 9B096DD8	
00407158	FF7424 10	PUSH DWORD PTR SS:[ESP+10]	

这里就是恢复 eax 的指令

## VMP 跳转

首先用随机算法生成两个地址的 key

VMP\_PUSH\_DWORD\_CONST 0x696F2C6D

VMP\_PUSH\_DWORD\_CONST 0xB1B55788

VMP\_PUSH\_DWORD\_CONST 0x4E8AF8C5

ADD\_DWORD 0x4E8AF8C5, 0xB1B55788 (0x40504D)

#解密数据表

VMP\_POPC\_IMM32 REG\_14

VMP\_PUSHC\_IMM32 REG\_20

#压入控制寄存器

ADD\_DWORD 0x0, 0x40504D (0x40504D)

VMP\_POPC\_IMM32 REG\_1C

VMP\_MOVIN\_DWORD\_ADDR\_DS [0x40504D] (0xCBB60F66)

ADD\_DWORD 0xCBB60F66, 0x696F2C6D (0x35253BD3)

VMP\_PUSH\_DWORD\_CONST 0x31012240

NAND\_DWORD 0x31012240, 0x35253BD3 (0xCADAC42C) #这个就是其中一个 key

VMP\_POPC\_IMM32 REG\_1C

第二个 key

VMP\_PUSH\_DWORD\_CONST 0xA94A4178

VMP\_PUSH\_DWORD\_CONST 0xC237D1A2

VMP\_PUSH\_DWORD\_CONST 0x3E087EAF

ADD\_DWORD 0x3E087EAF, 0xC237D1A2 (0x405051)

VMP\_PUSHC\_IMM32 REG\_20

ADD\_DWORD 0x0, 0x405051 (0x405051)

VMP\_POPC\_IMM32 REG\_0

VMP\_MOVIN\_DWORD\_ADDR\_DS [0x405051] (0x8BDAF766)

ADD\_DWORD 0x8BDAF766, 0xA94A4178 (0x352538DE)

VMP\_POPC\_IMM32 REG\_14

VMP\_PUSH\_DWORD\_CONST 0x24042098

NAND\_DWORD 0x24042098, 0x352538DE (0xCADAC721) #这里是第二个 key

VMP\_POPC\_IMM32 REG\_0

这时的堆栈如下

OllyDbg - test2.vmp.exe - [CPU - 主线程, 模块 - test2\_vm]

文件(F) 查看(V) 调试(D) 插件(P) 选项(O) 窗口(W) 帮助(H)

L E M T W H C / K B R ... S

地址 HEX 数据 反汇编 注释

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 &gt; F9 STC

00405D47 . 66:D3E0 SHL AX, CL

00405D4A . 9C PUSHFD

00405D4E . 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

DS:[00407B9C]=C8

AL=00

地址 HEX 数据 ASCII

0013FFD0 86 02 00 00 83 02 00 00 94 EB 92 7C B8 EB 00 00 ?..?. 旗挂...

0013FFED 80 FF 13 00 07 0A 00 00 FF FF FF FF 00 B1 00 00 ?..?..?..?

0013FFED 00 00 00 00 00 E0 FD 7F 00 E0 FD 7F F0 FF 13 00 ?..?..?..?

0013FF00 56 02 00 00 83 95 3D 20 38 07 93 7C 00 00 00 00 V...e? 8\*措...

0013FF10 88 08 AD 88 18 00 00 00 00 4C 96 B4 40 64 E2 E2 ?墟...L附@测

0013FF20 01 00 00 00 00 00 00 00 98 00 00 00 E3 62 61 80 ?..?..?..?..?..?..?..?

0013FF30 01 00 00 00 06 00 00 00 80 4C 96 B4 98 63 61 80 ?..?..?..?..?..?..?..?

0013FF40 B0 63 B2 E2 00 00 00 00 00 00 00 00 08 B0 D9 E3 测测...?..?..?..?..?..?..?..?

0013FF50 00 00 00 00 79 B3 D3 34 12 54 40 00 B1 2C A4 64 ...y秤4ITe? 白

0013FF60 64 FF 13 00 82 02 00 00 00 07 93 7C 28 7C 40 00 d ..?..?..?..?..?..?..?..?

0013FF70 F0 FF 13 00 88 FF 13 00 00 E0 FD 7F 94 EB 92 7C ?..?..?..?..?..?..?..?

0013FF80 B0 FF 13 00 00 00 00 00 82 02 00 00 82 02 00 00 ?..?..?..?..?..?..?..?

0013FF90 06 02 00 00 63 95 3D 20 83 02 00 00 F0 FF 13 00 ?..?..?..?..?..?..?..?

0013FFA0 B0 FF 13 00 16 02 00 00 B4 FF 13 00 AF 7E 92 00 ?..?..?..?..?..?..?..?

0013FFB0 00 B1 00 B1 02 02 B8 EB 86 02 00 00 21 C7 DA CA ??..?..?..?..?..?..?..?

0013FFC0 2C C4 DA CA D7 6F 81 7C 38 07 93 7C FF FF FF FF 内首。] 8\*措

0013FFD0 00 E0 FD 7F FD 4F 54 80 C8 FF 13 00 88 08 AD 88 愤 毓Te?..?墟

0013FFE0 FF FF FF FF A8 9A 83 7C E0 6F 81 7C 00 00 00 00 億部」

0013FFF0 00 00 00 00 00 00 00 00 00 10 40 00 00 00 00 00 .....按.....



地址	HEX 数据	反汇编	注释	寄存器 (FPU)
00405D25	. 8A46 FF	MOV AL, BYTE PTR DS:[ESI-1]		EAX 00000000
00405D28	. D2E5	SHL CH, CL		ECX 00407946 test2_vm.00407946
00405D2A	. 00D8	ADD AL, BL		EDX 00000286
00405D2C	. 66:0FACD9 02	SHRD CX, BX, 2		EBX 6E488F11
00405D31	. C0C1 07	ROL CL, 7		ESP 0013FED0
00405D34	. D2E1	SHL CL, CL		EBP 0013FFBC
00405D36	. 04 91	ADD AL, 91		ESI 00407B9D test2_vm.00407B9D
00405D38	. ^ E9 0DECFFFF	JMP test2_vm.0040494A		EDI 0013FED0
00405D3D	\$ 895424 48	MOV DWORD PTR SS:[ESP+48], EDX		EIP 00405D25 test2_vm.00405D25
00405D41	. ^ E9 210B0000	JMP test2_vm.00406867		C 0 ES 0023 32位 0 (FFFFFFFF)
00405D46	> F9	STC		P 0 CS 001B 32位 0 (FFFFFFFF)
00405D47	. 66:D3E0	SHL AX, CL		A 0 SS 0023 32位 0 (FFFFFFFF)
00405D4A	. 9C	PUSHFD		Z 0 DS 0023 32位 0 (FFFFFFFF)
00405D4E	. 66:8945 04	MOV WORD PTR SS:[EBP+4], AX		S 0 FS 003B 32位 7FFDD000 (FFF)
00405D4F	. ^ E9 52FEFFFF	JMP test2_vm.00405BA6		T 0 GS 0000 NULL
00405D54	\$ 89F6	MOV ESI, ESI		D 0
00405D56	. 9C	PUSHFD		O 0 LastErr ERROR_SUCCESS (00000000)
00405D57	. 8F4424 30	POP DWORD PTR SS:[ESP+30]		EFL 00000202 (NO, NB, NE, A, NS, PO, GE, G)
00405D5B	. F8	CLC		STD empty -UNORM BCE0 01050104 0079006C
DS:[00407B9C]=C8				ST1 empty 3.7898761729206960210e-4933
AL=00				ST2 empty 0.0
地址	HEX 数据	ASCII		
0013FFD0	86 02 00 00 83 02 00 00 94 EB 92 7C B8 EB 00 00	?..?. 旗挂...		0013FFBC CADAC721
0013FFED	80 FF 13 00 07 0A 00 00 FF FF FF FF 00 B1 00 00	?..?..?..?		0013FFC0 CADAC42C
0013FFED	00 00 00 00 00 E0 FD 7F 00 E0 FD 7F F0 FF 13 00	?..?..?..?		0013FFC4 7C816FD7 返回到 kernel32.7C816FD7
0013FF00	56 02 00 00 83 95 3D 20 38 07 93 7C 00 00 00 00	V...e? 8*措...		0013FFC8 7C930738 ntdll.7C930738
0013FF10	88 08 AD 88 18 00 00 00 00 4C 96 B4 40 64 E2 E2	?墟...L附@测		0013FFCC FFFFFFFF
0013FF20	01 00 00 00 00 00 00 00 98 00 00 00 E3 62 61 80	?..?..?..?..?..?..?..?		0013FFD0 7FFDE000
0013FF30	01 00 00 00 06 00 00 00 80 4C 96 B4 98 63 61 80	?..?..?..?..?..?..?..?		0013FFD4 80544FFD
0013FF40	B0 63 B2 E2 00 00 00 00 00 00 00 00 08 B0 D9 E3	测测...?..?..?..?..?..?..?..?		0013FFD8 0013FFC8
0013FF50	00 00 00 00 79 B3 D3 34 12 54 40 00 B1 2C A4 64	...y秤4ITe? 白		0013FFDC 88AD0888
0013FF60	64 FF 13 00 82 02 00 00 00 07 93 7C 28 7C 40 00	d ..?..?..?..?..?..?..?..?		0013FFE0 FFFFFFFF SEM 键尾部
0013FF70	F0 FF 13 00 88 FF 13 00 00 E0 FD 7F 94 EB 92 7C	?..?..?..?..?..?..?..?		0013FFE4 7C839AA8 SE处理程序
0013FF80	B0 FF 13 00 00 00 00 00 82 02 00 00 82 02 00 00	?..?..?..?..?..?..?..?		0013FFE8 7C816FE0 kernel32.7C816FE0
0013FF90	06 02 00 00 63 95 3D 20 83 02 00 00 F0 FF 13 00	?..?..?..?..?..?..?..?		0013FFEC 00000000
0013FFA0	B0 FF 13 00 16 02 00 00 B4 FF 13 00 AF 7E 92 00	?..?..?..?..?..?..?..?		0013FFFO 00000000
0013FFB0	00 B1 00 B1 02 02 B8 EB 86 02 00 00 21 C7 DA CA	??..?..?..?..?..?..?..?		0013FFFF 00000000
0013FFC0	2C C4 DA CA D7 6F 81 7C 38 07 93 7C FF FF FF FF	内首。] 8*措		0013FFFC 00401000 test2_vm.<模块入口点>
0013FFD0	00 E0 FD 7F FD 4F 54 80 C8 FF 13 00 88 08 AD 88	愤 毓Te?..?墟		0013FFFC 00000000
0013FFE0	FF FF FF FF A8 9A 83 7C E0 6F 81 7C 00 00 00 00	億部」		
0013FFF0	00 00 00 00 00 00 00 00 00 10 40 00 00 00 00 00	.....按.....		

命令 :

断点位于 test2\_vm.00405D25

暂停

这里可以看到两个 key 都已经在栈顶了。  
接着开始检测标志位

Vmp 的检测算法是这样的,首先将需要检测的标志位进行搜索,用一个 bit 来表示

```
VMP_PUSH_WORD_CONST    0xFFAF
VMP_PUSH_DWORD_CONST    0x890F95AC
VMP_PUSH_DWORD_CONST    0x7730BAA9
ADD_DWORD 0x7730BAA9 , 0x890F95AC    (0x405055)
VMP_POPC_IMM32 REG_0
VMP_PUSHC_IMM32 REG_20
VMP_MOVIN_WORD_ADDR_DS    [0x405055]    (0x55)
ADD_WORD    0x55 , 0xFFAF    (0x4)    #这个常量是位移的位数,这里也可以猜出是 AF
```

```
VMP_POPC_IMM32 REG_1C
VMP_PUSHC_IMM32 REG_4    #这个就是存放 eflags 的通用寄存器
VMP_PUSHEM_ESP    0x13FFAE
VMP_MOVIN_DWORD_ADDR_SS    [0x13FFAE]    (0x252)
NAND_DWORD    0x252 , 0x252    (0xFFFFFDAD)    #首先执行一个 not
```

再来是解密检测的标志位

```
VMP_PUSH_DWORD_CONST    0x2DC3EA42
VMP_PUSH_DWORD_CONST    0xFFBFAFA8
VMP_PUSH_DWORD_CONST    0xC73EAE08
NAND_DWORD    0xC73EAE08 , 0xFFBFAFA8    (0x405057)
VMP_POPC_IMM32 REG_1C
VMP_PUSHC_IMM32 REG_20
ADD_DWORD    0x0 , 0x405057    (0x405057)
VMP_MOVIN_DWORD_ADDR_DS    [0x405057]    (0xA8D0C8D)
ADD_DWORD 0xA8D0C8D , 0x2DC3EA42    (0x3850F6CF)
VMP_POPC_IMM32 REG_14
VMP_PUSH_DWORD_CONST    0xC7AF0920
ADD_DWORD 0xC7AF0920 , 0x3850F6CF    (0xFFFFFEEF)
VMP_POPC_IMM32 REG_C
```

好了,标志位解密出来了 not FFFFFFFEF 后为 10 ,很明显这个检测的是 AF.

