第 16 章　脱壳技术

16.9.2　Themida 的 SDK 分析[①]

Themida 是一款优秀的保护壳，在放弃使用驱动反调试后，其强度主要靠 SDK 的虚拟机技术来保证。SDK 之外的脱壳过程，和其他的壳没有太大区别。下面主要讨论 SDK 保护代码的修复。

本书配套文件中提供的演示程序 VMTest，使用了 SDK 的 Encode、Clear、CodeReplace 和 VM 宏，用 Themida 1.910 加壳，关闭 Protection Options 中的全部所有保护选项，将虚拟机保护水平设置为最低，处理器类型设置为选择 CISC-2，如图 16.1 所示。

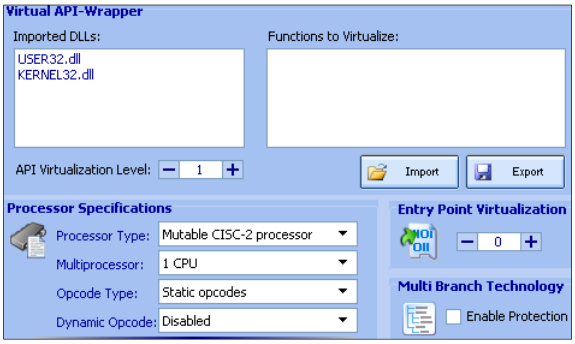

图 16.1　范例的 VM 加壳设置

1．Encode 与 Clear 宏保护代码的修复

下面是使用 ENCODE_START/END 宏保护的原始代码如下。

```
.text:00401001           jmp    short loc_401013
.text:00401003           dd 20204C57h, 4, 0, 20204C57h
.text:00401013 loc_401013:
.text:00401013           mov    esi, ds:MessageBoxA
.text:00401019           push   0                    ;uType
.text:0040101B           push   offset Caption       ;"Debug1"
.text:00401020           push   offset Text          ;"这段代码用 ENCODE 宏保护!"
.text:00401025           push   0                    ;hWnd
.text:00401027           call   esi                  ;MessageBoxA
.text:00401029           jmp    short loc_40103B
.text:0040102B           dd 20204C57h, 5, 0, 20204C57h
```

加壳后的代码如下。为：

```
00401001   E8 61970B00   call   004BA767
00401006   0000          add    byte ptr [eax], al
00401008   0000          add    byte ptr [eax], al
0040100A   0000          add    byte ptr [eax], al
0040100C   0000          add    byte ptr [eax], al
0040100E   1E            push   ds
0040100F   0000          add    byte ptr [eax], al
00401011   0020          add    byte ptr [eax], ah
00401013   71 55         jno    short 0040106A          ;在这里下断点
```

00401001 处变成了 call 指令。Encode 宏保护实际上保护的是 SMC，即在执行时会解出

明文代码。ENCODE_START 占 18 字节，在第 19 字节，即原始代码处（401013），下硬件执行断点，具体如下。

```
00401001   E8 61970B00   call   004BA767
00401006   0000          add    byte ptr [eax], al
00401008   0000          add    byte ptr [eax], al
0040100A   0000          add    byte ptr [eax], al
0040100C   0000          add    byte ptr [eax], al
0040100E   1E            push   ds
0040100F   0000          add    byte ptr [eax], al
00401011   0020          add    byte ptr [eax], ah
00401013   8B35 9C504000 mov    esi, dword ptr [40509C]  ;USER32.MessageBoxA
00401019   6A 00         push   0
0040101B   68 AC604000   push   004060AC                ;ASCII "Debug1"
00401020   68 94604000   push   00406094
00401025   6A 00         push   0
00401027   FFD6          call   esi
00401029   E8 62CF0A00   call   004ADF90                ;重新加密代码
0040102E   0000          add    byte ptr [eax], al
```

不仅解出了明文代码，在原 ENCODE_END 处的位置（00401029）还有一个 call 指令，执行后会将解出的代码重新加密。此时，只要保存解码结果，用 nop 指令替换掉 ENCODE_START/END 宏所对应的 36 字节即可。

Clear 宏与 Encode 相似，唯一的区别是 CLEAR_END 为 23 字节，其作用为擦除解出的代码，而不是重新加密，示例如下。

```
0040105D   60            pushad
0040105E   E8 00000000   call   00401063
00401063   5F            pop    edi
00401064   81EF 23000000 sub    edi, 23
0040106A   B9 23000000   mov    ecx, 23
0040106F   33C0          xor    eax, eax
00401071   F3:AA         rep    stos byte ptr es:[edi]
00401073   61            popad
```

