

VMProtect 学习

By: OoWoodOne

2015.8

VMProtect 之所以叫做 VMProtect，因为它是以 VM（Virtual Machine）虚拟机为核心来实现的，这里的虚拟机并不是传统意义上的虚拟机，其是将汇编指令进行虚拟化，让其失去原本容易理解的含义，增大对逆向工程的难度。

一. 虚拟机

1) 虚拟机指令（VM_Handler）

VMP 中的虚拟机指令可分为一下几类：

出入栈：PushReg、PushImm、PushEsp、PopReg、PopEsp...

运算：Add、Div、IDiv、Imul、Mul、Nor、Shl、Shr...

内存读写：ReadDs、ReadFs、WriteDs、WriteFs...

一些特殊的指令：GetHash、Jmp、Cpuid、CallApi、Rdtsc...

当然还有一些其他的如浮点运算的指令。

在这些指令中又可根据操作数大小的不同来进行细分。VMP 将原程序的指令进行分解转换为由这些 Handler 组成的新的指令操作集合，实现程序的虚拟化。这些 Handler 的地址存放于一张指令表中，通过调度执行。从 Opcode 的解码过程中可以看出，该表的大小为 0x100。

2) 虚拟机调度（VM_Dispatch）

VMP 中的虚拟机调度一般以这样的形式存在：

Opcode 解码

mov Reg0,dword ptr ds:[Reg1*4+<VMHandlerTable>]

Handler 地址解码

转到 Handler 执行

3) 虚拟机 Context

VM_Context 主要涉及对真实环境中上下文的保存，以及一些数据的传递。其组成主要有：

其中的寄存器存储位置在每个虚拟机中都有所不同。VMP 中对一些寄存器的使用也有规定比如：ESI 记录 opcode 的位置（VM_Eip），EBP 记录 VM 使用堆栈的位置（VM_Esp），EDI 中储存着 VM_Context 的偏移。

4) 虚拟机入口执行过程

以下是一个真实虚拟机入口的执行流程：

- Print Chain: 01160195 Unoptimize

01160195 push eax

01160196	mov dword ptr ss:[esp],0xC65BD31	Push INITKEY
0116019D	push esp	
0116019E	call 010AFFD7	
010AFFD7	call 0115896F	
0115896F	mov byte ptr ss:[esp],dl	
01158972	mov dword ptr ss:[esp+0x8],0xAF99D832	Push RETADDR
0115897A	pushad	
0115897B	push 0x58419810	
01158980	lea esp,dword ptr ss:[esp+0x2C]	
01158984	jmp 01163280	
01163280	pushad	
01163281	mov dword ptr ss:[esp+0x1C],esi	Push ESI
01163285	mov byte ptr ss:[esp+0x4],0xD0	
0116328A	pushad	
0116328B	push eax	
0116328C	pushfd	
0116328D	lea esp,dword ptr ss:[esp+0x44]	
01163291	jmp 01163020	
01163020	pushad	
01163021	pushad	
01163022	call 0116249A	
0116249A	xchg dword ptr ss:[esp+0x40],ebx	Push EBX
0116249E	movsx ebx,dl	
011624A1	jmp 011635CD	
011635CD	pushfd	
011635CE	mov dword ptr ss:[esp+0x40],eax	Push EAX
011635D2	jmp 0116227B	
0116227B	lea ebx,dword ptr ds:[0x434815CD]	
01162282	jmp 01162C4D	
01162C4D	xchg dword ptr ss:[esp+0x3C],esi	Push ESI
01162C51	movsx bx,cl	
01162C55	mov dword ptr ss:[esp+0x38],ebp	Push EBP
01162C59	bswap bp	
01162C5C	movzx esi,dl	
01162C5F	not si	
01162C62	prefix rep:	
01162C63	pushfd	
01162C64	pop dword ptr ss:[esp+0x34]	Push EFL
01162C68	movsx bp,dl	
01162C6C	pushfd	
01162C6D	mov dword ptr ss:[esp+0x34],edx	Push EDX
01162C71	movzx si,bl	
01162C75	push ecx	
01162C76	setge al	

01162C79	movsx si,dl	
01162C7D	mov dword ptr ss:[esp+0x34],edi	Push EDI
01162C81	not esi	
01162C83	pushfd	
01162C84	lea ebx,dword ptr ds:[ebp*4+0xEF0CD61]	
01162C8B	xchg dword ptr ss:[esp+0x34],ecx	Push ECX
01162C8F	jmp 01162F5C	
01162F5C	bswap cx	
01162F5F	push dword ptr ds:[0x11622ED]	
01162F65	pop dword ptr ss:[esp+0x30]	Push ANTIDUMP
01162F69	movsx di,dl	
01162F6D	mov dword ptr ss:[esp+0x2C],0x0	Push RELOC
01162F75	bswap ebp	
01162F77	setpe bl	
01162F7A	jmp 0116273A	
0116273A	mov esi,dword ptr ss:[esp+0x5C]	Initkey Load
0116273E	bswap edi	
01162740	setnb bl	
01162743	inc bp	
01162746	movzx cx,cl	
0116274A	dec esi	InitKey Decode
0116274B	bswap ebp	
0116274D	push dword ptr ss:[esp]	
01162750	rol esi,0x1F	InitKey Decode
01162753	call 01161CAC	
01161CAC	adc bl,0x35	
01161CAF	xor esi,0x739F6DE	InitKey Decode
01161CB5	sar cl,0x7	
01161CB8	xor edi,ebx	
01161CBA	rcr bx,0x7	
01161CBE	pop edi	
01161CBF	lea ebp,dword ptr ss:[esp+0x30]	Reloc Load
01161CC3	bswap ecx	
01161CC5	lea edi,dword ptr ds:[eax+0x426F19D8]	
01161CCB	sub di,bp	
01161CCE	sub esp,0x90	
01161CD4	add al,ch	
01161CD6	sub di,bx	
01161CD9	sbb di,ax	
01161CDC	mov edi,esp	
01161CDE	or al,bh	
01161CE0	adc al,0x74	
01161CE2	rcr al,1	
01161CE4	ror bl,0x5	

01161CE7	mov ebx,esi	Join Reg Load
01161CE9	shrd cx,si,0xB	
01161CEE	add esi,dword ptr ss:[ebp]	Reloc Add
01161CF1	cmp al,bh	
01161CF3	and al,0xF8	
01161CF5	pushfd	
01161CF6	sub esp,-0x4	
01161CF9	jb 0116304C	
0116304C	rol cx,cl	
0116304F	btc cx,0x6	
01163054	push 0xAB13427B	
01163059	mov al,byte ptr ds:[esi-0x1]	VMP Opcode Load
0116305C	call 01162CA5	
01162CA5	xor al,bl	Opcode Decode
01162CA7	lea ecx,dword ptr ss:[esp-0x7766A095]	
01162CAE	inc ecx	
01162CAF	sal ch,0x2	
01162CB2	bsf cx,dx	
01162CB6	neg al	Opcode Decode
01162CB8	bts cx,0xE	
01162CBD	inc ch	
01162CBF	btc ecx,ecx	
01162CC2	xor al,0xA	Opcode Decode
01162CC4	push 0xCB600CD6	
01162CC9	inc al	Opcode Decode
01162CCB	movzx ecx,cl	
01162CCE	btc cx,0x5	
01162CD3	shl cl,cl	
01162CD5	xor bl,al	
01162CD7	bswap ecx	
01162CD9	movzx eax,al	Opcode Decode
01162CDC	sbb cx,di	
01162CDF	stc	
01162CE0	bts cx,dx	
01162CE4	shld ecx,ebp,0x1A	
01162CE8	mov ecx,dword ptr ds:[eax*4+0x1161899]	Dispatch Cmd
01162CEF	test ah,bh	
01162CF1	clc	
01162CF2	clc	
01162CF3	add esi,-0x1	
01162CF6	mov byte ptr ss:[esp],0xF6	
01162CFA	rol ecx,0x4	Handler Decode
01162CFD	pushfd	
01162CFE	pushfd	

```

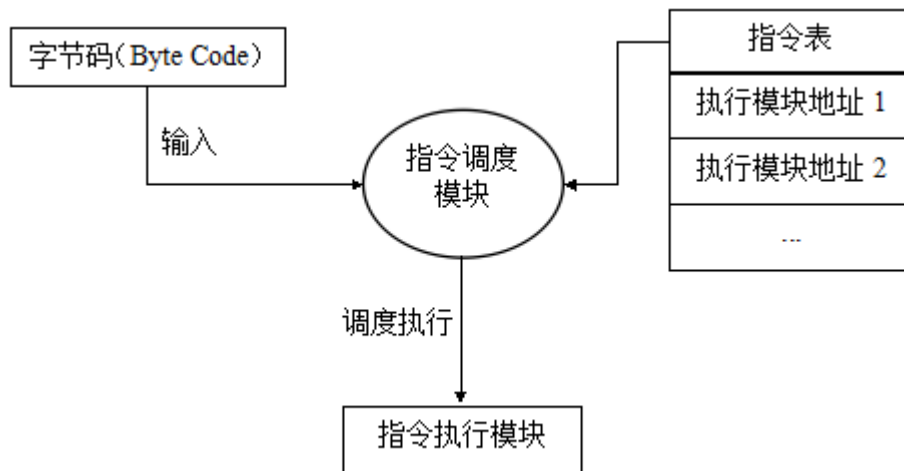
01162CFF    push dword ptr ss:[esp]
01162D02    mov dword ptr ss:[esp+0x14],ecx
01162D06    pushfd
01162D07    pushfd
01162D08    pushad
01162D09    push esi
01162D0A    push dword ptr ss:[esp+0x40]
01162D0E    retn 0x44