继续

```
NAND_DWORD 0xFFFFFEEF , 0xFFFFFDAD    (0x10)
VMP_POPC_IMM32 REG_14
SHR_DWORD 0x10 , 0x4    (0x1)
VMP_POPC_IMM32 REG_C
VMP_POPC_IMM32 REG_14    #暂时将结果保存到寄存器
```

```
VMP_PUSHC_IMM32 REG_4      #原 eflags
VMP_PUSHC_IMM32 REG_14
NAND_DWORD 0x1 , 0x252  (0x FFFFDAC)
VMP_POPC_IMM32 REG_0
VMP_PUSHVM_ESP  0x13FFB0
VMP_MOVIN_DWORD_ADDR_SS  [0x13FFB0]  (0x FFFFDAC)
NAND_DWORD 0x FFFFDAC , 0x FFFFDAC  (0x253)
VMP_POPC_IMM32 REG_1C
VMP_POPC_IMM32 REG_1C      #保存结果
```

这段代码相当于 `or eflags , AF`

```
VMP_PUSH_WORD_CONST  0xA046
VMP_PUSH_DWORD_CONST 0xFFBFAFA4
VMP_PUSH_DWORD_CONST 0xB40E2604
NAND_DWORD 0xB40E2604 , 0xFFBFAFA4  (0x40505B)
VMP_POPC_IMM32 REG_4
VMP_PUSHC_IMM32 REG_20
ADD_DWORD 0x0 , 0x40505B  (0x40505B)
VMP_POPC_IMM32 REG_C
VMP_MOVIN_WORD_ADDR_DS  [0x40505B]  (0x5FB7)
ADD_WORD 0x5FB7 , 0xA046  (0xFFD)
VMP_POPC_IMM32 REG_0
VMP_PUSH_WORD_CONST  0x2398
NAND_WORD  0x2398 , 0xFFFD  (0x2)      #解密位移常量
VMP_POPC_IMM32 REG_30  (0x202)
```

```
VMP_PUSHC_IMM32 REG_1C      #取出检测的 eflags
VMP_PUSHVM_ESP  0x13FFAE
VMP_MOVIN_DWORD_ADDR_SS  [0x13FFAE]  (0x253)
NAND_DWORD  0x253 , 0x253  (0xFFFDAC)
VMP_POPC_IMM32 REG_30
```

```
VMP_PUSH_DWORD_CONST 0xF04DBB37
VMP_PUSH_DWORD_CONST 0xACE36BDD
VMP_PUSH_DWORD_CONST 0x535CE480
ADD_DWORD  0x535CE480 , 0xACE36BDD  (0x40505D)
VMP_POPC_IMM32 REG_C
VMP_PUSHC_IMM32 REG_20
ADD_DWORD  0x0 , 0x40505D  (0x40505D)
VMP_POPC_IMM32 REG_4
VMP_MOVIN_DWORD_ADDR_DS  [0x40505D]  (0x4C5833E)
```

```
ADD_DWORD 0x4C5833E,0xF04DBB37 (0xF5133E75)
VMP_POPC_IMM32 REG_0
VMP_PUSH_DWORD_CONST 0xAECC189
ADD_DWORD 0xAECC189,0xF5133E75 (0xFFFFFFFF)
VMP_POPC_IMM32 REG_0
```

这里解出的 FFFFFFFE 代表 CF

```
NAND_DWORD 0xFFFFFFFF,0xFFFFDAC (0x1)
VMP_POPC_IMM32 REG_0
```

```
SHL_DWORD 0x1,0x2 (0x4)
VMP_POPC_IMM32 REG_30
VMP_POPC_IMM32 REG_30
```

好,到这里检测标志位已经完了其实上面的指令相当于

**bool(CF|AF)\*4**

这里检查的是 AF 和 CF,但实际上没有跳转指令检测 AF 的,这时因为 vmp 在内部使用模板来模拟指令,这里模拟的是 AAA 指令

AAA 指令的汇编模板

下面是我用 ASM 模拟 vmp 的代码

```
PUSHFD
TEST DWORD PTR SS:[ESP],10
JNZ edit
MOV DWORD PTR SS:[ESP],EAX
ADD BYTE PTR SS:[ESP],0F6
POP DWORD PTR SS:[ESP-4]
PUSHFD
TEST DWORD PTR SS:[ESP],1
JNZ end
edit:
POP DWORD PTR SS:[ESP-4]
ADD AX,106
AND AX,0FF0F
PUSHFD
OR DWORD PTR SS:[ESP],11
end:
POPFD
```

根据 VMP 的描述,AAA 指令首先检测 eflags 的 AF 位,置位则直接修改,然后检查 AL 是否大于 0A,如果大于也修改,修改是将 AX 加上 106 然后将 AH 清零,再将 eflags 的 PF 和 CF 置位.

紧接着往下看

VMP\_PUSHVM\_ESP 0x13FFBC

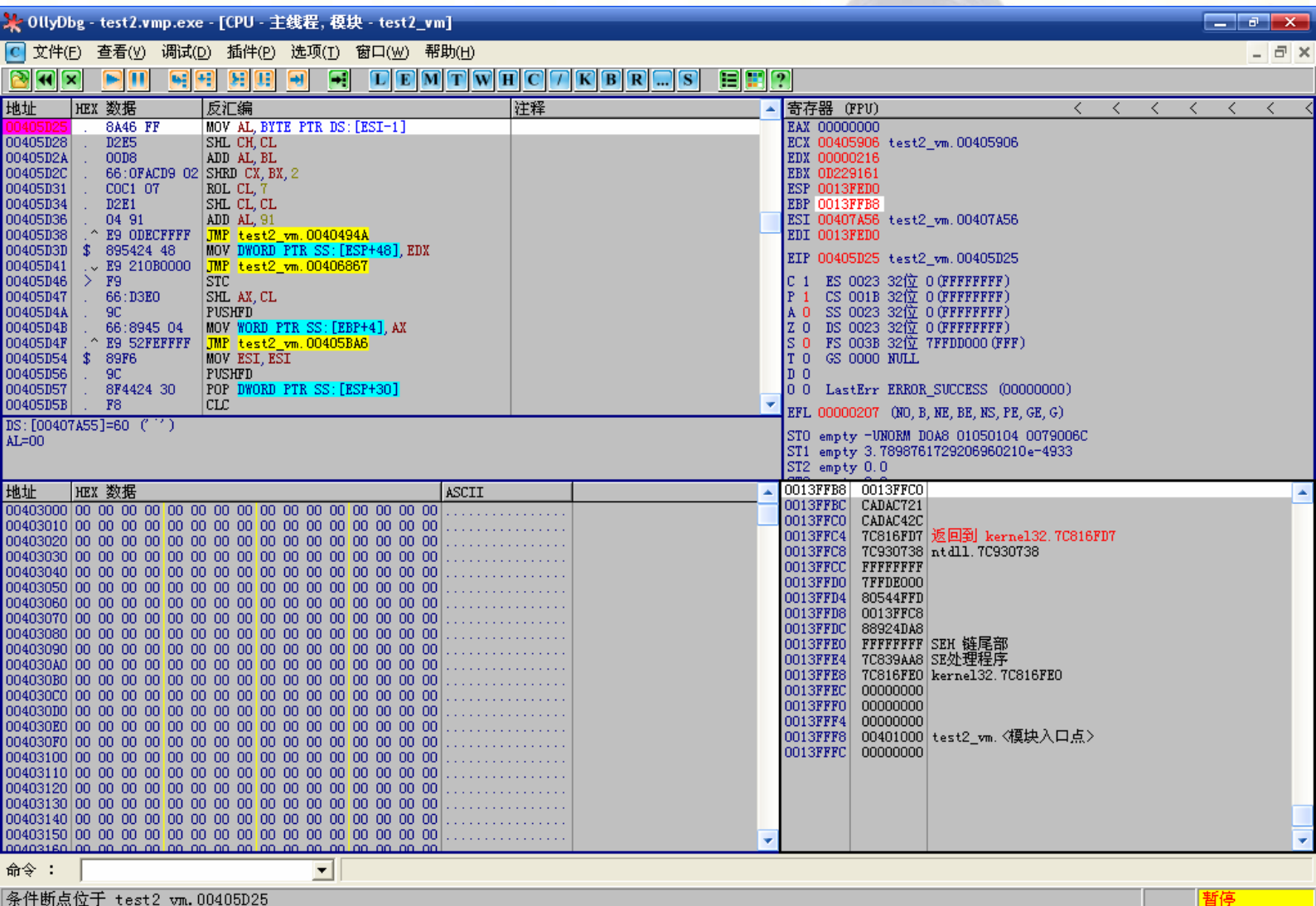
VMP\_PUSHC\_IMM32 REG\_30 #不跳时这里为 0,跳的话这里为 4

ADD\_DWORD 0x0, 0x13FFBC (0x13FFBC)

VMP\_POPC\_IMM32 REG\_1C

VMP\_MOVIN\_DWORD\_ADDR\_SS [0x13FFC0] (0xCADAC42C) #取出选择的 key

VMP\_POPC\_IMM32 REG\_C (0xCADAC42C)



接着

VMP\_PUSHC\_IMM32 REG\_C

VMP\_PUSHC\_IMM32 REG\_C

NAND\_DWORD 0xCADAC42C, 0xCADAC42C (0x35253BD3)

VMP\_POPC\_IMM32 REG\_0

```
VMP_PUSH_DWORD_CONST 0x356542FD
NAND_DWORD 0x356542FD , 0x35253BD3 (0xCA9A8400)
VMP_POPC_IMM32 REG_0
VMP_PUSH_DWORD_CONST 0xCA9ABD02
VMP_PUSHC_IMM32 REG_C (0xCADAC42C)
NAND_DWORD 0xCADAC42C , 0xCA9ABD02 (0x352502D1)
VMP_POPC_IMM32 REG_0
NAND_DWORD 0x352502D1 , 0xCA9A8400 (0x40792E)
VMP_POPC_IMM32 REG_1C
VMP_POPC_IMM32 REG_0
```

其中 0x40792E 就是通过选择的 key 解密出来的地址,这个地址就是新的起点的基址  
最后执行跳转

```
VMP_PUSHC_IMM32 REG_20
VMP_PUSHC_IMM32 REG_0
VMP_JMP 0x407A24 -----> 0x40792E (0x40792E+ 0x0)
```

跳转时要把控制寄存器作为一个偏移。

还记得在虚拟机入口压入的常量吗,那个常量也就是模拟控制寄存器的值,从而做成一个统一的接口。

因为控制寄存器在跳转后还要进行调整,vmp\_jump 是 vmp 指令里比较特殊的一个指令,这个指令会使堆栈失去平衡。

所以必须在入口压入一个常量来模拟控制寄存器,否则的话,当出现跳转到入口的情况时就会死机。

这里猜一下 VMP 的算法,首先当反汇编到 AAA,则用模板将指令进行替换,然后用随机算法加密常量,复杂度的控制应该是这样的  
例如

```
mov eax,1
```

先进行花指令,变成

```
mov eax,1112
```

```
sub eax,1100
```

```
sub eax,11
```

复杂度取决于分裂成多少指令,然后每条指令的常量再分别进行加密

## VMP 校验指令算法

加密时选择检查对象的完整性则会在函数中加入一个 VMP\_CRC 的指令,这个指令将校验虚拟机的内部指令是否被修改。

☒ 检查虚拟机对象的完整性 (降低运行速度)

当 VMP 的检测算法校验出异常时,会破坏进入 VMP 入口时的 dword [esp],而通常这里保存的是调用该函数的返回地址。

VMP 的检测算法很复杂,但是强度却不大。

具体的算法是这样的。

首先执行一个 VMP\_RDTSC 指令,取一个随机数

VMP\_RDTSC EAX = 0xFB1A2692 EDX = 0x16FC

VMP\_POPC\_IMM32 REG\_3C

#EDX 实际是不用的,只是用作花指令

VMP\_POPC\_IMM16 REG\_4

#保存 AX

VMP\_PUSHC\_IMM16 REG\_4

VMP\_PUSHC\_IMM16 REG\_4

NAND\_WORD 0x2692, 0x2692 (0xD96D)

#先 not 一下

VMP\_POPC\_IMM32 REG\_28

### 解密常量

VMP\_PUSH\_WORD\_CONST 0xE7D

VMP\_PUSH\_DWORD\_CONST 0xFFBFB3C

VMP\_PUSH\_DWORD\_CONST 0x430DB810

AND\_DWORD 0x430DB810, 0xFFBFB3C (0x4042C3)

