教程一：什么是逆向工程

一、什么是逆向工程？

逆向工程是通过编译的二进制文件，尝试重建（或简单理解）程序原始的工作方法。程序员最初在写程序时，一般使用像 C++、VB、God forbid、Delphi 等高级语言。因为计算机本身不能够理解这些语言，所以程序员所写的代码需要被组装成特定的更机器化的格式，也就是计算机所能理解的格式。这个足够原始的代码被叫做机器语言。对人类而言这些代码不太友好，经常需要耗费大量的脑力才能准确的明白程序员的思想。

二、逆向工程是干什么的？

逆向工程能够被用于计算机科学的很多领域，不过这里有几个通用分类：

- 它使得与历史遗留代码（就是已经没有了源代码）进行交互成为可能
- 打破拷贝保护（即打动你的朋友和省钱）
- 研究病毒和恶意软件
- 评估软件质量和稳健性
- 向软件中添加功能

第一个分类就是当源代码不可用时，通过逆向工程编码与已存在的二进制程序进行交互。关于这个我不会讨论太多，因为它太枯燥了。

分类二（也是最大的）是打破拷贝保护。就是禁用限时试用限制，干掉注册，以及免费获得商业软件的其他所有功能。这方面我们会进行大量的讨论。

分类三是学习病毒和恶意软件代码。之所以需要逆向工程，是因为没有几个病毒编写者会向外说出他是如何编写的代码，应该具有什么功能，以及怎样完成这些功能（除非他们真的很愚蠢）。这真是一个让人兴奋的领域，不过这也需要大量的知识。现在我们不会讨论太多，具体的都在后面章节。

分类四是评估软件安全和漏洞。当创建大型应用（想想 Windows 操作系统），逆向工程被用来确保系统不会包含任何主要的漏洞、安全缺陷。坦率的说，是让破解者破解软件时尽可能的困难。

最后一个分类是向现有软件中添加功能。就我个人来说，我认为这是最有趣的地方之一。不喜欢你的网站设计软件中的图片？换掉它们。想在你最喜欢的字处理软件中添加一个加密文档的菜单项？那就加上。想要在 windows 计算器中添加一个损人的消息框去无止尽的作弄你的同事？那就干他一票。在后面的系列中我们将进入这个世界。

三、需要什么知识？

与你猜测的一样，成为一名合格的逆向工程师需要大量的知识。幸运的是，在开始逆向工程时大量的知识都不是必须的。这正是我想要开始的地方。也就是说，享受逆向的乐趣以及从本教

程中收获一些东西。然后你应该至少对一个程序是如何工作的有一个基本的理解（比如，你应该知道一个基本的 if.....then 语句是什么样,数组是什么样,至少理解一个基本的 hello world 程序)。第二，强烈建议熟悉汇编语言。即使没有汇编基础你也可以通过本教程，不过在有些地方你就会希望自己是 ASM 的大牛，以便真正理解你正在做什么。另外，你需要大量的时间来学习怎样使用工具。这些工具对于逆向工程来说是无价的，也需要学习这些工具的快捷键、缺陷和特性。最后，逆向工程需要大量的实践。与不同的壳/保护/加密设计玩耍。学习编写程序的原始语言（甚至 Delphi)、破解反逆向工程的技巧等等。本教程的最后，我加上了“进一步阅读”部分，有一些建议。如果你真的想要学好逆向，强烈建议你进一步阅读其他内容。

四、使用哪种工具？

在逆向领域中有很多种不同的工具可用。在逆向二进制时有许多特别的保护需要被解决 。有一些可以让逆向者的生活更轻松。有一些我认为是“订书针”项目，就是经常用到的那些。对于大部分工具来说，是符合几种分类的：

1、反汇编器

反汇编器尝试将二进制形式的机器语言代码以一种友好的形式显示出来。它们也进行数据推断比如函数调用、传递变量和文本字符串。这就让可执行文件看起来更像人类可读的代码，而不是一串数字串起来的样子。反汇编器非常多，它们中的一些专

门做一些特定的工作（比如 Delphi 中的写二进制）。主要是你找一个你觉得最舒服的。我喜欢用 IDA（ http://www.hex-rays.com/上有免费版本可用），以及一些不太知名的在特定情况下比较有用的工具。

2、调试器

调试器是逆向工程师的面包和黄油。它们首先分析二进制文件，这一点特别像反汇编器。然后调试器允许逆向者单步执行代码，一次运行一行并且查看结果。这对于发现一个程序是如何工作来说是无价的。最后，一些调试器允许对改变的代码做说明，然后带着那些变化再次允许。示例调试器是 Windbg 和 Ollydbg。我几乎只用 Ollydbg（http://www.ollydbg.de/），除非调试内核模式的二进制文件，不过我们不久就会接触到的。

3、十六进制编辑器

十六进制编辑器可以让你查看二进制文件的指定字节，并且可以更改它们。也提供了搜索指定字节，保存部分二进制数据到磁盘等等。有许多免费的十六进制编辑器，并且大部分都挺好用。本教程的大部分都不会用到它们，不过有时候它们是无价的。

4、PE 和资源 查看器/编辑器

每一个被设计在 windows 上运行的二进制文件（linux 也是一样），在它的起始地方都有一个数据区用于告知操作系统如何设置和初始化程序。它告诉 OS 它需要多少内存、它需要借用哪些 DLL 的代码、对话框的相关信息等等。它叫做可移植的执行体

（Portable Executable），所有被设计用来在 windows 上运行的程序都需要有一个。

在逆向工程的世界里，这个结构的字节就变得非常重要，因为它给逆向者需要的关于二进制文件的信息。最终，你想要（或需要）改变这个信息，要么让程序做一些和它初衷不一样的事情，要么让程序回到它以前的样子（像保护器让代码变得很难理解之前的样子）。有非常多的 PE 查看器和编辑器工具。我用 CFF Explorer (http://www.ntcore.com/exsuite.php) 和 LordPE (http://www.woodmann.com/collaborative/tools/index.php/LordPE)，不过你可以随意使用你觉得舒服的工具。

大多数的文件也有资源区。包括图像、对话框、菜单、图标和文本字符串。有时候你只看（和修改）资源区就觉得很有意思。此教程的最后我会给你展示一个例子。

5、系统监视工具

在逆向程序时，有时候观察应用程序对系统做出的改变也很重要（在研究病毒和恶意软件时尤其重要），是不是有注册表项被创建或查询？是不是有.ini 文件被创建？是不是有进程被创建，或许用来阻挠软件被逆向？系统监视工具有 procmon、regshot 和 process hacker。后面我们会讨论这些。

6、其他工具和信息

在学习过程中有其他工具我们将用到，比如脚步、脱壳工具、

壳识别器等。一些 Windows API 参考工具也归于此类。这个 API 是巨大的，有时也很复杂。在逆向工程中知道被调用的 API 在干什么是非常有用的。

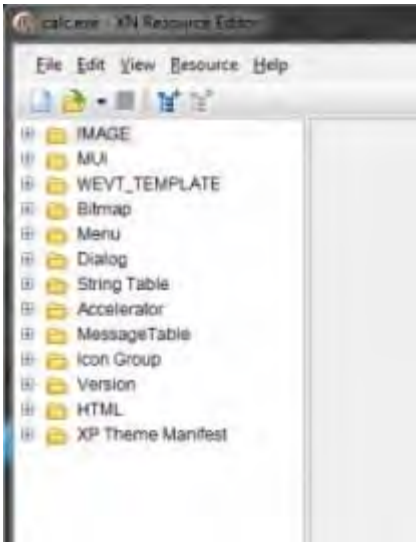
7、啤酒

五、那么，我们开始吧！

即使我们在拥有很少知识的情况下开始，在第一课中我想让你尝尝逆向的滋味。此教程中包含有一个资源查看/编辑器，叫做 XN Resource Editor。

它是免费的。基本上，这个程序可以让你查看 exe 文件的资源区，当然也可以修改这些资源。我已经感觉到你对“这些”的巨大兴趣，你可以修改程序中的菜单、图标、图片、对话框，你可以给它命名。下面我们来改一个试试：

首先，运行 XN。点击顶部的载入图标，找到 Windows\System32\并且载入 calc.exe（你的 windows 默认的位置可能不同）。你会看到一堆可用的文件夹：

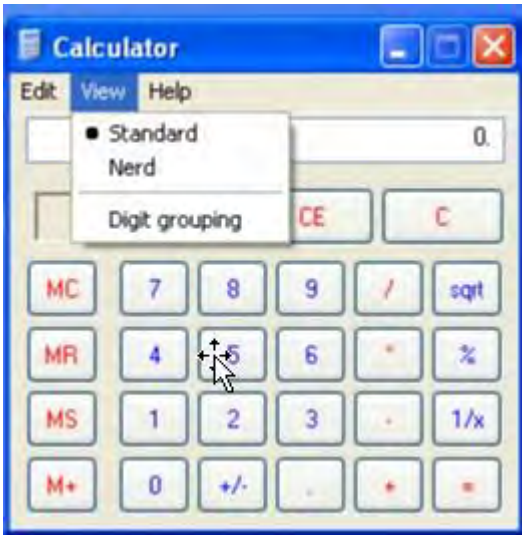


可以看到有 Bitmaps 文件夹（程序显示的任何图片），Menu（顶级菜单项），Dialog（对话框，相关文本和按钮），String Table,，IconGroup 等。你可以对它们为所欲为了。确保另存为一个不同的文件（你肯定不喜欢因为一个 XX 计算器就重装 windows）。细节如下：

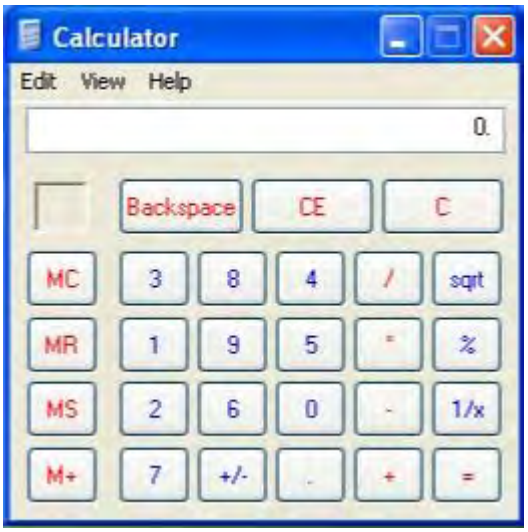
点击那个靠近 Menu 的加号。你会看到以数字命名的文件夹。它是程序中资源的 ID，以便 windows 用来访问相关资源。同样打开该文件夹。你可以看到一个“English (United States)”图标或者类似的东西。如果你点击它，你会看到一个有关菜单外观的图表（你能点击旁边一个类似真实菜单的菜单）。

现在，点击菜单项“Scientific”。那个 Caption 字段应该改成“&Scientific”。那个符号是告诉你“热键”是什么，这里是大写的“S”。假如我想用“e”代替作为热键，应该是这样

的“Sci&entific”。好了，和内建的计算器热键不一样吧？仅仅修改了它们!! 不过，我们做一些其他的吧。在 Captial 字段，将 &Scientific 替换成 “&Nerd”。这将会把菜单项修改成“Nerd”并且使用热键“N”（我们看看菜单中其他选项以确保没有其他的菜单项使用“N”作为热键)。你应该对所有的菜单项做这个工作。现在，到上面的 File 菜单（在 XN Resource 中）并且选择“Save As...”。用不同的名字保存你的新版计算器（最好保存到不同的路径)，然后运行它。



当然，你不需要止步于此。为了开动我同事的榆木脑袋，我修改了他的计算器的所有数字。



如你所见，限制你的只有天空。

延伸阅读（译者注：这里都是英文书，可能部分有中文翻译，未经查证）：

1、汇编语言。*Assembly Language For Intel Based Computers* 中是一本关于汇编的书。你也可以查看一些网站，提供了大量的下载 ，说明 ，示例代码 和帮助。另一个好的资源是“The Art of Assembly”。我将会在今后的某个章节中包含进来，不过你也可以从这里下载。

2、PE 文件结构。最好的资源是微软自己的“An in-depth look into the Win32 Portable Executable File Format”。另一个好的文档（有很多漂亮的图片）是“PE File Structure”。它是一个可下载的 PDF 文件。

3、Windows 操作系统内核。有 Mark Russinovich 的书

“Microsoft Windows Internals”。它想女人的棒球(baseball)一样让人兴奋，不过它是 THE 资源。

4、破解教程。www.Tuts4You.com 就是这样的一个地方。

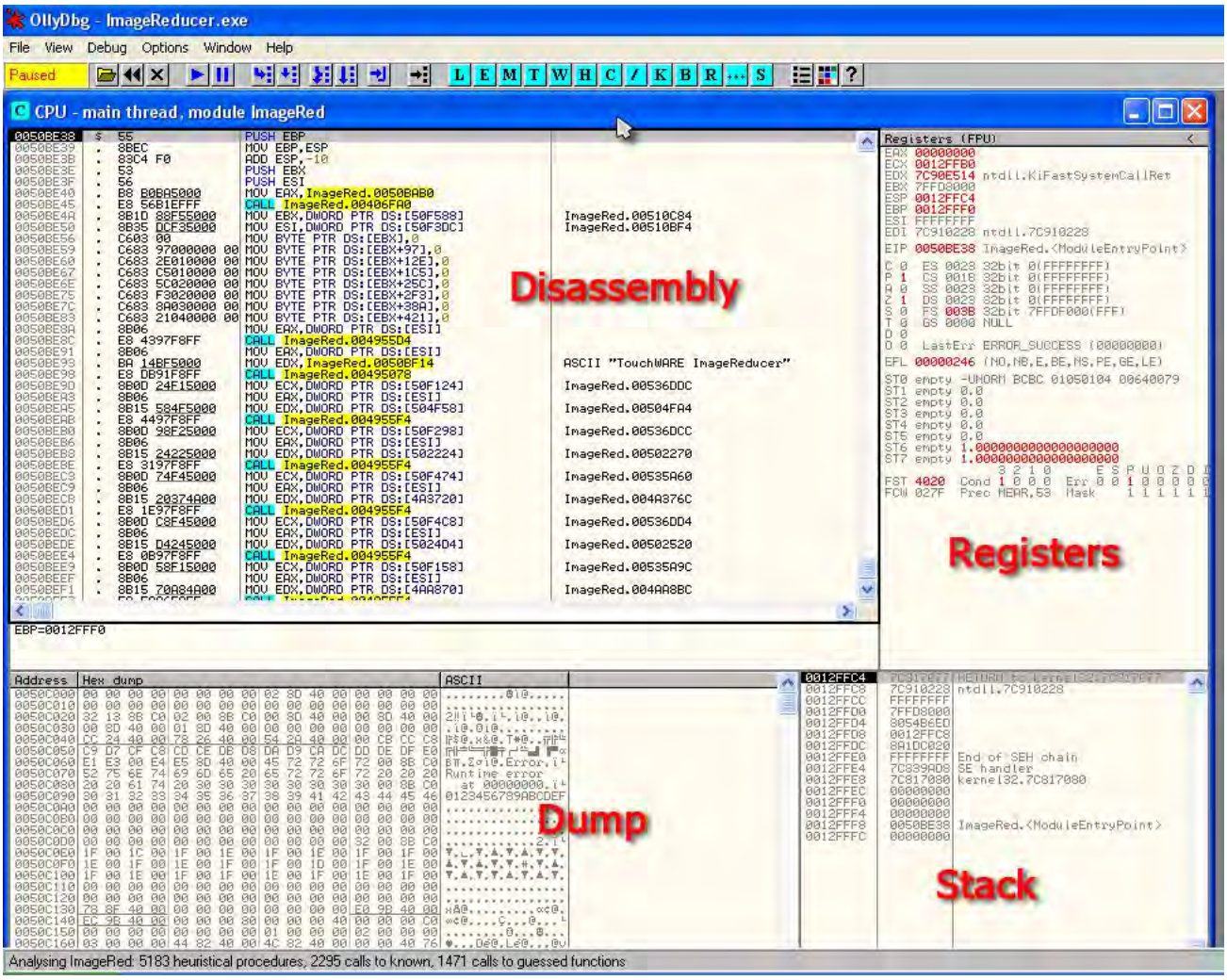
教程二、介绍 Olly Debug

一、什么是 Olly Debugger？

援引作者 Oleh Yuschuk 的话“OllyDbg 是一个用于微软 Windows 的 32 位汇编级分析调试器”。在没有源代码的情况下，二进制代码分析非常有用。Olly 也是一个动态调试器，意味着它允许用户在程序运行时修改一些东西。这在实际分析二进制文件尝试找出程序工作原理时非常的重要。Olly 有许多许多很棒的特性，这就是为什么它是逆向工程领域的天字第一号调试器（至少在 Ring3 级是，我们马上就接触到了）。

二、概览

下面是 Olly 的主界面图片，上面有一些说明性的标签。



打开 Olly 时有一个默认的子窗口是 CPU 窗口。这是那个“大图片”中大部分数据所在的地方，如果你什么时候把它关掉了，只需要点击工具栏中那个“C”图标就行了。窗口被分成了四个部分：反汇编区（Disassembly），寄存器区（Registers），堆栈区（Stack）以及内存数据区（Dump）。下面是对每个区的说明：

1、反汇编区

该部分主要包含了二进制文件的反汇编代码。这是 Olly 显示二进制信息的地方，包括操作码（opcode）和翻译的汇编代码。

第一列是指令的地址（内存中地址）。第二列按汇编语言叫操作码，每个指令至少对应一条代码（有很多对应多条）。这才是CPU真正需要并且是唯一能读懂的代码。这些操作码组成了“机器语言”，也就是计算机的语言。如果你看过二进制的原始数据（用十六进制编辑器），你除了看到这些操作码的字符串以外，就没有其他的了。Olly的一个主要工作是将这些“机器语言”“反汇编”成人类可读的汇编语言。第三列是汇编语言。不过退一步讲，对于不太懂汇编的人来说，汇编不比操作码好多少。不多随着学的越来越多，汇编提供了远多于代码所做的更多信息。

最后一列是Olly对于该行代码的注释。有时候会包含所调用API 的名字，比如 CreateWindow 和GetDlgItemX。Olly也会尝试通过将非API调用命名来帮助我们理解代码，上图中的“ImageRed.00510C84″ 和“ImageRed.00510BF4″就是此类情况。退一步讲，这些东西不是那么有用，Olly也允许我们将它们修改成一个有意义的名字。你也可以在该列写自己的注释。只要双击该列中的某行，就会弹出一个对话框让你输入注释。这些注释会自动保存到下一次。

2、寄存器区

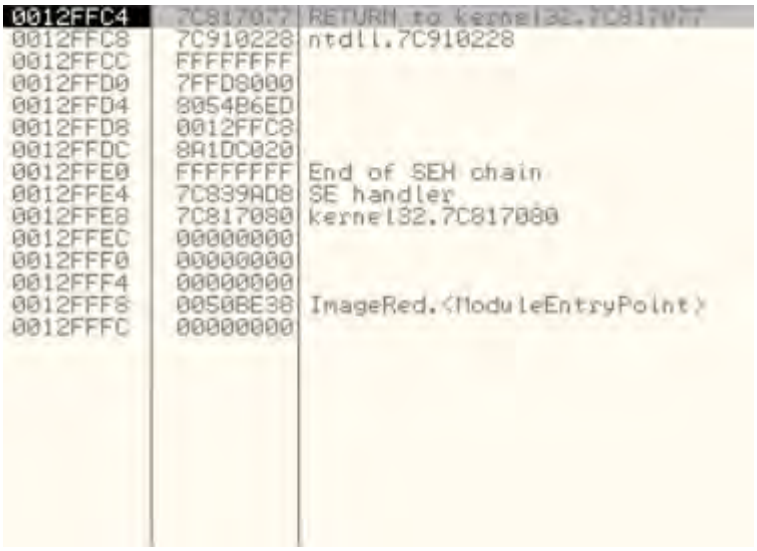
每个CPU都有一组寄存器。用来临时存放数值，和高级语言中的变量很像。下面是寄存器窗口的特写（有标记）：

顶部实际上是 CPU 的寄存器。如果值有变化，寄存器会从黑色变为红色（对于观察数值的变化真的非常有用）。你也可以双击任何一个寄存器来改变它的内容。这些寄存器能做很多事情，后面会讨论更多。

中间那块是标志寄存器，是 CPU 用来标记代码中一些事情的发生（两个数相等、一个数比另外一个大等等）。双击其中一个标志寄存器就可以修改它。这些玩意儿在我们的学习过程中扮演着重要的角色。

底下的部分是 FPU，或者叫浮点运算器。只要 CPU 执行任何涉及小数点的运算就会用到它们。逆向者很少用到它们，主要是在我们接触加密的时候用。

3、堆栈区



堆栈是内存中的一段区域，用于存储二进制数据的临时列表。这些数据包括指向内存中地址的指针，字符串，制造者（makers)及大部分重要的数据,还包括函数调用后的返回地址。当程序中的一个方法调用另一个方法时,控制权需要转移到新方法以便于它能够返回。CPU 必须知道一个新方法执行完后它是从哪被调用的，CPU 能够返回到它被调用的地方，继续执行该调用之后的代码。堆栈就是 CPU 保存返回地址的地方。

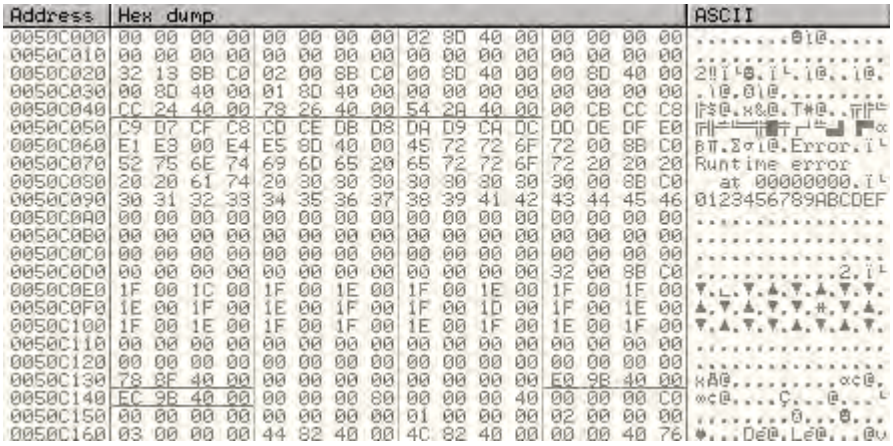
关于栈你需要知道一点，他是“先进后出”的数据结构。打个常用的比方，就像是自助餐厅里下面带有弹簧的一摞盘子一样。当你向顶部“压（PUSH)”进一个盘子，下面的所有盘子都会被往下压。当你移除（“POP”）顶部的一个盘子，下面的所有盘子都会被往上提升一级。下个教程我们会实际看看，所以这里别担心看不太懂。

图片中，第一列是每一个数据成员的地址，第二列是十六进

制的 32 位数据，如果 Olly 能够分析出来的话，那么最后一列是 Olly 关于数据项的注释。如果你注意看第一行的话，会看到“RETURN to kernel…”的注释。这里是 CPU 放在栈上的一个地址，以便于在当前的函数执行完后，CPU 知道返回到哪。

在 Olly 中，你可以右键点击堆栈区，并且选择“修改（modify)”来更改内容。

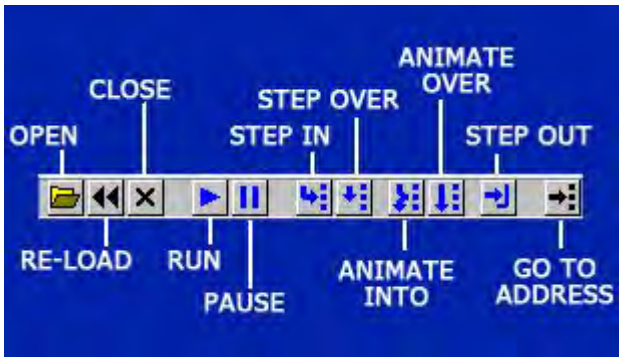
4、内存数据区



在教程的开始，当我们讨论 CPU 从二进制文件中读取的原生“操作码”时，我提到过你能在十六进制查看器中看到原始数据。不过，在 Olly 中你不需要这么做。因为内存数据区就是一个内置的十六进制查看器，以便于你查看原始的二进制数据，只查看内存中的而不是磁盘上的。通常对于同样的数据有两种查看方式，十六进制的和 ASCII 的。图片中右边的两列就是（第一列是数据驻留内存中的地址)。Olly 允许修改这些数据的显示方式，后面的教程就会看到。

三、工具栏

不幸的是，Olly 的工具栏给大家留下了一点念想（尤其是当英语并不是作者的母语）。我将左边的工具栏图标进行了注释：

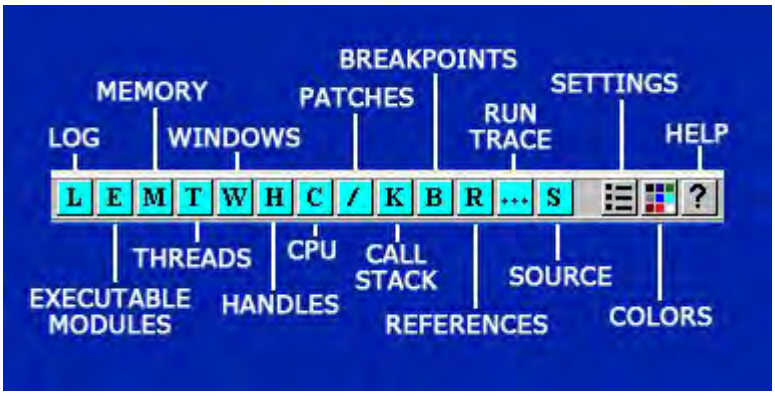


这些都是控制代码运行的主要工具。记住这些，尤其是你开始使用 Olly 的时候，这些按钮的所有功能都可以从“调试（Debug）”菜单的下拉菜单中访问到。如果你不知道某些东西是什么，你可以从菜单中看到。

关于一些图标我要多说几句。“Re-load”是用来重新启动应用并暂停在入口点处。所有的补丁（后面会看到）都会被删除，一些断点会失效，应用程序也不会运行任何代码。好吧，大部分情况下是这样的。“Run” 和 “Pause”做的就是你看到的那样。“Step In”意思是运行一行代码然后暂停，如果有的话它会跟进函数的内部。 “Step Over”做同样的事情，不过它会跳过对另一个函数的调用。“Animate”有点像 Step In 和 Step Over，不过它特别慢好让你观察。这个你用的不多，不过有时候看代码运行也挺有意思的，尤其是遇到多态二进制的时候能够观察到代

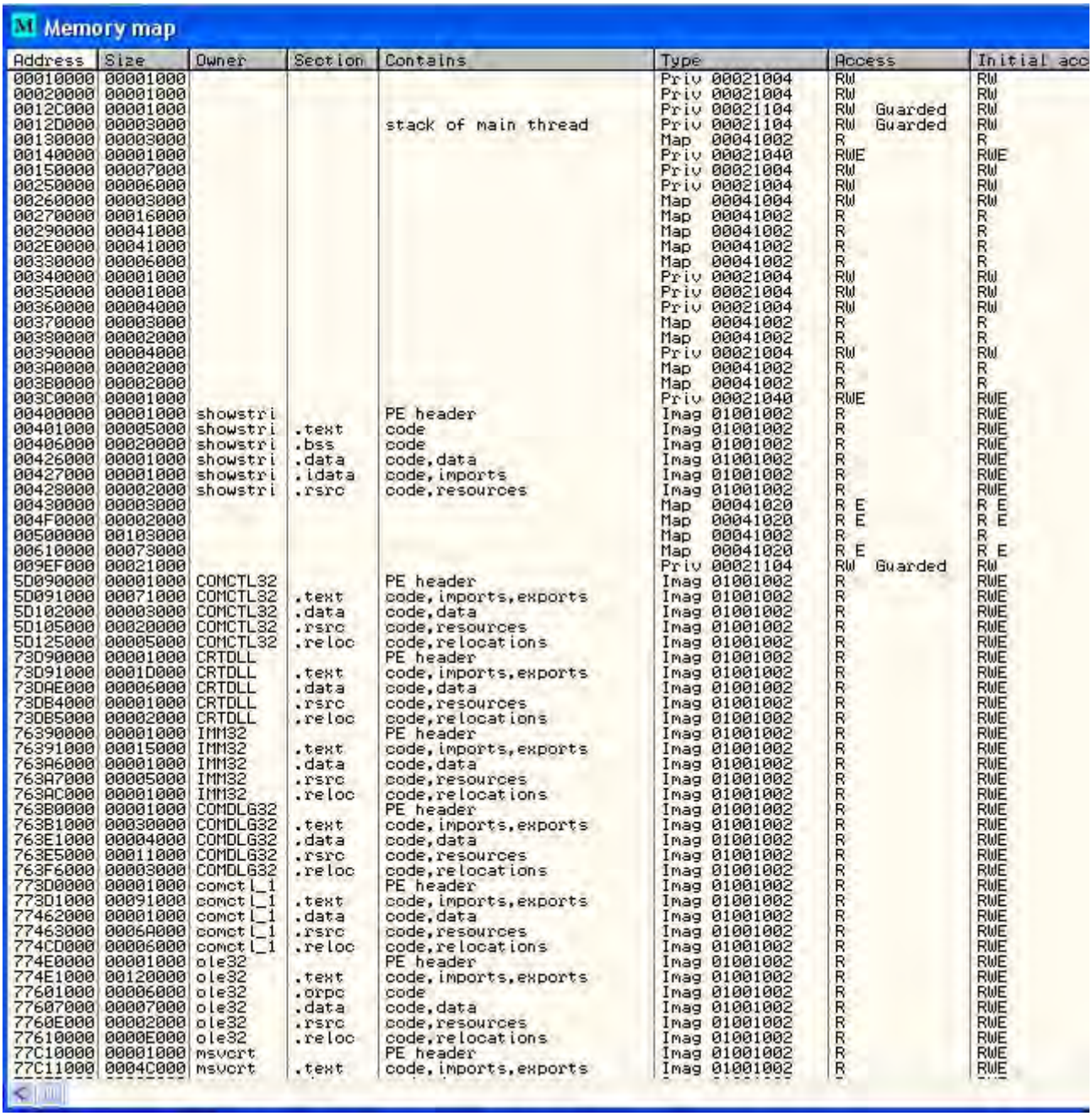
码的变化。讲的有点超前了.....

下面是各窗口的按钮图标（更加有点神秘）：



点击其中的任何一个按钮都会弹出一个窗口，有些你会经常用到，而有的却很少用。看这些字母并不是很直观，这点你可以像我学习，把它们都点一遍直到你找到你需要的那个。每一个都可以通过“View”菜单来访问，所以在第一次征程时你可以获得些许帮助。下面我会介绍最常用的窗口：

1、(M)emory——内存映射窗口



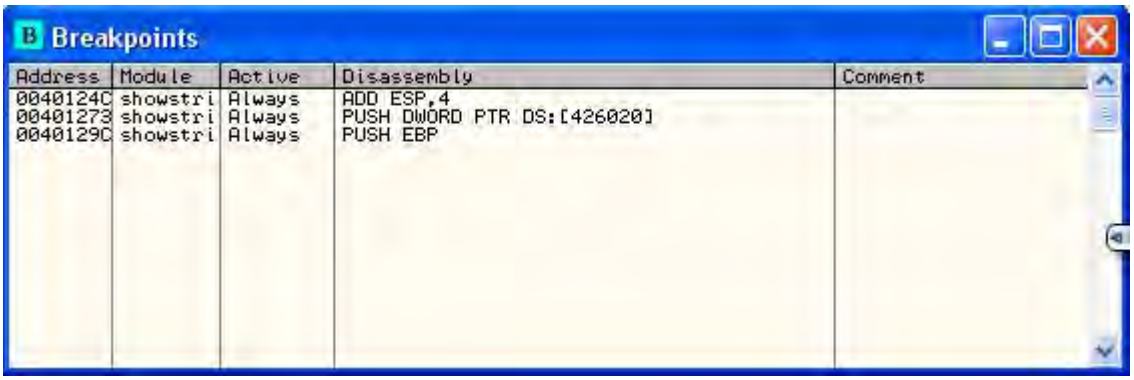
内存窗口显示程序已经分配的所有的内存块。它包括正在运行的程序的主段（本例中，是 Owner 列中的“Showstr”）。在下面你能看到很多其他的段，这些都是程序载入进内存的 DLL 的，准备将来用的。如果你双击其中的任何一行，都会打开一个显示

该段的反汇编代码（或十六进制数据）的窗口。这个窗口也显示了块的类型和访问权限、大小以及该段载入内存的地址。

2. (P)atches——补丁窗口

该窗口显示的是你做的任何“补丁”，即对原始代码的任何修改。注意那个状态（State 列）是激活的（Active）。如果你重新载入应用程序（通过点击 re-load 图标），这些补丁就会失效。为了简便的使它们重新生效（或失效），点击期望的补丁以及敲击空格键。这可以打开或关闭补丁。注意那个“Old”和“New”列，显示的是原始的指令和修改后的指令。

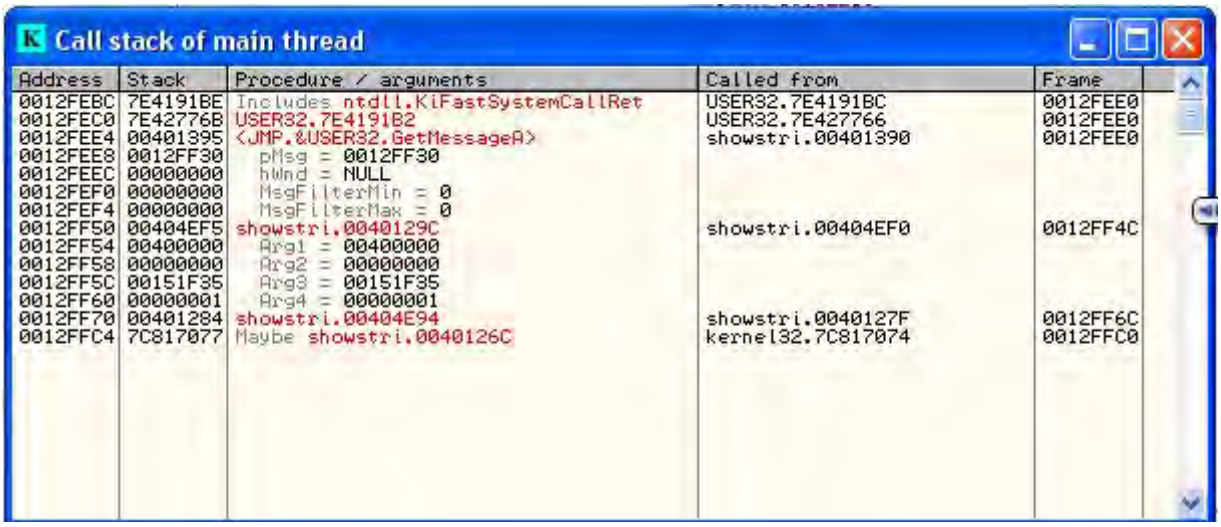
3. (B)reakpoints——断点窗口



该窗口显示了当前所有断点设置的位置。这个窗口将会是你的好朋友

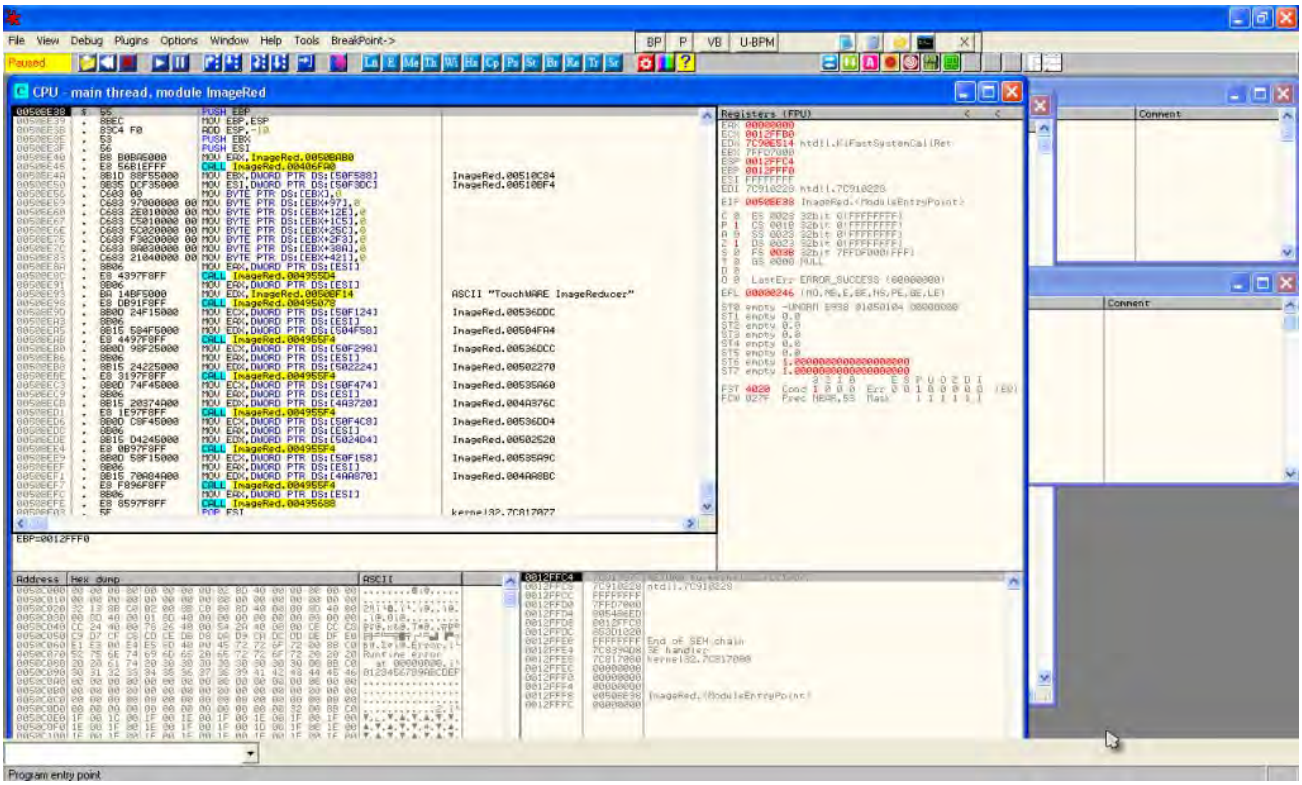
4. (K)all Stack——调用栈窗口

（哎呀，我知道为什么初学者记这些图标比较难了.....）



这个窗口与前面看到的“堆栈区”不一样，它显示了更多信息，有关于代码中的调用、发送给这些函数的值以及其他的东西。不久我们会了解到更多。

下一教程，我会包含我的经过“升级”的 Olly 版本，有些是你一看就明白的按钮。这里有张图片



四、上下文菜单

本教程的最后，我会快速介绍 OllyDbg 的右键菜单。它是许多操作产生的地方，所以你最少应该熟悉一下它。右键反汇编区的任何地方都会调出该菜单：

我只会介绍最常用的几项。随着经验的增多，你最终会遇到那些较少用到的选项。“Binary”菜单项允许你按字节编辑二进制数据。在这里你可以将埋在一堆二进制数据中的“未注册”几个字改成“已注册”。“Breakpoint”菜单可以设置断点。断点分好几种，下一章我会讲到。“Search for”有一个相当大的子菜单。这里你可以搜索类似字符串、函数调用等二进制数据。

“Analysis”菜单会强制 Olly 重新分析当前正在查看的代码段。有时候 Olly 会对你正在查看的是代码还是数据感到困惑(记住，它们俩都只是一些数字),这个可以强制 Olly 将你正在看的内容当做是代码，并且尝试猜测该部分看起来应该是什么样子的。

注意，我的菜单看起来和你的可能不太一样，因为我装了一些插件，这些插件在菜单中添加了一些功能。不过别担心，后面的教程中我会介绍这些菜单的。

教程三：OllyDbg 的使用（上）

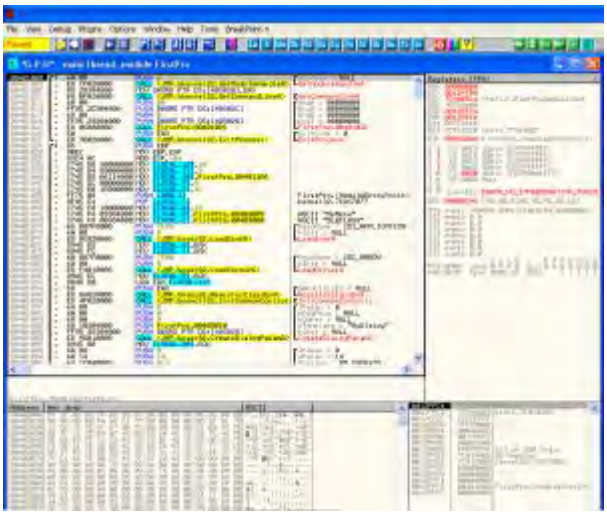
本章中我将会介绍 OllyDbg 的使用。Olly 有许多的功能，唯一学好它们的方式是实践和练习。也就是说，本教程也只是给你一个简单的概述。此教程不会涉及额外的内容，后面会进行重点讨论。到最后，你应该会比较好的掌握 Olly。

本章包含了一些文件。你能够下载那些文件，以及可以在这里下载到次教程的 PDF 版本。

它们包括一个我们将在 Olly 中用到的二进制文件、一个 Olly 备忘单、我使用的外观上有些不同的 Olly 以及一个新的 ini 文件。你可以用这个 ini 文件替换掉 Olly 默认的 ini，可以给新人提供一些帮助(感谢伟大的 Lena151 做的这些)。你可以从 这里 直接下载或者从教程页面下载。如果你更愿意用原版的 Olly，你可以从这里下载。

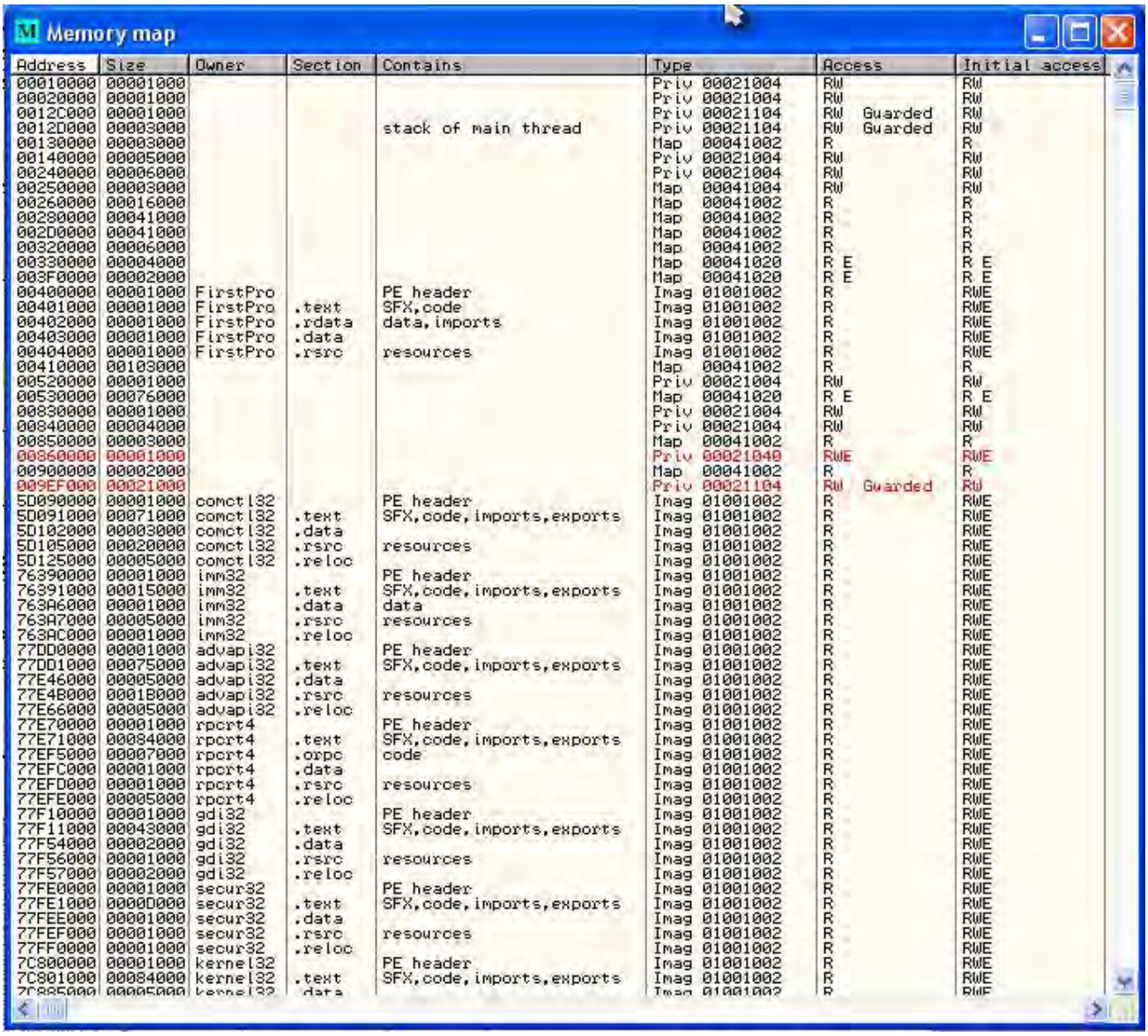
一、载入应用

第一步是将目的二进制文件载入 Olly。你可以将二进制文件拖放到 Olly 的反汇编窗口，或者点击顶部工具栏中的载入图标选择目的文件。我们这里载入 “FirstProgram.exe”，可以从本网站下载。Olly 会进行分析（Olly 的底部状态栏会显示分析进程）然后停在程序的入口点（EP）:



需要注意的第一件事是 EP 的地址是 401000，就是图片中的第一列。这是可执行文件的一个相当标准的起点（该可执行文件至少没有加过壳或混淆过）。如果你的看起来不太一样，并且 Olly 没有停在 401000，你可以尝试点击 Appearance 菜单，然后选择 debugging options，点击 “Events” 标签，并且确保 “WinMain(if location is known)” 被勾选上。然后重启应用。

让我们给 “FirstProgram.exe” 的内存空间占用情况来张快照。点击 “Me” 图标（如果你使用的是不同版本的 Olly 的话应该是 “M”）：



如果你看地址那一列，你会看到 401000 那行包含有大小 1000、名称“FirstPro”（FirstProgram 的简写形式）、区块名“.text”、包含里是“SFX,code”。随着学习进度的展开，我们就会知道 exe 文件中有不同的区块，包含不同的数据类型。该区块中是程序的“代码”。它有 1000 字节大，从内存的 401000 开始。

在这个的下面你会看到 FirstProgram 的其他区块。其中.rdata 区包含着数据，其导入地址是 402000，地址 403000 的.data 区中什么都没有。最后的那个.rsrc 区中存有资源（比如对话框、图片、文本等）。要注意的是这些区可以叫任何名字，这个完全依赖于程序员。

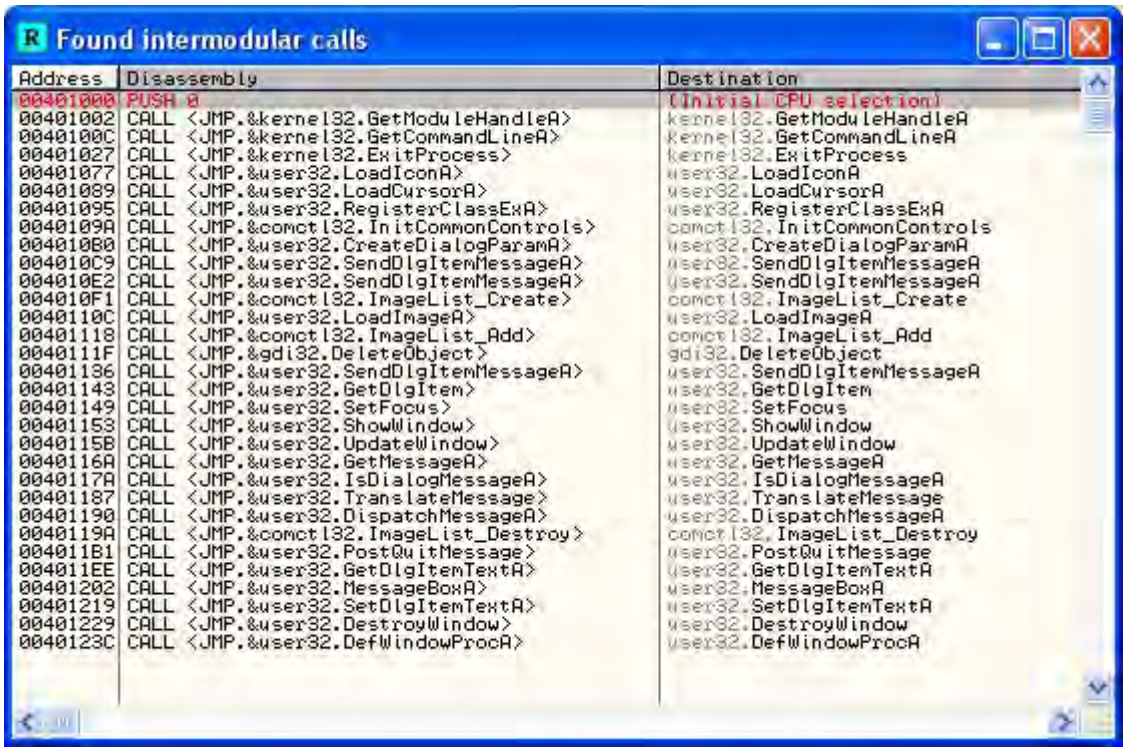
你可能会问为什么.data 区是空的。好吧，它事实上就是那样。它一般包含全局变量和随机数据。Olly 只是选择了不显示，因为它确实不知道那里存储了哪种数据。

区段的顶部是一个叫做“Pe Header”的区块。这是一个非常重要的区，一个我们会在将来文章中深入探讨的区。不过对于目前来说，我们只需要知道它对于 Windows 就像一本指令手册一样，用来按步将文件载入内存，程序运行需要多少空间，还有其他某些事情等。它在大约所有 exe 的头部（DLL 也是一样）。

如果你继续往下看，你可以看到不只是 FirstProgram 程序，还有其他的文件。我们看到有 comct132，imm32,gdi32,kerne132 等。这些 DLL 是我们程序运行所需要的。DLL 是函数的集合，我们的程序能够调用那些 Windows 已经提供的（或者其他程序员提供的）函数。比如打开对话框、比较字符串、创建窗口以及类似的功能。统称为 Windows API。程序使用这些函数的原因是，假如我们写每一个用到的函数，仅仅显示一个消息框就需要数千行

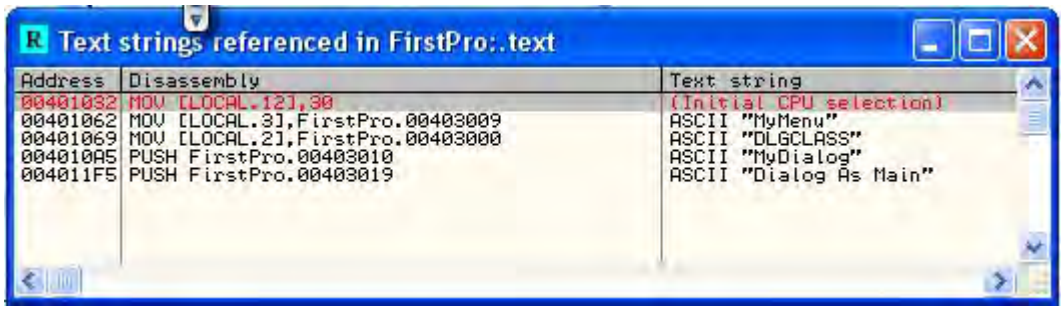
的代码。然而，Windows 已经提供了像 CreateWindow 这样的函数来为我们做这些工作。对于程序员来说这使得编程要简单的多。

你或许会问这些 DLL 是如何进入我们的地址空间的，windows 是怎么知道哪一个是我们需要的。好吧，这些信息是存储在上列出的 PE Header 中的。当 Windows 将我们的 exe 载入内存时，它会检查头并找出 DLL 的名字，以及每个 DLL 中我们程序需要的函数，然后将这些函数载入我们的程序内存空间，以便于我们的程序调用它们。每个程序被载入内存时，它所需要的 DLL 也会被载入它的内存空间。可以想象得到，当有好几个程序当前都需要被载入内存并且都需要某个特定的 DLL 时，那么有些 DLL 就有可能被载入内存好几次。如果你需要准确的知道我们的程序调用了哪些函数，你可以右键点击 Olly 的反汇编窗口，选择“Search for”——>““All Intermodular Calls”。会显示如下图:



这有点惊奇，不过这个列表非常的小。通常，对于一个商业产品来说，需要数百或数千函数。不过因为我们的程序太简单了，它需要的不是很多。你想想我们的程序干了什么，看起来好像是那么多的函数只完成了如此简单的功能！欢迎来到 Windows。该窗口首先显示了 DLL 的名字，紧跟着的是函数的名字。比如，User32.LoadIconA 是在 DLL User32 中，函数名字是 LoadIconA。该函数通常用来载入窗口左上角的图标。

