第三十章-P-CODE-Part2

(本章 CrackMe 支持库 MSVBVM50.DLL)

本章我们继续讨论 P-CODE。

以下是我从 JBDUC 的教程里收集的一些操作码:

6c → ILdRf 将指定操作数压入堆栈

1b → LitStr5 将字符串压入堆栈

fb → Lead0

30 → EqStr 比较两个字符串(与 Lead0 配合使用)

2f → FFree1Str 释放内存空间

1a → FFree1Ad 释放内存空间

0f → VCallAd 通过虚拟机运行操作码

1c → BranchF 条件跳转指令,如果栈顶的值为 false 则跳转(相当于汇编指令 JNE/JNZ)

1d → BranchT 条件跳转指令,如果栈顶的值为 true 则跳转(相当于汇编指令 JE/JZ)

1e → Branch 无条件跳转(嘿嘿,相当于汇编指令 JMP)

fc → Lead1

c8 → End 终止程序(与 Lead1 配合使用)

f3 → LitI2 将立即数压入堆栈

f4 → LitI2_Byte 将指定数据转化为字节整型并压入堆栈

70 → FStrI2 将栈顶的 WORD 型元素保存到内存单元中,然后执行出栈操作

6b → FLdI2 将 WORD 型参数压入堆栈

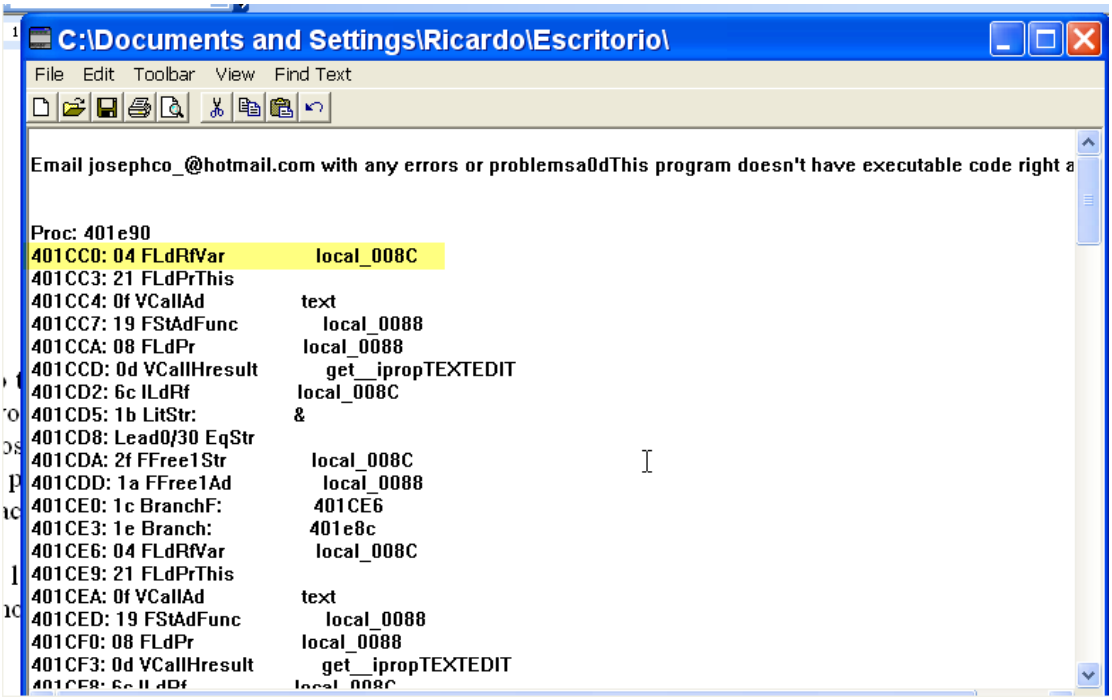
a9 → AddI2 栈顶两个 WORD 型元素相加,相加的结果置于栈顶

ad → SubI2 栈顶两个 WORD 型元素相减,相减的结果置于栈顶

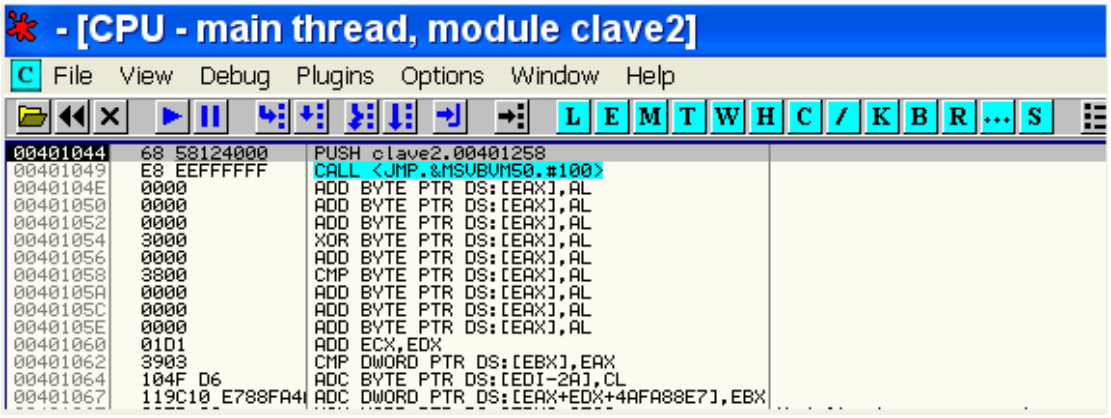
b1 → MulI2 栈顶两个 WORD 型元素相乘,相乘的结果置于栈顶

好了,以上列出了一些操作码以及相应的含义,这里还有一份<<P-Code_OPCODES>>文档,这份文档是关于 VB P-CODE 虚拟机的说明文档,其阐述了操作码的解析原理(但是并不全,嘿嘿)。如果大家遇到了不熟悉的操作码的话,可以参考一下该文档,可能有帮助。

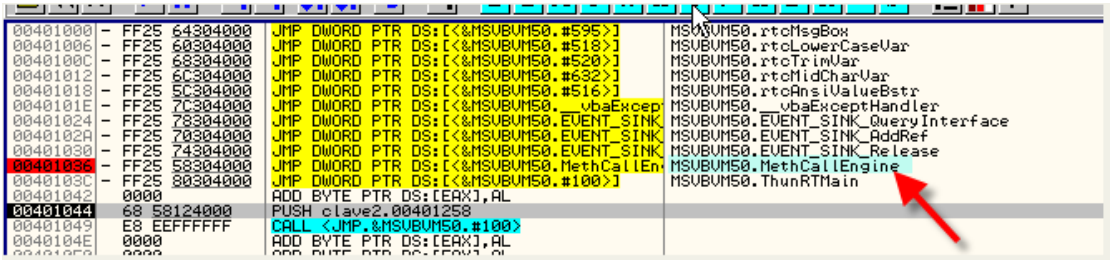
好了,这里我们首先来讲解 clave2 这个 CrackMe,将其加载到 ExDec 看看都显示些什么。



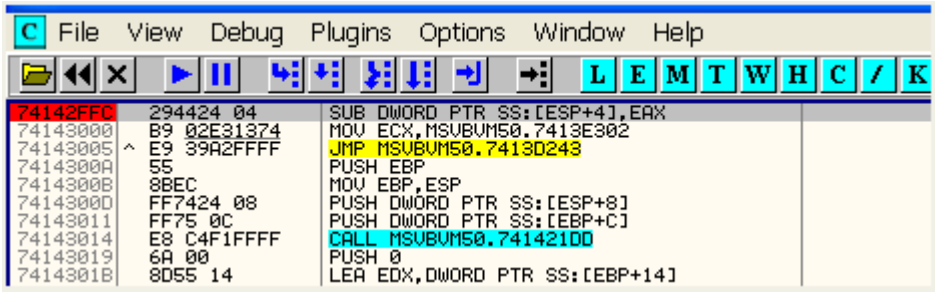
这里我们可以看到开始于 401CC0 处,这里不能完全依然于 ExDec,因为有时候它的分析不怎么准确,所以我们还是像上一章节一样手工来定位第一个操作码吧。



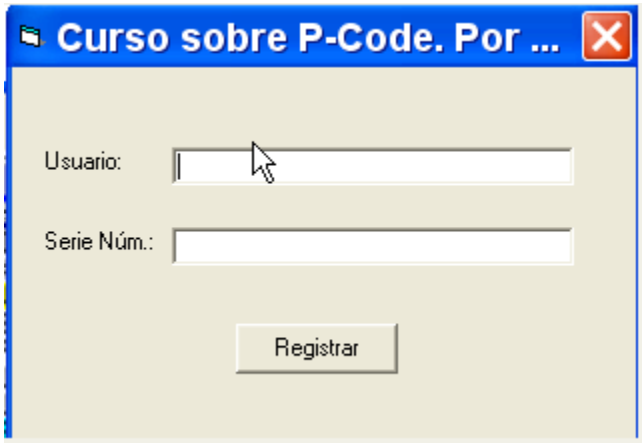
我们定位到入口点上面的 API 函数 MethCallEngine。



这里我们给 JMP MethCallEngine 这一行设置一个断点,为了防止还有其他地方调用 MethCallEngine,我们在 JMP MethCallEngine 这条指令上面单击鼠标右键选择-Follow 定位到 MethCallEngine 的入口点,在入口点处也设置一个断点。



