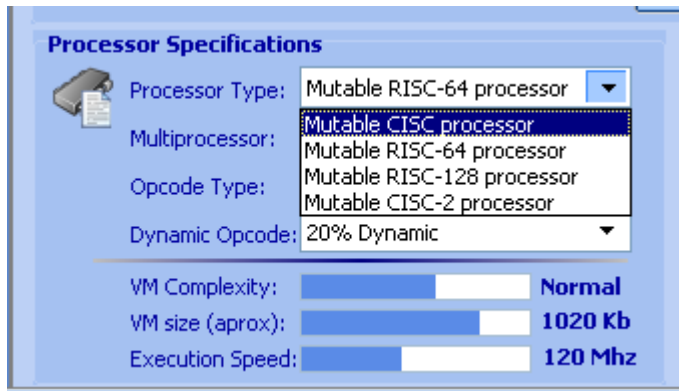


Themida 1.9.1.x CISC Processor VM 简单分析

by softworm

这篇文章不是 VM 的完整分析,我没有看完,看的也不是 Themida 自己,是个用 Themida 保护的软件,名字就不说了。第一次看 CISC 指令集的 VM,肯定有理解上的错误,仅供参考☺。



在加壳的时候,VM 处理器有 4 个选项,我以前写过的 Themida1800 Demo 虚拟机分析,使用的应该是 RISC-128。下面是文档中对 VM Processor 的描述:

Processor Type:

Mutable CISC Processor: This processor is based in CISC technology, where the size of each instruction is different. Each generated CISC processor will be totally different and unique (mutable) for each protected application to avoid a general attack over the embedded virtual machine. CISC processors run faster and with smaller size than RISC processors, though the complexity (security) level is bigger in RISC processors.

Mutable CISC-2 Processor (New): This processor is based in CISC technology, where the size of each instruction is different. Each generated CISC-2 processor will be totally different and unique (mutable) for each protected application to avoid a general attack over the embedded virtual machine. CISC-2 processors have a similar design than the above CISC processor but the internal microinstructions are more complex, this requires a bigger size in the generated virtual opcodes, producing a bigger final application but with higher security level than CISC processors. Notice that if you insert many VM / CodeReplace macros, this processor can produce a bigger size in your application than RISC processors.

Mutable RISC-64 Processor: This processor is based in RISC technology, where the size of each instruction is equal to 64 bits. Each generated RISC-64 processor will be totally different and unique (mutable) for each protected application to avoid a general attack over the embedded virtual machine. The RISC-64 processor is a complex processor with higher security than CISC processors but the size and execution speed are not as optimum as for CISC processors.

Mutable RISC-128 Processor: This processor is based in RISC technology, where the size of each instruction is equal to 128 bits. Each generated RISC-128 processor will be totally different and unique (mutable) for each protected application to avoid a general attack over the embedded virtual machine. The RISC-128 processor offers higher complexity level than CISC and RISC-64 processors, but the execution performance is lower.

从文档看,使用 RISC 指令集的保护更强。Processor Type 可分为 2 类,CISC/CISC-2 与 RISC-64/RISC-128。从文档中看不出 CISC 与 CISC-2 在细节上的区别,我也没有自己加壳去仔细验证,即不清楚这里用的究竟是 CISC 还是 CISC-2。用 Themida1.9.1.1 分别选择 4 种类型加壳测试了一下,RISC 与 CISC 是可以区分的:

使用 RISC processor,VM 相关部分在动态分配的 6 块内存中,这也是实际使用得较多的,脱壳后需要补 6 个区段(如果不还原代码)。使用 CISC processor,VM 的 context 及 handler 就在壳代码所在区段,脱壳后不需要补区段。VM 引擎与原程序共用 1 个堆栈。

值得注意的是,我读了一遍 scherzo 写的<Inside Code Virtualizer>,文章截图中的入口代码与我看到的极其相似,猜测 CV 使用的可能就是 CISC 指令集的某一种。

VM 入口代码

```
:11016C00 sub_11016C00 proc near ; CODE XREF
:11016C00
:11016C00 ; FUNCTION CHUNK AT Themida_:1142FB61 SIZE 00000000
:11016C00
:11016C00 mov     eax, large fs:0
:11016C06 push    0FFFFFFFh
:11016C08 push    offset sub_11129A2C
:11016C0D push    eax
:11016C0E mov     large fs:0, esp
:11016C15 sub     esp, 0Ch
:11016C18 push    edi
:11016C19 jmp     loc_1142FB61
:11016C19 sub_11016C00 endp
:11016C19
```