VMP\_POPC\_IMM32 REG\_28

VMP\_PUSHC\_IMM32 REG\_30

ADD\_DWORD 0x0, 0x4042C3 (0x4042C3)

VMP\_POPC\_IMM32 REG\_28

VMP\_MOVIN\_WORD\_ADDR\_DS [0x4042C3] (0x89D0)

ADD\_WORD 0x89D0, 0xE7D (0x984D)

#这里的常量好像也是随机数

VMP\_POPC\_IMM32 REG\_28

### AX and const

NAND\_WORD 0x984D, 0xD96D (0x2692)

#第一个 key

VMP\_POPC\_IMM32 REG\_28

VMP\_PUSHC\_IMM16 REG\_4

VMP\_PUSH\_WORD\_CONST 0x7EF1

VMP\_PUSH\_DWORD\_CONST 0x50E3084E

VMP\_PUSH\_DWORD\_CONST 0xAF5D3A77

ADD\_DWORD 0xAF5D3A77, 0x50E3084E (0x4042C5)

VMP\_POPC\_IMM32 REG\_C

VMP\_PUSHC\_IMM32 REG\_30

ADD\_DWORD 0x0, 0x4042C5 (0x4042C5)

VMP\_POPC\_IMM32 REG\_C

VMP\_MOVIN\_WORD\_ADDR\_DS [0x4042C5] (0xE8C1)

ADD\_WORD 0xE8C1, 0x7EF1 (0x67B2)

VMP\_POPC\_IMM32 REG\_28

NAND\_WORD 0x67B2, 0x2692 (0x984D)

#第二个 key

VMP\_POPC\_IMM32 REG\_0

NAND\_WORD 0x984D, 0x2692 (0x4120)

## VMP\_POPC\_IMM32 REG\_3C

vmp 通过 RDTSC 的 AX 生成 key,其实这个 key 的值并不重要,只是一个随机数而已

## VMP\_POPC\_IMM32 REG\_0

这个 key 加上 EAX 的高 16 位然后保存到寄存器.

接着开始解密校验参数



```
VMP_PUSH_WORD_CONST    0x16CB
VMP_PUSH_DWORD_CONST    0xFFBFBD3E
VMP_PUSH_DWORD_CONST    0x3A862D3C
NAND_DWORD 0x3A862D3C , 0xFFBFBD3E    (0x4042C1)
VMP_POPC_IMM32 REG_3C
VMP_PUSHC_IMM32 REG_30
ADD_DWORD 0x0 , 0x4042C1    (0x4042C1)
VMP_POPC_IMM32 REG_0
VMP_MOVIN_WORD_ADDR_DS    [0x4042C1]    (0xE980)
ADD_WORD 0xE980 , 0x16CB    (0x4B)    #记下这个数常量
VMP_POPC_IMM32 REG_0
VMP_PUSHC_IMM16    REG_0    #这个就是刚刚生成的随机数

还是解密常数
VMP_PUSH_WORD_CONST    0xED56
VMP_PUSH_BYTE_CONST    0x4
VMP_PUSH_WORD_CONST    0x4042C70
SHR_DWORD 0x4042C70 , 0x4    (0x4042C7)
VMP_POPC_IMM32 REG_1C
VMP_PUSHC_IMM32 REG_30
ADD_DWORD 0x0 , 0x4042C7    (0x4042C7)
VMP_POPC_IMM32 REG_C
VMP_MOVIN_WORD_ADDR_DS    [0x4042C7]    (0x12AA)
ADD_WORD 0x12AA , 0xED56    (0x0)
VMP_POPC_IMM32 REG_C

DIV_WORD    0x4120 , 0x4B    (A=0x16    D=0xDE)    #重要!!!
```

上面这段代码是什么意思呢,就是将刚刚生成的随机数取除以 0x4B 的余数,这个 0x4B 是 VMP 用作随机校验数据的组数,上面这么长的代码就是为了生成这个随机数,如果 hook 住这个随机数.那么 vmp 的随机校验将失效.

这个 0x48 也是一个随机数,每次 vmp 都会生成数量不等的校验块.

高位是没有用的.直接写 0.

```
VMP_PUSH_WORD_CONST    0x9
VMP_PUSH_DWORD_CONST    0x5CF3953
VMP_PUSH_DWORD_CONST    0xFA710976
```

```
ADD_DWORD 0xFA710976 , 0x5CF3953 (0x4042C9)
VMP_POPC_IMM32 REG_C
VMP_PUSHC_IMM32 REG_30
ADD_DWORD 0x0 , 0x4042C9 (0x4042C9)
VMP_POPC_IMM32 REG_C
VMP_MOVIN_WORD_ADDR_DS [0x4042C9] (0x0)
ADD_WORD 0x0 , 0x9 (0x9)
VMP_POPC_IMM32 REG_28
```

这里的 9 是每个校验数据的大小

```
MUL_WORD 0x16 , 0x9 (0xC6)
VMP_POPC_IMM32 REG_1C
VMP_POPC_IMM16 REG_0
VMP_POPC_IMM16 REG_1C
```

#保存计算出来的偏移

#这里保存的是除法的商,这个商是没有用的,只是用来做花指令

接着往下看

```
VMP_PUSHC_IMM16 REG_0
VMP_PUSH_BYTE_CONST 0x77
VMP_PUSH_DWORD_CONST 0x74198EA1
VMP_PUSH_DWORD_CONST 0x8C26B69E
ADD_DWORD 0x8C26B69E , 0x74198EA1 (0x40453F)
VMP_POPC_IMM32 REG_3C
VMP_PUSHC_IMM32 REG_30
ADD_DWORD 0x0 , 0x40453F (0x40453F)
VMP_POPC_IMM32 REG_C
VMP_MOVIN_BYTE_ADDR_DS [0x40453F] (0x89)
ADD_BYTE 0x89 , 0x77 (0x0)
VMP_POPC_IMM32 REG_3C
```

这段代码是 `vmp` 的 `movzx` 指令,首先压入一个 `word`,然后再压入一个 `word` 的 0.

可以理解成 `movzx eax,ax`,为什么要这样做呢,因为接下来执行的是

`add dword,dword`,因此要转换成相同的类型.

```
VMP_PUSH_DWORD_CONST 0x39A46AB4
VMP_PUSH_DWORD_CONST 0x4B7B4335
VMP_PUSH_DWORD_CONST 0xB4C5020B
ADD_DWORD 0xB4C5020B , 0x4B7B4335 (0x404540)
VMP_POPC_IMM32 REG_C
VMP_PUSHC_IMM32 REG_30
ADD_DWORD 0x0 , 0x404540 (0x404540)
VMP_POPC_IMM32 REG_C
VMP_MOVIN_DWORD_ADDR_DS [0x404540] (0xC69C0045)
ADD_DWORD 0xC69C0045 , 0x39A46AB4 (0x406AF9)
VMP_POPC_IMM32 REG_C
```

```
VMP_PUSHC_IMM32 REG_30
ADD_DWORD 0x0 , 0x406AF9 (0x406AF9)
VMP_POPC_IMM32 REG_28
```

0x406AF9 是校验数据的基址

```
ADD_DWORD 0x406AF9 , 0xC6 (0x406BBF)
VMP_POPC_IMM32 REG_3C
VMP_POPC_IMM32 REG_3C
```

0x406BBF 是当前校验数据结构的地址

开始进行校验了

```
VMP_PUSHC_IMM32 REG_3C
VMP_PUSH_DWORD_CONST 0x99E1F7E0
VMP_PUSH_BYTE_CONST 0x3
VMP_PUSH_DWORD_CONST 0x2022A28
SHR_DWORD 0x2022A28 , 0x3 (0x404545)
VMP_POPC_IMM32 REG_0
VMP_PUSHC_IMM32 REG_30
ADD_DWORD 0x0 , 0x404545 (0x404545)
VMP_POPC_IMM32 REG_28
VMP_MOVIN_DWORD_ADDR_DS [0x404545] (0x661E0824)
ADD_DWORD 0x661E0824 , 0x99E1F7E0 (0x4)
VMP_POPC_IMM32 REG_C
ADD_DWORD 0x4 , 0x406BBF (0x406BC3)
VMP_POPC_IMM32 REG_1C
VMP_MOVIN_BYTE_ADDR_DS [0x406BC3] (0xB)
```

#校验指令的大小参数,这里的大小是 byte,也就是说 vmp  
#一次最多校验 255 个字节

```
VMP_PUSH_BYTE_CONST 0xBC
VMP_PUSH_DWORD_CONST 0xFFBFBABB
VMP_EXTSPLIT_DWORD 0xAC972A99
NAND_DWORD 0xAC972A99 , 0xFFBFBABB (0x404544)
VMP_POPC_IMM32 REG_1C
VMP_PUSHC_IMM32 REG_30
ADD_DWORD 0x0 , 0x404544 (0x404544)
VMP_POPC_IMM32 REG_0
VMP_MOVIN_BYTE_ADDR_DS [0x404544] (0x44)
ADD_BYTE 0x44 , 0xBC (0x0)
VMP_POPC_IMM32 REG_C
```

每次转换类型时,vmp 都会用 movzx



最后是重点防御的对象,解密校验的地址

```
VMP_PUSHC_IMM32 REG_3C      (0x406BBF)
VMP_MOVIN_DWORD_ADDR_DS     [0x406BBF]   (0x555B985F)
VMP_PUSH_DWORD_CONST        0xEFDB9B76
VMP_PUSH_DWORD_CONST        0xBE65F8FB
VMP_PUSH_DWORD_CONST        0x41DA4C4E
ADD_DWORD 0x41DA4C4E, 0xBE65F8FB  (0x404549)
VMP_POPC_IMM32 REG_C
VMP_PUSHC_IMM32 REG_30
ADD_DWORD 0x0, 0x404549  (0x404549)
VMP_POPC_IMM32 REG_28
VMP_MOVIN_DWORD_ADDR_DS     [0x404549]   (0x10246489)
ADD_DWORD 0x10246489, 0xEFDB9B76  (0xFFFFFFFF)
VMP_POPC_IMM32 REG_C
ADD_DWORD 0xFFFFFFFF, 0x555B985F  (0x555B985E)
VMP_POPC_IMM32 REG_28
VMP_PUSVM_ESP
VMP_MOVIN_DWORD_ADDR_SS     [0x13FFBC]   (0x555B985E)
NAND_DWORD 0x555B985E, 0x555B985E  (0xAAA467A1)
VMP_POPC_IMM32 REG_1C
VMP_PUSH_DWORD_CONST        0xCFDB9B74
VMP_PUSH_DWORD_CONST        0xFFBFBAB2
VMP_PUSH_DWORD_CONST        0x44AAB230
NAND_DWORD 0x44AAB230, 0xFFBFBAB2  (0x40454D)
VMP_POPC_IMM32 REG_1C
VMP_PUSHC_IMM32 REG_30
ADD_DWORD 0x0, 0x40454D  (0x40454D)
VMP_POPC_IMM32 REG_1C
VMP_MOVIN_DWORD_ADDR_DS     [0x40454D]   (0x3024648D)
ADD_DWORD 0x3024648D, 0xCFDB9B74  (0x1)
VMP_POPC_IMM32 REG_C
ADD_DWORD 0x1, 0xAAA467A1  (0xAAA467A2)
VMP_POPC_IMM32 REG_28
VMP_PUSH_DWORD_CONST        0xFFE83718
VMP_PUSH_DWORD_CONST        0x15DB283C
VMP_PUSH_DWORD_CONST        0xEA651D15
ADD_DWORD 0xEA651D15, 0x15DB283C  (0x404551)
VMP_POPC_IMM32 REG_28
VMP_PUSHC_IMM32 REG_30
ADD_DWORD 0x0, 0x404551  (0x404551)
VMP_POPC_IMM32 REG_0
VMP_MOVIN_DWORD_ADDR_DS     [0x404551]   (0x17C8E9)
ADD_DWORD 0x17C8E9, 0xFFE83718  (0x1)
VMP_POPC_IMM32 REG_1C
ADD_DWORD 0x1, 0xAAA467A2  (0xAAA467A3)
```



```
VMP_POPC_IMM32 REG_28
VMP_PUSHVM_ESP
VMP_MOVIN_DWORD_ADDR_SS [0x13FFBC] (0xAAA467A3)
NAND_DWORD 0xAAA467A3 , 0xAAA467A3 (0x555B985C)
VMP_POPC_IMM32 REG_1C
VMP_POPC_IMM32 REG_28
```

555...终于复制完了第一阶段,这里可以看出,vmp 的加密时,内部有一个控制随机算法复杂度的机制,这里的算法明显要比上面那些复杂的多..