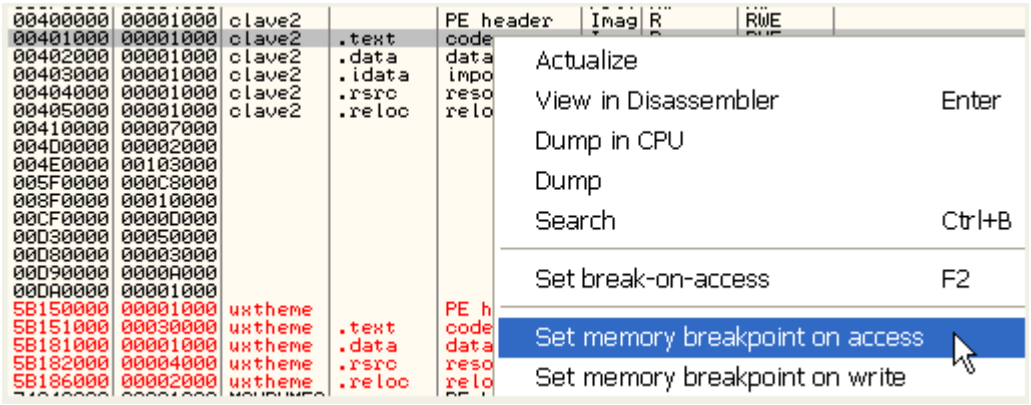
我们运行起来看看会不会触发刚刚设置的断点。



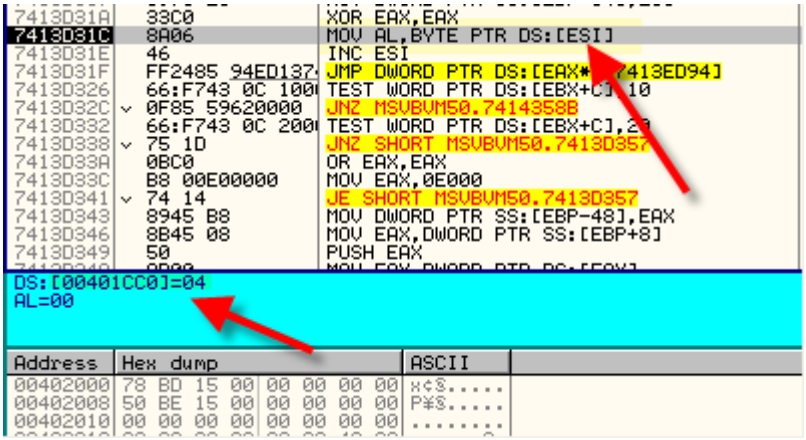
我们可以看到弹出了注册窗口,但是并没有触发我们设置的断点,说明在执行 P-CODE 之前注册窗口就产生了,可能有的程序执行 PCODE 在窗口产生之前,而我们这里刚好相反,其实这无关紧要,现在我们随便输入一个错误的用户名和序列号。

接着我们单击 Registrar(注册)按钮,就会断在 JMP MethCallEngine 这一行,好,现在单击工具栏中的 M 按钮打开区段列表窗口定位

到代码段(这里我们使用原版的 OD,不用那个 Patch 过的 OD,那个 Patch 过的 OD 对 P-CODE 应用程序并不奏效),对代码段设置内存访问断点。



接下来我们多运行几次,直到断在读取第一个操作码的指令处为止。



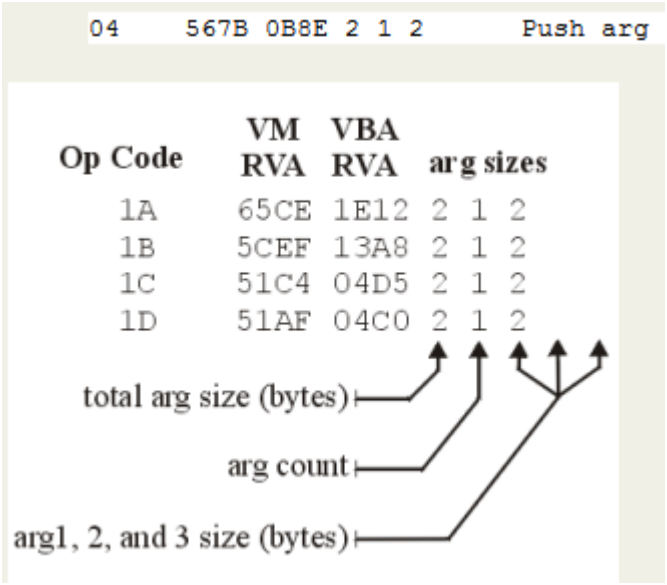
这里我们可以看到 ESI 指向了第一个操作码,并且该操作码将被保存到 AL 中。

和 ExDec 中显示的第一个操作码是位于 401CC0 处的 04 一致。

```
Proc: 401e90
401CC0: 04 FLdRfVar                    local_008C
```

我们应该还记得上一章介绍过的 04 这个操作码是将后面紧跟的参数压入堆栈,这里该参数是 EBP - 8C。

下面我们来看看 P-CODE 的说明文档中操作码是如何解析的,如下图:

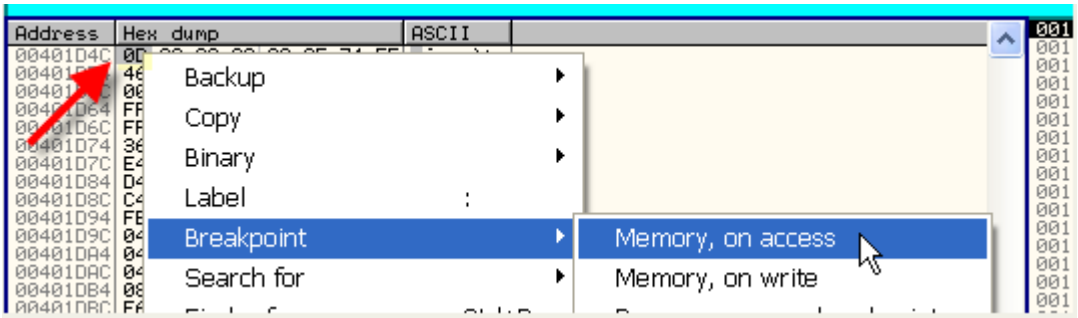


04 567B 0B8E 2 1 2 就是将一个参数压入堆栈, 0B8E 指的是对应参数的 RVA(相对虚拟地址), 第一个 2 指的是所有参数所占的总字节数,接下来的一个 1 指的是参数的个数,最后的一个 2 指的是单个参数所占的字节数,由于这个例子只有一个参数,所以最后只有一个 2,如果具有多个参数的话,后面会依次显示各个参数所占的字节数。

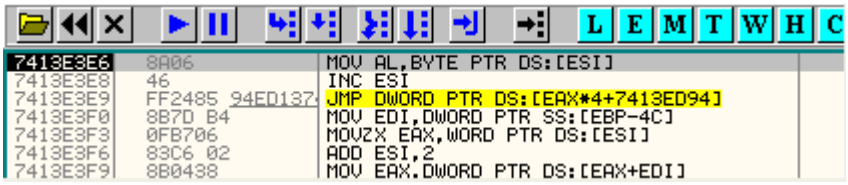
我们这里的第一个操作码所执行的操作即 PUSH EBP - 8C,继续看下面的操作码,但是本章我们不跟上一章那样从头到尾跟踪每个操作码,这里我们只跟踪关键的操作码。

```
401D4C: 0d VCallHresult              get__ipropTEXTEDIT

401DFB: 0d VCallHresult              get__ipropTEXTEDIT
```

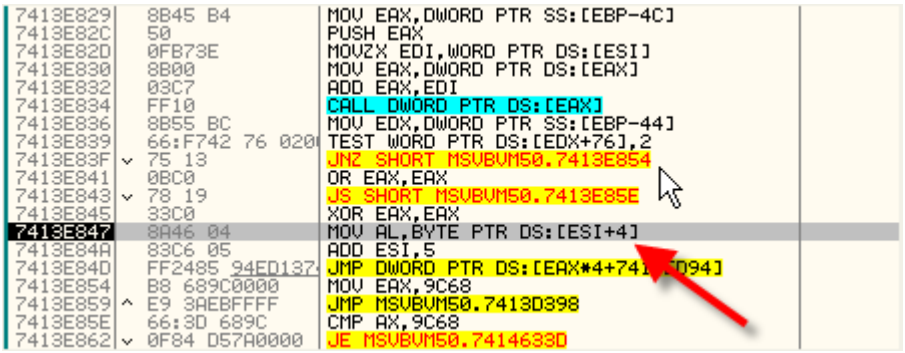
我们看到这两处 VCALLHresult,都是读取文本框中用户输入的信息,第一个有可能是读取用户输入的用户名,第二个可能是读取用户输入的序列号,我们直接给 401D4C 地址处的操作码设置内存访问断点。



接着我们运行起来。



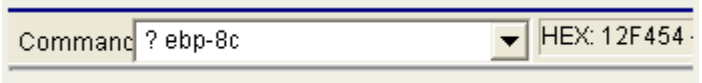
断了下来,继续往下跟踪直到读取下一个操作码的指令为止。



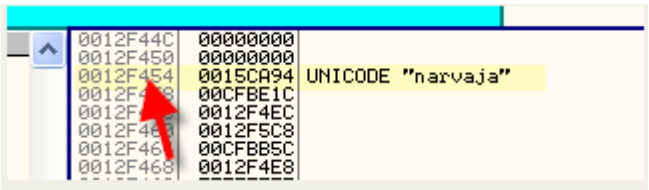
这里我们跟到了读取下一个操作码的指令处。

```
401D4C: 0d VCallHresult              get__ipropTEXTEDIT
401D51: 3e FLdZeroAd                 local_008C
```

