

## 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

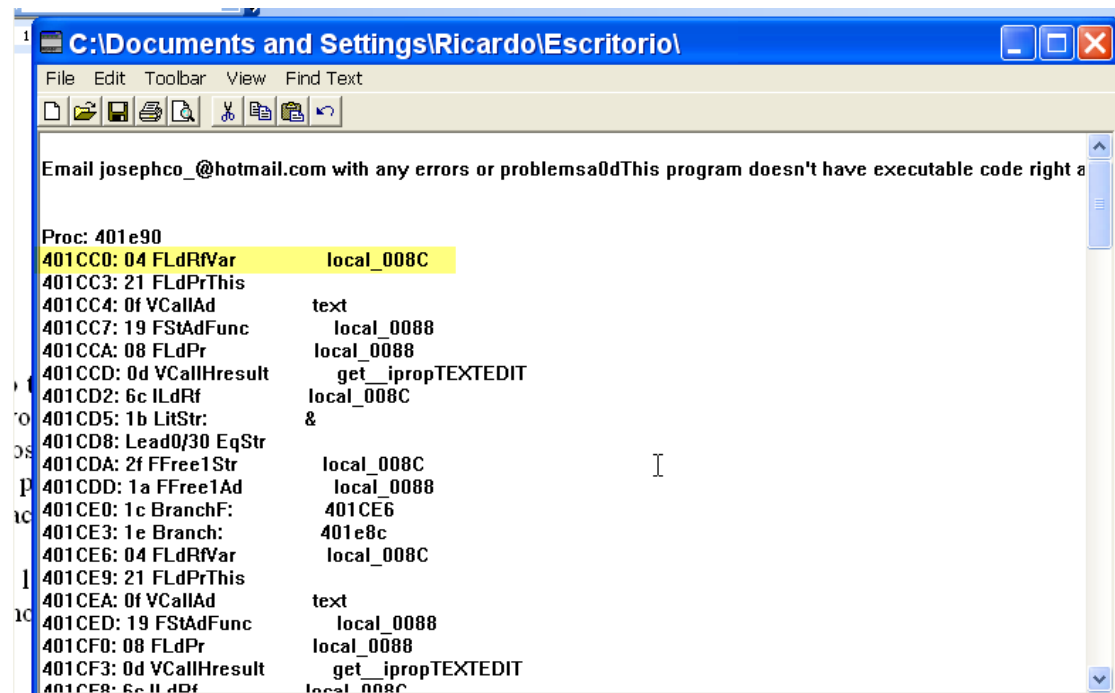
a9 → AddI2 栈顶两个 WORD 型元素相加,相加的结果置于栈顶

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。

**\* - [CPU - main thread, module clave2]**

Address	Disassembly	Comment
00401044	PUSH clave2.00401258	
00401049	CALL <JMP.&MSUBUM50.#100>	
0040104E	ADD BYTE PTR DS:[EAX],AL	
00401050	ADD BYTE PTR DS:[EAX],AL	
00401052	ADD BYTE PTR DS:[EAX],AL	
00401054	XOR BYTE PTR DS:[EAX],AL	
00401056	ADD BYTE PTR DS:[EAX],AL	
00401058	CMP BYTE PTR DS:[EAX],AL	
0040105A	ADD BYTE PTR DS:[EAX],AL	
0040105C	ADD BYTE PTR DS:[EAX],AL	
0040105E	ADD BYTE PTR DS:[EAX],AL	
00401060	ADD ECX,EDX	
00401062	CMP DWORD PTR DS:[EBX],EAX	
00401064	ADC BYTE PTR DS:[EDI-2A],CL	
00401067	ADC DWORD PTR DS:[EAX+EDX+4AFA88E7],EBX	

我们定位到入口点上面的 API 函数 MethCallEngine。

Address	Disassembly	Comment
00401000	JMP DWORD PTR DS:[&MSUBUM50.#595]	MSUBUM50.rtcMsgBox
00401006	JMP DWORD PTR DS:[&MSUBUM50.#518]	MSUBUM50.rtcLowerCaseVar
0040100C	JMP DWORD PTR DS:[&MSUBUM50.#520]	MSUBUM50.rtcTrinVar
00401012	JMP DWORD PTR DS:[&MSUBUM50.#632]	MSUBUM50.rtcMidCharVar
00401018	JMP DWORD PTR DS:[&MSUBUM50.#516]	MSUBUM50.rtcAnsIValueBstr
0040101E	JMP DWORD PTR DS:[&MSUBUM50.#vbaExcept	MSUBUM50.vbaExceptionHandler
00401024	JMP DWORD PTR DS:[&MSUBUM50.EVENT_SINK	MSUBUM50.EVENT_SINK_QueryInterface
0040102A	JMP DWORD PTR DS:[&MSUBUM50.EVENT_SINK	MSUBUM50.EVENT_SINK_AddRef
00401030	JMP DWORD PTR DS:[&MSUBUM50.EVENT_SINK	MSUBUM50.EVENT_SINK_Release
00401036	JMP DWORD PTR DS:[&MSUBUM50.MethCallEn	MSUBUM50.MethCallEngine
0040103C	JMP DWORD PTR DS:[&MSUBUM50.#100]	MSUBUM50.ThunRTMain
00401042	ADD BYTE PTR DS:[EAX],AL	
00401044	PUSH clave2.00401258	
00401049	CALL <JMP.&MSUBUM50.#100>	
0040104E	ADD BYTE PTR DS:[EAX],AL	

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。

Address	Disassembly	Comment
74142FFC	SUB DWORD PTR SS:[ESP+4],EAX	
74143000	MOV ECX,MSUBUM50.7413E302	
74143005	JMP MSUBUM50.7413D243	
7414300A	PUSH EBP	
7414300B	MOV EBP,ESP	
7414300D	PUSH DWORD PTR SS:[ESP+8]	
74143011	PUSH DWORD PTR SS:[EBP+C]	
74143014	CALL MSUBUM50.741421DD	
74143019	PUSH 0	
7414301B	LEA EDX,DWORD PTR SS:[EBP+14]	