接着第二阶段登场

```
VMP_PUSHC_IMM32 REG_28
VMP_PUSHC_IMM32 REG_28
NAND_DWORD 0x555B985C , 0x555B985C (0xAAA467A3)
VMP_PUSH_DWORD_CONST 0x78967F4E
VMP_PUSH_DWORD_CONST 0xFFBFB9C5
VMP_PUSH_DWORD_CONST 0xD4AD2801
NAND_DWORD 0xD4AD2801 , 0xFFBFB9C5 (0x40463A)
VMP_POPC_IMM32 REG_C
VMP_PUSHC_IMM32 REG_30
ADD_DWORD 0x0 , 0x40463A (0x40463A)
VMP_POPC_IMM32 REG_0
VMP_MOVIN_DWORD_ADDR_DS [0x40463A] (0x3C885242)
ADD_DWORD 0x3C885242 , 0x78967F4E (0xB51ED190)
VMP_POPC_IMM32 REG_C
NAND_DWORD 0xB51ED190 , 0xAAA467A3 (0x4041084C)
VMP_POPC_IMM32 REG_0
VMP_PUSHC_IMM32 REG_28 (0x555B985C)
VMP_PUSH_DWORD_CONST 0x2E58924B
VMP_PUSH_DWORD_CONST 0xF692864B
VMP_PUSH_DWORD_CONST 0x9ADBFF3
ADD_DWORD 0x9ADBFF3 , 0xF692864B (0x40463E)
VMP_POPC_IMM32 REG_0
VMP_PUSHC_IMM32 REG_30
ADD_DWORD 0x0 , 0x40463E (0x40463E)
VMP_POPC_IMM32 REG_1C
VMP_MOVIN_DWORD_ADDR_DS [0x40463E] (0x1C889C24)
ADD_DWORD 0x1C889C24 , 0x2E58924B (0x4AE12E6F)
VMP_POPC_IMM32 REG_C
NAND_DWORD 0x4AE12E6F , 0x555B985C (0xA0044180)
VMP_POPC_IMM32 REG_4
NAND_DWORD 0xA0044180 , 0x4041084C (0x1FBAB633)
VMP_POPC_IMM32 REG_4
VMP_PUSH_DWORD_CONST 0xDFF7A778
VMP_PUSH_DWORD_CONST 0xFFBFB9BD
```



```
VMP_PUSH_DWORD_CONST 0x1895A13D
NAND_DWORD 0x1895A13D , 0xFFBFB9BD (0x404642)
VMP_POPC_IMM32 REG_0
VMP_PUSHC_IMM32 REG_30
ADD_DWORD 0x0 , 0x404642 (0x404642)
VMP_POPC_IMM32 REG_0 (0x206)
VMP_MOVIN_DWORD_ADDR_DS [0x404642] (0x4DFF24)
ADD_DWORD 0x4DFF24 , 0xDFF7A778 (0xE045A69C)
VMP_POPC_IMM32 REG_4
ADD_DWORD 0xE045A69C , 0x1FBAB633 (0x5CCF)
VMP_POPC_IMM32 REG_4 (0x207)
```

第二阶段结束了,555555

这里解出的是 0x5CCF,这是一个段偏移。

第三阶段(一定要把你转到头晕为止 .....vmp 作者语)

```
VMP_PUSH_DWORD_CONST 0xC6DB5B00
VMP_PUSH_DWORD_CONST 0xFFBF8605
VMP_PUSH_DWORD_CONST 0xC4090604
NAND_DWORD 0xC4090604 , 0xFFBF8605 (0x4079FA)
VMP_POPC_IMM32 REG_C
VMP_PUSHC_IMM32 REG_30
ADD_DWORD 0x0 , 0x4079FA (0x4079FA)
VMP_MOVIN_DWORD_ADDR_DS [0x4079FA] (0x3964A500)
ADD_DWORD 0x3964A500 , 0xC6DB5B00 (0x400000)
VMP_POPC_IMM32 REG_C
```

这个 0x400000 猜都猜得到是什么了

```
ADD_DWORD 0x400000 , 0x5CCF (0x405CCF)
VMP_POPC_IMM32 REG_C
VMP_PUSHC_IMM32 REG_30
ADD_DWORD 0x0 , 0x405CCF (0x405CCF)
VMP_POPC_IMM32 REG_C
VMP_CRC addr=0x405CCF size=0xB crc=0xDC78A9D6
```

终于开始校验了,这里校验的到底是什么东西呢



00405CCF	. 8F4424 44	POP DWORD PTR SS:[ESP+44]	
00405CD3	. 66:0FCE	BSWAP SI	
00405CD6	. C74424 40 00	MOV DWORD PTR SS:[ESP+40],0	
00405CDE	. F8	CLC	
00405CDF	. F9	STC	
00405CE0	. 66:D3C6	ROL SI,CL	
00405CE3	. 8B7424 70	MOV ESI,DWORD PTR SS:[ESP+70]	
00405CE7	. FF7424 04	PUSH DWORD PTR SS:[ESP+4]	
00405CEB	. 9C	PUSHFD	
00405CEC	. F9	STC	
00405CED	. C74424 0C 6C	MOV DWORD PTR SS:[ESP+C],4EC06C6C	
00405CF5	. 0FCE	BSWAP ESI	
00405CF7	. 66:C70424 59	MOV WORD PTR SS:[ESP],6759	
00405CFD	. ^ E9 26E9FFFF	JMP test2_vm.00404628	

这里校验的是 **vm** 的入口,作者这里的意思我猜是为了防爆.

在这里可以使所有进入 **vm** 的入口都指向同一个函数,只要给 **ESI** 写入一个固定的 **key** 就可以了.

回到 **vm**,使用计算出来的 **crc** 值继续解密

VMP\_PUSHC\_IMM32 REG\_3C (0x406BBF) #取数据结构基址

VMP\_PUSH\_DWORD\_CONST 0xFFE7091D

VMP\_PUSH\_BYTE\_CONST 0x5

VMP\_PUSH\_DWORD\_CONST 0x808C8C0

SHR\_DWORD 0x808C8C0,0x5 (0x404646)

VMP\_POPC\_IMM32 REG\_C

VMP\_PUSHC\_IMM32 REG\_30

ADD\_DWORD 0x0,0x404646 (0x404646)

VMP\_POPC\_IMM32 REG\_4

VMP\_MOVIN\_DWORD\_ADDR\_DS [0x404646] (0x18F6E8)

ADD\_DWORD 0x18F6E8,0xFFE7091D (0x5)

VMP\_POPC\_IMM32 REG\_C

ADD\_DWORD 0x5,0x406BBF (0x406BC4)

VMP\_POPC\_IMM32 REG\_C

VMP\_MOVIN\_DWORD\_ADDR\_DS [0x406BC4] (0x2387562A)

#取出 key

ADD\_DWORD 0x2387562A,0xDC78A9D6 (0x0)

#校验错误的话,这里将不为 0

VMP\_POPC\_IMM32 REG\_4

ADD\_DWORD 0x0,0x7C816FD7 (0x7C816FD7)

#这里 0x7C816FD7 是进入 **vm** 时的返回地址

#校验错误的话,退出时将返回错误的地址或者

#破坏堆栈

VMP\_POPC\_IMM32 REG\_1C

好了第三阶段也完了,整个 **vm** 的校验指令都执行完了,这里指令在每个 **vm** 加密的函数里都存在,每个函数都只执行一次.

整理一下算法.

1. 首先用 **RDTS** 生成一个随机数.
2. 然后取校验数据组数的余数.
3. 将余数乘以 9 再加上基址,计算出一个指针.

4. 这个指针指向一个 9 个字节的结构.
- 0-3 : 解密校验地址的随机数.
- 4 : 校验的大小,这个字节好像没有加密
- 5-8 : 和 crc 匹配的校验值
5. 调用原子指令 `vmpr_crc` 计算一个校验值,并和设定的数据进行匹配
6. 用匹配的结果来修正返回值.

现在我们来看一下 `vmpr` 的作者都校验什么东西,顺便猜猜它们的用途.

首先通过上面的算法可以清楚地看到这次的随机算法为

`key-=3`

`key^=0x4AE12E6F`

`key+=0xE045A69C`

就只有这三条的指令,我们写个脚本,把所有的地址都提取出来,至于 `crc` 的值则忽略它

提取后的数据如下

0x004044EB

0xE1

地址	HEX 数据	反汇编	注释
004044EB	A2	DB A2	
004044EC	03	DB 03	
004044ED	00	DB 00	
004044EE	00	DB 00	
004044EF	\$ E9 85120000	JMP test2_vm.00405779	
004044F4	> 9C	PUSHFD	
004044F5	. 60	PUSHAD	
004044F6	. FF7424 14	PUSH DWORD PTR SS:[ESP+14]	
004044FA	. 68 DCF2986F	PUSH 6F98F2DC	
004044FF	. 8945 00	MOV DWORD PTR SS:[EBP],EAX	
00404502	. 66:891C24	MOV WORD PTR SS:[ESP],BX	
00404506	. C64424 10 D8	MOV BYTE PTR SS:[ESP+10],0D8	
0040450B	. 9C	PUSHFD	
0040450C	. 8D6424 30	LEA ESP,DWORD PTR SS:[ESP+30]	
00404510	\$ E9 09180000	JMP test2_vm.00405D1E	
00404515	\$ 68 BB88E92B	PUSH 2BE988BB	
0040451A	. 4E	DEC ESI	
0040451B	. 9C	PUSHFD	
0040451C	. 66:C74424 04	MOV WORD PTR SS:[ESP+4],0D11A	
00404523	. 8D6424 50	LEA ESP,DWORD PTR SS:[ESP+50]	
00404527	^ E9 88FCFFFF	JMP test2_vm.004041B4	
0040452C	> 38DC	CMP AH,BL	
0040452E	. E8 7D230000	CALL test2_vm.004068B0	
00404533	> F8	CLC	
00404534	. 38EB	CMP BL,CH	
00404536	. D3E8	SHR EAX,CL	
00404538	. 9C	PUSHFD	

校验指令

0x004048B6

0xB2

地址	HEX 数据	反汇编	注释
004048B6	883C24	MOV BYTE PTR SS:[ESP], BH	
004048B9	E9 A9FDFFFF	JMP test2_vm.00404667	
004048BE	E8 B9D00000	CALL test2_vm.0040567C	
004048C3	E8 0C000000	CALL test2_vm.004048D4	
004048C8	8810	MOV BYTE PTR DS:[EAX], DL	
004048CA	50	PUSH EAX	
004048CB	8D6424 28	LEA ESP, DWORD PTR SS:[ESP+28]	
004048CF	E9 4A140000	JMP test2_vm.00405D1E	
004048D4	89C7	MOV EDI, EAX	
004048D6	FC	CLD	
004048D7	9C	PUSHFD	
004048D8	FC	CLD	
004048D9	880C24	MOV BYTE PTR SS:[ESP], CL	
004048DC	E9 CE0F0000	JMP test2_vm.004058AF	
004048E1	83EC FC	SUB ESP, -4	
004048E4	0F8B B9100000	JPO test2_vm.004059A3	
004048EA	35 20F76431	XOR EAX, 3164F720	
004048EF	60	PUSHAD	
004048F0	48	DEC EAX	
004048F1	F9	STC	
004048F2	E8 02FBFFFF	CALL test2_vm.004043F9	
004048F7	66:890C24	MOV WORD PTR SS:[ESP], CX	
004048FB	4E	DEC ESI	
004048FC	68 76EC254F	PUSH 4F25EC76	
00404901	8D6424 4C	LEA ESP, DWORD PTR SS:[ESP+4C]	
00404905	E9 AAF8FFFF	JMP test2_vm.004041B4	
0040490A	66:0BRRR3	MOVSV ST, R1	

校验指令

0x0040496C

0x13

地址	HEX 数据	反汇编	注释
0040496C	F6C3 94	TEST BL, 94	
0040496F	81F1 44E3E13D	XOR ECX, 3FE1E344	
00404975	F6C4 55	TEST AH, 55	
00404978	68 F4BF4062	PUSH 6240BFF4	
0040497D	81C1 00000000	ADD ECX, 0	
00404983	E8 E1FBFFFF	CALL test2_vm.00404569	
00404988	C60424 53	MOV BYTE PTR SS:[ESP], 53	
0040498C	E8 48FDFFFF	CALL test2_vm.004046D9	
00404991	68 2D49BC59	PUSH 59BC492D	
00404996	FF7424 10	PUSH DWORD PTR SS:[ESP+10]	
0040499A	8F45 00	POP DWORD PTR SS:[EBP]	
0040499D	60	PUSHAD	
0040499E	68 B88A5766	PUSH 66578AB8	
004049A3	9C	PUSHFD	
004049A4	8D6424 3C	LEA ESP, DWORD PTR SS:[ESP+3C]	
004049A8	E9 9D000000	JMP test2_vm.00404A4A	
004049AD	C1CE 12	ROR ESI, 12	
004049B0	66:893C24	MOV WORD PTR SS:[ESP], DI	
004049B4	4E	DEC ESI	
004049B5	880424	MOV BYTE PTR SS:[ESP], AL	
004049B8	C64424 04 51	MOV BYTE PTR SS:[ESP+4], 51	
004049BD	66:C70424 9E	MOV WORD PTR SS:[ESP], 6D9E	
004049C3	66:895C24 08	MOV WORD PTR SS:[ESP+8], BX	
004049C8	8D6424 58	LEA ESP, DWORD PTR SS:[ESP+58]	
004049CC	E9 E3F7FFFF	JMP test2_vm.004041B4	
004049D1	C70424 11D28	MOV DWORD PTR SS:[ESP], 9081D211	
004049D8	FF7424 04	PUSH DWORD PTR SS:[ESP+4]	