可以看到局部变量 local_008C 即 EBP - 8C,我这里 EBP - 8C 等于 12F454,一起来看看该地址处保存了什么。

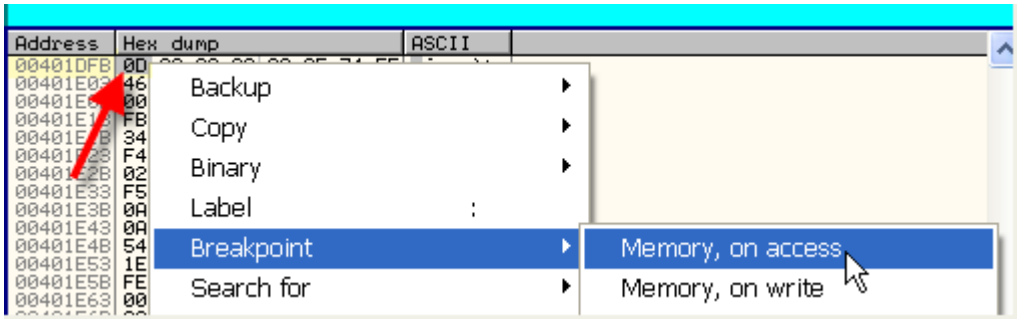


在堆栈窗口中看到 12F454 地址处保存了我们输入的用户名。

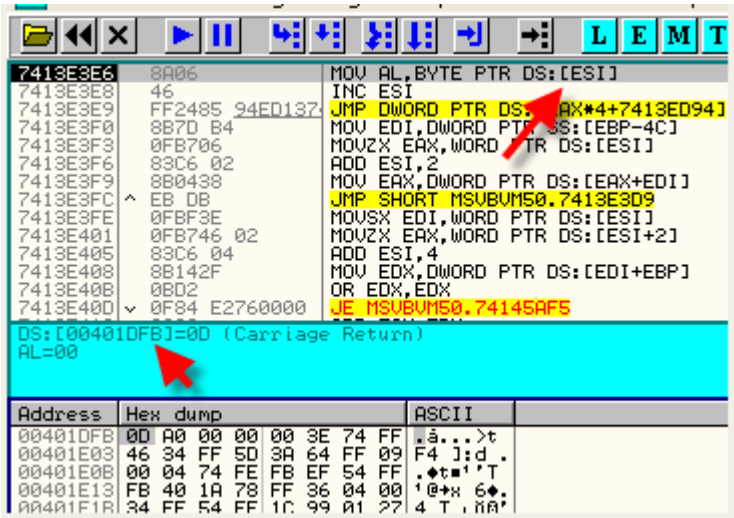


至此我们就定位到输入的用户名,接下来就是要定位输入的序列号,同理,给下图中的操作码设置内存访问断点。

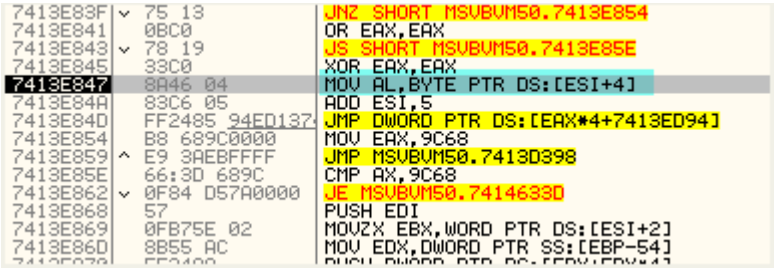
```
401DFB: 0d VCallHresult              get__ipropTEXTEDIT
```



运行起来,马上就断在读取该操作码的指令。



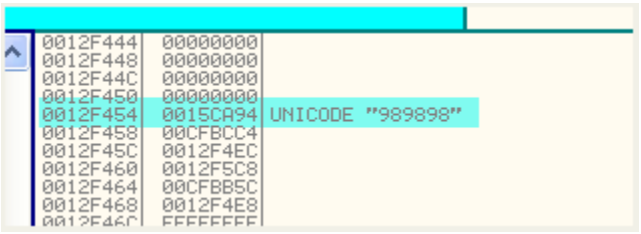
跟前面一样,跟踪到读取下一个操作码的指令处为止。



我们来看看输入的序列号是不是也被保存到了局部变量中。

```
401DFB: 0d VCallHresult              get__ipropTEXTEDIT
401E00: 3e FLdZeroAd                 local_008C
```

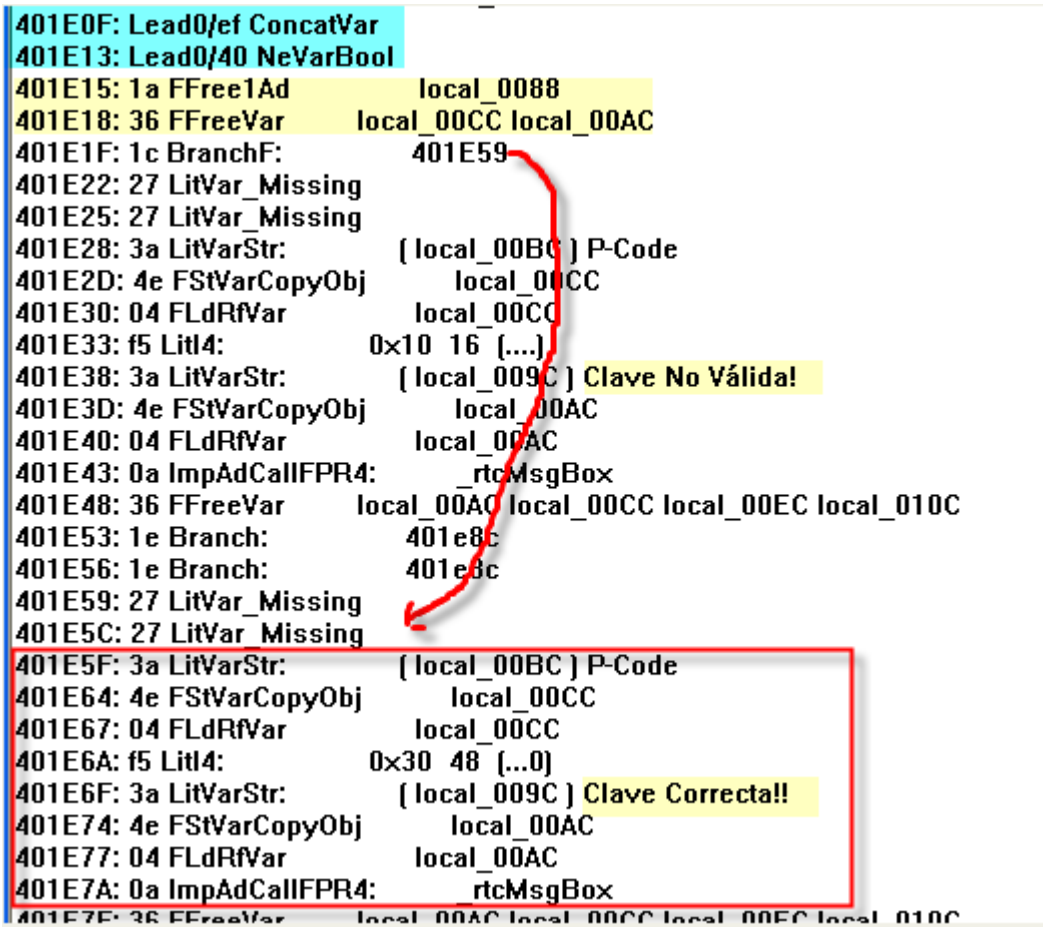
我们会发现跟之前一样输入的序列号也被保存到了 EBP - 8C 中。



至此我们又定位到了输入的序列号,我们不必跟踪每个操作码,只需要对关键的操作码设置内存访问断点,上一章,我们跟踪了每个操作码的执行过程,让大家可以更好的理解 P-CODE 的运行机制,本章的话,我们就没有像上一章那样赘述了,下面的跟踪步骤还是像刚刚那样定位关键点即可。

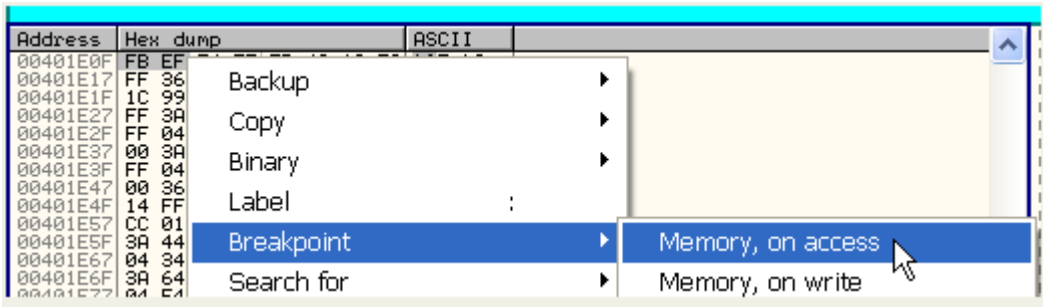
```
401E0F: Lead0/ef ConcatVar
401E13: Lead0/40 NeVarBool
401E15: 1a FFree1Ad                 local_0088
401E18: 36 FFreeVar          local_00CC local_00AC
401E1F: 1c BranchF:                 401E59
```

这里貌似在进行比较,接着释放局部变量的内存空间,然后根据刚刚比较的结果来决定是跳转到 401E59 处调用 rtcMsgBox 弹出正确序列号提示框还是直接往下执行弹出错误序列号的提示框。