可以在进行 Dump 之前通过编写 OllyDbg 脚本来修复这两种宏保护的代码。

2．CodeReplace 与 VM 宏保护代码的修复

对于当前的 Themida 当前版本，这两个宏是相同的，都使用虚拟机技术。加壳时将原始机器码反汇编，转换为伪码，执行时由虚拟机引擎解释执行。如果想还原代码，需要分析 VM 解释引擎的工作方式，编写 pcode 解码器。Themida 目前支持 4 种 VM 处理器类型，分为 CISC 和 RISC 两类，后者提供的保护强度更高，也更复杂。

出于自我保护的目的，VM 解释引擎是被混淆过的变形代码。原始代码按预先定义的模式膨胀，生成的结果被划分为若干代码块，随机置换各代码块的物理位置，再用 jmp 指令将其链接起来，示例如下面的例子。

```
01596EEF   push   edx
01596EF0   add    ebx, ecx
01596EF2   mov    edx, 0
01596EF7   mov    bh, 35h
01596EF9   add    edx, edi
01596EFB   shr    al, 3
01596EFE   push   ecx
01596EFF   sub    eax, ebx
```

```
01596F01   jmp    loc_14F562A
014F562A   mov    ecx, 7445h
014F562F   neg    ecx
014F5631   not    ecx
014F5633   not    ecx
014F5635   or     ecx, 629Ah
014F563B   jmp    loc_14FA94D
014FA94D   sub    al, dl
014FA94F   or     ecx, 205Ch
014FA955   xor    ecx, 0FFFFEBFFh
014FA95B   mov    bl, 8Ah
014FA95D   add    ecx, edx
014FA95F   add    ah, dh
014FA961   jmp    loc_1594A3C
01594A3C   mov    esi, [ecx]
01594A3E   mov    bl, ch
01594A40   pop    ecx
01594A41   mov    ebx, 7570h
01594A46   mov    edx, [esp]
01594A49   add    esp, 4
```

这段代码在逻辑上是连续的，在物理上被分成了 4 块，用 3 个 jmp 指令链接起来。eax 和 ebx 为空闲寄存器，用来生成干扰指令。以粗体显示的是有用的代码。从 014F562A 行开始对 ecx 的连续变换结果为 0。去掉 jmp 指令和干扰代码，后为结果如下。：

```
01596EEF   push   edx
01596EF2   mov    edx, 0
01596EF9   add    edx, edi
01596EFE   push   ecx
014F562A   mov    ecx, 0
014FA95D   add    ecx, edx
01594A3C   mov    esi, [ecx]
01594A40   pop    ecx
01594A46   mov    edx, [esp]
01594A49   add    esp, 4
```

这实际上只等于 1 条指令，具体如下。：

```
01596EEF   mov   esi, [edi]
```

如果不对变形代码进行清理，就很难理解 handler 的真正目的。遗憾的是，笔者没有完美可靠的办法，目前的做法是根据具体的代码变形模式进行匹配压缩的，得到的结果容易出错，只能用来进行辅助分析，。希望有技术人员能开发出更好的方法。

下面分析一下范例中被 CodeReplace 宏保护的代码的执行过程。VM 宏与此类似，留给读者练习。在本书配套文件中有一个简单的解码程序。，被保护的原始代码如下。为：

```
.text:00401086    push   0
.text:00401088    push   offset aDebug3  ;"Debug3"
.text:0040108D        push   offset aTICodereplaceG ;"这段代码用 CODEREPLACE 宏保
护!"
.text:00401092    push   0
.text:00401094    call   esi
```

加壳后的代码，从以 jmp 指令开始执行 VM 保护代码，具体如下。

```
00401074   E9 F7FE0B00   jmp    004C0F70     ;CODEREPLACE_START 处为 jmp 指令
```

```
004C0F70   68 FC5C2607   push   7265CFC
004C0F75   E9 701FF5FF   jmp    00412EEA
```

压栈的 imm32 代表了 pcode 数据的地址，同时被用作作为 pcode 数据解码 key。下面是 VM 的入口代码。

```
00412EEA   pushad                          ;保存进入 VM 时的执行环境
00412EEB   pushfd
00412EEC   cld
00412EED   call   00412EF2
00412EF2   pop    edi
00412EF3   sub    edi, 71B7EDE             ;delta=F925B014
00412EF9   mov    eax, edi
00412EFB   add    edi, 71B7BF6             ;edi=412C0A,，指向 ctx（上下文结构）
00412F01   cmp    eax, dword ptr [edi+2C]  ;delta 与 ctx 内的值（初值为 0）
00412F04   jnz    short 00412F08           ;是否相等
00412F06   jmp    short 00412F1B
00412F08   mov    dword ptr [edi+2C], eax
00412F0B   mov    ecx, 0A7                 ;167 个 handler
00412F10   jmp    short 00412F17
00412F12   add   [edi+ecx*4+40], eax       ;用 delta 计算 handler 的实际地址
00412F16   dec    ecx                      ;的实际地址并填到地址表中，这是第 1 次
00412F17   or     ecx, ecx                 ;这是在第 1 次执行 VM 保护代码时完成的
00412F19   jnz    short 00412F12
00412F1B   mov    esi, dword ptr [esp+24]  ;进入 VM 前 push 的 imm32
00412F1F   mov    ebx, esi                 ;pcode 解码 key
00412F21   add    esi, eax                 ;pcode 地址
00412F23   mov    ecx, 1
00412F28   xor    eax, eax
00412F2A   lock   cmpxchg [edi+30], ecx    ;检测设置 busy 标记
00412F2F   jnz    short 00412F28           ;等待 VM 空闲
00412F31   lods   byte ptr [esi]           ;取指令
00412F32   sub    al, 67
00412F34   jmp    0041B147
```