校验出口

0x00404A80

0xE6

地址	HEX 数据	反汇编	注释
00404A80	\$ 9C	PUSHFD	
00404A81	~ E9 371C0000	JMP test2_vm.004066BD	
00404A86	\$ 9C	PUSHFD	
00404A87	. FF7424 38	PUSH DWORD PTR SS:[ESP+38]	
00404A8B	. 8F45 00	POP DWORD PTR SS:[EBP]	
00404A8E	. C64424 08 3C	MOV BYTE PTR SS:[ESP+8], 3C	
00404A93	. 884424 04	MOV BYTE PTR SS:[ESP+4], AL	
00404A97	. 55	PUSH EBP	
00404A98	. C70424 EBCD05	MOV DWORD PTR SS:[ESP], E305CDEB	
00404A9F	. 8D6424 40	LEA ESP, DWORD PTR SS:[ESP+40]	
00404AA3	^ E9 A2FFFFFF	JMP test2_vm.00404A4A	
00404AA8	\$ 66:85C7	TEST DI, AX	
00404AAB	. 9C	PUSHFD	
00404AAC	. F8	CLC	
00404AAD	. 83EE 02	SUB ESI, 2	
00404AB0	. FF3424	PUSH DWORD PTR SS:[ESP]	
00404AB3	. 9C	PUSHFD	
00404AB4	. F8	CLC	
00404AB5	. 83ED 04	SUB EBP, 4	
00404AB8	. 8D6424 3C	LEA ESP, DWORD PTR SS:[ESP+3C]	
00404ABC	~ OF8D 3F1B0000	JGE test2_vm.00406601	
00404AC2	. 9C	PUSHFD	
00404AC3	. 8945 00	MOV DWORD PTR SS:[EBP], EAX	
00404AC6	. 60	PUSHAD	
00404AC7	. 8D6424 24	LEA ESP, DWORD PTR SS:[ESP+24]	
00404ACB	^ E9 7AFFFFFF	JMP test2_vm.00404A4A	
00404AD0	~ E9 99FFFFFF	JMP test2_vm.0040476E	

校验指令

0x00404B6A

0x10

地址	HEX 数据	反汇编	注释
00404B4F	. 66:8B00	MOV AX, WORD PTR DS:[EAX]	
00404B52	. 60	PUSHAD	
00404B53	. E8 DC0C0000	CALL test2_vm.00405834	
00404B58	> 9C	PUSHFD	
00404B59	. 8F4424 34	POP DWORD PTR SS:[ESP+34]	
00404B5D	. 66:0FABEE	BTS SI, BP	
00404B61	. 66:F7D6	NOT SI	
00404B64	. FF35 6B614000	PUSH DWORD PTR DS:[40616B]	
00404B6A	. 8F4424 30	POP DWORD PTR SS:[ESP+30]	
00404B6E	. 66:0FBFE3	MOVSI SI, BL	
00404B72	. 9C	PUSHFD	
00404B73	. 66:0FCE	BSWAP SI	
00404B76	. C74424 30 00	MOV DWORD PTR SS:[ESP+30], 0	
00404B7E	. F8	CLC	
00404B7F	. F8	CLC	
00404B80	. 8B7424 60	MOV ESI, DWORD PTR SS:[ESP+60]	
00404B84	. 66:0FA3D7	BT DI, DX	
00404B88	. 9C	PUSHFD	
00404B89	. 66:0FBAE0 02	BT AX, 2	
00404B8E	. 0FCE	BSWAP ESI	
00404B90	. 883C24	MOV BYTE PTR SS:[ESP], BH	
00404B93	. F9	STC	
00404B94	. FF7424 08	PUSH DWORD PTR SS:[ESP+8]	
00404B98	~ E9 D71B0000	JMP test2_vm.00406774	
00404B9D	\$ 84C7	TEST BH, AL	
00404B9F	. 66:F7C4 6D3A	TEST SP, 3A6D	
00404BA4	. 83ED 04	SUB EBP, 4	

校验控制变量,防 hook

0x00405064

0x16

地址	HEX 数据	反汇编	注释
00405064	. FEC1	INC CL	
00405066	. 29C0	SUB EAX, EAX	
00405068	> D2E9	SHR CL, CL	
0040506A	. ^ E9 52F2FFFF	JMP test2_vm.004042C1	
0040506F	\$ 66:0FBAF6 02	BTR SI, 2	
00405074	. D3D6	RCL ESI, CL	
00405076	. C74424 30 00	MOV DWORD PTR SS:[ESP+30], 0	
0040507E	. 66:0FBAEE 04	BTS SI, 4	
00405083	. F8	CLC	
00405084	. 8B7424 60	MOV ESI, DWORD PTR SS:[ESP+60]	
00405088	. E8 02030000	CALL test2_vm.0040538F	
0040508D	> 8945 00	MOV DWORD PTR SS:[EBP], EAX	
00405090	. C64424 04 BC	MOV BYTE PTR SS:[ESP+4], 0BC	
00405095	. 9C	PUSHFD	
00405096	. 9C	PUSHFD	
00405097	. 8D6424 48	LEA ESP, DWORD PTR SS:[ESP+48]	
0040509B	. ^ E9 AAF9FFFF	JMP test2_vm.00404A4A	
004050A0	\$ 9C	PUSHFD	
004050A1	. 66:8945 04	MOV WORD PTR SS:[EBP+4], AX	
004050A5	. FF7424 08	PUSH DWORD PTR SS:[ESP+8]	
004050A9	. 9C	PUSHFD	
004050AA	. 9C	PUSHFD	
004050AB	. 9C	PUSHFD	
004050AC	. 8F4424 1C	POP DWORD PTR SS:[ESP+1C]	
004050B0	. ^ E9 96F5FFFF	JMP test2_vm.0040464B	
004050B5	. 08E2	OR DL, AH	
004050B7	. 0FA3C2	BT EDI, EAX	

校验 vmp\_crc,防止写入固定的 crc

0x0040507E

0xE0

地址	HEX 数据	反汇编	注释
0040507E	. 66:0FBAEE 04	BTS SI, 4	
00405083	. F8	CLC	
00405084	. 8B7424 60	MOV ESI, DWORD PTR SS:[ESP+60]	
00405088	. E8 02030000	CALL test2_vm.0040538F	
0040508D	> 8945 00	MOV DWORD PTR SS:[EBP], EAX	
00405090	. C64424 04 BC	MOV BYTE PTR SS:[ESP+4], 0BC	
00405095	. 9C	PUSHFD	
00405096	. 9C	PUSHFD	
00405097	. 8D6424 48	LEA ESP, DWORD PTR SS:[ESP+48]	
0040509B	. ^ E9 AAF9FFFF	JMP test2_vm.00404A4A	
004050A0	\$ 9C	PUSHFD	
004050A1	. 66:8945 04	MOV WORD PTR SS:[EBP+4], AX	
004050A5	. FF7424 08	PUSH DWORD PTR SS:[ESP+8]	
004050A9	. 9C	PUSHFD	
004050AA	. 9C	PUSHFD	
004050AB	. 9C	PUSHFD	
004050AC	. 8F4424 1C	POP DWORD PTR SS:[ESP+1C]	
004050B0	. ^ E9 96F5FFFF	JMP test2_vm.0040464B	
004050B5	. 08E2	OR DL, AH	
004050B7	. 0FA3C2	BT EDI, EAX	
004050BA	. 8B55 00	MOV EDI, DWORD PTR SS:[EBP]	
004050BD	. C0F8 03	SAR AL, 3	
004050C0	. D2F8	SAR AL, CL	
004050C2	. 85D5	TEST EBP, EDI	
004050C4	. D0D0	RCL AL, 1	
004050C6	. 83C5 02	ADD EBP, 2	
004050C9	. 57	PUSH EDI	

接上面,还是校验 vmp\_crc,防 hook

0x00405162

0x0D

地址	HEX 数据	反汇编	注释
0040514F	. 8F4424 2C	POP DWORD PTR SS:[ESP+2C]	
00405153	. 66:0FBAAE 0D	BTS SI,0D	
00405158	. 66:C1D6 04	RCL SI,4	
0040515C	. FF35 6B614000	PUSH DWORD PTR DS:[40616B]	
00405162	. 8F4424 28	POP DWORD PTR SS:[ESP+28]	
00405166	. 66:0FBAAE 0A	BT BP,0A	
0040516B	. C74424 24 00	MOV DWORD PTR SS:[ESP+24],0	
00405173	. F5	CMC	
00405174	. C1C6 09	ROL ESI,9	
00405177	. F9	STC	
00405178	. 8D3465 B74F5	LEA ESI,DWORD PTR DS:[A6514FB7]	
0040517F	. 8B7424 54	MOV ESI,DWORD PTR SS:[ESP+54]	
00405183	. E9 74140000	JMP test2_vm.004065FC	
00405188	> 38DB	CMP BL,BL	
0040518A	. 66:3D FA2E	CMP AX,2EFA	
0040518E	. 83C5 06	ADD EBP,6	
00405191	. 60	PUSHAD	
00405192	. 9C	PUSHFD	
00405193	. 36:8810	MOV BYTE PTR SS:[EAX],DL	
00405196	. C60424 1D	MOV BYTE PTR SS:[ESP],1D	
0040519A	. 9C	PUSHFD	
0040519B	. FF3424	PUSH DWORD PTR SS:[ESP]	
0040519E	. 68 E8F98102	PUSH 281F9E8	
004051A3	. 8D6424 34	LEA ESP,DWORD PTR SS:[ESP+34]	
004051A7	. E9 720B0000	JMP test2_vm.00405D1E	
004051AC	> 57	PUSH EDI	
004051AD	. C60424 CF	MOV BYTE PTR SS:[ESP],0CF	

全局控制变量的多回路分支

0x00405367

0x52

地址	HEX 数据	反汇编	注释
00405360	. C60424 C1	MOV BYTE PTR SS:[ESP],0C1	
00405364	. 60	PUSHAD	
00405365	. 9C	PUSHFD	
00405368	. 8B5424 30	MOV EDX,DWORD PTR SS:[ESP+30]	
0040536A	. 66:0FBEBF 9	MOVSB DI,CL	
0040536E	. 9C	PUSHFD	
0040536F	. 54	PUSH ESP	
00405370	. 8B7C24 3C	MOV EDI,DWORD PTR SS:[ESP+3C]	
00405374	. 9F	LAHF	
00405375	. 58	POP EAX	
00405376	. 8B5C24 3C	MOV EBX,DWORD PTR SS:[ESP+3C]	
0040537A	. 66:0FB6C2	MOVZX AX,DL	
0040537E	. 98	CWDE	
0040537F	. 8B4424 40	MOV EAX,DWORD PTR SS:[ESP+40]	
00405383	. 9C	PUSHFD	
00405384	. 8B4424 08	MOV BYTE PTR SS:[ESP+8],AL	
00405388	. FF7424 48	PUSH DWORD PTR SS:[ESP+48]	
0040538C	. C2 4C00	RETN 4C	
0040538F	. \$ 9C	PUSHFD	
00405390	. E9 66010000	JMP test2_vm.004054FB	
00405395	> C64424 04 0A	MOV BYTE PTR SS:[ESP+4],0A	
0040539A	. E8 A4080000	CALL test2_vm.00405C43	
0040539F	> 896C24 38	MOV DWORD PTR SS:[ESP+38],EBP	
004053A3	. 8DB5 60FC11C	LEA ESI,DWORD PTR SS:[EBP+C911FC60]	
004053A9	. 5E	POP ESI	
004053AA	. 9C	PUSHFD	
004053AB	. 8F4424 30	POP DWORD PTR SS:[ESP+30]	