```

VM Entry

5) 虚拟机调度过程

虚拟机从 `initkey` 数据解码得到 Opcode 地址，然后逐个提取解码 Opcode，交由调度模块执行相应的 VM 指令。



二. 虚拟机的识别

1) 获取指令的 use-def 链

首先需要读取指令流，并且记录每个指令对哪些数据进行了定义或者使用（引用）了哪些数据，组成简单的 `ud-chain`。在 VMP 的混淆优化中，需要进行 `ud` 记录的数据主要是寄存器、堆栈位置数据以及 `ebp` 相关。在读取指令流的过程中，遇到可以进行跳转识别的指令也需要进行处理，比如遇到 `jb xxxxxxxx` 的跳转指令时，它的前一个指令是 `sub esp,-4`，则可以从标志位的变化得出该跳转是必定实现的，读取指令就跳到地址处继续。

2) VMP 中混淆的优化

VMP 中混淆可分为几类：

乱序：乱序就是将代码段分散到内存中不同位置来执行，使用跳转指令来连接。对于乱序混淆，在识别跳转读取指令链的时候就可以很好的处理。

无效指令：VMP 中的无效指令一般就是对某个数据的重复定义。如如下混淆：

```

Add eax,8
Xor eax,ebx
Sub ecx,eax
Mov eax,0

```

这里第 1、2 个指令对 `eax` 进行了重新定义，而在这之间有没有对 `eax` 的引用，因而第 1 个指令就是无效指令。而 2 和 4 之间因为有对 `eax` 的使用，所以不能将 2 作为无效指令，除

非第 3 个指令已经被定义为无效指令。

堆栈混淆：VMP 中基于堆栈的混淆类似于：

`Push xxx //push 无效数据`

`Pushx //push 无效数据`

`Lea esp,[esp+xx] // sub eap,-xx //add esp,xx //retn xx //恢复堆栈`

堆栈上混淆的优化需要记录每个指令对 esp 的操作大小以及对堆栈位置的引用，在优化时通过恢复堆栈的大小来判断之前哪些是无效的入栈指令。

3) VM_Context 识别

从虚拟机入口开始，记录堆栈位置以及每个堆栈位置中数据的类型（寄存器或其他），context 数据的入栈顺序是 `initkey->retaddr->原环境寄存器->antidump->relocation`。从头遍历指令链，遇到有对堆栈位置数据进行定义的，再根据指令来判断是什么类型的数据。

4) VM_Dispatch 识别

遍历指令链找到 `mov Reg0,dword ptr ds:[Reg1*4+const]` 格式的指令，如果 const 的值在模块范围内，该指令一般就是调度指令，const 就是 VM_Handler 表的首地址。

5) VM_Handler 识别

在获取 VM_Dispatch 后，从 VM_Dispatch 开始遍历，获取所有对 Reg0 进行定义指令，组成的指令链为 handler 的解码指令合集。然后创建一个虚拟的指令执行体来执行解码指令，这里可以对个指令进行模拟，也可以将指令的二进制码注入到自己的程序中来执行。

通过 VM_Handler 表的首地址获取 0-0xff 范围内所有的指令地址码，然后通过执行解码指令获取真实的 Handler 地址。获取地址后，读取已该地址为首的指令链，最后通过匹配特征码来解析出真实的 Handler。在 VMP 中虚拟机的堆栈位置（VM_Esp）是在 ebp 中记录的，所以对 ebp 指定位置有定义或者引用的指令都能拿来作为特征指令，另外特征指令也可以是 esi(VM_EIP)相关、关键操作指令、特殊指令等。比如 VM_ReadDs32 的指令特征：

`mov eax,dword ptr ss:[ebp]`

`mov eax,dword ptr ds:[eax]`

`mov dword ptr ss:[ebp],eax`

VM_CPUID 的特征指令：

`mov eax,dword ptr ss:[ebp]`

`cuid`

`mov dword ptr ss:[ebp+0xc],eax`

`mov dword ptr ss:[ebp+0x8],ebx`

`mov dword ptr ss:[ebp+0x4],ecx`

`mov dword ptr ss:[ebp],edx`

VM_Nor32 的指令特征：

`mov eax,dword ptr ss:[ebp]`

`mov edx,dword ptr ss:[ebp+0x4]`

`not eax`

`not edx`

`and eax,edx`

`mov dword ptr ss:[ebp+0x4],eax`

三. VMP Hash Check

1. Hash Check

VMP 使用 VMProtectIsValidImageCRC 来进行内存检测，一个关键的 Handler 就是 VM_GetHash，以下是一个完整的 VM_GetHash：

```
- Print Chain: 0116359A Unoptimize
0116359A btr cx,0xF VM_GetHash
0116359F not cl
011635A1 mov edx,dword ptr ss:[ebp] ;取需要校验的地址
011635A4 test ebp,ebp
011635A6 add ebp,0x4
011635A9 sets ch
011635AC xor ch,ch
011635AE xor eax,eax
011635B0 or ch,dh
011635B2 clc
011635B3 mov ecx,eax
011635B5 stc
011635B6 shl eax,0x7 ;shl
011635B9 push 0x645E01DB
011635BE pushad
011635BF test ax,di
011635C2 shr ecx,0x19 ;shr
011635C5 clc
011635C6 or eax,ecx ;or
011635C8 call 01162BFC
01162BFC cmp sp,0xC297
01162C01 cmp cx,bx
01162C04 xor al,byte ptr ds:[edx] ;读取字节
01162C06 pushfd
01162C07 inc edx ;地址+1
01162C08 jmp 01161314
01161314 push eax
01161315 push ecx
01161316 push dword ptr ss:[esp+0x4]
0116131A dec dword ptr ss:[ebp] ;size-=1
0116131D jmp 011614E3
011614E3 call 0116352B
0116352B push 0x6170E4CF
01163530 lea esp,dword ptr ss:[esp+0x40]
01163534 jnz 011635B0 ;循环
0116353A pushad
0116353B push dword ptr ss:[esp+0x8]
0116353F jmp 01162D64
```

```

01162D64    mov dword ptr ss:[ebp],eax      ;返回值
01162D67    pushad
01162D68    lea esp,dword ptr ss:[esp+0x44]
01162D6C    jmp 01161CF1
01161CF1    cmp al,bh
01161CF3    and al,0xF8
01161CF5    pushfd
01161CF6    sub esp,-0x4
01161CF9    jb 0116304C