执行初始化代码执行后，检测 VM 是否忙，忙则等待，（VM 不支持多线程访问）。如果 VM 空闲，开始取 pcode 解释执行。清理后的取指令代码如下。为：

```
l_FetchOpcode:
  lodsb                                ;取 opcode
  add      al, bl
  sub      al, 7
  xor      al, 82h                     ;解码 opcode
  add      bl, al                      ;用解码结果变换 key
  movzx    eax, al
  jmp     dword ptr [edi+eax*4]        ;跳到对应的 handler 处
```

handler 的最后一 1 条指令为 jmp，跳到 l_FetchOpcode 继续取指令/执行指令循环，直到遇到特定的 opcode（如 ExitVm）为止。

在调试代码前，我们来先了解一下 VM 的基本特征。最重要的是上下文结构 VMCtx，其中保存了原程序寄存器组及 VM 内部使用的变量，示例如下。其成员的含义要根据 handler 如何使用来确定。演示程序的 VMCtx 结构在 00412C0A 处。

```
00000000 VMCtx          struc       ;(sizeof=0x44)
00000000 edx            dd ?        ;原程序的寄存器组
00000004 ecx            dd ?
00000008 ebp            dd ?
```

```
0000000C eax             dd ?
00000010 esi             dd ?
00000014 edi             dd ?
00000018 ebx             dd ?
0000001C eflag           dd ?
00000020 jxxFlag         dd ?          ;是否执行控制转移的标记
00000024 counter         dd ?          ;模仿控制转移的 handler 时使用
00000028 indexOfEcx      dd ?          ;ecx 在 VMCtx 内的索引,，模仿 jcxz,和 jecxz 时用
0000002C delta           dd ?          ;VM 加载地址
00000030 busy            dd ?          ;VM 忙标记
00000034 field_34        dd ?
00000038 field_38        dd ?
0000003C relocDelta      dd ?          ;重定位 delta,，对 dll 加壳时使用
00000040 field_40        dd ?
00000044 VMCtx           ends
```

在 VMCtx 结构后面就是 handler 地址表，共 167 项。以下这里列出的数据已被替换为清理变形代码后的地址，注释中是原始的 handler 地址。

```
Themida_:00412C4E Opcode11       dd offset loc_4CF000     ;0041BE13
Themida_:00412C52 Opcode12       dd offset loc_4CF011     ;00413AF4
Themida_:00412C56 Opcode13       dd offset loc_4CF021     ;0041A819
Themida_:00412C5A Opcode14       dd offset loc_4CF03F     ;0041D14E
```

Themida CISC-2 处理器的指令长度为 1 字节，每条指令可以带有 0/1/2/4 字节的操作数。是否带有操作数，可以从相应的 handler 代码中看出来，。若有操作数，也需要解码。VMCtx 结构成员及 handler 地址表用 1 个字节作为索引即可寻址，opcode 编码直接从 0x11 开始。

进入 VM 后，有 3 个寄存器有特殊含义，在 handler 执行过程中保持不变，具体如下。：

```
ebx：    ->  解码 key。
esi：    ->  指向 pcode 数据。
edi：    ->  指向 VMCtx。
```

handler 的实现使用了 3 个寄存器，分别是 eax、/ecx、/edx，但未使用 ebp。可以认为这 3 个寄存器是 VM 内部的寄存器。为避免引起混淆，在解码时将这 3 个寄存器另外命名，具体如下。

```
将 eax 寄存器命名为 ->  “R0”。
将 ecx 寄存器命名为 ->  “R1”。
将 edx 寄存器命名为 ->  “R2”。
```

pcode 与被保护的原始机器码并非是一一对应关系，一条机器指令一般需要执行被几条 pcode 才能模仿，如果再加上 pcode 变形，则代码数量更多，对性能的损耗会相当大。另外，有少量 opcode 由仅在 VM 内部使用（如用于实现 pcode 变形），并不用于模仿原代码。

```
PUSH32     00000000
PUSH32     112ADEE4
POP32      R2
ADD32      R2,ctx.relocDelta      ；注意这里对重定位的处理
POP32      [R2]
```