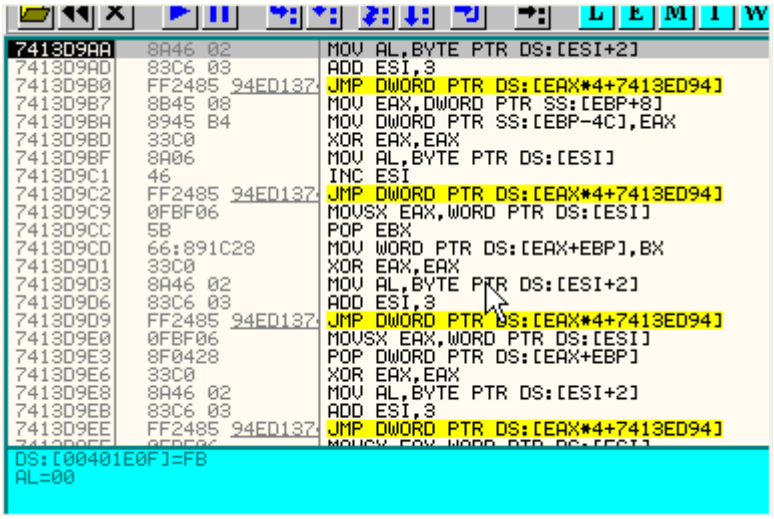


根据 ExDec 显示的内容来看这里很可能是进行序列号的比较,然后根据比较的结果来决定是条件跳转到提示序列号正确的消息框处还是提示序列号错误的消息框,所以我们可以给下图中的两个操作码设置内存访问断点,看看会发生什么。

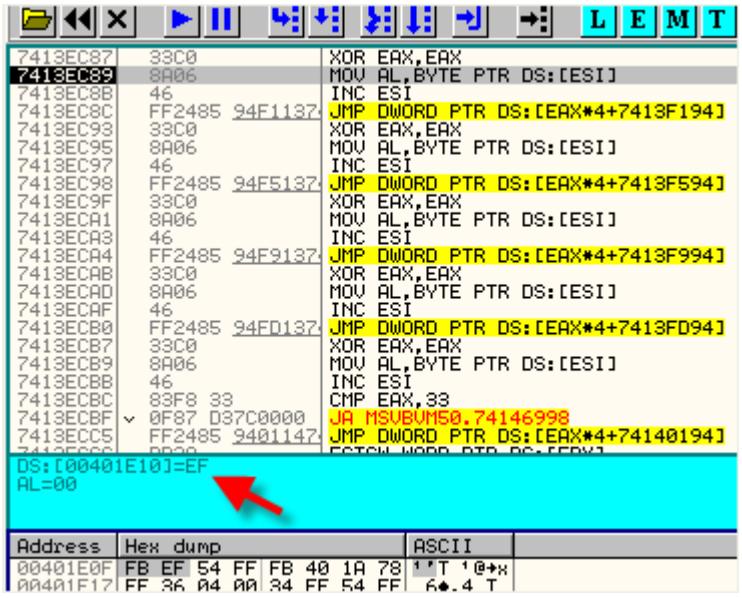
```
401E0F: Lead0/ef ConcatVar
401E13: Lead0/40 NeVarBool
```



运行起来。



断在了读取第一个操作码的指令处,继续往下跟踪直到读取第二个操作码的指令处为止,然后看看执行些什么操作。



第二个操作码是 EF。

接着查看一下操作码列表中关于 FB EF 的说明(见附件中的 OPCODES.TXT)。

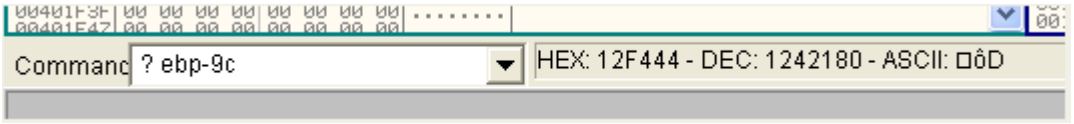
```
FB EF 6BAB 25AD 2
FB F0 6B99 259B 0          vbaStrCat
FB F1 C423 B981 0          Push [FC0D134]
```

操作码列表中并没有对 FB EF 进行相应的解释,但是根据 ExDec 中显示的内容 ConcatVar 字面意思可以理解为拼接变量值,一起来

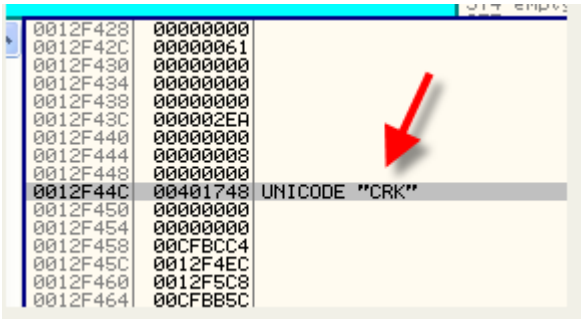
看一看 ExDec 中的描述。

```
401E07: 3a LitVarStr:                ( local_009C ) CRK
401E0C: 04 FLdRfVar                  local_018C
401E0F: Lead0/ef ConcatVar
401E13: Lead0/40 NeVarBool
```

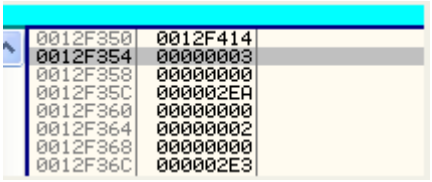
可以看到两个局部变量将进行拼接,其中一个是 EBP - 9C 即字符串“CRK”,另一个是 EBP - 018C,接下来我们分别定位到这两个变量。



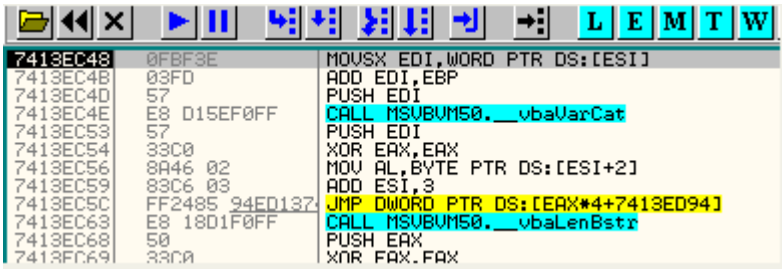
第一个变量是 EBP - 9C,在我机器上为 12F444。(PS:这里需要说明一下,这里的变量类型属于 variant 型,也是就传说中的变体类型,相信学习过 COM 组件的童鞋一定不会陌生,VB 中的 variant 类型属于一种结构体,该结构体的前两个字节表示类型,后面有 3 个 WORD 是保留的,接下来才是其真正的值。关于 variant 的类型定义这里大家可以看 jjnet 大哥在以前的帖子中给出的一段定义,见附录),前两个字节为 8,表示类型 8(b_str),也就是字符串类型,其实际指向的字符串首地址为 12F44C。



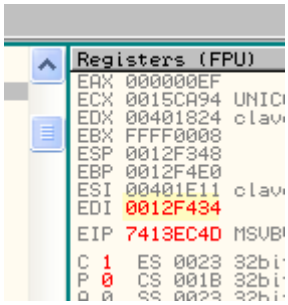
接下来我们看下一个变量 EBP - 18C,也就是 12F354,属于类型 3(I4),即占 4 字节的整型数,其真实值为 2EA。



我们跟进这个操作码中。

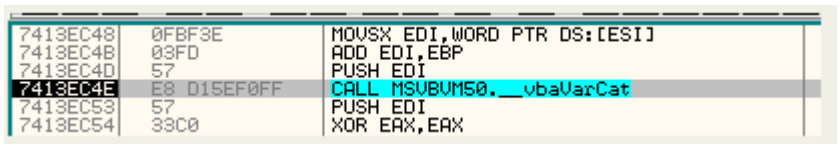


这里是读取参数并保存到 EDI 中。

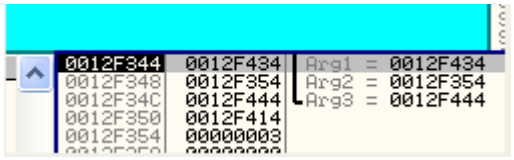


我们可以看到保存到 EDI 中参数值为 12F434。

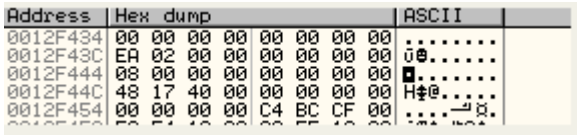
接着我们到了__vbaVarCat 这个 API 函数的调用处,堆栈中显示该函数有三个参数。