```

去除花指令和混淆后，VM_GetHash 可以化简为：

```

START:
mov edx,dword ptr ss:[ebp]
add ebp,0x4
LOOP:
xor eax,eax
mov ecx,eax
shl eax,0x7
shr ecx,0x19
or eax,ecx
xor al,byte ptr ds:[edx]
inc edx
dec dword ptr ss:[ebp]
jnz LOOP
mov dword ptr ss:[ebp],eax
END

```

GetHash 入口 VM 堆栈：

```

$ ==>    >0066A5CD      ;address
$+4      >000B1233      ;size

```

Hash 元素的结构如下：

```

Struct Hash_Block
{
    Ulong address;
    Ulong size;
    Ulong value;
};

```

需要校验的 Hash 数据被加密存储在一张表中，VMP 使用 VM_ReadDs32 从内存中读取

hash 元素数据，我这里写了个脚本来观察过程，在 VM_ReadDs32 和 VM_Hash 下断，获取相关信息。

```
VH:offset <notepad_.VM_RmDs32>    VM_EIP:10B558D    ADDR:10B919D    VALUE:F41B0BE0
VH:offset <notepad_.VM_RmDs32>    VM_EIP:10B194A    ADDR:10B91A1    VALUE:EDF4FECB
VH:offset <notepad_.VM_Hash>      VM_EIP:10B18CB    ADDR:3DA5CD     SIZE:B1233     VALUE:E0B3760D
VH:offset <notepad_.VM_RmDs32>    VM_EIP:10B182B    ADDR:10B91A5    VALUE:4340F6D3
VH:offset <notepad_.VM_RmDs32>    VM_EIP:10B558D    ADDR:10B91A9    VALUE:F3753EE0
VH:offset <notepad_.VM_RmDs32>    VM_EIP:10B194A    ADDR:10B91AD    VALUE:FEFFFECD
VH:offset <notepad_.VM_Hash>      VM_EIP:10B18CB    ADDR:3D0000     SIZE:138      VALUE:9A916A82
VH:offset <notepad_.VM_RmDs32>    VM_EIP:10B182B    ADDR:10B91B1    VALUE:C84CD8A9
VH:offset <notepad_.VM_RmDs32>    VM_EIP:10B558D    ADDR:10B91B5    VALUE:F376BEE0
VH:offset <notepad_.VM_RmDs32>    VM_EIP:10B194A    ADDR:10B91B9    VALUE:5BFFFEED
VH:offset <notepad_.VM_Hash>      VM_EIP:10B18CB    ADDR:3D0180     SIZE:A41D     VALUE:3DA4B672
VH:offset <notepad_.VM_RmDs32>    VM_EIP:10B182B    ADDR:10B91BD    VALUE:9800E76E
VH:offset <notepad_.VM_RmDs32>    VM_EIP:10B558D    ADDR:10B91C1    VALUE:F3767AE0
VH:offset <notepad_.VM_RmDs32>    VM_EIP:10B194A    ADDR:10B91C5    VALUE:FFFFFEC4
VH:offset <notepad_.VM_Hash>      VM_EIP:10B18CB    ADDR:3D013C     SIZE:3C       VALUE:B60DC125
VH:offset <notepad_.VM_RmDs32>    VM_EIP:10B182B    ADDR:10B91C9    VALUE:EBF75B06
.....
```

可以看出 VMP 会先从表中读取两个数据，按偏移也就是 Hash 元素中的 Address 和 Size，之后调用 VM_GetHash 获取给定地址和大小的 Hash 值，与再一次读取到的 Hash 元素中 Value 值进行比较。

在完整的结果中可以看出，调用 VM_GetHash 分为 4 个阶段：

第一阶段文件 Hash 校验，该阶段调用 hash 次数不是很多，但 size 一般较大，目的是校验文件的完整性。

第二阶段是 Code 校验，就是在壳执行过程中对壳代码数据的校验，检测 patch 代码或者 Int3 断点。

```
VH:offset <notepad_.VM_Hash>      VM_EIP:10AFC62    ADDR:10B1C55    SIZE:174      VALUE:6DFE368E
VH:offset <notepad_.VM_RmDs32>    VM_EIP:10AFC5B    ADDR:1159388    VALUE:6DFE368E
VH:offset <notepad_.VM_RmDs32>    VM_EIP:10AFB78    ADDR:10B1C55    VALUE:8ACF5537
VH:offset <notepad_.VM_RmDs32>    VM_EIP:11603BC    ADDR:10B1C59    VALUE:225E9A79
VH:offset <notepad_.VM_Hash>      VM_EIP:1160314    ADDR:10B1DC9    SIZE:73D4     VALUE:38F28D16
VH:offset <notepad_.VM_RmDs32>    VM_EIP:11602A2    ADDR:10B1C5D    VALUE:81E64450
VH:offset <notepad_.VM_RmDs32>    VM_EIP:10AFB78    ADDR:10B1C61    VALUE:874F09A5
VH:offset <notepad_.VM_RmDs32>    VM_EIP:11603BC    ADDR:10B1C65    VALUE:203B2A81
VH:offset <notepad_.VM_Hash>      VM_EIP:1160314    ADDR:1000180    SIZE:120      VALUE:DE9AA4A0
VH:offset <notepad_.VM_RmDs32>    VM_EIP:11602A2    ADDR:10B1C69    VALUE:278F5BDA
.....
```

该部分会先对 Hash 表的数据完整性进行检测，因此可以从第一个 VM_GetHash 中获取到该阶段 Hash 表的地址和大小。

第三阶段是内存校验，该阶段是在原程序数据解码后进行的，主要是对原程序数据的完整性校验，该阶段和第二阶段一样，第一个 VM_GetHash 会先对 Hash 表进行检测。

第四阶段是随机校验，其会先调用 VM_Rdstc 获取一个随机数，随机对内存中数据进行校验。

2. Patch Hash

这里 Patch Hash 有两种方案,第一种是将 Hash 表的数据进行替换,然后对 VM_GetHash 进行 Patch, 让它放回同个值。

第二种是对 Hash 检验中的循环变量进行修改,取消循环次数。在运行到 VM_GetHash 时 VM_Context(edi)中的某个值就是循环变量,但这个变量的位置在每个虚拟机中是不同的,需要手动观察判断,下面是某个 VM_GetHash 中的 Context:

```
000CF6A0  000B1233
000CF6A4  00000000
000CF6A8  000CFF60
000CF6AC  00000202
000CF6B0  010B919D  notepad_.010B919D
000CF6B4  000B1232
000CF6B8  00000202
000CF6BC  0066F4FF
000CF6C0  00660000
000CF6C4  000CF760
000CF6C8  00000206
000CF6CC  00000282
000CF6D0  00000004
000CF6D4  0000A5CD
000CF6D8  9C0FD700
000CF6DC  0066A5CD
```

其中 000CF6D0 位置的值就是循环变量,在这里将值改为 1 就能跳过接下来的 Hash 校验。

对于第四种随机 Hash 校验,需要 Patch VM_Rdstc 使其返回某个固定的值,最后每次 Hash 校验固定的位置。

四. VMProtect 脱壳

1. LocalAlloc Anti-Dump

为了防止程序被转储脱壳,VMP 会从多个方面来 AntiDump,其中第一个就是申请内存并在内存中放入一些关键数据,在 VMP 运行过程中会读取这些关键数据进行验证,如果脱壳后没有处理这部分数据,运行过程无法读取到相应的数据,程序将无法运行。如下是记录 LocalAlloc Anti-Dump 的过程:

```
.....
10B091B:VM_CallApi -->11614CD:LocalAlloc  Retn:5656F0
.....
1160786:VM_WriteEs32 -->Addr:5656F0
.....
1160786:VM_WriteEs32 -->Addr:5656F4
.....
```