这 5 行 pcode 代码模仿了原来的一条指令“mov ds:[112ADEE4], 0”。

Themida VM 提供的是“平面”式的保护，即被虚拟机保护的代码，如果其中有调用其他函数（或 API）的代码，则被调用的代码不会被纳入保护。当执行到 call 指令时会退出 VM，调用完成后重新进入 VM，这样，原始代码对应的 pcode 数据被明显地分为几段。如果另一种情况是被保护代码中含有 VM 不支持的指令（如浮点指令），也会到 VM 以外来执行。

下面是范例中被 CodeReplace 保护代码的 pcode 数据，被一个 call 指令分为两部分。pcode 数据的地址为进入 VM 时 push 的 imm32 的值加 delta 的值。

```
Themida_:004C0D06    push    7265E67h          ;第 2 个 push imm32
Themida_:004C0D0B    jmp     l_Vm_Entry        ;下面为 pcode 数据
Themida_:004C0D10    dd 35C218E2h, 0B5429460h, 36C313E0h, 0B9469261h, 38C519E4h
Themida_:004C0D10    dd 16E12A8Fh, 8F6035C2h, 15E5BA47h, 9B693ECBh, 9A05C04Dh
......
Themida_:004C0D10    dd 648970BAh, 3F644B93h, 14392074h, 0E90EF549h, 0C2E7CE1Ah
Themida_:004C0D10    dd 99BEA5F5h, 8F947BCDh
Themida_:004C0F70
Themida_:004C0F70 loc_4C0F70:                          ;  CODE XREF: start+74j
Themida_:004C0F70    push    7265CFCh          ;第 1 个 push imm32
Themida_:004C0F75    jmp     l_Vm_Entry
```

第 1 段 pcode 的数据地址= 7265CFC + F925B014 = 004C0D10

第 2 段 pcode 的数据地址= 7265E67 + F925B014 = 004C0E7B

在实际应用情况中可能并不容易迅速看出 pcode 数据究竟由几段组成，但在对 pcode 解码的过程中，这些信息最终就会显示出来。

在实际情况中可能并不容易迅速看出 pcode 数据究竟由几段组成，但在对 pcode 解码的过程中最终会显示出来。

下面分析范例中被 CodeReplace 保护代码的 pcode 解码结果。前 4 列分别为序号，pcode 数据地址，解码 key 和 opcode。addr_ctx.eax 表示 VMCtx 结构内 eax 成员的地址，ctx.eax 指 VMCtx 内 eax 成员的值。

```
00000   004C0D10  07265CFC  55  PUSH32      addr_ctx.eflag   ; 取 ctx.eflag 成员地址
00001   004C0D12  07265C58  91  POP32       R2               ; 送到 R2
00002   004C0D13  07265CE9  95  POP32       [R2]             ; 弹出栈内的 dword 到[R2]
00003   004C0D14  07265C7E  55  PUSH32      addr_ctx.edi     ; 保存 edi
00004   004C0D16  07265CD8  91  POP32       R2
00005   004C0D17  07265C69  95  POP32       [R2]
00006   004C0D18  07265CFE  55  PUSH32      addr_ctx.esi     ; 保存 esi
00007   004C0D1A  07265C57  91  POP32       R2
00008   004C0D1B  07265CE8  95  POP32       [R2]
00009   004C0D1C  07265C7D  55  PUSH32      addr_ctx.ebp      ; 保存 ebp
00010   004C0D1E  07265CD4  91  POP32       R2
00011   004C0D1F  07265C65  95  POP32       [R2]
00012   004C0D20  07265CFA  55  PUSH32      addr_ctx.ebx     ; 弹出 esp
00013   004C0D22  07265C55  91  POP32       R2
00014   004C0D23  07265CE6  95  POP32       [R2]
00015   004C0D24  07265C7B  81  SetEcxIdx   01               ; 设置 ecx 在 ctx 内的索引
00016   004C0D26  07265CFD  55  PUSH32      addr_ctx.ebx     ; 保存 ebx
00017   004C0D28  07265C58  91  POP32       R2
00018   004C0D29  07265CE9  95  POP32       [R2]
00019   004C0D2A  07265C7E  55  PUSH32      addr_ctx.edx     ; 保存 edx
00020   004C0D2C  07265CD3  91  POP32       R2
00021   004C0D2D  07265C64  95  POP32       [R2]
00022   004C0D2E  07265CF9  55  PUSH32      addr_ctx.ecx     ; 保存 ecx
00023   004C0D30  07265C4F  91  POP32       R2
00024   004C0D31  07265CE0  95  POP32       [R2]
00025   004C0D32  07265C75  55  PUSH32      addr_ctx.eax     ; 保存 eax
00026   004C0D34  07265CCD  91  POP32       R2
00027   004C0D35  07265C5E  95  POP32       [R2]
00028   004C0D36  07265CF3  73  MOV32       R2,esp           ; esp 加 4,丢弃进入 vm
```

```
00029   004C0D37  07265C66  7B  PUSH32      R2              ; 时 push 的 imm32
00030   004C0D38  07265CE1  7C  PUSH32      00000004
00031   004C0D3D  07265C61  6C  ADD32       [esp+4],[esp](丢弃 src)
00032   004C0D3E  07265CCD  A4  POP32       esp
00033   004C0D3F  07265C71  49  ClearKey                    ; key 清 0
```