原程序以 1 个 JMP 开始执行 VM 保护代码。

```
Themida_:1142FB61 loc_1142FB61: ; CODE
Themida_:1142FB61 push    79D7A4Ch
Themida_:1142FB66 jmp     1_VmEntry
```

进入 VM, push 的 imm32 是 PCODE 数据的地址,同时被用作 PCODE 数据解码 KEY。

```

1_UmEntry:                                     ; CODE XREF: Themida_:113A0877↓
                                                ; Themida_:113A0881↓j ...
        pusha
        pushf
        cld
        call    $+5
        pop     edi
        sub     edi, 7928265h    ; <-09A5262B
        mov     eax, edi
        add     edi, 7927F7Dh    ; 1137A5A8,指向VMContext
        cmp     eax, [edi+VMContext.DeltaOffset]
        jnz     short 1_FixHandlerAddr ; 若ctx内的deltaoffset
                                                ; 与实际值不等
        jmp     short loc_1137A8B9
; -----
1_FixHandlerAddr:                             ; CODE XREF: InitKey+347D92↑j
        mov     [edi+VMContext.DeltaOffset], eax
        mov     ecx, 0A7h        ; 167个handler
        jmp     short loc_1137A8B5
; -----
loc_1137A8B0:                                 ; CODE XREF: InitKey+347DA7↓j
        add     [edi+ecx*4+40h], eax ; 根据VM加载地址逐个校正表
                                                ; 内各handler的地址数据
        dec     ecx
loc_1137A8B5:                                 ; CODE XREF: InitKey+347D9E↑j
        or      ecx, ecx
        jnz     short loc_1137A8B0

```

这里的代码与 scherzo 文章中截图完全相同。

```

B9 loc_1137A8B9:                             ; CODE XREF: InitKey+347D99↑j
B9      mov     esi, [esp+24h]                ; ss:[003AF970]=079D7A4C
B9      ; 进入VM前push的imm32
BD      mov     ebx, esi                      ; push的imm32同时用作
BD      ; pcode解码key
BF      add     esi, eax                      ; <-pcode地址
C1      mov     ecx, 1
C6
C6 loc_1137A8C6:                             ; CODE XREF: InitKey+347DB1↑j
C6      xor     eax, eax
C8      lock cmpxchg [edi+VMContext.Busy], ecx
CD      jnz     short loc_1137A8C6
CF
CF 1_FetchOpcode:                             ; CODE XREF: Themida_:1137F0↓j
CF      ; Themida_:1137ABC7↓j ...
CF      lodsb
D0      push    ecx
D1      jmp     loc_1137BD86

```

ebx 为进入 VM 时 push 的 dword,esi 指向 PCODE 数据。

下面执行取指令代码,这部分代码不在 handler 地址表内,也是变形代码,esi 下面为清理后的结果。

vm:11469000	lodsb	
vm:11469001	add	al, bl
vm:11469003	xor	al, 53h
vm:11469006	xor	bl, al
vm:11469008	movzx	eax, al
vm:1146900B	jmp	dword ptr [edi+eax*4]

解码 opcode,跳到对应的 handler。注意进入 VM 时 push 的 dword,又被用作解码 key。所有的 handler(除了退出 VM 的),在执行完后都会跳到这里继续取指循环。xor 用的 imm8(也许包括解码算法)是加壳时随机生成的。

与 RISC 相似,被 VM 保护的代码也被 call 及非模仿指令分成若干段。但进入 VM 时的代码,不象 RISC 指令集那样表现为连续的 PUSH/JMP,第 1 个 PUSH/JMP 与其它的是分开的。

```

hemida_ :1142FB78 dd 01001287h, 002E0770h, 72810141h, 0
hemida_ :1142FB78 dd 0B90809A6h, 0EBDA5557h, 0F5787CDEh, 1
hemida_ :1142FB78 dd 8D20046Ah, 0EE328EC0h, 434E9DF0h, 1
hemida_ :1142FB78 dd 53760FD7h, 1545353Ch, 8B650941h, 68
hemida_ :1142FB78 dd 0FD7074D2h, 0C8634870h, 3E5C78C5h, 1
hemida_ :1142FB78 dd 3B8h, 95CA9300h, 9094EC60h, 333195B
hemida_ :1142FB78 dd 9193F15Eh, 313391BEh, 9795EF5Eh, 4E
hemida_ :1142FB60 db 40h ; @
hemida_ :1142FB61 ; -----
hemida_ :1142FB61 ; START OF FUNCTION CHUNK FOR sub_11016C00
hemida_ :1142FB61
hemida_ :1142FB61 loc_1142FB61: ; CODE XREF: s
hemida_ :1142FB61 push 79D7A4Ch
hemida_ :1142FB66 jmp 1_VmEntry
hemida_ :1142FB67 ; END OF FUNCTION CHUNK FOR sub_11016C00

