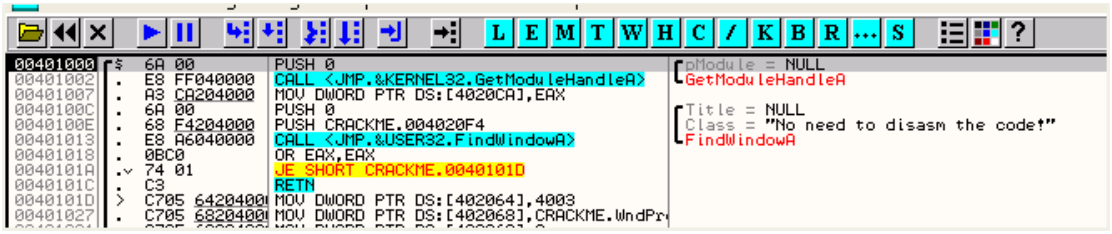
第 16 章-序列号生成算法分析-Part1

本章,我们分析的 CrackMe 与之前的不同之处在于序列号是基于名称变化的,也就是说我们将讨论序列号生成算法。

尽管分析的技巧和之前的很相似,我们还是得来看几个例子巩固一下。

我们先来分析一下 CrueHead'a 的 CrackMe 的序列号生成算法。

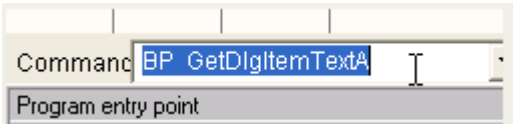
用 OD 加载它。



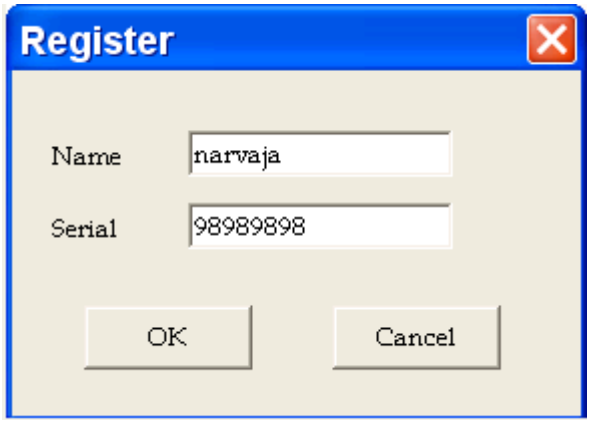
我们停在了入口点处。

我们来看看该程序使用什么 API 函数来获取用户输入的序列号。

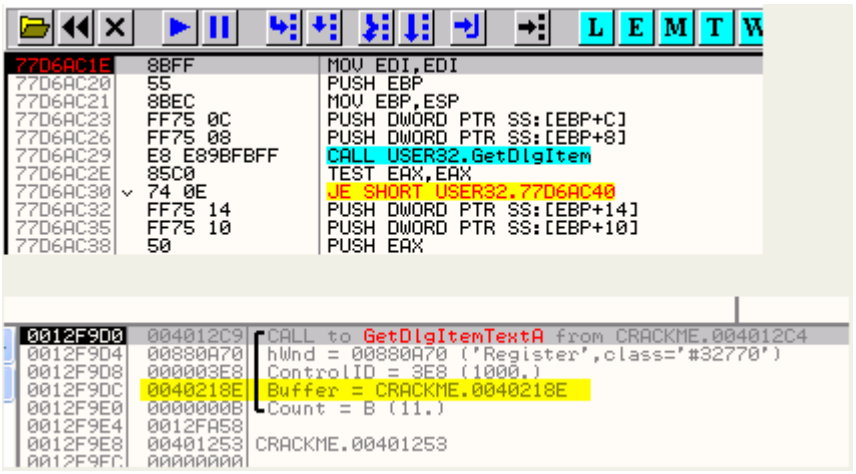
我们可以看到 GetDlgItemTextA 这个 API 函数,我们给这个 API 函数设置断点,输入用户名和正确的序列号看看是否会中断下来。



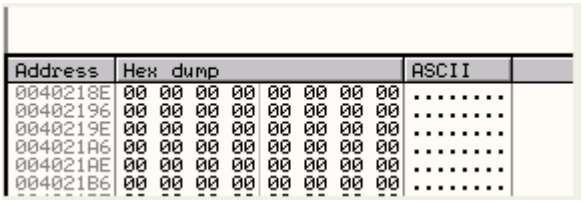
运行起来。



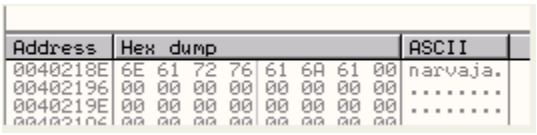
我们单击 OK 按钮,断在了刚刚设置的断点处,我们来看看堆栈中的参数情况。



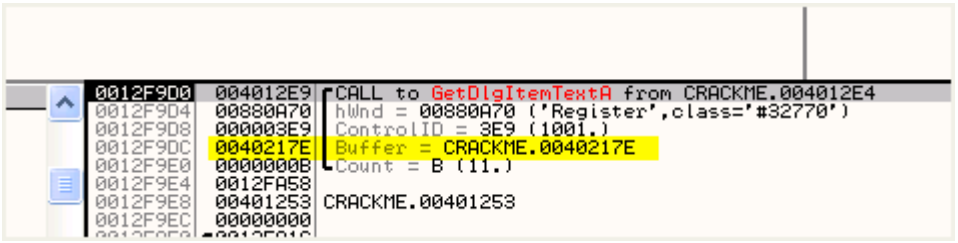
Buffer 指向的缓冲区存放用户输入的文本,起始地址为 40218E,我们在 Buffer 参数上面单击鼠标右键选择-Follow in Dump 来在数据窗口中定位到该缓冲区。



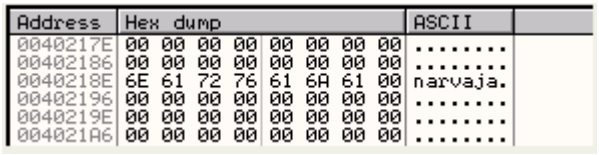
由于此函数还没有执行,所有缓冲区里面是空的,我们选择主菜单中的 Debug-Execute till return,我们就到了 ret 指令处,我们按 F7 键返回到主模块中。



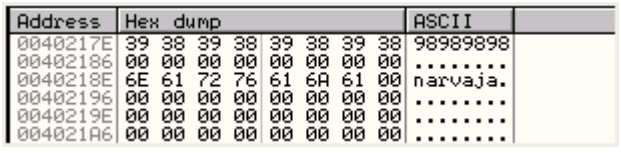
我们可以看到缓冲区中保存了我们输入的名字,程序会进行相应的处理生成正确的序列号。如果我们想编写注册机的话,我们得分析由名称生成序列号的算法,但是现在我们暂时不关心生成算法,我们只是想看看生成的序列号是多少。