VM 入口代码用 pushad/pushfd 保存了进入 VM 时的寄存器值。pcode 的开始部分将压栈的数据从 stack 取出，保存到 VMCtx 结构内。由于栈上数据的摆放顺序是已知的，这里可以确定出部分结构成员的含义。使用 CISC-2 类型处理器的 VM，与原程序使用同一个栈，所以没有保存 esp。栈内的 esp 值送到 VMCtx 内 ebx 成员，随后被真正的 ebx 值覆盖了。

如果加壳时使用了 Anti Dumpers 选项，接下来是 anti-dump 代码，约数千行 pcode。这里没有出现，下面开始执行原程序代码。

虽然在加壳时没有使用 Dynamic Opcodes 选项，这里的 pcode 却被变形了。似乎在被保护代码较少时(范例中的原机器码只有 5 行)，Themida 会出现一些文档上没有说明的行为，自行增加 pcode 的复杂性。当 pcode 以变形方式出现时，与 VM handler 类似，也需要进行清理。这里的代码量不大，可以直接分析。

```
00034   004C0D40  00000000  7C  PUSH32      C070EC99
00035   004C0D45  C070ED15  7C  PUSH32      5CE61492
00036   004C0D4A  1D570223  79  SUB32       [esp+4],[esp](丢弃 src)
; push (C070EC99 - 5CE61492) = 638AD807
; 这里的 SUB32(及下面的 ADD32,XOR32 等)是 VM 内部使用的。sub 指令要影响 eflag,
; handler 却没有相应的处理。如果是模仿原程序的指令,会使用另一个 handler:
; VM:004CF484                 pop     eax
; VM:004CF485                 sub     [esp], eax
; VM:004CF488                 pushf
; VM:004CF489                 jmp     l_FetchOpcode
;
; 举个例子(来自别的程序),对原程序 sub edx, ecx 的模仿:
; PUSH32      addr_ctx.edx
; POP32       R2
; PUSH32      [R2]
; PUSH32      addr_ctx.ecx
; POP32       R2
; PUSH32      [R2]
; SUB32       [esp+4],[esp](丢弃 32 位 src,eflag 压栈)
; POP32       [addr_ctx.eflag]  ; 保存 eflag
; PUSH32      addr_ctx.edx
; POP32       R2
; POP32       [R2]  ; 保存目的操作数

00037   004C0D4B  1D57029C  7C  PUSH32      97B616C1
00038   004C0D50  B50D18D9  7C  PUSH32      77AC65F9
00039   004C0D55  2CB97E4E  79  SUB32       [esp+4],[esp](丢弃 src)
; push (97B616C1 - 77AC65F9) =  2009B0C8

00040   004C0D56  2CB97EC7  7B  PUSH32      R2
00041   004C0D57  2CB97E42  6C  ADD32       [esp+4],[esp](丢弃 src)
; [esp]   = 2009B0C8 + R2 = 201CAFF8 (R2 = 0012FF30)
; [esp+4] = 638AD807

00042   004C0D58  2CB97EAE  7C  PUSH32      06E284E5
```

```
00043   004C0D5D  339C030F  7C  PUSH32      26EB342D
00044   004C0D62  5A8737B8  32  XOR32       [esp+4],[esp](丢弃 src)
; push (06E284E5 ^ 26EB342D) = 2009B0C8
; [esp]    =  2009B0C8
; [esp+4] =  201CAFF8
; [esp+8] =  638AD807

00045   004C0D63  5A8737EA  79  SUB32       [esp+4],[esp](丢弃 src)
; [esp]    =  201CAFF8 - 2009B0C8 = 0012FF30
; [esp+4]  = 638AD807

00046   004C0D64  5A873763  32  XOR32       [esp+4],[esp](丢弃 src)
; [esp] = 638AD807 ^ 0012FF30 = 63982737

00047   004C0D65  5A873795  7C  PUSH32      C1A22259
; push  C1A22259

00048   004C0D6A  1C29596A  7B  PUSH32      R2
00049   004C0D6B  1C2959E5  A6  MOV32       R2,702F842F
00050   004C0D70  6C06DDA4  A7  SUB32       R2,121839DD
00051   004C0D75  7E1EE496  25  XCHG32      R2,[esp]
; push (702F842F - 121839DD) = 5E174A52
; [esp]   = 5E174A52
; [esp+4] = C1A22259
; [esp+8] = 63982737

00052   004C0D76  7E1EE4BB  79  SUB32       [esp+4],[esp](丢弃 src)
; [esp]   = C1A22259 -  5E174A52 = 638AD807
; [esp+4] = 63982737

00053   004C0D77  7E1EE434  32  XOR32       [esp+4],[esp](丢弃 src)
; [esp] =  63982737 ^  638AD807 = 0012FF30
; 到这里[esp]=0012FF30,即 R2 的原始值,从 34-53 行只等于 push R2,上面的代码实际执行的是
; push  (638AD807 ^ ((2009B0C8 + R2) - 2009B0C8)) ^ 638AD807

00054   004C0D78  7E1EE466  A6  MOV32       R2,2E8106E4
00055   004C0D7D  509FE2E8  A7  SUB32       R2,2E8106E4
; MOV R2,0 真正要压栈的值

00056   004C0D82  7E1EE46B  25  XCHG32      R2,[esp]
```