```

这是第 1 个 PUSH/JMP,上面就是这段代码的 PCODE 数据。向上翻:

```

Themida_:11429EDB
Themida_:11429EDB loc_11429EDB: ; CODE XREF
Themida_:11429EDB mov     eax, 79D7898h
Themida_:11429EE0 add     eax, ebp
Themida_:11429EE2 jmp     loc_1142FB72
Themida_:11429EE7 ; -----
Themida_:11429EE7 push    79DB24Fh
Themida_:11429EEC jmp     1_VmEntry
Themida_:11429EF1 ; -----
Themida_:11429EF1 push    79DB2A1h
Themida_:11429EF6 jmp     1_VmEntry
Themida_:11429EFB ; -----
Themida_:11429EFB push    79DB2EFh
Themida_:11429F00 jmp     1_VmEntry
Themida_:11429F05 ; -----
Themida_:11429F05 push    79DB33Ch
Themida_:11429F0A jmp     1_VmEntry
Themida_:11429F0F ; -----
Themida_:11429F0F push    79DB388h
Themida_:11429F14 jmp     1_VmEntry
Themida_:11429F19 ; -----
Themida_:11429F19 push    79DB3DBh
Themida_:11429F1E jmp     1_VmEntry
Themida_:11429F23 ; -----
Themida_:11429F23 push    79DB420h
Themida_:11429F28 jmp     1_VmEntry

```

直到 PCODE 数据结束,出现 PUSH/JMP,这些 PUSH/JMP 与第 1 个是同属一段 VM 保护代码的。如果该段代码内没有 call 或非模仿指令,则只有第 1 对 PUSH/JMP。

VM Context

```

00000000 VMContext      struc ; (sizeof=0x44)
00000000 ecx            dd ?
00000004 eax            dd ?
00000008 edx            dd ?
0000000C edi            dd ?
00000010 ebx            dd ? ; 这不是 esp,而是 ebx,vm 不切换栈,不必保留 esp
00000014 esi            dd ?
00000018 ebp            dd ?
0000001C eflag          dd ?
00000020 JxxFlag        dd ? ; 是否执行控制转移的标记
00000024 counter        dd ? ; 模仿控制转移指令时使用
00000028 IndexOfEcxCtx dd ? ; ecx 在 ctx 内的 index,在模仿 jcxz/jecxz 时使用
0000002C DeltaOffset    dd ?
00000030 Busy           dd ?
00000034 field_34       dd ?
00000038 field_38       dd ?
0000003C RellocOffset   dd ? ; 处理重定位数据的 offset
00000040 field_40       dd ?
00000044 VMContext      ends

```

有几个 field 不清楚含义,还原代码时没有碰上,这篇文章不是完整分析☺。

```
Themida_:1137A5A8 Um_Ctx          dd 1
Themida_:1137A5AC          dd 1
Themida_:1137A5B0          dd 6F890h
Themida_:1137A5B4          dd 0
Themida_:1137A5B8          dd 112BD014h
Themida_:1137A5BC          dd 1
Themida_:1137A5C0          dd 6F88Ch
Themida_:1137A5C4          dd 200202h
Themida_:1137A5C8          dd 2
Themida_:1137A5CC          dd 1
Themida_:1137A5D0          dd 0
Themida_:1137A5D4          dd 9A5262Bh
Themida_:1137A5D8          dd 0
Themida_:1137A5DC          dd 0
Themida_:1137A5E0          dd 0
Themida_:1137A5E4          dd 0
Themida_:1137A5E8          dd 0
Themida_:1137A5EC Op11          dd offset Um_Mov_R2_esp ; 1138502E
Themida_:1137A5F0          dd offset loc_1146900D ; 1137F383
Themida_:1137A5F4          dd offset loc_1146901F ; 11387AA2
Themida_:1137A5F8          dd offset loc_11469034 ; 11385761
Themida_:1137A5FC Op15          dd offset Um_Push16_Ptr_R2 ; 11385728
Themida_:1137A600          dd offset loc_11469060 ; 11386802
Themida_:1137A604 Op17          dd offset Um_Add_R2_Reg ; 1137E9B7
Themida_:1137A608          dd offset loc_1146908D ; 113838DD
```