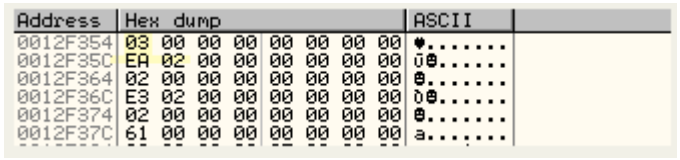
我们来看看每个参数的具体情况。



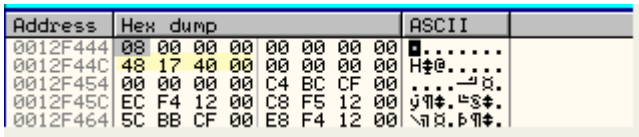
这是第一个参数。



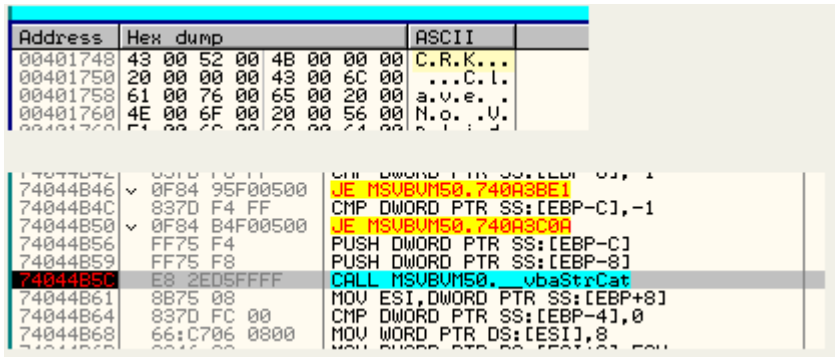
这是第二个参数,表示一个整型数,数值为 02EA(十六进制)。



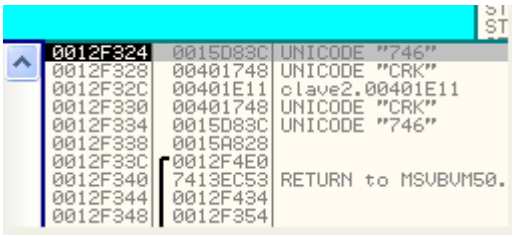
接着是第三个参数,表示一个字符串,其实际指向的字符串首地址为 401748,即 CRK。



好了,现在我们已经弄清楚了这几个参数的情况,接下来我们跟进到__vbaVarCat 这个 API 函数内部,看到其内部调用的一些其他的 API 函数,就会明白拼接过程是如何实现的了。

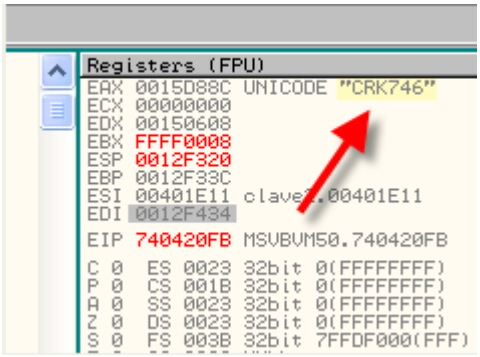


好,这里我们跟到了__vbaStrCat 的调用处,从堆栈中可以看出将要进行拼接的两个字符串分别是 CRK 和 746,而之前的那个数值 02EA 呢?

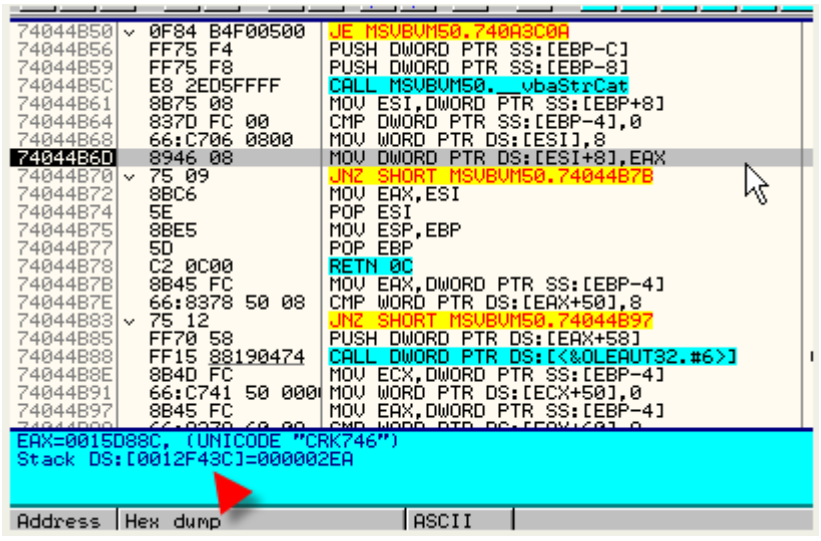


02EA(十六进制)对应的十进制数值如下:

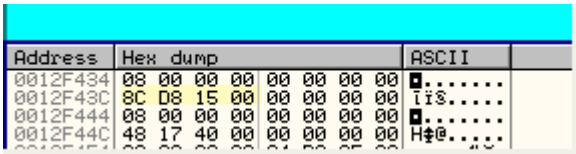
正好是 764,所以__vbaVarCat 这个函数首先会将数字型变量转化为了字符串(PS:十六进制数值转化为十进制数值),我们跟踪到该函数的 RET 处。



我们可以看到两个变量被拼接到了一起。

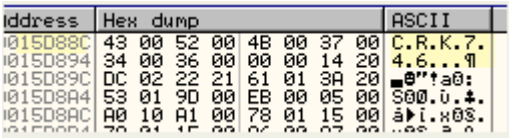


和前面一样,我们继续往下跟直到下一个操作码读取第一个参数为止。

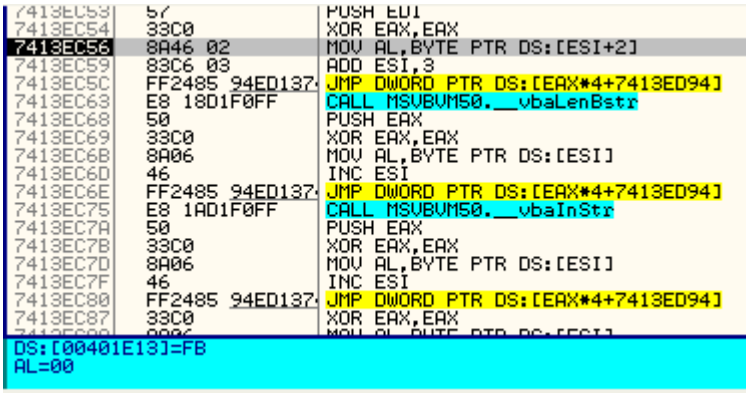


这里我们可以看到第一个参数的前两个字节值为 8,表示类型 8(b_str),也就是字符串类型,其指向的字符串首地址为 15D88C,这里

我们可以看到就是刚刚拼接的字符串。

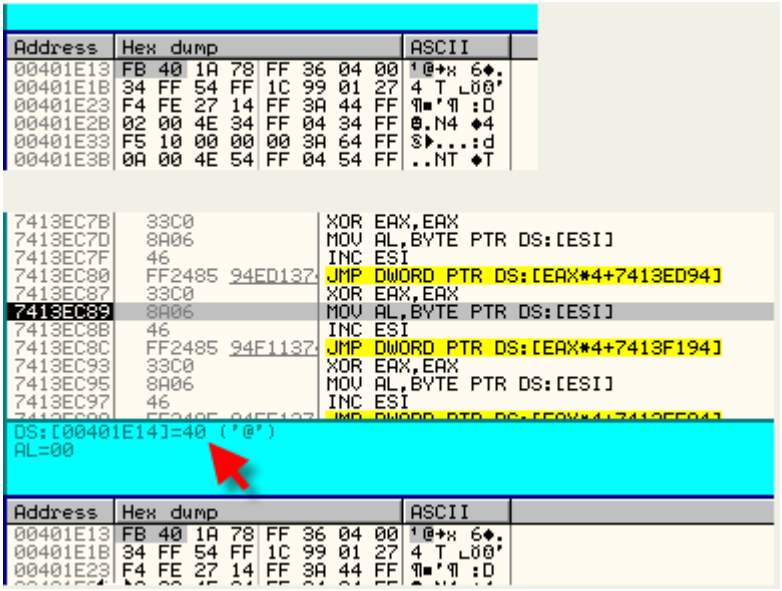


好了,接下来马上要进行序列号的比较了,我们往下跟直到读取下一个操作码为止。

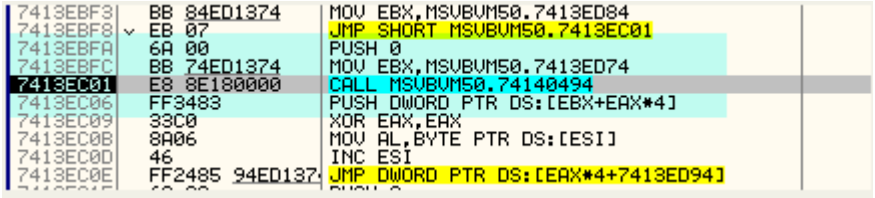



```
401E13: Lead0/40 NeVarBool
```

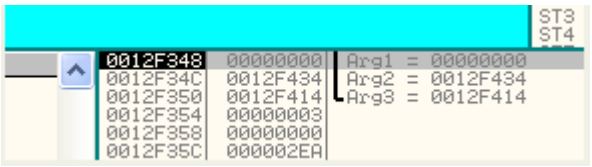
这里是一个双操作码操作,继续往下跟直到读取 FB40 的第二个操作码为止。



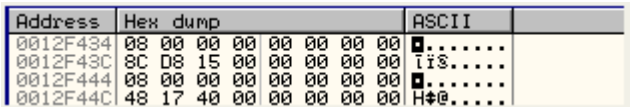
这里操作码列表中也没有 FB 40 解释,我们继续跟踪看看该操作码都干了些什么,嘿嘿。



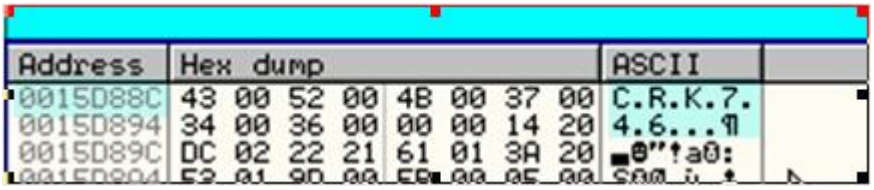
这里我们可以看到该操作码快结束的地方有一个 CALL 指令,我们来看看这个 CALL 的参数。