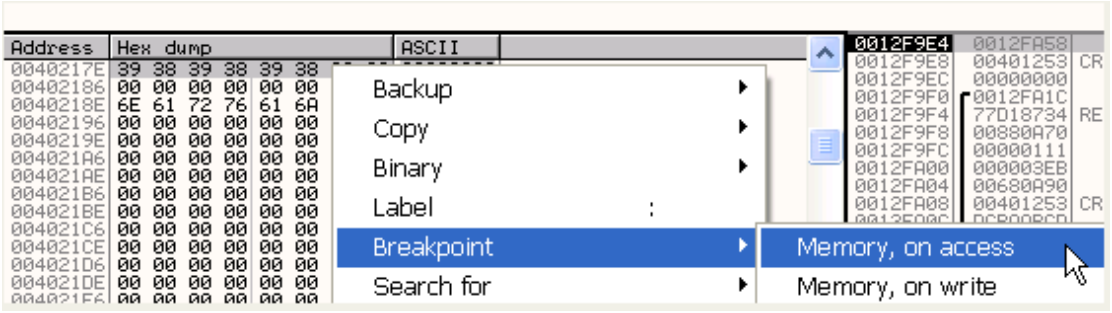
第二次断在了 GetDlgItemTextA 处,新的缓冲区保存用户输入的序列号,起始地址为 40217E,我们在数据窗口中定位到该缓冲区。



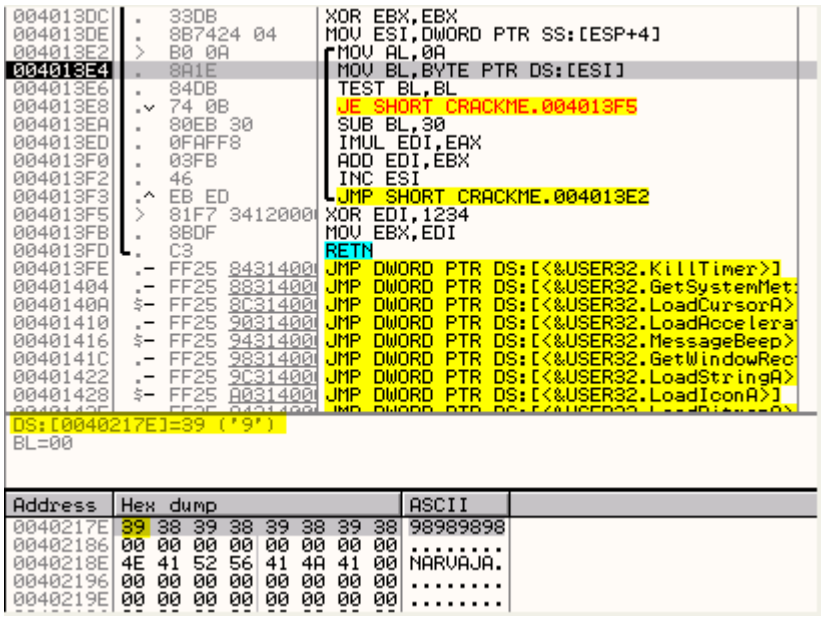
同样由于函数还没有执行,缓冲区里面是空的,我们通过选择主菜单中的 Debug-Execute till return 执行到返回,然后按 F7 键返回到主模块中。



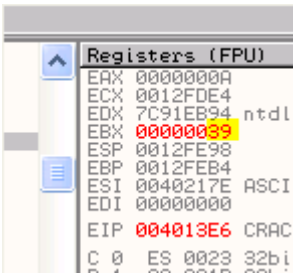
好了,现在缓冲区里面存放了我们输入的错误序列号,从程序的角度出发,程序就会取用户输入的错误的序列号与根据名称生成的正确序列号进行比较,所以我们可以对错误序列号设置内存访问断点,看看程序的哪些地方使用了。



我们拖选中错误的序列号,单击鼠标右键选择-Breakpoint-Memory,on access,然后运行起来。

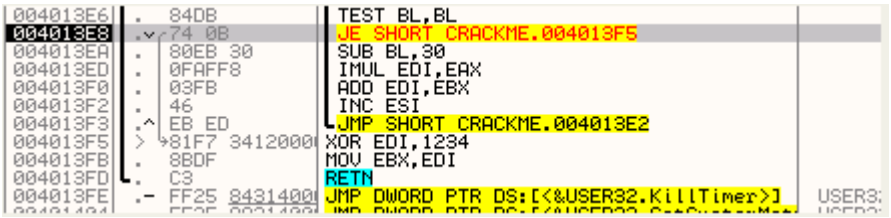


可以看到该程序尝试读取错误序列号的第一个字节并且将保存到 BL 寄存器中,我们按 F7 键。

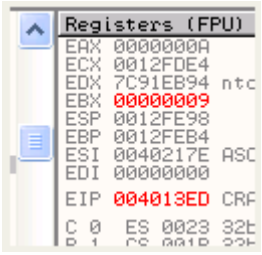


可以看到 BL 寄存器保存的值为 39,是错误序列号的第一个字节,我们需要跟踪程序的处理过程,如果可能的话,记录下该程序的相关数学运算。

这里判断 BL 的值是否为零,为零说明到了字符串结尾,如果到了字符串结尾将结束循环,我们按 F7 键单步。

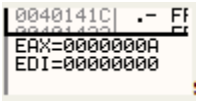


因为 BL 不为零,所以跳转不会发生,将执行 SUB BL,30 指令。

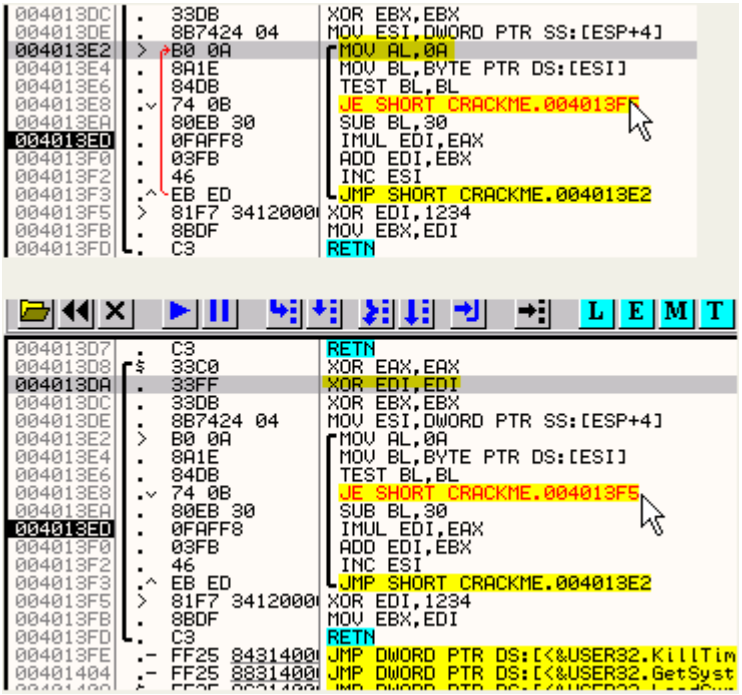


BL 的值减去 30 等于 9,即错误序列号第一个字符的值。

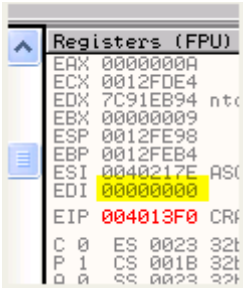
下一条指令 EDI 乘以 EAX。



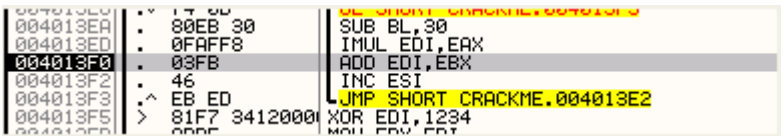
这两个寄存器被初始化为以下值:EAX 寄存器循环体开始处被初始化为了 0A,EDI 寄存器在循环体之前被 XOR EDI,EDI 指令初始化为零了。



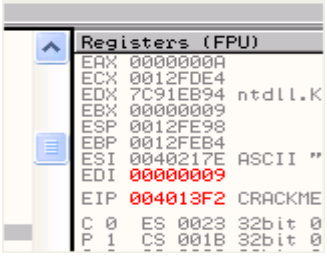
按 F7 键,我们可以看到两个操作数相乘,并且 IMUL 指令会考虑符号位,并且结果被保存到第一个操作数中。