1160786:VM_WriteEs32 -->Addr:5656F8
.....
1160786:VM_WriteEs32 -->Addr:5656FC
.....
1160786:VM_WriteEs32 -->Addr:565700
.....
1160786:VM_WriteEs32 -->Addr:565704
.....
1160786:VM_WriteEs32 -->Addr:565708
.....
1160786:VM_WriteEs32 -->Addr:56570C
.....
1160786:VM_WriteEs32 -->Addr:565710
.....
1160786:VM_WriteEs32 -->Addr:565714
.....
1160786:VM_WriteEs32 -->Addr:565718
.....
1160786:VM_WriteEs32 -->Addr:56571C
.....
1160786:VM_WriteEs32 -->Addr:565720
.....
1160786:VM_WriteEs32 -->Addr:565724
.....
1160786:VM_WriteEs32 -->Addr:565728
.....
1160786:VM_WriteEs32 -->Addr:56572C
.....
1160786:VM_WriteEs32 -->Addr:565730
.....
1160786:VM_WriteEs32 -->Addr:565734
.....
1160786:VM_WriteEs32 -->Addr:565738
.....
1160786:VM_WriteEs32 -->Addr:56573C
.....
1160786:VM_WriteEs32 -->Addr:565740
.....
1160786:VM_WriteEs32 -->Addr:565744
.....
1160786:VM_WriteEs32 -->Addr:565748
.....
1160786:VM_WriteEs32 -->Addr:56574C
.....

```

1160786:VM_WriteEs32 -->Addr:565750
.....
1160786:VM_WriteEs32 -->Addr:565754
.....
10B986E:VM_Cpuid
.....
10B9794:VM_WriteDs32 -->Addr:565748
.....

```

其中省略了一些对数据加密的指令，只记录保留一些关键指令和写数据指令，从中可以看出，在 LocalAlloc 申请到内存后，VMP 会向里面写入 $0x14 \times 4 = 0x68$ 大小的加密数据，之后会通过 CPUID 获取 CPU 的硬件绑定信息加密后存入 anti-dump 内存。

所以要处理 anti-dump 不仅要加密数据进行保存，还需要 Patch VM_CPUID 使其返回相同的值，这样才能完整的跨平台处理 LocalAlloc Anti-Dump。

2. Heap Anti-Dump

VMP 在壳运行过程中会调用 HeapCreate 来创建堆内存，用来储存一些关键数据（尚未清楚），以及提供后续解码资源数据的内存空间，所以该堆内存会在程序运行过程中将被申请使用。VMP 对这部分代码并没有 VM，但对 API 进行了隐藏，下面是调用 HeapCreate 部分：

```

0101C564  8BFF      mov edi,edi
0101C566  55        push ebp
0101C567  8BEC      mov ebp,esp
0101C569  33C0      xor eax,eax
0101C56B  3945 08   cmp dword ptr ss:[ebp+0x8],eax
0101C56E  6A 00     push 0x0
0101C570  0F94C0    sete al
0101C573  68 00100000 push 0x1000
0101C578  50        push eax
0101C579  E8 4F840376 call kernel32.HeapCreate          ; 调用
HeapCreate 创建堆
0101C57E  90        nop
0101C57F  A3 D4750201 mov dword ptr ds:[0x10275D4],eax    ; 堆地
址存入全局变量中
0101C584  85C0      test eax,eax
0101C586  75 02     jnz short notepad_.0101C58A
0101C588  5D        pop ebp
0101C589  C3        retn

```

申请的堆会在程序运行时随时被调用，因此要处理 Heap Anti-Dump 需要 dump heap 内存，并提供足够的空间（因为后续会多次申请，heap 内存需要足够）。

3. Resource Anti-Dump

VMP 对资源的处理个人觉得是难以完全修复的，能用的方法是沿用 VMP 对资源的处理。VMP 在壳运行阶段会对 4 个资源相关的 API 进行 HOOK，这 4 个 API 为：

LdrFindResource_U、LdrAccessResource、LoadStringA、LoadStringW。具体的 HOOK 只是跳转到 VMP 自身模拟的 API 中，

```
7762D2A1 ntdll.LdrFindResource_U    jmp notepad_.010118A0    ;
hook 到自身代码
```

```
7762D2A6    test byte ptr ds:[0x7FFE0389],0x1
7762D2AD    jnz ntdll.77676B88
7762D2B3    push esi
7762D2B4    push dword ptr ss:[ebp+0x14]
7762D2B7    push 0x0
7762D2B9    push dword ptr ss:[ebp+0x10]
7762D2BC    push dword ptr ss:[ebp+0xC]
7762D2BF    push dword ptr ss:[ebp+0x8]
7762D2C2    call ntdll.7762CA4C
7762D2C7    test byte ptr ds:[0x7FFE0389],0x1
7762D2CE    mov esi,eax
7762D2D0    jnz ntdll.77676B9F
7762D2D6    mov eax,esi
7762D2D8    pop esi
7762D2D9    pop ebp
7762D2DA    ret 0x10
```

```
7762D284 ntdll.LdrAccessResource    jmp notepad_.01011930    ;
hook 到自身代码
```

```
7762D289    je short ntdll.7762D2AF
7762D28B    adc bh,bh
7762D28D    je short ntdll.7762D2B3
7762D28F    adc bh,bh
7762D291    je short ntdll.7762D2B7
```

并且 VMP 也会申请内存记录被 hook 的代码，使执行完自身的代码后又去调用真实 API 执行。在申请这部分内存时，VMP 会从两个 API 相关模块的地址结束开始每次增加 0x1000 大小来调用 VirtualAlloc 申请内存，直到这两块内存被申请。

第一块申请 hook 内存：

```
77780000    8BFF    mov edi,edi
77780002    55      push ebp
77780003    8BEC    mov ebp,esp
77780005    - E9 9CD2EAF    jmp
ntdll.7762D2A6    ; 跳回到 LdrFindResource_U
7778000A    8BFF    mov edi,edi
7778000C    55      push ebp
7778000D    8BEC    mov ebp,esp
7778000F    FF7424 10    push dword ptr ss:[esp+0x10]
```

77780013	FF7424 10	push dword ptr ss:[esp+0x10]	
77780017		- E9 70D2EAF	jmp
ntdll.7762D28C		; 跳回到 LdrAccessResource	
7778001C	FF7424 10	push dword ptr ss:[esp+0x10]	
77780020	FF7424 10	push dword ptr ss:[esp+0x10]	
77780024	0000	add byte ptr ds:[eax],al	
77780026	0000	add byte ptr ds:[eax],al	
77780028	0000	add byte ptr ds:[eax],al	
7778002A	0000	add byte ptr ds:[eax],al	

第二块申请 hook 内存:

76A00000	8BFF	mov edi,edi	
76A00002	55	push ebp	
76A00003	8BEC	mov ebp,esp	
76A00005		- E9 4F4BF4FF	jmp
KERNELBA.76944B59		; 跳回到 LoadStringA	
76A0000A	8BFF	mov edi,edi	
76A0000C	55	push ebp	
76A0000D	8BEC	mov ebp,esp	
76A0000F	8BFF	mov edi,edi	
76A00011	55	push ebp	
76A00012	8BEC	mov ebp,esp	
76A00014		- E9 AD4BF4FF	jmp
KERNELBA.76944BC6		; 跳回到 LoadStringW	
76A00019	8BFF	mov edi,edi	
76A0001B	55	push ebp	
76A0001C	8BEC	mov ebp,esp	
76A0001E	0000	add byte ptr ds:[eax],al	
76A00020	0000	add byte ptr ds:[eax],al	
76A00022	0000	add byte ptr ds:[eax],al	

处理 VMP 资源的 anti-dump，需要添加自己的代码去 hook 这 4 个 api 到相关地址，并且为实现跨平台需要自己实现 hook 的调回。

4. OEP 定位

关于 VMP OEP 的查找网上也有很多的方法，其中最简单的就是在最后一次 VirtualProtect 后，对代码段下内存访问断点。先来看一下壳运行时 VirtualProtect 调用情况：