下一步，我们搜索下程序中的所有字符串。右键点击反汇编窗口，选择“SearchFor” -> “All Referenced Text Strings”:

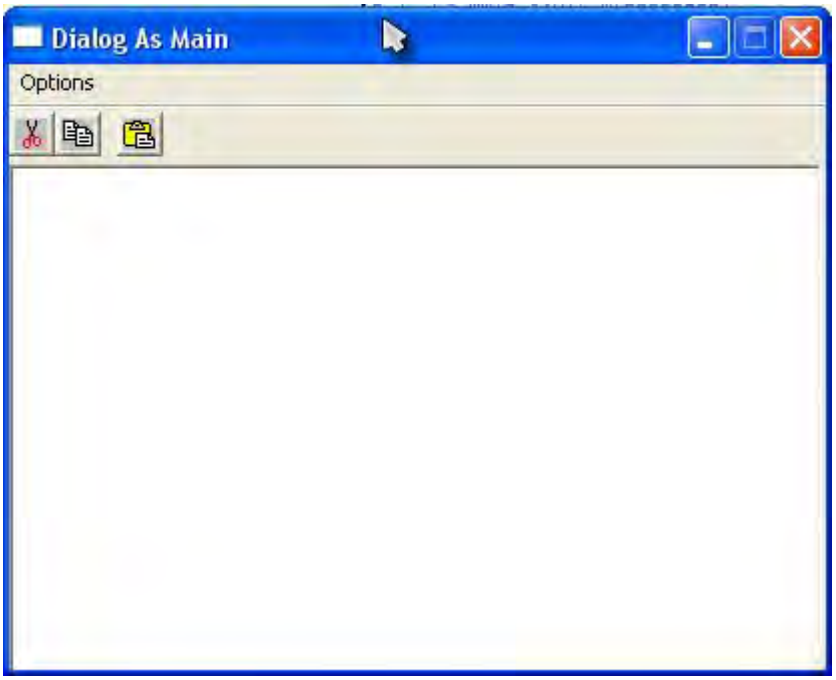


该窗口显示了我们程序中所有能找到的字符串。因为程序非常简单，所以这里只有一点。大多数的程序如果没有加壳或混淆的话，都有多得多的字符串（有时能达到十万）。这种情况下，你有可能一个也看不到！加壳工具这样做的原因是逆向工程师（至少新人是这样）严重依赖字符串来查找重要的函数。而删除了字符串后就会难的多。想象一下，如果你搜索字符串然后看到了“Congratulations! You entered the correct serial（恭喜！你输入了正确的序列号）”会怎么样？嗯，这对于逆向来说是巨大的帮助（我们会一次又一次的看到这个）。另外，双击其中的字符串，你会来到反汇编窗口中使用该字符串的指令那。这是一个很好的特性，你能够正确的跳转到使用字符串的代码。

二、运行程序

如果你看 Olly 的左上角的话，会看到一个黄色背景的小区块，里面写着“暂停（Pause）”。意思是程序已经暂停了（本例中是在开始的时候），等着你进行其他操作。所以，咱们开始干一票吧！按一下 F9（或者从“Debug”菜单中选择“Run”）。一

会儿后，我们的程序会弹出一个对话框（它有可能显示在 Olly 的后面，所以最小化 Olly 以确保能看见窗口）。



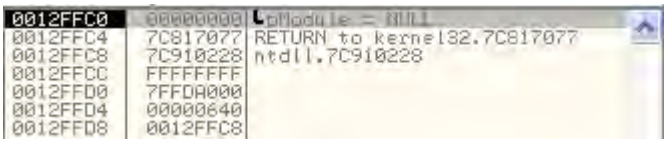
刚才显示“Pause”的地方现在应该显示的是“Runing”。意思是程序正在运行，不过是在 Olly 中运行的。你可能会与我们的程序进行一些交互，看看它是如何工作的以及它干了些什么。如果你不小心关了它的话，返回到 Olly 并且按下 Ctrl+F2（或选择 Debug->Restart）以重新载入程序，然后你可以点击 F9 让程序再一次运行起来。

现在照着做：程序运行的时候，点击回到 Olly 中，然后点击暂停图标（或点击 F12，也可以点击 Debug->Pause 菜单）。即使我们的程序正在运行，该操作会让程序暂停在内存中的任何地方。如果这时候你想看看程序，你会发现挺有意思的（程序一点

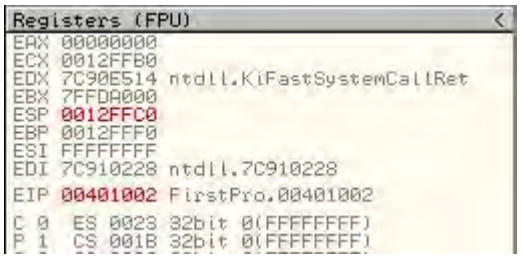
也不会显示出来)。这是因为当程序暂停的时候，Windows 不会更新视图。现在再一次点击 F9，你会发现你又可以和程序进行交互了。如果有什么问题的话，只需要点击那个双左尖括号图标或 Debug-restart (或者 ctrl-F2)，程序就会重新载入并暂停在入口处。如果你需要的话，你可以再一次运行它。

三、单步运行程序

运行一个程序确实挺爽，不过你却得不到有关于程序运行的太多信息。让我们试试单步运行。重新载入应用程序（重新载入按钮、Ctrl+F2 或 Debug->restart），然后我们会暂停在程序的开始处。按一下 F8，你就会发现当前的行选择器下移了一行。Olly 运行了一行指令，然后又暂停了下来。如果你够激灵的话，就会发现堆栈区下滚了一行，并且在顶部有了一个新的入口。



这是因为我们执行了一条指令，“PUSH 0”往堆栈里“压”了一个 0。在堆栈中的显示是“pModule=NULL”。NULL 是 0 的另一个名字。你有可能也注意到了那个寄存器区，ESP 和 EIP 寄存器变红了。



当一个寄存器变红的时候，这就意味着最后执行的指令修改了该寄存器。本例中，ESP 寄存器（用来存放指向栈顶的地址）增加了 1，因为我们向栈中压了一个新值。EIP 寄存器增加了 2，其中存放了将要运行的指令的地址。因为我们已经不在地址 401000 了，而是在 401002。因为上一个运行的指令是两个字节长。我们现在暂停在下一个指令处。这个指令是在 401002，这正是当前 EIP 的值。

Olly 现在暂停的指令是一个 CALL。CALL 指令意味着我们要临时暂停在我们当前所在的函数中，然后去运行另一个函数。这类似于高级语言中的方法调用，举个例子:

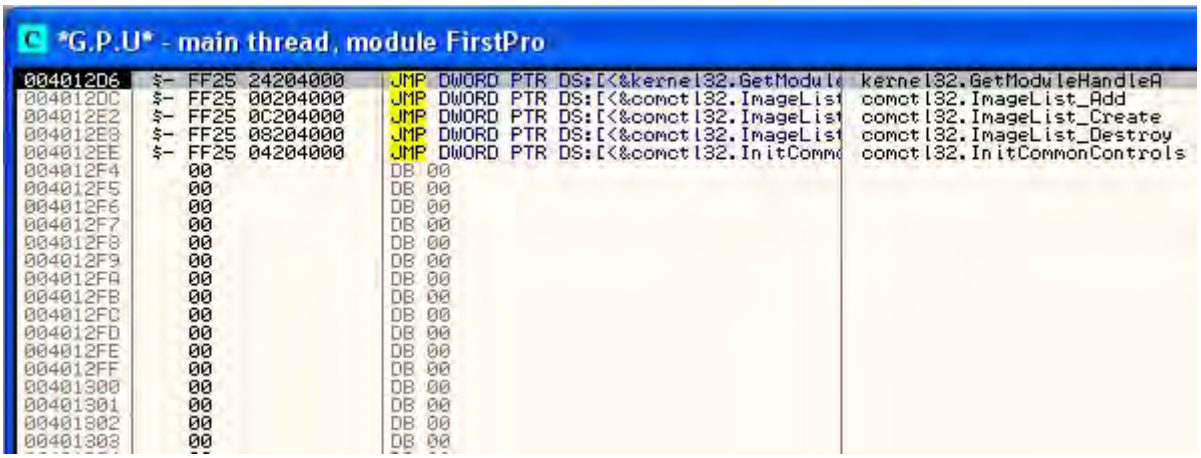
```
int main()
{
    int x = 1;
    call doSomething();
    x = x + 1;
}
```

这段代码中，我们首先让 x 等于 1，然后呢我们要在逻辑上暂停这行代码，转而去调用 doSomething()。当 doSomething()执行完毕后，我们会返回我们原来的逻辑，然后将 x 加 1。

当然，在汇编语言里也是一样。我们首先往栈中压了一个 0，现在呢我们又想调用一个函数，例子中调用了 Kernel32 d11 中的 GetModuleHandleA()：

好，再按一次 F8。当前的行指示器会下移一行，而 EIP 仍然会保持红色并且加了 5 因为刚刚运行的指令是 5 字节大小），堆栈也回到了它原来的地方。刚刚发生的这些是从我们按下 F8 开始的，F8 的意思是 “Step-Over（单步步过）”，CALL 中的代码被调用，然后 O11y 暂停在了 CALL 的下一行。也就是 CALL 中的程序执行了也做了某些事，但是我们跳过去了。

好了，现在我们看看其他的选项。重启程序（Ctrl+F2），按下 F8 步过第一条指令，不过在 CALL 指令上我们这次按 F7。你会注意到整个窗体都变得不一样了：



这是因为 F7 “Step-In（单步步入）” 那个 CALL，意思是 O11y

做了这个调用并暂停在了新函数的第一行。这种情况下，CALL 跳转到了一个新的内存区域（EIP=4012d6）。理论上，如果我们按行通过这个新函数的话，我们最终还是会回到将我们带进来的那个 CALL 后面的语句。当然，有快捷键可以完成同样的功能，不过目前来说，咱们还是重启程序从头来吧。因为我怕教的太多容易忘。

现在我们暂停在了程序的开始，按下 F8（单步步过）4 次，我们会停在如下图的语句处:

你会看到在一块的四个 PUSH 语句。这回当你四次按下 F8 的时候，注意观察堆栈区，会看到栈的增长（确实是向下增长，还记不记得那个盘子的例子？）。我觉得我们开始理解什么是压栈了.....

你可能会问我们为什么要将这些乱七八糟的数字往栈里压。本例中这四个数字是作为参数传递给函数的(那个函数是在地址 401021 处)。我们将前面的那个高级语言程序做一点点修改就会比较清楚了:

```
int main()
```

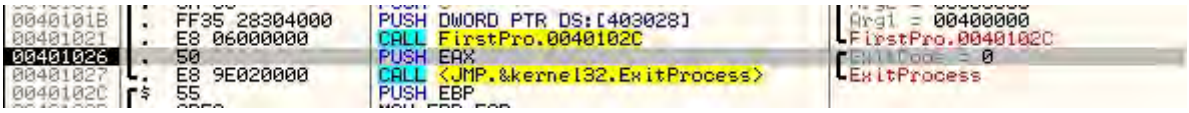
```
{
    int x = 1;
    int y = 0;
    call doSomething( x, y );
    x = x + 1;
}
```

这里我们声明了两个变量 x 和 y，并且将它们传递给了 doSomething()函数。doSomething 函数将会（可能）对这些变量做些什么，然后将控制权还给调用该函数的程序。通过堆栈是将变量传递给函数的主要方法之一：每个变量被压进堆栈，然后调用函数。然后在函数中，这些变量被访问到。通常 PUSH 指令的逆操作是 POP。

堆栈并不是做这件事的唯一方法，它只是最常用的。这些变量也可以被放到寄存器中，然后在被调用的函数内部访问寄存器。不过本例中，我们程序的编译器选择将变量放到堆栈中。在你学了汇编语言后，这些东西都会变得清晰（你正在学习汇编语言，不是吗？）。后面我们还会复习几次的。

现在，如果我们再按一次 F8，你会注意到 Olly 的工具栏中会显示“Runing”，我们程序的对话框就会显示。这是因为我们单步步过了那个 CALL，说明那个 CALL 中存在程序的大部分。这个调用的代码是等待用户进行一些操作的循环，所以我们永远也不会将控制权交给 CALL 的下一行。那么，让我们修复它.....。

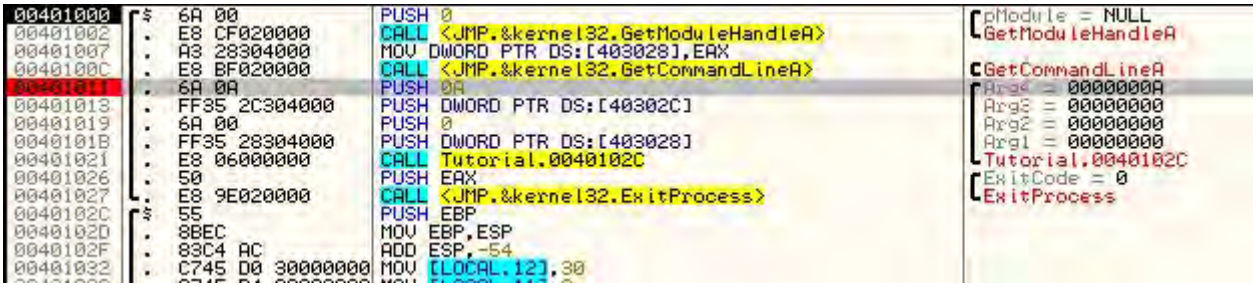
点击回到我们的程序，点那个关闭按钮来结束应用。OllyDbg 会立即暂停在那个 CALL 的下一行:



你会注意到我们的程序也消失了。那是因为，在那个 CALL 的某个地方，对话框窗口被关闭了。如果你看下一行，你会发现我们正准备调用 kernel32.dll -> ExitProcess。这是一个停止应用程序的 Windows API。所以，基本上 OllyDbg 在窗口被关闭了之后就暂停了，不过是在程序确实被终止之前。如果你这时按 F9，程序就会终止，OllyDbg 的活动栏就会显示“Terminated（已终止）”，我们就再也不能调试任何东西了。

四、断点

我们试试别的东西，重新载入应用（Ctrl+F12），然后在地址 401011 处的第二列上双击（也就是双击那个“6A 0A”操作码）。然后地址 401011 就会变红:



你刚才做的就是在地址 401011 处设置断点。当 OllyDbg 到达该处时，断点就会强制 OllyDbg 暂停。有好几种不同的断点会因为不

同的事件而阻止程序运行。

1、软件断点（Software Breakpoints）

软件断点就是将断点所在地址处的字节用 0xCC 操作码替换掉，也就是 int 3 指令。这是一个特殊的中断，用以告知操作系统调试器希望在这里暂停，并且在执行该指令之前将控制权交给调试器。你不会看到指令被修改成 0xCC，因为 Olly 在背后做了这个。当 Olly 遇到异常时它会设一个陷阱，让用户做他们希望做的事。如果你选择让程序继续运行（通过运行它或单步运行），0xCC 操作码就会被原来的操作码替换回来。为了设置一个操作码，你可以双击操作码那一列，也可以先选中你想设置断点的那一行，然后右键点击它，选择 Breakpoints->Troggle（或按下 F2）。要删除断点你可以双击同一行，或右键点击选择 Breakpoints->Remove Software Breakpoint（或再次按下 F2）。

现在我们在 401011 处设置了一个 BP（Breakpoints），让程序暂停在第一行指令处。按下 F9，程序将运行并在我们设置的断点处暂停。

这里我告诉大家一些有用的东西。点击工具栏上的“Br”图标或选择菜单中的 View->Breakpoints。你会看到一个断点窗口，里面显示了我们设置的断点。

通过它你可以快速的浏览所有你设置的断点。你可以双击任何一个断点，然后反汇编窗口就会跳转到那个断点处（如果你没有改变程序控制流的话，EIP 仍然停在原来的地方。双击 EIP 寄存器会回到当前的行，并准备执行下一行）。

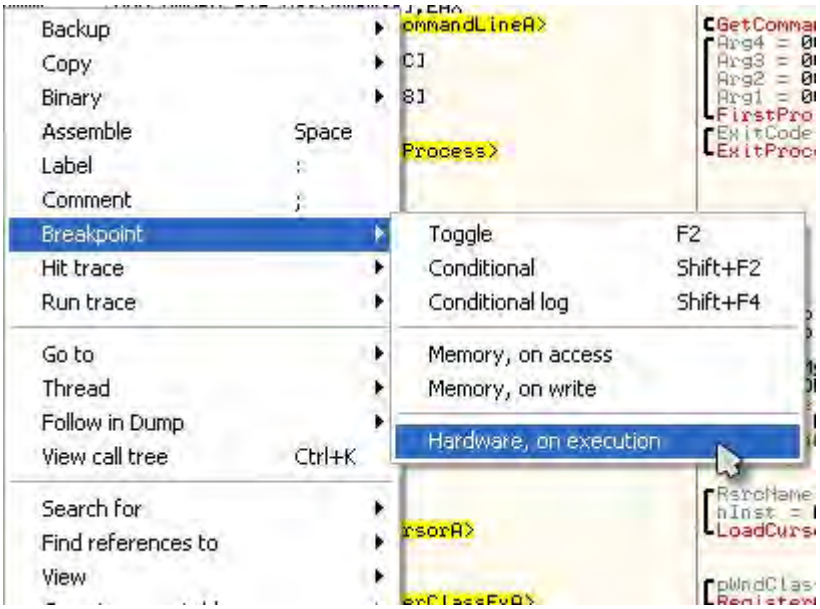
如果你选中一个断点，然后敲击空格键，断点就会在可用和禁用之间来回切换。你可以选中一个断点，然后敲一下“Del”键就会删除断点。

最后，重启程序，打开断点窗口，选中 401011 处的断点。敲一下空格键，然后“Active”列将会变成“Disable（禁用）”。现在运行程序（F9），你会发现 Olly 不会停在我们设置的断点处，因为它被禁用了。

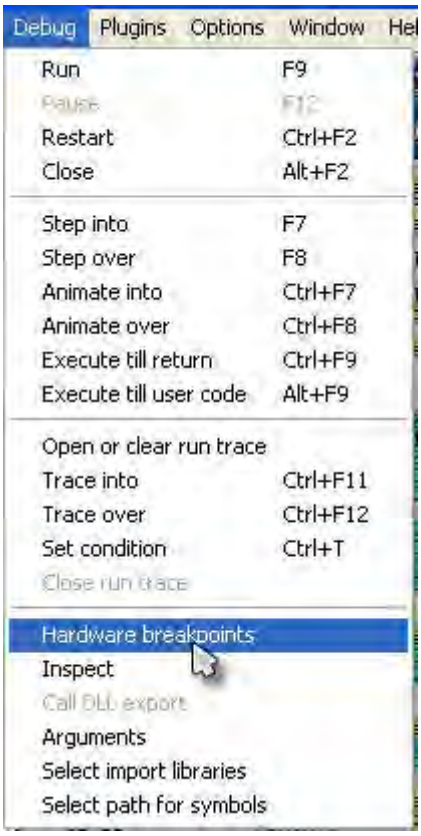
2、硬件断点（Hardware Breakpoints）

硬件断点使用的是 CPU 的调试寄存器。CPU 内建的有 8 个寄存器，是 R0-R7。即使芯片中内建了 8 个，但是我们只能使用四个。它们可以被用来断在内存区的读、写和执行。硬件断点和软件断点的不同之处在于，硬件断点不会修改进程的内存，所以它

更可靠，尤其是在加壳或被保护的软件中。通过右键点击相关行可以设置硬件断点，选择 Breakpoints，然后选择 Hardware,on Execution。



唯一查看你已经设置的内存断点（译者注：这里应该是硬件断点）是打开“Debug”菜单，选择“Hardware Breakpoints”。有个插件可以提供方便，不过我们后面再讨论 。

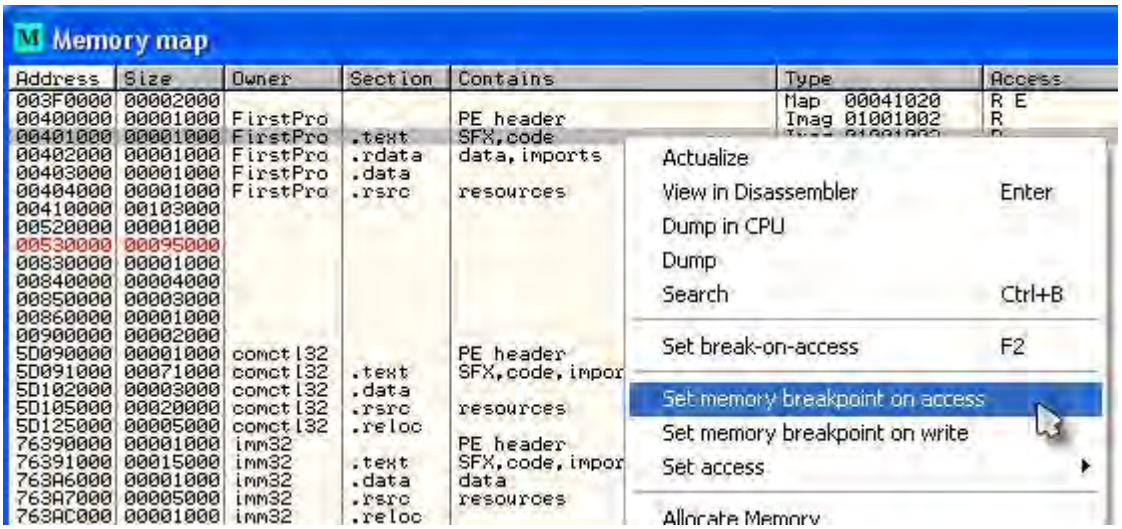


3、内存断点（Memery Breakpoints）

有时候你想查找程序内存中的字符串或在常量，但是你又不知道程序在内存的什么地方。你可以用内存断点来告诉 Olly 只要任何一条指令读或写一个内存地址（或许多内存地址），然后暂停就行，在任何地方都无所谓。有三种方法设置内存断点。

- 对于一条指令，右键点击该行，然后选择 Breakpoint->Memory, On Access or Memory, On Write。
- 要在内存数据区设置断点，在数据窗口中选中一个或多个字节，然后右键选择和上面一样的操作。
- 你也可以对整个内存区域设置断点。打开内存映射窗口（“Me”

图标或 View->Memery)，右键相关内存区域，在弹出菜单中选择“ Set Break On Access for either Access or Write”。

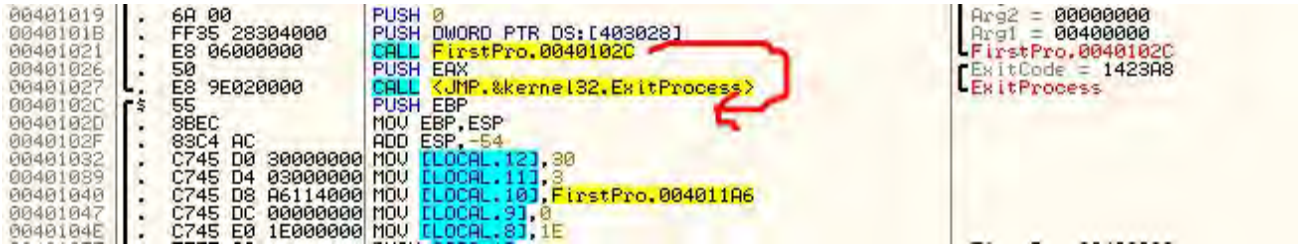


五、内存数据面板的使用

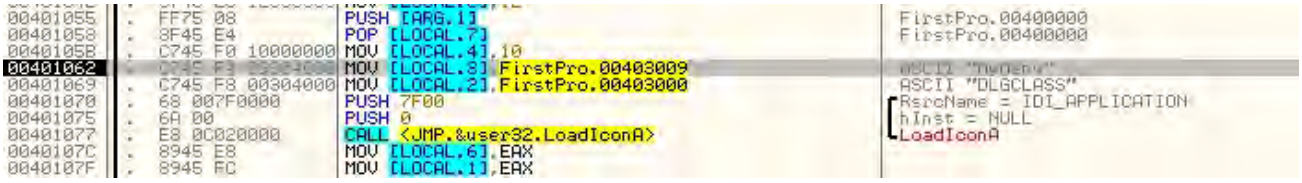
你可以用数据面板检查被调试进程内存空间中的内容。如果反汇编窗口的指令、寄存器或堆栈中的任何一项包含了对内存位置的引用，你可以在该引用上右键然后选择“Follow in Dump”，随即数据面板就会向你显示该地址引用的内容。你也可以在数据面板的任何地方右键单击选择“GoTo”，然后输入要查看的地址。咱们现在试试。

确保 FirstProgram 已经载入并且停在了入口处。现在，按下 F8 八次，来到了 401021 地址指令处，该处指令是 CALL FirstPro.40102c。如果你注意看这行的话，会注意到这个 CALL 会向下跳转到 40102c 处，在当前行下面的第三行的地方。按下 F7 我们单步步入那个跳转，然后我们就来到了 40102c。记住这

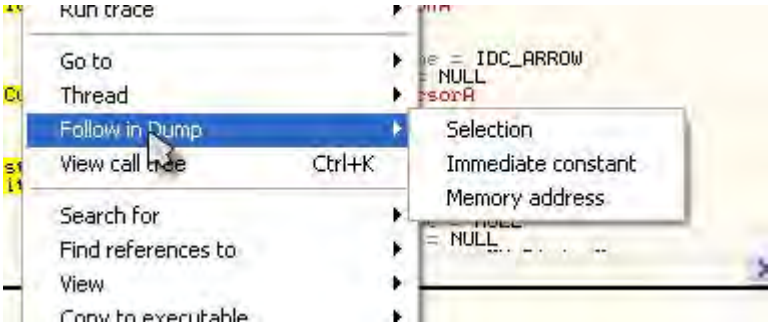
是一个 CALL 指令，所以我们最后还是会回到 401021 的（至少是该条指令后面的那条）。



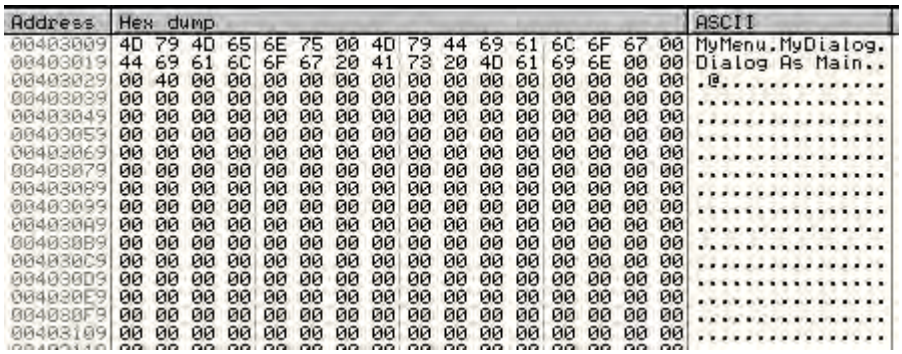
现在，单步执行代码（F8）直到 401062。你也可以在这行设置断点，然后按 F9 运行。还记得怎么设置断点吗？双击你想设置断点的那行的操作码列。你也可以选中该行，然后按 F2 去设置或取消断点。现在我们断在了 401062:



现在，我们看看断下的那行。相关指令是 MOV [LOCAL.3], FirstPro.00403009。我确定你知道（因为你已经学了汇编语言:P）这条指令是将地址 00403009 中的内容移动到堆栈中（这里 Olly 是用 LOCAL.3 表示的）。你可以在注释列看到 Olly 已经发现了该地址处的内容是字符串 “MyMenu”。好，下面让我们看看。在指令上右键，选择 “Follow in Dump(数据窗口跟随)”。注意这里有好几选项:



这里我们选择“Immediate constant”。这回载入指令中的任何地址。如果你选择了“Selection”，数据窗口会显示高亮行所在的地址，这里是 401062（也就是我们暂停的地方）。基本上我们在数据窗口中看到的就是我们在反汇编窗口中看到的。最后，如果我们选择了 Memory address”，数据窗口会显示 LOCAL.3 的内存区域。这会显示我们正在使用的变量（在堆栈中）的内存。下面是选择了 Immediate constant 之后的数据窗口的样子:



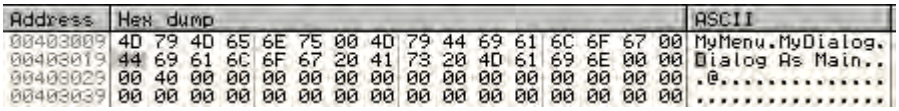
Address	Hex dump	ASCII
00403009	4D 79 4D 65 6E 75 00 4D 79 44 69 61 6C 6F 67 00	MyMenu.MyDialog.
00403019	44 69 61 6C 6F 67 20 41 73 20 4D 61 69 6E 00 00	Dialog As Main..
00403029	00 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.@..............
00403039	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403049	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403059	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403069	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403079	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403089	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403099	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004030A9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004030B9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004030C9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004030D9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004030E9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
004030F9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
00403109	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

就像你看到的，数据窗口显示的内存是从 403009 开始的。正是 011y 按指令从中载入字符串的地址。在右边你可以看到字符串“MyMenu”。左边你可以看到每个字符的十六进制数据。你可能注意到了在“MyMenu”后面有些其他的字符串。这些字符串会在程序的其他部分被用到。

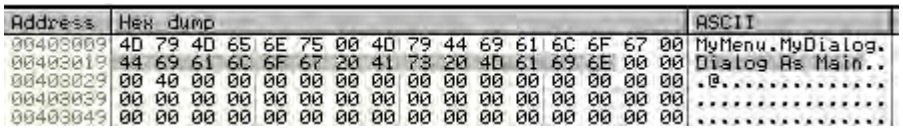
六、最后，来点有意思的！

此次教程的最后，让我们做些有意思的事情。让我们修改二进制数据来显示我们自己的信息！我们将字符串“Dialog As Main”改成我们自己的，然后看看效果。

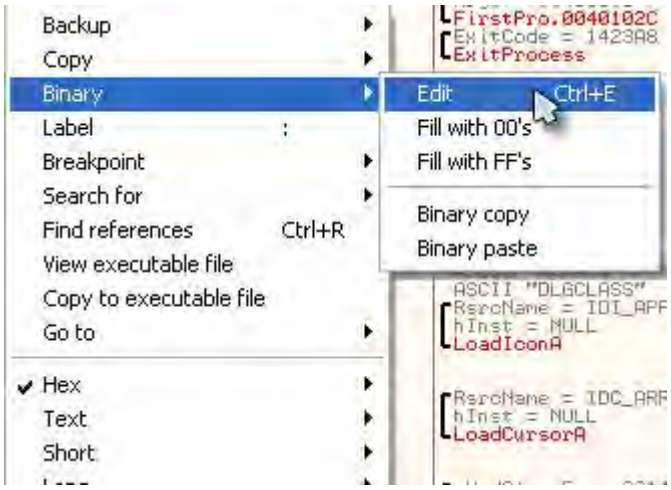
首先，在数据窗口的 ASCII 列，点那个“Dialog As Main”中的“D”:



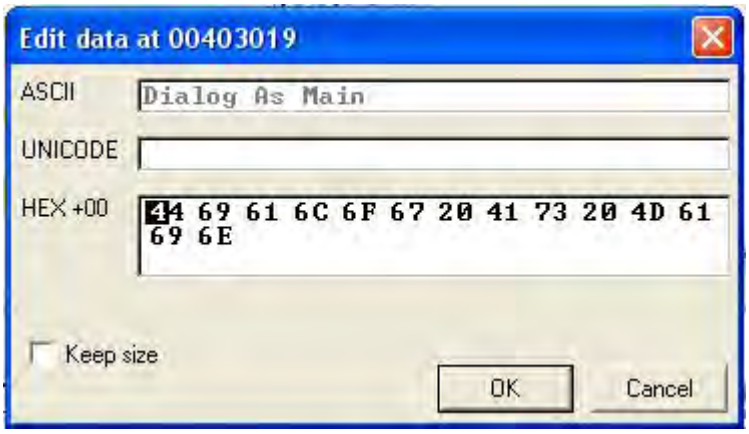
注意，左边的第一个十六进制数据也高亮了。这个数字对应字母“D”。如果你查查 ASCII 码表的话，就会发现字母“D”的十六进制数正是 0x44。现在，选中整个“Dialog As Main”字符串:



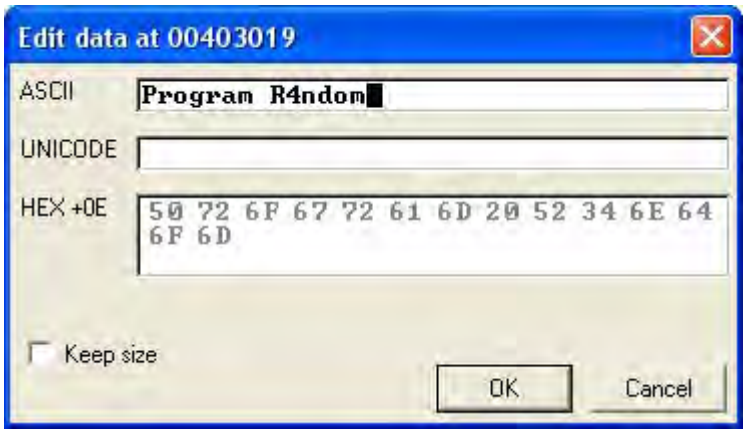
在选中的内容上右键，选择“Binary” -> “Edit”。我们就可以修改我们程序在内存的内容:



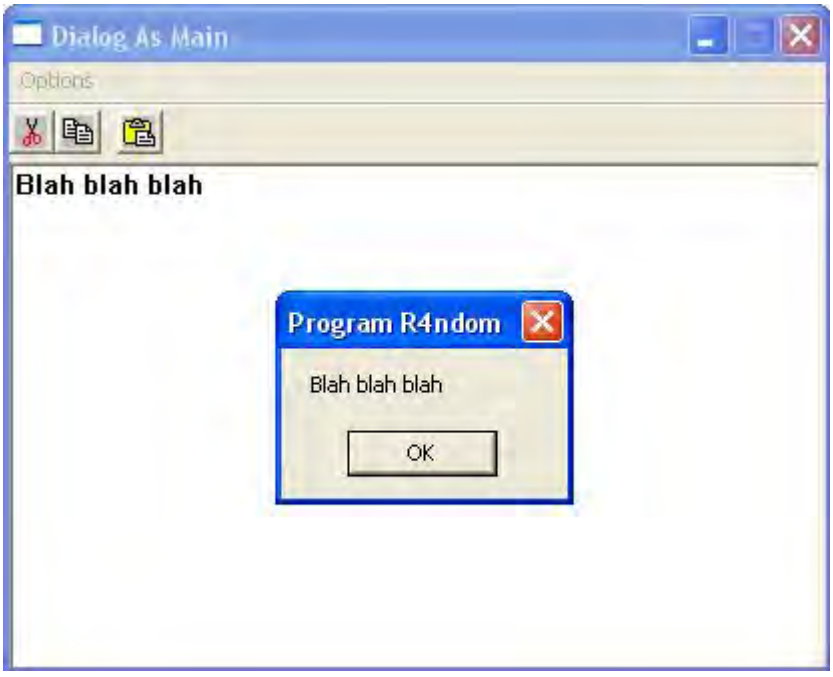
然后就会弹出一个如下的窗口:



第一个文本框以 ASCII 码的形式显示字符串。第二个文本框是以 UNICODE 形式（我们的程序用不着，所以空着），最后那个文本框是相关字符串的原始数据。好，咱们改一下。点一下字符串的第一个字母（“D”），然后输入任何你想要将“Dialog As Main”覆盖掉的内容。要注意的是你输入的长度，别多了。否则你就会覆盖掉程序需要的其他字符串，或者更糟糕是覆盖掉了程序需要的代码!!! 这里呢，我输入的是 “Program R4ndom”:



完了之后呢点 OK 按钮，并允许程序（点 Olly 的运行按钮或按下 F9）。切换到我们的程序，然后随便输入什么都行，然后选择菜单“Option”->"Get Text"。现在看看我们的对话框!



注意到对话框的标题有什么不同没有 。

（这一章真 TM 长啊，翻的我累死了!!!）

教程四：OllyDbg 的使用（下）

一、简介

此次教程我们继续学习 Olly 的使用。我们将继续使用上一章的程序（我也会将它包含在下载里）。

你可以在 tutorials 中下载文件和 PDF 版的教程。

二、DLLS

就像我前面说的，当你启动程序时，DLL 被系统载入器载入。这回我会细致的讲解。DLL（Dynamic Link Libraries）是函数的集合，通常由 Windows 提供（当任何人都可以提供），其中含有很多 Windows 程序要用的函数。这些函数可以让程序员更容易的完成一些乏味的重复性的任务。

例如，将字符串全部转换成大写是许多程序要实现的功能。如果你的程序要多次使用该功能的话，你有三个选择：一是在你的程序中自己编码实现；问题是，你不知道你的下一个程序是不是也会用到该功能很多次。你可能需要在你使用到的程序里复制粘贴很多次相同的代码。二是创建一个自己的库，这样任何程序都可以调用。这种情况下，你可以创建一个 DLL，然后包含在程序中。该 DLL 可能有像 convertToUpper 这样的通用函数以便于程序调用，因此你只需要写一次代码就行了。这样做的另一个好处是，你可以说你为字符串转大写想到了一个很好的优化方案。第一个例子中，你需要将代码拷贝到所有要用到该代码的程序中，但是在那个通用 DLL 例子中，你只需要修

改 DLL 的代码，然后所有使用该 DLL 的程序都可以以最快的速度获益。爽吧！这就是 DLL 产生的真正原因。

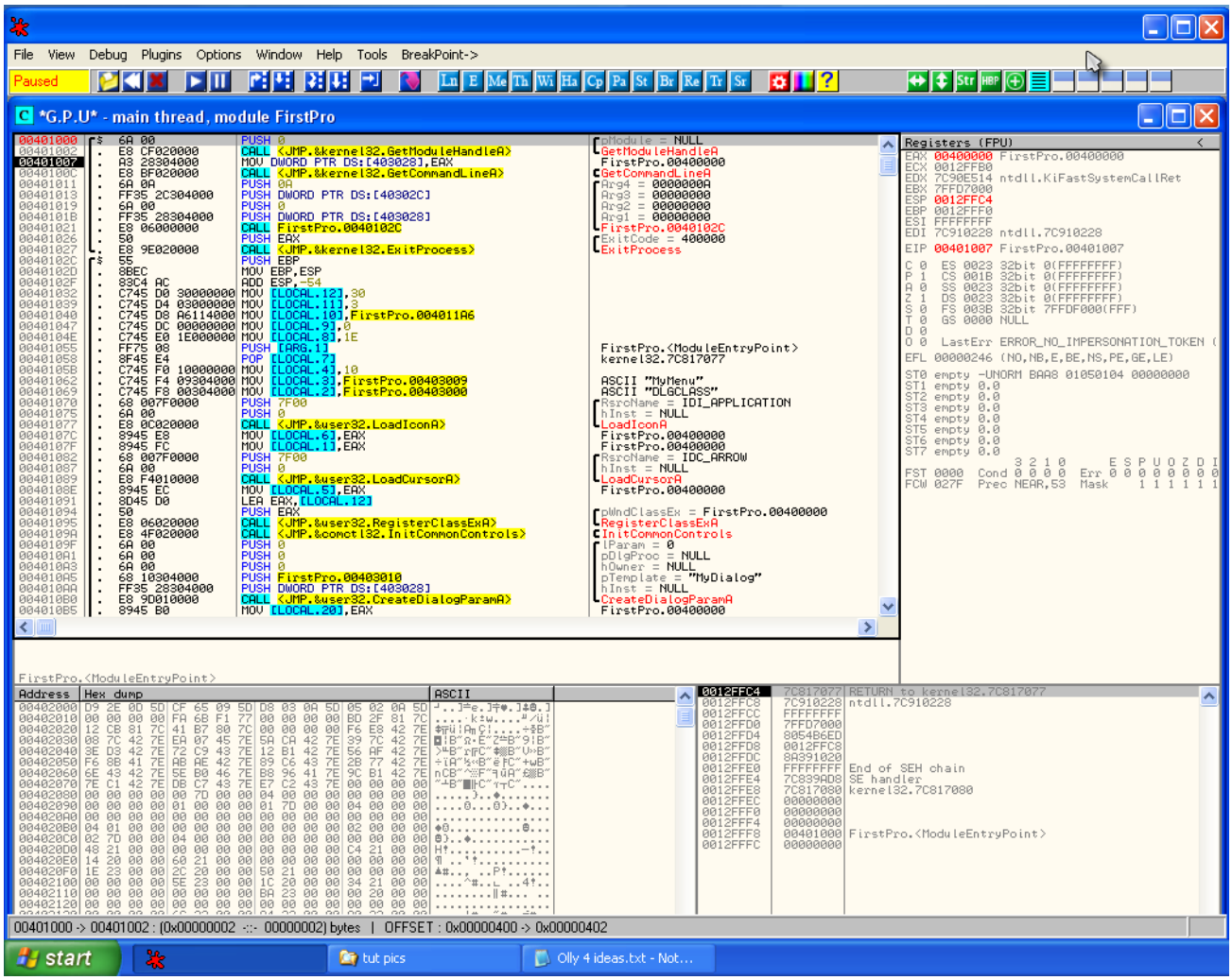
最后一个选择是，使用 Windows 提供的一堆 DLL 中包含的数千个函数中的一个。这样做有很多好处。第一个是，Microsoft 的程序员已经花了多年时间来优化他们的函数，他们在很大程度上要比你牛逼。第二，你不需要将你的 DLL 包含在应用中，因为 Windows 操作系统已经内建了这些 DLL。最后，如果 Windows 决定修改他们的操作系统，你自己的 DLL 有可能和新系统不兼容。同时，如果你使用 Windows 的 DLL，它们肯定是兼容的。

三、如何使用 DLL

现在你已经知道了什么是 DLL，那就谈谈如何使用它们。DLL 基本上就是一个你的程序可以调用的函数库。在你第一次载入应用程序时，Windows 载入器就会检查 PE 头（还记不记得 PE 头？）的特定区段，看看你的程序调用了哪些函数，以及这些函数都在哪些 DLL 中。在将你的程序载入内存后，载入器就迭代这些 DLL，将它们载入到你的应用程序的内存空间。然后它再仔细检查你的程序的代码，并将你的程序调用的 DLL 函数注入到正确的地址。例如，如果你的程序调用 Kernel32 DLL 中（只是一个例子啊）的 StrToUpper 函数来将一个缓冲区里的字母转换成大写，载入器要找到 Kernel32 DLL，找到 StrToUpper 函数的地址，并将地址注入到你的程序中调用该函数的那行代码处。你的程序就会通过调用进入到 Kernel32 DLL 的内存空间，执行 StrToUpper 函数，最后再返回到程序中。

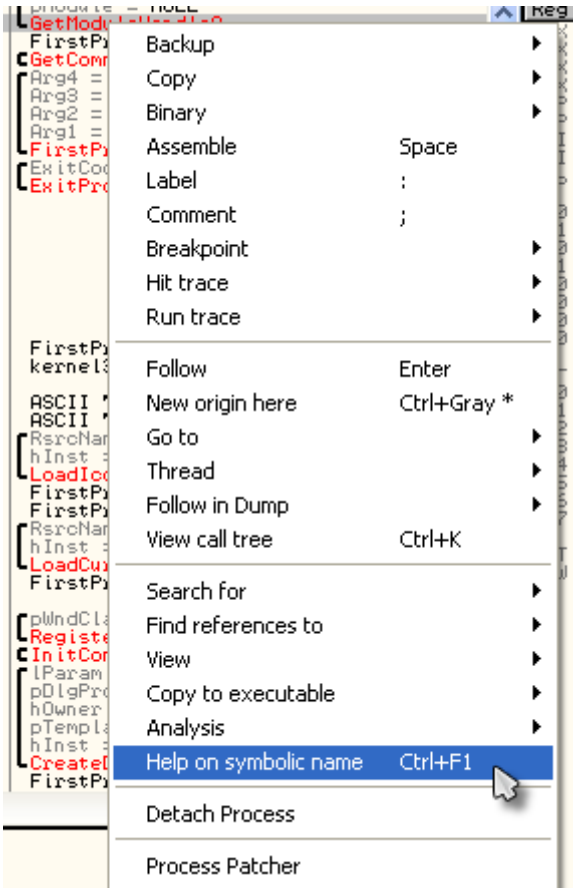
让我们实际看看这个过程。Olly 载入本教程包含的

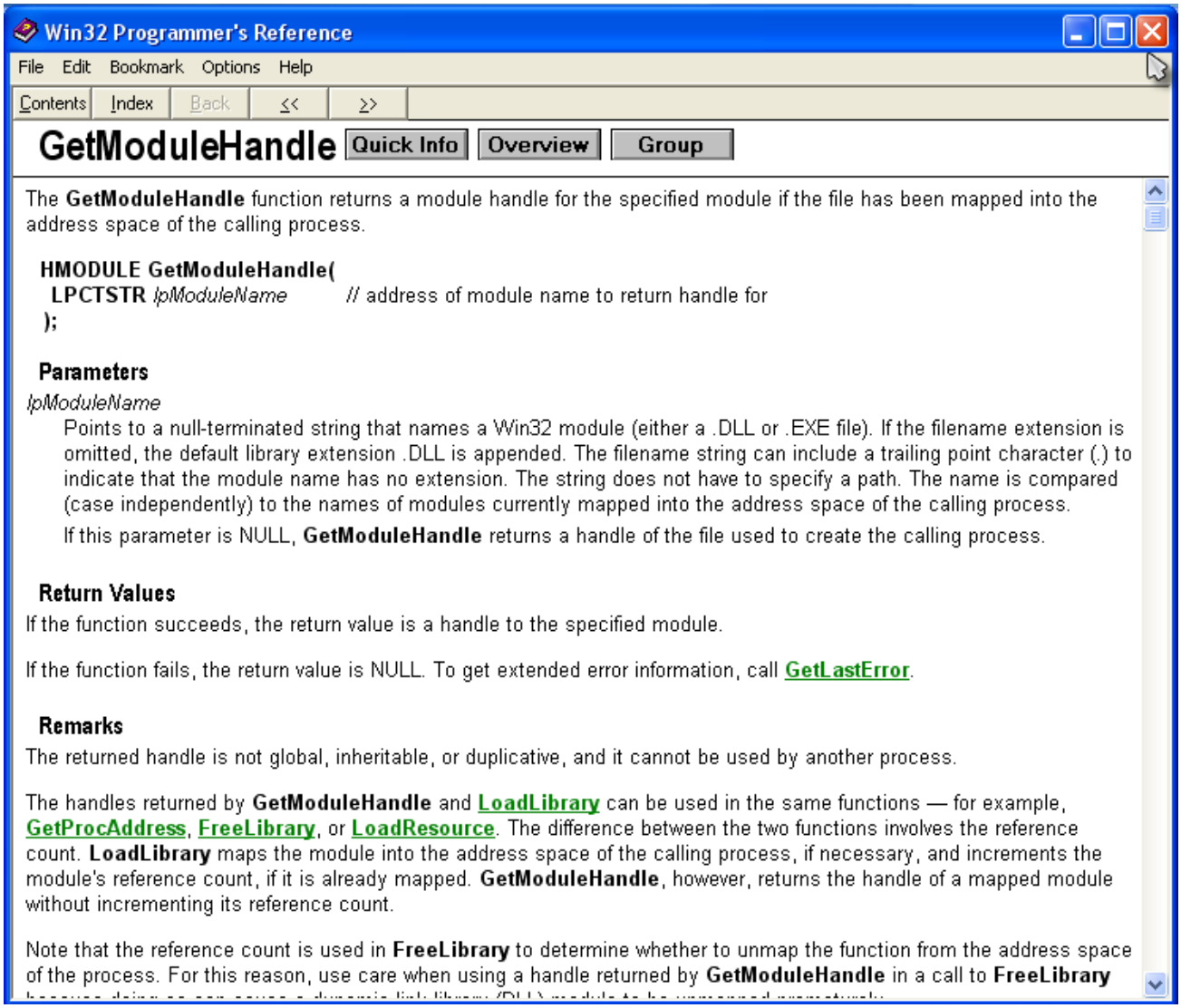
FirstProgram.exe。Olly 断在了第一行代码（从现在起我们就叫它入口点（Entry Point）——这很重要，因为这是我们详细讨论 PE 头的时候 PE 头中的叫法）。



如果你看第二行代码的话，你会看到一个对函数 kernel32.GetModuleHandleA 的调用。第一步，我们看看这个函数是干嘛的。我已经将 WIN32.HLP 文件以及一个教你怎样将它安装到你的 Olly 中的文本文档包含在了本课的下载里，就是为了防止你上一课没有拿到它。安装该文件后，你在你不熟悉的 Windows API 上右键，会显示一个该 API 是干什么的菜单。在你拷贝过去后，你需要重启下 Olly。现在，在 GetModuleHandleA 上右键，选择“Help on Symbolic

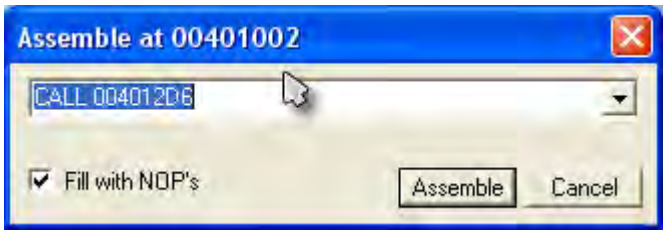
Name”。然后 Olly 会显示一个该函数的备忘单：





那么，基本上这个函数就是为了获取我们程序内存空间的句柄。在 Windows 中，如果你想对一个窗口（或者是相当一部分的其他对象）做任何事情，你必须取得它的句柄。这基本上是 Windows 知道你正在操作的对象的唯一标识符。GetModuleHandle 其实比这个稍微复杂点，不过当我们经历了更多知道了更多知识以后再回过头来讨论这个。

关闭帮助窗口，我们看看这个 CALL 去了哪里。Olly 已经试着帮助我们，它用函数名替换掉了 GetModuleHandleA 的真实地址。让我们看看它驻留的地址是什么。点一下调用 GetModuleHandleA 的那行代码，再按一下空格键，就会打开一个汇编窗口：



该窗口有两个目的：第一它向你显示了正在被计算的（以防 Olly 帮助性的替换了地址）真实的汇编语言指令，第二它允许我们编辑汇编语言指令。在下一课前我们不会做任何编辑，这次我们只是看看地址：4012D6。有两种方法可以跳转到该地址看看那儿有什么（而不用真的运行程序）。选中“Call GetModuleHandleA”然后按下“Enter”，你也可以按 Ctrl+G 手动输入地址。我们试试第一种方法，选中 401002 那行（第三列有相关指令）然后按回车键，你就会来到该 CALL 要调用的地方：

```
004012D6   $- FF25 24204000   JMP DWORD PTR DS:[<&kernel32.GetModuleHandleA>]   kernel32.GetModuleHandleA
004012DC   $- FF25 00204000   JMP DWORD PTR DS:[<&comctl32.ImageList_Add>]     comctl32.ImageList_Add
004012E2   $- FF25 0C204000   JMP DWORD PTR DS:[<&comctl32.ImageList_Create>]  comctl32.ImageList_Create
004012E8   $- FF25 08204000   JMP DWORD PTR DS:[<&comctl32.ImageList_Destroy>] comctl32.ImageList_Destroy
004012EE   $- FF25 04204000   JMP DWORD PTR DS:[<&comctl32.InitCommonControls> comctl32.InitCommonControls
004012F4      00              DB 00
004012F5      00              DB 00
004012F6      00              DB 00
004012F7      00              DB 00
004012F8      00              DB 00
004012F9      00              DB 00
004012FA      00              DB 00
004012FB      00              DB 00
004012FC      00              DB 00
```