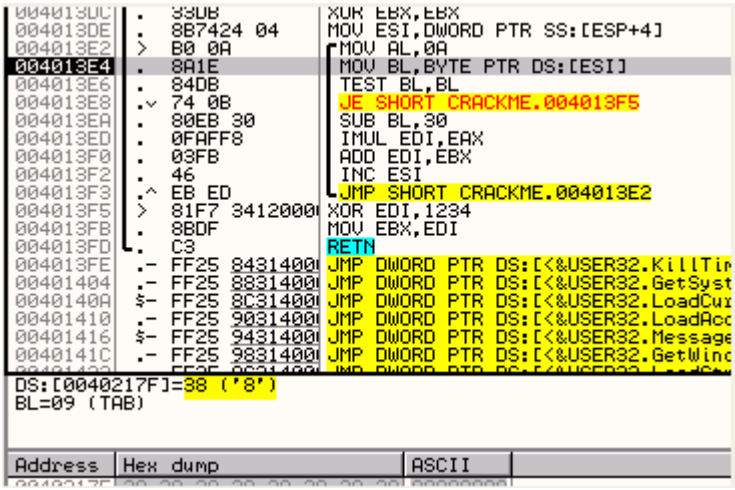
相乘的结果为零,保存在 EDI 中。



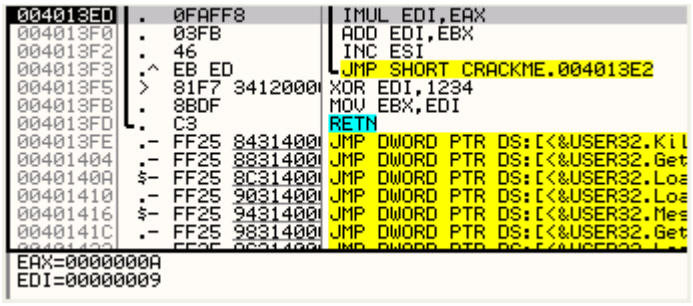
接着 EDI 加上 EBX。



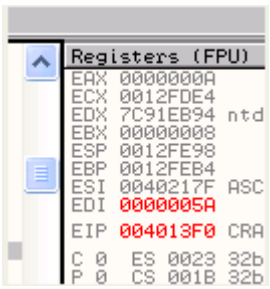
EDI 的结果为 9,下一条指令 INC ESI,ESI 递增 1,然后跳转到循环开始处,读取错误序列号的下一个字节。



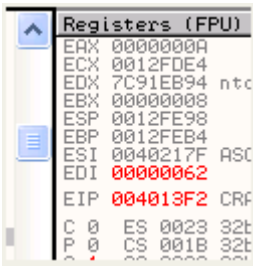
AL 依然被初始化为了 0A,错误序列号的下一个字节值为 38,不为零,所以将减去 30,然后执行 IMUL 指令。



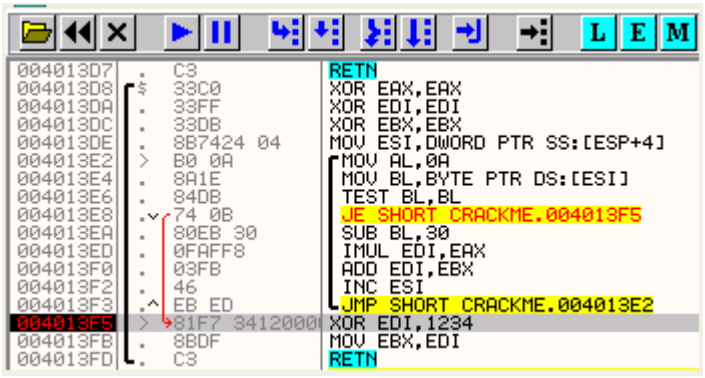
可以看到上次循环的结果 EDI 乘以 EAX 的值(依然是初始化为了 0A),结果依然保存在 EDI 中。



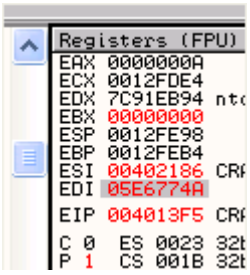
现在 EDI 的值为 5A,下一条指令,EDI 将加上 EBX。



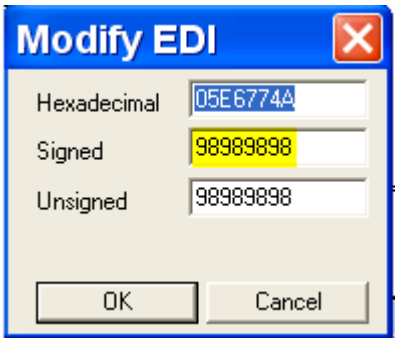
以上结果依然保存到 EDI 中,一步步跟踪这个循环是烦的,如果略过这个过程呢?我们接下来将介绍。



我们只需要在循环的下一条指令处设置一个断点,然后按 F9 键,就会中断下来,我们看看 EDI 的值为多少。

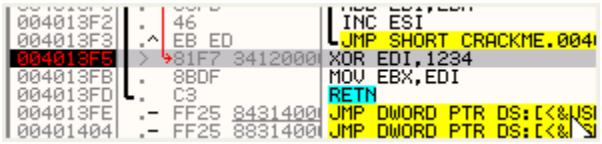


我们双击 EDI 寄存器。

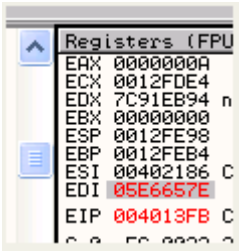


EDI 的值为十六进制形式,第二行显示的是有符号的十进制值,也就是我们输入的错误序列号的值,即以上循环结束 EDI 保存了错误序列号的十六进制值。

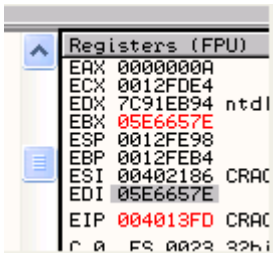
概括来说就是: “98989898”序列号将被转化为十进制的 98989898 或者十六进制的 5E6774A。