还是校验出口

0x004053BD

0x12

地址	HEX 数据	反汇编	注释
004053AA	. 9C	PUSHFD	
004053AB	. 8F4424 30	POP DWORD PTR SS:[ESP+30]	
004053AF	. 66:F7D6	NOT SI	
004053B2	. F7D6	NOT ESI	
004053B4	. 66:D3DE	RCR SI,CL	
004053B7	. FF35 6B614000	PUSH DWORD PTR DS:[40616B]	
004053BD	. 8F4424 2C	POP DWORD PTR SS:[ESP+2C]	
004053C1	. 66:F7D6	NOT SI	
004053C4	. 66:F7D6	NOT SI	
004053C7	. 66:0FBF1	MOVSI SI,CL	
004053CB	. C74424 28 00	MOV DWORD PTR SS:[ESP+28],0	
004053D3	. 5E	POP ESI	
004053D4	. 60	PUSHAD	
004053D5	. 0FABDE	BTS ESI,EBX	
004053D8	. 8B7424 74	MOV ESI,DWORD PTR SS:[ESP+74]	
004053DC	. F5	CMC	
004053DD	. 9C	PUSHFD	
004053DE	. F8	CLC	
004053DF	. F9	STC	
004053E0	. 0FCE	BSWAP ESI	
004053E2	. E8 FF110000	CALL test2_vm.004065E6	
004053E7	\$ F7D6	NOT ESI	
004053E9	. 895C24 50	MOV DWORD PTR SS:[ESP+50],EBX	
004053ED	. 8D34E5 57E921	LEA ESI,DWORD PTR DS:[422DE957]	
004053F4	. E8 3F100000	CALL test2_vm.00406438	
004053F9	> C1DE 09	RCR ESI,9	
004053FC	. 66:D1D6	RCL SI,1	

全局控制变量的多回路分支

0x004053D3

0x30

地址	HEX 数据	反汇编	注释
004053D3	. 5E	POP ESI	
004053D4	. 60	PUSHAD	
004053D5	. 0FABDE	BTS ESI,EBX	
004053D8	. 8B7424 74	MOV ESI,DWORD PTR SS:[ESP+74]	
004053DC	. F5	CMC	
004053DD	. 9C	PUSHFD	
004053DE	. F8	CLC	
004053DF	. F9	STC	
004053E0	. 0FCE	BSWAP ESI	
004053E2	. E8 FF110000	CALL test2_vm.004065E6	
004053E7	\$ F7D6	NOT ESI	
004053E9	. 895C24 50	MOV DWORD PTR SS:[ESP+50],EBX	
004053ED	. 8D34E5 57E921	LEA ESI,DWORD PTR DS:[422DE957]	
004053F4	. E8 3F100000	CALL test2_vm.00406438	
004053F9	> C1DE 09	RCR ESI,9	
004053FC	. 66:D1D6	RCL SI,1	
004053FF	. C74424 38 00	MOV DWORD PTR SS:[ESP+38],0	
00405407	. 5E	POP ESI	
00405408	. F8	CLC	
00405409	. 8B7424 64	MOV ESI,DWORD PTR SS:[ESP+64]	
0040540D	. E8 AB0E0000	CALL test2_vm.004062BD	
00405412	\$ 66:8945 00	MOV WORD PTR SS:[EBP],AX	
00405416	. FF7424 04	PUSH DWORD PTR SS:[ESP+4]	
0040541A	. 8B1424	MOV BYTE PTR SS:[ESP],DL	
0040541D	. 51	PUSH ECX	
0040541E	. 66:893424	MOV WORD PTR SS:[ESP],SI	
00405422	. 8B6424 44	LEA ESP,DWORD PTR SS:[ESP+44]	

接上面,还是校验入口

0x00405407

0xAC

地址	HEX 数据	反汇编	注释
004053F9	> C1DE 09	RCR ESI, 9	
004053FC	. 66:D1D6	RCL SI, 1	
004053FF	. C74424 38 00	MOV DWORD PTR SS:[ESP+38], 0	
00405407	. 5E	POP ESI	
00405408	. F8	CLC	
00405409	. 8B7424 64	MOV ESI, DWORD PTR SS:[ESP+64]	
0040540D	. E8 A80E0000	CALL test2_vm.004062BD	
00405412	\$ 66:8945 00	MOV WORD PTR SS:[EBP], AX	
00405416	. FF7424 04	PUSH DWORD PTR SS:[ESP+4]	
0040541A	. 8B1424	MOV BYTE PTR SS:[ESP], DL	
0040541D	. 51	PUSH ECX	
0040541E	. 66:893424	MOV WORD PTR SS:[ESP], SI	
00405422	. 8D6424 44	LEA ESP, DWORD PTR SS:[ESP+44]	
00405426	. ^ E9 1FF6FFFF	JMP test2_vm.00404A4A	
0040542B	. 83ED 02	SUB EBP, 2	
0040542E	. 38CC	CMP AH, CL	
00405430	. 85E3	TEST EBX, ESP	
00405432	. F8	CLC	
00405433	. 66:D3E8	SHR AX, CL	
00405436	. 60	PUSHAD	
00405437	. ^ E9 CEFCFFFF	JMP test2_vm.0040410A	
0040543C	> 68 6B8B3BE1	PUSH E13B8B3BE1	
00405441	. 60	PUSHAD	
00405442	. ^ E9 A8150000	JMP test2_vm.004069EF	
00405447	\$ D2DA	RCR DL, CL	
00405449	. F8	CLC	
0040544A	. 66:8B55 04	MOV DY, WORD PTR SS:[EBP+4]	

继续接上面,校验入口

0x004054B7

0x0B

地址	HEX 数据	反汇编	注释
004054A7	. 9C	PUSHFD	
004054A8	. 8F4424 1C	POP DWORD PTR SS:[ESP+1C]	
004054AC	. 66:0FB4FE 06	BTC SI, 6	
004054B1	. FF35 6B614000	PUSH DWORD PTR DS:[40616B]	
004054B7	. 8F4424 18	POP DWORD PTR SS:[ESP+18]	
004054BB	. 66:0FCE	BSWAP SI	
004054BE	. C74424 14 00	MOV DWORD PTR SS:[ESP+14], 0	
004054C6	. 68 4D7D8B3B	PUSH 3B8B7D4D	
004054CB	. 8B7424 48	MOV ESI, DWORD PTR SS:[ESP+48]	
004054CF	. ^ E9 3F040000	JMP test2_vm.00405913	
004054D4	\$ 9C	PUSHFD	
004054D5	. 53	PUSH EBX	
004054D6	. 897C24 2C	MOV DWORD PTR SS:[ESP+2C], EDI	
004054DA	. 51	PUSH ECX	
004054DB	. 9C	PUSHFD	
004054DC	. 66:C70424 2A	MOV WORD PTR SS:[ESP], 782A	
004054E2	. 895424 30	MOV DWORD PTR SS:[ESP+30], EDX	
004054E6	. 9C	PUSHFD	
004054E7	. 9C	PUSHFD	
004054E8	. 66:C70424 C7	MOV WORD PTR SS:[ESP], 4AC7	
004054EE	. 897424 34	MOV DWORD PTR SS:[ESP+34], ESI	
004054F2	. 55	PUSH EBP	
004054F3	. 66:F7D6	NOT SI	
004054F6	. ^ E9 7DEFFFFF	JMP test2_vm.00404478	
004054FB	> 0FCE	BSWAP ESI	
004054FD	. F8	CLC	
004054FF	. C1CF 12	RCR ESI, 12	

继续全局控制变量

0x004054C6

0x44

地址	HEX 数据	反汇编	注释
004054B7	. 8F4424 18	POP DWORD PTR SS:[ESP+18]	
004054BB	. 66:0FCE	BSWAP SI	
004054BE	. C74424 14 00	MOV DWORD PTR SS:[ESP+14], 0	
004054C6	. 68 4D7D8B3B	PUSH 3B8B7D4D	
004054CB	. 8B7424 48	MOV ESI, DWORD PTR SS:[ESP+48]	
004054CF	~ E9 3F040000	JMP test2_vm.00405913	
004054D4	\$ 9C	PUSHFD	
004054D5	. 53	PUSH EBX	
004054D6	. 897C24 2C	MOV DWORD PTR SS:[ESP+2C], EDI	
004054DA	. 51	PUSH ECX	
004054DB	. 9C	PUSHFD	
004054DC	. 66:C70424 2A	MOV WORD PTR SS:[ESP], 782A	
004054E2	. 895424 30	MOV DWORD PTR SS:[ESP+30], EDX	
004054E6	. 9C	PUSHFD	
004054E7	. 9C	PUSHFD	
004054E8	. 66:C70424 C7	MOV WORD PTR SS:[ESP], 4AC7	
004054EE	. 897424 34	MOV DWORD PTR SS:[ESP+34], ESI	
004054F2	. 55	PUSH EBP	
004054F3	. 66:F7D6	NOT SI	
004054F6	^ E9 7DEFFFFF	JMP test2_vm.00404478	
004054FB	> 0FCE	BSWAP ESI	
004054FD	. F8	CLC	
004054FE	. C1CE 12	ROR ESI, 12	
00405501	. E8 BC040000	CALL test2_vm.004059C2	
00405506	\$ C74424 04 00	MOV DWORD PTR SS:[ESP+4], 0	
0040550E	. F8	CLC	
0040550F	. 68 3424	PUSH DWORD PTR SS:[ESP]	

还是入口.....

0x00405609

0x0A

地址	HEX 数据	反汇编	注释
00405604	. FF7424 44	PUSH DWORD PTR SS:[ESP+44]	
00405608	. C2 4800	RETN 48	
0040560B	\$ 0FBAE5 14	BT EBP, 14	
0040560F	. C74424 48 00	MOV DWORD PTR SS:[ESP+48], 0	
00405617	. 66:0FBF2	MOVSI SI, DL	
0040561B	. 66:89D6	MOV SI, DX	
0040561E	. 8B7424 78	MOV ESI, DWORD PTR SS:[ESP+78]	
00405622	. F8	CLC	
00405623	. E8 D3080000	CALL test2_vm.00405EFB	
00405628	\$ E8 D90F0000	CALL test2_vm.00406606	
0040562D	\$ FF3424	PUSH DWORD PTR SS:[ESP]	
00405630	. 895C24 04	MOV DWORD PTR SS:[ESP+4], EBX	
00405634	. 60	PUSHAD	
00405635	. 60	PUSHAD	
00405636	. 68 29F81114	PUSH 1411F829	
0040563B	. 897C24 44	MOV DWORD PTR SS:[ESP+44], EDI	
0040563F	. 9C	PUSHFD	
00405640	. E8 F8060000	CALL test2_vm.00405D3D	
00405645	\$ 36:8B00	MOV EAX, DWORD PTR SS:[EAX]	
00405648	. 68 ACE3B792	PUSH 92B7E3AC	
0040564D	. 8D6424 10	LEA ESP, DWORD PTR SS:[ESP+10]	
00405651	^ 0F80 9DEEFF	J0 test2_vm.004044F4	
00405657	. 8945 00	MOV DWORD PTR SS:[EBP], EAX	
0040565A	. 60	PUSHAD	
0040565B	. 8D6424 20	LEA ESP, DWORD PTR SS:[ESP+20]	
0040565F	~ E9 BA060000	JMP test2_vm.00405D1E	
00405664	> 68 F98F352	PUSH 52F38F90	

除了这个我不到它校验些什么东西了.....

0x00405A06

0x76

地址	HEX 数据	反汇编	注释
00405A03	\$ 897424 28	MOV DWORD PTR SS:[ESP+28], ESI	
00405A07	. 66:0FB8F0	MOVSI SI, AL	
00405A0B	. E8 71EEFFFF	CALL test2_vm.00404881	
00405A10	> 20D0	AND AL, DL	
00405A12	. E8 89F6FFFF	CALL test2_vm.004050A0	
00405A17	. 66:0FB8C3	MOVSI AX, BL	
00405A1B	. 80CE 79	OR DH, 79	
00405A1E	. D40A	AAM	
00405A20	. 66:8B45 00	MOV AX, WORD PTR SS:[EBP]	
00405A24	. F8	CLC	
00405A25	. 66:0FCA	BSWAP DX	
00405A28	. 66:8B55 02	MOV DX, WORD PTR SS:[EBP+2]	
00405A2C	. 84C1	TEST CL, AL	
00405A2E	. E8 EA030000	CALL test2_vm.00405E1D	
00405A33	\$ 83ED 02	SUB EBP, 2	
00405A36	~ E9 410D0000	JMP test2_vm.0040677C	
00405A3B	> E8 69EDFFFF	CALL test2_vm.004047A9	
00405A40	> E8 AE080000	CALL test2_vm.004062F3	
00405A45	\$ 4E	DEC ESI	
00405A46	. 882424	MOV BYTE PTR SS:[ESP], AH	
00405A49	. 60	PUSHAD	
00405A4A	. 54	PUSH ESP	
00405A4B	. 8D6424 38	LEA ESP, DWORD PTR SS:[ESP+38]	
00405A4F	^ E9 60E7FFFF	JMP test2_vm.004041B4	
00405A54	\$ F8	CLC	
00405A55	. C1CE 12	ROR ESI, 12	
00405A58	. E8 12070000	CALL test2_vm.0040616F	