VMConext 下就是 handler 表,所有 Context 成员,加上 handler 表,用 1 个 byte 即可寻址,为避免混淆,opcode 的编码没有从 0 开始,而是直接从 0x11 开始的。handler 表内的数据已经用清理变形代码后的代码地址替换了,注释是原代码地址。

进入 VM 时保存执行环境,退出 VM 时恢复各寄存器。这些代码都放到 VM 内执行了。

Opcode 与 PCODE 数据

这个 VM 的结构与以前看过的 RISC 指令集 VM 相比,的确简单很多,绝大部分 handler 一目了然,清理后只有几行,不需要进行分类。所有的操作都通过 stack 实现,这给分析 PCODE 带来了一些麻烦,尤其是操作数在栈上的时候,后面再讲。

所有的 Opcode 只有 1 字节。每个 handler 可以有 0,1,2,4 字节的操作数,如果有操作数,需要对 operand 解码,如下面的 handler 解码 1 个 dword 并压栈。

```
vm:114690F8 Vm_Push32_Imm32:
vm:114690F8          lodsd
vm:114690F9          sub     eax, ebx
vm:114690FB          xor     eax, 33B54307h
vm:11469101          sub     eax, 45BA9539h
vm:11469107          add     ebx, eax ; 用解码结果更新 key
vm:11469109          push    eax
vm:1146910A          jmp     l_FetchOpcode
```

进入 VM 后,有几个寄存器是有特殊含义的:

ebx -> 解码 key
esi -> 指向 PCODE 数据
edi -> 指向 Context

handler 没有使用 ebp(变形代码会 PUSH/POP 保护后使用)。handler 实际使用了 3 个寄存器,为避免混淆,另外命名,可以认为这 3 个寄存器是 VM 内部的寄存器。

eax -> R0 用得很少
ecx -> R1 主要用于实现移位指令,移位次数放到 **ecx**
edx -> R2 主要使用这个

下面以进入 VM 时保存 context 为例,其中从栈上取的数据是进入 VM 时的 pushad/pushfd 压入的。列出的 PCODE 解码结果,格式为:

第 1 列 -> counter
第 2 列 -> PCODE 数据地址,
第 3 列 -> 解码 KEY
第 4 列 -> OPCODE
第 5 列 -> 助记符

00000	113D9802	079871D7	33	PUSH32	addr_ctx.eflag
00001	113D9804	079871DD	93	POP32	R2 ; R2<-context 内 eflag 的地址
00002	113D9805	0798714E	1F	POP32	[R2] ; 弹出栈上的 eflag 保存
00003	113D9806	07987151	33	PUSH32	addr_ctx.edi
00004	113D9808	0798715F	93	POP32	R2
00005	113D9809	079871CC	1F	POP32	[R2] ; 弹出栈上的 edi 保存
00006	113D980A	079871D3	33	PUSH32	addr_ctx.esi
00007	113D980C	079871DB	93	POP32	R2
00008	113D980D	07987148	1F	POP32	[R2] ; 弹出栈上的 esi 保存
00009	113D980E	07987157	33	PUSH32	addr_ctx.ebp
00010	113D9810	0798715E	93	POP32	R2
00011	113D9811	079871CD	1F	POP32	[R2] ; 弹出栈上的 ebp 保存
00012	113D9812	079871D2	33	PUSH32	addr_ctx.ebx
00013	113D9814	079871DD	93	POP32	R2
00014	113D9815	0798714E	1F	POP32	[R2] ; 弹出栈上的 esp 保存
00015	113D9816	07987151	AE	SetEcxCtx	00 ; 设置 context 内 IndexOfEcxCtx,即 ecx ; 为 ctx 内第 1 个 dword
00016	113D9818	079871FF	33	PUSH32	addr_ctx.ebx
00017	113D981A	079871C8	93	POP32	R2
00018	113D981B	0798715B	1F	POP32	[R2] ; 弹出栈上的 ebx 保存,注意与上面保存 esp 使 ;用 ctx 同一个 field,即丢弃了 esp,因为 vm ; 与原程序共用栈,不需要保存 esp
00019	113D981C	07987144	33	PUSH32	addr_ctx.edx