我们运行起来看看会不会触发刚刚设置的断点。

**Curso sobre P-Code. Por ...**

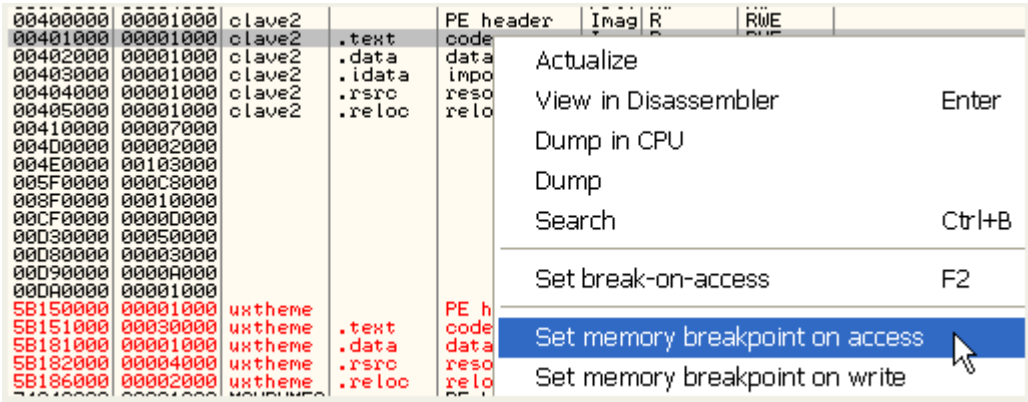
Usuario:

Serie Núm.:

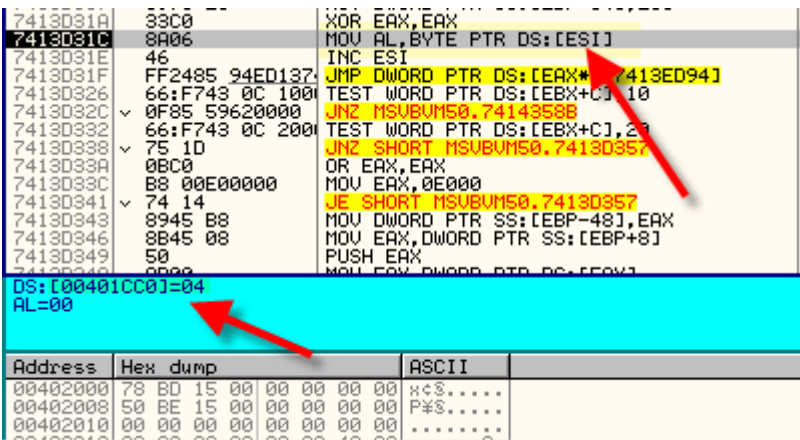
我们可以看到弹出了注册窗口,但是并没有触发我们设置的断点,说明在执行 P-CODE 之前注册窗口就产生了,可能有的程序执行 PCODE 在窗口产生之前,而我们这里刚好相反,其实这无关紧要,现在我们随便输入一个错误的用户名和序列号。

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。



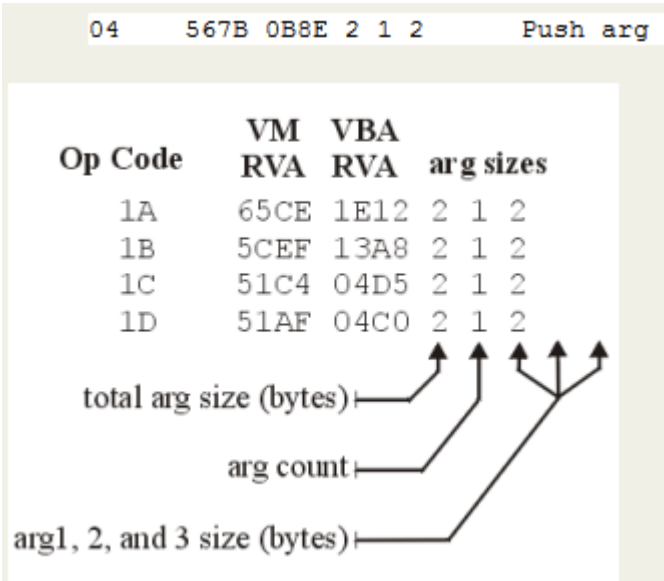
这里我们可以看到 ESI 指向了第一个操作码,并且该操作码将被保存到 AL 中。

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
```

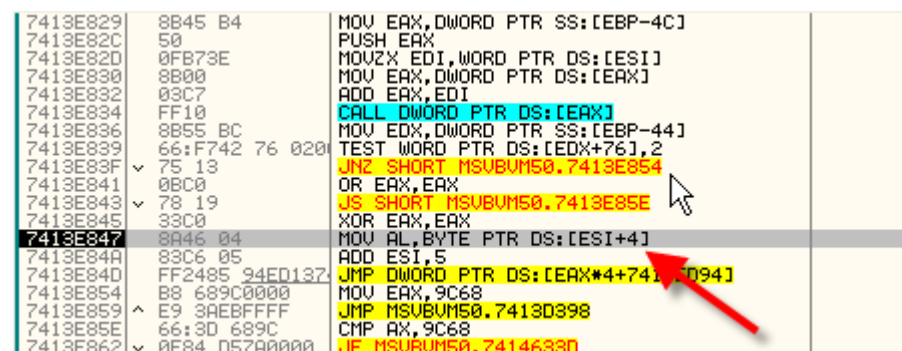
我们看到这两处 VCALLHresult, 都是读取文本框中用户输入的信息, 第一个有可能是读取用户输入的用户名, 第二个可能是读取用户输入的序列号, 我们直接给 401D4C 地址处的操作码设置内存访问断点。



接着我们运行起来。



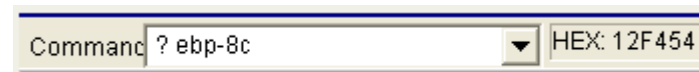
断了下来, 继续往下跟踪直到读取下一个操作码的指令为止。



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。

0012F44C	00000000	
0012F450	00000000	
0012F454	0015CA94	Unicode "narvaja"
0012F458	00CFBE1C	
0012F45C	0012F4EC	
0012F460	0012F5C8	
0012F464	00CFB85C	
0012F468	0012F4E8	

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

Address	Hex dump	ASCII
00401DFB	00 00 00 00 00 00 00 00	
00401E03	46 34 FF 5D 3A 64 FF 09	F4 l;d.
00401E0B	00 04 74 FE FB EF 54 FF	.t' T
00401E13	FB 40 1A 78 FF 36 04 00	'@+x 6.
00401E1B	34 FF 54 FF 1C 99 01 27	4 T , nA'

运行起来,马上就断在读取该操作码的指令。

7413E3E6	8A06	MOV AL, BYTE PTR DS:[ESI]
7413E3E8	46	INC ESI
7413E3E9	FF2485 94ED137	JMP DWORD PTR DS:[EAX*4+7413ED94]
7413E3F0	8B7D B4	MOV EDI, DWORD PTR SS:[EBP-4C]
7413E3F3	0FB706	MOVZX EAX, WORD PTR DS:[ESI]
7413E3F6	83C6 02	ADD ESI, 2
7413E3F9	8B0438	MOV EAX, DWORD PTR DS:[EAX+EDI]
7413E3FC	EB DB	JMP SHORT MSUBUM50.7413E3D9
7413E3FE	0FBF3E	MOVZX EDI, WORD PTR DS:[ESI]
7413E401	0FB746 02	MOVZX EAX, WORD PTR DS:[ESI+2]
7413E405	83C6 04	ADD ESI, 4
7413E408	8B142F	MOV EDX, DWORD PTR DS:[EDI+EBP]
7413E40B	0BD2	OR EDX, EDX
7413E40D	0F84 E2760000	JE MSUBUM50.74145AF5
DS:[00401DFB]=00 (Carriage Return)		
AL=00		

跟前面一样,跟踪到读取下一个操作码的指令处为止。

7413E83F	75 13	JNZ SHORT MSUBUM50.7413E854
7413E841	0BC0	OR EAX, EAX
7413E843	78 19	JS SHORT MSUBUM50.7413E85E
7413E845	33C0	XOR EAX, EAX
7413E847	8A46 04	MOV AL, BYTE PTR DS:[ESI+4]
7413E84A	83C6 05	ADD ESI, 5
7413E84D	FF2485 94ED137	JMP DWORD PTR DS:[EAX*4+7413ED94]
7413E854	B8 689C0000	MOV EAX, 9C68
7413E859	E9 3AE8FFFF	JMP MSUBUM50.7413D398
7413E85E	66:3D 689C	CMP AX, 9C68
7413E862	0F84 D57A0000	JE MSUBUM50.7414633D
7413E868	57	PUSH EDI
7413E869	0FB75E 02	MOVZX EBX, WORD PTR DS:[ESI+2]
7413E86D	8B55 AC	MOV EDX, DWORD PTR SS:[EBP-54]
7413E870	FF3400	PUSH DWORD PTR DS:[EBX+4]

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。

0012F444	00000000	
0012F448	00000000	
0012F44C	00000000	
0012F450	00000000	
0012F454	0015CA94	UNICODE "989898"
0012F458	00CFBCC4	
0012F45C	0012F4EC	
0012F460	0012F5C8	
0012F464	00CFB85C	
0012F468	0012F4E8	
0012F46C	FFFFFFFF	

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。