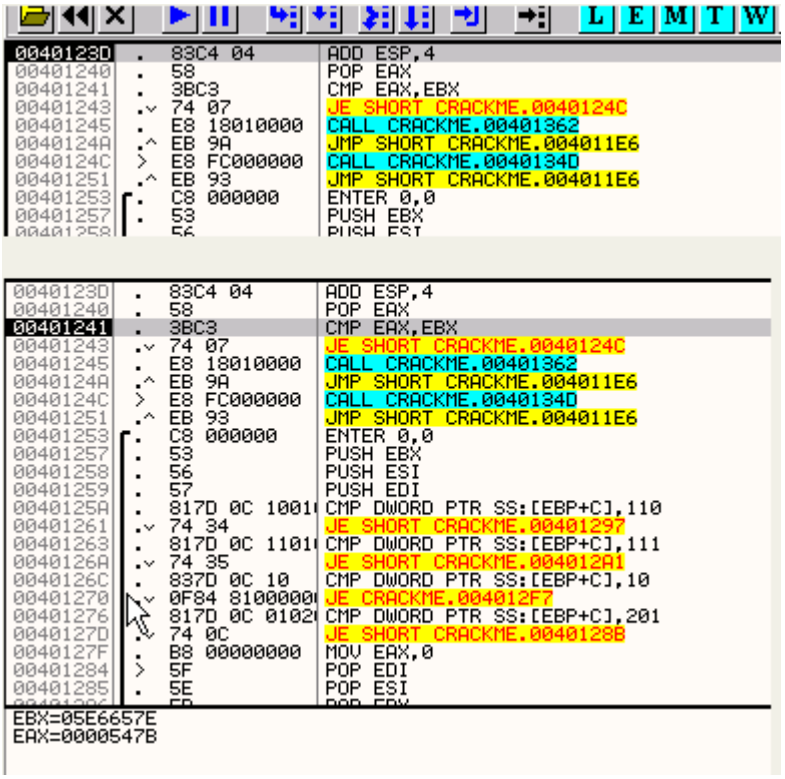
下一条指令,EDI 异或 1234。



结果为 5E6657E,然后保存到 EBX 寄存器中,接着就到了 RET 指令。



我们可以看到 RET 返回以后,EAX 与 EBX 进行比较,根据比较的结果来决定是否跳转到正确序列号部分。



我们可以看到 EBX 与 EAX 进行比较,EAX 的值为 547B。

由于 EAX 的由程序计算出来的,EBX 是由我们输入的错误序列号计算出来的,由于我们输入的序列号是错误的,所以这两个寄存器不相等。

如果 EBX 等于 EAX 的话,将跳转到正确的序列号提示窗口处,现在两寄存器的值不相等,我们需要分析原因。

笔记如下:

EBX(错误序列号的十六进制值) XOR 1234

我们需要的是 EAX 与 EBX 相等。

如果 EAX=EBX

用 EAX 替换 EBX

也就是说

EAX(正确序列号的十六进制值)XOR 1234

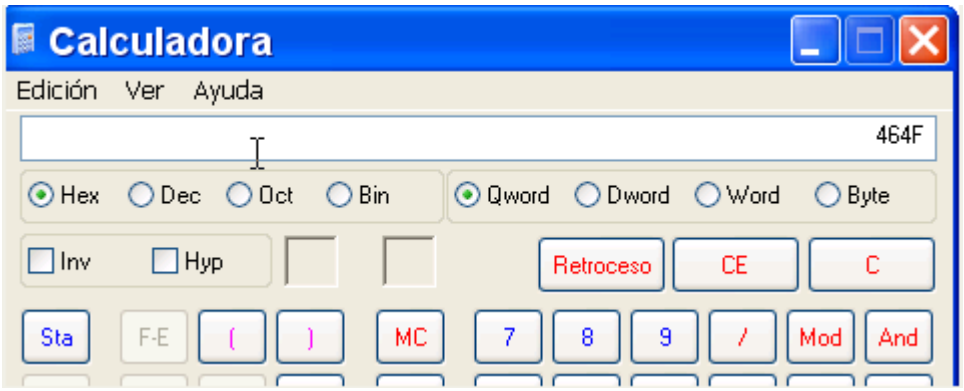
此时 EAX 的值为 547B。我们用这个值替换 EAX。

547B XOR 1234 =

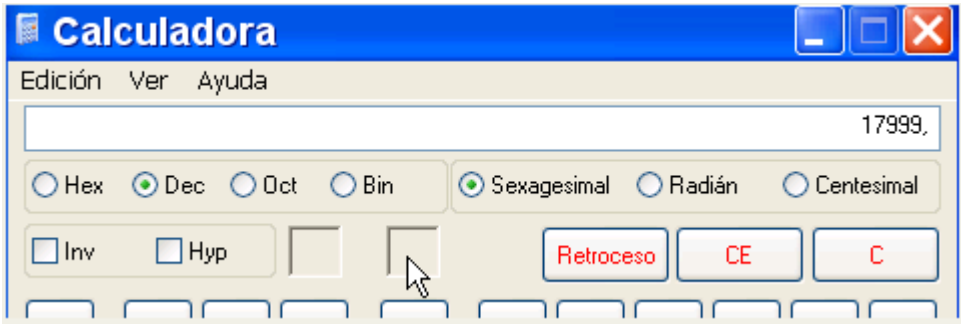
异或运算的结果为:

464F

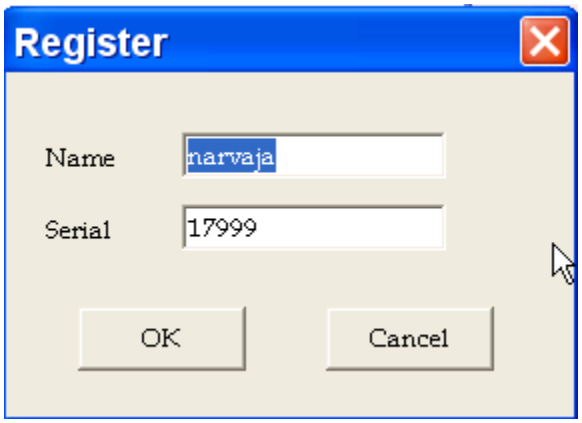
464F 为十六进制值,对应的十进制值为:



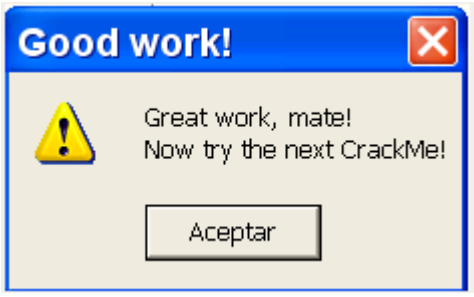
转化为十进制为:



17999 就是”narvaja”这个名称对应的正确序列号。我们删除之前设置的所有断点。



单击 OK 按钮。



这样我们就得到了我们输入名称所对应的正确序列号了。

这是一种解决方案,另一种解决方案如下:

如果我们把对错误序列号进行的一系列操作称之为函数 F。

F(错误序列号) = EBX

对错误序列号进行操作的结果被存放到 EBX 中。

与之进行比较的 EAX 寄存器,我们的公式如下:

F(正确序列号) = EAX

正确序列号进行一系列操作结果保存到 EAX 中。

为了得到正确的序列号,我们对 EAX 进行反向的操作。

正确序列号 = 反向 F(EAX)

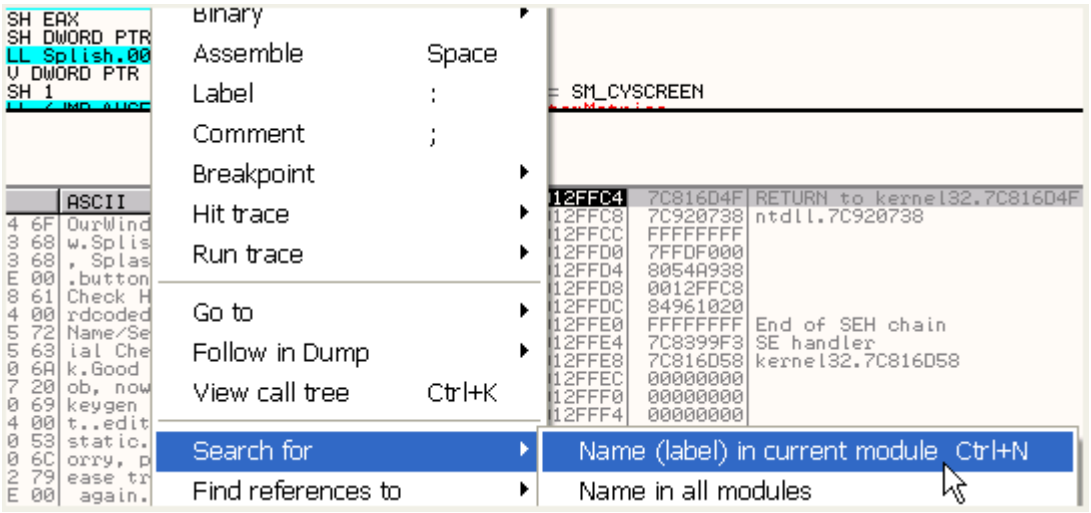
当前情况下,反向操作依然是异或。

因此 XOR EAX 的结果就是正确序列号的十六进制值,我们将其转化为十进制值就得到了正确的序列号。

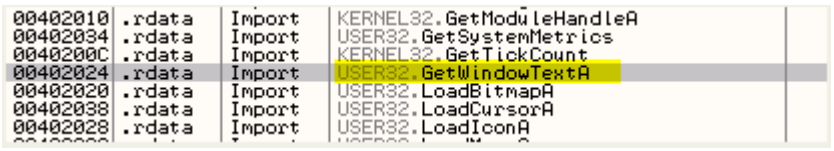
因此,我们也可以利用这种方法来获取正确的序列号,除非反向操作是不等价的,我们来看一个这样的例子。