现在这里比较有趣：它看起来确实不像执行 GetModuleHandleA 的代码。更像是一些跳转。对此有一个很好的原因说明，不过不幸的是，需要解释一下。

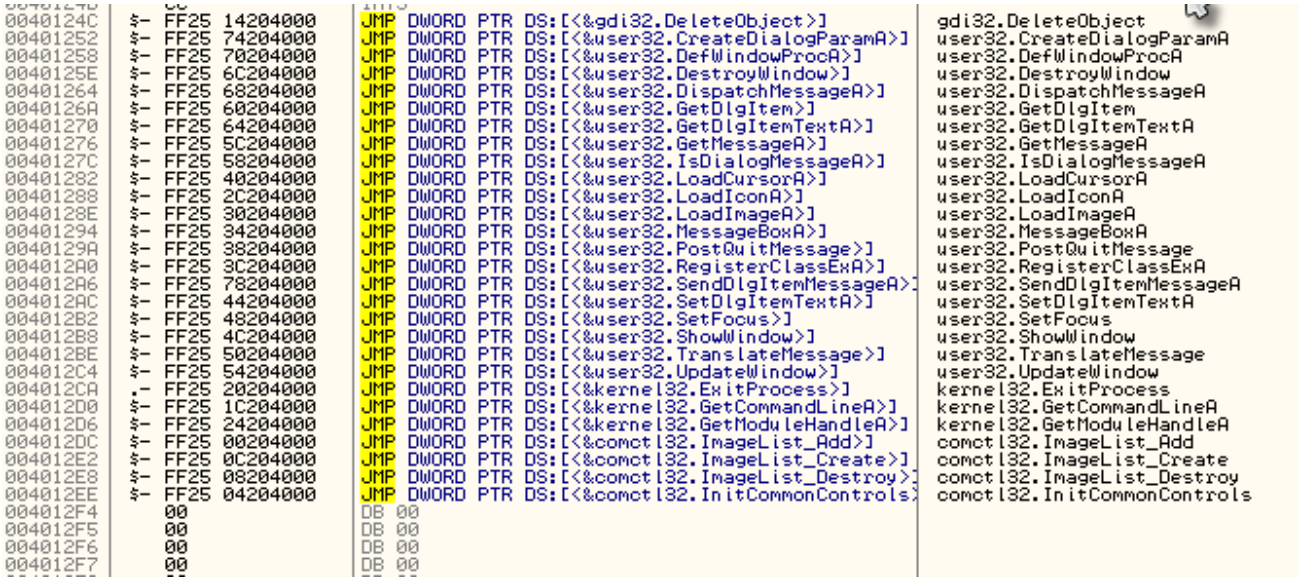
四、地址跳转表

有件事你需要知道，DLL 并不是总是一次性全部载入到内存。Windows 载入器负责载入你的程序和所有需要的 DLL，它可以修改被载入的 DLL 在内存中的位置（坦白的说，它甚至能够修改你的程序被载入的位置，这些我们后面再说）。原因是这样的，现在有一个 Windows DLL 属于最先载入的那种，被映射到地址 80000000。好吧，恰好你自

己的程序也带有一个 DLL 且需要载入到地址 80000000。两个 DLL 当然不能被载入到同一个地址，载入器必须将其中一个移到另一个地址。这种情况时常发生，还被叫做重定位。

这里有个问题：在你首次编写一个程序并写了一个调用 GetModuleHandleA 的指令，编译器会准确的知道正确的 DLL 在哪，然后它会放一个地址在指令里，有些类似于“Call 80000000”。现在，当你的程序被载入内存时，它仍然会让这个 CALL 调用 80000000（我这里说的有点过于简单了）。不过，如果载入器将这个 DLL 移到 80000E300 会怎么样？你的 CALL 会调用错误的函数！

PE 文件和此后的 Windows 文件围绕这个问题提出的解决方法是建立一个跳转表。意思是你的代码在首次编译时，每一个对 GetModuleHandleA 的调用都指向你的程序的一个地点，然后这个地点就会立即跳转到一个随意的地址（这是最后的正确的地址）。事实上，所有对 DLL 函数的调用都采用了同样的技术。它们每一个调用特定的地址，然后立即跳转到一个随意的地址。当载入器载入所有的 DLL 时，它会遍历“跳转表”，然后在内存中用真实的函数地址替换掉所有的随意地址。下面是所有真实地址被填充后的跳转表的样子：



这个有点复杂，下面我举个例子。我会写一个短程序，使用完全随意的信息（只是为了证明我们的观点）来调用一个 Kernel32 DLL 中的函数 ShowMarioBrosPicture。下面是我的程序(没有特指哪种语言)：

```
main()
{
      call ShowMarioBrosPicture();
      call ShowDoYouLikeDialog()
      exit();
}
ShowDoYouLikeDialog()
{
      If ( user clicks yes )
      {
          call ShowMarioBrosPicture();
          Call ShowMessage( "Yes, it's our favorite too!")
      }
      else
      {
          call showMessage( "You obviously never played Super Mario Bros.");
      }
}
```

这些代码被编译之后，对函数的调用将会被真实的地址替换，就像下面这样（再次声明，这里没有特指某种语言）：

```
401000    call 402000     // Call ChowMarioBrosPicture
401002    call 401006     // Call showDoYouLikeDialog
401004    call ExitProcess
401006    Code for "Do You like It" dialog
.
.
.
40109A    if (user clicks yes)
40109C  call 402000    // call showMarioBrosPicture
40109E    call 4010FE    // call show message
4010a1  call ExitProcess
4010a3  if (user clicks no)
4010a5  call 4010FE    // call show message
4010a7  call ExitProcess

4010FE  code for show message
...
40110A  retn
```

这些代码的后面就有可能是我们的跳转表（本例中，跳转表中只有 ShowMarioBrosPicture）。

```
402000  JMP XXXXXXXX
```

我们的程序（译者注：这里作者应该是将 our program 写成了 out program，所以我给翻译成我们的，小伙伴们可以自己查阅）并不知道 ShowMarioBrosPicture 在哪（或者说不知道 Kerner32 DLL 在哪），我们程序的编译器只是用实际的调用地址填充 X（并不是正真的地址，你知道那么意思就行）。

当 Windows 载入器载入我们的程序时，它首先将二进制文件载入内存，完成跳转表的构建，不过跳转表里没有任何真实的地址。然后开始载入 DLL 到我们的内存空间，最后开始找出所有函数驻留的地方。一旦它找到了 showMarioBrosPicture 的地址，它就准备进入跳转表并

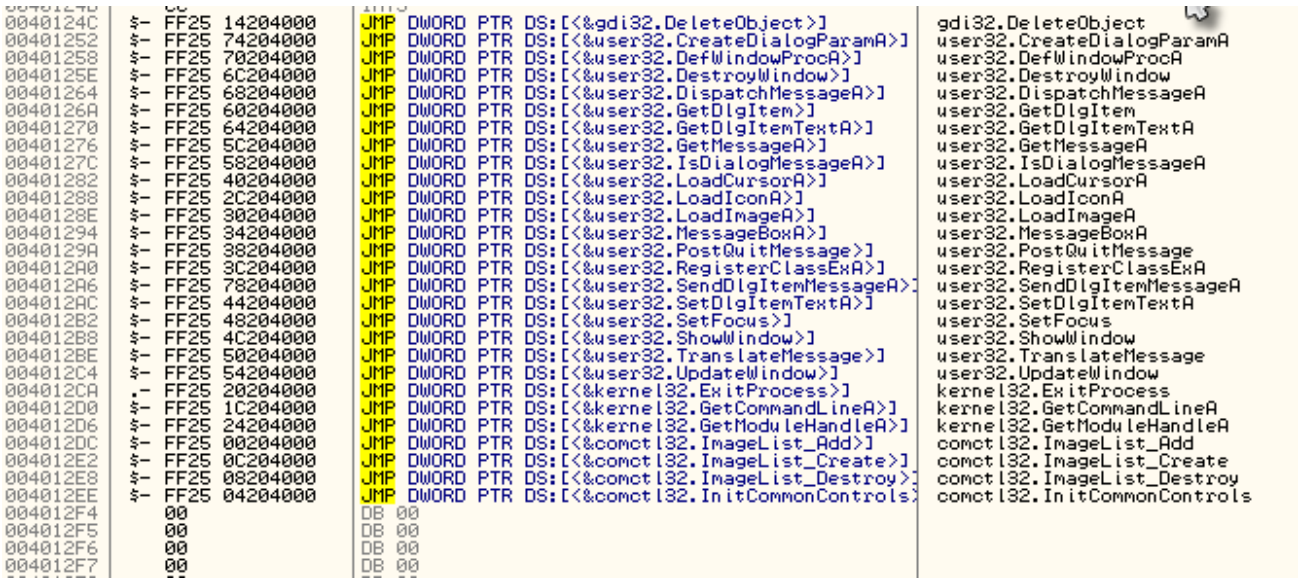
用函数的真实地址替换掉 X。假定 showMarioBrosPicture 的地址是 77CE550A。我们的跳转表代码就会被替换成如下：

```
402000 JMP 77CE550A
```

因为 Olly 能够发现该地址指向的是 showMarioBrosPicture，所以它会帮助性的进入跳转表并将跳转表显示如下：

```
402000 JMP DWORD PTR DS:[<&kernel32.showMarioBrosPicture>]
```

现在，让我们回到 FirstProgram 看看跳转表：



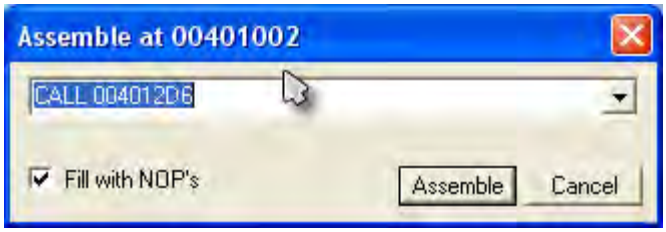
在首次编写这个程序时，各种 DLL 中的函数被调用，但是编译器不知道我们的程序在运行的时候这些函数是在内存的什么地方，所以它要创建一些像下面这样的东西（不是很准确这里）：

```
40124C  JMP XXXXX    // gdi32.DeleteObject
401252  JMP XXXXX    // user32.CreateDialogParamA
401258  JMP XXXXX    // user32.DefWindowProcA
40125E  JMP XXXXX    // user32.DestroyWindow
...
```

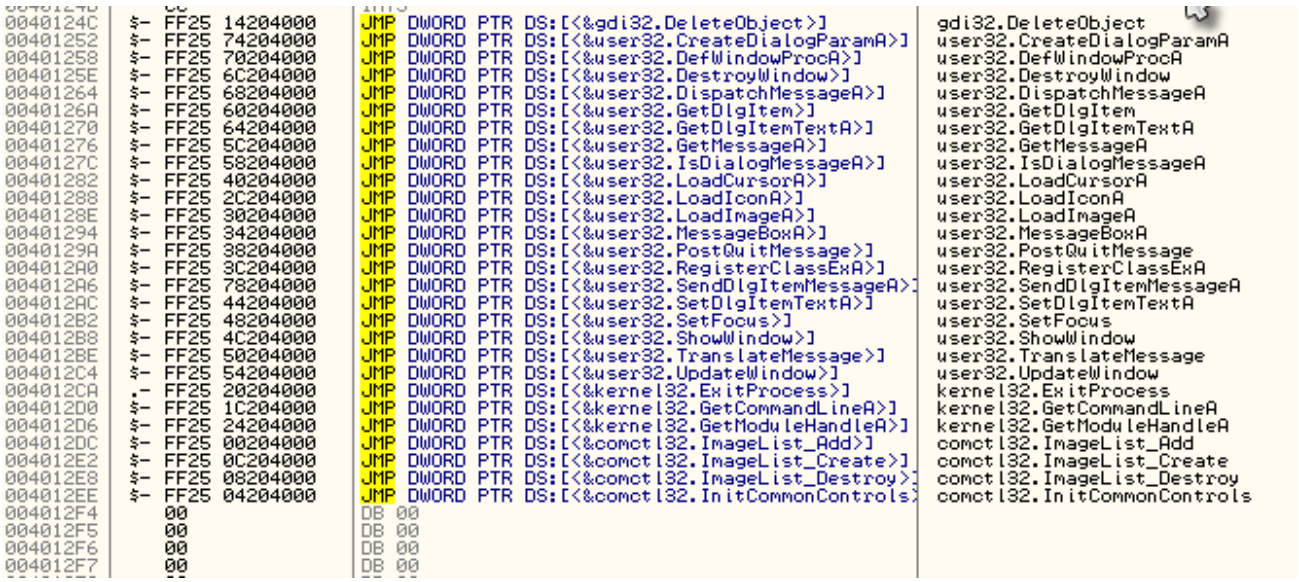
在载入器将我们的程序载入之后，再载入所有的 DLL 并查找所有的函数的地址。然后它会遍历每一个函数，用这些函数当前驻留的真实地址替换，就行前面图片中的那样。如果仔细想想的话，这确实是

相当巧妙的处理方法。如果不这样做的话，那么载入器就得遍历整个程序，并对每个 DLL 中的每个函数的调用都用真实的地址进行替换。那个工作量就大了。使用这种方法，载入器对于每个函数的调用只需要替换一个地方，就是跳转表那样的。

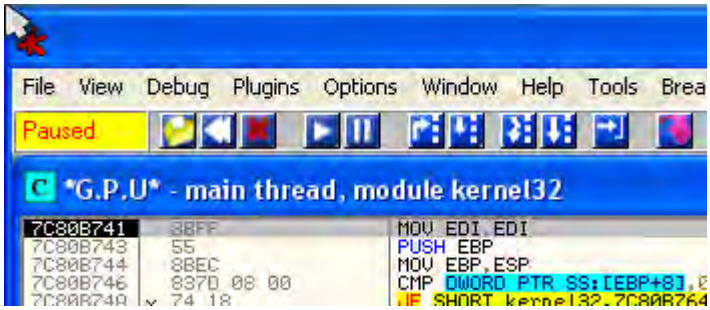
还是看看我们自己的程序吧。重载应用并按下F7。点击选中401002那行指令（和前面做的一样），再按下空格（和前面做的一样）：



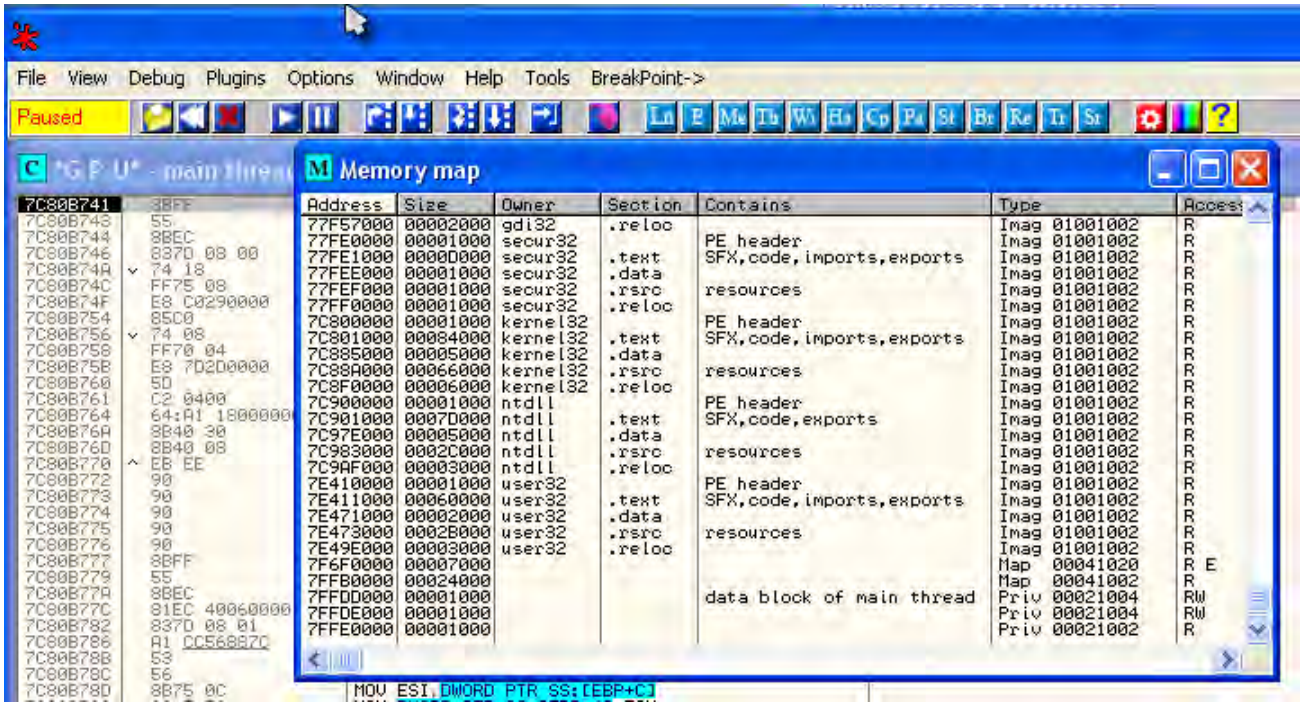
再一次提醒你注意那个地址，4012D6。现在按 F7 步入那个 CALL，注意我们来到了 4012D6。如果你向上翻，你会注意到我们来到了跳转表的中间：



现在，再点一下 F7，我们就会来到 GetModuleHandleA 的真正的地址 7780B741。有两种方法可以知道我们现在正在模块 kernel32 中，两者在不同的场合你都可能用到。第一个是 Olly 的 CPU 窗口标题：

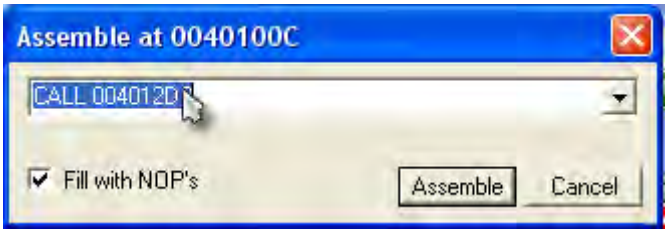


你能看到它显示的是“module kernel32”。第二种方法是到内存映射窗口查看地址：



你会发现我们所在的地址（7780B741）是在 kernel32 的代码段地址空间中。

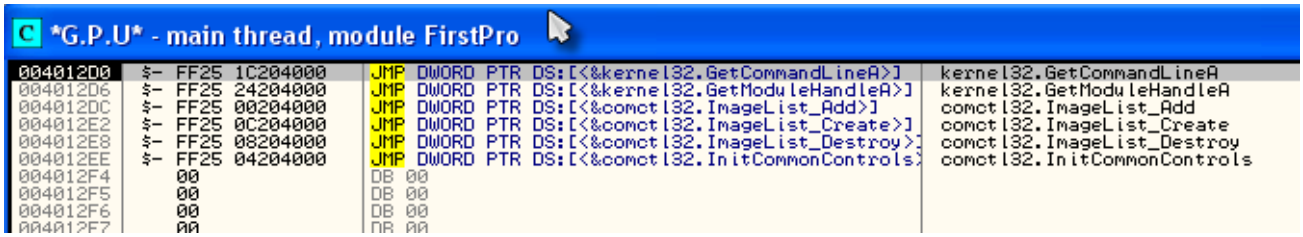
现在我们回头看看其他的函数调用。重启应用，按 F8 直到 40100C 处。那行代码是对 GetCommandLineA 的调用。点击选中指令再按下空格键，你就能够看见它指向的地址，是 4012D0：



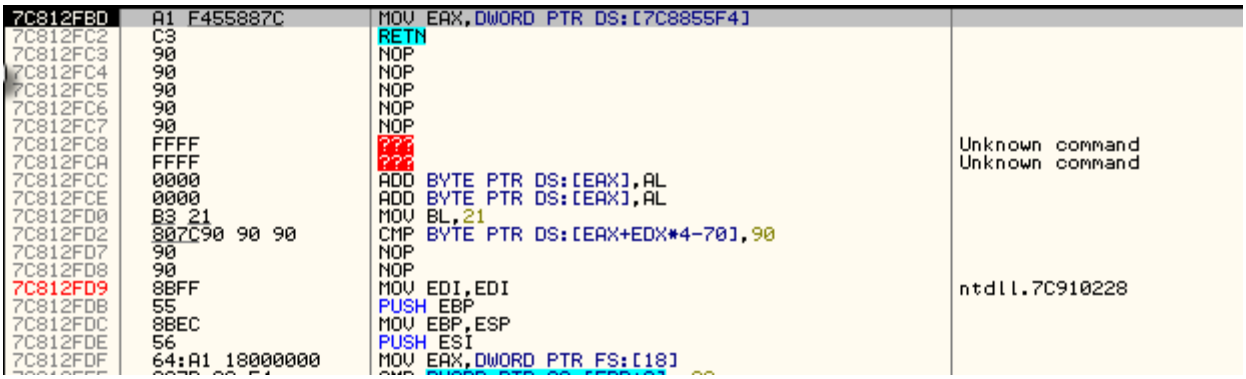
（不好意思，鼠标把地址挡住了，它是 4012D0）现在我们来试试

手动定位该地址，你会经常用到这种方法的。按 Ctrl+G 或点那个**转到**图标，输入我们想要转到的地址：

你的“GOTO（转到）”窗口可能不太一样，这个问题待会解决。现在点击 OK，我们就会跳到跳转表中 GetCommandLineA 的位置：



按下 F7 我们就来到了 kernel32 中的 GetCommandLineA 的开始处。这个函数从 7C812FBD 开始：



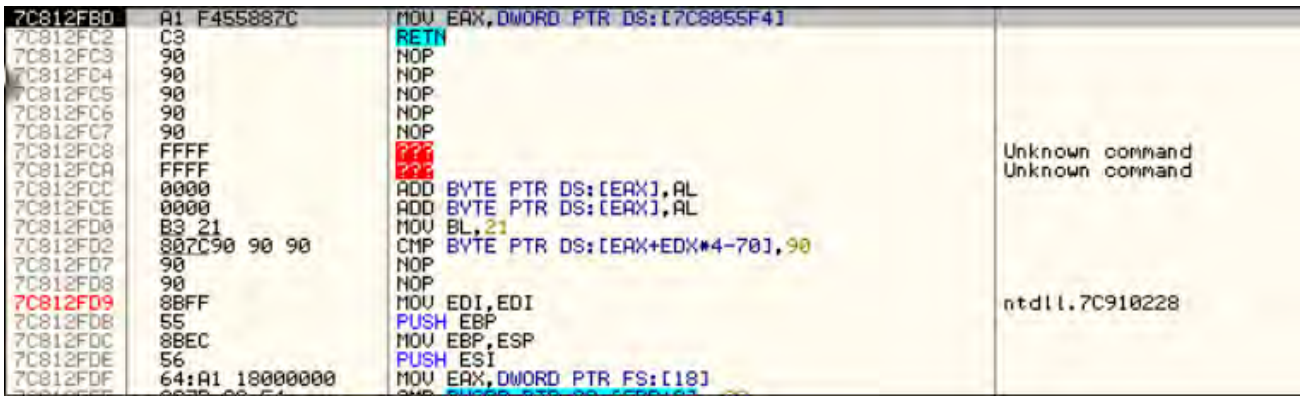
五、跳入及跳出 DLL

当我们围着一个程序转的时候，你不知道什么时候就在 DLL 中结束了。如果你正在尝试攻克一些保护方案时，通常你是不愿意在 DLL 中转的，因为 Windows DLL 中真的没什么东西。关于这方面的一个告诫，如果你正试着逆向的程序本身就带有 DLL 并且你就是想将它们也进行逆向工程（或者是保护机制确实在 DLL 中）。这里有几种从 DLL 回

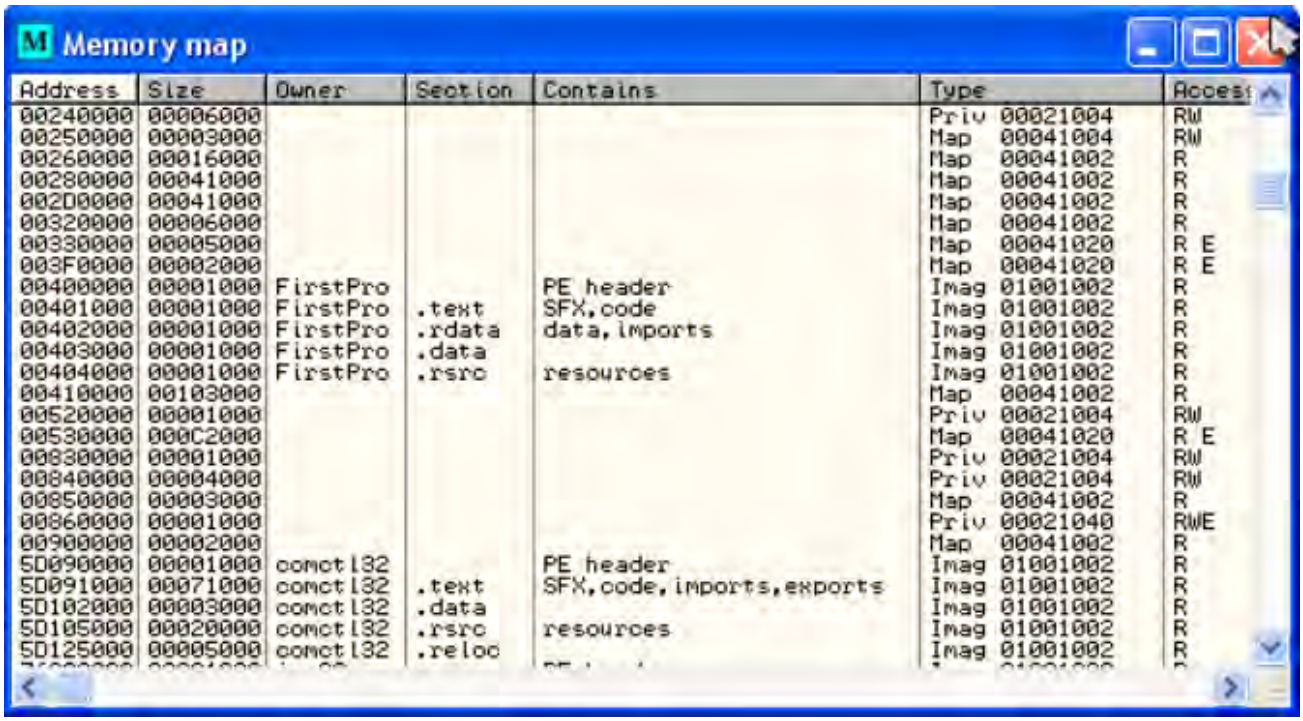
到我们的程序的方法。一个方法是单步通过所有的 DLL 函数代码直到最后你返回到程序，当然这可能得一会时间（有些情况下像 VB 程序，就是永远）。第二个选择是，点开“Debug”菜单并选择“Execute till user code（执行直到用户代码）”或者按 Alt+F9。意思是执行 DLL 中的代码直到我们返回到我们自己的程序代码。要注意的是，有时候这不一定好使，因为如果 DLL 访问了一个在我们的程序空间中的 buffer 或者变量的话，Olly 就会停在那儿，所以你最终可能会按 Alt+F9 好几次才能回来。

我们来试试这个方法。我们当前应该暂停在 7C812FBD，也就是 GetCommandLineA 的开始处。好，按下 Alt+F9。我们会回到程序中对 kernel32 调用的指令的后面那条指令（往上一行就是那个 CALL）。

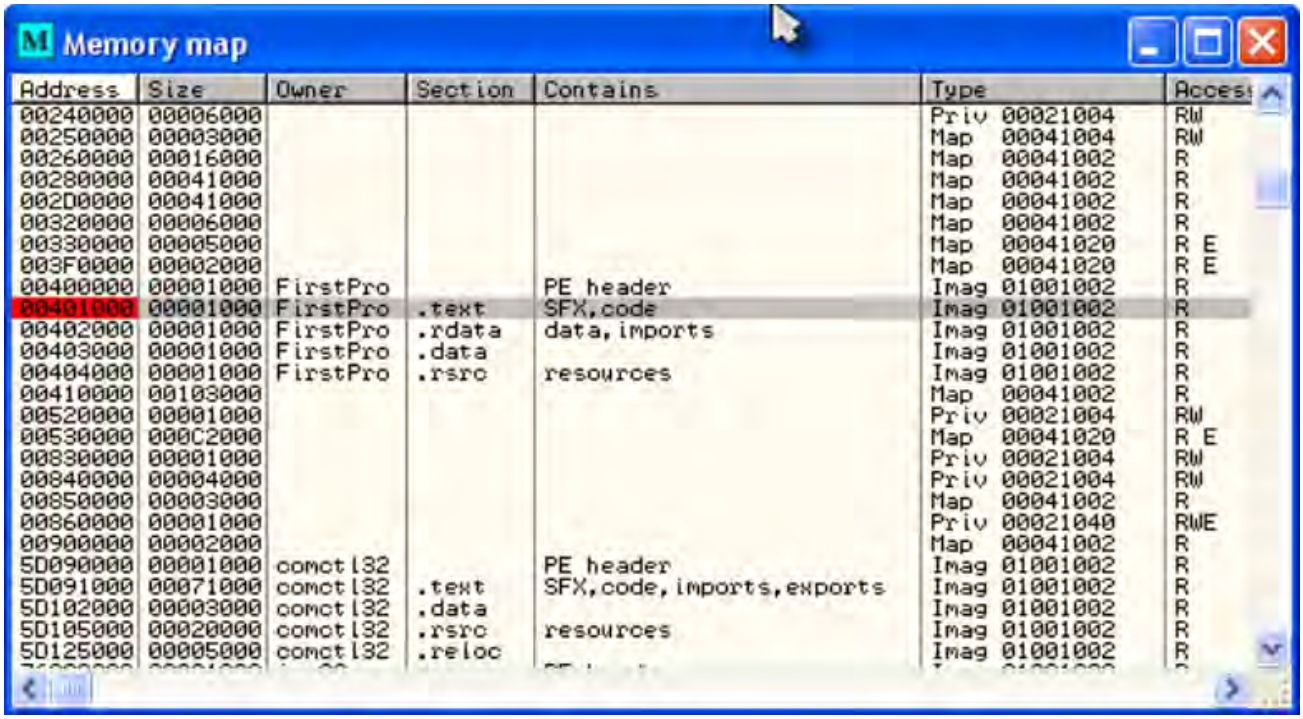
现在我们来试试另外一个回到我们的代码的方法。重启程序，单步步过（F8）直到对 GetCommandLineA 调用的那个 CALL（40100C）。单步步入（F7）那个 CALL，并且单步步入那个 jmp 进入跳转表。现在，我们回到了 GetCommandLineA 的开始处：



现在打开内存映射窗口，滚动到我们的程序的代码段那块（起始地址是 400000，写着 PE Header）：



现在，点击选中 401000 那行，我们的.text 区段在那行。按下 F2 设一个内存访问断点（或右键选择 Breakpoint on access）：

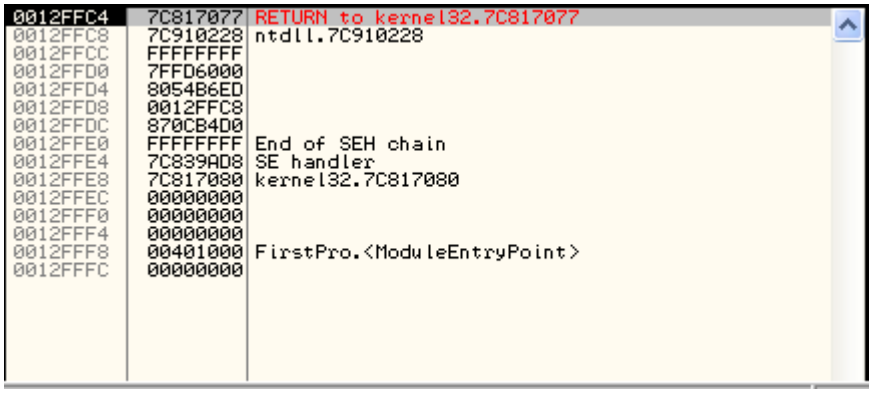


现在，运行程序。Olly 会断在和上面相同的那行，就是 401011 处，也就是我们对 DLL 调用 CALL 之后的那行!!! 好，现在删除内存断点，否则你会纳闷，为什么每次你运行程序的时候它都会断在下一行

六、再议堆栈

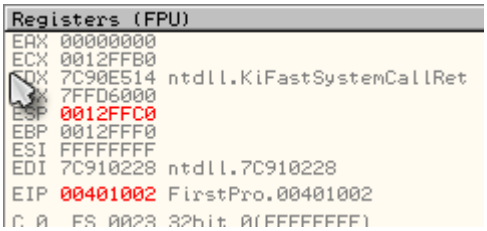
堆栈是逆向工程中的非常重要的一部分，如果对它理解的不够深入的话，你永远也不会成为一个伟大的逆向工程师。下面我们针对它做几个实验：

首先，看看寄存器窗口（在重启应用之后），看那个 ESP 寄存器。该寄存器中的地址指向栈顶。本例中，ESP 的值是 12FFC4。现在看看下面的堆栈窗口，列表中的顶部地址和 ESP 中的地址是一样的。



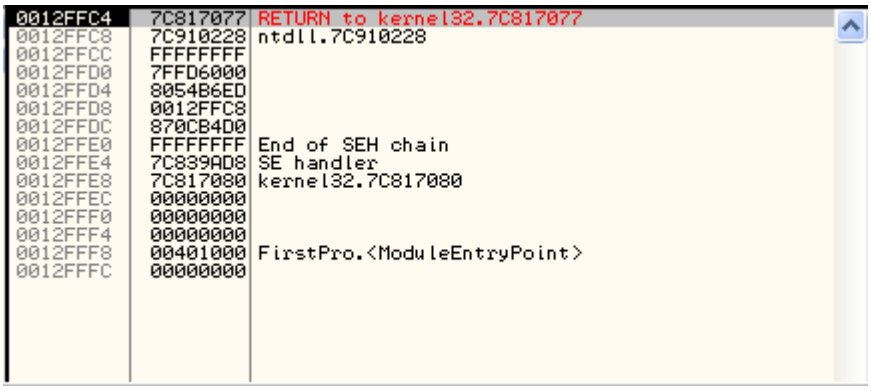
现在按 F8（或者 F7）一次，将 0 压入堆栈，再看看堆栈窗口：

就像我们上次课提到的那样，该操作将 0（null）压入堆栈。现在看看 ESP 寄存器：



已经变成了 12FFC0。因为，在向堆栈中压入一个字节后，该字节就变成了新的栈顶。按 F8 一次，单步步过对 GetModuleHandleA 的调

用，再看看堆栈窗口：



注意我们的堆栈已经向下回退了一位(ESP 寄存器也回到了原来的值)。这是因为 GetModuleHandleA 函数使用了这个被压入堆栈的 0，并把它作为参数。然后把它“POP（弹）”出了堆栈，因为这个 0 已经没用了。就行上一课提到的，这是向函数传递参数的一种方法：将参数压栈，被调用的函数将它们弹出栈，使用它们，然后返回，通常我们需要的信息都在寄存器里（后面会看到）。

接着继续....。如果你按 F8 两次单步步过对 GetCommandLineA 的调用，会发现堆栈并没有改变。因为，我们没有向堆栈中压入任何信息以供函数使用。接下来，是一个 PUSH 0A 的指令。这是准备传递给下一个被调用函数的第一个参数。单步步过，然后你会发现 0A 出现在了栈顶，ESP 寄存器下移了 4（当你向堆栈压入一个值时，ESP 寄存器会向下移，因为堆栈在内存中是向下“增长”的。译者注：堆栈是从高址向低址增长。）现在再按一次 F8，ESP 寄存器会再次下移 4。因为我们向堆栈中压入了一个 4 字节的值。如果你看堆栈的顶部，就会发现我们向堆栈中压入了 00000000。为什么呢？

我们看看做这个压入操作的那行代码，在 401013 处：

```
PUSH DWORD PTR DS:[40302c]
```

这行代码的意思（我保证你知道什么意思，因为你已经学了汇编

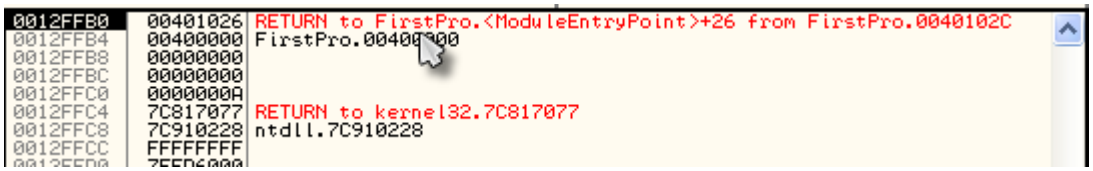
语言:p）是取地址 40302C 开始的 4 字节内容，然后将它们压入堆栈。那么在 40302C 的是什么呢？好吧，当然是 00000000！（开个玩笑）我们来自己看看。右键 401013 处的指令，选择“Follow in Dump（数据窗口跟随）”->“Memory Address（内存地址）”。然后会在内存数据窗口中显示以 40302C 开始的内存中的内容：

显然，哪里可没有那么多内容！不过你至少知道 0 是从哪儿来的。如果你想知道更多的细节比如这块内存是干什么的，这块内存空间被用来存储变量，并且最终会被这些变量填充。不过对于目前来说，所有的变量都被初始化为 0。

现在按一下 F8，我们遇到了另一个 PUSH 指令，不过这次是从 403028 开始。如果你在数据窗口中向上翻，会看到该地址处也是 0（在我们上一次课修改的字符串的后面）。这一块正在做的是将内存指针压栈，当前被设置为 0，我们的代码将会以变量的形式使用。单步步过上一个 PUSH 然后单步步入对地址 40101C 的调用。你应该注意的第一件事是有什么东西被压入堆栈里了：我们的 CALL 的返回地址，401026。

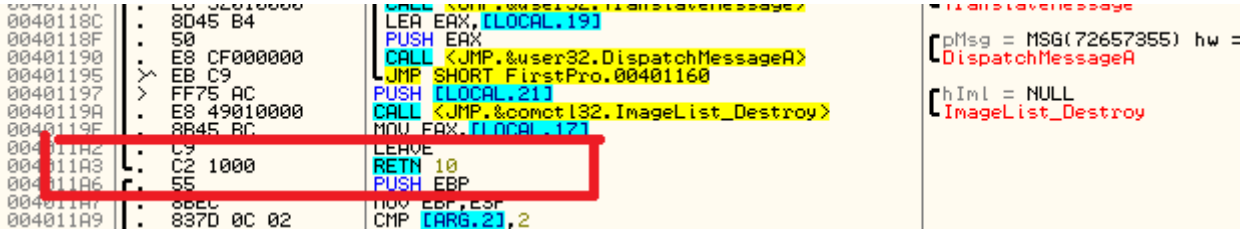
任何代码在使用 CALL 指令时，在我们还没有执行这个调用前，下一条将要被执行的指令（译者注：非 CALL 内部的指令）的地址会被自动的压入堆栈。原因是，我们调用的函数执行完后，它需要知道返回

到什么地方。被自动压入堆栈的地址就是返回地址。看那个堆栈窗口的顶部：



可以看到 Olly 已经指出了它是一个返回地址，并且它指回到我们的程序（FirstPro），需要被返回的地址是 40102C（CALL 的下一条指令）。

现在，在函数的结尾，一个 RETN 指令将会被执行（你肯定知道它是“return”的意思，因为它出现在你的汇编语言书的开头处）。这个返回指令真正的意思是“弹出栈顶的地址，将正在运行的代码指向这个地址”（它主要是用弹出的值替换 EIP 寄存器——存储当前正在运行的行的地址）。那么现在，被调用的函数在执行完后准确的知道了要返回到哪！事实上，如果你向下滚动一点，就会发现 4011A3 处的 RETN 语句会从堆栈中弹出这个地址，然后从该地址开始运行：



（RETN 语句后面的那个 10，意思是给我返回地址，然后再从堆栈中删除 10h 字节的空间，因为我再也不需要它们了。看看你汇编语言书籍的下一页吧）

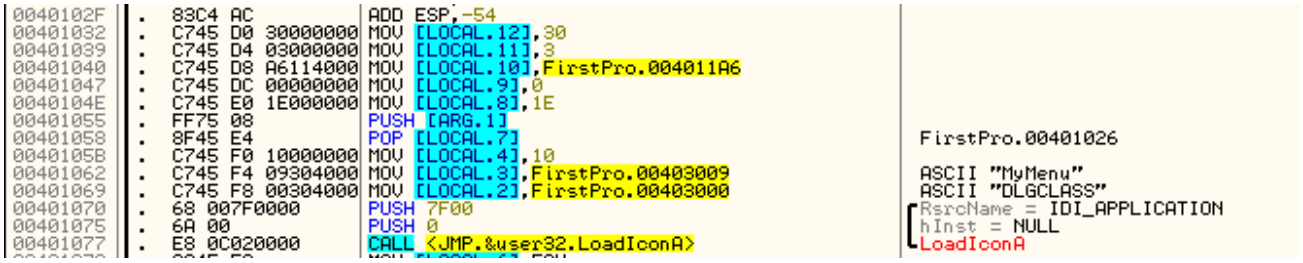
这里我们花点时间来启动一句，我保证在逆向工程社区会火的口头禅。我喜欢叫它“Random’s Essential Truths About Reversing Data（Random 关于逆

向数据的必备真言——译者注：大体这个意思吧，就这么翻吧，反正咱们也不会喊）”，或者 R.E.T.A.R.D（首字母缩写的听起来还不错）。我正式开启下面这个即将成为传奇的戒律：

#1. You MUST learn assembly language（#1、你必须学习汇编语言）.

如果你还没有的话，在逆向工程领域你不会取得成功。就是那么简单。

本次教程我准备最后谈论的是，Olly 怎么处理参数和本地变量的显示。如果你双击 EIP 寄存器，我们就能跳回到代码的当前行（在 40101C 处），往下可以看到好几行蓝色标记的行，显示的有 LOCAL 字样（其中一个显示 ARG）：



如果你没有任何编程经验，你可能不太知道本地变量和参数之间有什么不同。对于参数，就像我们早些时候讨论的，是传递给函数的变量，通常通过堆栈传递。本地变量是被调用函数“创建”的用来临时性存储数据的一种变量。下面是一个例子程序，其中有两个不同的概念：

```
main()
{
    sayHello( "R4ndom");
}

sayHello( String name)
{
```

```
        int numTimes = 3;
        String hello = "Hello, ";

        for( int x = 0; x < numTimes; x++)
            print( hello + name );
    }
```

程序中，字符串“R4ndom”是传递给 sayHello 函数的参数。在汇编语言中，这个字符串（至少是这个字符串的地址）会被压入堆栈，以便于 sayHello 函数引用。一旦控制权转给了 sayHello 函数，sayHello 需要设置一对本地变量（LOCAL VARIABLES），这对变量函数会使用，不过一旦函数执行完毕就不再需要它们了。例子中的本地变量是整形数据 numTimes、字符串 hello、整形 x。不幸的是，为了防止堆栈不够负责，参数和本地变量都存储在堆栈中。堆栈通过 ESP 寄存器来实现这个，不过寄存器可没有超能力。它通常指向栈顶，不过它是可以被修改的。所以，可以说我们进入了 sayHello 函数，并且堆栈中有下面的数据：

1、字符串“R4ndom”的地址

2、让我们进入函数的那个 CALL 的返回地址。

如果我们想要创建一个本地变量，我所需要做的的是从 ESP 寄存器中减去一定的值，这样就会在堆栈中创建一定的空间！假如我们将 ESP 减去 4（会有 4 个字节大小，或者一个 32 位的数）。堆栈会像下面这样：

1、空的 32 位数

2、字符串“R4ndom”的地址

3、让我们进入函数的那个 CALL 的返回地址。

现在，我们可以在这个地址里放任何数据，比如，我们可以让它

存储 sayHello 函数中的变量 numTimes。因为我们的函数使用了三个变量(所有的都是 32 位长),需要从 ESP 减去 12 字节(或十六进制的 0xC),然后我们就有了三个可以使用的变量。堆栈就会像下面这样:

1、指向字符串“hello”的空的 32 位地址。

2、变量“x”的空的 32 位数

3、变量“numTimes”的空的 32 位数

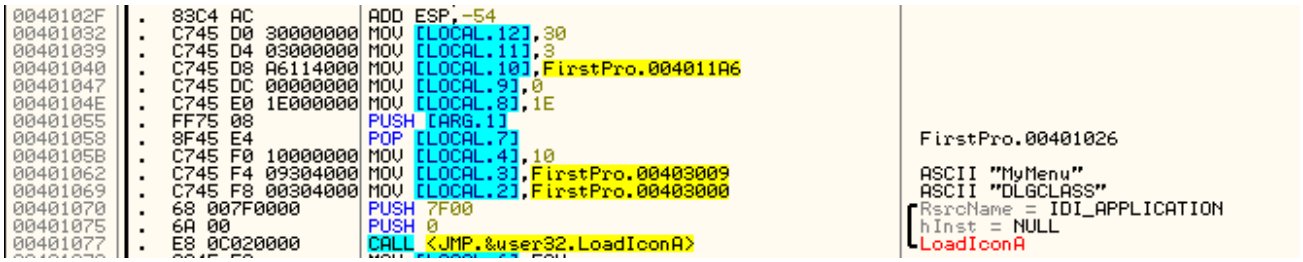
4、字符串“R4ndom”的地址

5、让我们进入函数的那个 CALL 的返回地址。

现在，sayHello 可以填充、修改以及重用这些地址以用于我们的变量，在第一个位置处有传递给函数的参数（就是字符串“R4ndom”)。当 sayHello 执行完毕后，它有两种方法来删除这些变量和参数（因为函数执行完毕后不在需要它们)，然后将堆栈还原：1）它可以将 ESP 寄存器修改回它被修改之前；2）使用后面带数字的 RETN 指令。第一种方法，为了让程序能够记住 ESP 的原始数据，它使用了另一个寄存器——EBP，目的是当我们第一次进入 sayHello 函数时能够追踪到堆栈指向的原始位置。当函数准备返回时，它从 EBP 中拷贝 ESP 的原始值（开始的时候存储在 EBP 中）到 ESP 和 BAM 中。返回地址现在在堆栈的顶部,当 RETN 指令运行时,它用通过这个返回到我们的主程序中。

第二种方法，你可以告诉 CPU 堆栈中有多少字节你不再需要了，然后它就会从栈顶删除这些字节。在我们的例子里，我们用 RETN 16（十六进制就是 0xF),这样就会从栈顶除去 16 字节(或 4 个 32 位数),将返回我主程序的地址留在新的栈顶。具体的返回机制依赖于编译器,不过你两个都会看到。

现在，我们回到我们的 FirstProgram.exe：



可以看到 Olly 已经注释出了一个参数和 12 个本地变量。在我们的程序中这些本地变量是用来追踪类似于图标、我们输入的文本的缓存地址、输入的文本长度等。完成后，就会弹出这些值、将 ESP 寄存器值改回 EBP 或 RETN 一个数字（本例中，三个都有!!!）

我知道堆栈是非常复杂的设计，但是我保证在混乱一段时间以后你会掌握它的窍门。汇编语言的书也会帮很大忙的 。

（最近忙着装修进度较慢，而且第三章和第四章真的好长，这一章近万字，翻译不易呀）

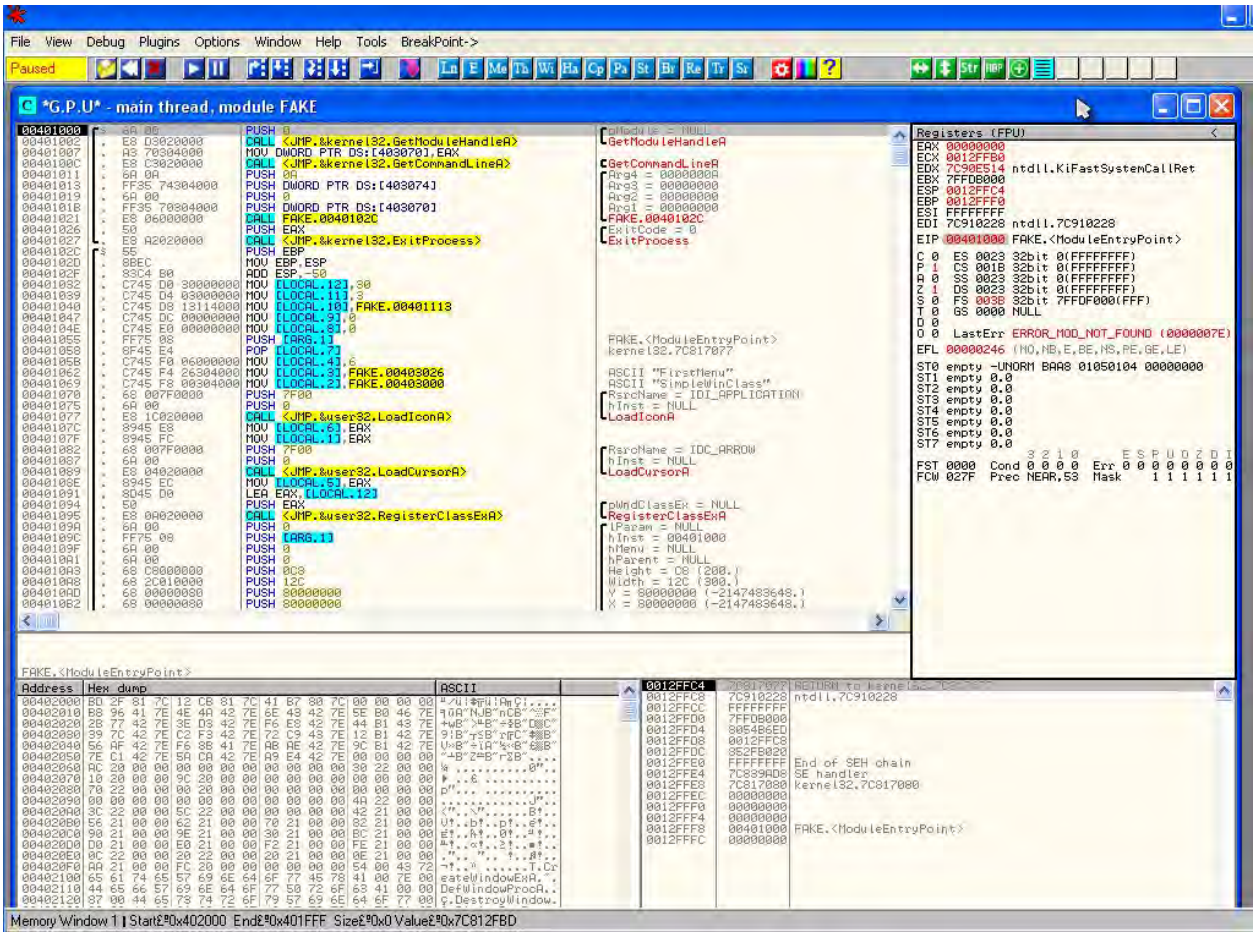
教程五：第一次破解（算是）

一、简介

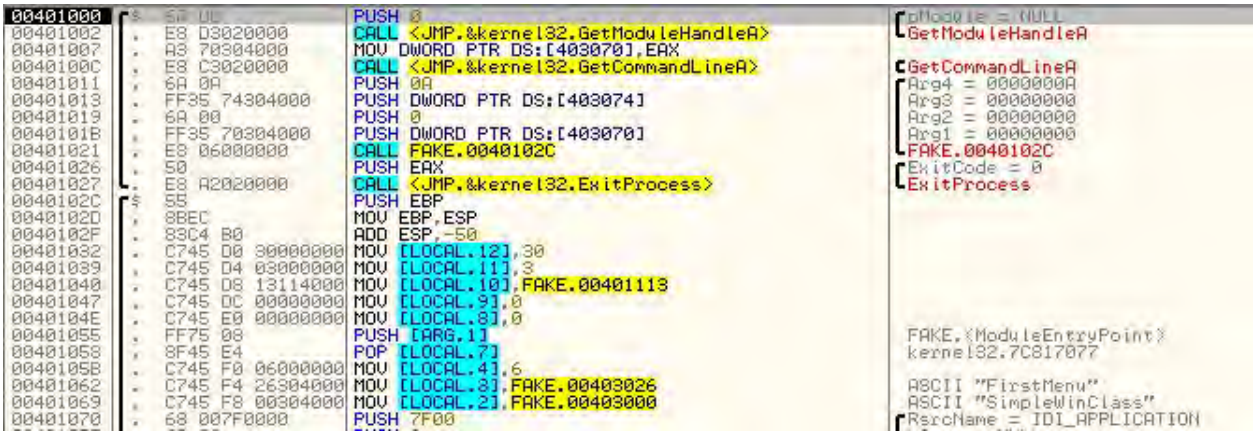
此次教程通过预览一个 crackme，我们会结束 Olly 使用方面剩下的内容。好吧，算是一个 crackme。其实就是我们前面使用的程序，不过被修改成需要序列号注册了，如果输入正确序列号会显示一个好消息，否则显示一个坏消息。我选择这么做而不是用一个完全不同的 crackme，是因为我想要你能够专注于序列号校验程序部分，而不是陷入其他的代码中。下一课我们会研究一个真正的 crackme（我保证）。

此次教程你所需要的就是一个 OllyDBG（我的版本或者原始版本都可以），以及一个我改进了的 crackme。顺便说一下，我把改进后的 crackme 叫做 “First Assembly Kracking Engine”，或者是 F.A.K.E。它包含在此次教程的文件下载中。(是的，Gdogg，我知道 Kracking 不是以字母 “K” 开头的。译者注：如果取 crack 首字母，最后缩写就是 face 了，还有那个 Gdogg 我不造啥意思）

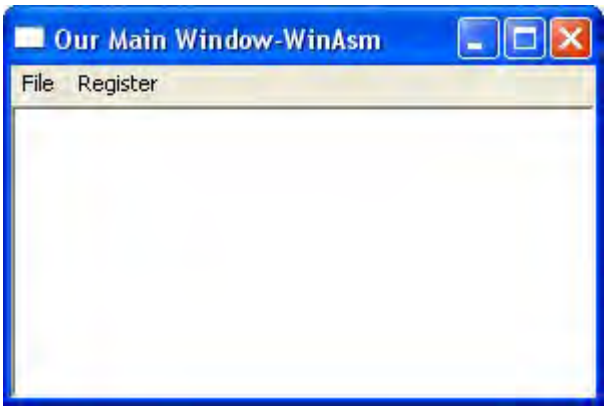
我们开始吧。



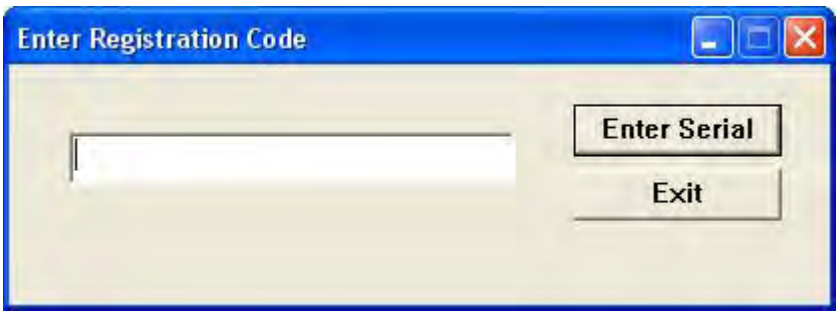
如果你在 Olly 中载入了 FAKE.exe 的话，就会发现第一页代码和我们上一次学习用的程序一样。



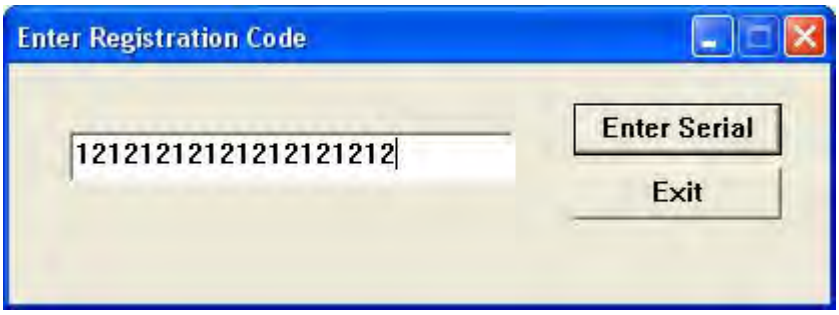
运行下程序，看看它如何工作的是及其重要的。



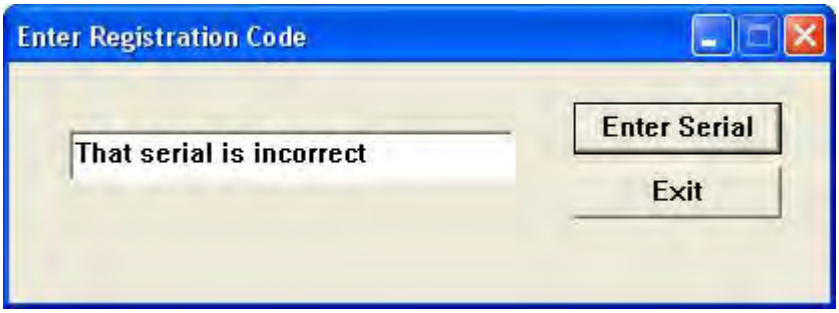
点击注册弹出下面的对话框。



输入序列号。



在点了 Enter Serial 后，出现了下面这个坏消息。



真见鬼！我那么努力的尝试!!!