完成原程序的第 1 条指令 push 0。

```
00057   004C0D83  7E1EE490  24  XOR32       R2,28B84794
00058   004C0D88  55669D20  7B  PUSH32      R2
; push (R2 ^ 28B84794)

00059   004C0D89  55669D9B  7B  PUSH32      R2
00060   004C0D8A  55669D16  A6  MOV32       R2,81C8A5F9
00061   004C0D8F  D4AE3845  A7  SUB32       R2,59105E65
00062   004C0D94  8DBE6689  25  XCHG32      R2,[esp]
; push (81C8A5F9 - 59105E65) = 28B84794

00063   004C0D95  8DBE66AE  32  XOR32       [esp+4],[esp](丢弃 src)
```

```
; 到这里[esp]=R2,即 57-63 行等于 push R2

00064   004C0D96  8DBE66E0  A6  MOV32       R2,3EC4D9D8
00065   004C0D9B  B37ABF5E  A7  SUB32       R2,3E84796C
; R2=0040606C

00066   004C0DA0  8DFEC669  25  XCHG32      R2,[esp]
```

完成原程序第 2 行 push 0040606C。

```
00067   004C0DA1  8DFEC68E  7B  PUSH32      R2
00068   004C0DA2  8DFEC609  A6  MOV32       R2,6E06E454
00069   004C0DA7  E3F822FB  A7  SUB32       R2,41A58971
00070   004C0DAC  A25DABD3  25  XCHG32      R2,[esp]
; push (6E06E454 - 41A58971) = 2C615AE3

00071   004C0DAD  A25DABF8  7B  PUSH32      R2
00072   004C0DAE  A25DAB73  A6  MOV32       R2,0CFF842B
00073   004C0DB3  AEA22F32  A7  SUB32       R2,6D0F02C0
00074   004C0DB8  C3AD2D19  25  XCHG32      R2,[esp]
; push (0CFF842B - 6D0F02C0) = 9FF0816B

00075   004C0DB9  C3AD2D3E  7B  PUSH32      R2
00076   004C0DBA  C3AD2DB9  A6  MOV32       R2,78069F63
00077   004C0DBF  BBABB23C  24  XOR32       R2,0BC9D9DF
00078   004C0DC4  AFE1D881  25  XCHG32      R2,[esp]
; push (78069F63 ^ 0BC9D9DF) = 73CF46BC

00079   004C0DC5  AFE1D8A6  79  SUB32       [esp+4],[esp](丢弃 src)
; [esp] = 9FF0816B - 73CF46BC =  2C213AAF

00080   004C0DC6  AFE1D81F  32  XOR32       [esp+4],[esp](丢弃 src)
; [esp] = 2C615AE3 ^ 2C213AAF = 0040604C
```