接下来的例子是 Splish,这个 CrackMe 的第一部分是找到硬编码序列号,现在我们来看看它的第二部分。

加载 Splish,断在入口点处。

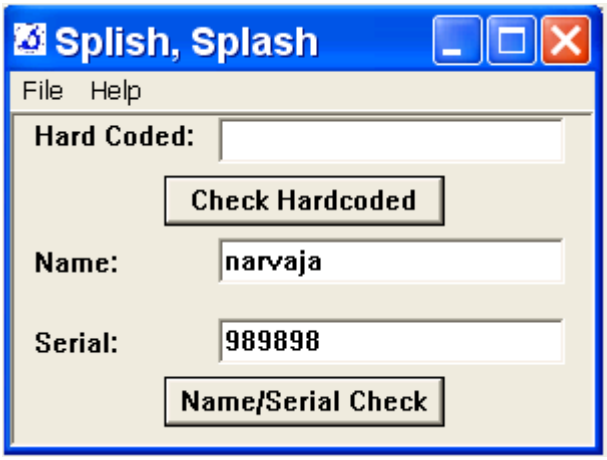


看看程序使用了哪些 API 函数。

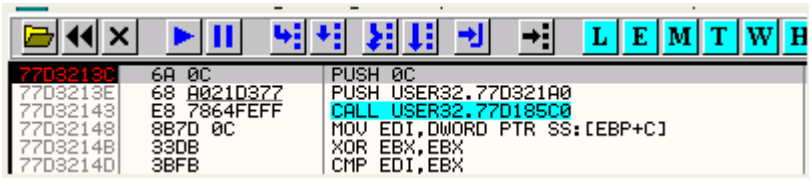


给我们熟悉的 GetWindowTextA 设置断点。

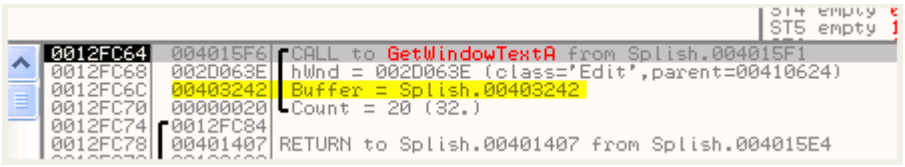
我们按 F9 键运行起来。



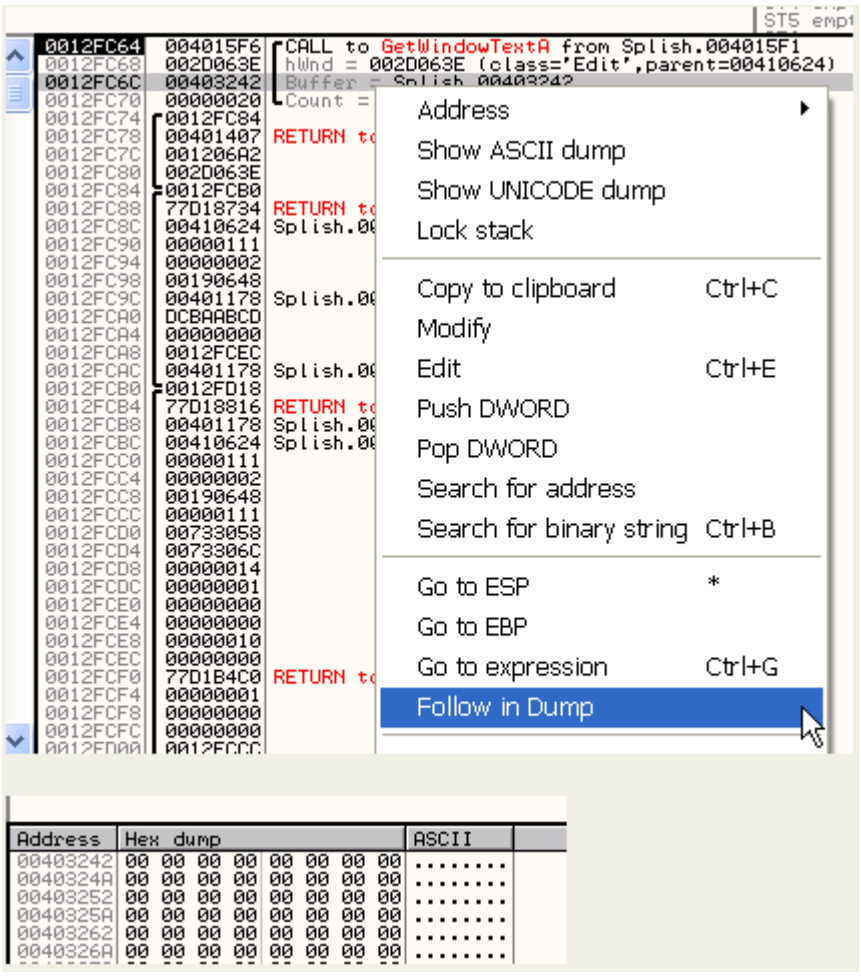
我们随便输入一个错误的名称和序列号,然后单击 Name/Serial Check 按钮。