00020	113D981E	07987175	93	POP32	R2
00021	113D981F	079871E6	1F	POP32	[R2] ; 弹出栈上的 edx 保存
00022	113D9820	079871F9	33	PUSH32	addr_ctx.ecx
00023	113D9822	079871CA	93	POP32	R2
00024	113D9823	07987159	1F	POP32	[R2] ; 弹出栈上的 ecx 保存
00025	113D9824	07987146	33	PUSH32	addr_ctx.eax
00026	113D9826	07987174	93	POP32	R2
00027	113D9827	079871E7	1F	POP32	[R2] ; 弹出栈上的 eax 保存
00028	113D9828	079871F8	11	MOV32	R2,esp
00029	113D9829	079871E9	AD	PUSH32	R2
00030	113D982A	07987144	1E	PUSH32	00000004
00031	113D982F	0798715E	66	ADD32	[esp+4],[esp](丢弃 src)
00032	113D9830	07987138	26	POP32	esp ; add esp,4 丢弃进入 vm 时 push 的 dword
00033	113D9831	0798711E	7D	ClearKey	; 解码 key(即 ebx)清 0

大部分 OPCODE 根据操作数的 size 分为 3 组,即 8/16/32 位。由于操作需要通过栈实现,对于 8/16 位操作数,有时候需要将操作数零扩展压栈。

前面已经提到了 Vm_Push32_Imm32,下面看看 8 位和 16 位操作。

```

vm:114698B0 Vm_Push8_Imm8:
vm:114698B0          lodsb
vm:114698B1          sub    al, bl
vm:114698B3          sub    al, 0EAh
vm:114698B6          xor    al, 0A8h
vm:114698B9          xor    bl, al
vm:114698BB          movzx  eax, al
vm:114698BE          push   ax      ; 压栈的是 16 位值
vm:114698C0          jmp    I_FetchOpcode

```

PUSH 8 位立即数。

```

vm:11469E5F Vm_Push16_Imm16:
vm:11469E5F          lodsw
vm:11469E61          add    ax, bx
vm:11469E64          add    ax, 0C1E0h
vm:11469E69          sub    ax, 2BF8h
vm:11469E6E          sub    bx, ax
vm:11469E71          movzx  eax, ax ; 这句多余? 大概是程序生成的代码
vm:11469E74          push   ax
vm:11469E76          jmp    I_FetchOpcode

```

PUSH 16 位立即数。

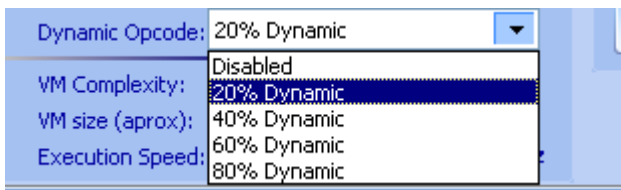
OPCODE 中复杂一点的是控制转移指令,这个 handler 与我在<Themida1800 Demo VM 分析>中的完全相同,就不啰嗦了。不同的是 Jxx 是由 2 个 handler 实现的:

00115	113D98B5	0792BE55	3E	JZ	
00116	113D98BA	0792B26A	87	SetDst	113D9932

第 2 句才真正将 PCODE 指针置到 dst。

PCODE 变形

这个是以以前没有看到过的,即 PCODE 也是变形的。不知道下一步 Themida 会不会把 VM 保护的程序代码也变形☺。



估计是这个选项造成的。

下面举几个例子:

00191	1142A1A0	0000564C	AD	PUSH32	R2
00192	1142A1A1	000056E1	33	PUSH32	addr_ctx.edx
00193	1142A1A3	000056D0	93	POP32	R2
00194	1142A1A4	00005643	58	POP32	R2

明显的垃圾☺。

01305	1142A846	00000000	33	PUSH32	addr_ctx.edi
01306	1142A848	00000030	93	POP32	R2
01307	1142A849	000000A3	1C	PUSH32	[R2]
01308	1142A84A	000000BF	93	POP32	R2
01309	1142A84B	0000002C	11	MOV32	R2,esp

前 4 句等于 mov R2,edi。有了最后一句,前 4 句是垃圾。

03540	1142B516	00000000	33	PUSH32	addr_ctx.ebx
03541	1142B518	0000002F	93	POP32	R2
03542	1142B519	000000BC	1C	PUSH32	[R2]
03545	1142B51D	0000001C	33	PUSH32	addr_ctx.ebp
03546	1142B51F	00000029	93	POP32	R2
03547	1142B520	000000BA	1C	PUSH32	[R2]
03548	1142B521	000000A6	95	SUB32	[esp+4],[esp](丢弃 32 位 src,eflag 压栈)
03549	1142B522	00000033	79	POP32	[addr_ctx.eflag]
03550	1142B524	00000043	93	POP32	R2