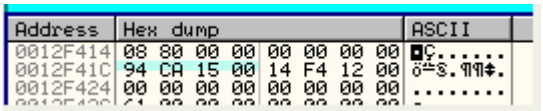
第一个参数为零,第二个参数为 12F434,在数据窗口中定位到 12F434。



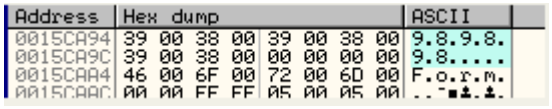
前两个字节值为 8,表示类型 8(b_str),即字符串类型,其实际指向的字符串首地址为 15D88C。



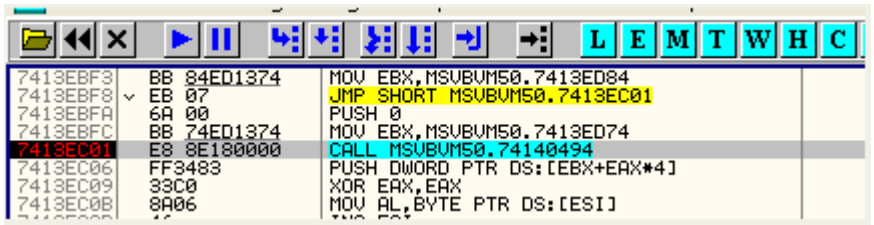
接下来我们在数据窗口中定位到另一个参数。



前两个字节为 8008,表示组合类型 8000 | 8(VT_RESERVED | VT_BSTR),即字符串型和保留类型的组合,实际指向的字符串首地址为 15CA94,也就是输入的序列号的首地址。

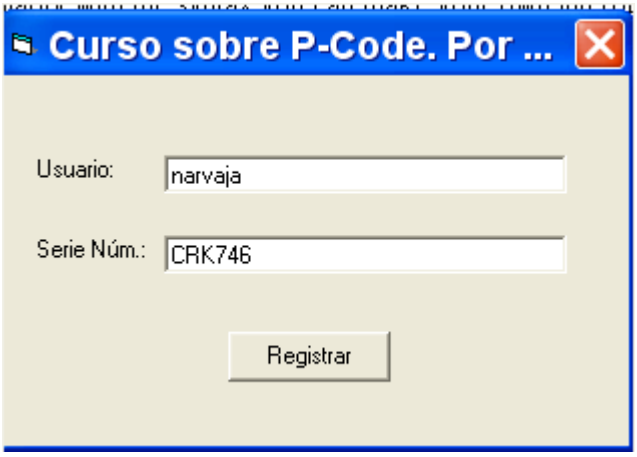


这个 CALL,OD 中没有解析其函数名称,我猜测可能是比较以上两个字符串。



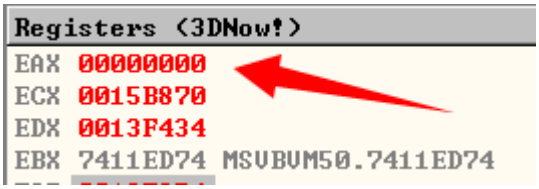
我们在这个 CALL 这里设置一个断点,接着按 F8 单步执行这个 CALL。

可以看到 EAX 的结果为 FFFFFFFF(PS:这里作者描述有误,该函数的结果是保存在 EAX 中,而并非堆栈中),很可能说明刚刚比较的两个字符串不相等,这里我们直接输入正确的序列号,接着按注册按钮看看会不会断在刚刚设置的这个断点处。

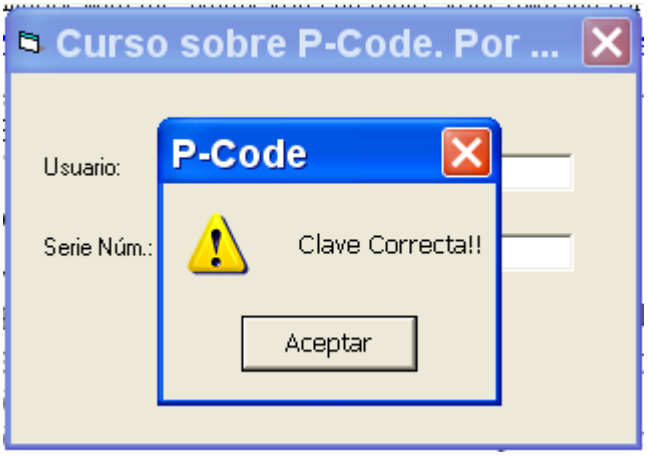


单击 Registrar 按钮。

断了下来,我们直接按 F8 键执行这个 CALL。



我们可以看到比较的结果为零(PS:这里这个函数的返回值是保存在EAX中,并非栈顶,作者描述有误,现已更正),说明两个字符串相等。



正如大家所看到的,我们没有必要从头到尾把整个程序跟一遍,我们只需要观察一下有没有什么关键的操作码,直接跟关键的操作码就能够轻松的找到正确的序列号。

下面是该 CrackMe 的详细的操作码注释清单:

```
401CC0: 04 FLdRfVar                local_008C
401CC3: 21 FLdPrThis               ;[SR] = [stack2]
401CC4: 0f VCallAd                 text          ;获取窗口句柄
401CC7: 19 FStAdFunc               local_0088
401CCA: 08 FLdPr                   local_0088
401CCD: 0d VCallHresult            get__ipropTEXTEDIT   ;读取文本框中的内容
401CD2: 6c ILdRf                   local_008C           ;读取到的文本
401CD5: 1b LitStr:                 &                    ;将字符串压入堆栈
401CD8: Lead0/30 EqStr                                  ;比较两个字符串
401CDA: 2f FFree1Str               local_008C
401CDD: 1a FFree1Ad                local_0088
401CE0: 1c BranchF:                401CE6               ;如果不相等则跳转
401CE3: 1e Branch:                 401e8c               ;无条件跳转
401CE6: 04 FLdRfVar                local_008C
401CE9: 21 FLdPrThis
```

```
401CEA: 0f VCallAd                  text                    ;获取窗口句柄
401CED: 19 FStAdFunc                local_0088
401CF0: 08 FLdPr                    local_0088
401CF3: 0d VCallHresult             get__ipropTEXTEDIT      ;读取文本框中的内容
401CF8: 6c ILdRf                    local_008C              ;读取到的文本
401CFB: 4a FnLenStr
401CFC: f5 LitI4:                   0x6    6    (....)      ;将占 4 个字节的立即数压入堆栈
401D01: d1 LtI4                                             ;是否小于(?)
401D02: 2f FFree1Str                local_008C
401D05: 1a FFree1Ad                 local_0088
401D08: 1c BranchF:                 401D3F                  ;如果不成立则跳转(>=6)
401D0B: 27 LitVar_Missing
401D0E: 27 LitVar_Missing
401D11: 3a LitVarStr:               ( local_00BC ) P-Code
401D16: 4e FStVarCopyObj            local_00CC
401D19: 04 FLdRfVar                 local_00CC
401D1C: f5 LitI4:                   0x40    64    (...@)
401D21: 3a LitVarStr:               ( local_009C ) Minimum 6 characters
401D26: 4e FStVarCopyObj            local_00AC
401D29: 04 FLdRfVar                 local_00AC
401D2C: 0a ImpAdCallFPR4:           _rtcMsgBox
401D31: 36 FFreeVar                 local_00AC local_00CC local_00EC local_010C
401D3C: 1e Branch:                  401e8c          ;如果小于 6 个字符则跳转
401D3F: 04 FLdRfVar                 local_008C
401D42: 21 FLdPrThis
401D43: 0f VCallAd                  text
401D46: 19 FStAdFunc                local_0088
401D49: 08 FLdPr                    local_0088
401D4C: 0d VCallHresult             get__ipropTEXTEDIT      ;读取文本框中的内容
401D51: 3e FLdZeroAd                local_008C              ;读取到的文本
401D54: 46 CVarStr                  local_00AC
401D57: 04 FLdRfVar                 local_00CC
401D5A: 0a ImpAdCallFPR4:           _rtcLowerCaseVar        ;转化为小写字母
401D5F: 04 FLdRfVar                 local_00CC
401D62: 04 FLdRfVar                 local_00EC
401D65: 0a ImpAdCallFPR4:           _rtcTrimVar
401D6A: 04 FLdRfVar                 local_00EC
401D6D: Lead1/f6 FStVar             local_011C
401D71: 1a FFree1Ad                 local_0088
401D74: 36 FFreeVar                 local_00AC local_00CC
401D7B: 04 FLdRfVar                 local_011C
401D7E: Lead0/eb FnLenVar
401D82: Lead1/f6 FStVar             local_012C
401D86: 28 LitVarI2:                ( local_00BC ) 0x1    (1)
```

```
401D8B: 04 FLdRfVar                    local_013C
401D8E: 04 FLdRfVar                    local_012C
401D91: Lead3/68 ForVar:               (when done) 401DE0          ; FOR i=1 to m 开始循环
401D97: 28 LitVarI2:                   ( local_00AC ) 0x1    (1)
401D9C: 04 FLdRfVar                    local_013C
401D9F: Lead1/22 CI4Var
401DA1: 04 FLdRfVar                    local_011C
401DA4: 04 FLdRfVar                    local_00CC
401DA7: 0a ImpAdCallFPR4:              _rtcMidCharVar      ;截取字符串的部分字符
401DAC: 04 FLdRfVar                    local_00CC          ;...用户名
401DAF: Lead2/fe CStrVarVal            local_008C
401DB3: 0b ImpAdCallI2                 _rtcAnsiValueBstr   ;取字符的十六进制值
401DB8: 44 CVarI2                      local_00BC
401DBB: Lead1/f6 FStVar                local_016C
401DBF: 2f FFree1Str                   local_008C
401DC2: 36 FFreeVar            local_00AC local_00CC
401DC9: 04 FLdRfVar                    local_017C
401DCC: 04 FLdRfVar                    local_016C
401DCF: Lead0/94 AddVar                local_00AC
401DD3: Lead1/f6 FStVar                local_017C
401DD7: 04 FLdRfVar                    local_013C          ;设置循环变量
401DDA: Lead3/7e NextStepVar:          (continue) 401D97   ;循环
401DE0: 04 FLdRfVar                    local_017C
401DE3: 04 FLdRfVar                    local_012C
401DE6: Lead0/94 AddVar                local_00AC
401DEA: Lead1/f6 FStVar                local_018C
401DEE: 04 FLdRfVar                    local_008C
401DF1: 21 FLdPrThis
401DF2: 0f VCallAd                     text
401DF5: 19 FStAdFunc                   local_0088
401DF8: 08 FLdPr                       local_0088
401DFB: 0d VCallHresult                get__ipropTEXTEDIT  ;读取文本框中的内容
401E00: 3e FLdZeroAd                   local_008C          ;序列号
401E03: 46 CVarStr                     local_00CC
401E06: 5d HardType
401E07: 3a LitVarStr:                  ( local_009C ) CRK
401E0C: 04 FLdRfVar                    local_018C
401E0F: Lead0/ef ConcatVar
401E13: Lead0/40 NeVarBool
401E15: 1a FFree1Ad                    local_0088
401E18: 36 FFreeVar            local_00CC local_00AC
401E1F: 1c BranchF:                    401E59
401E22: 27 LitVar_Missing
401E25: 27 LitVar_Missing
```