现在我向你介绍每个新手查找注册校验代码的第一个方法。

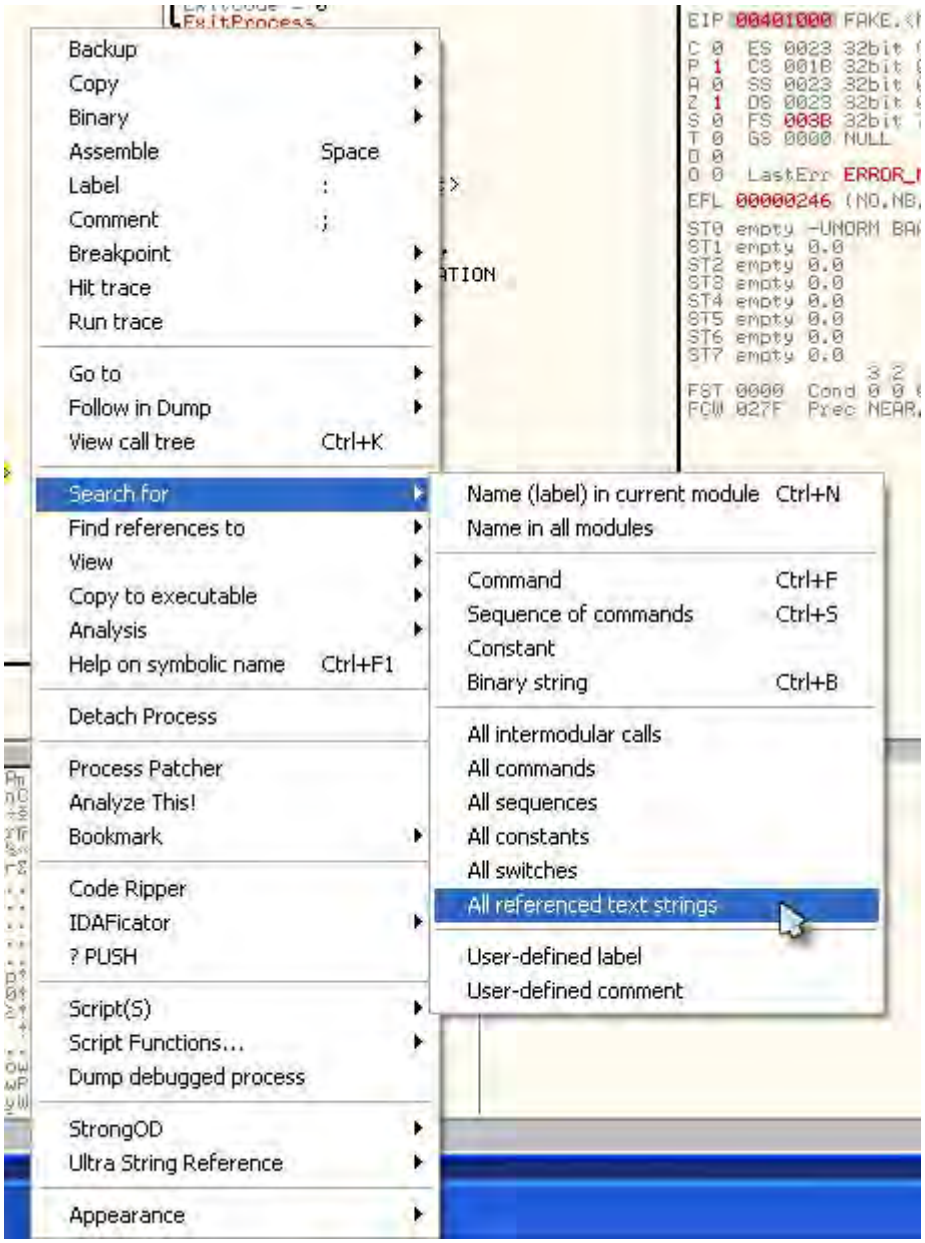
二、搜索所有的文本字符串

先说一下，有许多“老练”的逆向者（或破解者）觉得这个方法已经很少用了。因为这个方法太过于明显了，所以凡是想保护自己的软件不被逆向的人都会让这招失效。这些软件被压缩、保护、加密或修改，只要作者不是一个完完全全的傻子，就会加密字符串以让“Search for strings（搜索字符串）”方法失效。话虽如此，不过我还是发现有许多傻子，这个消息可别告诉任何老鸟，所以我做的第一件事就是检查这个（ps.这其实也是老鸟做的第一件事）。

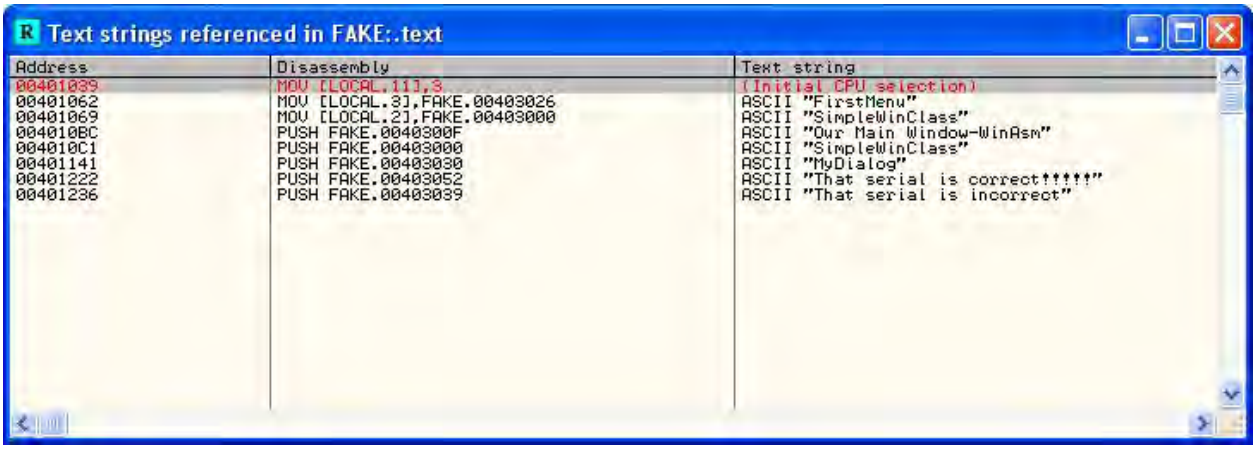
基本上，该方法都会涉及到让 Olly 搜索你的程序的内存空间，搜索任何看起来像是 ASCII 或 Unicode 文本字符串。通常，可以立即发现该方法好不好用，会有大量的文本字符串，许多看起来很诱人（比如“Thank you for registering!!!（谢谢注册!!!）”）。或者是有很少的字符串，而且许多都像这样“F@7=”。

了解一个二进制文件中是否有合法字符串可以给你一些有价值的信息。比如二进制文件是否通过某种方法被压缩或保护，是否是一个恶意二进制文件（毕竟，“Send all user's passwords to www.badguys.com”这样的句子不会是一个非常负责任的病毒所写吧），甚至二进制文件是用非常少见的语言所写。

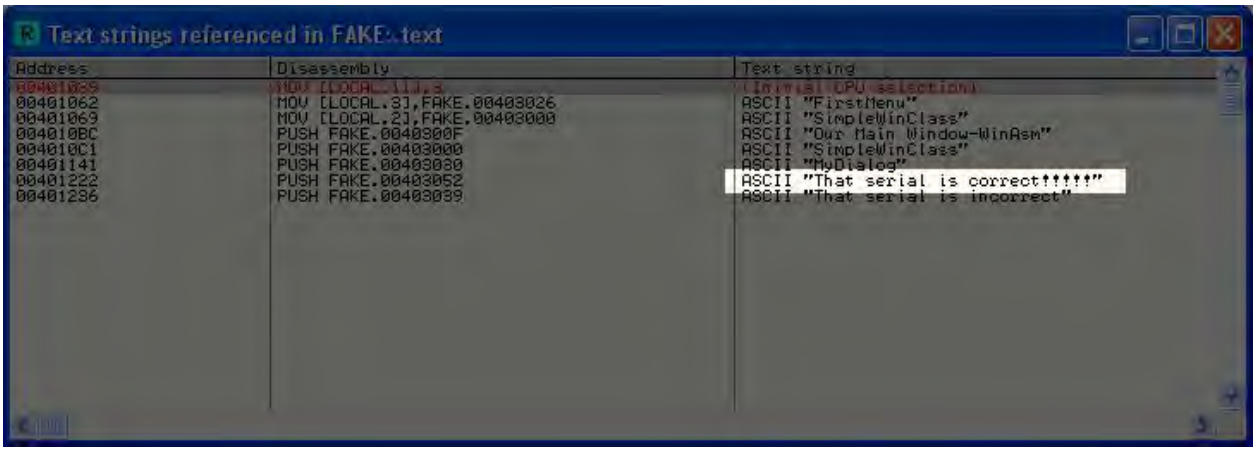
咱们来看看具体怎么做。右键反汇编窗口，选择“Search for” -> “All Referenced Text Strings”。



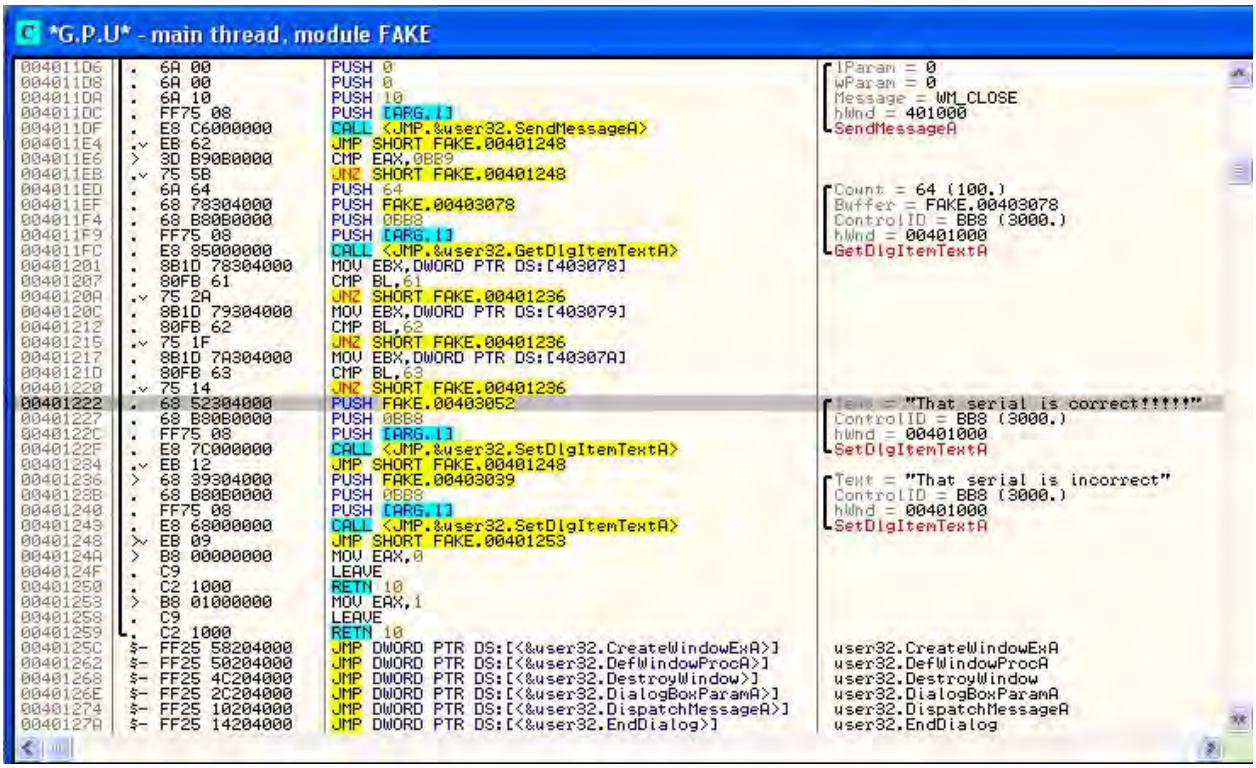
然后 Olly 就会搜索程序的内存空间，并显示文本字符串窗口（Text Strings Window）：



嗯，看起来很有意思吧:)注意这个列表是真的短，因为这个程序确实非常短小。一般来说，会有数千行字符串。还有，你注意到我注意的了吗:



前途有望啊。咱们跳到代码那看看有什么：双击“That serial is correct!!!!!”那一行，Olly就会在反汇编窗口显示那一块代码:



是时候介绍第二条规则了

R4ndom’s Essential Truths About Reversing Data:

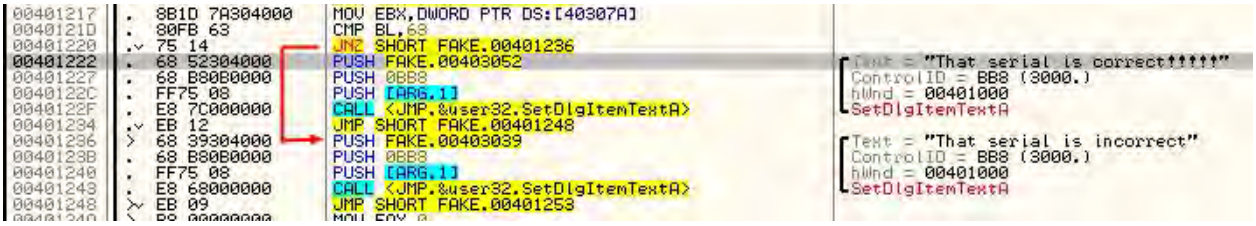
R4ndom 关于逆向数据的必备真理：

#2：大多数的保护机制是可以简单的通过修改一个跳转指令来绕过“坏”代码直接跳到“好”代码的。

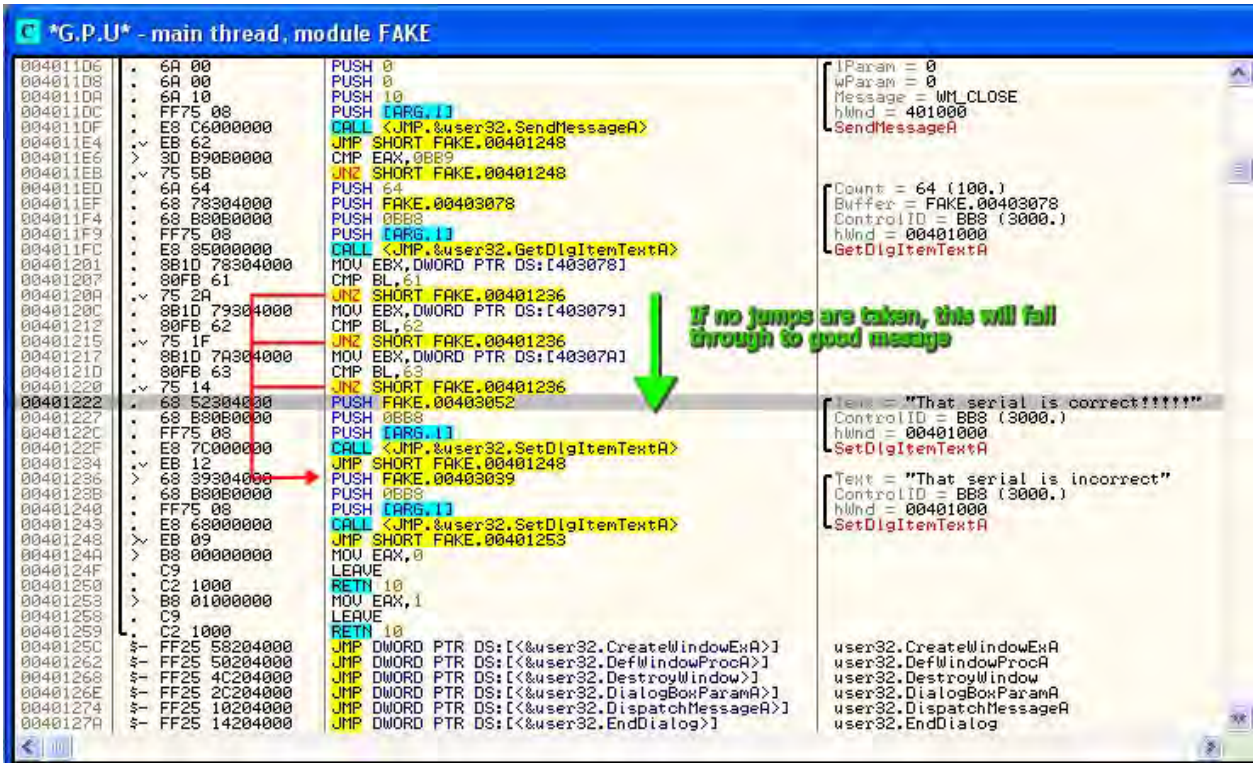
意思是几乎每一次在坏消息显示之前，就会有某种检查（我们注册了吗？注册码对不对？试用时间过了吗？.....），对比之后就有一个跳转，至于是跳到好消息还是坏消息则依赖于对比的结果。

我们自己来找找看啊....。好消息“This serial is correct!!!!”是从 401222 开始的，向上翻找跳转语句，尤其是

它前面有某种比较（或 CALL）的跳转语句。如果是一个 CALL，可以猜测比较是在 CALL 内部进行的....。我们的例子中，第一个跳转是在 401220 的 JNZ。我在图中加了一个箭头，向你演示了如果跳转成立的话，将会跳到哪去:



嗯。注意它刚好跳过了我们想要的消息，跳到了我们不想要的消息😟。不过，注意在 JNZ 指令的前面是一个 CMP 指令😀。意思是，这个是 Olly 决定显示我们想要还是不想要的消息的关键点。我们再向上翻翻:



在 401212 有另一对 CMP/JNZ，在 401207 有最后一对。凑近点看，你会发现所有的三个跳转都跳过了好消息，跳到了坏消息那。逻辑上，这意味着有三件事被检查，触发任何一个都会命中坏消息。不过，如果我三个跳转都不跳会怎么样？好吧，你会看到我们将空降到好消息那。所以，真正的意思是，如果我们让这

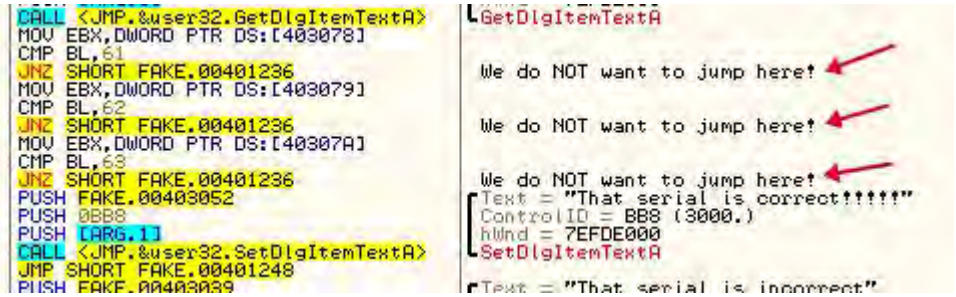
些跳转都不跳，程序会“空降”到好消息那里（译者注：这里作者用的是“fall through”，大概意思是如果将三个 jmp 指令当做一层层的阻碍的话，我们直接穿过这些阻碍到达显示好消息的代码，就是将这三个 jmp 无视掉当作透明的。有些东西可意会，不好言传，所以我将它翻成“空降”）。

我们运行下程序看看它做了什么，不过我先向大家介绍点别的。

三、如何添加注释

注释是很重要的，尤其是在你开始分析错综复杂的代码时。代码本来就很难读，不过有了注释后，我们就可以在非常重要的地方提醒自己。我准备为每个 JNZ 指令添加注释，以此来提醒我们自己什么需要被发生。

要添加注释，你可以双击要添加注释的那行的最后一列（那里，Olly 已经放置了类似于“This is the correct serial!!!”这样的其他注释），也可以先选中要添加注释的那行，然后按一下“;”键。好，我们先选中 40120A 那行，然后按一下分号键，接着输入“We do NOT want to jump here!”。现在，给 401215 和 401220 添加同样的注释。这样就给每个 JNZ 指令添加了注释：



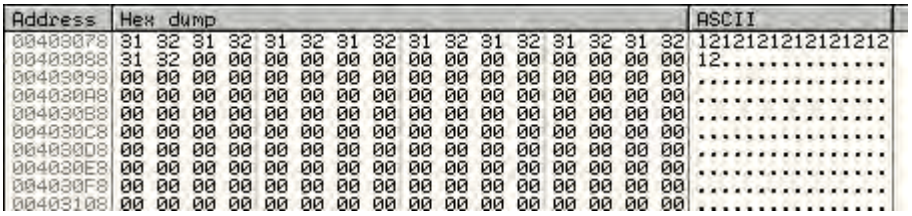
现在让我们在 401201 处设置一个断点（在跳转指令前面的其他地方设断点也行）：

让程序跑起来。点击 crackme 上面的 “Register”，输入序列号，再点一下 “Enter serial”。Olly 就会暂停在断点处:

现在，我们第一个要注意的是我们停止的那行:

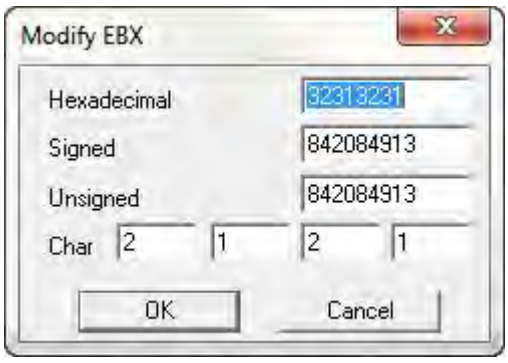
MOV EBX, DWORD PTR DS:[403078]

从上一课中我们知道该如何查看该内存地址的内容，在指令上右键，选择 “Follow in Dump” ->"Memory Address"。然后我们就可以在 Olly 的数据窗口中看到该内存的内容:



好，好，好。这不就是我们刚刚输入的序列号嘛。所以，根据这条指令，我们知道了前面四个字节（因为 EAX 是 32 为寄存器）被载入 EBX,也就是 31 32 31 32,用 ASCII 码表示就是 “1212”。按一下 F8 再检查 EBX:

如果你想看看 EBX 中的 ASCII 字符,你可以双击 EBX 寄存器,就会显示几组不同格式的数据，其中一组就是 ASCII:



**为了后面用到，如果你想对不同的寄存器尝试不同的值得话，记住这也是“即时”修改寄存器的一种方法。*

我猜你已经从汇编语言的书中知道了这种方法(我的意思是,来吧！我甚至在工具区上传了一个!!!)，我不需要讨论这个，只需要复习下。

四、小端序列

（至少你需要了解这方面内容）

处理器在内存中存储数据是不同的,这依赖于处理器的架构。内存中的数据存储有两种方法: 一个叫大端（Big-Endian）,另一个叫小端（Little-Endian）。Intel 用的是小端，你必须要适应这个,否则你会晕头转向的。举个例子:假定一个地址 7E04F172（是一个 4 字节，32 位数）。将其按字节拆分，会得到 7E、04、F1、72。现在，人们可能会认为将这些字节存储在内存（假定地址是 1000）中时应该是这样的:

1000::7E

1001::04

1002::F1

1003::72

任何正常人都会这样想。但是 Intel 的开发人员比我们这些普通人更聪明，他们决定以一种更加符合逻辑的方法来存储:

1000::72

1001::F1

1002::04

1003::7E

上面的第一个例子是大端序列，意思是数字的最大的那端（以十进制序列形式）在内存中最先被存储。因为 7E000000 比 040000 大，所以第一个字节被存储在第一个位置，第二个字节被存储在第二个位置，以此类推。第二个例子（明显更加的聪明）叫做小端序列，意思是首选存储最小的字节（案例中是 4 号字节），后面依次是第三个字节、第二个字节、第一个字节。因为 72 小于 F100，所以会被先存储。

当你在内存中从一边往另一边看的时候，就会发现用小端而不是它大哥真的很天才。在大端中，数字 7E04F172 看起来像这样:

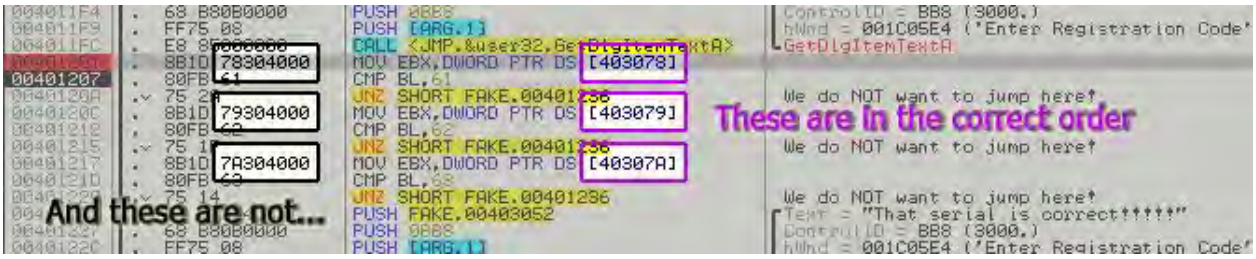
7E04F172

明显很乱。感谢上帝，使用小端的话，同样的数字 7E04F172 看起来更具有逻辑性:

72F1047E

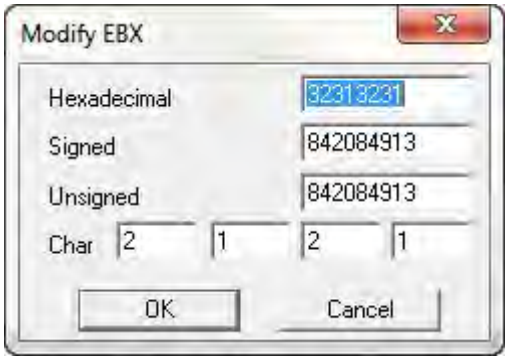
你说啥？蠢的太明显了吧，大端更合理吧。但话又说回来，你又不是 Intel 的半人半神的开发者，所以你甚至不具备弄明白为什么这要优越得多的脑力。无论如何，先不管各种讽刺，这意

味着当你看代码时，无论是磁盘里的还是内存里的，你必须将 4 字节数字反过来。当然，Olly 有时已经为你做了这些，这让情况变的更糟了，就像下面的图片这样:



到目前为止，这是我想要说的全部。不过，过会我就会告诉你字节序。

现在，回到我们的寄存器窗口:



注意,十六进制的是小端序列(应该是 31323132),那个 Char 是向后的，因为我的序列号是以 1212 开头的，而不是 2121。相信我，你会用到这些的。

现在看看下一条指令:

CMP BL, 61

这是一个很明显的比较语句，比较 BL 的值，也就是 EBX 寄存器的第一个字节与 61（hex）进行比较。我们真的没什么线索来了解这是什么意思，所以我们单步步过它。最后我们来到了第一条 JNZ 指令:

JNZ SHORT FAKE.401236

这里我们回想一下，我们可以看到我们前面做的注释，就是我们不想让这个跳转实现。这里提醒一下，JNZ 的意思是非 0 的时候跳转。所以，这两行的意思是“如果 BL 的值不等于 61h，就跳转到坏消息”。我们可以清楚的看到 EBX 寄存器的右边的字节（BL）不是 61h，而是 31h。我们已经卡在这了，那个跳转会实现的，但是我们又非常的不想要它实现😟。

等等！Olly 是一个“动态”的调试器，我们应该可以动态的实现跳转！好吧，因为你很可能已经读了汇编语言书籍中关于标志位的整个章节，所以我不准备讨论这个。

五、CPU 标志位

前面的章节中我们简要的讨论了标志位，我也确实不准备深入的探讨这个问题，因为我确信你的汇编语言书籍的目录中有一个“F”章节。标志位可以让处理器知道某条指令的输出是什么。在 Intel 库中有大量的指令可以影响到标志位，不过最重要（至少对于逆向来说）的是“compare(比较)”指令。基本上，CPU 比较两个项目之后，会根据它们的相互关系属性（相同？一个大？一个小？）来设置标志位，再根据这些标志位来执行相应的跳转语句。这其实是表达 IF THEN 语句的非常奇特的方式。例如，在高级语言中有如下代码：

```
if( serialNumber == 3 )
    dontShowNag();
else
    showNag();
```

用伪汇编语言来表示，同样的指令应该类似下面的代码：

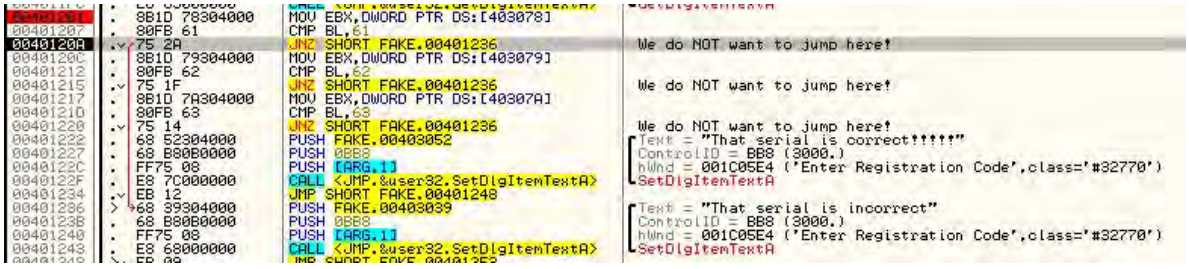
```
compare serialNumber with 3
  jump (if they are equal) to dontShowNag();
  jump to showNag();
```

用真正的汇编表示有可能像这样：

MOV EAX, addressOfSerialNumber CMP EAX, 3 JE addressOfDontShowNag JMP adressOfShowNag

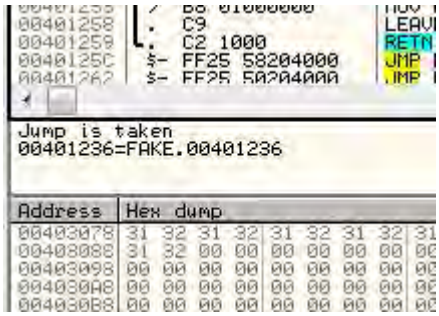
首先，EAX 中存储着我们的序列号。下一步，它和“3”进行比较。如果等于 3 就跳到 dontShowNag()。如果不等于 3，就跳过 JE（如果相等就跳转——jump if equal）指令，执行 JMP（JuMP）指令。不管任何标志位，自动跳到 showNag()。

重要的标志位（对于我们来说）有 0 标志位和进位标志位，在 O11y 中分别显示为“Z”和“C”。基本上，通过修改两个标志位中的一个，我们就可以阻止（或者强制）程序中的任何跳转，

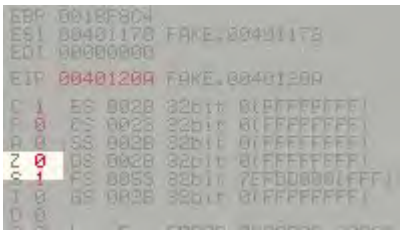


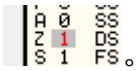
就像我们下面要介绍的：

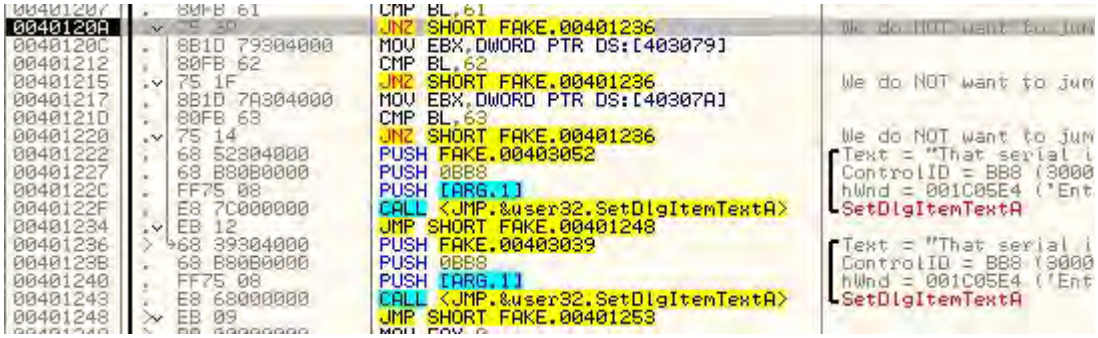
在暂停的那行（第一个 JNZ），通过那个红色的箭头，我们可以看到 O11y 准备执行这个跳转。如果该跳转不会被执行，这条线就会显示灰色。如果你没有用我所用的 O11y，就不会有这个箭头，这样的话你可以看反汇编窗口和数据窗口中间的那一块，O11y 会告诉你跳转会不会被执行。本例中，会有如下显示：



现在我们知道了，如果不做点什么的话，O11y 就会执行该跳转。那我们就干点什么吧。看看寄存器窗口，找到“Z”标志位：

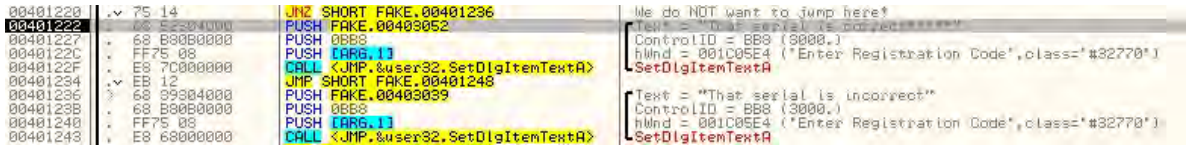


注意那有个 0。意思是，在 61h 和 BL 的内容（31h）之间的比较结果是 0，或者叫 false，所以它们不相等。现在我们明白了为什么 不是 0 就跳转 指令会跳转了，因为就目前来说，0 标志位没有被置位，所以它是“非 0”。现在，双击零标志位后面的那个 0，它就会变成一个 1:  。然后注意看那个箭头，变成灰色了（O11y 也会提示跳转不成立）:

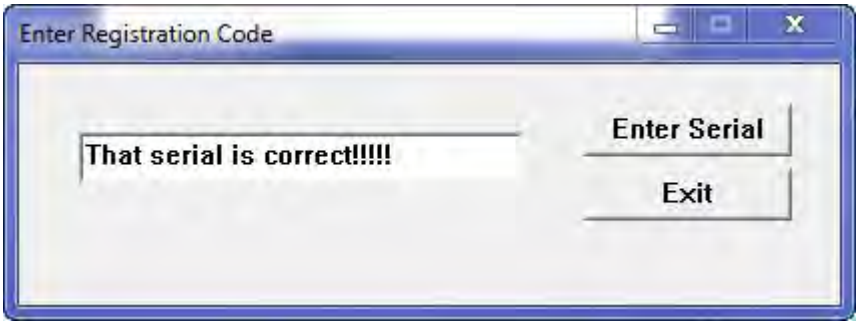


我们已经修改了 O11y 的标志位，同时我们也修改了程序的行为😀。大哥继续，按下 F8（你已经学会了），我们不会执行该跳转:O。我们现在来到了看起来一样的代码段，除了 EBX 中存储的是我们序列号的第二个字符，它将会与 62h 进行比较而不是 61h:

我们知道我们序列号的第二个数字并不是 62h，现在知道该怎么做了吧。F8 直到 JNZ 语句，双击零标志位，继续下去!!! 你会跳过那个 JNZ 指令。快要成功了!!! 最后一个是将我们序列号的第三个数字与 63h 进行比较。我们序列号的第三个数字是 31h，所以该跳转正常来说是要执行的。继续，你知道该怎么做的。跳过了第三个跳转，我们来到了 401222:



你的心是不是开始扑通扑通的了，因为我认为我们都知道接下来会发生什么。在我们和救世主之间再也没有跳转了，所以无论你是单步步过下面的指令（如果你喜欢留悬念的话）还是直接运行程序（如果你和我一样不能忍受悬念的话），我们终会到达天堂（译者注：这段比喻感觉好猥琐）：



六、家庭作业

我知道你不喜欢，这一章已经让大家很兴奋了，不过我会以两件事来结束本章。第一个是：

R4ndom’s Essential Truths About Reversing Data:

R4ndom 关于逆向数据的必备真理：

#3：仅仅读教程，你是学不会逆向工程的。你必须亲自操作，而且必须大量的实践。

根据新规则，我会留一些作业。你的任务，你应该已经接受了，找出序列号。意思是，在让 JNZ 跳转不实现的情况下，你必须在序列号文本框输入的内容是什么？在不以任何方式修改应

用程序的情况下，在输入正确的序列号以后，程序显示了“That Serial is Correct!!!!!!”，你就知道你找对了。

ps.如果你需要提示，你可以点击这个链接。

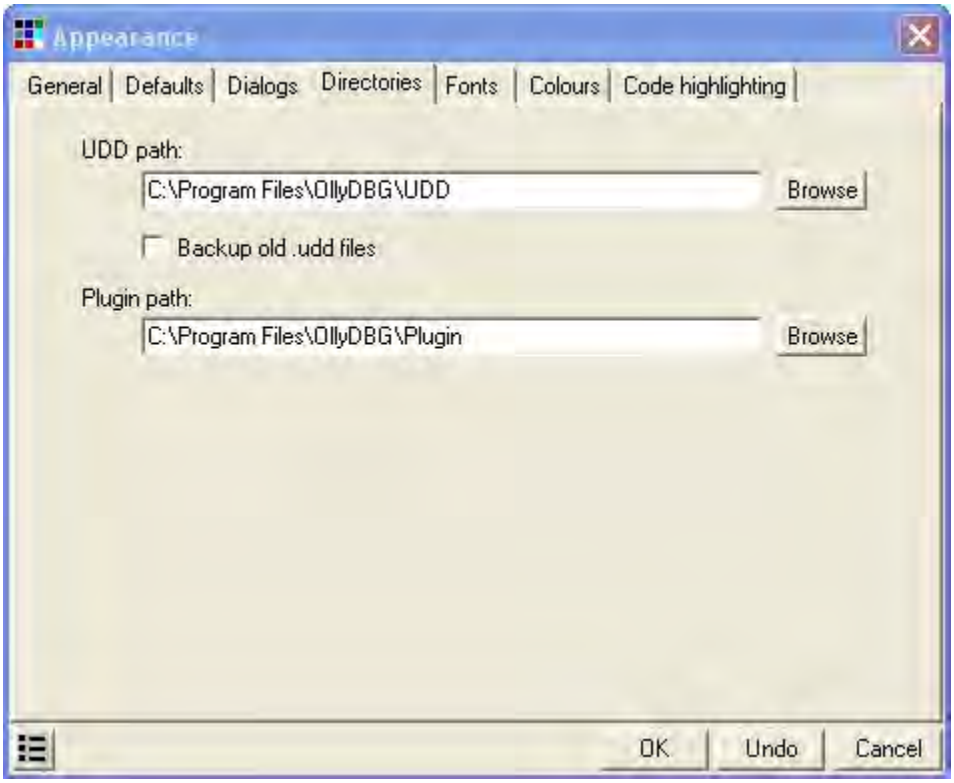
第六章：第一次（真正的）破解

一、简介

欢迎来到我的教程的第六章。本章我打算离一个真家伙近点：一个真正的crackme。它也包含在本章的下载中。crackme是一个渐进式学习逆向工程的好方法，而不应该直接从“真正”的程序入手，crackme可以从易到难进行，这样你就可以以线性方式学习。最终，我们会一路走到真正的程序，不过也要看到我们才刚刚起步，这些crackme也会带给我们巨大的挑战。

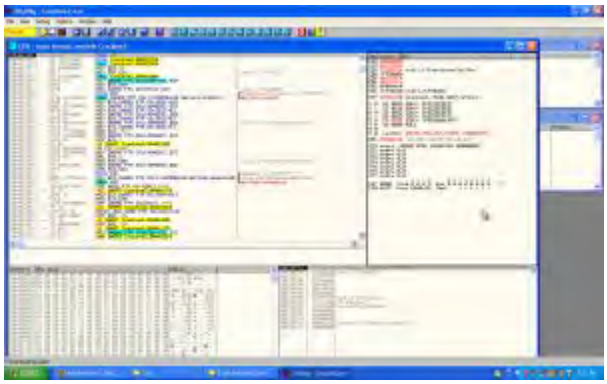
你可以在教程页下载到相关文件和本教程的PDF版（译者注：英文版的，此中文翻译我会在教程的最后放出）。

我将会用OllyDbg1.10（我的版本或原始版本都行，不过如果你用我的版本的话，它看起来和图片一样）。我推荐你从工具页面的Olly Plugins下载“MnemonicHelp”插件，因为本教程将会用到（教程的下载中也包括的有）。解压后，将其与x86eas.hlp文件放到Olly文件夹下的plugins目录下。如果没有plugins文件夹，就在Olly的主目录下创建一个。然后打开Olly的Options->Appearance->Directories标签，然后选择你放置插件的目录。你再在Olly的主目录下创建一个叫“UDD”的文件夹，然后让当前设置页的另一个选项也指向这个文件夹。UDD文件是Olly给一个程序做的“便条”，你设置的所有断点、做的注释、一个二进制文件的特有设置都会存储在UDD文件中，通常叫做“程序的名字.udd”。如果你在逆向时需要离开一段时间做别的工作，UDD文件可以让你回来继续对程序进行逆向，因为所有的都被保存起来了。下面是设置两个目录的窗口（带有我的设置）：



二、探究二进制文件

先载入 Crackme2.exe:

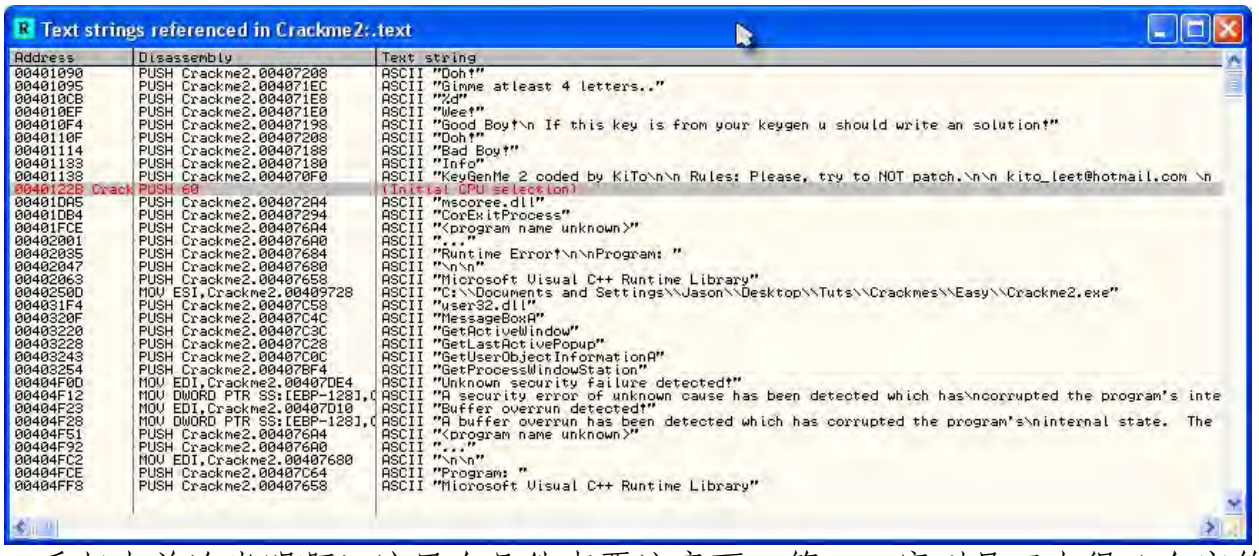


我前面说过，在开始之前的最重要的事情是运行程序看看情况。这可以给你大量的信息：有没有试用时间？是不是有些特性被禁用？是不是只能在有限次数内运行？有没有注册窗口让你输入注册码？

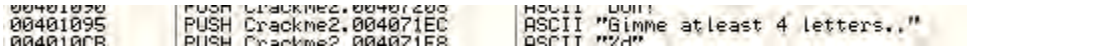
这些都是需要知道的很重要的东西，随着你在逆向领域做得越来越好，你会获得越来越多的经验让你知道应该找什么（需要多长时间来验证注册码？是不是强制你访问一个网站？.....）

看起来比较直接了当。我们来试试:

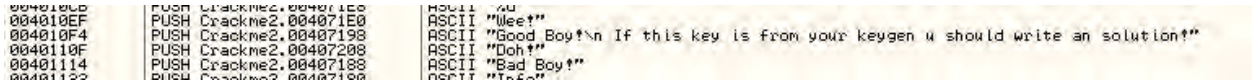
这不是我们想要的。我们看看能不能让 OllyDbg 做些有用的事。回到 OllyDbg，我们来试试我们当前知道的第一个(也是唯一一个)工具。搜索下字符串。右键->Search For->All Referenced Text Strings:



看起来前途光明啊！这里有几件事要注意下。第一，序列号至少得 4 个字符。



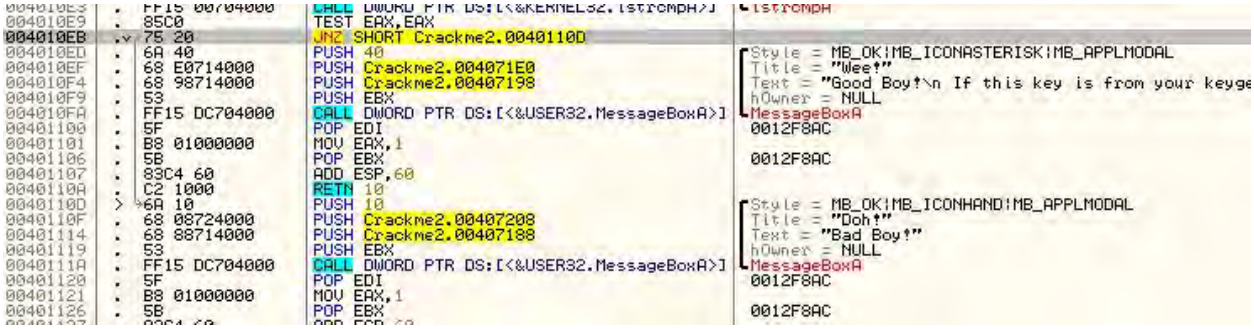
第二，我们准确的知道了显示好消息和坏消息的地方在哪了。



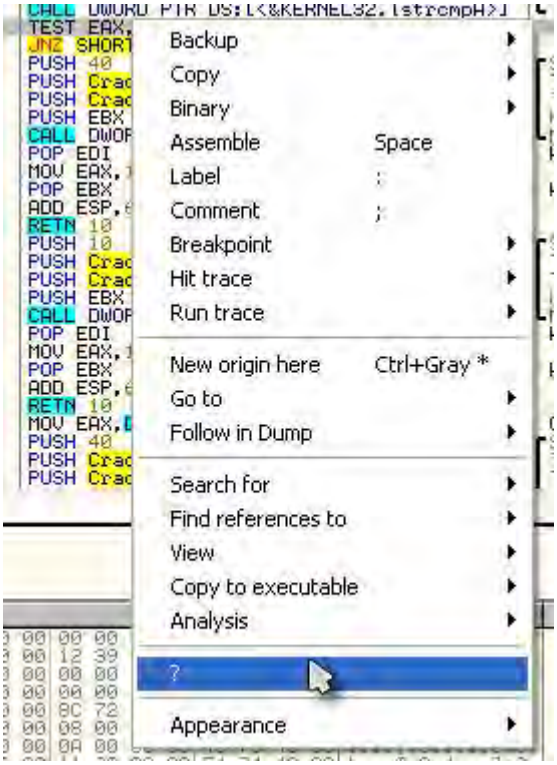
点一下 4010F4 那行，看看有什么东西:

这是处理简单 crackme 相当标准的流程（简单点的商业程序也是一样）。搜索文本字符串、找到显示你的注册码/密码/许可证正确与否的相关信息，然后转到那部分代码，你就会看到好的和坏的消息彼此间靠的相当的近呢。那么，根据 R.E.T.A.R.D. 的 2 号规则，查找 比较/跳转语句，以及你想要的那个 CALL。咱们来找找那个跳转语句。

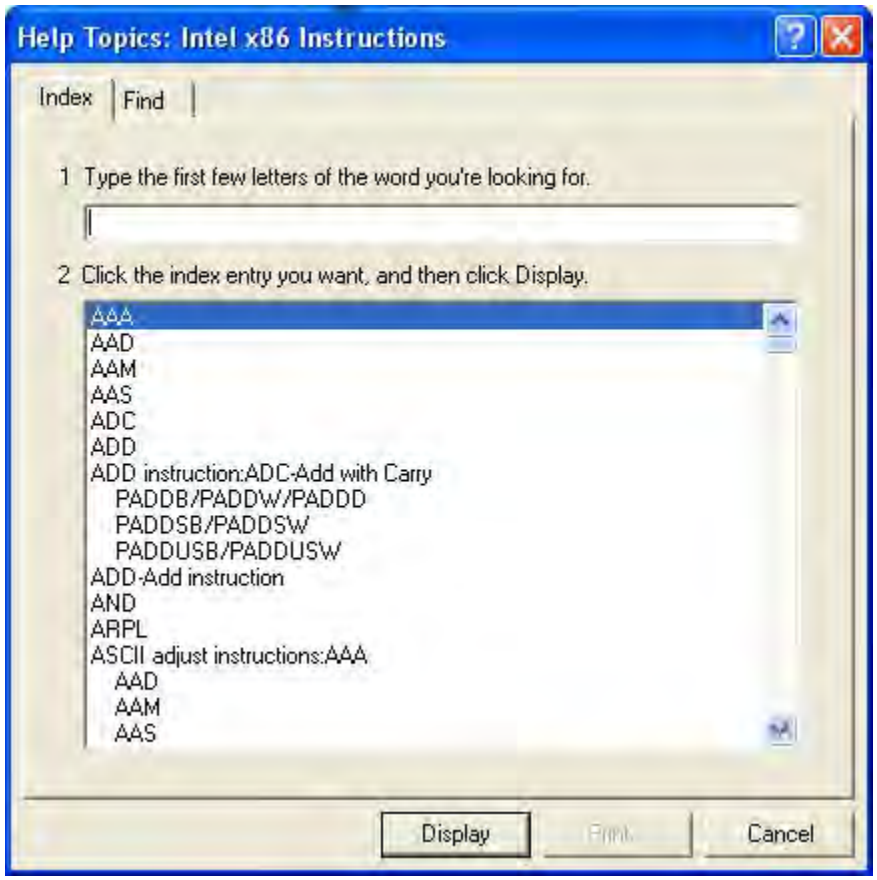
我们找到的第一个跳转是在 4010EB，一个 JNZ 语句。如果我们点击这一行，Olly 就会向我显示它会跳向哪里。