这个要复杂一些。前 6 行等于 push ebx,push ebp,看起来似乎是 sub ebx,ebp,但最后一句的 dst 却是 R2。这种代码一般后面还跟着一个 ADD/SUB/XOR... 指令,真正的操作数在前面压栈了。这几行全是垃圾。如果真正是 sub ebx,ebp,最后不会是 pop R2,而应该是这样的代码:

00140	113D98E1	079250AB	33	PUSH32	addr_ctx.ebx
00145	113D98EB	079250DF	93	POP32	R2
00146	113D98EC	0792504C	1F	POP32	[R2]

即将结果写入回目的操作数。

PCODE 变形是递归生成的,各种变形模式可以相互嵌套,需要清理,否则难以理解。

由于 VM 通过栈来实现操作,VM 又与原程序共用一个栈,有的代码看起来比较复杂,特别是操作数本身就在栈上的时候。下面举一个例子。

03025	1142B226	00000000	11	MOV32	R2,esp
03030	1142B22C	00000044	7D	ClearKey	
03035	1142B232	00000051	1C	PUSH32	[R2] ; 原 dword ptr [esp]压栈
03044	1142B243	0000A8D4	7D	ClearKey	
03045	1142B244	00000000	33	PUSH32	addr_ctx.ebx
03046	1142B246	0000002F	93	POP32	R2
03055	1142B251	000000FA	7D	ClearKey	
03056	1142B252	00000000	1C	PUSH32	[R2] ; ebx 压栈
03070	1142B269	0000EB95	7D	ClearKey	
03078	1142B276	000027F2	11	MOV32	R2,esp
03085	1142B27E	00002733	7D	ClearKey	
03092	1142B286	000000DD	AD	PUSH32	R2
03093	1142B287	00000070	1E	PUSH32	00000008
03094	1142B28C	00000076	66	ADD32	[esp+4],[esp](丢弃 src)
03095	1142B28D	00000010	93	POP32	R2 ; esp 加 8,指向 2 个 push 之前 ; 的位置
03096	1142B28E	00000083	7D	ClearKey	
03102	1142B298	0000DC2F	1F	POP32	[R2] ; 弹出栈内 ebx 值
03110	1142B2A5	00005188	7D	ClearKey	
03111	1142B2A6	00000000	33	PUSH32	addr_ctx.ebx
03112	1142B2A8	0000002F	93	POP32	R2
03124	1142B2BA	00009742	7D	ClearKey	
03125	1142B2BB	00000000	1F	POP32	[R2] ; 弹出栈内原[esp]的 dword
03132	1142B2C3	0000003F	7D	ClearKey	

注意 counter 的值,这已经是清理过的 PCODE,原始的有 107 行。这段代码实现的是 xchg ebx,[esp],是 AntiDump 相关代码,原始代码就是变形代码。

```

push    ebx
mov     ebx,esp
push    ebp
mov     ebp,00000004
add     ebx,ebp
pop     ebp
sub     ebx,00000004
xchg    ebx,[esp] <- here
pop     esp
mov     [esp],edx

```

全部代码等于 PUSH EDX,由此可见 Themida VM 的性能损耗有多大☺。

Anti-Debug

我自己原来改过的 OllyDbg 在 WinXP SP2 下会被检测到,所以看了一下这部分代码,下面是修复的代码(你的 OD 不一定是这里过不去☺)。

```

or      eax,eax      ; IsDebuggerPresent 的返回
jnz     I_113D98C9
cmp     dword ptr [ebp+07922CB1],00000000 ; <- 这里为 1,被发现了
jz      I_113D9932

```

I_113D98C9:

```

lea     edi,[ebp+0792501D]
mov     eax,00000001
jmp     edi

```

I_113D9932:

```

xor     [ebp+07920ABD],eax
add     eax,[ebp+07921585]
add     [ebp+0792219D],eax
xor     eax,ebx
.....省略

```

; 下面是将该 dword 置 1 的代码,被 call 分为 9 段

; 第 1 段

```

mov     eax,eax
cmp     dword ptr [ebp+0792299D],0
jnz     loc_1

cmp     dword ptr [ebp+07920499],0