第一次调用：

```
000CF68C 011614CD /CALL 到 VirtualProtect
000CF690 01001000 |Address = notepad_.01001000
000CF694 00007748 |Size = 7748 (30536.)
000CF698 00000040 |NewProtect = PAGE_EXECUTE_READWRITE
000CF69C 000CFF60 \pOldProtect = 000CFF60
```

可以看出首次调用 VirtualProtect 是先将代码段内存的权限变成可写，这是在为数据的

解码做准备，从中也可以看出代码段的基址为 01001000，大小为 7748。

第二次调用：

```
000CF61C  011614CD  /CALL 到 VirtualProtect
000CF620  01001000  |Address = notepad_.01001000
000CF624  00007748  |Size = 7748 (30536.)
000CF628  00000020  |NewProtect = PAGE_EXECUTE_READ
000CF62C  000CFF60  \pOldProtect = 000CFF60
```

此次断下后，可以看出壳又将代码段权限改成了不可写，说明数据已经解码完毕，查看代码段数据也已经不为 0。数据解码后，当然很快就要将程序控制权交给原程序代码了，OEP 也即将到达，此时在代码段下访问断点，断下后就在 OEP 或者 OEP 附近了。在这里其实最好是在 VM_Retn 出口下断，VMP 退出虚拟机后就要将 eip 转给原程序了，在 OEP 被 VM 的情况下，如果在 VM_Retn 出口处进行 OEP 的修补，工作量会小很多。

5. IAT (API) 修复

VMP 对 API 进行了混淆，它将所有 IAT 相关的 API 调用都转变成为 pushx;call 或者 call;xxx 的形式，这对修复造成了不小的影响，也对程序的理解造成了困难，下面是某个被混淆的 API 调用指令流程已经相关 esp 信息：

```
OoWoodOne Log
Address      Info                                     Note
-           Print Chain: 010073AD Unoptimize
010073AD     call 01074896                 Def:[00000000],
Ref:[00000000], Sk:[-4], EspDef:[-4], EspRef:[0], EspRRef:[0]
01074896     pushfd                          Def:[00030000],
Ref:[00030000], Sk:[-4], EspDef:[-8], EspRef:[0], EspRRef:[0]
01074897     mov bp,0xE81E                       Def:[00100000],
Ref:[00000000], Sk:[0], EspDef:[0], EspRef:[0], EspRRef:[0]
0107489B     not edi                             Def:[30000000],
Ref:[30000000], Sk:[0], EspDef:[0], EspRef:[0], EspRRef:[0]
0107489D     not di                               Def:[10000000],
Ref:[10000000], Sk:[0], EspDef:[0], EspRef:[0], EspRRef:[0]
010748A0     nop                                  Def:[00000000],
Ref:[00000000], Sk:[0], EspDef:[0], EspRef:[0], EspRRef:[0]
010748A1     not di                               Def:[10000000],
Ref:[10000000], Sk:[0], EspDef:[0], EspRef:[0], EspRRef:[0]
010748A4     bswap di                             Def:[10000000],
Ref:[10000000], Sk:[0], EspDef:[0], EspRef:[0], EspRRef:[0]
010748A7     movsx di,al                          Def:[10000000],
Ref:[00000001], Sk:[0], EspDef:[0], EspRef:[0], EspRRef:[0]
010748AB     xchg edi,ebp                          Def:[30000000],
Ref:[00300000], Sk:[0], EspDef:[0], EspRef:[0], EspRRef:[0]
010748AD     mov ebp,dword ptr ss:[esp+0x4]        Def:[00300000],
Ref:[00030000], Sk:[0], EspDef:[0], EspRef:[-4], EspRRef:[0]
```

010748B1	movzx di,cl	Def:[10000000],
Ref:[00000010],	Sk:[0], EspDef:[0], EspRef:[0], EspRRef:[0]	
010748B5	xchg dword ptr ss:[esp+0x8],ebp	Def:[00000000],
Ref:[00330000],	Sk:[0], EspDef:[0], EspRef:[0], EspRRef:[0]	
010748B9	push dword ptr ss:[esp+0x4]	Def:[00030000],
Ref:[00030000],	Sk:[-4], EspDef:[-12], EspRef:[-4], EspRRef:[0]	
010748BD	xchg dword ptr ss:[esp+0x8],ebp	Def:[00000000],
Ref:[00330000],	Sk:[0], EspDef:[-4], EspRef:[0], EspRRef:[0]	
010748C1	bswap ebp	Def:[00300000],
Ref:[00300000],	Sk:[0], EspDef:[0], EspRef:[0], EspRRef:[0]	
010748C3	xchg bp,di	Def:[00100000],
Ref:[10000000],	Sk:[0], EspDef:[0], EspRef:[0], EspRRef:[0]	
010748C6	movzx bp,al	Def:[00100000],
Ref:[00000001],	Sk:[0], EspDef:[0], EspRef:[0], EspRRef:[0]	
010748CA	pushad	Def:[00000000],
Ref:[33337777],	Sk:[-32], EspDef:[-44], EspRef:[0], EspRRef:[0]	
010748CB	mov ebp,0x1005480	Def:[00300000],
Ref:[00000000],	Sk:[0], EspDef:[0], EspRef:[0], EspRRef:[0]	
010748D0	call 01016AB0	Def:[00000000],
Ref:[00000000],	Sk:[-4], EspDef:[-48], EspRef:[0], EspRRef:[0]	
01016AB0	mov ebp,dword ptr ss:[ebp+0x37FBE]	Def:[00300000],
Ref:[00300000],	Sk:[0], EspDef:[0], EspRef:[0], EspRRef:[0]	
01016AB6	pushfd	Def:[00030000],
Ref:[00030000],	Sk:[-4], EspDef:[-52], EspRef:[0], EspRRef:[0]	
01016AB7	lea ebp,dword ptr ss:[ebp+0x19534205]	Def:[00300000],
Ref:[00300000],	Sk:[0], EspDef:[0], EspRef:[0], EspRRef:[0]	
01016ABD	movsx di,al	Def:[10000000],
Ref:[00000001],	Sk:[0], EspDef:[0], EspRef:[0], EspRRef:[0]	
01016AC1	bswap di	Def:[10000000],
Ref:[10000000],	Sk:[0], EspDef:[0], EspRef:[0], EspRRef:[0]	
01016AC4	mov edi,ebp	Def:[30000000],
Ref:[00300000],	Sk:[0], EspDef:[0], EspRef:[0], EspRRef:[0]	
01016AC6	push dword ptr ss:[esp+0x4]	Def:[00030000],
Ref:[00030000],	Sk:[-4], EspDef:[-56], EspRef:[-48], EspRRef:[0]	
01016ACA	mov ebp,dword ptr ss:[esp+0x34]	Def:[00300000],
Ref:[00030000],	Sk:[0], EspDef:[0], EspRef:[-4], EspRRef:[0]	
01016ACE	pushfd	Def:[00030000],
Ref:[00030000],	Sk:[-4], EspDef:[-60], EspRef:[0], EspRRef:[0]	
01016ACF	push 0x600A1D36	Def:[00030000],
Ref:[00030000],	Sk:[-4], EspDef:[-64], EspRef:[0], EspRRef:[0]	
01016AD4	push 0x4E94F530	Def:[00030000],
Ref:[00030000],	Sk:[-4], EspDef:[-68], EspRef:[0], EspRRef:[0]	
01016AD9	push dword ptr ss:[esp+0x44]	Def:[00030000],
Ref:[00030000],	Sk:[-4], EspDef:[-72], EspRef:[0], EspRRef:[0]	

```

01016ADD    retn 0x48                                Def:[00030000],
Ref:[00030000], Sk:[76], EspDef:[0], EspRef:[0], EspRRef:[0]
                ESP Pos                                4