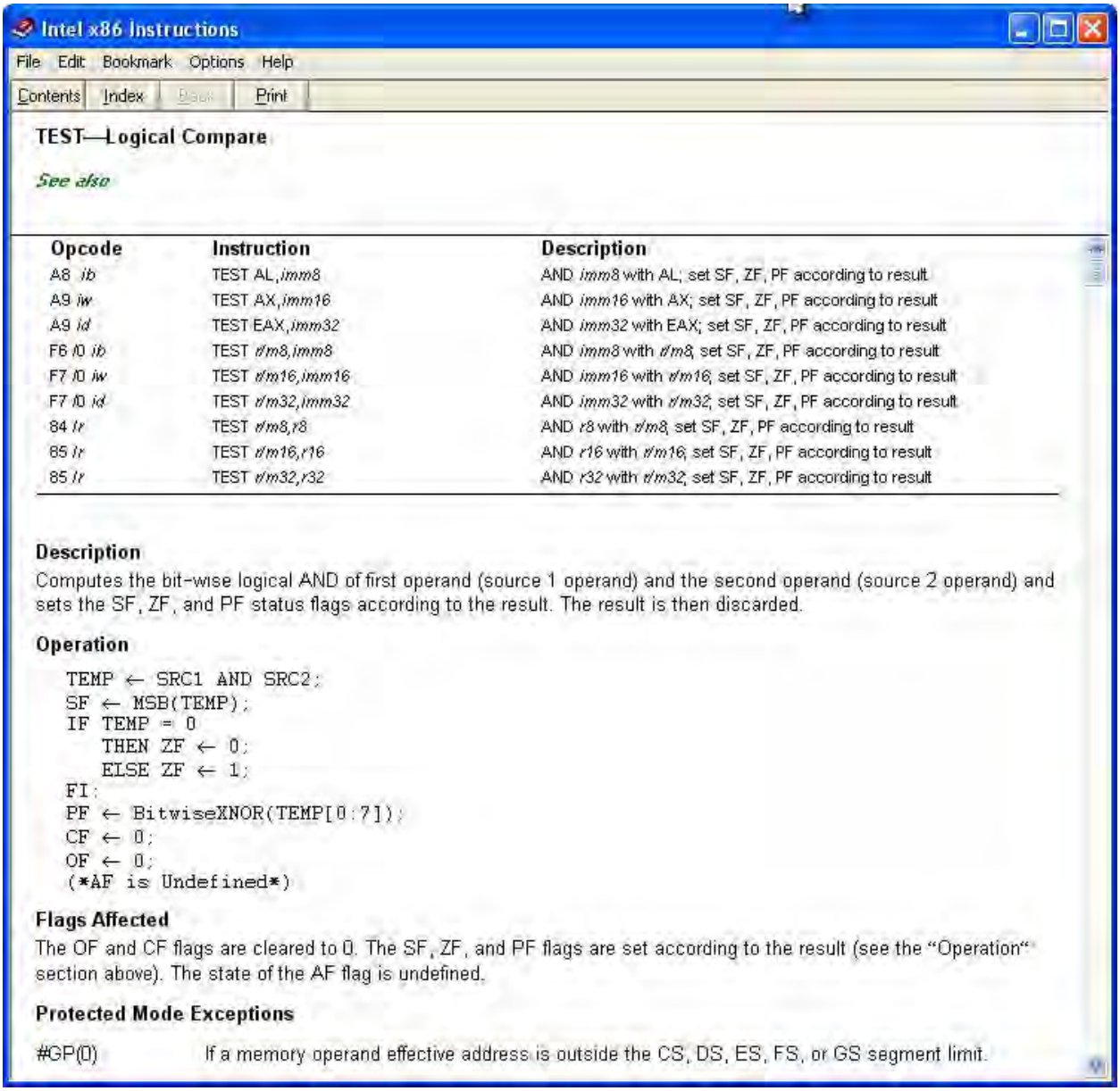
可以看到，这条指令跳过了 “Good Boy”，直接跳到了 “Bad Boy”。这看起来是一个关键点。我们都知道，一个跳转的前面一般都会进行比较，以此来决定是不是要进行跳转。往 JNZ 指令的上面看，我们可以看到一条 TEST EAX, EAX。你可能还没有学到汇编语言书籍关于 TEST 指令的部分，我们来看看能不能找到这个 TEST 指令是干什么的。在本章的前面你已经安装了 MnemonicHelp 插件，那就是我们要用到的。在 TEST 指令上右键，你会在右键菜单中看到一个问号。点它:



就会打开 Mnemonic 帮助窗口:



在上面的文本框中输入“Test”，然后选择（双击）“TEST”。然后就会显示相关指令助记符的帮助:

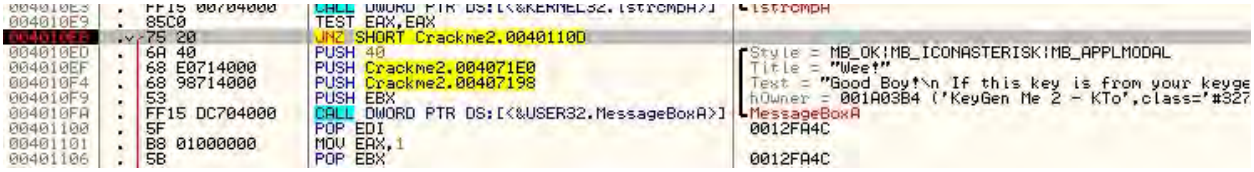


我们就可以看到 TEST 指令意思是“*Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.*（译者注：这段就不翻译了，一是这是帮助中的原文，主要是向大家演示；二是，TEST 指令的意思咱们也可以 GOOGLE 的，中文比看这个容易多了。)”。大部分的时间里，如果 TEST 指令正在测试的两个寄存器的指令相同，就意味着它正在检查它们是不是 0。所以这个满足我们跳转之前要进行比较的需求：

```
CALL DWORD PTR DS:[<&KERNEL32.lstrcmpA>]
TEST EAX,EAX
JNZ SHORT Crackme2.00401100
PUSH 40
```

这两条语句的意思是“如果 EAX 不等于 0，就跳到 40110D”，也就是“Bad Boy”那里。好吧，这当然不是我们想要的，咱们来试试我们的推测。在 JNZ 指令处设置一个断点，重启应用。输入用户名和序列号（记住，至少四个字符☺），点击 crackme

上的 check 按钮。Olly 就会断在我们的 BP（译者注：BP 即是 breakpoint，以后就直接用 BP 不再注释了）处：



现在，我们可以看到我们将会跳过 good boy，直接到 bad boy。咱们来让它不发生。帮 Olly 翻转 0 标志位（参见前面的教程）：

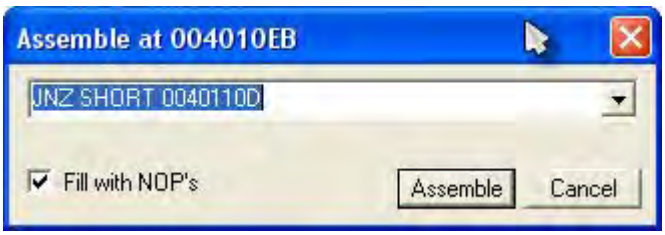
我们可以看到，现在那个跳转没有实现。运行程序看看：

耶，这就是我们想要的。***忽略那个关于 keygen 的消息，有些 crackme 还有其他的目的要求，不过我还是用它们，我们也需要来学习它的其他两点。一旦我们从这个系列教程中学到了更多的知识，我们还会回来使用它们中的许多。

三、打补丁

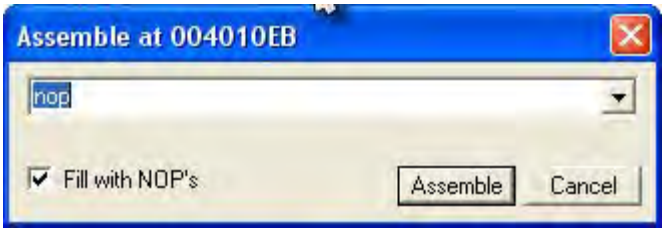
重启 crackme，运行之，输入用户名和序列号，Olly 就会断在我们的断点处。你会注意到，我们会再次跳到 bad boy，因为改变 Olly 的标志位只是临时的方法。这回我们不去临时修改标志位，我打算修改二进制文件中的代码来完全我们想要的。这个叫做打补丁。

点击我们暂停的那行（4010EB），点一下该行的指令列（有 JNZ SHORT...的那部分），然后按一下空格键。会有一个显示该行指令的窗口弹出，也是修改指令的对话框：



现在，我们要做的是将这个跳转到 bad boy 消息处的跳转改成永远不会跳，意思是我们确实不想让这个跳转实现。我们准备做的是，将其替换成一个什么都不做

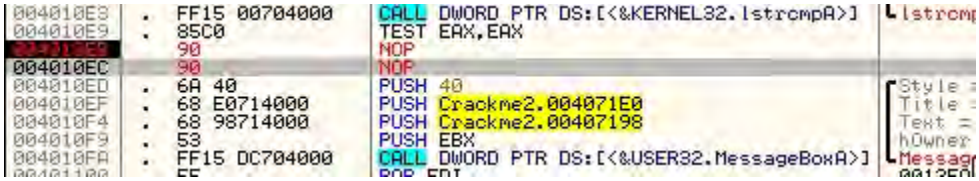
的指令，那就是 NOP 指令。NOP 意思是 No OPeration（不操作）。将对话框中的 JNZ SHORT 0040110D 修改成 NOP:



那个“Fill with NOP’ s”复选框就留那不用管。现在点一下 Assemble 按钮，提交所做的修改，再点一下 Cancel 按钮关闭窗口。

***顺便说一下，如果你没有点那个 Cancel 按钮，而是一直点 Assemble 的话，你会一行一行的修改每一行。这是 Olly 的一个“特性”，用来让你一次修改好几行代码用的。可以让你不用每行都敲空格键。我保证你第一次打补丁的时候会让你疯掉的:X。*

注意我们暂停的那行已经改变了，那条指令现在变成了两个 NOP，而不是 JNZ 指令了，并且变成了红色（因为 Olly 对于被修改的项目以红色显示）。

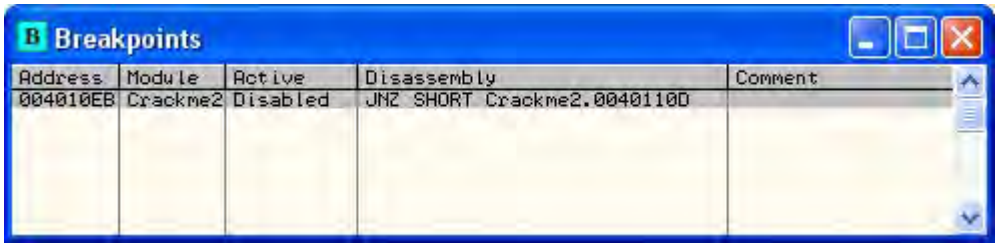


有两个 NOP 的原因是，NOP 操作码只有一个字节长，而被替换的 JNZ 指令有两个字节长，所以 Olly 用两个 NOP 来替换。你也会注意到跳转箭头消失了，因为这行已经不再有任何跳转了！现在单步运行，你会走到 good boy 处。然后 good boy 显示出来了，你开心的笑了☺。

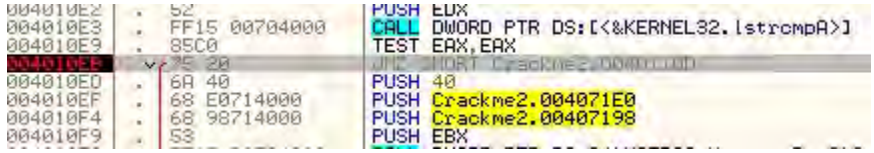
四、保存补丁

有一个重要的事情要注意，如果你重载或重启应用的话你所打的补丁就没有了，除非你将补丁保存到二进制文件中。你可以看到补丁在起作用，回到 Olly 打开 Patch 窗口（点击 Pa 图标或 Ctrl+p）:

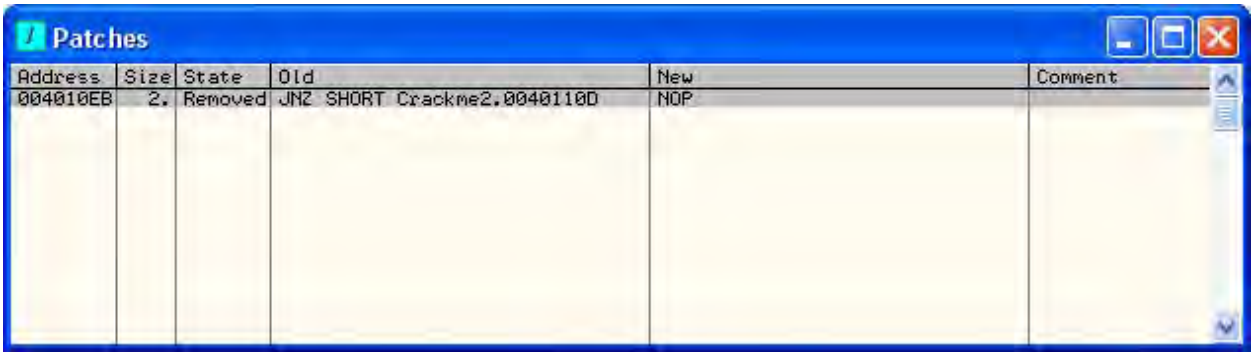
Patch 窗口显示的是我们给程序打的所有补丁。注意地址是红色的，以及 State 列的“Active“。我们的程序仍然在运行，就意味着我们的补丁已经实现，如果 CPU 运行了这个代码，它运行的将是打过补丁的版本。现在，重启应用（Ctrl+F2）。首先，Olly 可能会显示一个错误，一个很长很复杂的错误，基本上是告诉我们补丁（以及断点）没有”坚守“在原来位置，因为 Olly 无法追踪它们（其实比这个要复杂一点，我们后面会看到）。关掉那个窗口，打开断点窗口：



看看我们的断点已经失效了☹。重新激活断点（空格键），Olly 会再次断在该断点。运行程序，输入用户名和密码，我们会停止我们前面打补丁的那一行（它上面的断点又可用了）：

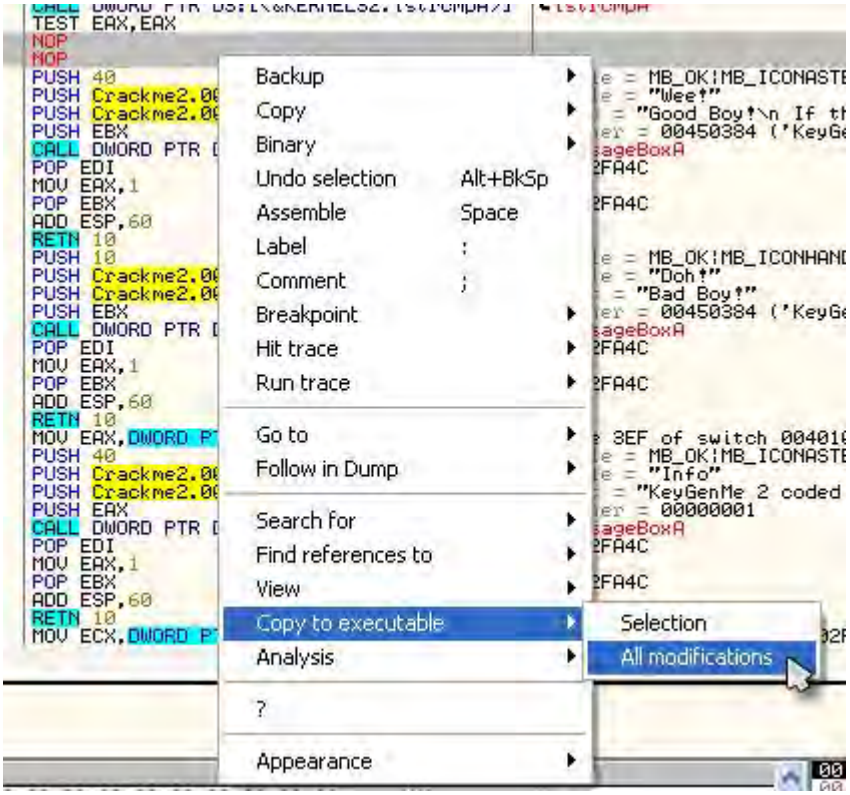


看看，我们的两个 NOP 消失了，原始的代码又回来了（不过变成了灰色）。我们的补丁被回收了！现在回到 Patch 窗口：

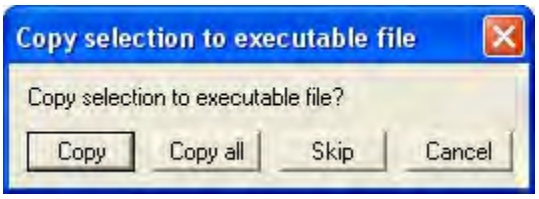


注意那个地址不是红色的了，State 也变成了”Removed“。Olly 已经禁用了我们的补丁，并且在我们每一次重启程序时都会这么做。我们想要做的就是让这个补丁永远有效，而不用每一次都激活它。

为了让我们的补丁能够长久有效，我们必须将修改的版本保存到磁盘。首先，选中补丁再按下空格键以重新启用补丁。那个 JNZ 指令就会变回到我们的 NOP，那两个 NOP 也会以红色字体重新出现在反汇编窗口。现在，在反汇编窗口的任何地方右键，选择”Copy to executable “->” All modifications “:



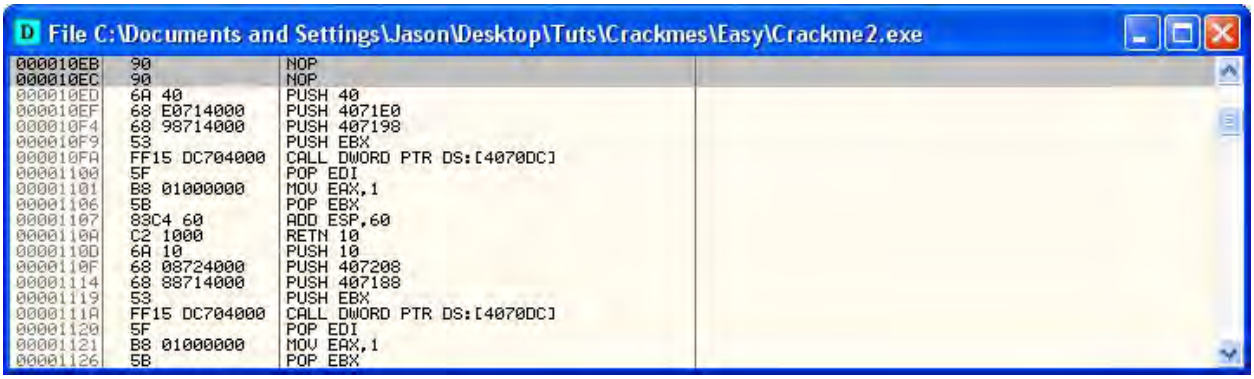
如果弹出窗口问你是否要保存所有的修改，你就选”Copy All “:



当你打了多个补丁，并且想一次性全部保存的话，这么做很重要。因为有时候，你很容易就忘记你打过多个补丁。本例中，即使我们只打了一个补丁，选择所有的补丁也只会保存这一个。当然，只有在 Patch 窗口中被激活的补丁才会被保存。

后面，你可能想要选择”Selection “而不是”All Modifications “，但是你必须保证你在反汇编窗口所做的修改是高亮显示(通过点击或拖拽以选中所有修改的行)。如果你选中的行比修改过的行要多也行，因为 Olly 只会更改已经修改的行。

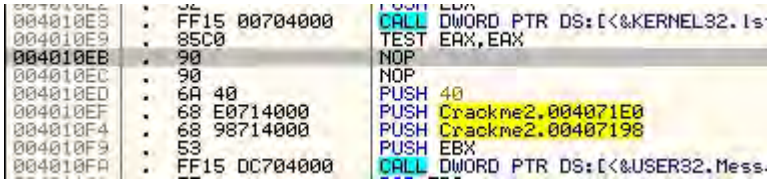
在点击”Copy All “以后会打开一个新窗口，里面基本上是整个进程的数据，不过我们的补丁也在里面:



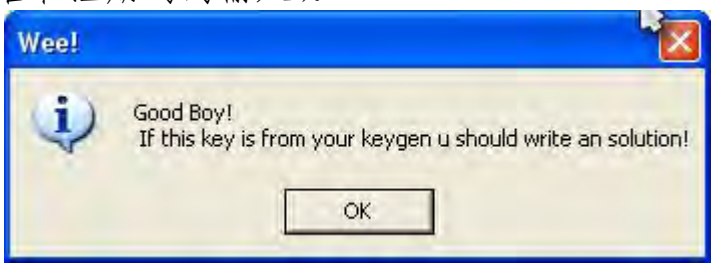
在顶部你可以看到我们的补丁。但要意识到这个只是在**内存**中的修订版本，还没有保存到磁盘呢。不过，如果你关了这个窗口或重启了程序，它是不会被保存的！咱们来保存好它：右键新窗口的任意位置，选择”save file“。这会将该进程的内存空间数据保存到一个文件中。一个另存为对话框会显示出来。将文件另存为 Crackme2_patched（我通常在后门加一个”_patched“用来区分，你也可以加任何你喜欢的）：

我们现在有了一个 crackme 的打补丁版本。咱们来试试看。在 Olly 中打开这个新文件（打过补丁的）。按下 Ctrl+G 或点击 GOTO 图标，输入我们打过补丁的地址：

看看咱们的补丁:



yes，补丁还在那。现在运行程序，输入 info 和 viola (译者注: info 和 viola 是作者用了当用户名和注册码的输入):



现在，我们有了我们第一个破解过并打过补丁的二进制文件: 0。

五、作业

本章的作业很简单（只要你一直在学习汇编语言☺）。

思考题: 你可以将 4010E9 处的”TEST EAX, EAX“修改成什么，来防止跳转到显示 bad boy 处?

要注意的是无论你将 TEST 指令修改成什么,都不能超过 2 字节,那是 TEST EAX, EAX 指令的长度。如果你打了一个长点的补丁，就会覆盖掉 JNZ 指令后面的指令.....

ps. 如果你需要提示的话，请点这里。不过你应该真正意义上自己试着做。那是学习的最好方法!

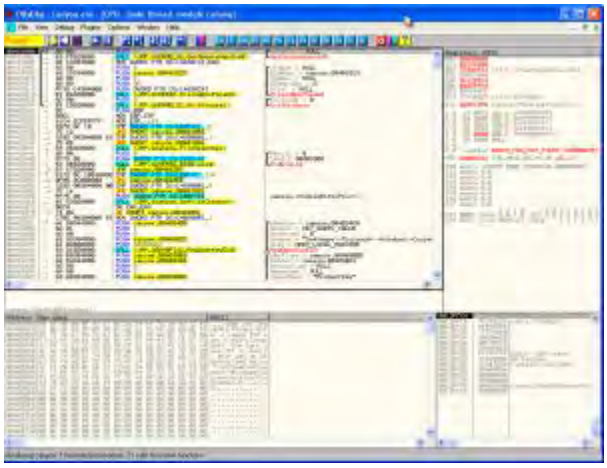
第七章：更多破解练习

一、简介

欢迎来到 R4ndom 逆向工程教程第七章。今天，我们破解两个 crackme: 一个我们用来复习上一章的相关概念，另一个我打算用来做一些有趣的事。在本教程的相关下载中，你可以找到这两个 crackme，以及在第二个程序中要用到的软件“Resource Hacker”。你也可以在工具下载页面下载这些工具。你可以在本教程的教程页面下载相关文件，以及本文 PDF 版本（译者注：英文版）。

二、探究二进制文件

直入主题吧。Olly 载入 canyou.exe（要确保 canyou.dll 在同一目录下）: (p1)



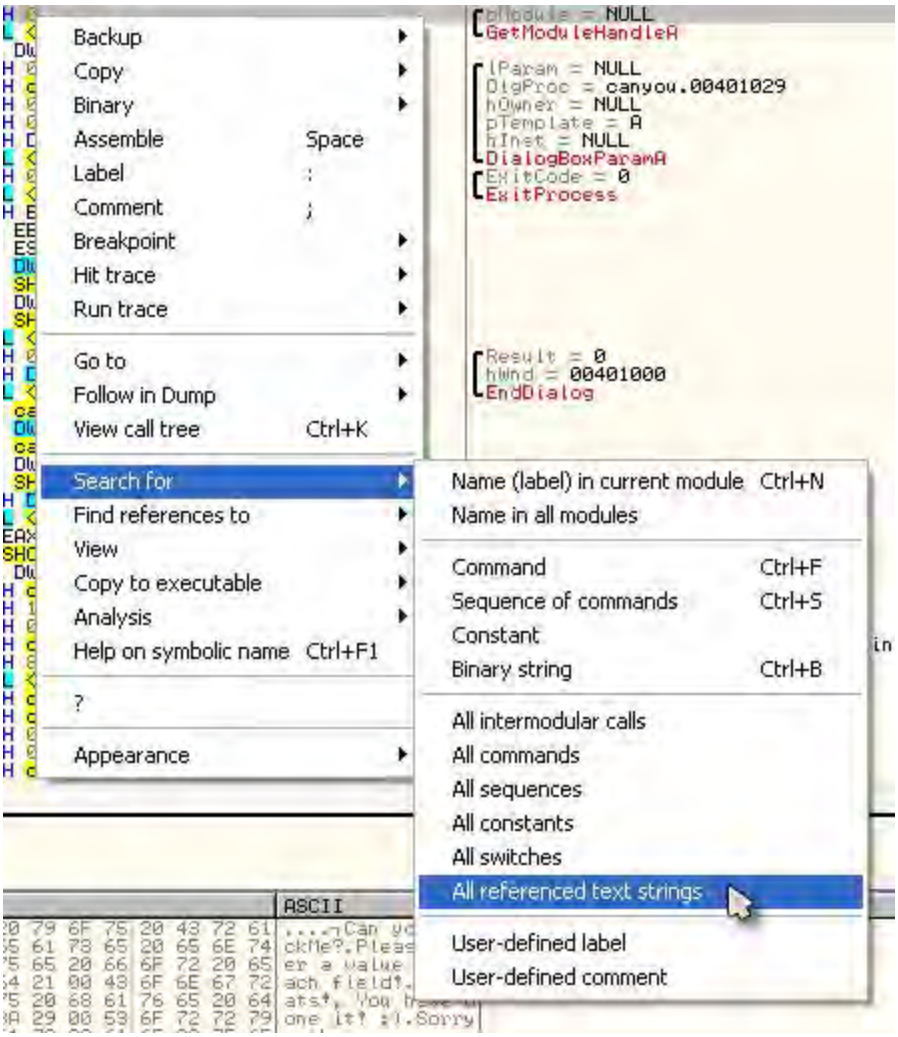
就像我以前说的，在开始之前的最重要的事情是运行程序看看情况。这可以给你大量的信息：有没有试用时间？是不是有些特性被禁用？是不是只能在有限次数内运行？有没有注册窗口让你输入注册码？

这些都是需要知道的很重要的东西，随着你在逆向领域做得越来越好，你会获得越来越多的经验让你知道应该找什么（需要多长时间来验证注册码？是不是强制你访问一个网站？.....）

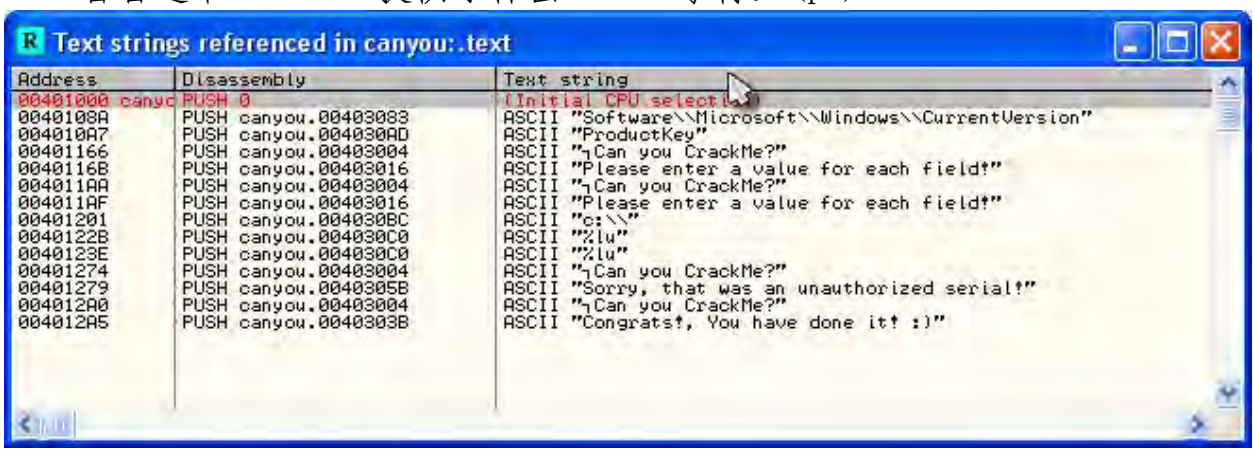
程序运行情况如下：(p2)

输入一些数据之后的结果：(p3)

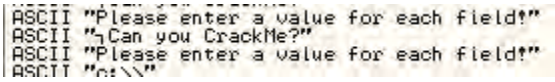
现在你应该知道怎么搞了。回到 011y，看看我们能够查找到哪些字符串：(p4)



看看这个 crackme 提供了什么 ASCII 字符: (p5)



好，这里我们可以很明显的看到几个比较重要的。我们首先注意到的是，我们必须在每个文本框中都要输入信息: (p6)



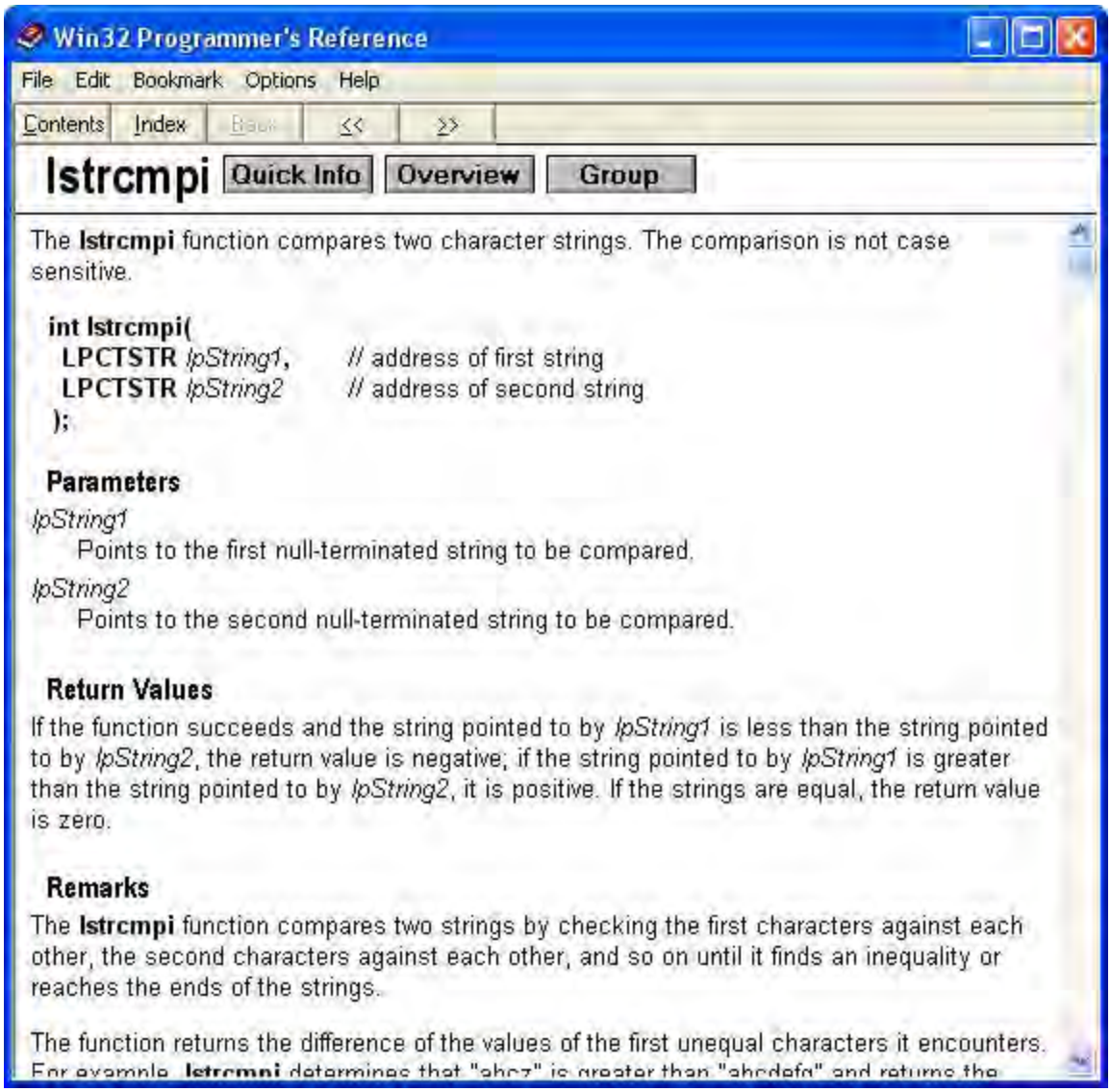
然后，我们才是真正重要的东西：(p7)

```
ASCII "Sorry, that was an unauthorized serial!"
ASCII "¬Can you CrackMe?"
ASCII "Congrats!, You have done it! ;)"
```

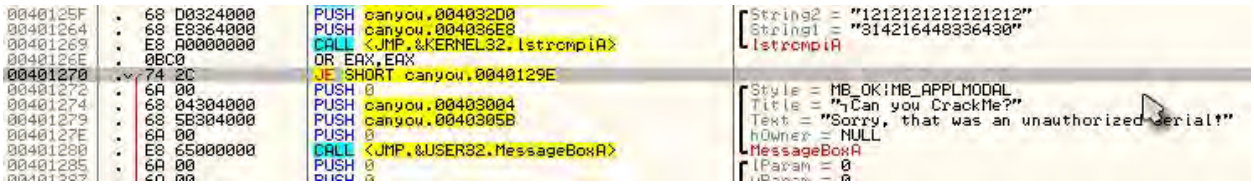
双击其中一个，看看我们会去哪：(p8)

```
00401250  .  68 E8384000     PUSH canyou.004038E8                    ; StringToAdd = ""
00401255  .  68 E8364000     PUSH canyou.004036E8                    ; ConcatString = ""
0040125A  .  E8 A9000000     CALL <JMP.&KERNEL32.lstrcatA>           ; lstrcatA
0040125F  .  68 D0324000     PUSH canyou.004032D0                    ; String2 = ""
00401264  .  68 E8364000     PUSH canyou.004036E8                    ; String1 = ""
00401269  .  E8 A0000000     CALL <JMP.&KERNEL32.lstrcmpiA>          ; lstrcmpiA
0040126E  .  0BC0            OR EAX,EAX
00401270  .v 74 2C           JE SHORT canyou.0040129E
00401272  .  6A 00           PUSH 0                                  ; Style = MB_OK|MB_APPLMODAL
00401274  .  68 04304000     PUSH canyou.00403004                    ; Title = "¬Can you CrackMe?"
00401279  .  68 5B304000     PUSH canyou.0040305B                    ; Text = "Sorry, that was an unauthorized serial!"
0040127E  .  6A 00           PUSH 0                                  ; hOwner = NULL
00401280  .  E8 65000000     CALL <JMP.&USER32.MessageBoxA>          ; MessageBoxA
00401285  .  6A 00           PUSH 0                                  ; lParam = 0
00401287  .  6A 00           PUSH 0                                  ; wParam = 0
00401289  .  6A 10           PUSH 10                                 ; Message = WM_CLOSE
0040128B  .  FF75 08         PUSH DWORD PTR SS:[EBP+8]               ; hWnd = 401000
0040128E  .  E8 5D000000     CALL <JMP.&USER32.SendMessageA>         ; SendMessageA
00401293  .  B8 00000000     MOV EAX,0
00401298  .  C9              LEAVE
00401299  .  C2 1000         RETN 10
0040129C  .v EB 13           JMP SHORT canyou.004012B1
0040129E  >  6A 00           PUSH 0                                  ; Style = MB_OK|MB_APPLMODAL
004012A0  .  68 04304000     PUSH canyou.00403004                    ; Title = "¬Can you CrackMe?"
004012A5  .  68 3B304000     PUSH canyou.0040303B                    ; Text = "Congrats!, You have done it! ;)"
004012AA  .  6A 00           PUSH 0                                  ; hOwner = NULL
004012AC  .  E8 39000000     CALL <JMP.&USER32.MessageBoxA>          ; MessageBoxA
004012B1  >v EB 09           JMP SHORT canyou.004012BC
004012B3  >  B8 00000000     MOV EAX,0
004012B8  .  C9              LEAVE
```

这个看起来比较熟悉吧：有坏消息部分，紧跟其后的是好消息部分，并且在坏消息的前面有一个非常明显的跳转，想必是跳转到好消息的。这里我想让你注意的是，在跳转的前面有一个对 Windows API 函数 *lstrcmpi* 的 CALL。如果我们在其上右键，选择 “Help on symbolic name”，会有如下显示：(p9)



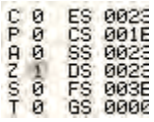
如你所见，lstrcmp 是对两个字符串进行比较操作。这个函数在逆向工程领域非常的重要，你会一次又一次的看到。它被用于 注册码/密码 比较机制中，用于比较用户输入的字符串与程序内置的硬编码或被创建的字符串。如果字符串比较返回 0，说明用户的输入是正确的，意味着比较的两个字符串是一样的。如果返回非 0，说明两个字符串不匹配。本例中的 crackme，我们输入的字符串可能与一个内置的或动态生成的字符串进行核对，如果 EAX 返回的是 0，说明它们是相同的，否则就不相同。现在，Olly 不知道这些字符串是什么，因为我们还没有启动应用，也没有输入任何信息。不过一旦我们开始了，Olly 就会将 String1=""、String2="" 这两行替换成真正的字符串。如果我们在跳转那里设置一个 BP，然后运行程序，输入一个字符串（本例中是“12121212121212121212”），Olly 就会给我们显示被比较的字符串：(p10)



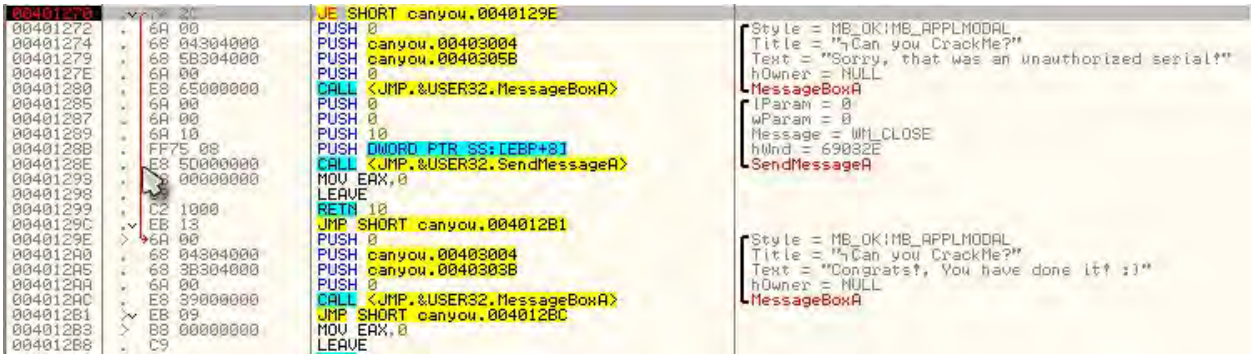
如果你看跳转指令上面的那几行，你会看到我们的密码与“314216448336430”进行比较，无论它是什么都一样。在返回值上，如果它们相同 EAX 中就是 0，如果不相同就可能是任意值。很明显，本例中，它们不匹配。OR EAX, EAX 是一个判断 EAX 是否为 0 的非常巧妙的方法。如果 EAX 是 0 的话，“JE SHORT canyou.0040129E”就会跳到好消息部分。我之所以给你指出字符串比较部分，是因为在将来的教程中，我们需要找出这 15 个数字是如何被创建出来的。搜索 lstrcmp 能够引导我们找到它的创建过程。

不过现在，我们只做我们知道的。在 401270 处的 JE 指令处设置一个 BP，然后重启程序。输入一个用户名和序列号，O11y 会断在我们的 BP：(p11)

通过那个灰色的箭头，我们知道 O11y 不会跳到好消息部分，而是落在坏消息部分。所以咱们来帮帮它吧：(p12)



现在，O11y 做对了：(p13)

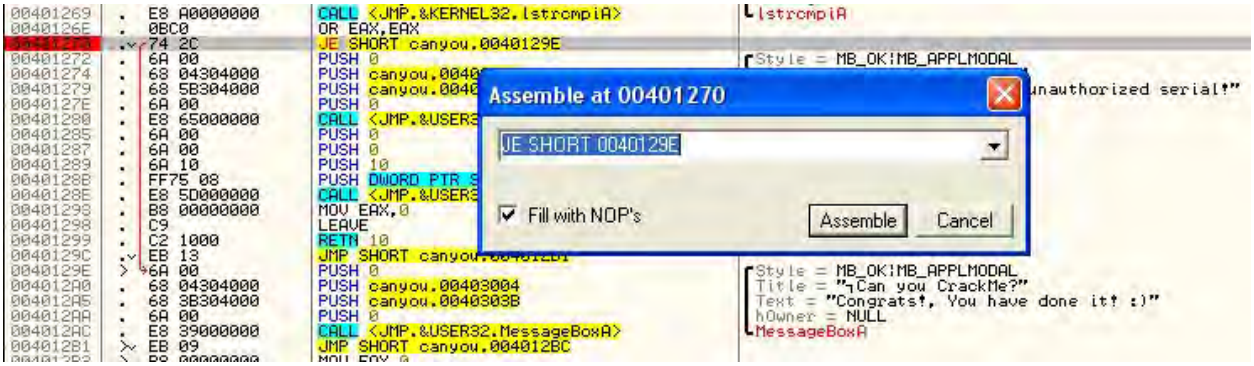


为了确定下，咱们运行程序看看：(p14)

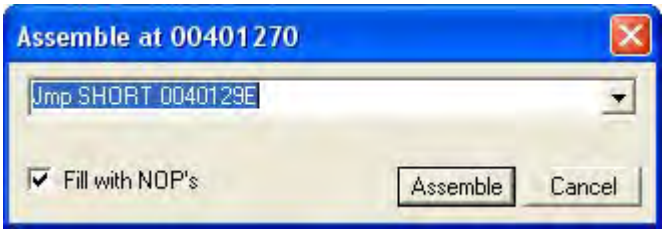
现在，让我们.....

三、给程序打补丁

这回我不打算将跳转 NOP 掉，因为这样会让程序每一次都显示坏消息。相反，我想要确保跳转每一次都成功，跳转到我们好消息部分。转到设置 BP 的那行（如果你找不到的话，打开 “Breakpoint Window”，然后在 BP 上双击），修改那行指令。选中 JE 指令那行，然后按一下空格键：(p15)



注意我们选中的指令已经在文本框中了。现在，我们将 JE (Jump on Equal) 修改成 JMP（无条件跳转）：(p16)

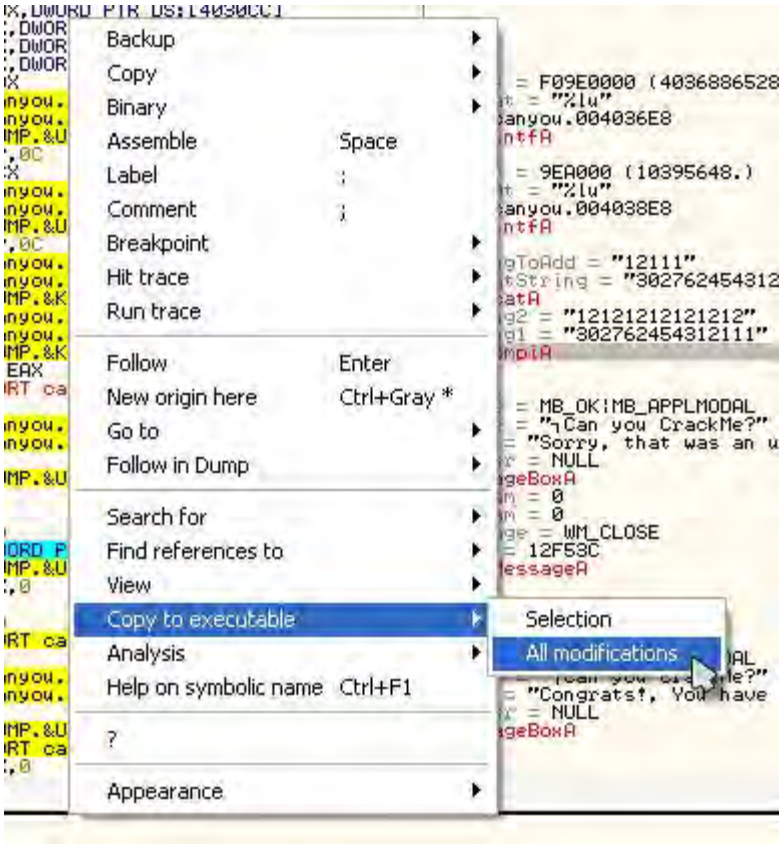


点击那个 Assemble 按钮，然后点 Cancel 按钮。你就会发现我们的修改已经放到了代码中：(p17)

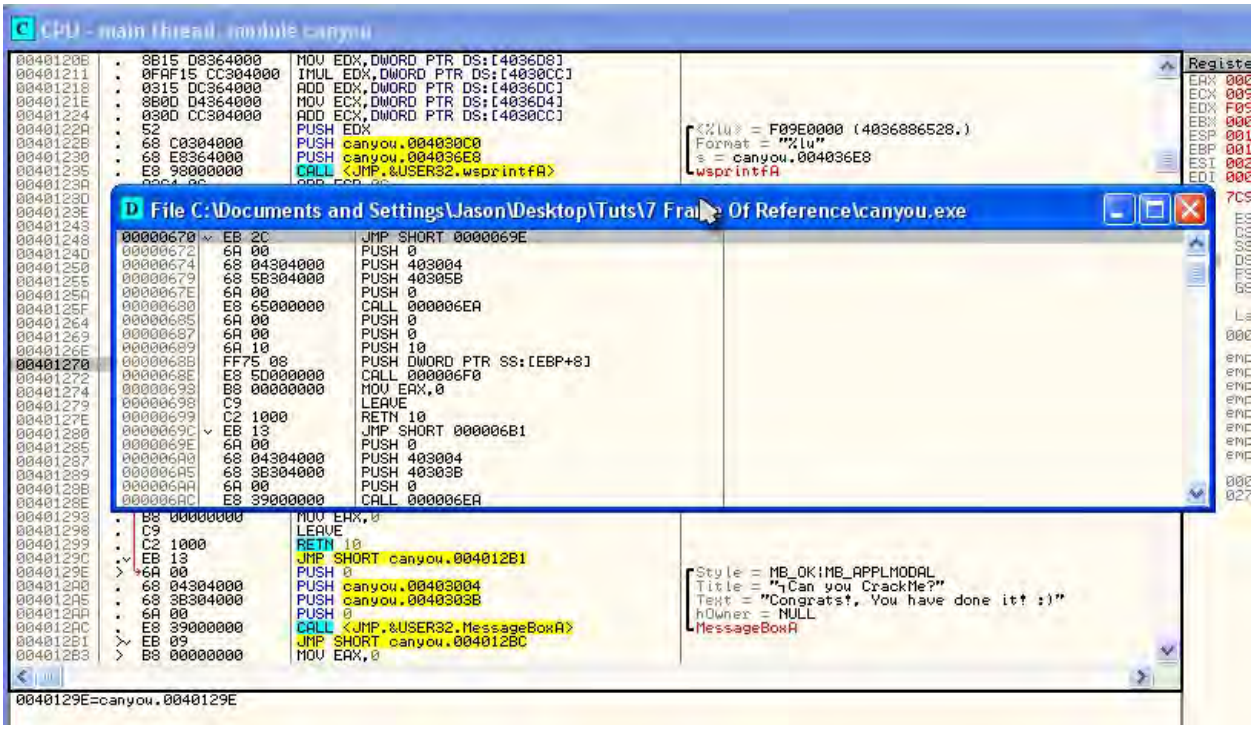
```
00401269   .  E8 A0000000     CALL <JMP.&KERNEL32.lstrcmpiA>
0040126E   .  0BC0           OR EAX,EAX
00401270   .  EB 2C           JMP SHORT canyou.0040129E
00401272   .  6A 00           PUSH 0
00401274   .  68 04304000     PUSH canyou.00403004
00401279   .  68 5B304000     PUSH canyou.0040305B
```

现在，运行下程序以确保没什么问题：(p18)

现在，咱们将打过补丁的程序保存到磁盘。要记住，如果你重启应用的话，你需要重新启用补丁（Patch 窗口中，选中补丁再按一下空格键），不过我们的程序还在运行，只需要点一下 Olly，右键反汇编窗口，选择“Copy to executable” -> “All modifications”：(p19)



选择”Save all “，弹出进程内存窗口（顶部就是我们的补丁）：(p20)

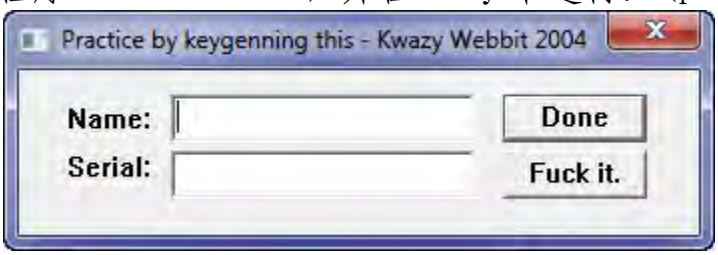


现在，咱们把它保存到磁盘....在新弹出窗口中右键，选择”Save File “。另存为 canyou_patched（或任何你喜欢的名字），将打过补丁的文件载入

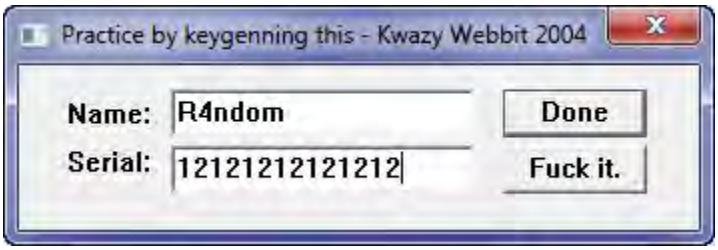
Olly 并允许。如果你不想这么做的话，事实上你再也不用将其载入 Olly 了。因为补丁已经被保存到磁盘，你可以从任何地方运行它。你要你运行的是打过补丁的就行。现在，无论你输入什么名字和序列号，都会弹出好消息窗口。

四、另一个 crackme

载入第二个程序 Crackme8.exe，并在 Olly 中运行：(p21)