```

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 Litl4:              0x10 16 [...]
401E38: 3a LitVarStr:          [ local_009C ] Clave No Válida!
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 Litl4:              0x30 48 [...]
401E6F: 3a LitVarStr:          [ local_009C ] Clave Correcta!!
401E74: 4e FStVarCopyObj       local_00AC
401E77: 04 FLdRfVar            local_00AC
401E7A: 0a ImpAdCallFPR4:      _rtcMsgBox
401E7E: 36 FFreeVar             local_00AC local_00CC local_00EC local_010C

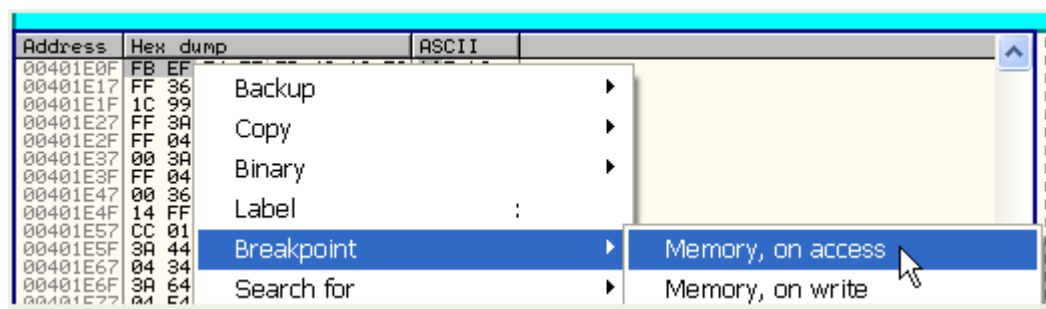
```

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。

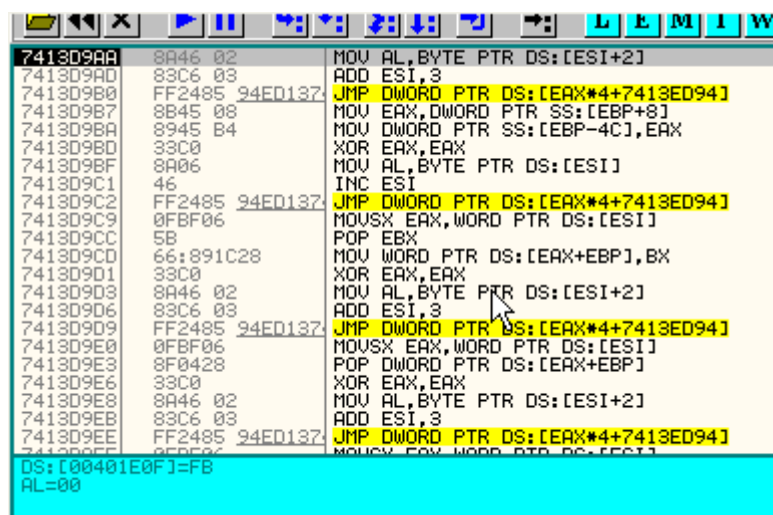
```