401E28: 3a LitVarStr: (local_00BC) P-Code

401E2D: 4e FStVarCopyObj local_00CC

401E30: 04 FLdRfVar local_00CC

401E33: f5 LitI4: 0x10 16 (....)

401E38: 3a LitVarStr: (local_009C) Key nonValid!

401E3D: 4e FStVarCopyObj local_00AC

401E40: 04 FLdRfVar local_00AC

401E43: 0a ImpAdCallFPR4: _rtcMsgBox

401E48: 36 FFreeVar local_00AC local_00CC local_00EC local_010C

401E53: 1e Branch: 401e8c

401E56: 1e Branch: 401e8c

401E59: 27 LitVar_Missing

401E5C: 27 LitVar_Missing

401E5F: 3a LitVarStr: (local_00BC) P-Code

401E64: 4e FStVarCopyObj local_00CC

401E67: 04 FLdRfVar local_00CC

401E6A: f5 LitI4: 0x30 48 (...0)

401E6F: 3a LitVarStr: (local_009C) Key Correct !

401E74: 4e FStVarCopyObj local_00AC

401E77: 04 FLdRfVar local_00AC

401E7A: 0a ImpAdCallFPR4: _rtcMsgBox

401E7F: 36 FFreeVar local_00AC local_00CC local_00EC local_010C

401E8A: Lead1/c8 End

401E8C: 13 ExitProcHresult

好了,那么这个 CrackMe 我们就搞定了...

现在大家应该知道了如何用 OllyDbg 来调试 P-CODE 应用程序了吧,不像以前,WKT 可能调试 P-CODE 程序很方便,但是现在很多 P-CODE 程序会检测 WKT,ExDec 等工具,所以我们就可以用 OD 配置好反反调试插件来分析。

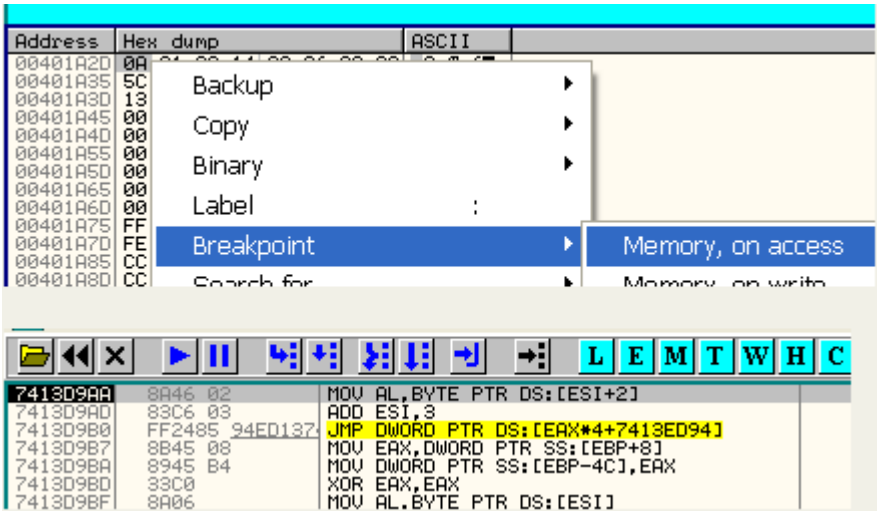
现在我们再来看一个名字叫做 nags1 的 CrackMe,我们需要剔除掉其 NAG 窗口,首先我们将其加载到 ExDec 看看显示了些什么。

```
Proc: 401a40
401A14: 27 LitVar_Missing
401A17: 27 LitVar_Missing
401A1A: 27 LitVar_Missing
401A1D: f5 LitI4:                   0x0  0  (....)
401A22: 3a LitVarStr:               ( local_0094 ) NAG
401A27: 4e FStVarCopyObj            local_00A4
401A2A: 04 FLdRfVar                 local_00A4
401A2D: 0a ImpAdCallFPR4:           _rtcMsgBox
401A32: 36 FFreeVar         local_00A4 local_00C4 local_00E4 local_0104
401A3D: 13 ExitProcHresult

Proc: 401958

401954: Lead1/c8 End
401956: 13 ExitProcHresult
```

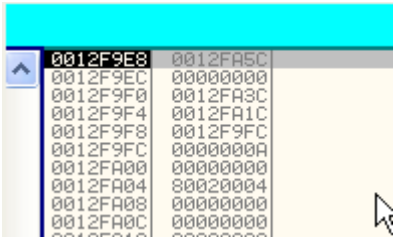
这里我们可以看到 NAG 窗口实际上仅仅是调用了 rtcMsgBox 弹出的一个消息框。但是 P-CODE 代码我们不能像常规的汇编指令那样直接 NOP 掉。

我们用 OllyDbg 加载该 CrackMe,然后给 rtcMsgBox 的操作码设置一个内存访问断点。

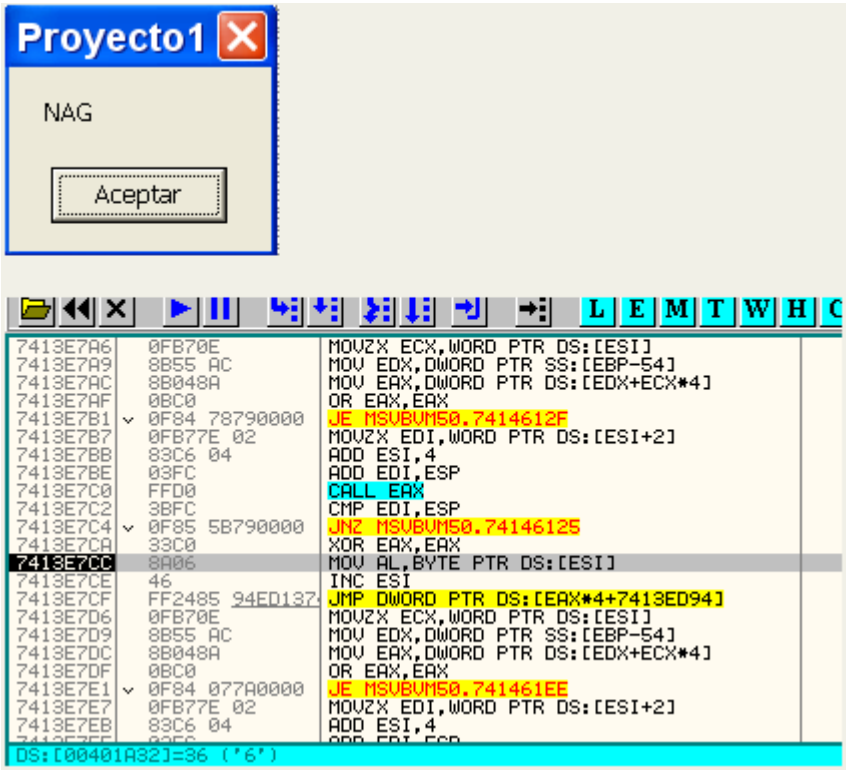
```
401A2D: 0a ImpAdCallFPR4:           _rtcMsgBox
```



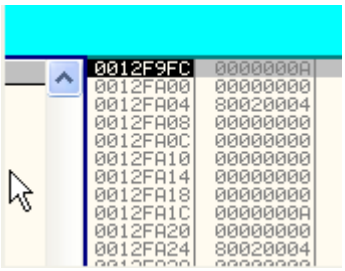
运行起来,马上就会断在了读取 rtcMsgBox 操作码的指令处。



当前栈顶指针指向的是 12F9E8,我们继续往下跟踪直到读取下一个操作码的指令处为止,但是还没到读取下一个操作码的指令处,NAG 窗口就弹出来了,我们单击 Aceptar(OK)按钮。