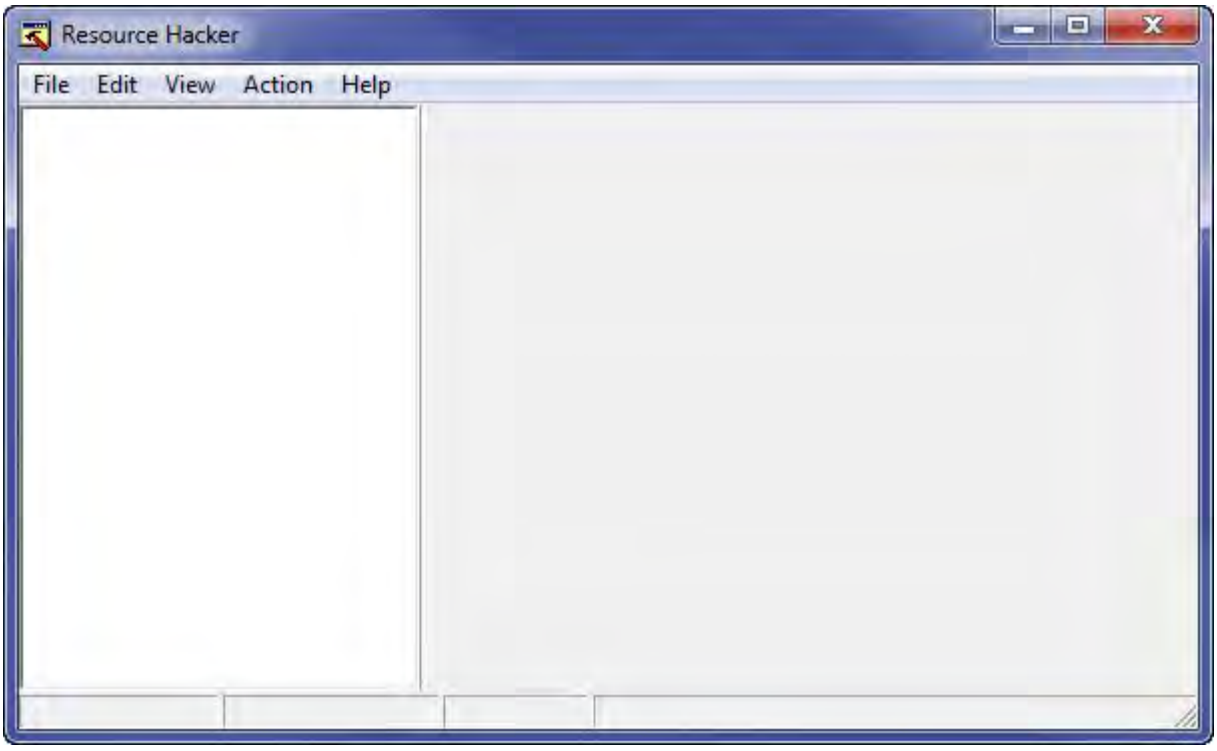
好吧，这里有点点疑惑：0。嗯，在输入了用户名和密码后，我该点哪个按钮呢？好吧，试试吧：(p22)



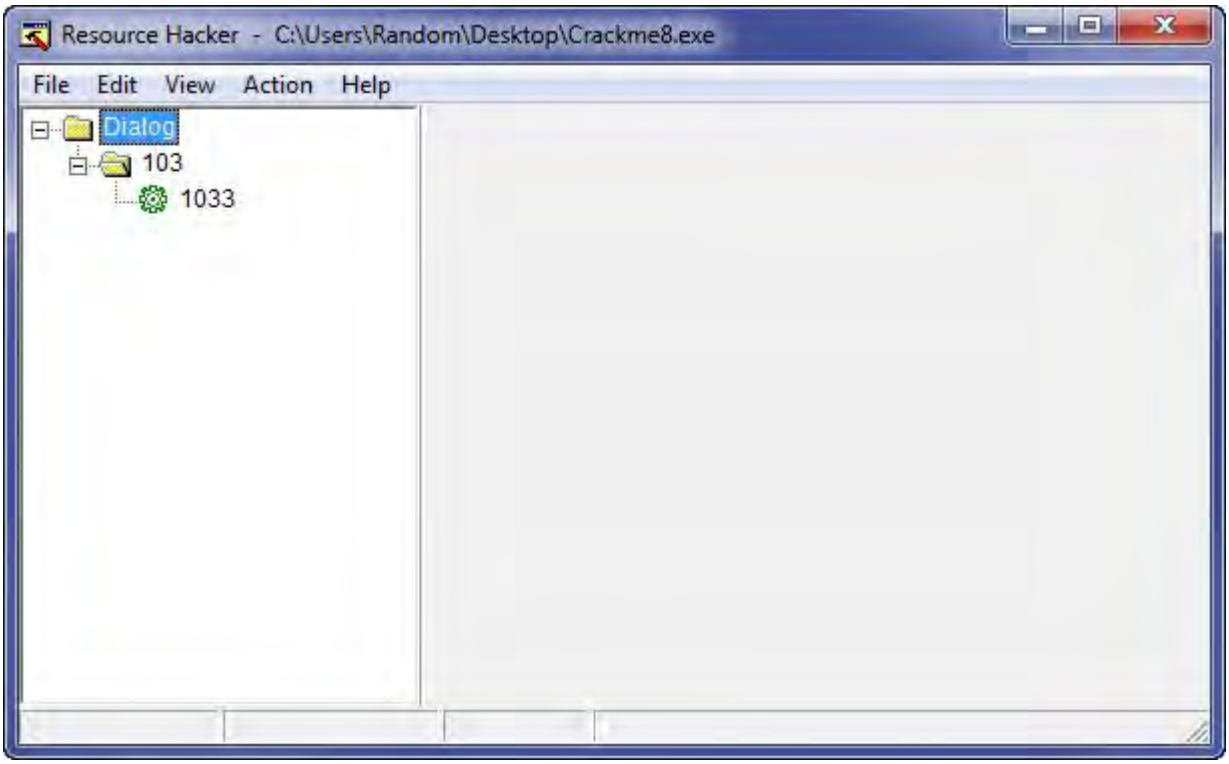
这里，Done 通常意味着退出，所以我试试另一个。嗯.....，程序退出了。很明显我应该点 Done 的（？）。不管了，借此机会咱们改改程序，做些有趣的事。咱们将按钮改成更加有意义的“Check”和“Done”，或者是任何你喜欢的都行。

五、使用 Resource Hacker

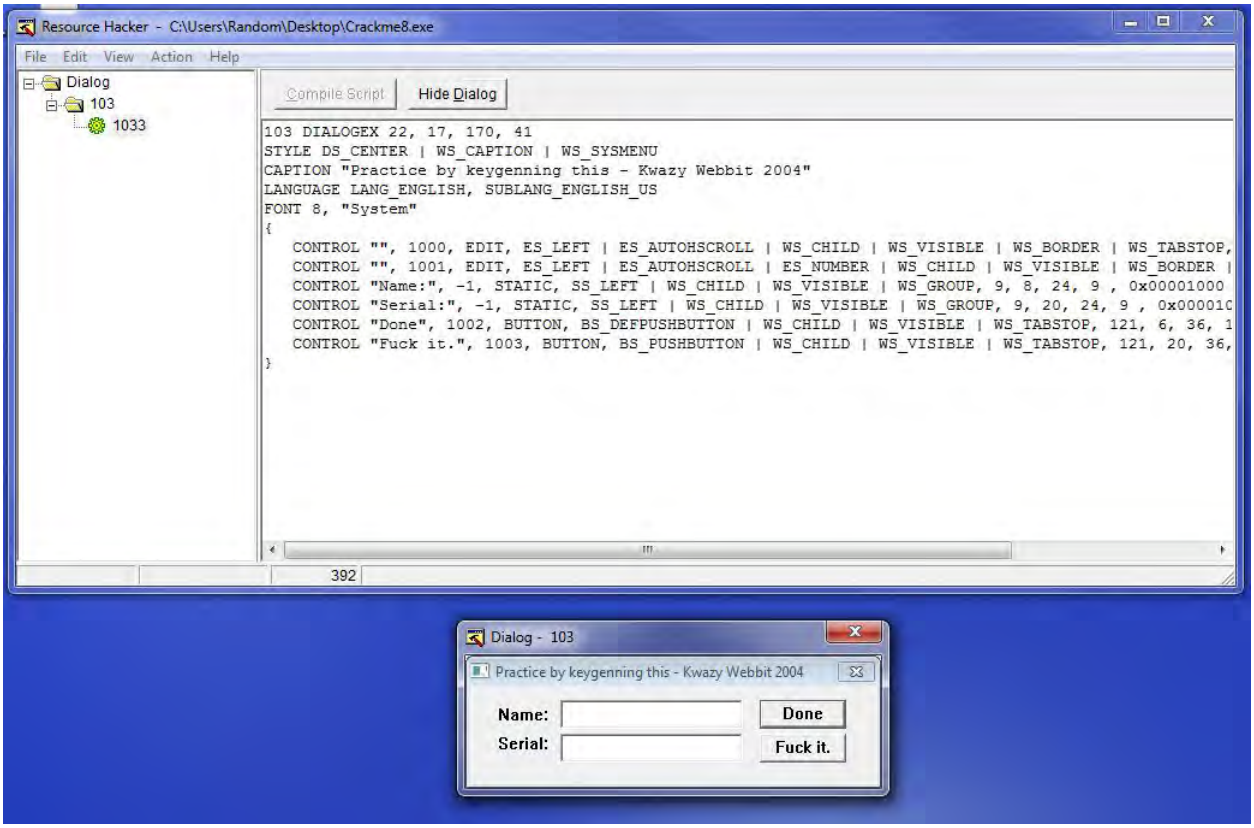
如果你还没准备好，先安装 Resource Hacker。第一次运行如下所示：(p23)



将 Crackme8 载入到 Resource Hacker，你就会看到一个叫 Dialog 的文件夹，它旁边有个+号。展开+号，点一下下一个文件夹（103）边上的+号，你会看到如下所示的内容：(p24)



现在，点那个 1033，然后右边面板就会显示对话框的相关数据，同时会有一个窗口显示它（译者注：就是 crackme8 的窗口样式）的样子：(p25)



在右侧面板的顶部，你可以看到一些关于窗口的数据，比如字体、标题、类型等等：(p26)

```
103 DIALOGEX 22, 17, 170, 41
STYLE DS_CENTER | WS_CAPTION | WS_SYSMENU
CAPTION "Practice by keygenning this - Kwazy Webbit 2004"
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
FONT 8, "System"
```

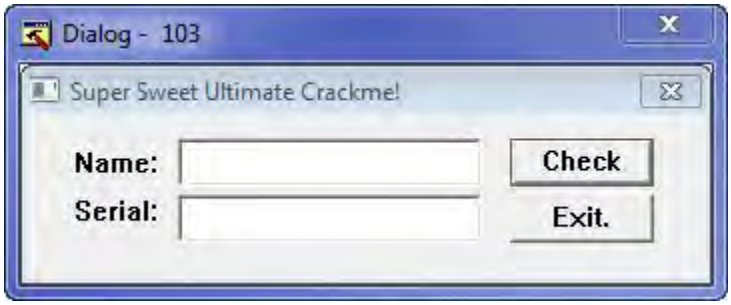
在下面你可以看到对话框中所有元素的细节，包括“Name”、“Serial”标签和两个按钮。咱们把这个对话框修改成我们喜欢的，好不好？首先将两个按钮的名字修改成“Check”和“Exit”：(p27)

```
CONTROL "Serial:", -1, STATIC, SS_
CONTROL "Check", 1002, BUTTON, BS_
CONTROL "Exit.", 1003, BUTTON, BS_
```

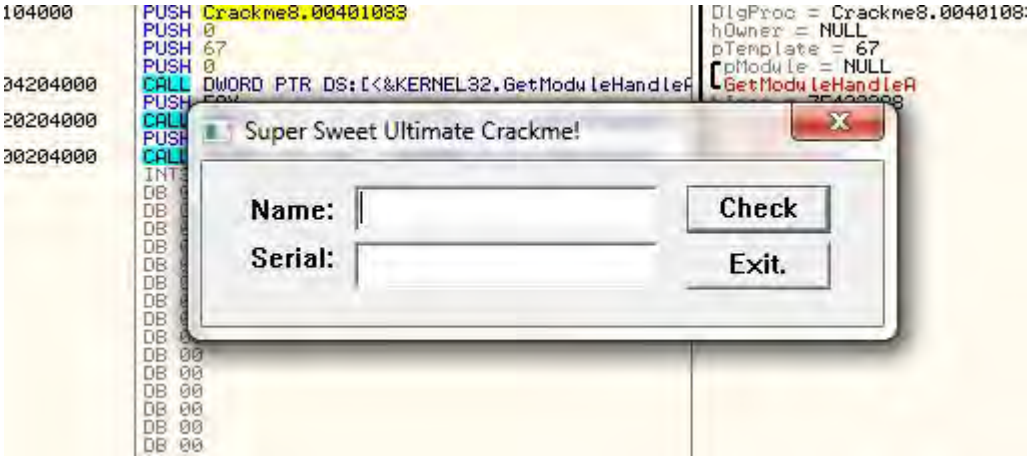
现在，我们修改顶部的标题：(p28)

```
103 DIALOGEX 22, 17, 170, 41
STYLE DS_CENTER | WS_CAPTION | WS_SYSMENU
CAPTION "Super Sweet Ultimate Crackme!"
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
FONT 8, "System"
{
   CONTROL "", 1000, EDIT, ES_LEFT | ES_AU
```

现在点击“Compile”按钮，就会看到我们的窗口更新了：(p29)



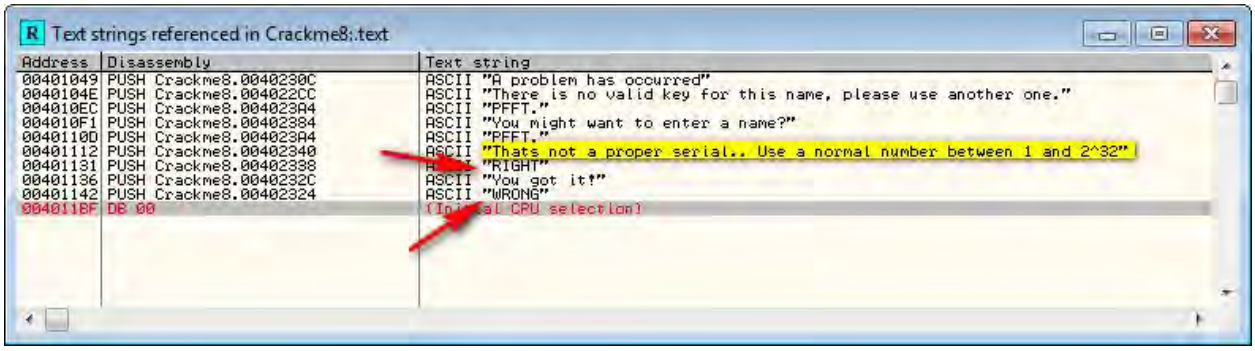
好，相当不错。将其保存（“File” -> “Save”），将新的 crackme 载入 Olly（原始的 crackme 被 Resource Hacker 以 Crackme8_original 名字保存），运行它：(p30)



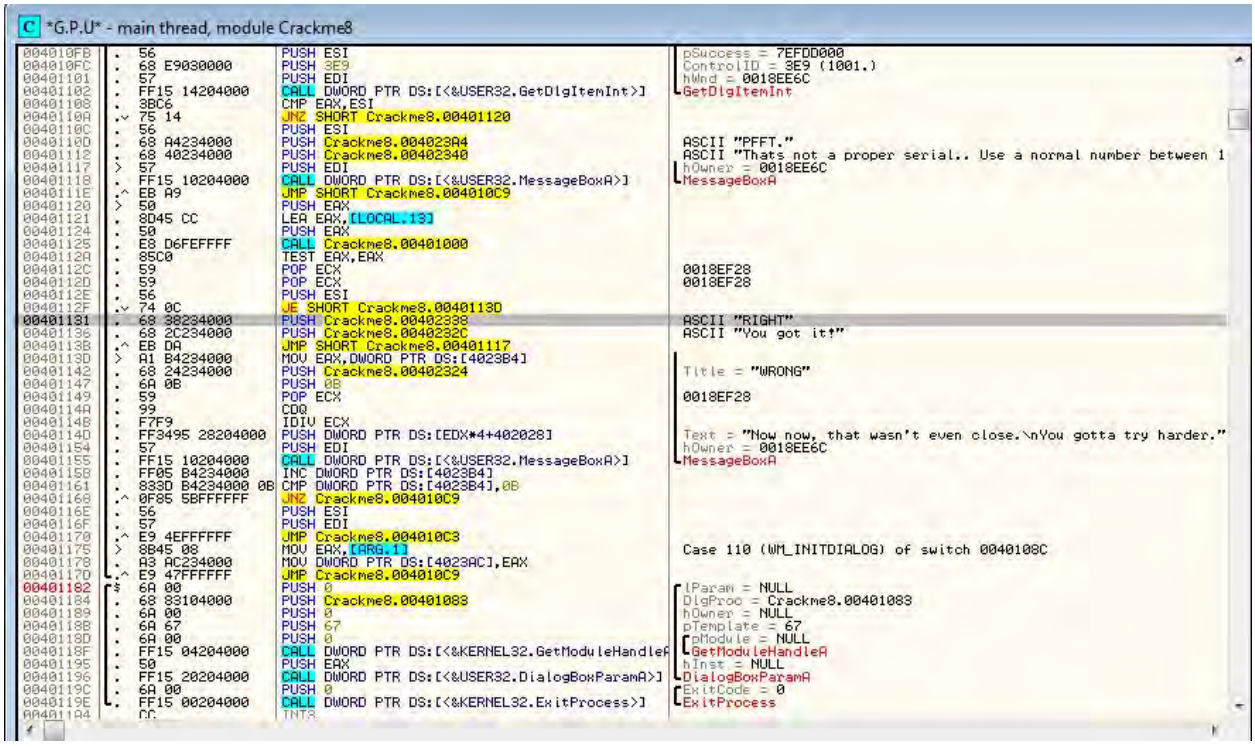
啊，相当好。现在我们正式开始...

六、破解程序

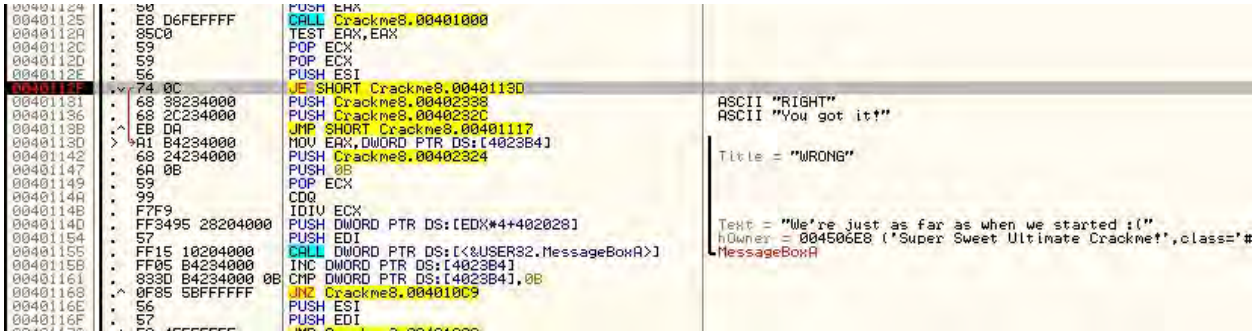
现在你应该知道怎么做了。搜索文本字符串：(p31)



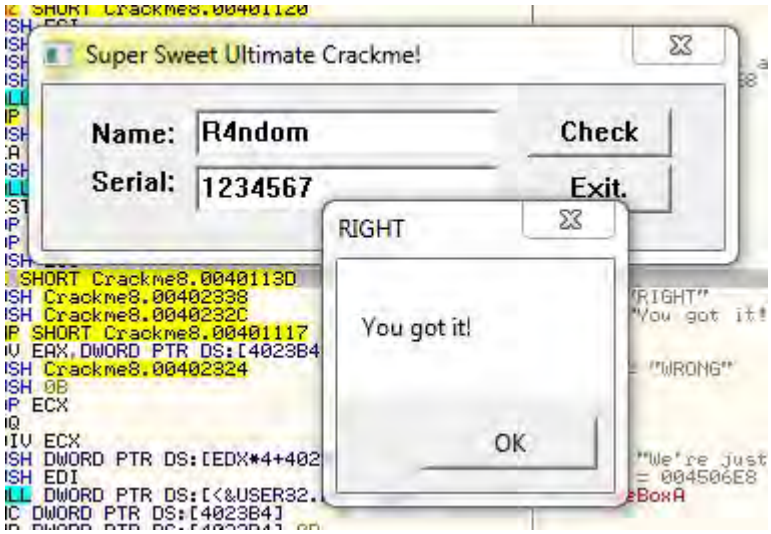
我们了解了两件事：1）序列号必须是一个 1 到非常大的数字；2）我们知道了好消息和坏消息生成的地方。咱们转到好消息那：（p32）



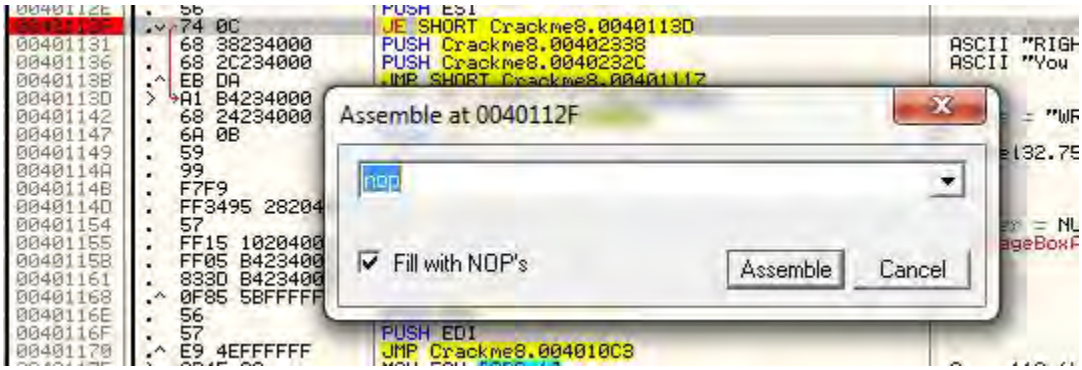
双击进到相关领域。我们看到好消息的路径是从 401131 开始的，坏消息从 40113D 开始。我们看到那个跳转指令（JE SHORT Crackme8.0040113D）在 401131 处，比较指令（TEST EAX, EAX）在 40112A 处。咱们在 40112F 处设置 BP，然后运行程序。输入用户名和序列号后点击 “Check”。O11y 随后断在了我们的断点处：（p33）



我们可以看到，O11y 依旧要跳过好消息部分，直达坏消息部分。你知道了那个路径...，清除 0 标志位运行程序：（p34）



成功了！现在咱们快速的创建一个补丁。重启应用，找到断点（通过断点窗口），在 JE 指令上点一下，再按一下空格键，NOP 掉跳转指令，这样我们就能够一直的直达好消息部分：（p35）



先点“Assemble”然后是“Cancel”。右键然后选择“Save to executable” -> “All modifications”，再选择“Copy all”。右键弹出的窗口，选择“Save file”保存它。现在你有了一个打过补丁的并且修改过资源的 crackme，你输入的任何序列号都会让他显示好消息😀。

七、值得思考的事

我想说的是，Resource Hacker 是一个有意思的非常有用的小程序。通过它你不仅仅可以修改一个文件的许多东西（字符串、图标、标签、按钮、标题），你也可以用它修改 Windows 自身的许多东西（开始按钮、上下文菜单、计算机的“关于”对话框等）。事实上，Resource Hacker 正是我的版本的 Olly 的图标修改工具！

第八章：参考引用框架

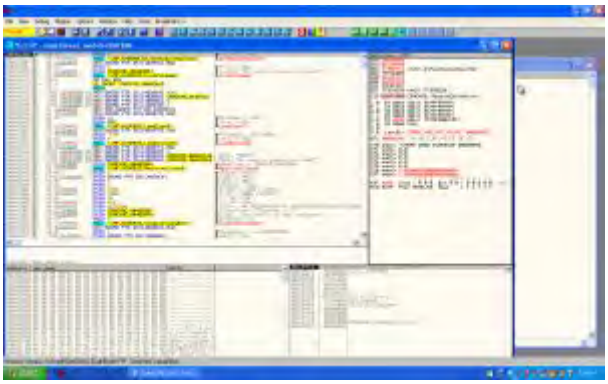
一、简介

我们现在要研究的 crackme，相比来说更具挑战性。它就是 Crackme3.exe。咱们也会学习几个新技巧。

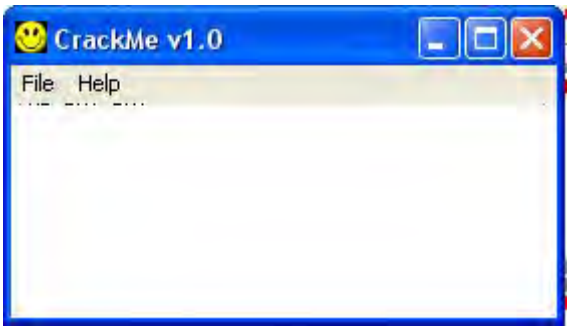
你可以在教程页下载相关文件以及本文的 PDF 格式版本。

二、探究二进制文件

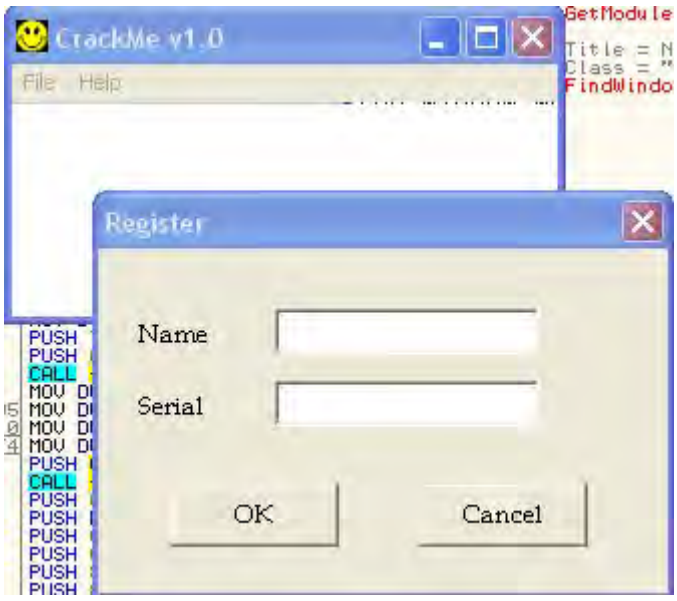
启动 Olly 并载入 crackme。它会载入、分析并暂停在第一行：（p1）



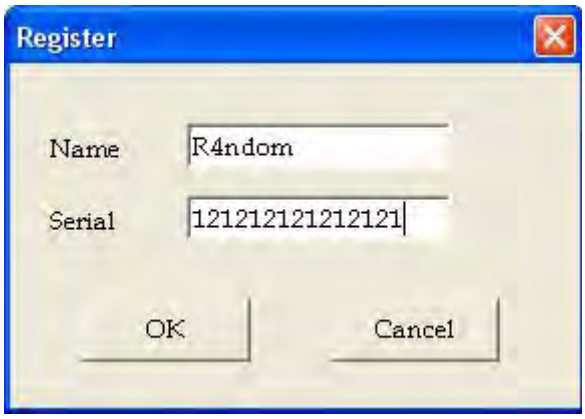
运行下程序看看什么样：（p2）



好吧，没啥东西。选择 “Help” ->” Register”：（p3）

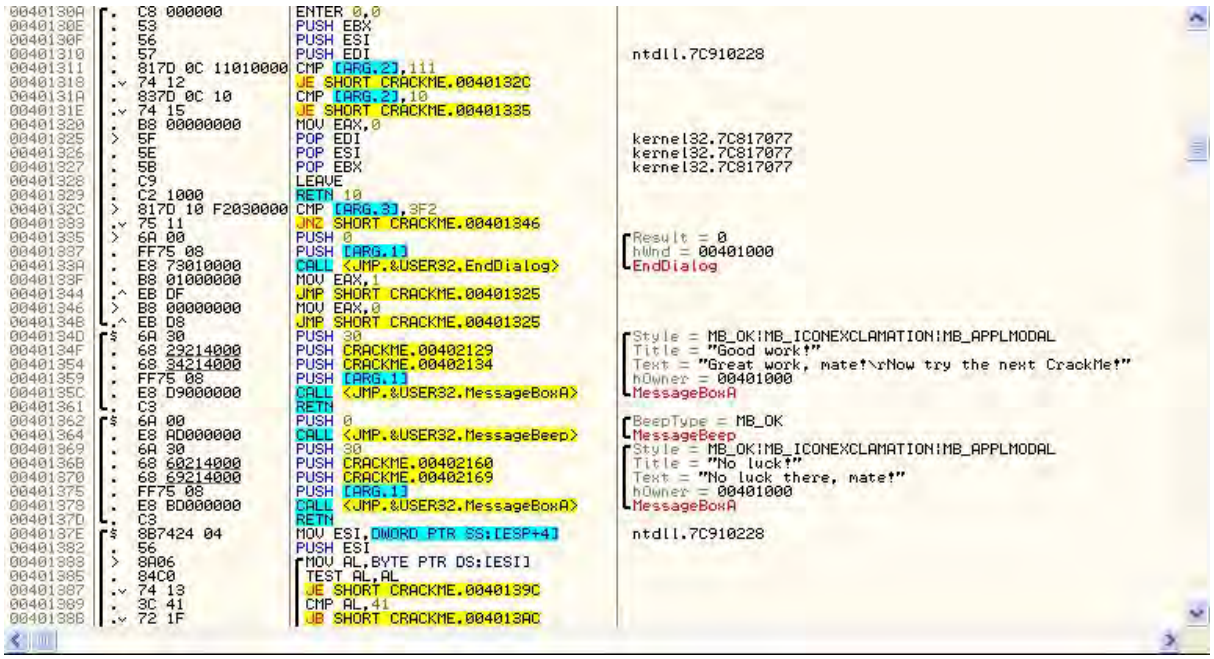


现在咱们来到了某个地方。奇怪了，怎么和我们的 FAKE 那么像。试着输入用户名和序列号看看程序有什么反应：(p4)

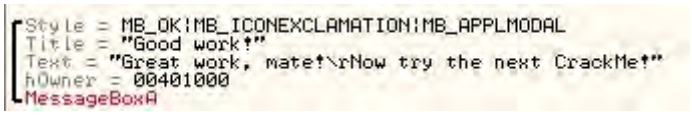


嗯。这回弹出了一个显示坏消息的对话框：(p5)

有时候，对于一个比较小的程序，我喜欢向下多翻几页看看有没有什么有意思的东西。我向下翻了大概 6 页，然后我看到了一些相当有趣的东西：(p6)

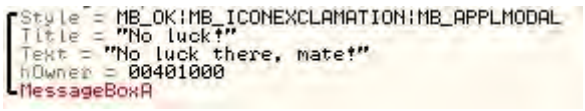


看看在 MessageBoxA 函数前面的文本。如果你往 MessageBoxA 函数上面的文本的左边看的话，你会看到一条黑线将函数的参数框起来：(p7)



Olly 给你显示的是准备传递给函数的参数，就是被调用的那个函数的。本例中个，参数 1 是窗口的类型，参数 2 是窗口的标题（“Good work!”），参数 3 是窗口显示的文本（“Great work...”），参数 4 是窗口所有者的句柄。最后，MessageBoxA 函数被调用。你可以在 MessageBoxA 上右键，选择 “Help on symbolic names” 来查看传递给函数的参数以及返回值。

现在，我们对比着看下紧随其后的那部分：(p8)



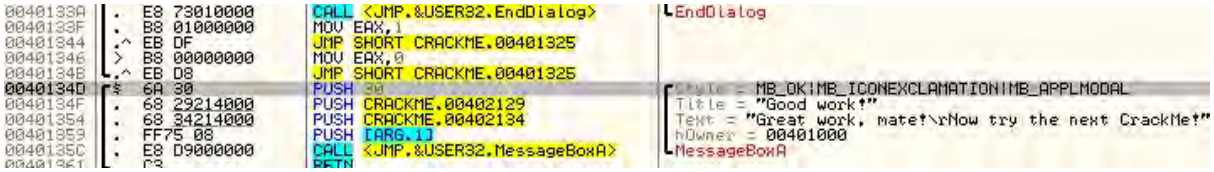
对这两个函数的调用有很大的不同。一个看起来真的不错，而另一个却不是。我想我们大家都承认，我们宁愿要第一个调用。现在咱们要记住

R4ndom’s Essential Truths About Reversing Data #2:
R4ndom 关于逆向数据的必备真言 2：

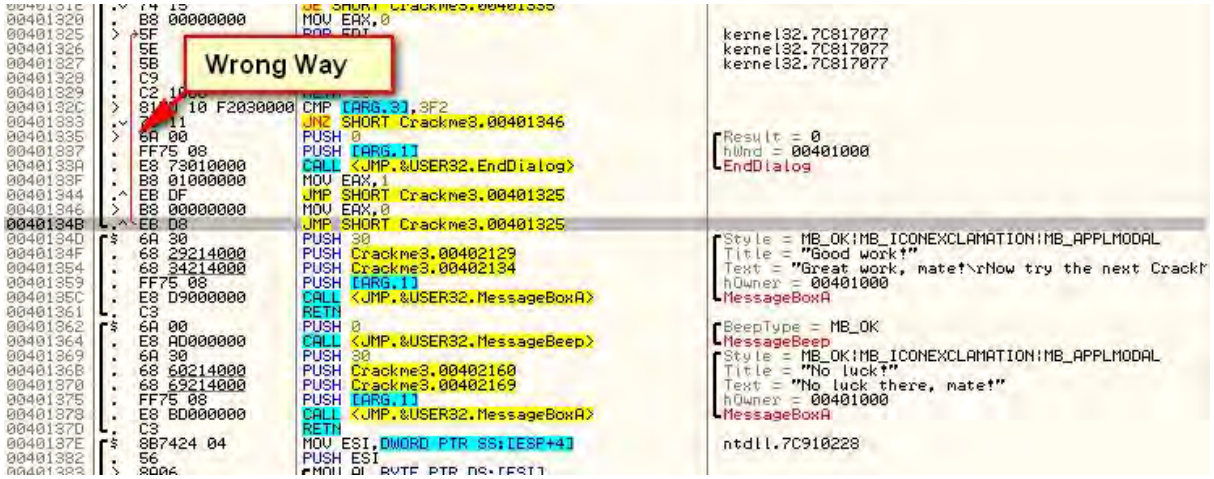
2. 大部分的保护机制都可以被绕过，通过修改一个简单的跳转指令来跳转到“好的”代码处，而不是“坏的”代码处（或者避免跳转跳过“好的”代码）。

如果你看两个函数的上面几行，你会看到几个 jmp 语句，它们决定了你将走哪条路，好的路或者坏的路。99%的应用里的 99%的时间都是这样。窍门就是

找到这个跳转。(当然剩下的 1%要难得多，不过我们不会接触。)我们的例子中，在 401344 和 40134B 有跳转。现在，作为一个已经训练过的逆向工程师，这些跳转很快就被略过(如果你想知道为啥，是因为它们和我们的消息框在不同的函数中，所以它们不会跳过我们的坏消息或跳转至好消息处，后面会讨论这个)。咱们来研究研究它们: (p9)

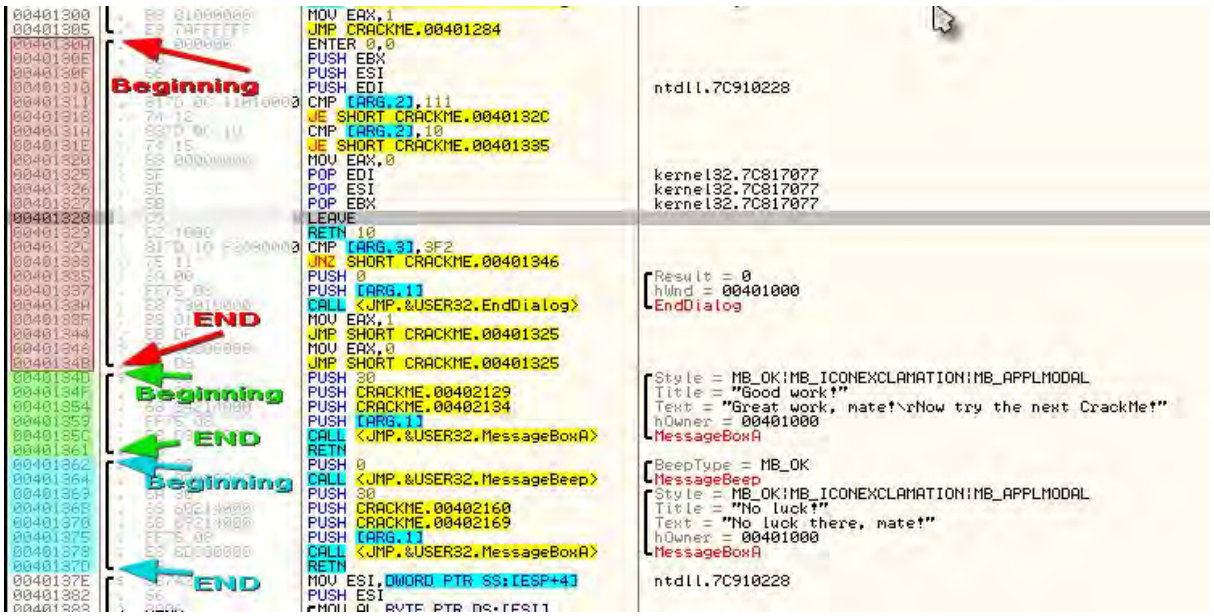


首先，点一下 40134B 处的 JMP 指令。会看到一个红线指示该 JMP 将跳到哪，我们看到它走的是一条错误的路！(p10)



它没有跳到我们的好消息那，也没有跳过坏消息，反而往上跳到前面的代码了。我们试试 401344 那个 JMP。这个事实上和那个指向的一样(仍然是错误的路)，所以看起来我们的第一个猜测是错的。

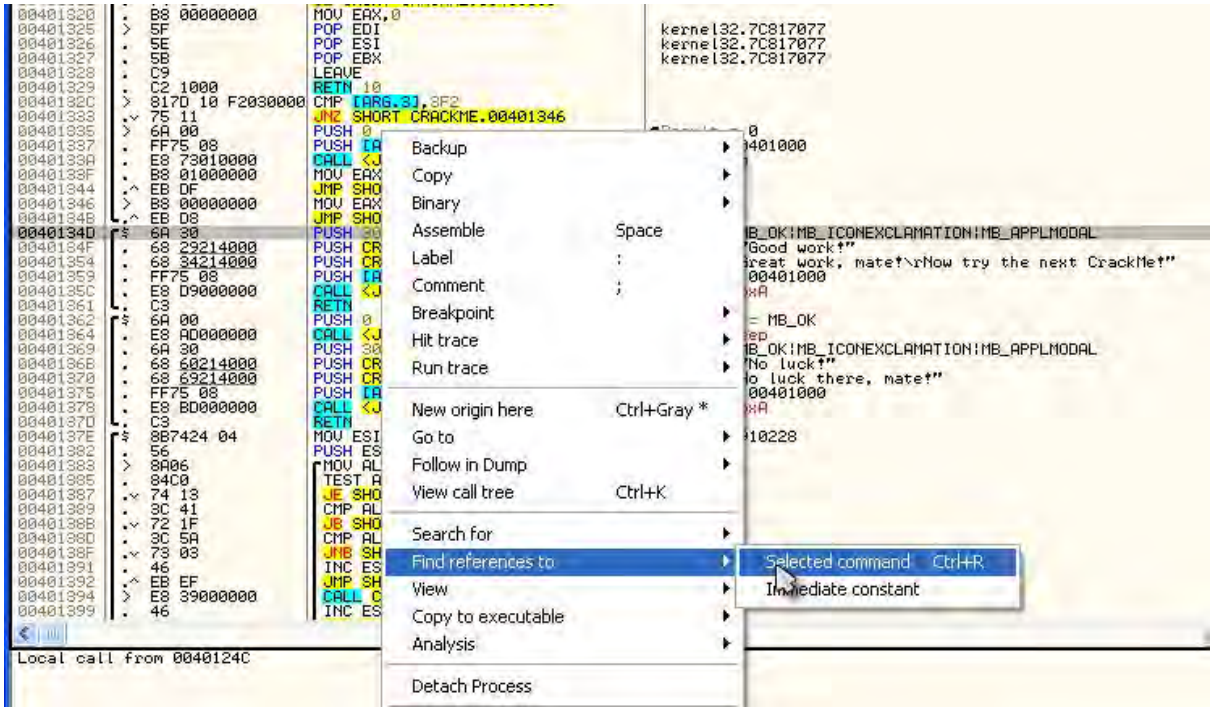
顺便说一下，就像我早些说的，老鸟忽视这些跳转的原因是因为 OllyDbg 显示函数的方式。如果你注意看第一列(地址)和第二列(操作码)之间的话，会看到一些粗黑线。这些线是 OllyDbg 放进去的，用来区分函数(有时候 OllyDbg 无法指出函数的起始点和结束点，所以就不会有这些线): (p11)



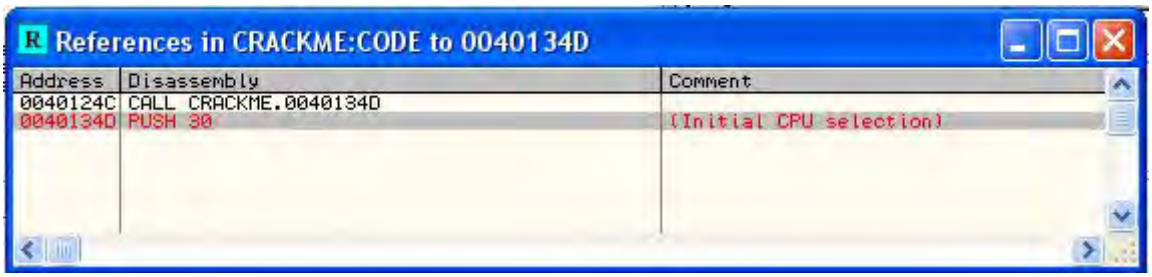
本例中，你可以看到那两个 JMP 是在我们的好消息和坏消息的上面的函数中。因为它不会跳转到好消息或者坏消息处，它们真的对我们没有任何帮助。这也告诉你另一件事，第一个消息框（好消息那个）和坏消息框不是在同一个函数中。这些都告诉我们，这些函数都是在别的地方被调用的，并且在它们被调用之前的某处决定了哪个被调用，是好的还是坏的。咱们看看怎么才能绕过这些干扰.....

三、查找参考

在好消息函数的第一行，也就是 40134D 那行上右键。选择 "Find References To" ->" Selected Command"（或者按 Ctrl+R）：（p12）



弹出“References(参考)”窗口：(p13)



该窗口显示的是 Olly 能够找到的 CALL 或 JMP 到*这个*地址的所有参考（CALL 和 JMP）。现在，双击列表中的第一个（就是那个不是红色的），然后你就会来到调用这个（好的）消息的那行：(p14)

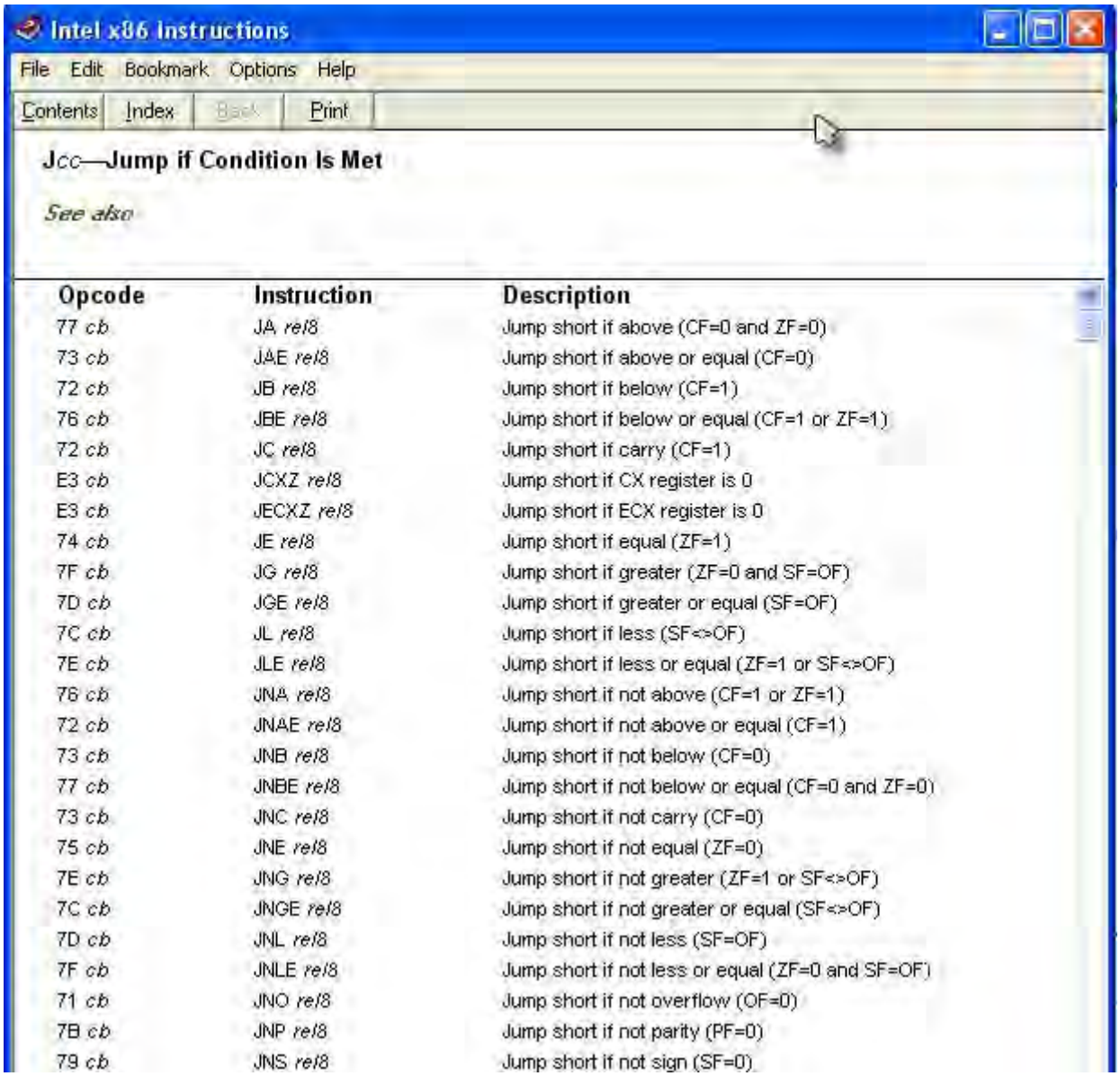
在 40124C 那行你可以看到指令 CALL CRACKME.0040134D。40134D 就是好消息对话框的第一行。咱们在这里设置一个断点：(p15)

现在，咱们对另一个函数做相同的操作，也就是坏消息那个。转到 401362 那行，就是坏消息函数的第一行，右键选择“Find References To”->“Selection (or ctrl-R)”。这会再一次调出参考窗口。双击第一条，我们就会来到调用坏消息的地方：(p16)

有意思的是，它就在我们刚才设置的断点的上面 2 行！咱们在这里也设置一个断点：(p17)

****注意，有时候你选中一行然后查找参考，但是一个都没有。导致这个结果的原因有两种：1）你选择了错误的函数“入口点”，也就是调用这个函数应该 call 或 jump 其他的地方，但是它们却调用了别的行，有可能就是你选择行的前面或后面那行。选择正确的行来查找参考需要花时间和技巧，不过要坚持下去。2）代码中没有明显指向这一行的指令。记住，程序运行时有许多数字被动态的操纵，call 或 jump 指向的地址也不例外。所以，如果 call 的地址是动态创建的话，所以 OllyDbg 就没有办法提前知道这行会被调用，所以 OllyDbg 也就不会将这个参考列出来。关于这个也是有方法的，不过这会我不打算讨论。*

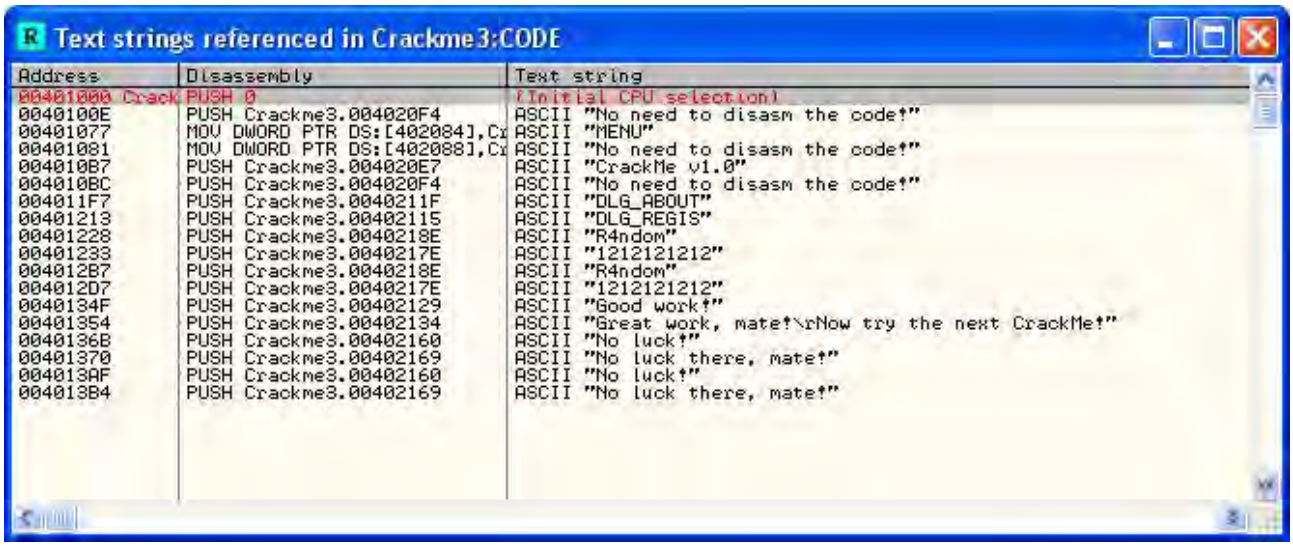
现在，如果我们看看这两个 CALL 的附件的话，会看到几个 jmp 指令。第一个，401243 的 JE SHORT CRACKME.0040124C。当然，你知道 JE 是啥意思，因为你已经读过汇编语言的书（参见 R.E.T.A.R.D. 规则#1），不过为了证实，我们假定你不知道这个特别的助记符（指令）是啥意思。这就是插件 MnemonicHelp 存在的原因。右键 JE 指令，在上下文菜单中选择“? JE”：(p18)



这个窗口比较长，因为有大量的跳转指令。如果我们向下看那个“JE”的话，会看到它是“Jump if Equal (ZF = 1)”。意思是如果 0 标志位被置 1 就跳转（或者被比较的两个项目相等）。前面的教程中我们复习过标志位，所以你应该知道，如果被比较的两个对象相等，JE 就会跳转。我们也能够发现，这个 JE 跳过了对坏消息的调用，并且紧随跳转的那条指令是对好消息的调用。如果 JE 没有跳，我们就会调用坏消息。所以，我们**想要**这个跳转实现，以便我们能够调用好消息。咱们操作下看看。在 JE 指令上设置一个断点，重启（或运行）应用。点击 crackme 中的“Help”->”Register”，输入用户名和序列号，然后点 OK: (p19)

哇！等等！显示了坏消息，并且 Olly 也没有断下来？也就是说 Olly 永远也不会运行到我们的断点！这是咋回事呢。

事实上，这个在逆向工程领域里是可以用得着的。我向你保证，每一个专家级逆向工程师/破解者这时候都会想“我错过什么了吗？一个 0xcc 中断？IsDebuggerPresent（译者注：一个 Windows API）？NTFlags？TLS 回调？”，然后白费力气去寻找一些过于复杂的解决方案。但是我们只是初学者，我们只有几个工具可以使用，其中一个就是搜索字符串，那就试试这个吧: (p20)



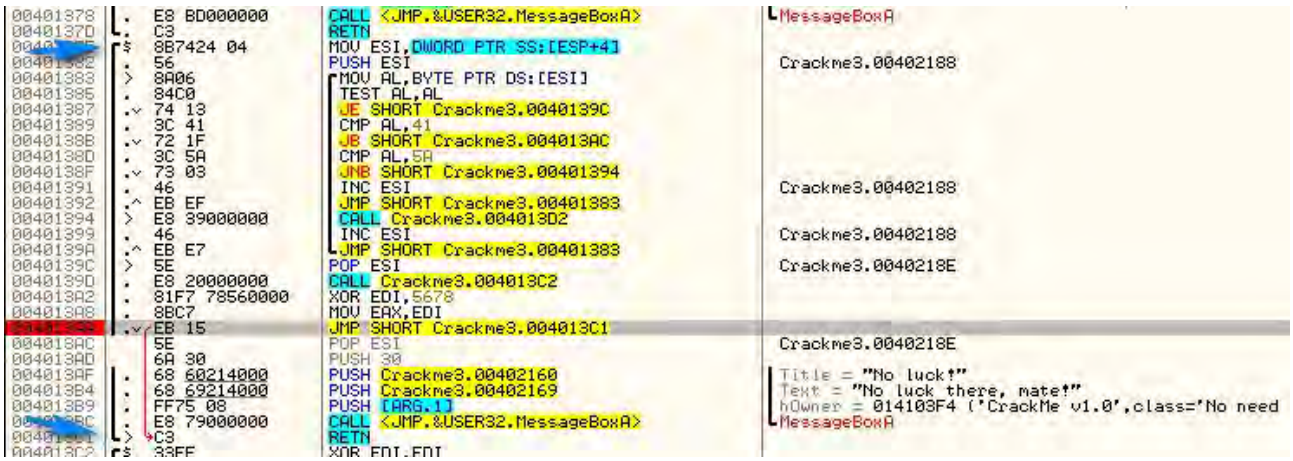
现在，你可以看到一些相当有趣的东西....有两个“No luck!”坏消息，但是只有一个好消息。也就是说，代码中的某个地方做了检查，如果没有通过就会显示坏消息。这是一个在反逆向工程中非常流行的技术：找一个非常明显的地方放好的/坏的消息，然后添加一个不是那么明显的检测。如果你看看代码