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

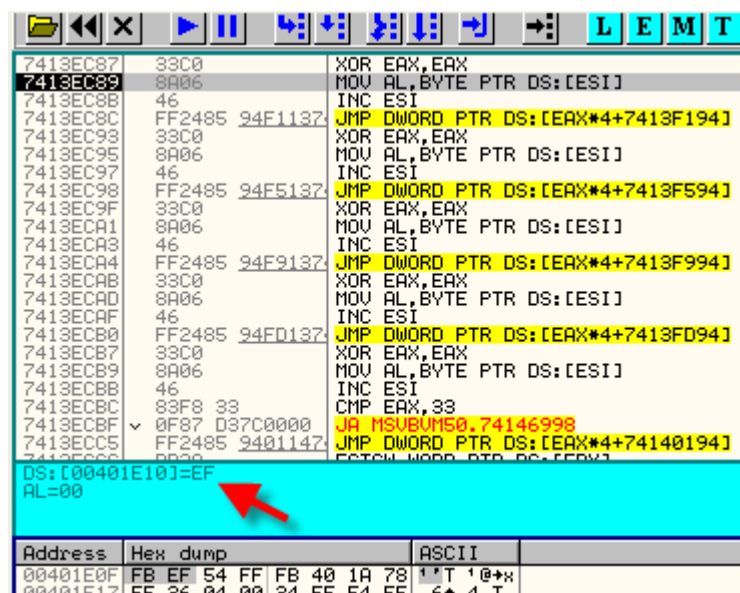
```



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]

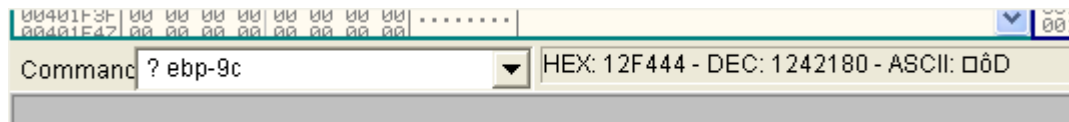
操作码列表中并没有对 FB EF 进行相应的解释,但是根据 ExDec 中显示的内容 ConcatVar 字面意思可以理解为拼接变量值,一起来



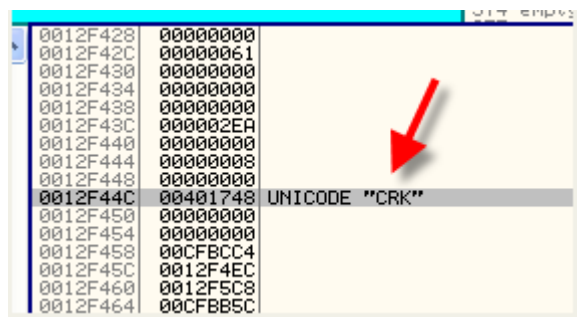
看一看 ExDec 中的描述。

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

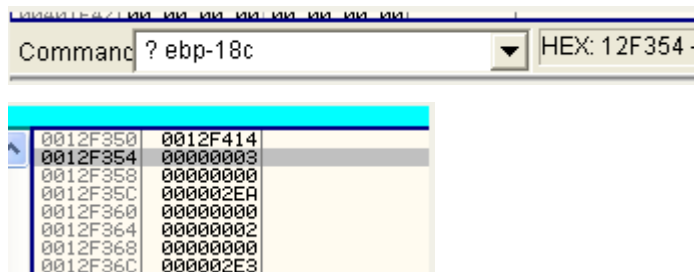
可以看到两个局部变量将进行拼接,其中一个 EBP - 9C 即字符串“CRK”,另一个是 EBP - 018C,接下来我们分别定位到这两个变量。



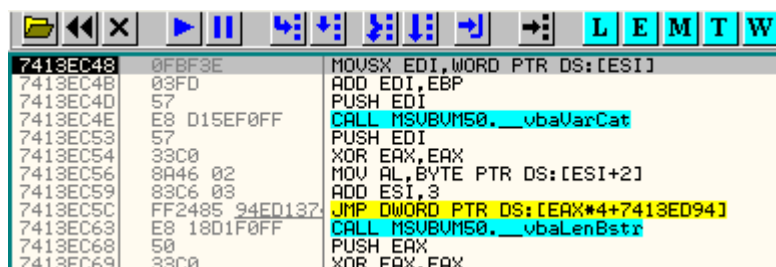
第一个变量是 EBP - 9C,在我机器上为 12F444。(PS:这里需要说明一下,这里的变量类型属于 **variant** 型,也就是传说中的变体类型,相信学习过 COM 组件的童鞋一定不会陌生,VB 中的 **variant** 类型属于一种结构体,该结构体的前两个字节表示类型,后面有 3 个 **WORD** 是保留的,接下来才是其真正的值。关于 **variant** 的类型定义这里大家可以看 [jjnet](#) 大哥在以前的帖子中给出的一段定义,见[附录](#),前两个字节为 8,表示类型 8(b\_str),也就是字符串类型,其实际指向的字符串首地址为 12F44C。



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。



Registers (FPU)		
EAX	000000EF	
ECX	0015CA94	UNIC
EDX	00401824	clav
EBX	FFFF0008	
ESP	0012F348	
EBP	0012F4E0	
ESI	00401E11	clav
EDI	0012F434	
EIP	7413EC4D	MSVB
C 1	ES 0023	32bi
P 0	CS 001B	32bi
D 0	SS 0023	32bi

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。

7413EC48	0FBF3E	MOVSX EDI,WORD PTR DS:[ESI]
7413EC4B	03FD	ADD EDI,EBP
7413EC4D	57	PUSH EDI
7413EC4E	E8 D1EF0FF	CALL MSVBUM50.__vbaVarCat
7413EC53	57	PUSH EDI
7413EC54	33C0	XOR EAX,EAX

我们来看看每个参数的具体情况。

0012F344	0012F434	Arg1 = 0012F434
0012F348	0012F354	Arg2 = 0012F354
0012F34C	0012F444	Arg3 = 0012F444
0012F350	0012F414	
0012F354	00000003	
0012F358	00000000	

这是第一个参数。

Address	Hex dump	ASCII
0012F434	00 00 00 00 00 00 00 00	.....
0012F43C	EA 02 00 00 00 00 00 00	00.....
0012F444	08 00 00 00 00 00 00 00	00.....
0012F44C	48 17 40 00 00 00 00 00	H0.....
0012F454	00 00 00 00 C4 BC CF 00	....-00.
0012F45C	00 00 00 00 00 00 00 00	.....

这是第二个参数,表示一个整型数,数值为 02EA(十六进制)。

Address	Hex dump	ASCII
0012F354	03 00 00 00 00 00 00 00	00.....
0012F35C	EA 02 00 00 00 00 00 00	00.....
0012F364	02 00 00 00 00 00 00 00	00.....
0012F36C	E3 02 00 00 00 00 00 00	00.....
0012F374	02 00 00 00 00 00 00 00	00.....
0012F37C	61 00 00 00 00 00 00 00	a.....

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。

Address	Hex dump	ASCII
0012F444	08 00 00 00 00 00 00 00	00.....
0012F44C	48 17 40 00 00 00 00 00	H0.....
0012F454	00 00 00 00 C4 BC CF 00	....-00.
0012F45C	EC F4 12 00 C8 F5 12 00	00000000
0012F464	5C BB CF 00 E8 F4 12 00	00000000

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。

Address	Hex dump	ASCII
00401748	43 00 52 00 48 00 00 00	C.R.K...
00401750	20 00 00 00 43 00 6C 00	...C.L.
00401758	61 00 76 00 65 00 20 00	a.v.e...
00401760	4E 00 6F 00 20 00 56 00	N.o..U.
00401768	00 00 00 00 00 00 00 00	.....
74044B46	0F84 95F00500	JE MSVBUM50.740A3BE1
74044B4C	837D F4 FF	CMP DWORD PTR SS:[EBP-C],-1
74044B50	0F84 B4F00500	JE MSVBUM50.740A3C0A
74044B56	FF75 F4	PUSH DWORD PTR SS:[EBP-C]
74044B59	FF75 F8	PUSH DWORD PTR SS:[EBP-8]
74044B5C	E8 2ED5FFFF	CALL MSVBUM50.__vbaStrCat
74044B61	8B75 08	MOV ESI,DWORD PTR SS:[EBP+8]
74044B64	837D FC 00	CMP DWORD PTR SS:[EBP-4],0
74044B68	66:C706 0800	MOV WORD PTR DS:[ESI],8
74044B6D	0000 0000	MOV DWORD PTR DS:[ESI],0

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?