断在了 GetWindowTextA 这个 API 函数入口处,我们来看看缓冲区。

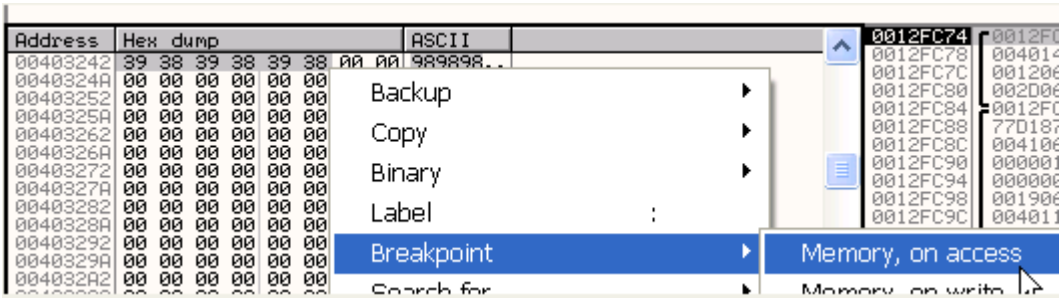


我们在数据窗口中定位到该缓冲区。

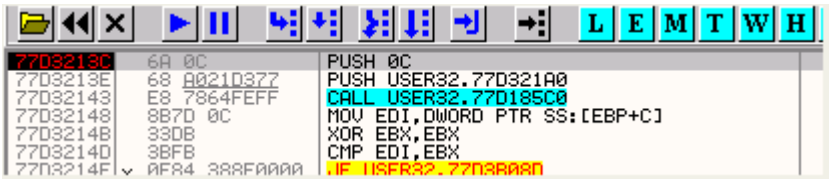


缓冲区里面是空的,我们选择主菜单中的 Debug-Execute till return。就会停在 ret 指令处,接着我们按 F7 键返回到主程序模块中。

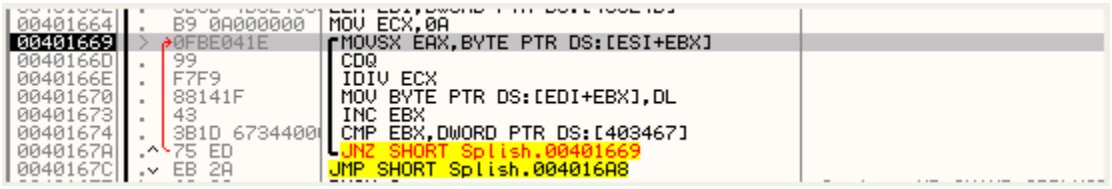
可以看到缓冲区中保存了错误的序列号,我们对错误的序列号设置内存访问断点。



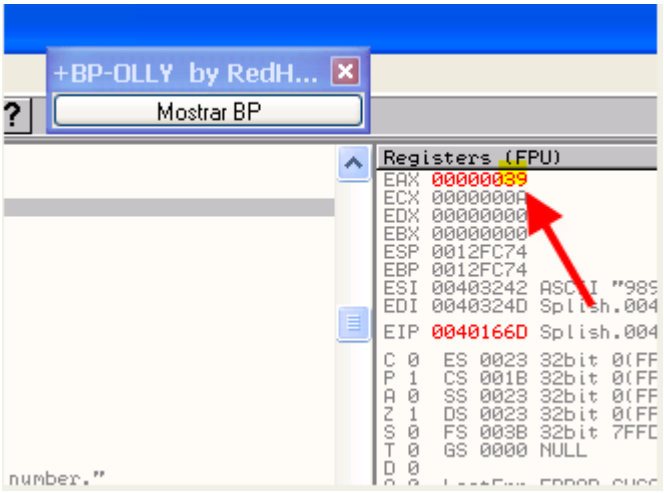
运行起来。



再次断在了 GetWindowTextA 这个 API 函数入口处,这里获取的是窗体的名称,我们不感兴趣,继续运行起来。



断在了这个函数里面,如果我们执行这行代码的话,会发现将错误序列号的第一个字节移动到 EAX 中了。



接下来我们看看下面的 CDQ 指令的解释,我们直接来看 Google 里面的解释,我们在谷歌中直接搜索 CDQ Assembler。

例如如下指令:

CDQ

IDIV ESI

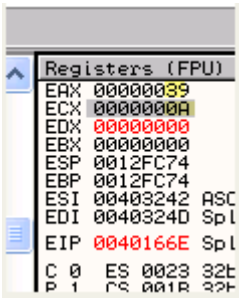
CDQ 指令双字扩展,把 EAX 中的符号位扩展到 EDX 中去,然后 EDX:EAX 对应的值除以 ESI,商保存到 EAX 中,余数保存到 EDX 中。

EAX 符号位扩展到 EDX 中,EDX 的值应该变为零,相当于对 EDX 进行 XOR EDX,EDX 操作。现在不需要将 EDX 清零了,因为 CDQ 指令已经帮我们完成了该操作。

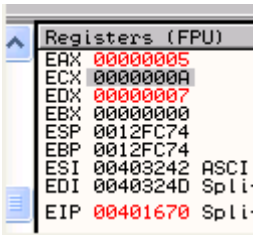
所以当前情况下我们不必每次循环之前将 EDX 赋值为零,我们只需要在 IDIV 指令之前加上一个 CDQ 指令即可。

EDX:EAX 除以 ECX,商存放在 EAX 中,余数存放到 EDX 中。好了,我们现在来看看具体的实现。

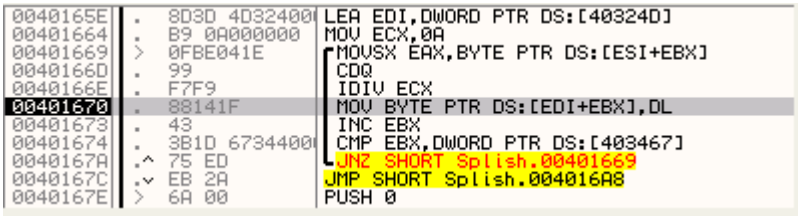
第一个字节为 39,除以 ECX(值为 0A)。



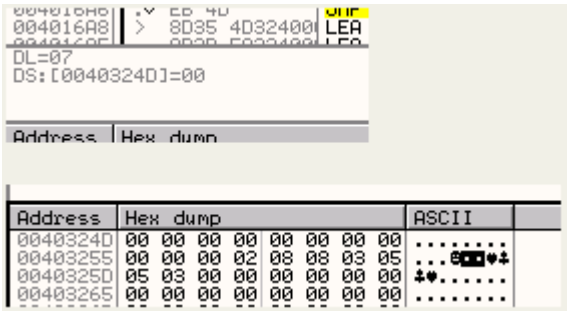
看看发生了什么。



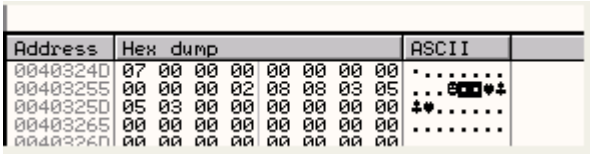
这里商 5 被保存到了 EAX 中,余数 7 被保存到了 EDX 中。



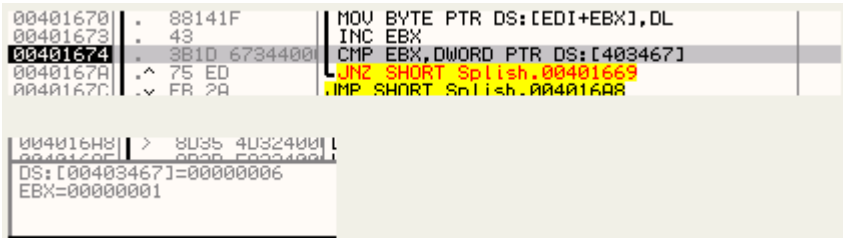
可以看到下一行将 DL 的值保存到 40324D 指向的内存单元中。在数据窗口中定位到 40324D 这个内存单元。