窗口我们的好消息和坏消息所在的位置，你会发现字符串“No luck!”是在40136B，所以我们知道那不是我们要找的字符串。所以咱们双击下4013AF那个:(p21)

这个坏消息是在程序内存中完全不同的区！我认为这个crackme是在太简单了！好，咱们深呼吸然后想想RETARD 规则 #2，找找 比较/跳转。本例中在4013AA处有个JMP，当你点击它的时候，OllyDbg会显示一个箭头刚好跳过了坏消息。看起来前途光明啊....那就试试吧！在那个jmp指令处设置一个断点，重启应用并运行。

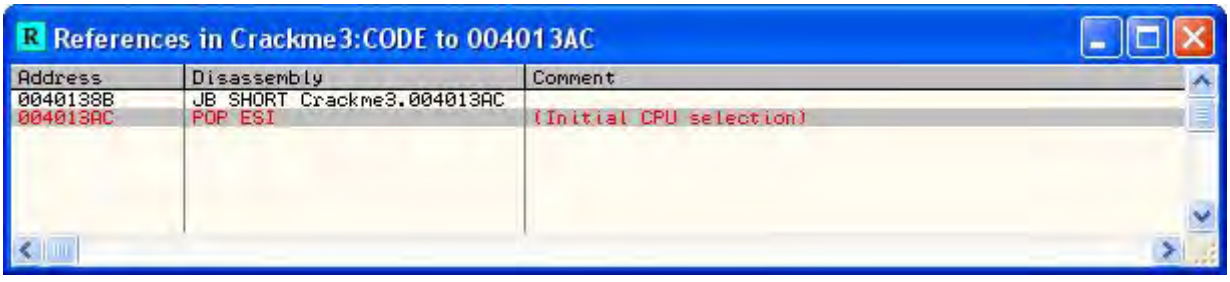
****你有可能得到错误的消息，就像我们上一章中断点被破坏那样。如果发生了，像上一次那样做就行了。打开BP窗口，在你运行程序前重新启用所有的断点:)*(p22)

操蛋!!! 好吧，不起作用，所以我猜我们得深入挖掘了。咱们看看代码，试试理解到底是什么个情况（该是你组合阅读大放异彩的时候了😀）: (p23)

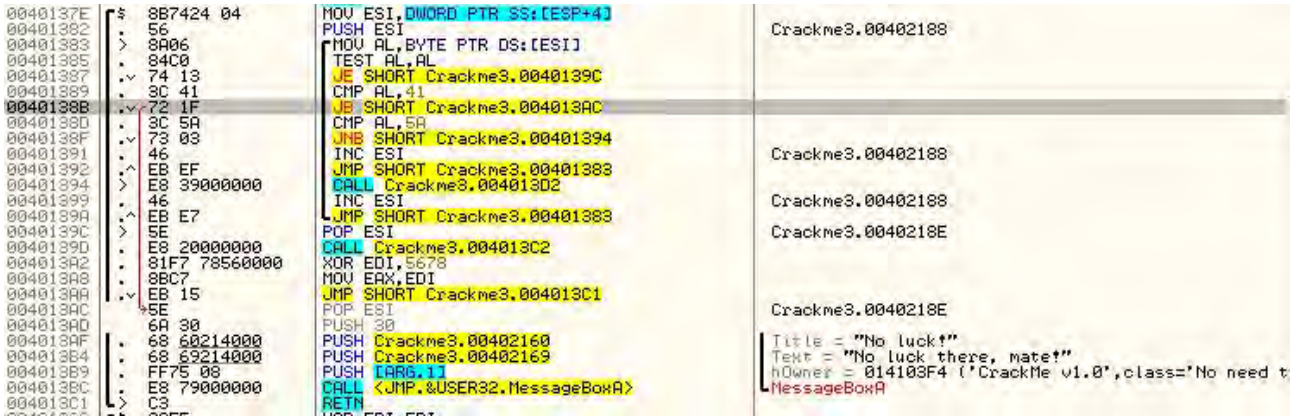


好，我们知道了一件事，因为教程的前面我们学过，就是函数的开始和结束点。图片中你能通过蓝色箭头看到。所以，从函数的起始点开始，有一个循环首先检查 AL 是不是 0(TEXT AL, AL)，然后循环将 AL 和一组数值（41，5a）进行比较。期间，有一些依赖于 AL 值得跳转。首先，咱们看看到底哪个跳转会调用我们的坏消息（有一个 JMP 指令刚好在坏消息前面，没有什么可以“空降”直达它。所以，必须有什么东西跳过那个跳转来运行坏消息代码。最有可能的跳转是在 4013AC）。

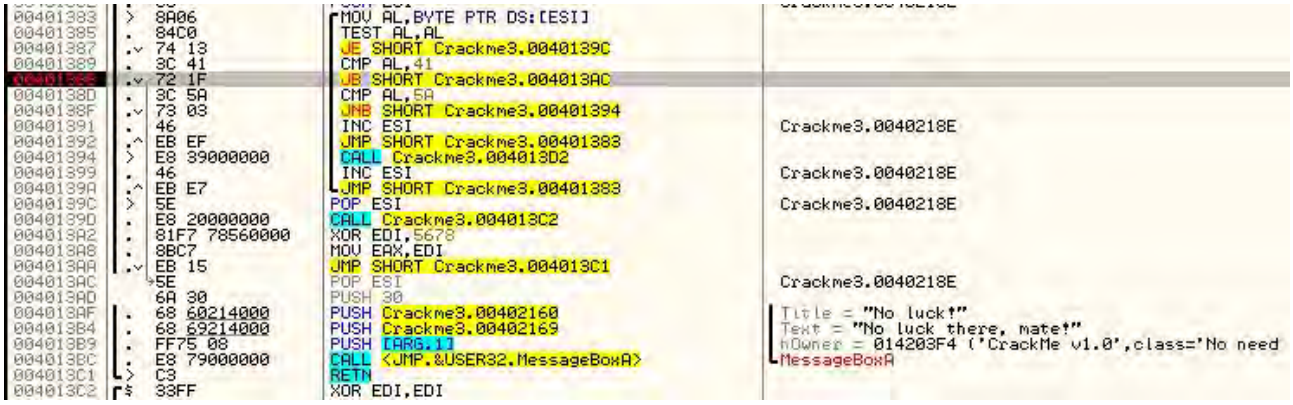
点一下 4013AC，也就是坏消息框的第一条指令，右键该行选择 “Find References To” -> “Selected Address”。（我知道一旦你点了这行，就会显示一个红色箭头，显示了哪条指令调用了它，但是我们怎么才能知道就没有别的指令调用坏消息呢。找到所有的参考可以帮助我们确定，有可能只有一个。）然后我们就再次看到了参考窗口：(p24)



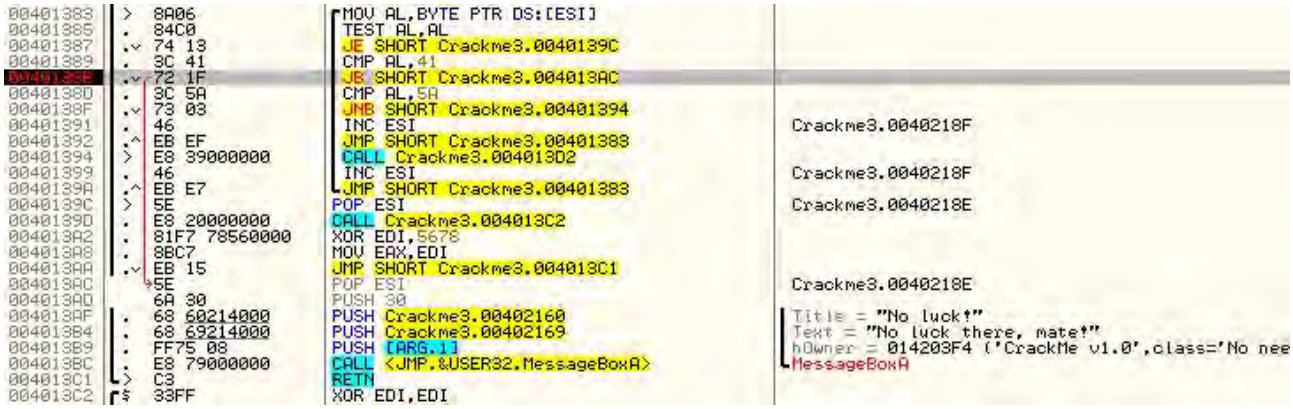
现在，双击第一个，咱们来看看哪一行正在调用坏消息：(p25)



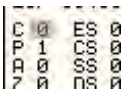
噢，原来是循环中的一个。注意参考窗口中旁边的那个红色的那行（我们现在可以忽略它），只有一个到该地址的参考，所以我们可以保证 40138B 这行是调用坏消息的唯一代码。所以 40138B 的 JB SHORT 4013AC 就是那个罪魁祸首。咱们试着在它上面设置一个 BP,临时修改下看看能否绕过这个坏消息。在 40138B 设断点，重新运行程序：（p26）



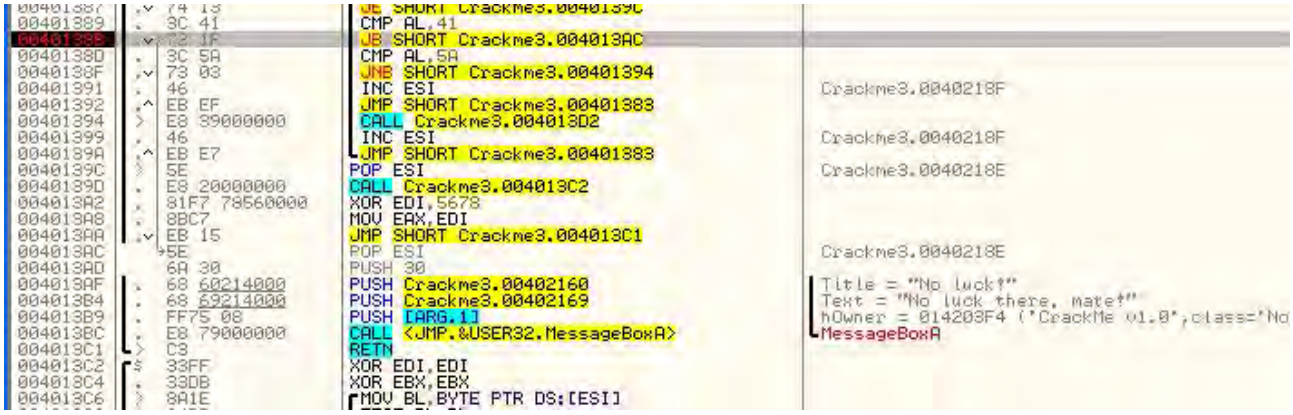
嗯。箭头是灰色的，我们知道在这次的循环迭代中我们没有跳到坏消息那。按下 F9 执行循环体：（p27）



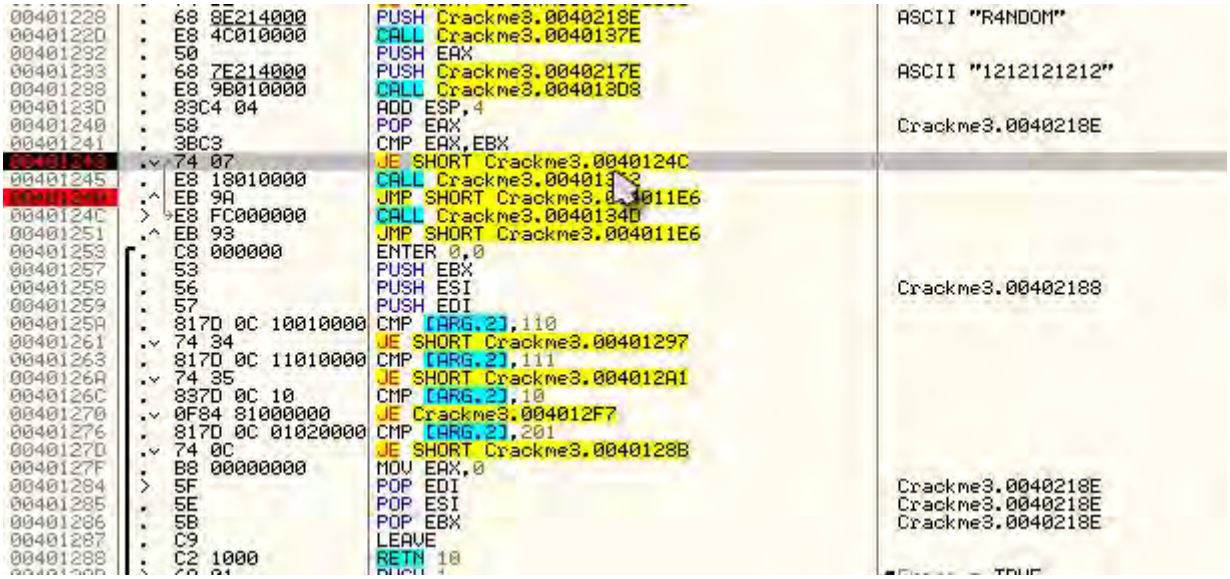
啊！第二次循环时它就要调用坏消息了。好，就让它那么干，看看我们跟踪的对不对。你可能注意到了，如果修改了 0 标志位，跳转仍然实现了。这是因为 JB 指令是跳转指令集中略有不同的那部分，它用进位标志位而不是 0 标志位（别担心，这些你的汇编语言书籍中全都有）。所以双击那个进位标志位（“C”）：（p28）



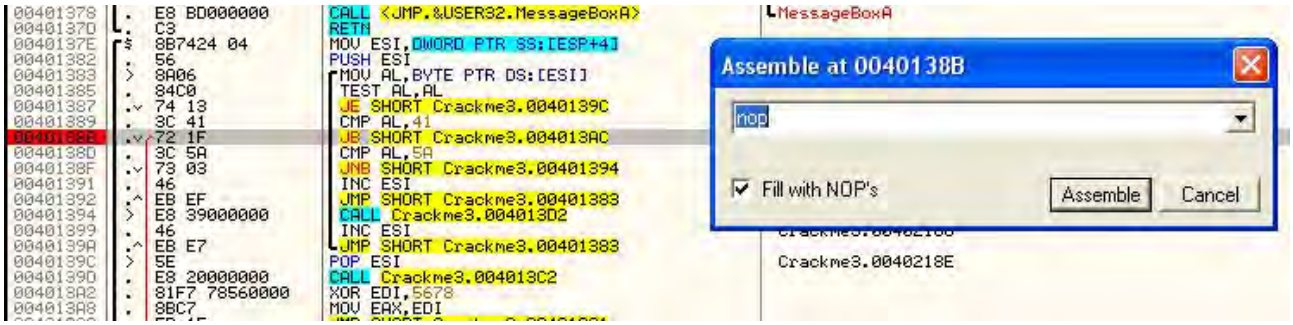
然后那个箭头就会变为灰色：（p29）



现在我们再次运行循环，看看循环中还有没有调用坏消息的。按 5 次 F9，没有一次调用坏消息。事实上，在第五次 F9 之后，我断在了一个旧断点处，我们首先想到的是补丁：(p30)

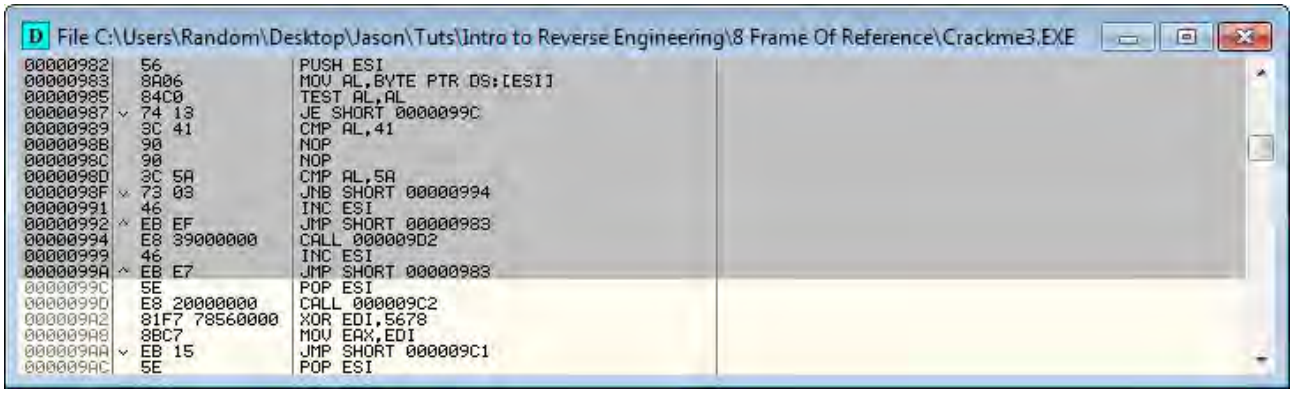


那么，这就意味着我们已经成功的绕过了对坏消息的第一个检测，并且回到了原来的检测点。咱们给第一个检测打个补丁，这样就再也不用操心它了，就可以将注意力集中在主要的检测点。回到 40138B 的断点处，我们得想想怎么给它打补丁而不让它跳转到坏消息那。记住，跳转是在第二次循环时实现的，只有在 AL 的值小于 41 时才成立（相关指令是 CMP AL, 31, JB SHORT 4013AC）。如果我们只 NOP 掉这个跳转会怎么样？它就再也不会跳了，我们一点也不用担心它会跳到坏消息那😀：(p31)

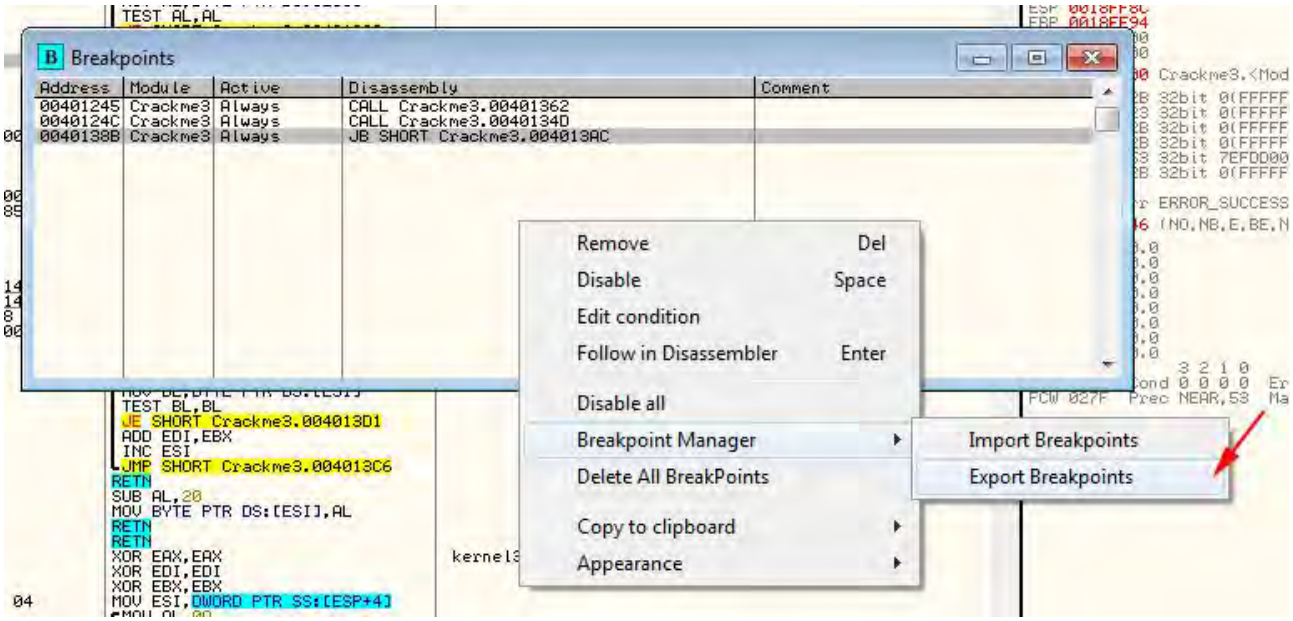


(p32)

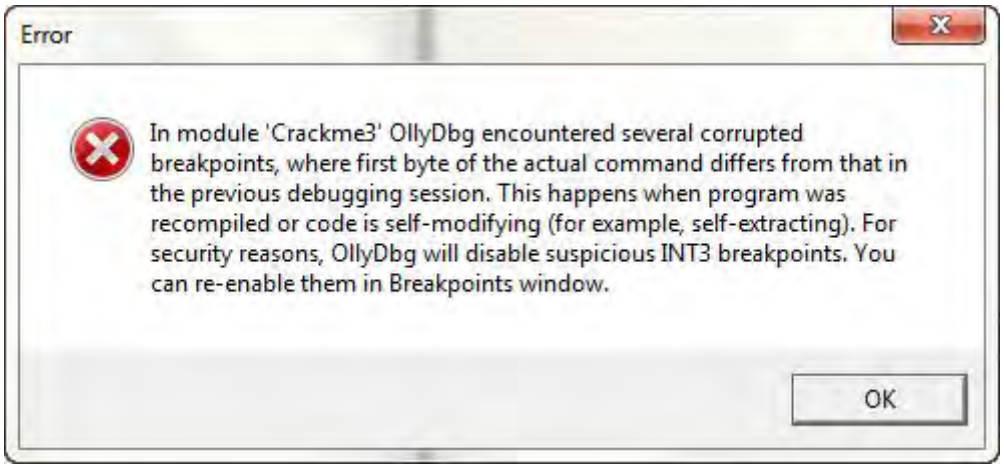
右键，选择“Copy to executable” -> “All modifications”。弹出内存窗口，右键该窗口，选择“Save File”，将其另存为 crackme_patch1.exe: (p33)



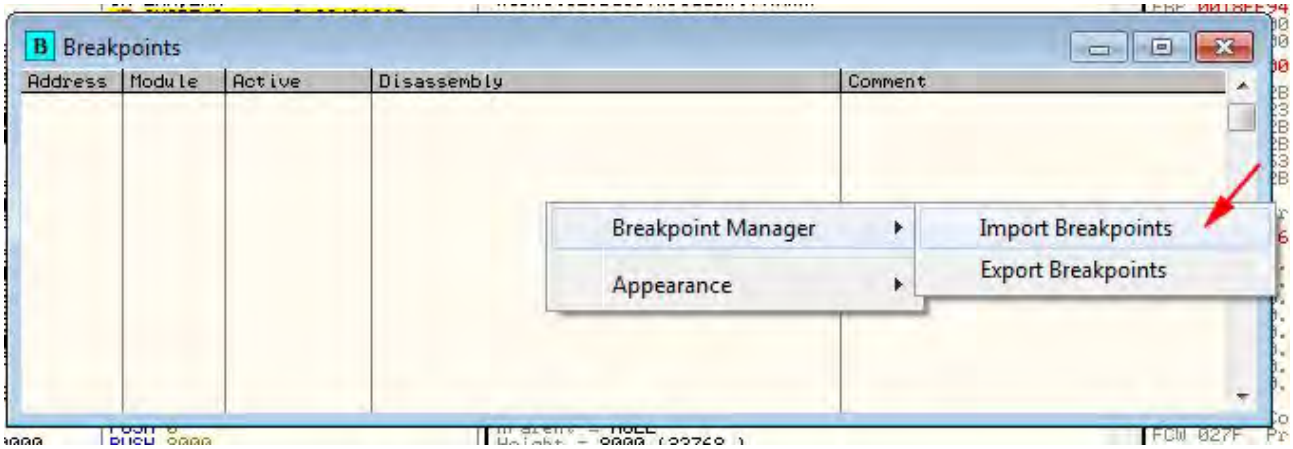
现在，在重新载入刚刚打过补丁版本前，我们要明白所有的补丁、注释和（尤其是）断点都会被删除，因为所有的信息都存储在 Crackme3.udd 这个 UDD 文件中。我们将要打开的 Crackme3_Patch1，并没有和它相关的 UDD 文件。不过还是有几个好消息的。本文的相关下载中包含有断点管理插件。如果你没有准备好，那就将其拷贝到你的插件目录下，然后重启 Olly。如果你一开始就安装好了，那你就已经载入了它。现在打开断点窗口，右键并选择“Breakpoint manager” ->” Export Breakpoints”: (p34)



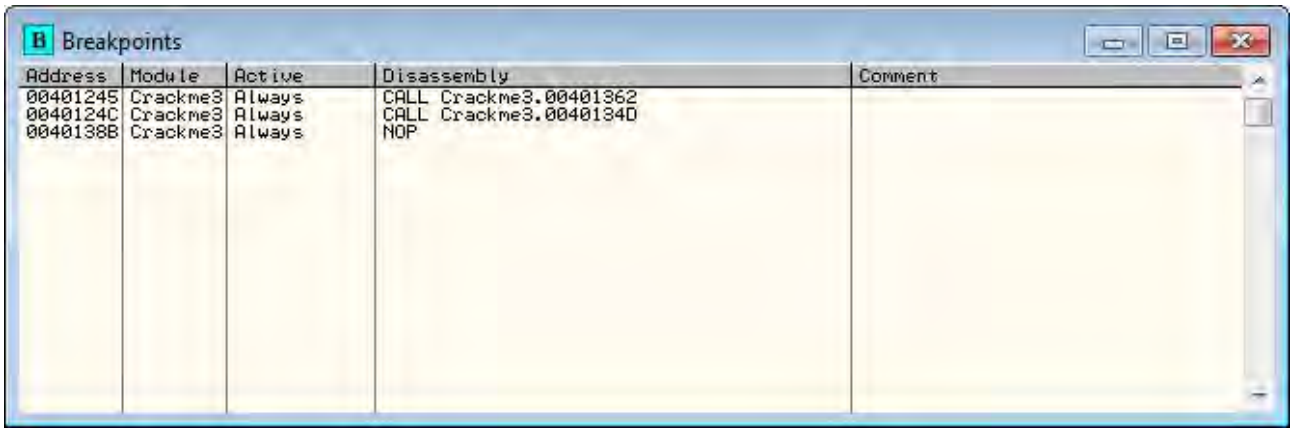
保存文件，因为我们将会将其导入到新文件。现在，将新文件（打过补丁的）载入 OllyDbg。它很可能会弹出一个消息框，告诉你断点被破坏了：(p35)



点 OK 就行了。在我们的打过补丁的程序中打开断点窗口，很可能所有的（或大部分）断点消息了。现在，右键并选择“Breakpoint Manager” -> “Import breakpoints”：(p36)

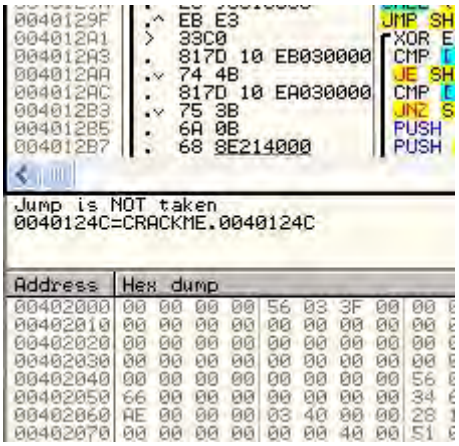


现在你会看到我们原来的断点又回来了：(p37)



运行程序，Olly 会断在我们的第一个断点也就是 401243 的 JE 指令处（如果你没有在该行设置断点，那就设一个。译者注：吐一下槽，原作者太操蛋，从来都只打圆括号的左半个，右半个就不管了，我还得自己琢磨把右边的圆括号放哪）。重启应用并运行，你会断在这里：(p38)

现在，看那个灰色的箭头，就是从当前暂停行往下到 40124C。因为箭头是灰色的，所以跳转不会实现。你也可以看反汇编窗口与数据窗口中间的那块，它告诉你跳转**没有**实现：(p39)



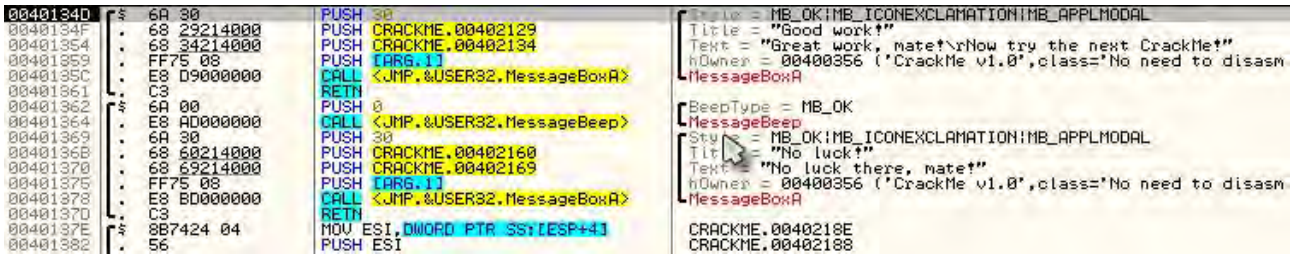
这意味着，什么都不用做，程序不会跳到第二个调用，会直达第一个调用。第一个调用会跳到坏消息那，所以我们真心不想让它发生。按一下 F8 单步步过。就像 Olly 告诉我们的，没有跳转，我们当前的位置就在调用坏消息的地方。按一下 F7 单步步入那个 CALL，我们来到了坏消息所在函数的第一条指令处。现在，如果我们按 F9 让程序运行，我们看到的正是意料之中的：(p40)

咱们来看看能不能解决这个😀。重启应用，按 F9 运行之，选择“Help”->”Register”，然后输入用户名和序列号。现在，当你按下 OK 按钮时，Olly 会再次停在我们的第一个断点：(p41)

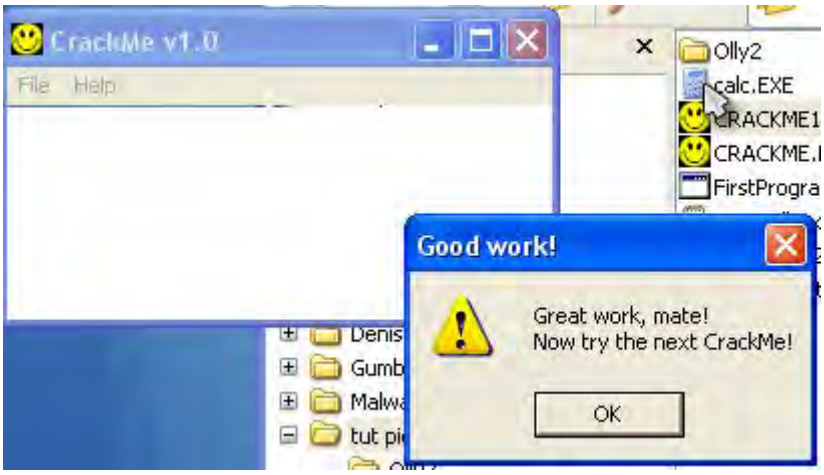
这回，咱们来帮助 O11y 走正确的路。浏览下寄存器窗口，注意到 Z 标志位是红色的。嗯，你知道该怎么做了：(p42)

注意，箭头是灰色的，显示跳转不会发生，不过现在变红了，在反汇编窗口和数据窗口之间的那个区域已经变成了“Jump will be taken（跳转将会发生）”。我们所做的就是告诉 O11y 去修改标志位，该标志位用来判定两个东西是否相同，为了让它认为它们是相同的。所以现在，我们会跳过对坏消息的调用，转而去调用好消息!!!

咱们试试。按 F8 执行跳转，再按 F7 单步步入到那个 CALL。现在我们将跳转到好消息的起始处：(p43)

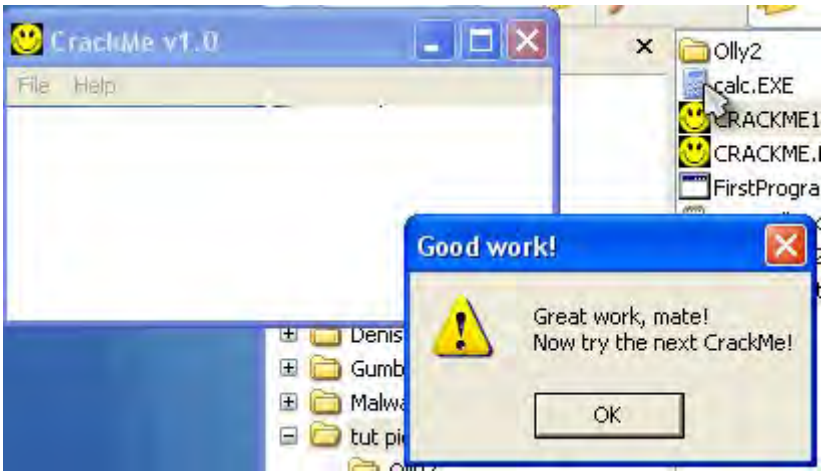


现在，按几次 F8，每按一次就观察一下堆栈窗口。你会看到 MessageBoxA 的参数被压入堆栈，本例中确实是好消息被压入堆栈。只要你单步步过 40135C 处的函数，新的消息对话框就会显示出来。我们已经破解了我们的第一个程序!!! (p44)



现在的问题是，我们只是临时性的修改了标志寄存器，当程序再次运行时，它却不会再次的修改标志位，所以我们还是会得到坏消息。我们需要做的是通过某种方式将修改保存起来，以便于每一次程序运行的时候，我们都能强制它做跳转。这时候补丁要派上用场了。和我们以前做的一样：选中所有已修改的行，右键并选择 "Copy to executable"。在弹出的窗口中右键，点击 "Save file"。选一个名字，这就是你的打过补丁的版本：(p45)

现在可以关掉数据窗口和 OllyDbg 了。打开你保存的打补丁版本文件夹，运行打过补丁的程序。输入你的信息并验证：(p46)



干的漂亮！你已经真正的破解了一个有些挑战的 crackme。

第九章：无相关字符串

一、简介

此次教程中我将会向我们的武器库中加入一个新的装备。如果搜索二进制文件时发现没有可用的字符串你怎么办？我将会介绍一个新的 R.E.T.A.R.D. 规则。此次教程（下一章也是）我们将研究“TDC”写的一个 crackme 叫 Crackme6，相关下载里面包含有。总之，它不是一个硬骨头，不过我将会对其进行一些高级分析，好为将来的教程做准备。

你可以在教程页下载相关文件以及本教程的 PDF 版本。

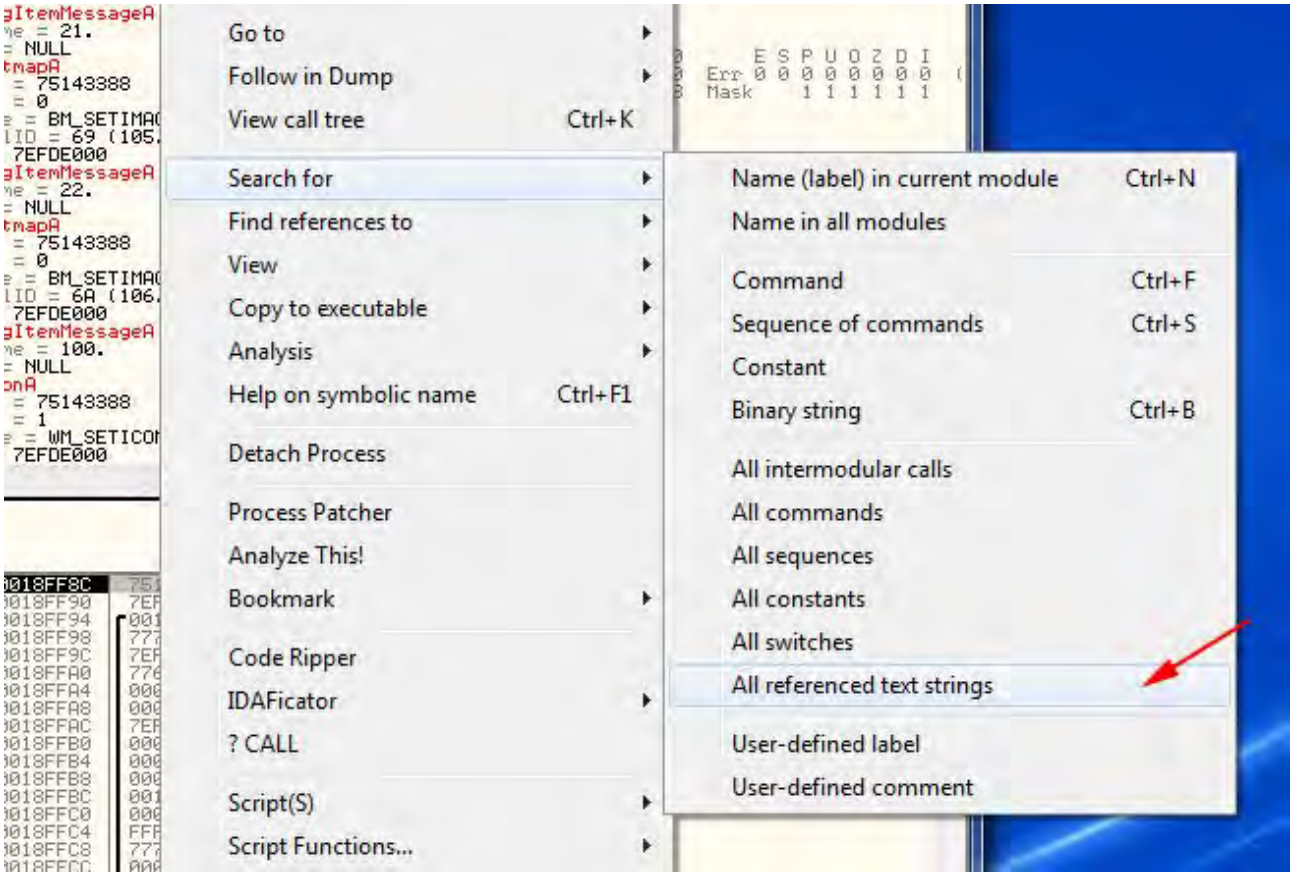
那么，咱们开始吧.....

OllyDbg 载入 Crackme6：（p1）

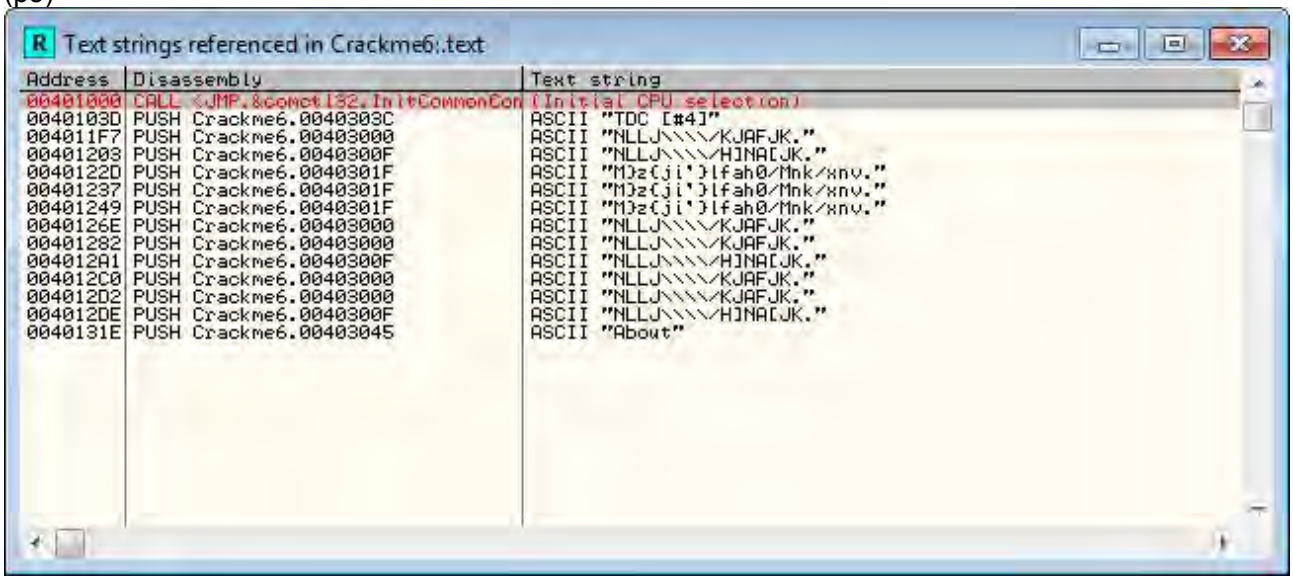
现在，我们已经知道操作程序了。运行程序看看情况：（p2）

嗯，看起来挺简单的。我输入了一个密码 1212121212，下面就是返回的情况：（p3）

相当直接呀。试试我们拿手的“字符串搜索”，看看有什么：(p4)



(p5)



搞什么鬼这是!!! 这些一点用也没有啊😟。我们可以拿这些字符串干啥！？！？明显，这个 crackme 将字符串加密了（或者是作者说一种很奇怪的语言:D）。好，是个好时候介绍

R4ndom's Essential Truths About Reversing Data #3:
R4ndom 关于逆向数据的必备真理#3：

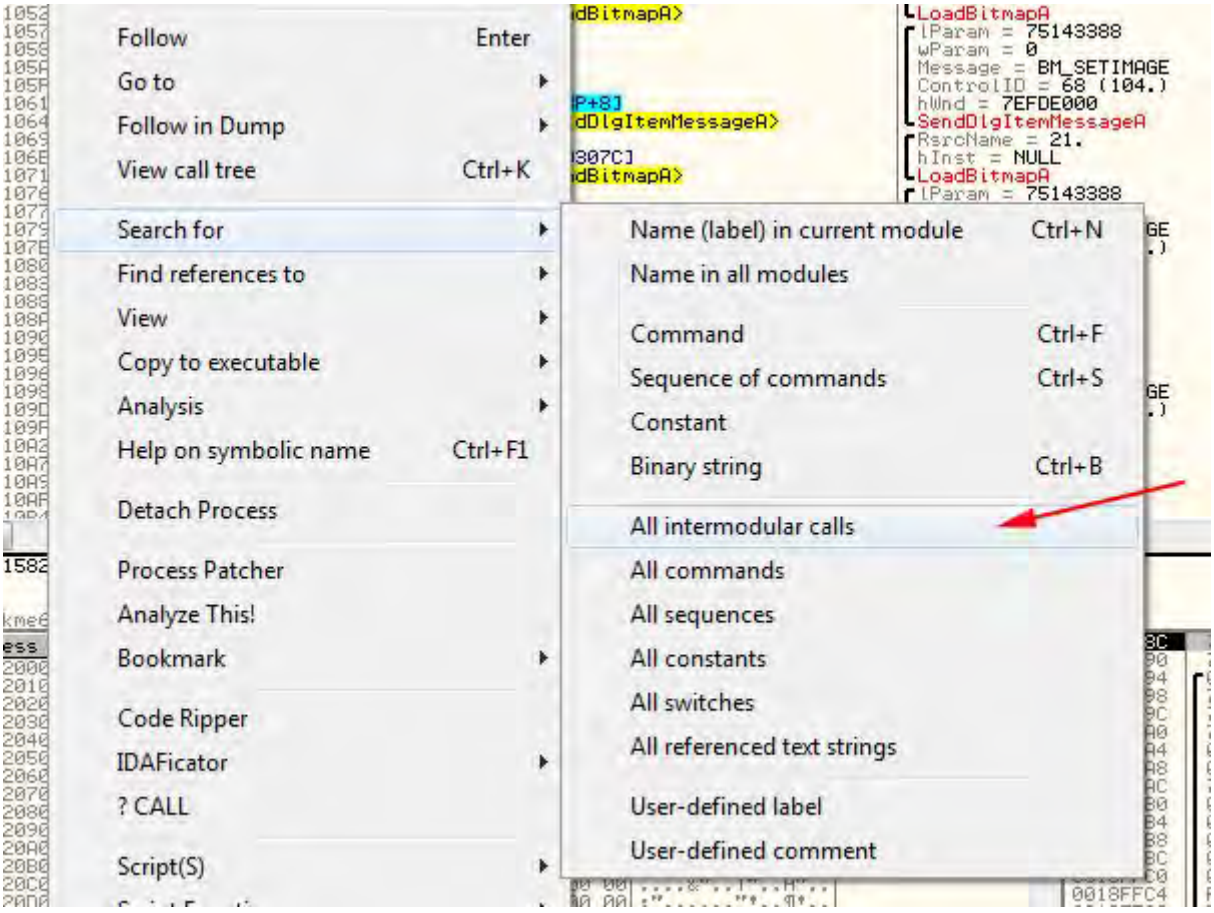
#3.不要依赖二进制文件当前已有的字符串。

不幸的是，在你开始研究真正的二进制文件（比如商业产品）时，它们中的大部分被以某种方式打包 以及/或 保护。干扰逆向工程师的一个最明显的方法是加密字符串。坦率地说，在逆向工程领域当我第一次研究一个感兴趣的新的二进制文件时，如果我搜索字符串并且搜出来了，我能够假定那个二进制文件很可能没有多少挑战。所以，你不能够依赖于那些东西（如果有当然更好🙂）。

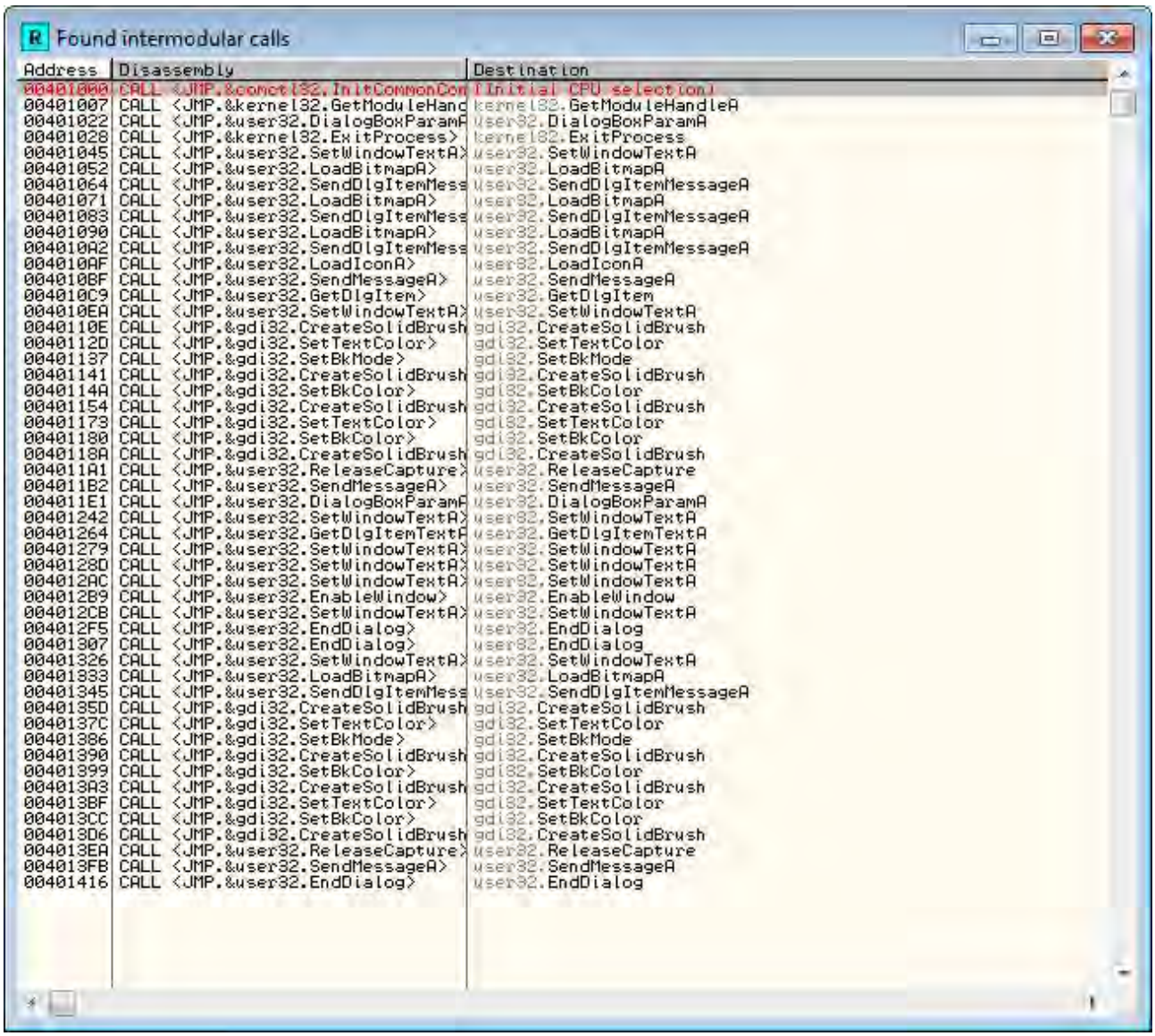
二、模块间的调用

有鉴于此，我向你展示一个新的在没有字符串的情况下的技巧。大部分的 Windows 应用程序使用一个标准的 API 集来完成特定的动作。例如，如果需要一个简单的消息框的话就调用 MessageBoxA，当程序想要退出的时候就调用 TerminateProcess。因为大部分的应用都使用这些相同的 API，我们可以用这个获利。例如，有些 API 可以用于从对话框的输入框（类似用户名和序列号）获取文本。有可以被调用用来比较两个字符串的字符串比较函数（输入的密码和程序中存储的密码相同吗？）。有读写注册表的 API（存储和读取你的注册状态）。

Olly 提供了一种搜索所有被调用的 API 的方法。在反汇编窗口右键，选择“Search for” -> “All intermodular calls”：(p6)

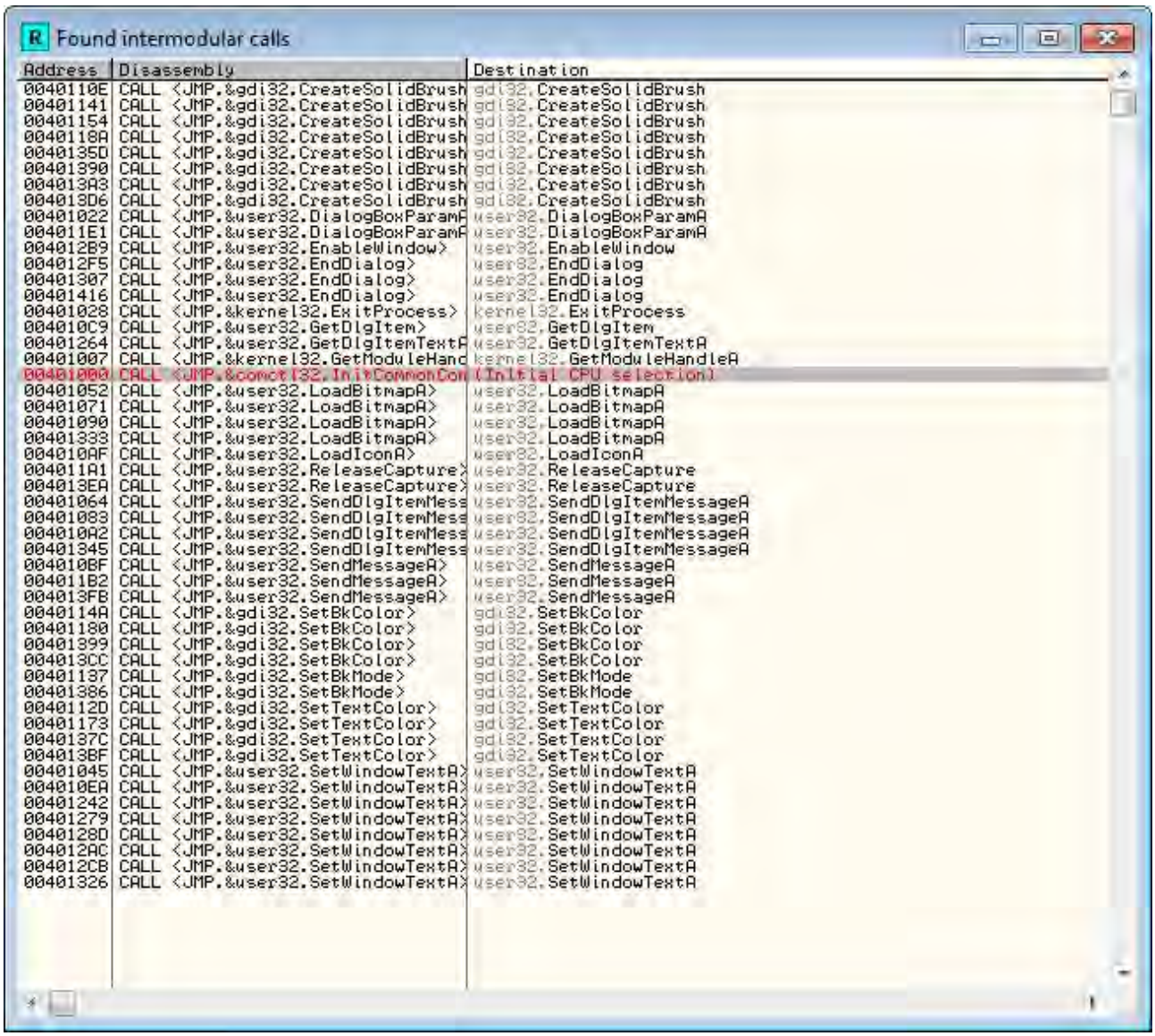


Olly 会弹出 Found intermodular calls 窗口：(p7)