完成原程序第 3 行 push 0040604C。

```
00081   004C0DC7  AFE1D851  7B  PUSH32      R2
00082   004C0DC8  AFE1D8CC  72  PUSH16      6BCD
00083   004C0DCB  AFE1B3F3  72  PUSH16      75D9
00084   004C0DCE  AFE1C6BC  91  POP32       R2
00085   004C0DCF  AFE1C64D  6E  POP32       R2
; 这几行是垃圾指令

00086   004C0DD0  AFE1C6BB  7B  PUSH32      R2
00087   004C0DD1  AFE1C636  A6  MOV32       R2,A76615AF
00088   004C0DD6  0887D373  A7  SUB32       R2,51B9C7B6
00089   004C0DDB  593E14AC  25  XCHG32      R2,[esp]
; push (A76615AF - 51B9C7B6) = 55AC4DF9

00090   004C0DDC  593E14D1  7C  PUSH32      5DBC31E0
00091   004C0DE1  B6FA462D  7B  PUSH32      R2
00092   004C0DE2  B6FA46A8  32  XOR32       [esp+4],[esp](丢弃 src)
00093   004C0DE3  B6FA46DA  7C  PUSH32      5DBC31E0
00094   004C0DE8  14B67836  32  XOR32       [esp+4],[esp](丢弃 src)
```

```
; push R2,第90行与93行的imm32相同

00095   004C0DE9  14B67868  6C  ADD32       [esp+4],[esp](丢弃src)
; [esp] = 55AC4DF9 + R2

00096   004C0DEA  14B678D4  7B  PUSH32      R2
00097   004C0DEB  14B6784F  A6  MOV32       R2,C71247A6
00098   004C0DF0  D3A43F53  A7  SUB32       R2,7165F9AD
00099   004C0DF5  A2C1C657  25  XCHG32      R2,[esp]
; push (C71247A6 - 7165F9AD) = 55AC4DF9,和上面加的是同1个值

00100   004C0DF6  A2C1C67C  79  SUB32       [esp+4],[esp](丢弃src)
; 仍然是push R2

00101   004C0DF7  A2C1C6F5  A6  MOV32       R2,5E2F7D26
00102   004C0DFC  FCEEBBBD  A7  SUB32       R2,239DC1CD
00103   004C0E01  DF737AA9  25  XCHG32      R2,[esp]
; 5E2F7D26 - 239DC1CD = 3A91BB59, 从81行到这里等于push 3A91BB59

00104   004C0E02  DF737ACE  7C  PUSH32      77CEA383
00105   004C0E07  57421DCD  7B  PUSH32      R2
00106   004C0E08  57421D48  6C  ADD32       [esp+4],[esp](丢弃src)
; push (77CEA383 + R2)

00107   004C0E09  57421DB4  7C  PUSH32      8819EC3E
00108   004C0E0E  DF5C096E  7C  PUSH32      104B48BB
00109   004C0E13  EFA752A5  79  SUB32       [esp+4],[esp](丢弃src)
; push (8819EC3E - 104B48BB) = 77CEA383

00110   004C0E14  EFA7521E  79  SUB32       [esp+4],[esp](丢弃src)
; 又1个push R2

00111   004C0E15  EFA75297  A6  MOV32       R2,796AE93B
00112   004C0E1A  96CDBB06  24  XOR32       R2,43FB5262
00113   004C0E1F  52D268C8  25  XCHG32      R2,[esp]
; 从104行到这里等于push 3A91BB59
; [esp]    = 3A91BB59
; [esp+4] = 3A91BB59

00114   004C0E20  52D268ED  32  XOR32       [esp+4],[esp](丢弃src)
```

完成原程序第4行push 0。

```
00115   004C0E21  52D2681F  55  PUSH32      addr_ctx.esi
; ctx.esi的地址压栈

00116   004C0E23  52D26878  72  PUSH16      2907
00117   004C0E26  52D241ED  72  PUSH16      0F60
00118   004C0E29  52D24E3F  91  POP32       R2
00119   004C0E2A  52D24ED0  A7  SUB32       R2,7CD38EF1
00120   004C0E2F  2E01C086  7B  PUSH32      R2
00121   004C0E30  2E01C001  7C  PUSH32      7CD38EF1
00122   004C0E35  AAD54F6E  6C  ADD32       [esp+4],[esp](丢弃src)
00123   004C0E36  AAD54FDA  A6  MOV32       R2,52CDCDE0
```

```
00124   004C0E3B  F8188260  24  XOR32       R2,7AA234AE
00125   004C0E40  7D764DD6  25  XCHG32      R2,[esp]
; push 286FF94E

00126   004C0E41  7D764DFB  32  XOR32       [esp+4],[esp](丢弃 src)
; [esp] = addr_ctx.esi ^ 286FF94E = 282ED554 加密的 ctx.esi 地址

00127   004C0E42  7D764D2D  7C  PUSH32      2513F616
00128   004C0E47  A28A43BF  1C  PUSH32      ctx.field40
00129   004C0E48  A28A43DB  6C  ADD32       [esp+4],[esp](丢弃 src)
00130   004C0E49  A28A4347  7C  PUSH32      2513F616
00131   004C0E4E  C79E39D9  79  SUB32       [esp+4],[esp](丢弃 src) ;
; push ctx.field40
; [esp]    = ctx.field40(这里为 0)
; [esp+4] = 282ED554 加密的 ctx.esi 地址

00132   004C0E4F  C79E3952  18  Opcode18
; 这个 handler 的含义不确定,将加密过的 ctx.esi 地址写入 ctx.field40,[esp]为原来的
; ctx.field40,类似 xchg 指令。从下面的代码看,ctx.field40 似乎是被用作临时变量
; 004CF091    mov     eax, dword ptr [esp+4]
; 004CF098    mov     dword ptr [edi+40], eax
; 004CF09E    pop     eax
; 004CF09F    add     esp, 4
; 004CF0A5    push    eax
; 004CF0A6    jmp     l_FetchOpcode

00133   004C0E50  C79E396A  7C  PUSH32      2B17DEC4
00134   004C0E55  F2B618AA  1C  PUSH32      ctx.field40
00135   004C0E56  F2B618C6  6C  ADD32       [esp+4],[esp](丢弃 src)
00136   004C0E57  F2B61832  7C  PUSH32      2B17DEC4
00137   004C0E5C  1DCDF772  79  SUB32       [esp+4],[esp](丢弃 src)
00138   004C0E5D  1DCDF7EB  7C  PUSH32      294F313C
00139   004C0E62  471D28A3  79  SUB32       [esp+4],[esp](丢弃 src)
00140   004C0E63  471D281C  91  POP32       R2
00141   004C0E64  471D28AD  83  ADD32       R2,294F313C
; mov R2, ctx.field40  加密的 ctx.esi 地址

00142   004C0E69  706C596C  26  POP32       ctx.field40
; 还原 ctx.field40

00143   004C0E6A  706C5992  24  XOR32       R2,286FF94E
; 见 126 行 xor 的值。R2=addr_ctx.esi,上面的全是干扰代码

00144   004C0E6F  47FC6068  33  PUSH32      004C0D06
; 调用 API 后的返回地址压栈，是第 2 个 push imm32/jmp l_Vm_Entry 的地址。

00145   004C0E74  40D603A9  A1  NOP
00146   004C0E75  40D6034A  3F  PUSH32      [R2]
; ctx.esi 的值即 MessageBoxA 地址压栈,到这里已完成调用 API 的准备,栈内数据为:
; 0012FF1C   77D36476  user32.MessageBoxA
; 0012FF20   004C0D06  dumped_.004C0D06
; 0012FF24   00000000
; 0012FF28   0040604C  dumped_.0040604C
```

```
; 0012FF2C   0040606C  ASCII "Debug3"

00147   004C0E76  40D60389  8A  JMP         004C0F4B
; 跳转到退出 VM 的准备代码。如果 pcode 由几段组成，这部分代码是共用的
```