0012F324	0015D83C	Unicode "746"
0012F328	00401748	Unicode "CRK"
0012F32C	00401E11	clave2.00401E11
0012F330	00401748	Unicode "CRK"
0012F334	0015D83C	Unicode "746"
0012F338	0015A828	
0012F33C	0012F4E0	
0012F340	7413EC53	RETURN to MSVBVM50.
0012F344	0012F434	
0012F348	0012F354	

02EA(十六进制)对应的十进制数值如下:

0015D964	FC 00 FD 00 FE 00 FF 00	0. . . . .
0015D96C	E0 00 C0 00 80 00 B0 00	0. . . . .
0015D974	C0 00 50 00 80 00 C0 00	0. . . . .
Command	? 02ea	HEX: 2EA - DEC: 746 - ASCII: 0ê
Breakpoint at MSVBVM50.74044B5C		

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。

Registers (FPU)		
EAX	0015D88C	Unicode "CRK746"
ECX	00000000	
EDX	00150608	
EBX	FFFF0008	
ESP	0012F320	
EBP	0012F33C	
ESI	00401E11	clave2.00401E11
EDI	0012F434	
EIP	740420FB	MSVBVM50.740420FB
C 0	ES 0023	32bit 0(FFFFFFFF)
P 0	CS 001B	32bit 0(FFFFFFFF)
A 0	SS 0023	32bit 0(FFFFFFFF)
Z 0	DS 0023	32bit 0(FFFFFFFF)
S 0	FS 003B	32bit 7FFDF000(FFF)

我们可以看到两个变量被拼接到了一起。

74044B50	0F84 B4F00500	JE MSVBVM50.74043C0A
74044B56	FF75 F4	PUSH DWORD PTR SS:[EBP-C]
74044B59	FF75 F8	PUSH DWORD PTR SS:[EBP-8]
74044B5C	E8 2ED5FFFF	CALL MSVBVM50.__vbaStrCat
74044B61	8B75 08	MOV ESI,DWORD PTR SS:[EBP+8]
74044B64	837D FC 00	CMP DWORD PTR SS:[EBP-4],0
74044B68	66:C706 0800	MOV WORD PTR DS:[ESI],8
74044B6D	8946 08	MOV WORD PTR DS:[ESI+8],EAX
74044B70	75 09	JNZ SHORT MSVBVM50.74044B7B
74044B72	8BC6	MOV EAX,ESI
74044B74	5E	POP ESI
74044B75	8BE5	MOV ESP,EBP
74044B77	5D	POP EBP
74044B78	C2 0C00	RETN 0C
74044B7B	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]
74044B7E	66:8378 50 08	CMP WORD PTR DS:[EAX+50],8
74044B83	75 12	JNZ SHORT MSVBVM50.74044B97
74044B85	FF70 58	PUSH DWORD PTR DS:[EAX+58]
74044B88	FF15 88190474	CALL DWORD PTR DS:[<&OLEAUT32.#6>]
74044B8E	8B4D FC	MOV ECX,DWORD PTR SS:[EBP-4]
74044B91	66:C741 50 000	MOV WORD PTR DS:[ECX+50],0
74044B97	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]
74044B99	66:8378 50 08	CMP WORD PTR DS:[EAX+50],8
EAX=0015D88C, (Unicode "CRK746")		
Stack DS:[0012F43C]=000002EA		
Address	Hex dump	ASCII

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。

Address	Hex dump	ASCII
0012F434	08 00 00 00 00 00 00 00	.....
0012F43C	0C 08 15 00 00 00 00 00	i\$.....
0012F444	08 00 00 00 00 00 00 00	.....
0012F44C	48 17 40 00 00 00 00 00	H#@.....

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。

address	Hex dump	ASCII
015D88C	43 00 52 00 4B 00 37 00	C.R.K.7.
015D894	34 00 36 00 00 00 14 20	4.6...n
015D89C	DC 02 22 21 61 01 3A 20	0"ta0:
015D8A4	53 01 9D 00 EB 00 05 00	00.0.0.
015D8AC	A0 10 A1 00 78 01 15 00	0i.x0\$.

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。

7413EC53	57	PUSH EDI
7413EC54	33C0	XOR EAX,EAX
7413EC56	8A46 02	MOV AL,BYTE PTR DS:[ESI+2]
7413EC59	83C6 03	ADD ESI,3
7413EC5C	FF2485 94ED137	JMP DWORD PTR DS:[EAX*4+7413ED94]
7413EC63	E8 18D1F0FF	CALL MSUBUM50. vbaLenBstr
7413EC68	50	PUSH EAX
7413EC69	33C0	XOR EAX,EAX
7413EC6B	8A06	MOV AL,BYTE PTR DS:[ESI]
7413EC6D	46	INC ESI
7413EC6E	FF2485 94ED137	JMP DWORD PTR DS:[EAX*4+7413ED94]
7413EC75	E8 1AD1F0FF	CALL MSUBUM50. vbaInStr
7413EC7A	50	PUSH EAX
7413EC7B	33C0	XOR EAX,EAX
7413EC7D	8A06	MOV AL,BYTE PTR DS:[ESI]
7413EC7F	46	INC ESI
7413EC80	FF2485 94ED137	JMP DWORD PTR DS:[EAX*4+7413ED94]