```

从中获取最主要的解密指令:

```

mov ebp,0x1005480
mov ebp,dword ptr ss:[ebp+0x37FBE]
lea ebp,dword ptr ss:[ebp+0x19534205]
mov edi,ebp

```

观察几个 API 的混淆可以发现,所有的 API 就是参照如下规则解密的:

```

Mov reg,const0
Mov reg,[reg+const1]
Lea reg,[reg+const2]
Mov reg0,reg//如果有这条指令说明是 mov reg,[apiaddr]类型

```

所以要获取 API 地址只要获取 const0、const1、const2 这 3 个常数就行了。另外光知道 API 地址还不行,还需要知道它的调用类型。这里需要观察指令链最后的 ESP 位置,以及最后入栈的指令所引用的 esp 地址。

在上述指令链中:

```

01016AD9    push dword ptr ss:[esp+0x44]                Def:[00030000],
Ref:[00030000], Sk:[-4], EspDef:[-72], EspRef:[0], EspRRef:[0]

```

最后入栈的指令所引用的 esp 地址为 0,最后 Esp pos 为 4。经过观察分析,调用类型一般可分为以下几类:

EspPos	EspRef	HaveReg1	type
4	-4		x;jmp[]
0	-8		jmp[];x
0	-4	yes	mov[];x
0	-4	no	x;call[]
-4	-8	yes	pop;mov[]
-4	-8	no	call[];x
4	0		push;mov[]

按照这样大部分的 api 可以完全修复,最后不能修复的几个是与资源相关的 api,只要资源 anti-dump 修复正确,这几个 api 可以不去管。

6. 脱壳总结

VMProtect 保护的脱壳修复的繁琐是免不了的,对于 anti-dump 可以采用补区段的方法,为了 dump 区段的位置紧凑,可以在系统断点时就在模块后面申请相关内存区段,然后再脱

壳是 patch 一些 anti-dump 位置到该区段。

Api 的修复可能不仅仅只对代码段，在壳区段中可是需要修复一些 api 的，这暂时还没找到合理的解决方法。

五. VMProtect 代码还原尝试

1. 代码静态识别

在 VM 代码识过程中，需要注意的是 3 个寄存器的使用：ESI 作为 VM_Eip 也就是 Opcode 的地址，它在每次虚拟调度的过程中都会改变，在一些虚拟指令包含的数据也是 Opcode 的一部分；EAX 是 Opcode 或者 Opcode 数据读取以及解密时使用的相关寄存器；EBX 则是全程参与解密的寄存器。下面是某虚拟机入口的 Initkey 解密 VM_Eip 的过程：

0116273A	mov esi,dword ptr ss:[esp+0x5C]	Initkey Load
0116274A	dec esi	InitKey Decode
01162750	rol esi,0x1F	InitKey Decode
01161CAF	xor esi,0x739F6DE	InitKey Decode
01161CBF	lea ebp,dword ptr ss:[esp+0x30]	Reloc Load
01161CE7	mov ebx,esi	Join Reg Load
01161CEE	add esi,dword ptr ss:[ebp]	Reloc Add

虚拟机从 InitKey 解密得到 VM_EIP，将没加重定位前的 esi 做我 EBX 的初始值。Opcode 的解密流程如下：

01163059	mov al,byte ptr ds:[esi-0x1]	VMP Opcode Load
01162CA5	xor al,bl	Opcode Decode
01162CB6	neg al	Opcode Decode
01162CC2	xor al,0xA	Opcode Decode
01162CC9	inc al	Opcode Decode
01162CD5	xor bl,al	Join Reg Do
01162CD9	movzx eax,al	Opcode Decode

数据的导入从 Opcode 读取进行解密，其中也有 EBX 的参与，下面是某 VM_PushI32 的数据导入解密过程：

01162C17	mov eax,dword ptr ds:[esi-0x4]	VM_PushI32 Data Load
01162C1F	bswap eax	Data Decode
01162C27	sub eax,ebx	Data Decode
01162C34	sub eax,0x1DD7F782	Data Decode
01162C3F	neg eax	Data Decode
01162C43	ror eax,0x17	Data Decode
01161E40	sub ebx,eax	Join Reg Do

对于之前版本的 VM_PushR32 与 VM_PopR32 寄存器偏移的解密不是从 Opcode 数据中获取的，而是直接从该 Opcode 解密得到：

005C66DC	shl dh,cl	VM_PopR32 Start
005C66DE	not al	Data Decode
005C66E0	movsx dx,dl	
005C66E4	rcr dl,0x2	
005C66E7	neg al	Data Decode
005C66E9	bts dx,bp	

005C66ED	lea edx,dword ptr ds:[ebx*8-0x63E720B6]	
005C66F4	not al	Data Decode
005C66F6	bswap edx	
005C66F8	add al,0xFE	Data Decode
005C66FA	add dl,ch	
005C66FC	sub dh,0xE5	
005C66FF	ror dh,cl	
005C6701	xor dl,0x10	
005C6704	and al,0x3C	Data Decode
005C6707	movsx dx,cl	
005C670B	mov edx,dword ptr ss:[ebp]	Get Context
005C670E	bt di,0x5	
005C6713	pushad	
005C6714	pushad	
005C6715	add ebp,0x4	
005C6718	jmp 005C6F82	
005C6F82	pushfd	
005C6F83	mov dword ptr ds:[edi+eax],edx	Set Reg Value
005C6F86	push esi	
005C6F87	mov dword ptr ss:[esp],edx	
005C6F8A	push 0xBCA19189	
005C6F8F	lea esp,dword ptr ss:[esp+0x4C]	
005C6F93	jmp 005C655B	
005C655B	btr cx,sp	

静态解密引擎的主要实现就是对这些解密指令的提取以及模拟执行。提取这些解密指令的要点就是识别对 ESI、EAX、EBX 这 3 个寄存器的定义以及引用。然后再引擎中实现一个虚拟机循环，对 Opcode 和相应数据解密并静态输出，直到遇到 VM_Jmp 或者 VM_Retn 退出。VMP 中解密指令固定涉及这几个寄存器并且不会涉及到堆栈的操作，因此在模拟执行的过程中可以直接注入这些指令到当前的程序来执行解密。

静态解密到 VM_Jmp 后将没法确切获取将跳转的 VM_Eip 地址，因此需要程序执行到 VM_Jmp 再重新进行下一步解密，这进一步的解密需要获取关键的 ESI 和 EBX 的确切值才能继续，要获取完整的虚拟机执行代码还是需要结合动态的执行。

2. 寄存器轮转

虚拟机入口在调度过程中，首先会将入口入栈的 Context 数据 pop 到相应的 VM 寄存器中，隐藏真实的 context 含义，很好的实现了混淆的目的。一下是某个虚拟机起初调度的过程：

-	VMP Context:	
00 (+00)	RELOC	00E50000
01 (+04)	ANTIDUMP	00000000
02 (+08)	EDI	00000000
03 (+0C)	EDX	01261159

04 (+10)	ESI	00000000
05 (+14)	ESP	0016FC34
06 (+18)	EBX	7EFDE000
07 (+1C)	EBP	0016FC3C
08 (+20)	ECX	00000000
09 (+24)	EAX	75A93358
10 (+28)	EFL	00000246
11 (+2C)	RETADDR	6B92ECE9
12 (+30)	INITKEY	1DF2DDB2

-->

VM_EIP	VM_ESP	Handler	Value	
0125CBB4	4	VM_PopR32	R1	RELOC
0125CBB2	0	VM_PushI32	C62B3BEB	
0125CBAD	0	VM_Add32		
0125CBAC	4	VM_PopR32	R2	
0125CBAA	8	VM_PopR32	R5	ANTIDUMP
0125CBA8	12	VM_PopR32	R3	EDI
0125CBA6	16	VM_PopR32	R8	EDX
0125CBA4	20	VM_PopR32	R2	ESI
0125CBA2	24	VM_PopR32	R12	ESP
0125CBA0	28	VM_PopR32	R0	EBX
0125CB9E	32	VM_PopR32	R14	EBP
0125CB9C	36	VM_PopR32	R11	ECX
0125CB9A	40	VM_PopR32	R6	EAX
0125CB98	44	VM_PopR32	R13	EFL
0125CB96	48	VM_PopR32	R15	RETADDR
0125CB94	52	VM_PopR32	R4	INITKEY
.....				