```

```

    jz         loc_2

loc_1:
    push      eax
    push      ebx
    mov       eax,eax
    mov       eax,000004D1
    mov       [ebp+0792309D],eax
    lea       ebx,[ebp+0794FBCC]
    call      ebx

; 第 2 段

    pop       ebx
    pop       eax

loc_2:
    cmp       dword ptr [ebp+07920499],0
    jz        loc_3 ;

    push      eax
    push      ebx
    mov       eax,000004D1
    mov       [ebp+0792309D],eax
    lea       ebx,[ebp+0794F8BB]
    call      ebx

; 第 3 段

    pop       ebx
    pop       eax

loc_3:
    mov       eax,eax
    cmp       dword ptr [ebp+0792340D],00000001
    jnz       loc_4 ; 跳到 loc_4 为 OK

    cmp       dword ptr [ebp+07920621],00000000
    jnz       loc_4

    cmp       dword ptr [ebp+07921F75],00000000
    jnz       loc_4

    mov       byte ptr [ebp+07920325],49
    push      8C1529E9

```

```

push    dword ptr [ebp+079228D1]
lea     eax,[ebp+07923BFA]
call    eax      ; 调用 homemade_GetProcAddress 获取 IsBadReadPtr

; 第 4 段

mov     [ebp+07981030],eax
mov     eax,fs:[00000030]
mov     eax,[eax+0000000C]

mov     ecx,00000010
and     eax,FFFFFF00
add     eax,00001000      ; 从下一页开始
jmp     loc_5

loc_8:
push    eax
push    ecx
push    00000004          ; 4 bytes
push    eax              ; _PEB_LDR_DATA 下一页起始地址
call    [ebp+07981030]    ; IsBadReadPtr

; 第 5 段

mov     ebx,eax
pop     ecx
pop     eax
or      ebx,ebx
jz      loc_6            ; 可读?
jmp     loc_7            ; 开始检查

loc_6:
add     eax,00001000      ; 到下 1 页
dec     ecx              ; 读 16 页(每页的前 4 bytes)

loc_5:
or      ecx,ecx
jnz     loc_8
jmp     loc_4            ; 若 16 页均可读,则跳过下面的 anti;-)

loc_7:
sub     eax,00000010      ; 退到最后 1 个可读页(的最后 16 bytes)
push    eax
push    00000010
push    eax

```

call **[ebp+07981030]** ; 这 16 bytes 是否可读?

; 第 6 段

mov **ebx,eax**

pop **eax**

or **ebx,ebx**

jnz **loc_4** ; 不能读则跳过

; 4 个 DWORD 全为 FEEEFEEE 则设置标记

cmp **dword ptr [eax],FEEEFEEE**

jnz **loc_4**

cmp **dword ptr [eax+00000004],FEEEFEEE**

jnz **loc_4**

cmp **dword ptr [eax+00000008],FEEEFEEE**

jnz **loc_4**

cmp **dword ptr [eax+0000000C],FEEEFEEE**

jnz **loc_4**

mov **eax,eax**

cmp **dword ptr [ebp+0792299D],00000000**

jnz **loc_8**

cmp **dword ptr [ebp+07920499],00000000**

jz **loc_9**

loc_8:

push **eax**

push **ebx**

mov **eax,eax**

mov **eax,000004D1**

lea **ebx,[ebp+0794F939]**

call **ebx**

; 第 7 段

pop **ebx**

pop **eax**

mov **eax,eax**

loc_9:

mov **dword ptr [ebp+07922CB1],00000001** ; debugger detected

```

loc_4:
    mov     eax,eax
    cmp     dword ptr [ebp+0792299D],00000000
    jnz     loc_10

    cmp     dword ptr [ebp+07920499],00000000
    jz      loc_11

```

```

loc_10:
    push    eax
    push    ebx
    mov     eax,eax
    mov     eax,000004D1
    mov     [ebp+07920651],eax
    lea     ebx,[ebp+0794FC33]
    call    ebx

```

; 第 8 段

```

    pop     ebx
    pop     eax

```

```

loc_11:
    cmp     dword ptr [ebp+07920499],00000000
    jz      loc_12

```

```

    push    eax
    push    ebx
    mov     eax,000004D1
    mov     [ebp+07920651],eax
    lea     eax,[ebp+0794F8D4]
    call    ebx

```

; 第 9 段

```

    pop     ebx
    pop     eax

```

```

loc_12:
    mov     eax,eax
    mov     eax,[ebp+0798102C]
    xor     eax,35F9E374
    add     eax,ebp
    jmp     eax

```

这样的检测,我的 OllyDbg 本来应该能通过的,问题在 WinXP SP 自己。SP2 为防止溢出攻击,对 PEB 的地址作了随机化处理,如果用来隐藏 OllyDbg 的代码使用了 7FFDF000 的硬编码地址,就会对错误的地址进行数据清理。

如果清理代码是注入的,直接从 fs:[30h]取,不会有问题。如果是在被调试进程之外用 WriteProcessMemory 实现的,需要用 GetThreadSelectorEntry 获取 fs 段的地址。