校验指令

0x00405A80

0x27

地址	HEX 数据	反汇编	注释
00405A80	. 66:0FB8FE 0B	BTC SI, 0B	
00405A85	. 66:D3DE	RCR SI, CL	
00405A88	. 66:0FABC6	BTS SI, AX	
00405A8C	~ E9 79080000	JMP test2_vm.0040630A	
00405A91	. D0D8	RCR AL, 1	
00405A93	. 89E8	MOV EAX, EBP	
00405A95	. F8	CLC	
00405A96	. E8 6AECFFFF	CALL test2_vm.00404705	
00405A9B	. 9C	PUSHFD	
00405A9C	. 890424	MOV DWORD PTR SS:[ESP], EAX	
00405A9F	. E8 E4EEFFFF	CALL test2_vm.00404988	
00405AA4	\$ 9C	PUSHFD	
00405AA5	. FF35 6B614000	PUSH DWORD PTR DS:[40616B]	
00405AAB	. 8F4424 04	POP DWORD PTR SS:[ESP+4]	
00405AAF	. E8 52FAFFFF	CALL test2_vm.00405506	
00405AB4	> 68 DD5BCB62	PUSH 62CB5BDD	
00405AB9	. 894424 04	MOV DWORD PTR SS:[ESP+4], EAX	
00405ABD	. 9C	PUSHFD	
00405ABE	. 60	PUSHAD	
00405ABF	. 68 A84594B7	PUSH B79445A8	
00405AC4	. 895C24 28	MOV DWORD PTR SS:[ESP+28], EBX	
00405AC8	~ E9 1B000000	JMP test2_vm.00405AE8	
00405ACD	\$ 9C	PUSHFD	
00405ACE	. 66:0FCE	BSWAP SI	
00405AD1	. 896C24 28	MOV DWORD PTR SS:[ESP+28], EBP	
00405AD5	. 66:0FB8F3	MOVSI SI, BL	
00405AD8	. 68 100D0000	PUSH 0D0D0000	

校验 pushesp

0x00405AAB

0x5A

地址	HEX 数据	反汇编	注释
00405A9F	E8 E4EEFFFF	CALL test2_vm.00404988	
00405AA4	\$ 9C	PUSHFD	
00405AA5	. FF35 6B614000	PUSH DWORD PTR DS:[40616B]	
00405AAB	. 8F4424 04	POP DWORD PTR SS:[ESP+4]	
00405AAF	. E8 52FAFFFF	CALL test2_vm.00405506	
00405AB4	> 68 DD5BCB62	PUSH 62CB5BDD	
00405AB9	. 894424 04	MOV DWORD PTR SS:[ESP+4], EAX	
00405ABD	. 9C	PUSHFD	
00405ABE	. 60	PUSHAD	
00405ABF	. 68 A84594B7	PUSH B79445A8	
00405AC4	. 895C24 28	MOV DWORD PTR SS:[ESP+28], EBX	
00405AC8	√ E9 1B000000	JMP test2_vm.00405AE8	
00405ACD	\$ 9C	PUSHFD	
00405ACE	. 66:0FCE	BSWAP SI	
00405AD1	. 896C24 28	MOV DWORD PTR SS:[ESP+28], EBP	
00405AD5	. 66:0FB6F3	MOVZX SI, BL	
00405AD9	. 68 199D9FBD	PUSH BD9F9D19	
00405ADE	. 9C	PUSHFD	
00405ADF	. 8F4424 28	POP DWORD PTR SS:[ESP+28]	
00405AE3	√ E9 ED040000	JMP test2_vm.00405FD5	
00405AE8	√ E9 F50F0000	JMP test2_vm.00406AE2	
00405AED	> 0FBAB4 1D	BT ESP, 1D	
00405AF1	. C1CE 12	ROR ESI, 12	
00405AF4	. E8 38EBFFFF	CALL test2_vm.00404631	
00405AF9	\$ 9C	PUSHFD	
00405AFA	. 8F4424 40	POP DWORD PTR SS:[ESP+40]	
00405AFE	. 68 2E3C1FCD	PUSH CD1F3C2E	

接上面,校验全局控制变量

0x00405B09

0x0F

00405AF9	\$ 9C	PUSHFD	
00405AFA	. 8F4424 40	POP DWORD PTR SS:[ESP+40]	
00405AFE	. 68 2E3C1FCD	PUSH CD1F3C2E	
00405B03	. FF35 6B614000	PUSH DWORD PTR DS:[40616B]	
00405B09	. 8F4424 40	POP DWORD PTR SS:[ESP+40]	
00405B0D	. 66:0FBAB6 0C	BTR SI, 0C	
00405B12	. D3DE	RCR SI, CL	
00405B14	. C74424 3C 00	MOV DWORD PTR SS:[ESP+3C], 0	
00405B1C	. F5	CMC	
00405B1D	. 66:D3D6	RCL SI, CL	
00405B20	. F7D6	NOT ESI	
00405B22	. 66:0FBEB2	MOVSX SI, DL	
00405B26	. 8B7424 6C	MOV ESI, DWORD PTR SS:[ESP+6C]	
00405B2A	. F5	CMC	
00405B2B	. E8 B2020000	CALL test2_vm.00405DE2	
00405B30	> 66:D3DE	RCR SI, CL	
00405B33	. FF35 6B614000	PUSH DWORD PTR DS:[40616B]	
00405B39	. 8F4424 1C	POP DWORD PTR SS:[ESP+1C]	
00405B3D	. 89EE	MOV ESI, EBP	
00405B3F	. C74424 18 00	MOV DWORD PTR SS:[ESP+18], 0	
00405B47	. 66:C1C6 05	ROL SI, 5	
00405B4B	. 66:0FABF6	BTS SI, SI	
00405B4F	. 8B7424 48	MOV ESI, DWORD PTR SS:[ESP+48]	
00405B53	. 60	PUSHAD	
00405B54	. 896424 04	MOV DWORD PTR SS:[ESP+4], ESP	
00405B58	. 9C	PUSHFD	
00405B59	. E9	STC	

这个不用说了吧

0x00405B1C

0x19

地址	HEX 数据	反汇编	注释
00405B1C	F5	CMC	
00405B1D	66:D3D6	RCL SI,CL	
00405B20	F7D6	NOT ESI	
00405B22	66:0FBF2	MOVSI SI,DL	
00405B26	8B7424 6C	MOV ESI,DWORD PTR SS:[ESP+6C]	
00405B2A	F5	CMC	
00405B2B	E8 B2020000	CALL test2_vm.00405DE2	
00405B30	66:D3DE	RCR SI,CL	
00405B33	FF35 6B614000	PUSH DWORD PTR DS:[40616B]	
00405B39	8F4424 1C	POP DWORD PTR SS:[ESP+1C]	
00405B3D	89EE	MOV ESI,EBP	
00405B3F	C74424 18 0000	MOV DWORD PTR SS:[ESP+18],0	
00405B47	66:C1C6 05	ROL SI,5	
00405B4B	66:0FABF6	BTS SI,SI	
00405B4F	8B7424 48	MOV ESI,DWORD PTR SS:[ESP+48]	
00405B53	60	PUSHAD	
00405B54	896424 04	MOV DWORD PTR SS:[ESP+4],ESP	
00405B58	9C	PUSHFD	
00405B59	F9	STC	
00405B5A	0FCE	BSWAP ESI	
00405B5C	F9	STC	
00405B5D	C1CE 12	ROR ESI,12	
00405B60	E9 31E7FFFF	JMP test2_vm.00404296	
00405B65	66:C1D0 0A	RCL AX,0A	
00405B69	C0D9 07	RCR CL,7	
00405B6C	60	PUSHAD	
00405B6D	9F	LAHF	

参考上面的算法

0x00405B39

0x0A

地址	HEX 数据	反汇编	注释
00405B2B	E8 B2020000	CALL test2_vm.00405DE2	
00405B30	66:D3DE	RCR SI,CL	
00405B33	FF35 6B614000	PUSH DWORD PTR DS:[40616B]	
00405B39	8F4424 1C	POP DWORD PTR SS:[ESP+1C]	
00405B3D	89EE	MOV ESI,EBP	
00405B3F	C74424 18 0000	MOV DWORD PTR SS:[ESP+18],0	
00405B47	66:C1C6 05	ROL SI,5	
00405B4B	66:0FABF6	BTS SI,SI	
00405B4F	8B7424 48	MOV ESI,DWORD PTR SS:[ESP+48]	
00405B53	60	PUSHAD	
00405B54	896424 04	MOV DWORD PTR SS:[ESP+4],ESP	
00405B58	9C	PUSHFD	
00405B59	F9	STC	
00405B5A	0FCE	BSWAP ESI	
00405B5C	F9	STC	
00405B5D	C1CE 12	ROR ESI,12	
00405B60	E9 31E7FFFF	JMP test2_vm.00404296	
00405B65	66:C1D0 0A	RCL AX,0A	
00405B69	C0D9 07	RCR CL,7	
00405B6C	60	PUSHAD	
00405B6D	9F	LAHF	
00405B6E	8B45 00	MOV EAX,DWORD PTR SS:[EBP]	
00405B71	80C9 49	OR CL,49	
00405B74	8A4D 04	MOV CL,BYTE PTR SS:[EBP+4]	
00405B77	FF3424	PUSH DWORD PTR SS:[ESP]	
00405B7A	68 D39BD700	PUSH 0D79BD3	
00405B7E	F9	STC	

没什么好说的,自己看吧

0x00405C43

0x88

地址	HEX 数据	反汇编	注释
00405C43	\$ F8	CLC	
00405C44	. C0C8 06	ROR AL, 6	
00405C47	. E8 89E5FFFF	CALL test2_vm.004041D5	
00405C4C	> E9 C7E9FFFF	JMP test2_vm.00404618	
00405C51	\$ 81FC E0AFE395	CMP ESP, 95E3AFE0	
00405C57	. F9	STC	
00405C58	. F9	STC	
00405C59	. 86E0	XCHG AL, AH	
00405C5B	. 60	PUSHAD	
00405C5C	. C74424 04 49	MOV DWORD PTR SS:[ESP+4], 66899F49	
00405C64	. F5	CMC	
00405C65	. 66:01D8	ADD AX, BX	
00405C68	. C80424 24	MOV BYTE PTR SS:[ESP], 24	
00405C6C	. 66:FFC0	INC AX	
00405C6F	. F5	CMC	
00405C70	. 66:C1C0 0B	ROL AX, 0B	
00405C74	. F9	STC	
00405C75	. 68 DDE1720D	PUSH 0D72E1DD	
00405C7A	. 66:35 7CF3	XOR AX, 0F37C	
00405C7E	. 66:892C24	MOV WORD PTR SS:[ESP], BP	
00405C82	. 52	PUSH EDX	
00405C83	. 66:01C3	ADD BX, AX	
00405C86	. 66:85F3	TEST BX, SI	
00405C89	. F5	CMC	
00405C8A	> E9 6C050000	JMP test2_vm.004061FB	
00405C8F	> 9C	PUSHFD	
00405C90	E8 1EEBFFFF	CALL test2_vm.004047B4	

## 解密常量

0x00405CCF

0x0B

地址	HEX 数据	反汇编	注释
00405CC4	. 8F4424 48	POP DWORD PTR SS:[ESP+48]	
00405CC8	. F8	CLC	
00405CC9	. FF35 6B614000	PUSH DWORD PTR DS:[40616B]	
00405CCF	. 8F4424 44	POP DWORD PTR SS:[ESP+44]	
00405CD3	. 66:0FCE	BSWAP SI	
00405CD6	. C74424 40 00	MOV DWORD PTR SS:[ESP+40], 0	
00405CDE	. F8	CLC	
00405CDF	. F9	STC	
00405CE0	. 66:D3C6	ROL SI, CL	
00405CE3	. 8B7424 70	MOV ESI, DWORD PTR SS:[ESP+70]	
00405CE7	. FF7424 04	PUSH DWORD PTR SS:[ESP+4]	
00405CEB	. 9C	PUSHFD	
00405CEC	. F9	STC	
00405CED	. C74424 0C 6C	MOV DWORD PTR SS:[ESP+C], 4EC06C6C	
00405CF5	. 0FCE	BSWAP ESI	
00405CF7	. 66:C70424 59	MOV WORD PTR SS:[ESP], 6759	
00405CFD	^ E9 26E9FFFF	JMP test2_vm.00404628	
00405D02	\$ C1CE 12	ROR ESI, 12	
00405D05	. E8 D5FCFFFF	CALL test2_vm.004059DF	
00405D0A	\$ 8D7C24 04	LEA EDI, DWORD PTR SS:[ESP+4]	
00405D0E	. 8D6424 04	LEA ESP, DWORD PTR SS:[ESP+4]	
00405D12	> D3E3	SHL EBX, CL	
00405D14	. 89F3	MOV EBX, ESI	
00405D16	. D2F1	SAL CL, CL	
00405D18	. D2E0	SHL AL, CL	
00405D1A	. 49	DEC ECX	
00405D1B	. 0375 00	ADD ESI, DWORD PTR SS:[EBP]	