7413EC87	33C0	XOR EAX,EAX
7413EC88	8A06	MOV AL,BYTE PTR DS:[ESI]
DS:[00401E13]=FB AL=00		

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。

Address	Hex dump	ASCII
00401E13	FB 40 1A 78 FF 36 04 00	'@+x 6+
00401E1B	34 FF 54 FF 1C 99 01 27	4 T L00'
00401E23	F4 FE 27 14 FF 3A 44 FF	'n'q :D
00401E2B	02 00 4E 34 FF 04 34 FF	0.N4 +4
00401E33	F5 10 00 00 00 3A 64 FF	3>...:d
00401E3B	0A 00 4E 54 FF 04 54 FF	..NT +T

7413EC7B	33C0	XOR EAX,EAX
7413EC7D	8A06	MOV AL,BYTE PTR DS:[ESI]
7413EC7F	46	INC ESI
7413EC80	FF2485 94ED137	JMP DWORD PTR DS:[EAX*4+7413ED94]
7413EC87	33C0	XOR EAX,EAX
7413EC89	8A06	MOV AL,BYTE PTR DS:[ESI]
7413EC8B	46	INC ESI
7413EC8C	FF2485 94F1137	JMP DWORD PTR DS:[EAX*4+7413F194]
7413EC93	33C0	XOR EAX,EAX
7413EC95	8A06	MOV AL,BYTE PTR DS:[ESI]
7413EC97	46	INC ESI
7413EC98	FF2485 94FF137	JMP DWORD PTR DS:[EAX*4+7413FF94]
DS:[00401E14]=40 ('@') AL=00		

Address	Hex dump	ASCII
00401E13	FB 40 1A 78 FF 36 04 00	'@+x 6+
00401E1B	34 FF 54 FF 1C 99 01 27	4 T L00'
00401E23	F4 FE 27 14 FF 3A 44 FF	'n'q :D

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

7413EBF3	BB 84ED1374	MOV EBX,MSUBUM50.7413ED84
7413EBF8	EB 07	JMP SHORT MSUBUM50.7413EC01
7413EBFA	6A 00	PUSH 0
7413EBFC	BB 74ED1374	MOV EBX,MSUBUM50.7413ED74
7413EC01	E8 8E180000	CALL MSUBUM50.74140494
7413EC06	FF3483	PUSH DWORD PTR DS:[EBX+EAX*4]
7413EC09	33C0	XOR EAX,EAX
7413EC0B	8A06	MOV AL,BYTE PTR DS:[ESI]
7413EC0D	46	INC ESI
7413EC0E	FF2485 94ED137	JMP DWORD PTR DS:[EAX*4+7413ED94]

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。

0012F348	00000000	Arg1 = 00000000
0012F34C	0012F434	Arg2 = 0012F434
0012F350	0012F414	Arg3 = 0012F414
0012F354	00000003	
0012F358	00000000	
0012F35C	000002EA	

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

Address	Hex dump	ASCII
0012F434	08 00 00 00 00 00 00 00	.....
0012F43C	8C 08 15 00 00 00 00 00	i3....
0012F444	08 00 00 00 00 00 00 00	.....
0012F44C	48 17 40 00 00 00 00 00	H*0....

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。

Address	Hex dump	ASCII
0015D88C	43 00 52 00 4B 00 37 00	C.R.K.7.
0015D894	34 00 36 00 00 00 14 20	4.6...7
0015D89C	DC 02 22 21 61 01 3A 20	0"1a0:
0015D8A4	52 01 90 00 00 00 05 00	500 u *

接下来我们在数据窗口中定位到另一个参数。

Address	Hex dump	ASCII
0012F414	08 00 00 00 00 00 00 00	.....
0012F41C	94 CA 15 00 14 F4 12 00	03.774.
0012F424	00 00 00 00 00 00 00 00	.....
0012F42C	00 00 00 00 00 00 00 00	.....

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。

Address	Hex dump	ASCII
0015CA94	39 00 38 00 39 00 38 00	9.8.9.8.
0015CA9C	39 00 38 00 00 00 00 00	9.8.....
0015CAA4	46 00 6F 00 72 00 6D 00	F.o.r.m.
0015CAAC	00 00 FF FF 05 00 05 00	...1.1.

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。

Address	Hex dump	Disassembly
7413EBF8	BB 84ED1374	MOV EBX,MSUBUM50.7413ED84
7413EBF8	EB 07	JMP SHORT MSUBUM50.7413EC01
7413EBFA	6A 00	PUSH 0
7413EBFC	BB 74ED1374	MOV EBX,MSUBUM50.7413ED74
7413EC01	E8 8E180000	CALL MSUBUM50.74140494
7413EC06	FF3483	PUSH DWORD PTR DS:[EBX+EAX*4]
7413EC09	33C0	XOR EAX,EAX
7413EC0B	8A06	MOV AL,BYTE PTR DS:[ESI]

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。

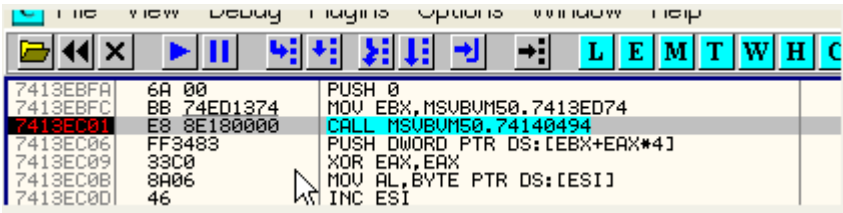
可以看到 EAX 的结果为 FFFFFFFF(PS:这里作者描述有误,该函数的结果是保存在 EAX 中,而非堆栈中),很可能说明刚刚比较的两个字符串不相等,这里我们直接输入正确的序列号,接着按注册按钮看看会不会断在刚刚设置的这个断点处。