在 VM_Retn 中，又会将隐藏的真实数据赋值到真实的 Context 中：

-	VMP Retn Context:	
00 (+00)		-
01 (+04)		-
02 (+08)		EDI
03 (+0C)		EDX
04 (+10)		ESI
05 (+14)		-
06 (+18)		EBX
07 (+1C)		EBP

08 (+20)	ECX
09 (+24)	EAX
10 (+28)	EFL

—>

VM_EIP	VM_ESP	Handler	Value	
.....				
0125CBCB	40	VM_PushR32	R7	EFL
0125CBC9	36	VM_PushR32	R8	EAX
0125CBC7	32	VM_PushR32	R6	ECX
0125CBC5	28	VM_PushR32	R11	EBP
0125CBC3	24	VM_PushR32	R12	EBX
0125CBC1	20	VM_PushR32	R15	
0125CBBF	16	VM_PushR32	R1	ESI
0125CBBD	12	VM_PushR32	R10	EDX
0125CBBB	8	VM_PushR32	R4	EDI
0125CBB9	4	VM_PushR32	R5	
0125CBB7	0	VM_PushR32	R6	
0125CBB5	0	VM_Retn		

真实寄存器的虚拟机寄存器中的位置是没法预测的，即使知道虚拟机入口出口的 context 数据出入。在虚拟机中还会遇到寄存器的轮转，比如在遇到 VM_Jmp 时的情况：

VM_EIP	VM_ESP	Handler	Value	
.....				
0125CB3E	44	VM_PushR32	R14	寄存器轮转入栈
0125CB3C	40	VM_PushR32	R12	
0125CB3A	36	VM_PushR32	R0	
0125CB38	32	VM_PushR32	R6	
0125CB36	28	VM_PushR32	R2	
0125CB34	24	VM_PushR32	R7	
0125CB32	20	VM_PushR32	R14	
0125CB30	16	VM_PushR32	R11	
0125CB2E	12	VM_PushR32	R3	
0125CB2C	8	VM_PushR32	R13	
0125CB2A	4	VM_PushR32	R8	
0125CB28	0	VM_PushI32	39D4C415	
0125CB23	-4	VM_PushR32	R5	
0125CB21	-4	VM_Add32		
0125CB20	0	VM_PopR32	R9	
0125CB1E	-4	VM_PushR32	R1	
0125CB1C	-8	VM_PushR32	R13	
0125CB1A	-4	VM_Jmp		
寄存器轮转出栈				

0125CC4E	0	VM_PopR32	R9	=R1
0125CC4C	-4	VM_PushI32	C62B3BEB	
0125CC47	-4	VM_Add32		
0125CC46	0	VM_PopR32	R8	
0125CC44	4	VM_PopR32	R5	=R5 (ANTIDUMP)
0125CC42	8	VM_PopR32	R10	=R8
0125CC40	12	VM_PopR32	R13	=R13
0125CC3E	8	VM_PushR32	R13	
0125CC3C	4	VM_PushR32	R13	
0125CC3A	4	VM_Nor32		Nor (R13, R13)
0125CC39	8	VM_PopR32	R12	
0125CC37	4	VM_PushI32	BFD6FBD2	
0125CC32	4	VM_Nor32		
0125CC31	8	VM_PopR32	R15	
0125CC2F	4	VM_PushI32	4029042D	
0125CC2A	0	VM_PushR32	R13	
0125CC28	0	VM_Nor32		
0125CC27	4	VM_PopR32	R7	
0125CC25	4	VM_Nor32		
0125CC24	8	VM_PopR32	R3	
0125CC22	12	VM_PopR32	R15	
0125CC20	16	VM_PopR32	R4	=R3
0125CC1E	20	VM_PopR32	R6	=R11
0125CC1C	24	VM_PopR32	R11	=R14
0125CC1A	28	VM_PopR32	R8	=R7
0125CC18	32	VM_PopR32	R1	=R2
0125CC16	36	VM_PopR32	R3	=R6
0125CC14	40	VM_PopR32	R12	=R0
0125CC12	44	VM_PopR32	R7	=R12
0125CC10	48	VM_PopR32	R14	=R14
0125CC0E	52	VM_PopR32	R2	=EFL

经过寄存器轮转后寄存器所代表的含义进一步被打乱。

3. 分析尝试

以下是对一个简单代码（就加密了一句汇编）虚拟机加密的分析：

VM_EIP	VM_ESP	Handler	Value	
0125CBB4	4	VM_PopR32	R1	RELOC
0125CBB2	0	VM_PushI32	C62B3BEB	
0125CBAD	0	VM_Add32		
0125CBAC	4	VM_PopR32	R2	
0125CBAA	8	VM_PopR32	R5	ANTIDUMP

	0125CBA8	12	VM_PopR32	R3	EDI
	0125CBA6	16	VM_PopR32	R8	EDX
	0125CBA4	20	VM_PopR32	R2	ESI
	0125CBA2	24	VM_PopR32	R12	ESP
	0125CBA0	28	VM_PopR32	R0	EBX
	0125CB9E	32	VM_PopR32	R14	EBP
	0125CB9C	36	VM_PopR32	R11	ECX
	0125CB9A	40	VM_PopR32	R6	EAX
	0125CB98	44	VM_PopR32	R13	EFL
	0125CB96	48	VM_PopR32	R15	RETADDR
	0125CB94	52	VM_PopR32	R4	INITKEY
	0125CB92	48	VM_PushR32	R13	
据 试 试 据	0125CB90	44	VM_PushI32	4068CEC3	真实 Jmp 地址数
	0125CB8B	40	VM_PushI32	4068C862	检测到有单步调
					试 Jmp 地址数据
	0125CB86	36	VM_PushEsp		Push(ESP{40})
	0125CB85	32	VM_PushI16To32	FEFF	
	0125CB82	28	VM_PushR32	R13	
	0125CB80	24	VM_PushR32	R13	
	0125CB7E	24	VM_Nor32		Nor(EFL, EFL)
	0125CB7D	28	VM_PopR32	R9	
	0125CB7B	28	VM_Nor32		
	Nor(Nor(EFL, EFL), FEFF) 检测 TF 标志位				
	0125CB7A	32	VM_PopR32	R7	
	R7=EFlag(Nor(Nor(EFL, EFL), FEFF))				
	0125CB78	36	VM_PopR32	R9	
	R9=Nor(Nor(EFL, EFL), FEFF)				
	0125CB76	34	VM_PushI8To16	4	
	0125CB74	30	VM_PushI8To32	BF	
	0125CB72	26	VM_PushR32	R7	
	0125CB70	26	VM_Nor32		Nor(R7, BF)
	检测 R7 的 ZF 标志位, 如果 EFL 的 TF 标志位为 0, R7 的 ZF 标志位为 1				
	0125CB6F	30	VM_PopR32	R4	
	0125CB6D	28	VM_Shr32		
	Shr(Nor(R7, BF), 4) 获取 ZF*4 值				
	0125CB6C	32	VM_PopR32	R4	
	0125CB6A	32	VM_Add32		
	Add(ESP{40}, Shr(Nor(R7, BF), 4)) ESP{40}+ZF*4 (jmp 地址数据)				