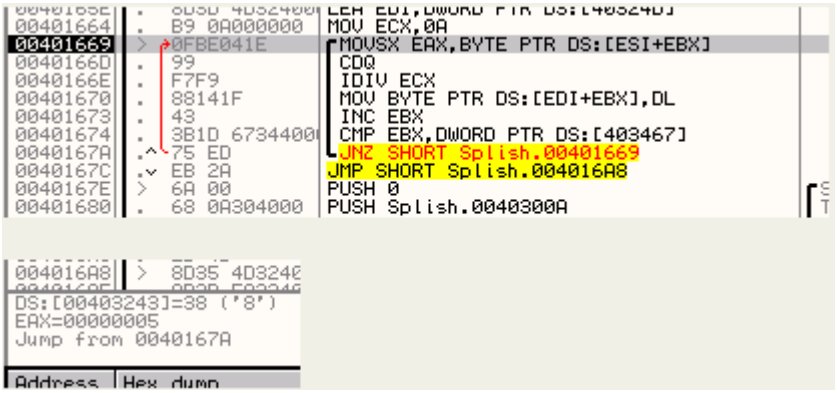
继续按 F7 键。



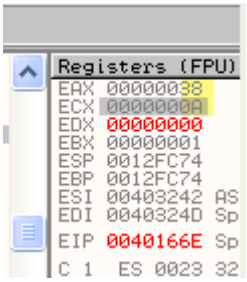
7 被保存到了该内存单元中。



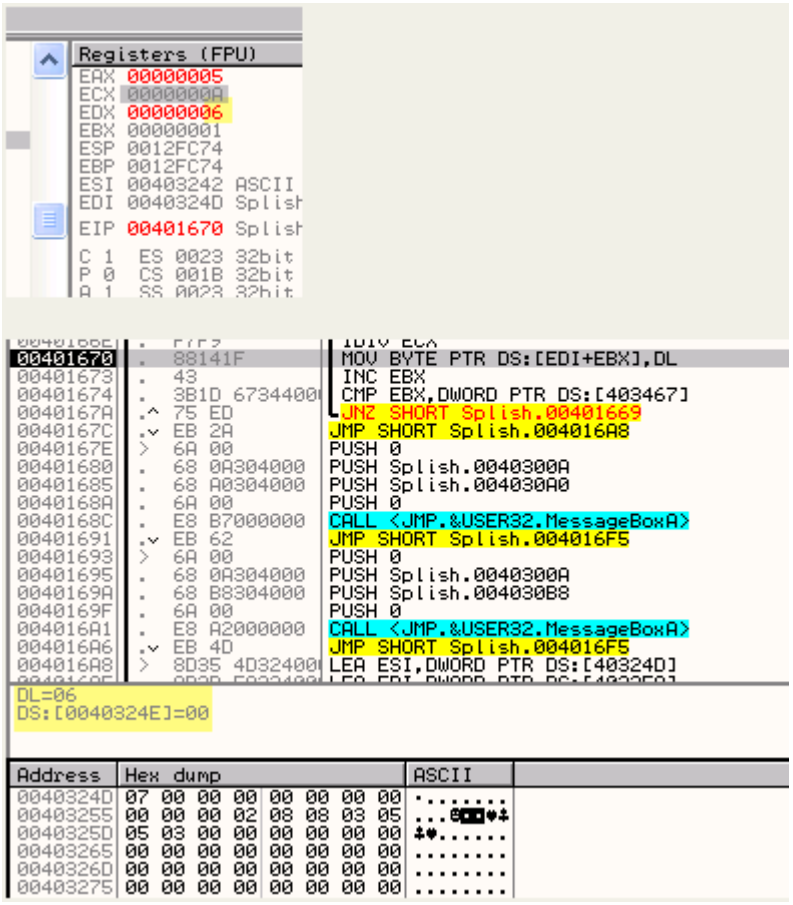
我们可以 EBX 的值为零,然后递增 1,接着与 6 进行比较,如果不相等将继续循环。



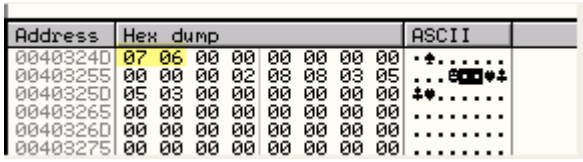
现在我来看看第二个字节将发生什么。



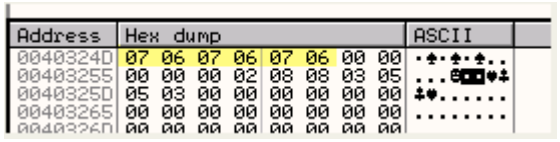
通过是 EDX:EAX 除以 0A,商保存到 EAX 中,余数保存到 EDX 中。



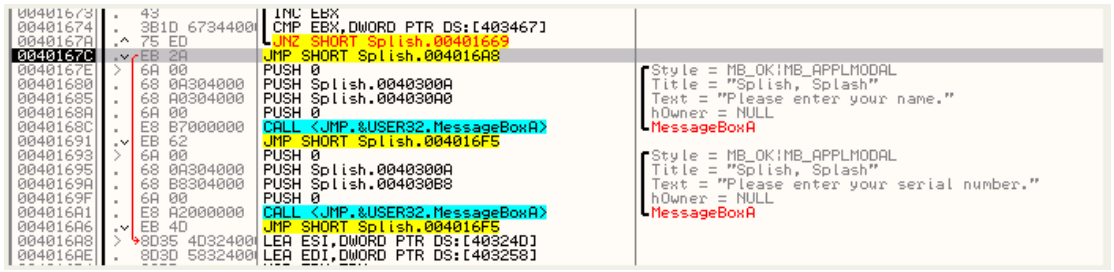
保存余数。



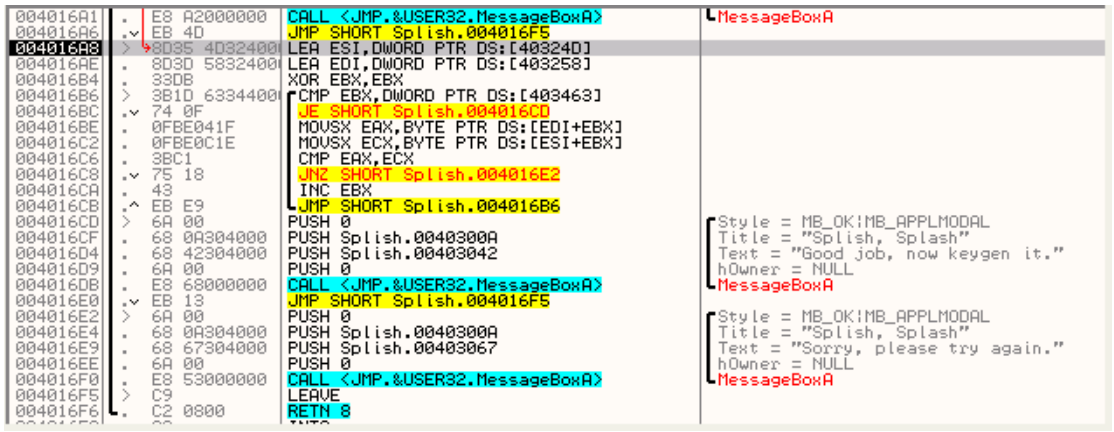
好了,对所有字节进行以上操作。



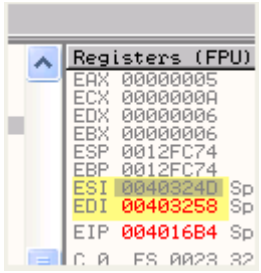
接着 JNZ 指令跳转没有发生退出循环。



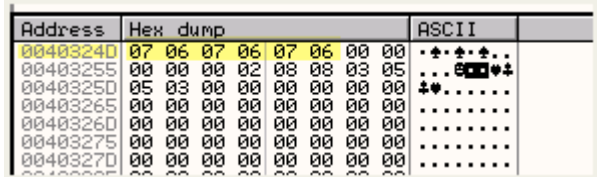
接下里我们看到正确序列号以及错误序列号的消息框代码,说明离找到正确序列号已经不远了。