将 VMContext 内的寄存器数据压栈。执行 ExitVM 时用 popad/popfd 恢复执行环境。

```
00148   004C0F4B  00000000  55  PUSH32      addr_ctx.eflag
00149   004C0F4D  0000005C  91  POP32       R2
00150   004C0F4E  000000ED  3F  PUSH32      [R2] ; ctx.eflag 压栈
00151   004C0F4F  0000002C  55  PUSH32      addr_ctx.eax
00152   004C0F51  00000084  91  POP32       R2
00153   004C0F52  00000015  3F  PUSH32      [R2] ; ctx.eax 压栈
00154   004C0F53  00000054  55  PUSH32      addr_ctx.ecx
00155   004C0F55  000000AA  91  POP32       R2
00156   004C0F56  0000003B  3F  PUSH32      [R2] ; ctx.ecx 压栈
00157   004C0F57  0000007A  55  PUSH32      addr_ctx.edx
00158   004C0F59  000000CF  91  POP32       R2
00159   004C0F5A  00000060  3F  PUSH32      [R2] ; ctx.edx 压栈
00160   004C0F5B  0000009F  55  PUSH32      addr_ctx.ebx
00161   004C0F5D  000000FA  91  POP32       R2
00162   004C0F5E  0000008B  3F  PUSH32      [R2] ; ctx.ebx 压栈
00163   004C0F5F  000000CA  55  PUSH32      addr_ctx.ebx
00164   004C0F61  00000025  91  POP32       R2
00165   004C0F62  000000B6  3F  PUSH32      [R2] ; ctx.ebx 压栈,popad 时丢弃
00166   004C0F63  000000F5  55  PUSH32      addr_ctx.ebp
00167   004C0F65  0000004C  91  POP32       R2
00168   004C0F66  000000DD  3F  PUSH32      [R2] ; ctx.ebp 压栈
00169   004C0F67  0000001C  55  PUSH32      addr_ctx.esi
00170   004C0F69  00000075  91  POP32       R2
00171   004C0F6A  00000006  3F  PUSH32      [R2] ; ctx.esi 压栈
00172   004C0F6B  00000045  55  PUSH32      addr_ctx.edi
00173   004C0F6D  0000009F  91  POP32       R2
00174   004C0F6E  00000030  3F  PUSH32      [R2] ; ctx.edi 压栈
00175   004C0F6F  0000006F  75  ExitVm
```

ExitVm 最后执行 ret 到 MessageBoxA，完成调用后返回到这段代码的第 2 个 VM 入口。

```
Themida_:004C0D06                push    7265E67h
Themida_:004C0D0B                jmp     l_Vm_Entry
```

第 2 次进入 VM 后没有实质性的代码，唯一的动作是以变形方式计算出原程序的返回地址（4010A8）压栈并退出 VM。这部分 pcode 解码结果就不列出了。

这样的解码结果离真正还原代码还有相当距离。需要解决的问题包括 pcode 变形代码的清理，及分析 pcode 与被模仿的机器码间的对应关系，实现 pcode 到机器码的直接转换。