0125CB69	36	VM_PopR32	R15	
0125CB67	36	VM_ReadSs32		
0125CB66	40	VM_PopR32	R13	R13=获取 Jmp 地
址数据				
0125CB64	44	VM_PopR32	R10	
0125CB62	48	VM_PopR32	R15	
0125CB60	44	VM_PushR32	R13	
0125CB5E	48	VM_PopR32	R15	R15=R13
0125CB5C	44	VM_PushR32	R15	
0125CB5A	40	VM_PushR32	R15	
0125CB58	40	VM_Nor32		Nor (R15, R15)
0125CB57	44	VM_PopR32	R4	
0125CB55	40	VM_PushI32	BFD6FBD2	
0125CB50	40	VM_Nor32		
Nor (Nor (R15, R15), BFD6FBD2)				
0125CB4F	44	VM_PopR32	R10	
0125CB4D	40	VM_PushR32	R15	
0125CB4B	36	VM_PushI32	4029042D	
0125CB46	36	VM_Nor32		
Nor (R15, 4029042D)				
0125CB45	40	VM_PopR32	R13	
0125CB43	40	VM_Nor32		
Nor (Nor (Nor (R15, R15), BFD6FBD2), Nor (R15, 4029042D))				
0125CB42	44	VM_PopR32	R10	
0125CB40	48	VM_PopR32	R13	
R13=Nor (Nor (Nor (R15, R15), BFD6FBD2), Nor (R15, 4029042D)) 解密真实 Jmp VM_Eip				

//寄存器轮转入栈

0125CB3E	44	VM_PushR32	R14
0125CB3C	40	VM_PushR32	R12
0125CB3A	36	VM_PushR32	R0
0125CB38	32	VM_PushR32	R6
0125CB36	28	VM_PushR32	R2
0125CB34	24	VM_PushR32	R7
0125CB32	20	VM_PushR32	R14
0125CB30	16	VM_PushR32	R11
0125CB2E	12	VM_PushR32	R3
0125CB2C	8	VM_PushR32	R13
0125CB2A	4	VM_PushR32	R8
0125CB28	0	VM_PushI32	39D4C415
0125CB23	-4	VM_PushR32	R5
0125CB21	-4	VM_Add32	
0125CB20	0	VM_PopR32	R9
0125CB1E	-4	VM_PushR32	R1

0125CB1C	-8	VM_PushR32	R13	Push_Jmp VM_Eip
0125CB1A	-4	VM_Jmp		
0125CC4E	0	VM_PopR32	R9	=R1
0125CC4C	-4	VM_PushI32	C62B3BEB	
0125CC47	-4	VM_Add32		
0125CC46	0	VM_PopR32	R8	
0125CC44	4	VM_PopR32	R5	=R5 (ANTIDUMP)
0125CC42	8	VM_PopR32	R10	=R8
0125CC40	12	VM_PopR32	R13	=R13
0125CC3E	8	VM_PushR32	R13	
0125CC3C	4	VM_PushR32	R13	
0125CC3A	4	VM_Nor32		Nor (R13, R13)
0125CC39	8	VM_PopR32	R12	
0125CC37	4	VM_PushI32	BFD6FBD2	
0125CC32	4	VM_Nor32		
Nor (Nor (R13, R13), BFD6FBD2)				
0125CC31	8	VM_PopR32	R15	
0125CC2F	4	VM_PushI32	4029042D	
0125CC2A	0	VM_PushR32	R13	
0125CC28	0	VM_Nor32		
Nor (R13, 4029042D)				
0125CC27	4	VM_PopR32	R7	
0125CC25	4	VM_Nor32		
0125CC24	8	VM_PopR32	R3	
0125CC22	12	VM_PopR32	R15	
R15=Nor (Nor (R13, 4029042D), Nor (Nor (R13, R13), BFD6FBD2))				
0125CC20	16	VM_PopR32	R4	=R3
0125CC1E	20	VM_PopR32	R6	=R11
0125CC1C	24	VM_PopR32	R11	=R14
0125CC1A	28	VM_PopR32	R8	=R7
0125CC18	32	VM_PopR32	R1	=R2
0125CC16	36	VM_PopR32	R3	=R6
0125CC14	40	VM_PopR32	R12	=R0
0125CC12	44	VM_PopR32	R7	=R12
0125CC10	48	VM_PopR32	R14	=R14
0125CC0E	52	VM_PopR32	R2	=EFL
0125CC0C	48	VM_PushR32	R2	
0125CC0A	44	VM_PushR32	R2	
0125CC08	44	VM_Nor32		Nor (EFL, EFL)
0125CC07	48	VM_PopR32	R14	
0125CC05	44	VM_PushI16To32	8FF	

0125CC02	44	VM_Nor32		
Nor (Nor (EFL, EFL), 8FF)				
0125CC01	48	VM_PopR32	R8	
0125CBFF	52	VM_Popfd		
EFL=Nor (~EFL, 8FF) DF, IF, TF 取反, 其他位清零				
0125CBFE	48	VM_PushI8To32	4	
0125CBFC	44	VM_PushI8To32	8	
0125CBFA	40	VM_PushEsp		
0125CBF9	40	VM_Add32		
0125CBF8	44	VM_PopR32	R7	
0125CBF6	44	VM_Add32		
0125CBF5	48	VM_PopR32	R7	
0125CBF3	??	VM_PopEsp		
//push retnAddr data				
0125CBF2	44	VM_PushI32	41151C	
0125CBED	40	VM_PushR32	R9	Reloc
//真实虚拟代码				
0125CBEB	36	VM_PushR32	R3	
0125CBE9	32	VM_PushR32	R3	
0125CBE7	32	VM_Nor32		Nor (R3, R3)
0125CBE6	36	VM_PopR32	R8	
0125CBE4	32	VM_PushR32	R3	
0125CBE2	28	VM_PushEsp		
0125CBE1	28	VM_ReadSs32		
0125CBE0	28	VM_Nor32		Nor (R3, R3)
0125CBDF	32	VM_PopR32	R0	
0125CBDD	32	VM_Nor32		
Nor (Nor (R3, R3), Nor (R3, R3))				
0125CBDC	36	VM_PopR32	R0	
0125CBDA	32	VM_PushR32	R3	
0125CBD8	28	VM_PushEsp		
0125CBD7	28	VM_ReadSs32		
0125CBD6	28	VM_Nor32		Nor (R3, R3)
0125CBD5	32	VM_PopR32	R0	
0125CBD3	32	VM_Nor32		
Nor (Nor (Nor (R3, R3), Nor (R3, R3)), Nor (R3, R3))				
0125CBD2	36	VM_PopR32	R7	
R7=Eflag (Nor (Nor (Nor (R3, R3), Nor (R3, R3)), Nor (R3, R3)))=Eflag (Xor (R3, R3))				
0125CBD0	40	VM_PopR32	R8	
R8=Nor (Nor (Nor (R3, R3), Nor (R3, R3)), Nor (R3, R3))=Xor (R3, R3)				
0125CBCE	40	VM_Add32		

0125CBCD	44	VM_PopR32	R3	
retAddr=Add(41151C+R9)				
//退出虚拟机				
0125CBCB	40	VM_PushR32	R7	EFL
0125CBC9	36	VM_PushR32	R8	EAX
0125CBC7	32	VM_PushR32	R6	ECX
0125CBC5	28	VM_PushR32	R11	EBP
0125CBC3	24	VM_PushR32	R12	EBX
0125CBC1	20	VM_PushR32	R15	
0125CBBF	16	VM_PushR32	R1	ESI
0125CBBD	12	VM_PushR32	R10	EDX
0125CBBB	8	VM_PushR32	R4	EDI
0125CBB9	4	VM_PushR32	R5	
0125CBB7	0	VM_PushR32	R6	
0125CBB5	0	VM_Retn		

该虚拟机简单的过程描述:

- 1) Pop Context 到相应 VM 寄存器
- 2) Push 正确跳转 VM_Eip 和检测到单步调试后的 VM_Eip 地址数据
- 3) 检测 EFL 中的 TF 标志位
- 4) 根据检测结果从堆栈读取跳转数据
- 5) 解密跳转数据
- 6) 寄存器轮转入栈
- 7) VM_Jmp
- 8) 寄存器轮转出栈
- 9) Push VM_Retn address 数据
- 10) 真实代码执行, 这里是 Xor eax, eax
- 11) 真实寄存器数据入栈
- 12) VM_Retn