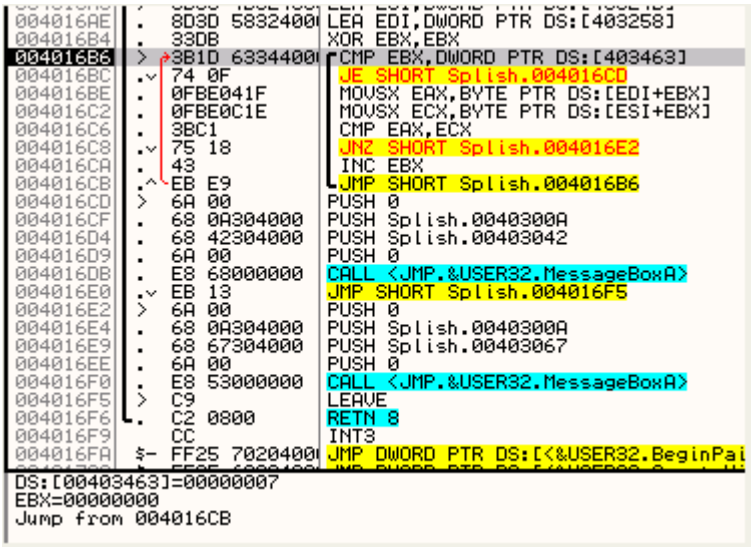
接下来 LEA 指令分别将两个地址保存到 ESI,EDI 寄存器中。



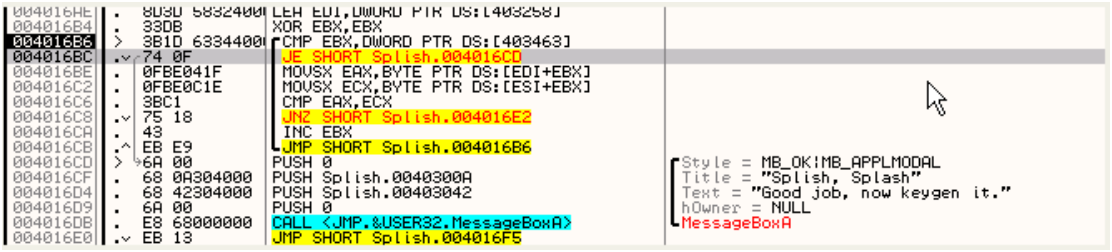
ESI 指向了保存刚刚运行的结果,EDI 指向哪里呢?嘿嘿



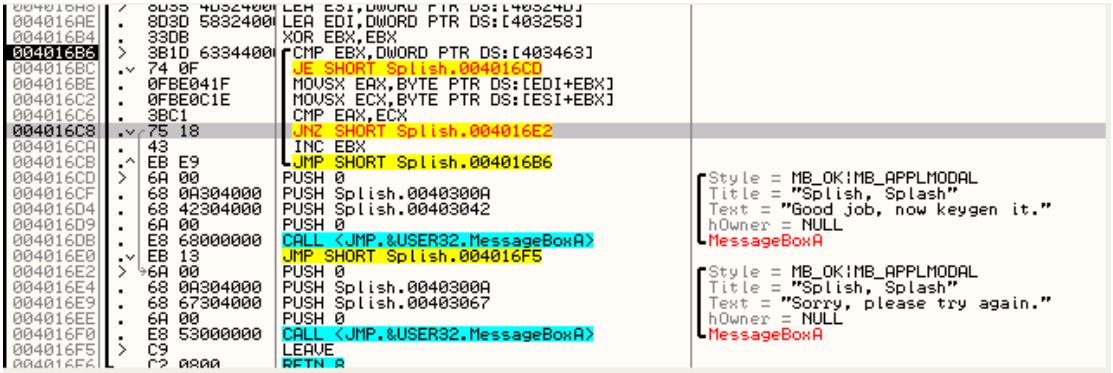
继续



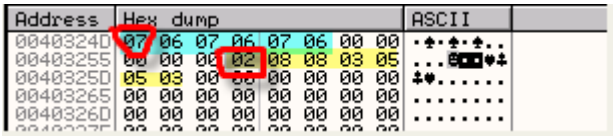
可以 EBX 值为零,与 7 进行比较,如果它们相等...



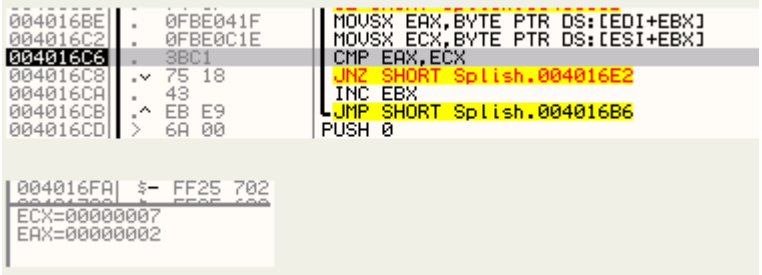
将跳转到正确序列号的提示框处。中间的循环体中还有另一个 JNZ 指令会跳转到错误序列号的消息框处。



好了,我们看看比较。



EAX 将保存 EDI 指向的内存单元的第一个字节即 02,ECX 将保存之前计算结果的第一个字节 7。



如果我们计算结果的第一个字节为 02 的话就好了。

我们当前情况是:

39-5*0A = 7。

也就是说除法运算的结果是 39/0A 商为 5,余数为 7。所以我们可以通过反向运算 5 乘以 0A 然后加上 7 得到 39。

39 = 5*0A + 7

对于正确序列号的情况如下:

正确的字节 = 5*0A + 2

即正确字节 = 32 + 2 = 34 (注意是十六进制)

34 这个 ASCII 码对应的字符'4'.

我们可以看到,

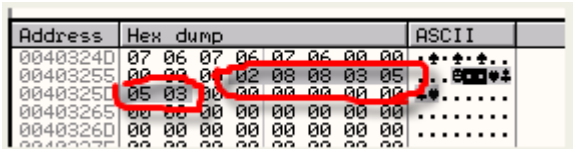
正确字节值 = 5*0A + 2

得到是一个(30 ~ 39)之间的值,如果结果超出了这个范围,我们可以做如下变换:

正确字节值 = 4*0A + 平衡值

好了,我们的第一个字符是'4'。

接下来计算剩下的字节。



02 已经计算过了

然后是 08。

正确字节 = 5 * 0A + 8

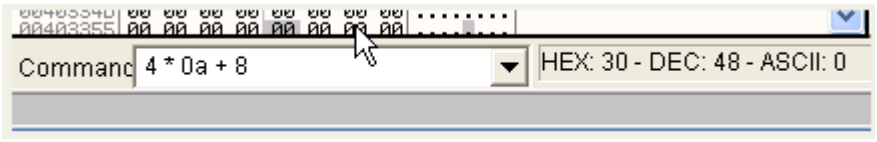
即正确字节 = 32 + 8 (十六进制)

等于 40,超过了 39('9'的 ASCII 码)这个上限,我们按变换的公式计算:

正确字节 = 4 * 0A + 8

即正确字节 = 28 + 8 = 30,即'0'的 ASCII 码。

你也可以在命令栏窗口中验证一下:



因此第二个字节等于 30,即字符'0'的 ASCII 码。

下一个字节依然是 08,所以结果依然是 30,即字符'0'的 ASCII 码。

然后是 03。

正确字节 = 5 * 0A + 3

即正确字节 = 32 + 3 = 35(注意是十六进制),即字符'5'的 ASCII 码。

然后是 05。

正确字节 = 5 * 0A + 5

即正确字节 = 32 +5 = 37,即字符'7'的 ASCII 码。

接着还是 05,正确字节 = 32 +5 = 37,即字符'7'的 ASCII 码。

最后是 03,前面计算过了字符'5'的 ASCII 码。

因此,名称"narvaja"对应的正确序列号是 4005775。删除之前设置的所有断点运行起来。

单击 Name/Serial Check 按钮。

嘿嘿,这个 CrackMe 就完成了。

留个练习的 CrackMe,名字叫 mexcrk1。