现在我们终于跟到读取下一个操作码的指令处,刚刚跟踪的过程中,如果你足够留心的话,你就会发现是执行了上面的 CALL EAX 指令弹出的 NAG 窗口,这个 CALL 其实就是调用 rtcMsgBox 这个 API 函数,当前栈顶指针指向的是 12F9FC。



这里为了不让该程序调用 rtcMsgBox 弹出 NAG 窗口,我们可以用别的操作码来代替这个 0A 操作码,这里我就用 F5 这个操作码来替换。

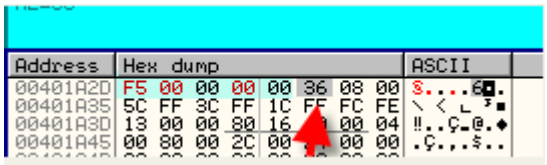
```
F5     5CBE 1377 4 1 4        Push imm#4
```

F5 表示 PUSH imm#4,说明只有一个参数且该参数占 4 个字节,刚好可以与 0A 这个操作码的参数所占的大小匹配上。

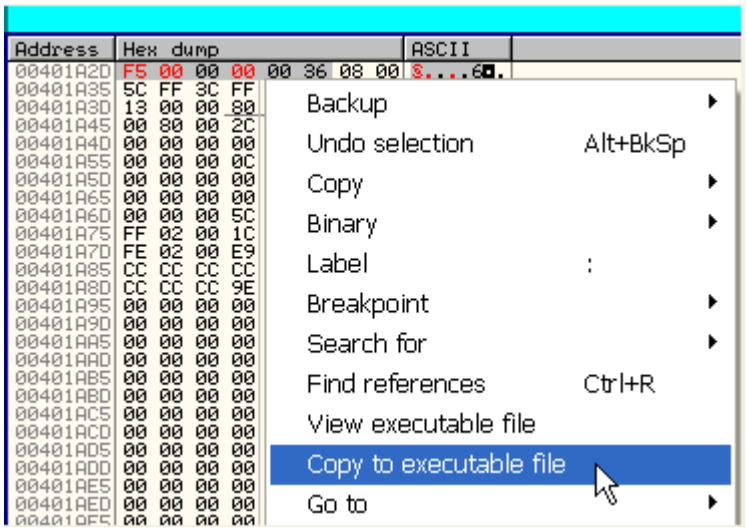
```
0A     664E 1F30 4 2 2 2
```

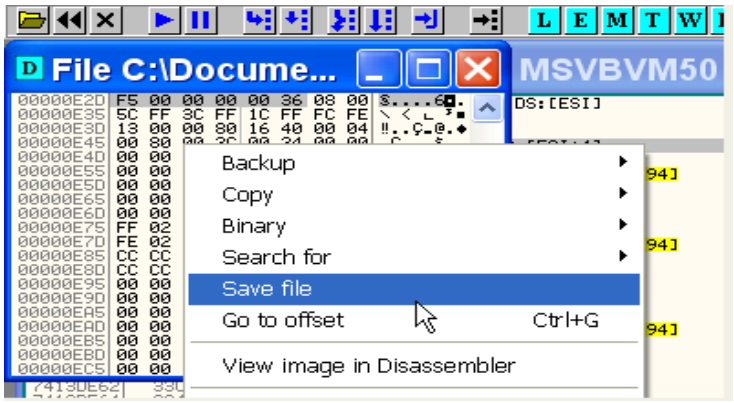
我们可以看到 0A 操作码有两个参数,每个参数占两个字节,所以两个参数总共占 4 个字节。

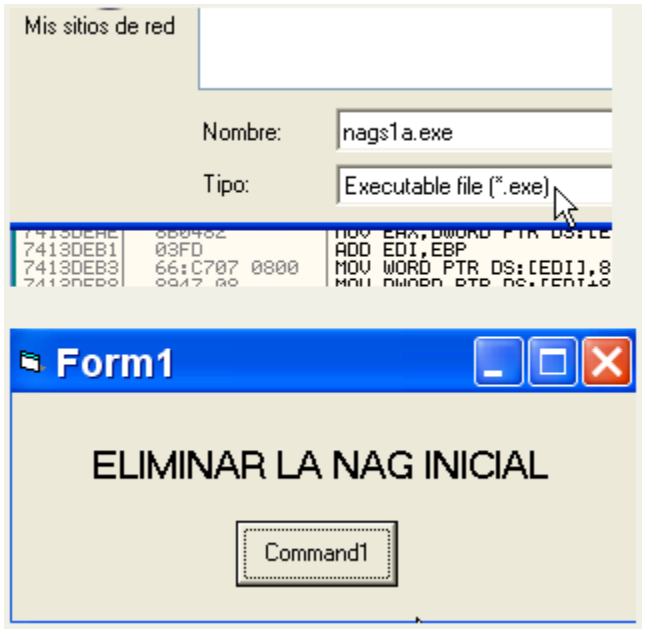
这里我们就将 0A 替换成了 F5,然后将参数值设置为零,即将零压入堆栈。



根据 ExDec 显示的信息我们知道下一个操作码为 36,我们替换操作码的时候必须保证新的操作码与原操作码参数所占大小相等,如果参数大小不匹配的话,那么替换以后,程序很可能会出错,(PS:你可能联想一下我们以前常讲到的的堆栈平衡,当然这里跟堆栈平衡是两码事,我只是打个比喻而已,嘿嘿),接下来我们将刚刚所做的修改保存到文件。







我们直接运行起来,可以看到直接弹出了主窗口,并没有出现 NAG 窗口,说明我们修改成功了,这里你也可以使用其他的操作码来替换,给大家留一个练习的小程序 nag2,大家可以自行尝试剔除掉 NAG 窗口。

至此,我们关于 P-CODE 的内容就介绍完了,下一章我们开始介绍脱壳。

附录:

Variant 结构体的定义,变体的类型

```
enum VARENUM
    {  VT_EMPTY  = 0,
  VT_NULL  = 1,
  VT_I2  = 2,
  VT_I4  = 3,
  VT_R4  = 4,
  VT_R8  = 5,
  VT_CY  = 6,
  VT_DATE  = 7,
  VT_BSTR  = 8,
  VT_DISPATCH  = 9,
  VT_ERROR  = 10,
  VT_BOOL  = 11,
  VT_VARIANT  = 12,
  VT_UNKNOWN  = 13,
  VT_DECIMAL  = 14,
  VT_I1  = 16,
  VT_UI1  = 17,
  VT_UI2  = 18,
  VT_UI4  = 19,
  VT_I8  = 20,
  VT_UI8  = 21,
  VT_INT  = 22,
  VT_UINT  = 23,
  VT_VOID  = 24,
  VT_HRESULT  = 25,
  VT_PTR  = 26,
  VT_SAFEARRAY  = 27,
  VT_CARRAY  = 28,
  VT_USERDEFINED  = 29,
  VT_LPSTR  = 30,
  VT_LPWSTR  = 31,
  VT_RECORD  = 36,
  VT_INT_PTR  = 37,
  VT_UINT_PTR  = 38,
  VT_FILETIME  = 64,
  VT_BLOB  = 65,
  VT_STREAM  = 66,
  VT_STORAGE  = 67,
  VT_STREAMED_OBJECT  = 68,
  VT_STORED_OBJECT  = 69,
  VT_BLOB_OBJECT  = 70,
  VT_CF  = 71,
  VT_CLSID  = 72,
  VT_VERSIONED_STREAM  = 73,
  VT_BSTR_BLOB  = 0xfff,
  VT_VECTOR  = 0x1000,
  VT_ARRAY  = 0x2000,
  VT_BYREF  = 0x4000,
  VT_RESERVED  = 0x8000,
  VT_ILLEGAL  = 0xffff,
  VT_ILLEGALMASKED  = 0xfff,
  VT_TYPEMASK  = 0xfff
    } ;
```

```
struct tagVARIANT
    {
    union
        {
        struct __tagVARIANT
            {
            VARTYPE vt;
            WORD wReserved1;
            WORD wReserved2;
            WORD wReserved3;
            union
                {
                LONGLONG llVal;
                LONG lVal;
                BYTE bVal;
                SHORT iVal;
                FLOAT fltVal;
                DOUBLE dblVal;
                VARIANT_BOOL boolVal;
                _VARIANT_BOOL bool;
                SCODE scode;
                CY cyVal;
                DATE date;
                BSTR bstrVal;
                IUnknown *punkVal;
                IDispatch *pdispVal;
                SAFEARRAY *parray;
                BYTE *pbVal;
                SHORT *piVal;
                LONG *plVal;
                LONGLONG *pllVal;
                FLOAT *pfltVal;
                DOUBLE *pdblVal;
                VARIANT_BOOL *pboolVal;
                _VARIANT_BOOL *pbool;
                SCODE *pscode;
                CY *pcyVal;
                DATE *pdate;
                BSTR *pbstrVal;
                IUnknown **ppunkVal;
                IDispatch **ppdispVal;
                SAFEARRAY **pparray;
                VARIANT *pvarVal;
                PVOID byref;
                CHAR cVal;
                USHORT uiVal;
                ULONG ulVal;
                ULONGLONG ullVal;
                INT intVal;
                UINT uintVal;
                DECIMAL *pdecVal;
                CHAR *pcVal;
                USHORT *puiVal;
                ULONG *pulVal;
                ULONGLONG *pullVal;
                INT *pintVal;
                UINT *puintVal;
                struct __tagBRECORD
                    {
                    PVOID pvRecord;
                    IRecordInfo *pRecInfo;
                    }    __VARIANT_NAME_4;
                }    __VARIANT_NAME_3;
            }    __VARIANT_NAME_2;
        DECIMAL decVal;
        }    __VARIANT_NAME_1;
    } ;
```