## Curso sobre P-Code. Por ...

Usuario:

Serie Núm.:

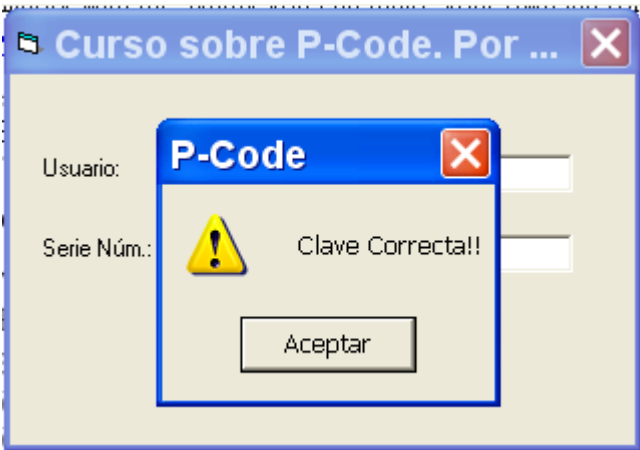
单击 Registrar 按钮。



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 FFreeI1Str	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 FFreeI1Ad	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。

0040102C	68 E8114000	PUSH nags1.004011E8
00401031	E8 EFFFFFFF	CALL <JMP.&MSUBUM50.#100>
00401036	0000	ADD BYTE PTR DS:[EAX],AL
00401038	0000	ADD BYTE PTR DS:[EAX],AL
0040103A	0000	ADD BYTE PTR DS:[EAX],AL

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

Address	Hex dump	ASCII
00401A2D	0A	
00401A35	5C	Backup
00401A3D	13	
00401A45	00	Copy
00401A4D	00	
00401A55	00	Binary
00401A5D	00	
00401A65	00	Label
00401A6D	00	:
00401A75	FF	
00401A7D	FE	Breakpoint
00401A85	CC	
00401A8D	CC	Search for

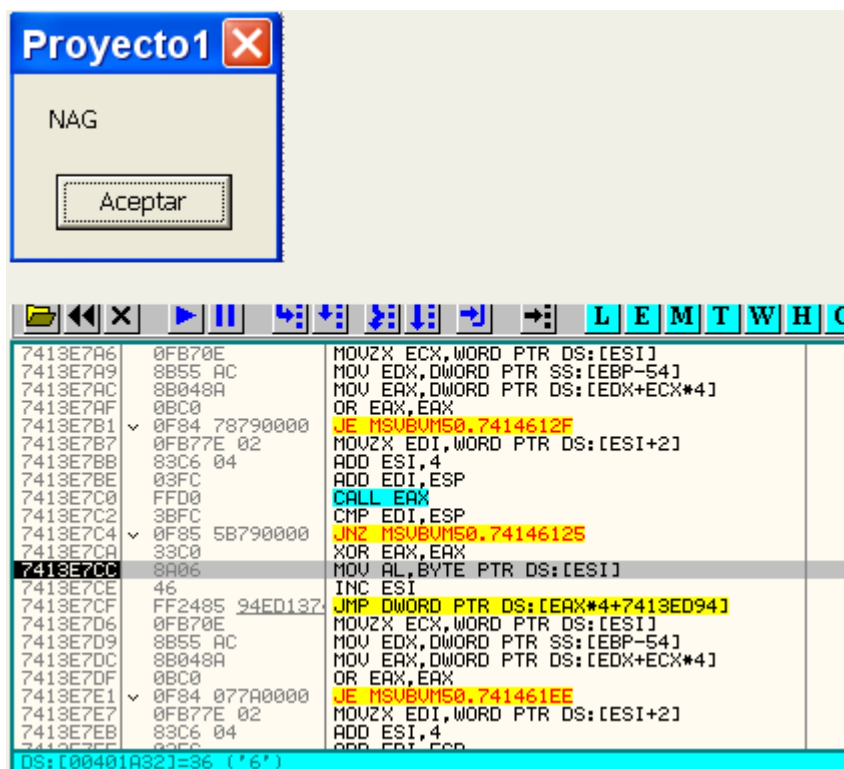
  

7413D9AA	8A46 02	MOV AL,BYTE PTR DS:[ESI+2]
7413D9AD	83C6 03	ADD ESI,3
7413D9B0	FF2485 94ED137	JMP DWORD PTR DS:[EAX*4+7413ED94]
7413D9B7	8B45 08	MOV EAX,DWORD PTR SS:[EBP+8]
7413D9BA	8945 B4	MOV DWORD PTR SS:[EBP-4C],EAX
7413D9BD	33C0	XOR EAX,EAX
7413D9BF	8A06	MOV AL,BYTE PTR DS:[ESI]

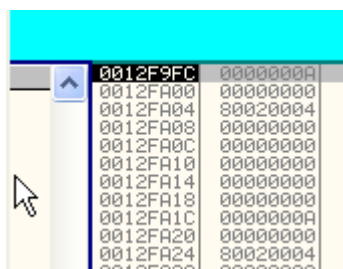
运行起来,马上就会断在了读取 rtcMsgBox 操作码的指令处。

0012F9E8	0012FA5C
0012F9EC	00000000
0012F9F0	0012FA3C
0012F9F4	0012FA1C
0012F9F8	0012F9FC
0012F9FC	0000000A
0012FA00	00000000
0012FA04	80020004
0012FA08	00000000
0012FA0C	00000000

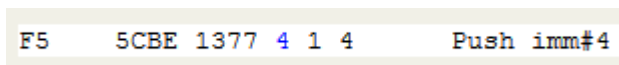
当前栈顶指针指向的是 12F9E8,我们继续往下跟踪直到读取下一个操作码的指令处为止,但是还没到读取下一个操作码的指令处,NAG 窗口就弹出来了,我们单击 Aceptar(OK)按钮。



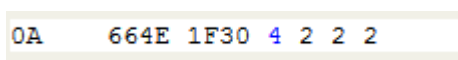
现在我们终于跟到读取下一个操作码的指令处,刚刚跟踪的过程中,如果你足够留心的话,你就会发现是执行了上面的 CALL EAX 指令弹出的 NAG 窗口,这个 CALL 其实就是调用 rtcMsgBox 这个 API 函数,当前栈顶指针指向的是 12F9FC。



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。

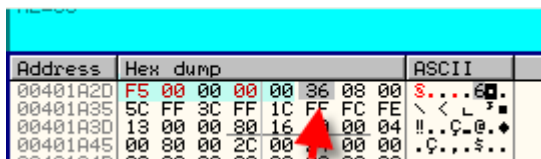


F5 表示 PUSH imm#4,说明只有一个参数且该参数占 4 个字节,刚好可以与 0A 这个操作码的参数所占的大小匹配上。

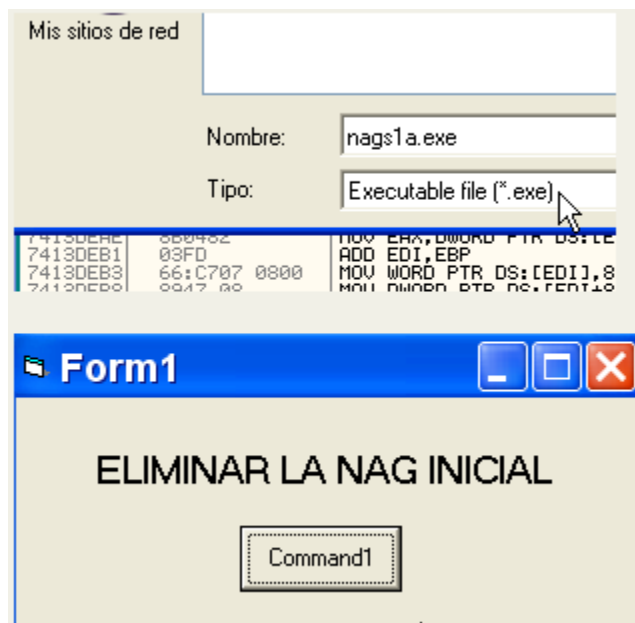
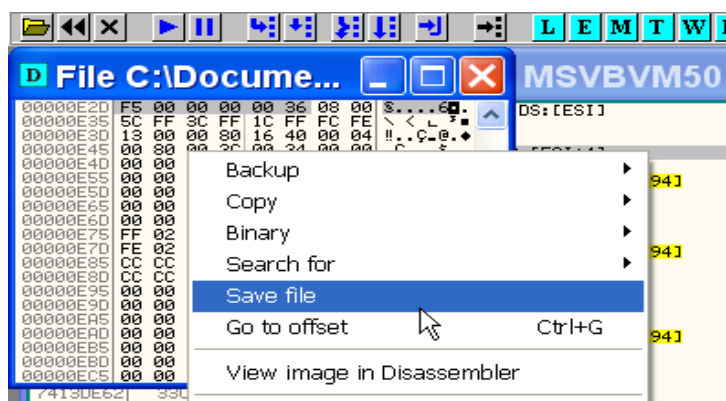
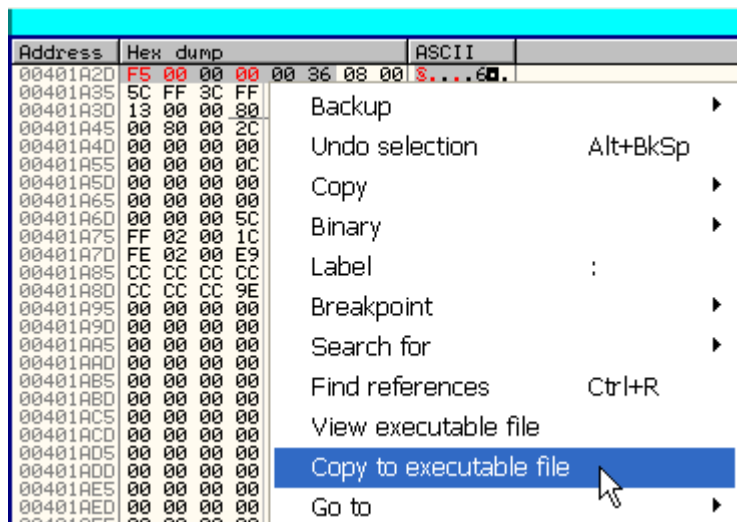


我们可以看到 0A 操作码有两个参数,每个参数占两个字节,所以两个参数总共占 4 个字节。

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
{
    UT_EMPTY = 0,
    UT_NULL = 1,
    UT_I2 = 2,
    UT_I4 = 3,
    UT_R4 = 4,
    UT_R8 = 5,
    UT_CY = 6,
    UT_DATE = 7,
    UT_BSTR = 8,
    UT_DISPATCH = 9,
    UT_ERROR = 10,
    UT_BOOL = 11,
    UT_VARIANT = 12,
    UT_UNKNOWN = 13,
    UT_DECIMAL = 14,
    UT_I1 = 16,
    UT_UI1 = 17,
    UT_UI2 = 18,
    UT_UI4 = 19,
    UT_I8 = 20,
    UT_UI8 = 21,
    UT_INT = 22,
    UT_UINT = 23,
    UT_VOID = 24,
    UT_HRESULT = 25,
    UT_PTR = 26,
    UT_SAFEARRAY = 27,
    UT_CARRAY = 28,
    UT_USERDEFINED = 29,
    UT_LPSTR = 30,
    UT_LPWSTR = 31,
    UT_RECORD = 36,
    UT_INT_PTR = 37,
    UT_UINT_PTR = 38,
    UT_FILETIME = 64,
    UT_BLOB = 65,
    UT_STREAM = 66,
    UT_STORAGE = 67,
    UT_STREAMED_OBJECT = 68,
    UT_STORED_OBJECT = 69,
    UT_BLOB_OBJECT = 70,
    UT_CF = 71,
    UT_CLSID = 72,
    UT_VERSIONED_STREAM = 73,
    UT_BSTR_BLOB = 0xffff,
    UT_VECTOR = 0x1000,
    UT_ARRAY = 0x2000,
    UT_BYREF = 0x4000,
    UT_RESERVED = 0x8000,
    UT_ILLEGAL = 0xfffff,
    UT_ILLEGALMASKED = 0xffff,
    UT_TYPMASK = 0xffff
};