地址	HEX 数据	反汇编	注释
00405CDE	. F8	CLC	
00405CDF	. F9	STC	
00405CE0	. 66:D3C6	ROL SI, CL	
00405CE3	. 8B7424 70	MOV ESI, DWORD PTR SS:[ESP+70]	
00405CE7	. FF7424 04	PUSH DWORD PTR SS:[ESP+4]	
00405CEB	. 9C	PUSHFD	
00405CEC	. F9	STC	
00405CED	. C74424 0C 6C	MOV DWORD PTR SS:[ESP+C], 4EC06C6C	
00405CF5	. 0FCE	BSWAP ESI	
00405CF7	. 66:C70424 59	MOV WORD PTR SS:[ESP], 6759	
00405CFD	. ^ E9 26E9FFFF	JMP test2_vm.00404628	
00405D02	\$ C1CE 12	ROR ESI, 12	
00405D05	. E8 D5FCFFFF	CALL test2_vm.004059DF	
00405D0A	\$ 8D7C24 04	LEA EDI, DWORD PTR SS:[ESP+4]	
00405D0E	. 8D6424 04	LEA ESP, DWORD PTR SS:[ESP+4]	
00405D12	> D3E3	SHL EBX, CL	
00405D14	. 89F3	MOV EBX, ESI	
00405D16	. D2F1	SAL CL, CL	
00405D18	. D2E0	SHL AL, CL	
00405D1A	. 49	DEC ECX	
00405D1B	. 0375 00	ADD ESI, DWORD PTR SS:[EBP]	
00405D1E	> 66:0FBF01	BTC CX, SI	
00405D22	. 0F94C0	SETB AL	
00405D25	. 8A46 FF	MOV AL, BYTE PTR DS:[ESI-1]	
00405D28	. D2E5	SHL CH, CL	
00405D2A	. 00D8	ADD AL, BL	
00405D2C	. 66:0BACD9 02	SHRD CX, BX, 2	

0x00405D69

0x0B

地址	HEX 数据	反汇编	注释
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 00	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	
00405D93	> FF7424 3C	PUSH DWORD PTR SS:[ESP+3C]	
00405D97	. 8F45 00	POP DWORD PTR SS:[EBP]	
00405D9A	. FF3424	PUSH DWORD PTR SS:[ESP]	
00405D9D	. 8D6424 44	LEA ESP, DWORD PTR SS:[ESP+44]	
00405DA1	. ^ E9 A4ECFFFF	JMP test2_vm.00404A4A	
00405DA6	. D2E0	SHL AL, CL	
00405DA8	. F9	STC	
00405DA9	. 66:0FBCC0	BSF AX, AX	
00405DAD	. F6DC	NEG AH	
00405DAF	. 66:0FB746 FE	MOVZX AX, WORD PTR DS:[ESI-2]	
00405DB4	. 53	PUSH EBX	
00405DB5	. 86E0	XCHG AL, AH	
00405DB7	. F9	STC	
00405DB8	. 66:C70424 50	MOV WORD PTR SS:[ESP], 0B150	

其它的省略.....上海烛龙信息科技有限公司

VMP 校验的大部分都是几个地方,入口,出口,校验指令,解密常数指令,esi 赋值的时候。

## VMP 花指令

当我把所有的常量收缩后,发现这里的花指令要比 X86 下面的花指令类型简单多了,仅仅只有两种。

就是 mov reg,reg 和 mov reg,const



但是有一个问题需要搞清楚，就是种类少并不等于容易识别。

我们来看一组好了，还是上面跳转的例子，VMP 在每次跳转前都会将数据移动，打乱寄存器，那么原来的数据干什么东西呢，答案就是做花指令，恶心死你。

VMP\_PUSHC\_IMM32 REG\_2C

VMP\_POPC\_IMM32 REG\_14

这两条指令发生在跳转前，这里的 REG 保存的是真正 EBP 的值(0x13FFF0)，已经从 2C 移到 14，2C 寄存器被废弃了。

VMP\_PUSHC\_IMM32 REG\_2C

#先来一个假的 mov reg,reg 忽悠一下

VMP\_POPC\_IMM32 REG\_10

VMP\_PUSHC\_IMM32 REG\_2C

VMP\_PUSHVM ESP

VMP\_MOVIN\_DWORD\_ADDR\_SS [0x13FFB4] (0x13FFF0)

NAND\_DWORD 0x13FFF0, 0x13FFF0 (0xFFEC000F)

VMP\_POPC\_IMM32 REG\_1C

VMP\_PUSH\_DWORD\_CONST 0x356542FD

NAND\_DWORD 0x356542FD, 0xFFEC000F (0x12BD00)

VMP\_POPC\_IMM32 REG\_38

VMP\_PUSHC\_IMM32 REG\_10

VMP\_PUSH\_DWORD\_CONST 0xCA9ABD02

NAND\_DWORD 0xCA9ABD02, 0x13FFF0 (0x3564000D)

VMP\_POPC\_IMM32 REG\_1C

NAND\_DWORD 0x3564000D, 0x12BD00 (0xCA8942F2)

VMP\_POPC\_IMM32 REG\_34

VMP\_POPC\_IMM32 REG\_34

第一阶段暂时完成，先忽悠一下，执行个 XOR EBP,CA9ABD02

地址	HEX 数据	ASCII
0013FED0	2E 79 40 00 FF FF FF FF 00 00 00 00 00 40 FD 7F	.y@. ....
0013FEE0	FO FF 13 00 FO FF 13 00 FF FF FF FF 02 02 00 00	??.??. ..
0013FEF0	38 07 93 7C 00 00 00 00 00 40 FD 7F 93 02 00 00	8•措.....@??..
0013FF00	63 95 3D 20 F2 42 89 CA 06 02 00 00 94 EB 92 7C	c? 駢塢...旌祛
0013FF10	20 90 95 88 18 00 00 00 00 5C B8 B4 18 1C 24 E5	恩?...\复佻
0013FF20	01 00 00 00 00 00 00 00 98 00 00 00 E3 62 61 80	r.....?.絞a€
0013FF30	01 00 00 00 06 00 00 00 80 5C B8 B4 98 63 61 80	r.....€\复楷a€
0013FF40	88 1B 24 E5 00 00 00 00 00 00 00 00 08 00 EB E4	?\$......o.腕
0013FF50	00 00 00 00 79 B3 D3 34 12 54 40 00 B1 2C A4 64	...y秤4IT@.?
0013FF60	64 FF 13 00 92 02 00 00 00 07 93 7C 28 7C 40 00	d .?.?...措( @.
0013FF70	FO FF 13 00 88 FF 13 00 00 40 FD 7F 94 EB 92 7C	??.??.@?旌祛
0013FF80	B0 FF 13 00 00 00 00 00 92 02 00 00 2E 79 40 00	??.?...?.y@.
0013FF90	06 02 00 00 63 95 3D 20 FO FF 13 00 94 EB 92 7C	...c? ??.旌祛
0013FFA0	93 02 00 00 38 07 93 7C 00 40 FD 7F 02 02 00 00	?..8•措.@?..
0013FFB0	82 02 00 00 F2 42 89 CA B0 FF 13 00 06 02 00 00	?..駢塢?..

图上面绿色的是真正的 EBP, 蓝色的是忽悠值, 红色是变形后的忽悠值, 白色的是被抹掉的数据源。因为现在看到的算法其实在 VMP 里面都不是连续的, 有些寄存器在引用后就马上给抹掉了, 这里的 REG\_2C 就是这样, 这里的指令是我识别后再重新组合的。中间继续正常的运算, 直到第二次的跳转, 这个时候所有寄存器再次移动, 忽悠的寄存器也跟着开始动作。真正的 EBP 已经不用管它了, 我们追踪的是忽悠人的寄存器。

VMP\_PUSHC\_IMM32 REG\_34

VMP\_PUSHVM\_ESP

VMP\_MOVIN\_DWORD\_ADDR\_SS [0x13FFB4] (0xCA8942F2)

VMP\_POPC\_IMM32 REG\_0

#还是一个忽悠人的 mov reg,reg 做得和真的一样



VMP\_PUSHC\_IMM32 REG\_0

NAND\_DWORD 0xCA8942F2, 0xCA8942F2 (0x3576BD0D)

VMP\_POPC\_IMM32 REG\_C

VMP\_PUSH\_DWORD\_CONST 0x356542FD

NAND\_DWORD 0x356542FD, 0x3576BD0D (0xCA880002)

VMP\_POPC\_IMM32 REG\_4

VMP\_PUSH\_DWORD\_CONST 0xCA9ABD02

VMP\_PUSHC\_IMM32 REG\_0

NAND\_DWORD 0xCA8942F2, 0xCA9ABD02 (0x3564000D)

VMP\_POPC\_IMM32 REG\_2C

NAND\_DWORD 0x3564000D, 0xCA880002 (0x13FFF0)

VMP\_POPC\_IMM32 REG\_1C

VMP\_POPC\_IMM32 REG\_1C

看到 0x13FFF0 我想你已经明白怎么回事了吧, 没有错, 还是 XOR EBP,CA9ABD02



那么这个花指令到底是什么东东呢, 以我的理解, 这里执行的是 MOV REG\_1C,(EBP^ CA9ABD02^ CA9ABD02)只不过在执行中顺便抹掉了几个寄存器。

地址	HEX 数据	ASCII
0013FED0	F2 42 89 CA 82 02 00 00 06 01 00 00 02 02 00 00	驗墟?... ..
0013FEE0	63 95 3D 20 94 EB 92 7C 53 7F 40 00 F0 FF 13 00	c? 旗祛S @.?..
0013FEF0	FF FF FF FF F0 FF 13 00 17 02 00 00 02 02 00 00	?..4?... ..
0013FF00	00 00 00 00 00 40 FD 7F 38 07 93 7C 94 EB 92 7C	....@?8*措旗祛
0013FF10	20 90 95 88 18 00 00 00 00 5C B8 B4 18 1C 24 E5	恩?... \复祛
0013FF20	01 00 00 00 00 00 00 00 98 00 00 00 E3 62 61 80	.....?. 祛a€
0013FF30	01 00 00 00 06 00 00 00 80 5C B8 B4 98 63 61 80	.....€\复祛a€
0013FF40	88 1B 24 E5 00 00 00 00 00 00 00 00 08 00 EB E4	?\$. .... . 祛
0013FF50	00 00 00 00 79 B3 D3 34 12 54 40 00 B1 2C A4 64	....秤4!祛?
0013FF60	64 FF 13 00 92 02 00 00 00 07 93 7C 28 7C 40 00	d ..?. ...*措( @.
0013FF70	F0 FF 13 00 88 FF 13 00 00 40 FD 7F 94 EB 92 7C	?..?.. @?旗祛
0013FF80	B0 FF 13 00 00 00 00 00 92 02 00 00 53 7F 40 00	?.. ....?. S @.
0013FF90	06 02 00 00 63 95 3D 20 F0 FF 13 00 94 EB 92 7C	... c? ?.. 旗祛
0013FFA0	17 02 00 00 38 07 93 7C 00 40 FD 7F 02 02 00 00	...8*措.@?..
0013FFB0	06 02 00 00 F0 FF 13 00 B0 FF 13 00 06 02 00 00	...?..?..

## 五. 最后的一点废话

终于都完成了，追踪 VMP 一共花了我两年多的时间，终于都暂时告一段落了。这里写的东西都是一些很基本的东西，在其它的虚拟机里也或多或少也会遇到。学习虚拟机其实就是学习汇编，VMP 的作者实在是太牛了，不知道什么时候才能有这样的境界.....如果你看不懂上面写的东西，那么这是正常的，因为我从小学开始语文就从来没有及格过。如果你想对 VMP 有更深入的了解，那么不要犹豫，马上拿起 OD。如果你有什么问题不明白的话，请不要问我，因为我要暂时放下 VM，去追比 VM 更高难度、对所有

算法都免疫、任何预测也无效、连名字都更加恐怖的.....MM 。你可以到看雪或者一蓑烟雨膜拜一下大牛，网址请看左上角。

## 相关工具

VMProtect.v1.63.Unpacked.by.海风月影[CUG]

VMProtect.Professional.V1.8.Custom.Build.Cracked.By.Nooby[UnPacKcN]

VMProtect.v2.03.Ultimate.Cracked.By.Nooby[UnPacKcN]

VMProtect.Ultimate.V2.0.4.4140.Incl.License.Offer.By.1<sup>ST</sup>

Immunity Debugger 1.5 汉化修正 080417

ImmunityDebugger\_1.73\_RemoveAD\_KuNgBiM

HA\_OllyDBG\_1.10\_second\_cao\_cong\_fix22

古剑奇谭  
碧天新境令服社

一只小菜鸟

2010-05-17



上海烛龙信息科技有限公司  
Aurogon Info&Tec(Shanghai)Co.,Ltd