Virtual Machine Anti-Dump

这里的讨论不适用于 1.8.x.x。在写这篇文章的时候,Themida 升级到 1.9.4.0,专门提到增强了 Anti-Dump,所以估计也对付不了 1940 了。

原程序中一段被 VM 保护的代码,开始执行时首先就是 AntiDump。我看的这个程序,用于 AntiDump 的 PCODE 约 9500 行。AntiDump 原始代码就是变形代码。

脱壳后运行,出现内存访问异常,异常出现在 VM 内,试图访问 8D8C0 的数据。下面是部分还原代码:

```
.....
push    dword ptr [ebp+0792090D] ; [11372F38]= 8D8C0h

push    ebp

push    ebx
mov     ebx,501B7E96
mov     ebp,C605CE17
add     ebp,ebx
pop     ebx

add     [esp+4],ebp
pop     ebp

mov     eax,[esp]
add     esp,00000004
sub     eax,16214CAD

push    00001901
mov     [esp],ecx

push    esi
mov     esi,0A09435A ; Magic Number 为硬编码值
sub     esi,C18ED54D ; 487a6e0d
mov     ecx,esi
pop     esi

push    esi
```

```

mov     esi,ecx
mov     edx,esi
pop     esi
pop     ecx

cmp     [eax],edx      ; Anti-Dump
jz      I_1142D17E

```

这里 `eax` 为 `8D8C0`,这里的 `dword` 必须等于硬编码值 `487A6E0D`。这是壳代码分配的内存,dump 出来后自然没有。这个地址保存在壳代码区段内:

```

Themida_:11372ED4      dd 0EEC5EAA1h, 3 dup(0)
Themida_:11372EE4      dd 0C09C0528h, 3 dup(0)
Themida_:11372EF4      dd 9A31E5h, 9 dup(0)
Themida_:11372F1C      dd 11A7C40h, 113B21E3h, 77D1A8ADh, 4 dup(0)
Themida_:11372F38      dd 8D8C0h, 113CFCEBh, 17h dup(0)
Themida_:11372F9C      dd 7C80ABDEh, 12h dup(0)
Themida_:11372FE8      dd 6F820h, 4 dup(0)
Themida_:11372FEF      dd 05F50000h, 1130722E

```

知道为什么就简单了☺。

还出现了另外一种。我看的是个 DLL,却检测了宿主 EXE 的 MZ 头。至于 EXE 文件是否存在这种 AntiDump,我没有验证。

这段代码直接看了一下,没有还原代码了。从 `11374BA8` 获取 EXE 的加载地址。

```

Themida_:11374BA6      db 0
Themida_:11374BA7      db 0
Themida_:11374BA8      dd 4000000h
Themida_:11374BAC      dd 0
Themida_:11374BB0      dd 0
Themida_:11374BB4      dd 0

```

检测 `MZ+0Ch` 的 `word`。

```

07311 1144CAA8 00000000 15 PUSH16 [R2]
07328 1144CABF 000000DC A2 PUSH16 0600
07339 1144CACF 0000008F 4F CMP16 [esp+2],[esp](丢弃 16 位 src,dst,eflag 压栈)
07358 1144CAE9 000034E8 79 POP32 [addr_ctx.eflag]
07360 1144CAEC 00000000 3E JZ
07361 1144CAF1 00000C3F 87 SetDst 1144CE3B ; Anti-Dump

```

即 `MZ+0Ch` 应为 `600h`。正常的文件这里为 `FFFF`。

00400000	4D 5A	ASCII "MZ"	DOS EXE Signature
00400002	9000	DW 0090	DOS_PartPag = 90 (144.)
00400004	0300	DW 0003	DOS_PageCnt = 3
00400006	0000	DW 0000	DOS_ReloCnt = 0
00400008	0400	DW 0004	DOS_HdrSize = 4
0040000A	0000	DW 0000	DOS_MinMem = 0
0040000C	FFFF	DW FFFF	DOS_MaxMem = FFFF (65535.)
0040000E	0000	DW 0000	DOS_ReloSS = 0
00400010	B800	DW 00B8	DOS_ExeSP = B8
00400012	0000	DW 0000	DOS_ChkSum = 0
00400014	0000	DW 0000	DOS_ExeIP = 0

AntiDump 我就碰到这 2 种,不清楚还有没有别的。

Themida 的 CISC 指令集 VM 相比之下比较简单,实战中也出现较少,倒是可以用来学学 VM©。

致谢

www.pediy.com

www.unpack.cn