```

```

1 struct tagVARIANT
2 {
3     union
4     {
5         struct __tagVARIANT
6         {
7             VARTYPE vt;
8             WORD wReserved1;
9             WORD wReserved2;
10            WORD wReserved3;
11            union
12            {
13                LONGLONG llVal;
14                LONG lVal;
15                BYTE bVal;
16                SHORT iVal;
17                FLOAT fltVal;
18                DOUBLE dblVal;
19                VARIANT_BOOL boolVal;
20                _VARIANT_BOOL bool;
21                SCODE scode;
22                CY cyVal;
23                DATE date;
24                BSTR bstrVal;
25                IUnknown *punkVal;
26                IDispatch *pdispVal;
27                SAFEARRAY *parray;
28                BYTE *pbVal;
29                SHORT *piVal;
30                LONG *plVal;
31                LONGLONG *pllVal;
32                FLOAT *pfltVal;
33                DOUBLE *pdblVal;
34                VARIANT_BOOL *pboolVal;
35                _VARIANT_BOOL *pbool;
36                SCODE *pscode;
37                CY *pcyVal;
38                DATE *pdate;
39                BSTR *pbstrVal;
40                IUnknown **ppunkVal;
41                IDispatch **ppdispVal;
42                SAFEARRAY **pparray;
43                VARIANT *pvarVal;
44                PVOID byref;
45                CHAR cVal;
46                USHORT uiVal;
47                ULONG ulVal;
48                ULONGLONG ullVal;
49                INT intVal;
50                UINT uintVal;
51                DECIMAL *pdecVal;
52                CHAR *pcVal;
53                USHORT *puiVal;
54                ULONG *pulVal;
55                ULONGLONG *pullVal;
56                INT *pintVal;
57                UINT *puintVal;
58                struct __tagBRECORD
59                {
60                    PVOID pvRecord;
61                    IRecordInfo *pRecInfo;
62                } __VARIANT_NAME_4;
63            } __VARIANT_NAME_3;
64        } __VARIANT_NAME_2;
65        DECIMAL decVal;
66    } __VARIANT_NAME_1;
67 } ;

```