通常我做的第一件事是点一下“Destination”，将列出的函数按字母顺序进行排序（而不是按地址排序）：(p8)

现在，如果你看第三列的话，你可以看到该 crackme 调用的所有 API：(p9)



这是一个小程序，所以调用的函数不是那么多。大部分的程序都有数百个。不过通过这个列表，你可以了解到一个二进制文件的很多信息。你可以发现它用一个对话框作为主窗口。它载入了一个自定义的位图。它修改了对话框中的一些颜色。

在更大些的应用中，这个窗口的价值更高，因为它能告诉你这些事情：1）是否有注册表相关 API 被调用用来存储和获取信息？是不是有 API 呼叫网站来验证我们确实注册了？3）有没有读写一个可能存有注册码文件的 API？当我们研究一个加壳的二进制文件时，这个窗口将更加重要（这个后面讨论）。

尽管如此，有几个特定 API 逆向工程师总是会留意，因为这几个在保护机制中用的比较多。包括：

DialogBoxParamA
GetDlgItem
GetDlgItemInt
GetDlgTextA

GetWindowTextA
GetWindowWord

LoadStringA
lstrcmpA

wsprintfA

MessageBeep
MessageBoxA
MessageBoxExA
SendMessageA
SendDlgItemMessageA

ReadFile
WriteFile
CreateFileA

GetPrivateProfileIntA
WritePrivateProfileStringA
GetPrivateProfileStringA

不幸的是，这里没有包括你可能遇到的所有 API，不过幸运的是，大部分应用使用下面的其中一个:

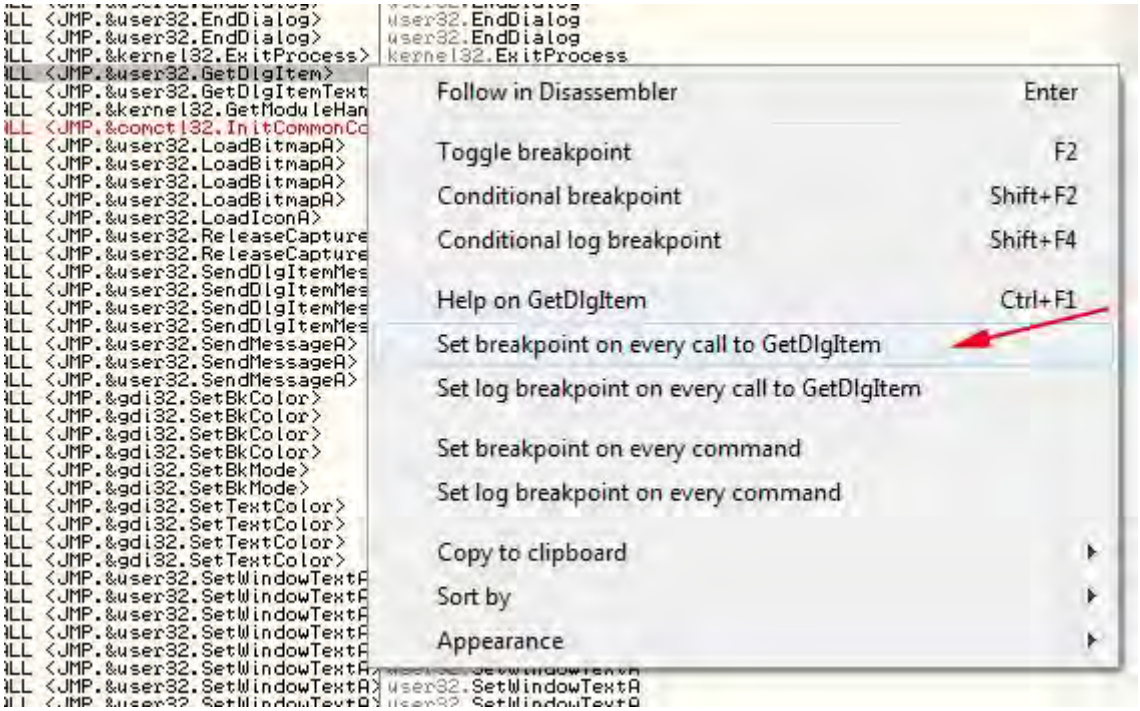
GetDlgItemTextA
GetWindowTextA
lstrcmpA
GetPrivateProfileStringA
GetPrivateProfileIntA
RegQueryValueExA
WritePrivateProfileStringA
GetPrivateProfileIntA

如果你专注这 8 个 API 的调用，你就可以处理绝大多数的实例。还有别忘了，“Get help on symbolic name”是可以给你提供帮助的。

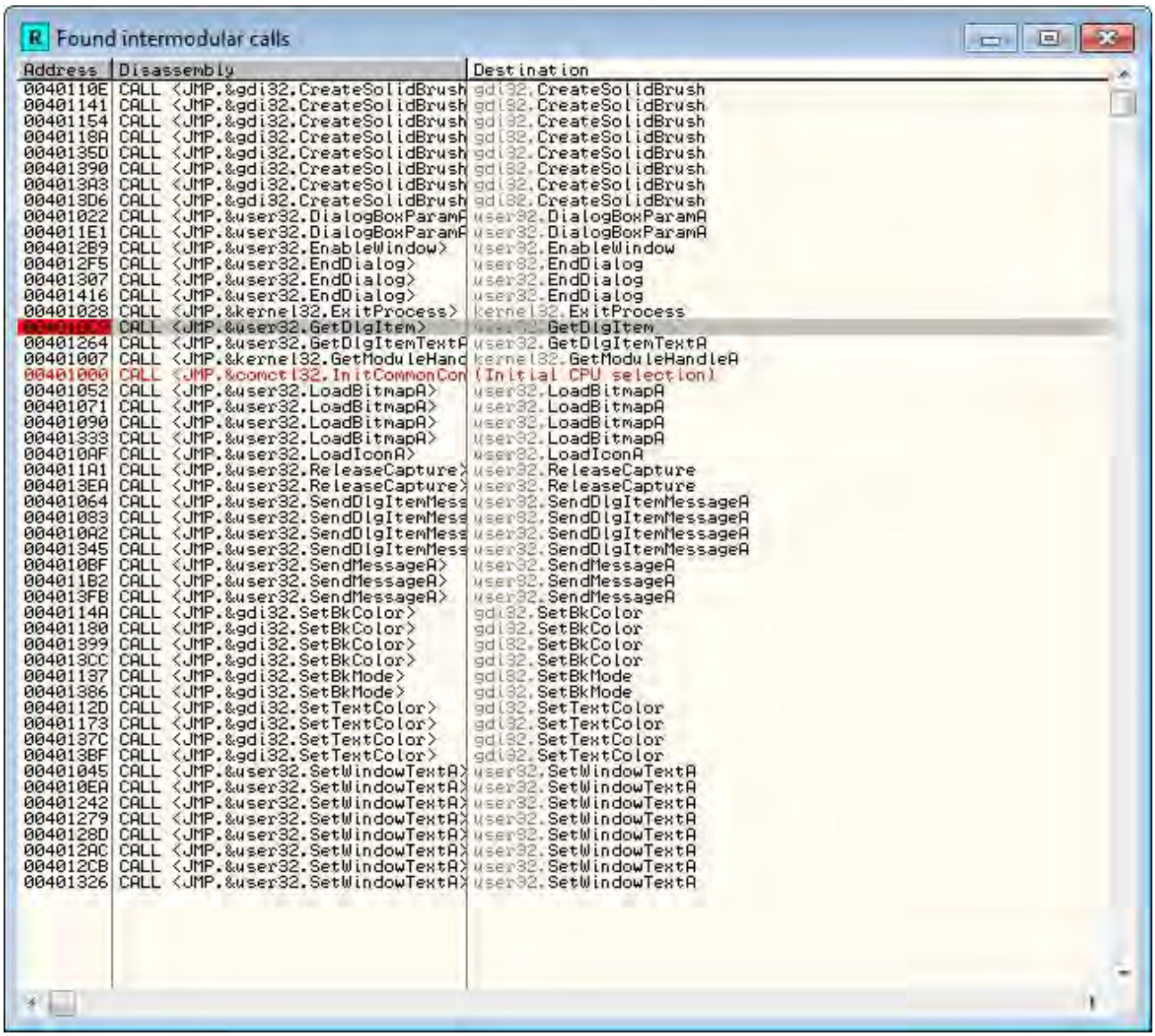
现在，在 Olly 查找出的 crackme 的调用列表中往下看，在那个简短的列表中有两个 API:

GetDlgItem 和 **GetDlgItemTextA**

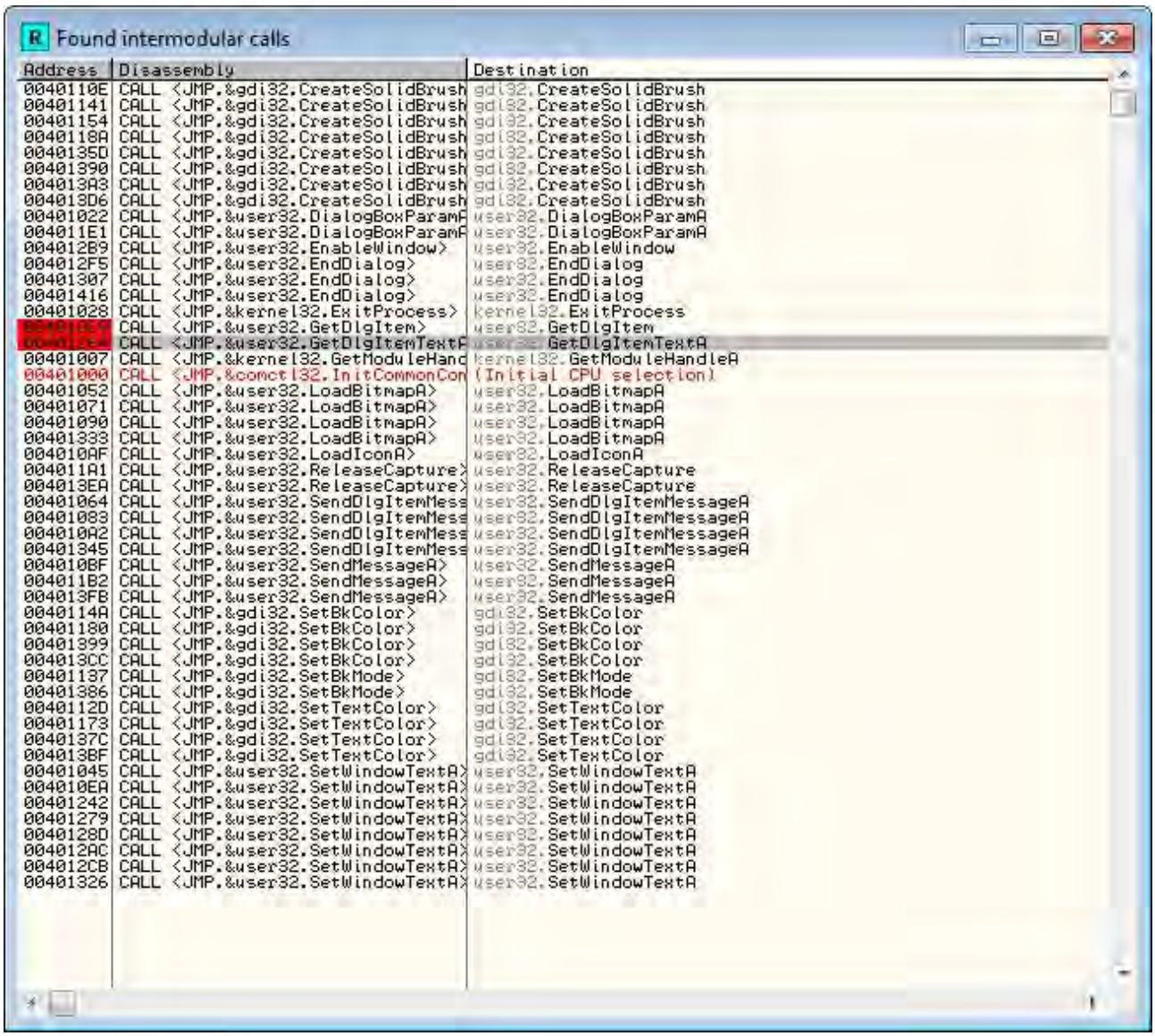
这两个函数是用来获取输入到对话框中文本框的文本。好吧，在我们的教程中，这只可能说明一件事，获取我们输入的密码。我们想要做的是，不管什么时候只要 Olly 遇到两者中的一个就暂停。方法是，选中你想要关注的 API 那行，右键然后选择 “Set breakpoint on every call to ____”，这里的____是 API 的名称（这里是 GetDlgItem）：（p10）



现在，我们看到 Olly 已经在该行设置了一个 BP：（p11）



我们也想在另一个 API GetDlgItemTextA 那暂停，那么点击选中它，右键然后和前面一样进行操作: (p12)

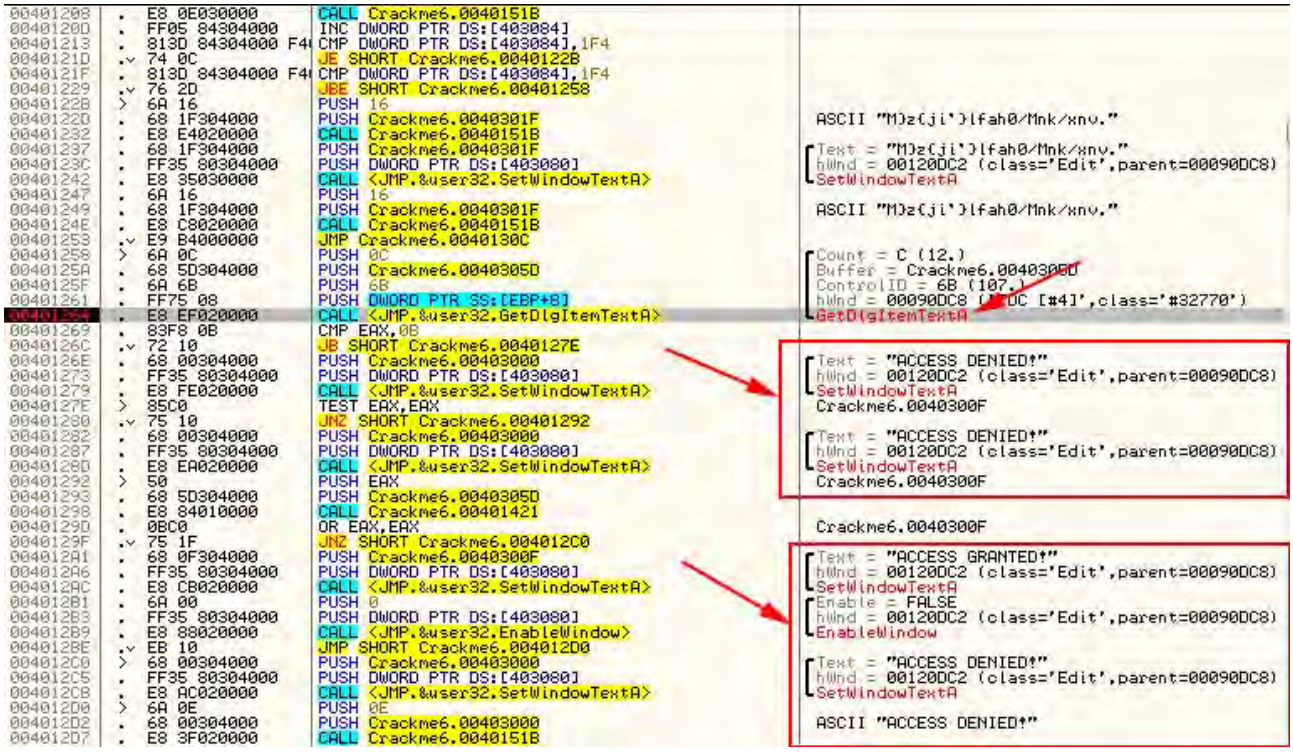


现在，不管什么时候只要 O11y 遇到了对这两个 API 的调用，它都会断下来（在调用被执行前）。咱们来试试看。重启 crackme 并运行。O11y 会断在对 GetDlgItem 的调用处：（p13）

现在，因为我们还没有输入任何内容，我们对 GetDlgItem 取到了什么东西不感兴趣，好咱们继续（F9）：（p14）

现在输入一个密码，然后点“check”：(p15)

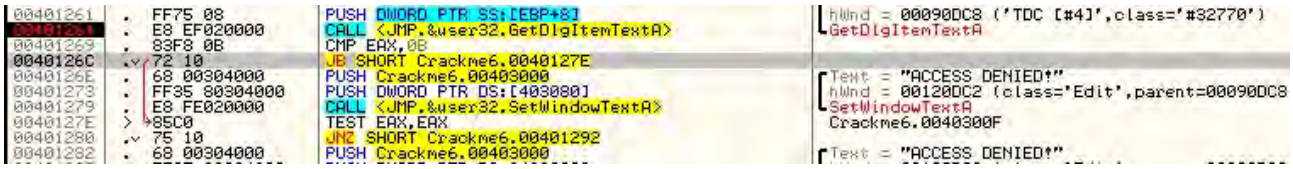
Olly 会再次断下来，这次是在 GetDlgItemTextA：(p16)



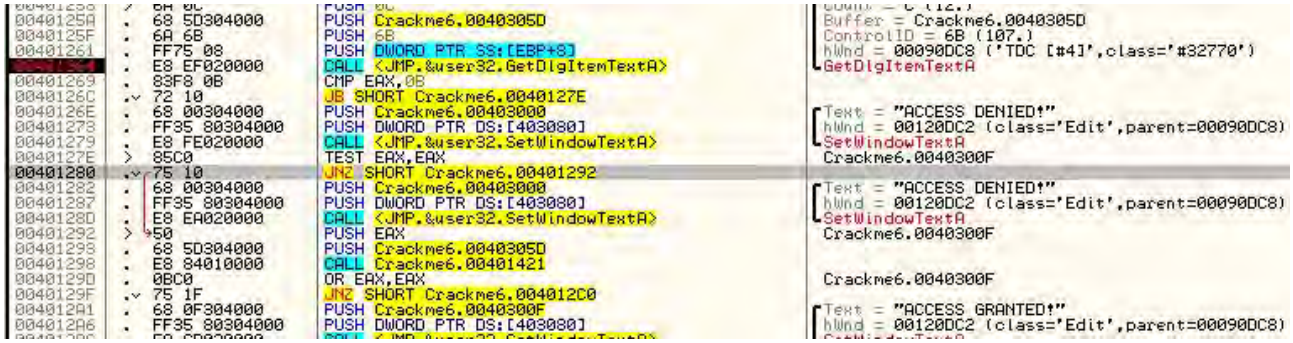
如果你看看周围，你会注意到我们已经来到正确的位置🙂。搞笑的是，我们起初搜索字符串的时候，没有一个是这些字符串中的🙂。

三、破解应用

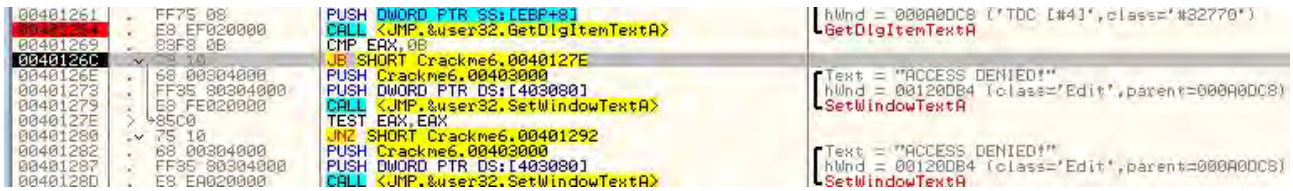
咱们快速浏览下附近的....。我们注意到有一个跳转（JB）跳过了第一个“ACCESS DENIED”，所以我们关注一下它：（p17）



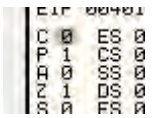
有一个跳转（JNZ）跳过了第二个坏消息，所以我们也将其加入关注名单。然后我们就会直接穿过到达好消息，所以基本上我们想要确保我们跳过了这两个跳转：（p18）



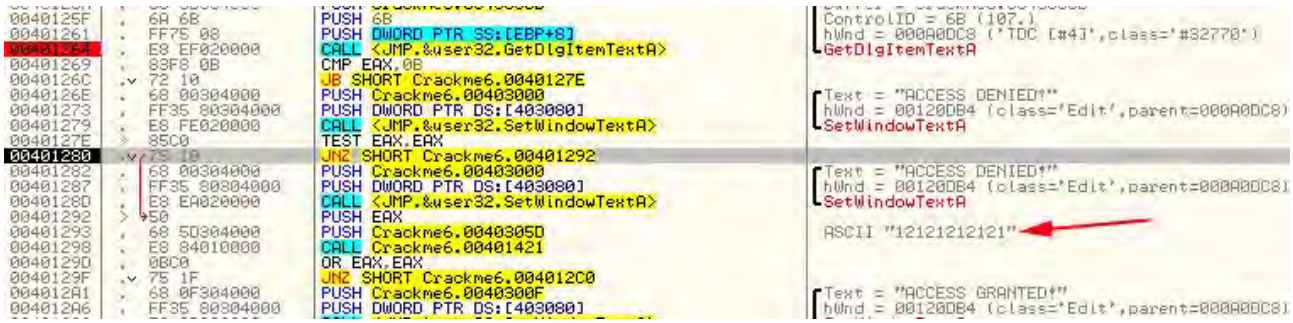
咱们试试，看看咱们是不是对的。再一次运行程序，我们应该断在 GetDlgItemTextA 指令处（记住绕过第一个断点）：（p19）



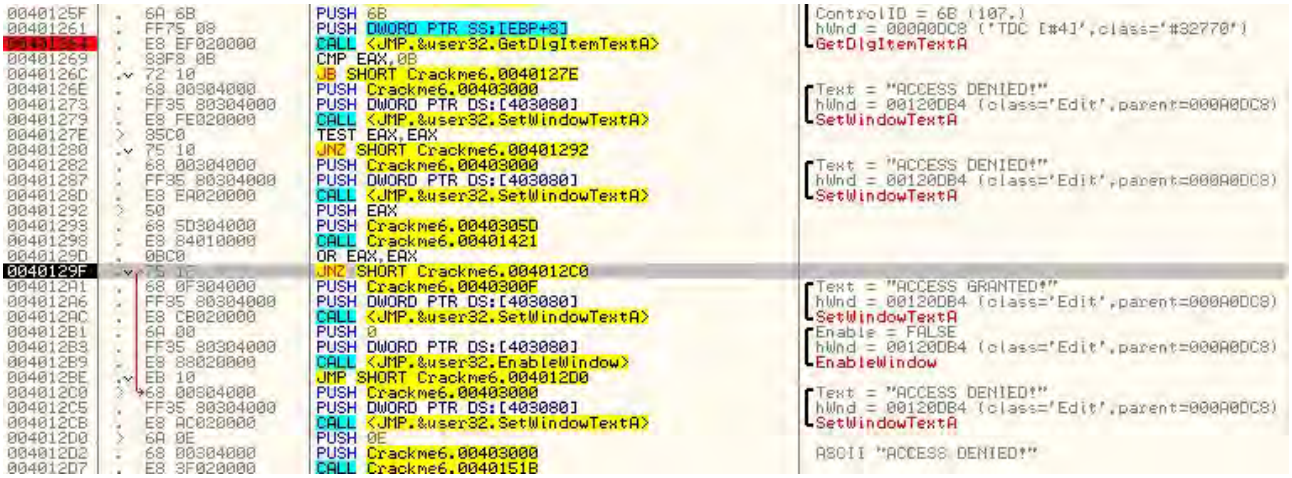
因为这是一个 JB 跳转，所以我们需要翻转进位标志位：（p20）



这样就会强制跳转。现在我们将做另一个 TEST，停在了 401280 处的跳转那。注意，我们的密码已经出现了注释列😀：（p21）



我们想要那个跳转实现，因为它跳过了第二个坏消息，所以我们只需要继续单步直到到达 40129F 的 JNZ 指令：（p22）



好，这条指令将会跳过我们的好消息，所以我们想要阻止它跳。你知道该怎么做：(p23)

现在运行程序（F9），看看我们已经成功的破解了程序：(p24)

四、家庭作业

作为一个挑战，试着给这个 crackme 打补丁，基于那个我们已经修改的标志位。在将打过补丁的程序保存后，你应该可以运行它，输入任何密码（少于 11 个数字）它都会提示 “Access Granted”。记住有几个补丁可以完成这个任务，所以如果一个不起作用，那就找下一个。

加分题：给 crackme 打补丁，让它接受任意长的密码。

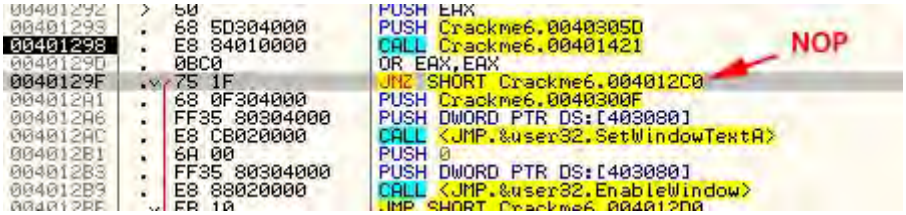
第九章续：“无相关字符串”的解决方案

一、简介

针对第九章的家庭作用，这里我提供几个解决方案。记住，有无数种方法可以破解这个程序，而这只是一个很小的例子。如果你自己找到了一个方法，那么恭喜你。如果没有的话，也别着急，我们将分多次解决它。

二、方案一

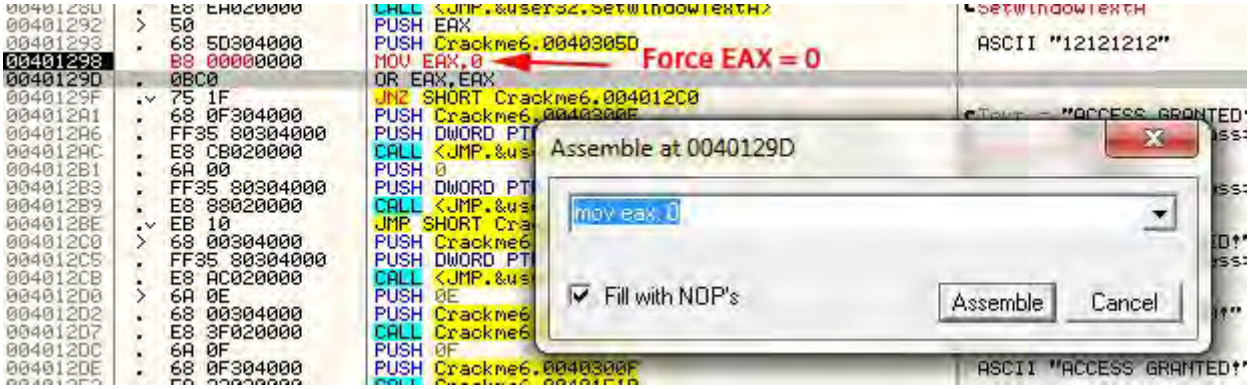
最简单的一种方法是给程序打补丁，只需要将 40129F 处的 JNZ 指令 NOP 掉就行:



这会强制程序每一次都直接空降到好消息那。

三、方案二

另一个可行性方案是,让 EAX 一直等于 0,将那个检测密码的 CALL 用 MOV EAX, 0 替换掉:



这个基本上将检测密码可行性的 CALL 整个删除了，程序将总是跳转到好消息那😀。

四、方案三

继续方案二的思想，我们将那个 CALL 留下，在它返回以后，我们再强制 EAX 等于 0。只需要将 OR EAX,EAX 替换成 XOR EAX, EAX 即可:

我喜欢这个解决方案，对此还有一定的讽刺意味（你只打了一个字节的补丁，只加了一个字母😀）。（译者注：我觉得讽刺意味应该是，一个程序的保护机制，加一个字母就搞定了，确实挺讽刺的。）

五、加分题

我希望加分题没有给你带来烦恼。移除密码长度限制的最简单的方法是替换掉原始的跳转，如果密码太长的话，就用一个直接跳转到好消息的 JMP 替换掉原始跳转。

```
00401268  .  E8 EF020000    CALL <JMP.&user32.GetDlgItemTextA>          GetDlgItemTextA
00401269  .  83F8 0B        CMP EAX,0B
0040126C  .  EB 33          JMP SHORT Crackme6.004012A1
0040126E  .  68 00304000    PUSH Crackme6.00403000                      Text = "ACCESS DENIED!"
00401273  .  FF35 80304000  PUSH DWORD PTR DS:[403080]                  hWnd = 000D083A (class='Edit'
00401279  .  E8 FE020000    CALL <JMP.&user32.SetWindowTextA>           SetWindowTextA
0040127E  >  85C0           TEST EAX,EAX
00401280  .v 75 10          JNZ SHORT Crackme6.00401292
00401282  .  68 00304000    PUSH Crackme6.00403000                      Text = "ACCESS DENIED!"
00401287  .  FF35 80304000  PUSH DWORD PTR DS:[403080]                  hWnd = 000D083A (class='Edit'
0040128D  .  E8 EA020000    CALL <JMP.&user32.SetWindowTextA>           SetWindowTextA
00401292  >  50             PUSH EAX
00401293  .  68 5D304000    PUSH Crackme6.0040305D                      ASCII "12121212"
00401298     E8 84010000    CALL Crackme6.00401421
0040129D     0BC0           OR EAX,EAX
0040129F  .v 75 1F          JNZ SHORT Crackme6.004012C0
004012A1  .  68 0F304000    PUSH Crackme6.0040300F                      Text = "ACCESS GRANTED!"
004012A6  .  FF35 80304000  PUSH DWORD PTR DS:[403080]                  hWnd = 000D083A (class='Edit'
004012AC  .  E8 CB020000    CALL <JMP.&user32.SetWindowTextA>           SetWindowTextA
004012B1  .  6A 00          PUSH 0                                      Enable = FALSE
004012B3  .  FF35 80304000  PUSH DWORD PTR DS:[403080]                  hWnd = 000D083A (class='Edit'
004012B9  .  E8 88020000    CALL <JMP.&user32.EnableWindow>             EnableWindow
004012BE  .v EB 10          JMP SHORT Crackme6.004012D0
004012C0  >  68 00304000    PUSH Crackme6.00403000                      Text = "ACCESS DENIED!"
004012C5  .  FF35 80304000  PUSH DWORD PTR DS:[403080]                  hWnd = 000D083A (class='Edit'
004012CB  .  E8 AC020000    CALL <JMP.&user32.SetWindowTextA>           SetWindowTextA
004012D0  >  6A 0E          PUSH 0E
```

这个相当的巧妙（在下一章中有更好的办法），不过确实有用。这样做的好处是，不仅修补了程序让它总是接受你的密码，而且和上面的方法不一样的是，它也移除了对密码的所有限制。

第十章：打补丁的层次级别

一、简介

本章我们会讨论给二进制文件打补丁的不同的层次级别。本章有点长而且详细，涵盖较多的背景知识，有些还不简单。我想给你展示一个深入分析二进制文件的例子，以及它需要什么。你可能大部分都不能够理解，不过它会给你一个非常好的总览逆向工程的一个好的机会。这样在将来的教程中，你会有一个参考框架。我们用上一章的那个 crackme 来研究，就是“TDC”写的 Crackme6，相关下载中包括的有。

你可以在教程页现在相关文件和本文的 PDF 版本。

总之，从上一章我们就知道这个 crackme 不是一个硬骨头，不过这里我打算对它做高级分析，也为将来的教程做准备。现在坐好，准备一杯咖啡/香烟/巧克力棒/注射器，任何能让你坚持下去的东西都行，那咱们开始了.....。

二、破解的等级

逆向工程领域（尤其是破解）中关于打补丁的不同级别有几个不成文的规则。基本上可以分为四级（我保证，至少有一半的逆向工程师会因为那个数字和我吵起来）。当然，因为缩写神马的听起来都不错，所以我给四个级别的每一个都想了一个缩写。事不宜迟，下面就是补丁级别的介绍及具体意义：

级别一：LAME

LAME 方法，就是 Localized Assembly Manipulation and Enhancing，这个方法目前我们已经学习过。意思是找到代码中的第一个魔术 比较/跳转 指令，然后将其 NOP 掉或强制它跳转。到目前为止，这个方法都很神奇的好用。当然，我们都是在简单的 crackme 上做实验（有一半都是我专门为教程写的）。不幸的是，外面的大部分应用都不会这么简单。用 LAME 方法，有许多东西都会出问题，包括：

1、许多许多的应用会在程序的不同地方对程序是否已经注册进行检测，所以如果你仅仅打了一个补丁的话，并不意味着就没有其他的地方需要打补丁（我想我见过最多的分布检测点是 19 个）。并且有时候这些其他的检测点并不会起作用，除非某些特定的事件发生，所以你会发现自己又得回头对同一个程序进行搜索，以找到替代检测点并打补丁。

2、许多程序也会采用多种特别的技巧以避免 比较/跳转 指令组合的暴露。无论是在 DLL 中执行、在另一个线程中执行还是以多态的方式修改，都有许多种方法来实现。

3、有时候你将会修补大量的代码。你有可能会给七个检测点打补丁，将其他的检测点 NOP 掉等等。这会让你头昏脑涨的，而且对你来说也不是那么的优雅。

4、使用该方法你不需要学习太多东西，如果你正在阅读本系列教程，很可能是因为你对相关主题感兴趣并有学习的欲望。

尽管如此，有时候最优雅的解决方案，通常也是最简单的解决方案，仅仅是一个 比较/跳转 指令组合的补丁即可，所以别让我走错路并认为你不应该使用它。事实上，我逆向过的许多程序中，我猜大概有 25-40%就是用像这样的一个简单补丁搞定的。所以它是一个强大的方法。

级别二：NOOB

NOOB 方法，也就是 Not Only Obvious Breakpoints 方法，通常要比 LAMP 方法更深入一步。它通常涉及到要单步步入到 比较/跳转 指令组合的前面的那个 CALL，以了解是什么让 比较/跳转 指令组合决定走这条路的。这样做的好处是，你将有更多机会捕获到调用相同方法进行注册验证的其他部分代码，所以给一处打补丁就可以真正的补好几处，也就是所有调用相同注册验证方法的那几处。当然，该方法也有几个缺点，比如:

1、有时该方法用于超过一个注册验证的程序。例如，有一个用于比较两个字符串的通用函数，它返回真或假。在我们序列号匹配的案例中，这就是打补丁的地方，不过同样的方法被调用以比较两个不同的字符串，并且我们已经将其打过补丁以让它始终返回 true(或者视情况也有可能是 false)结果会怎样?

2、该方法需要更多的时间和实验，以判定能够返回正确值的最好选择是什么。这个需要时间和技巧。

这是本章中我们将会用到的第一个方法。

级别三：SKILLED

SKILLED 方法，也就是 Some Knowledge In Lower Level Engineered Data 方法，和 NOOB 方法有点像，除了它需要你仔细审查程序并且将其完全逆向以研究到底是什么情况。这样做有许多好处，比如理解所使用的任何技巧（像在内存中存储变量以便于后面获取），提供更多的打补丁的地方以更简单并且少侵入，从内部了解程序是如何工作的。它也给了你作为一个逆向工程师在将来会用到的许多知识，更不用说你的汇编语言技能。

该方法的主要缺点是，它更难并且需要更多的时间。我建议你至少找几个程序试试这个方法，因为没有什么能够比花时间深挖代码、堆栈、寄存器以及内存能够让你成为更好的逆向工程师，尝试去感受下作者曾经试过的。本章的最后我们将会用到这个方法。

级别四：SK1LL$

思考下破解的圣杯，Serial Keygenning In Low-level Languages, Stupid 意味着你不仅要仔细研究并且准确找出注册进程是如何执行的，还要重建它。这就得能够让新用户随意输入任何用户名，然后 keygen 者的代码能够算出对于该二进制文件管用的序列号。制作一个 keygen 的通常的方法是用程序自身的代码来对付它，意思是拷贝作者用来解密序列号的代码并用它来进行加密。这些代码通常放置在某种专门用来接收被拆分的代码的程序中（它提供有 GUI 等类似功能）。

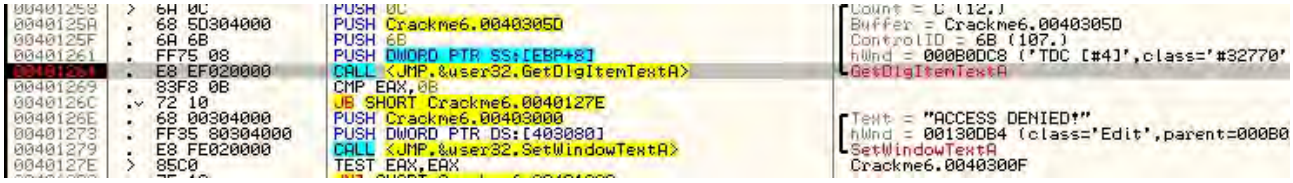
skill$（译者注：小标题中的第三个字母为 1，这里又变成 i，我不知道是不是作者故意的）的最高境界是，如果不能够从应用中提取代码，就必须自己编写代码来提供可用的序列号。意识是你必须完全理解程序是如何解密序列号并将其与你输入的进行对比。你必须自己写程序来完成相同的功能，仅在逆向领域中，有很多次是用汇编语言写的。

很明显，该方法的主要缺点是 skill$的复杂难懂。

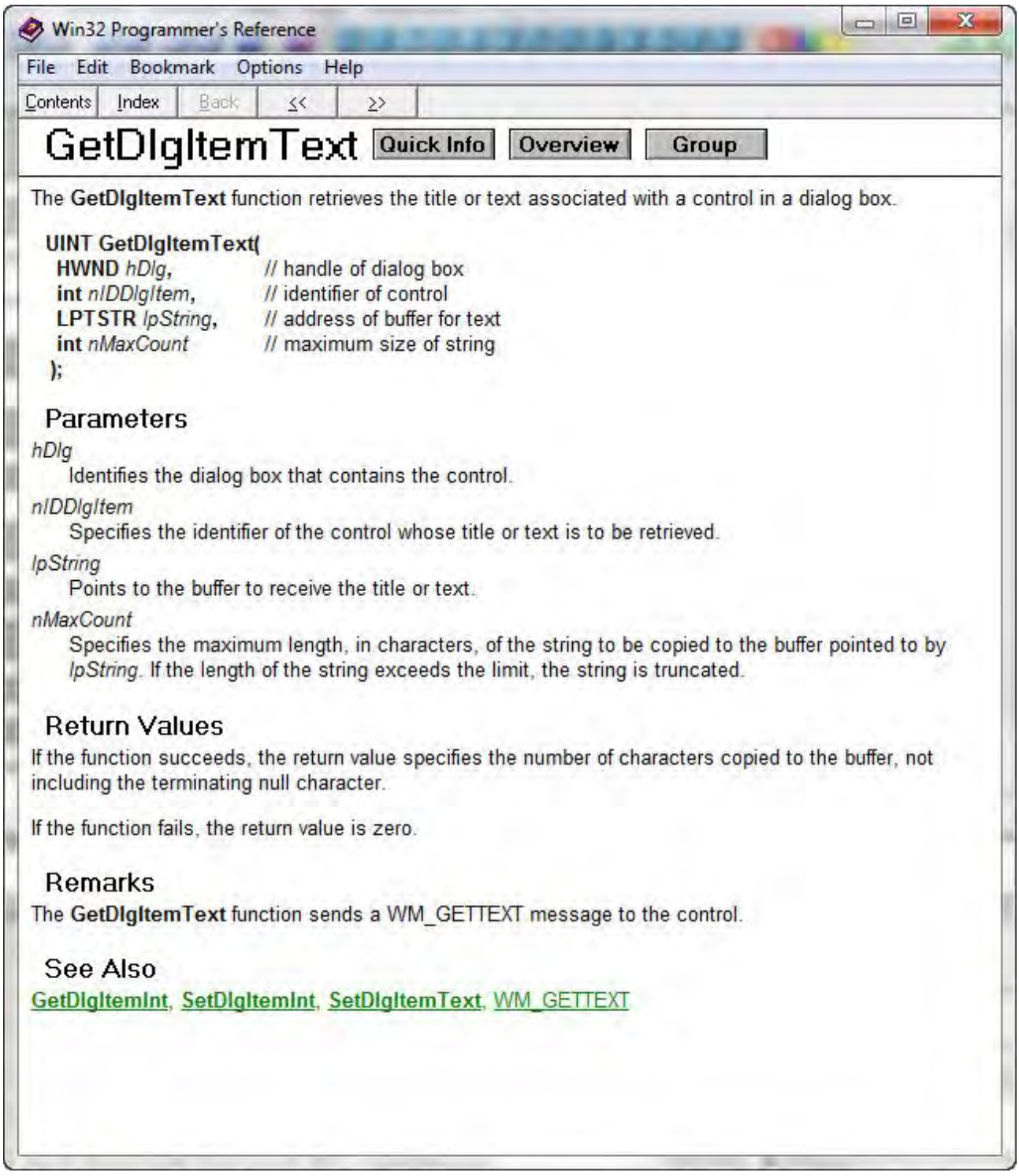
那么，鉴于我们对逆向工程级别的新的理解.....

三、用级别二来研究应用程序

重启应用并运行。给 GetDlgItemTextA 设置断点（参见上一章），输入密码（我输入的是“12121212”）然后点“Check”，Olly 就断在了 GetDlgItemTextA:



咱们来看看 GetDlgItemTextA:

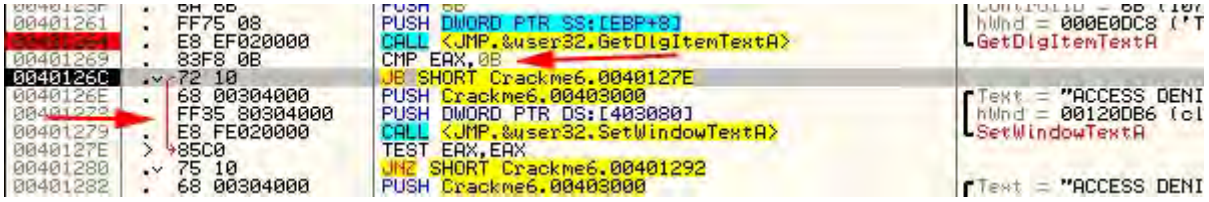


需要重点注意的是：其中一个参数是一个指向缓冲区的指针，该缓冲区是用来存储密码的（lpString）。返回值保存在 EAX 中，它保存的是字符串的长度：

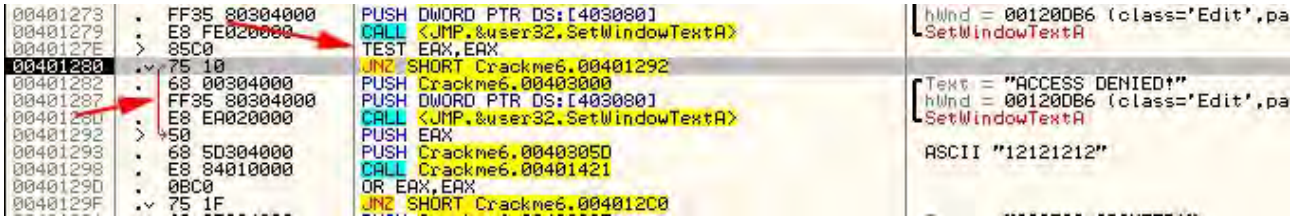
> **Return Values**
> If the function succeeds, the return value specifies the number of characters copied to the buffer, not including the terminating null character.
> If the function fails, the return value is zero.

你在 40125A 处看到那个指向字符串缓冲区的指针，它是 40205D（011y 加了一个注释 “Buffer=”，因为它能够猜测参数。译者注：作者写的是 40205D，

不过看图片实际上是 40305D，估计作者弄错了）。意思是该函数会拷贝我们的对话框文本到一个以 40305D 开始的 buffer 中，将返回的字符串的长度保存在 EAX 中。所以，在本例中，我们输入的密码“12121212”将被获取到，返回的密码长度保存在 EAX 中，这里是 8。现在，如果你看接下来的两行，你会发现这个值与 0x0B（十进制的 11）进行比较，并且如果 EAX 比它小的话程序就会跳转。真正的意思是，如果我们的密码长度（EAX）小于 0x0B（11 个数字）就会跳转。注意如果我们不跳的话，我们就会直接到坏消息那，所以实际上，这就意味着我们的密码长度必须比 11 小:



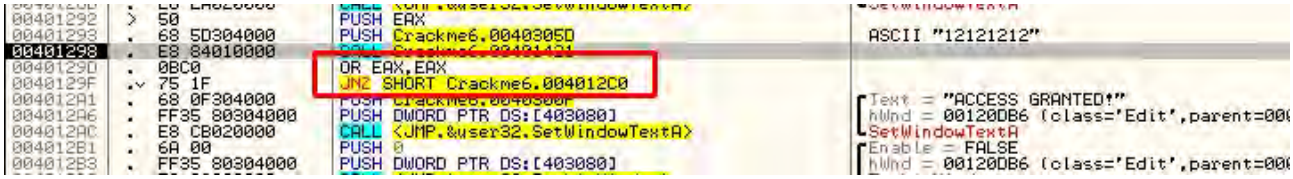
看吧!! 咱们已经了解了一些东西了，我们的密码最多只能有 11 个数字。因为我们的密码少于 11 个数字，所以咱们继续并让跳转实现。（如果你输入的密码大于 11 个数字，重启应用然后再输入一个小于 11 个数字的密码，再单步到我们所在的位置。）



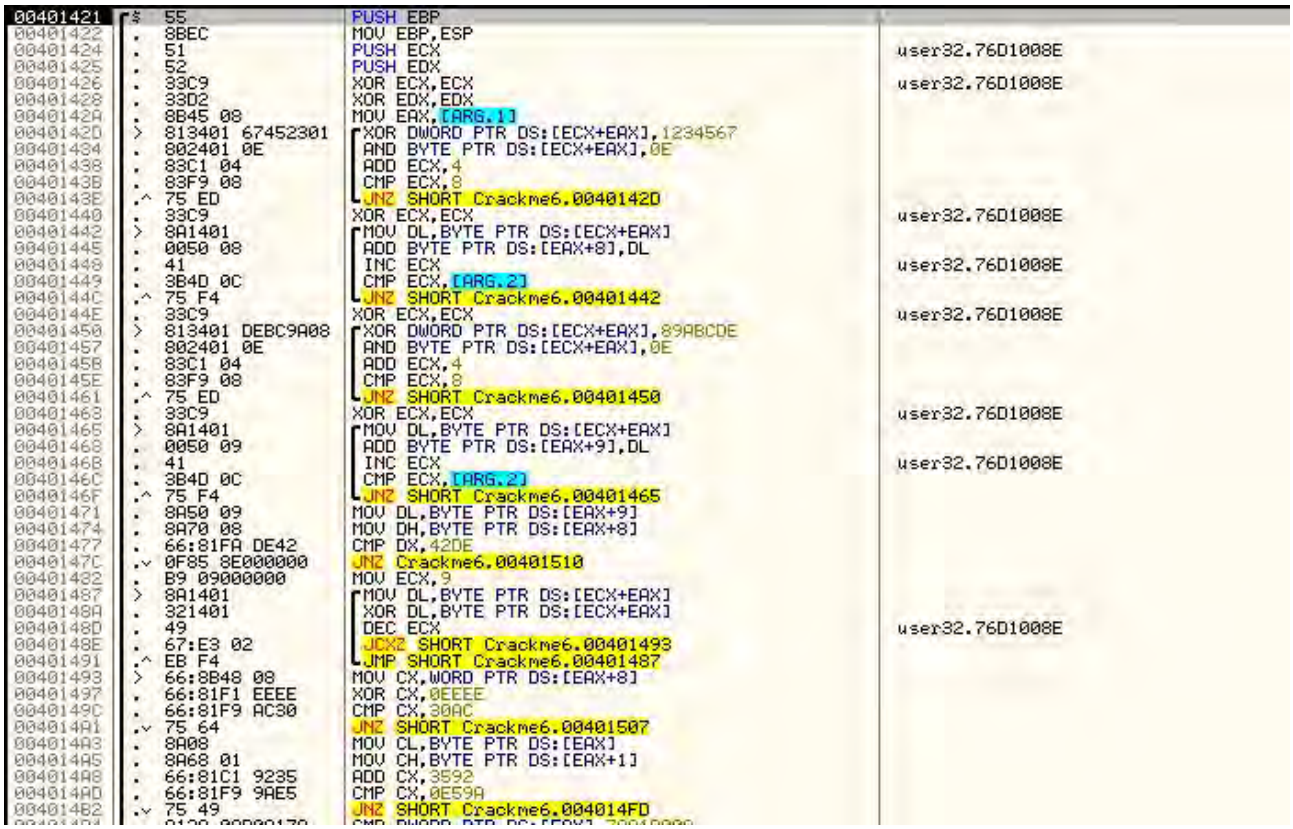
接下来你注意 EAX 的值，它仍然保存着密码的长度，并被测试是否为 0，如果不是 0 的话，就跳过第二个坏消息。那么现在我们知道了第一个坏消息是当我们的密码长度小于 11 时显示，第二个坏消息是在密码为空的时候显示。

注意，在跳转实现的接下来的两行，是从 401282 开始的，PUSH EAX（密码长度），地址 40305D（存储密码的 buffer）入栈。看看堆栈，可以看到确实如此:

首先要注意的是（在地址 18FAB0 处）是长度（8）被压入栈，其次是在地址 18FAAC 处的地址 40305D 被压入栈，Olly 也向我们显示了“12121212”也就是我们的密码。现在我们知道了，我们的密码是存储在内存的 40305D 处。这一点在后面会很重要。后面，Olly 会将这两个值叫做 ARG.1 和 ARG.2，因为它们是传递给函数的参数。这两个值入栈以后，我们就可以调用 401298 处的主要注册程序了（我们之所以知道这个，是因为所有重要的 比较/跳转 指令组合的前面都有个 CALL，所以它的结果将决定我们是跳到好消息还是坏消息）:



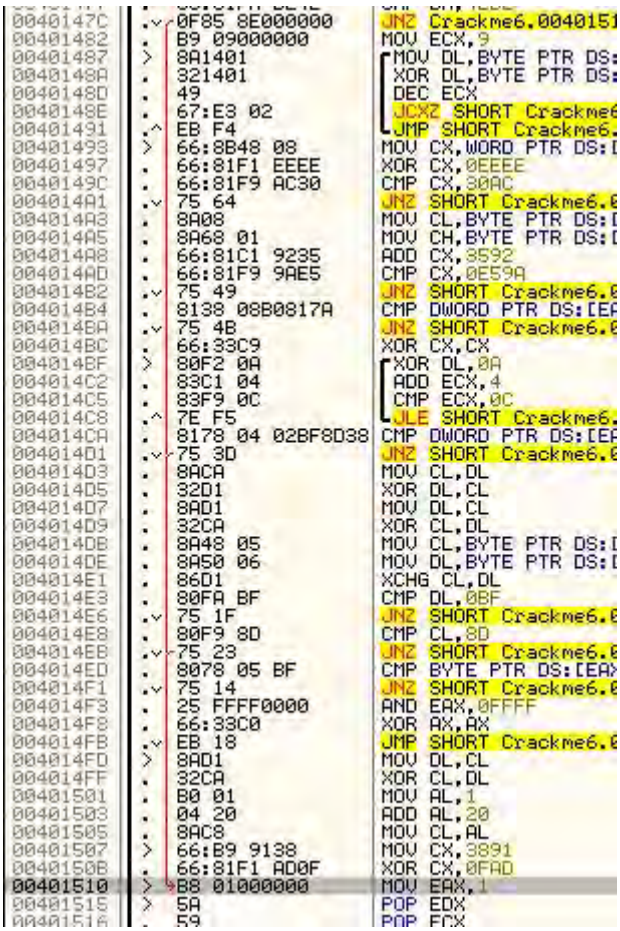
让 O11y 就暂停在 CALL 那行，不过要注意 CALL 后面那行，40129D 处指令对 EAX 自身做了 OR 操作（该操作会根据 EAX 是否为 0 来设置 0 标志位），如果 EAX 不是 0 的话就会跳过好消息。这就意味着将在 401298 处调用注册程序，并在某个时刻在 EAX 中保存一个值并将该值 RETN。返回值将会被检查是否为 0，如果不是就显示坏消息。所以我们必须保证在这个 CALL 中，当它返回时 EAX 等于 0!! 如果我们能够做到的话，它就是我们需要的唯一一个补丁（密码被限制在 0 到 11 个数字之间也算一个，不过那是一个简单的补丁）。咱们继续，单步步入到 401298 处的注册程序，总览一下：



哇噢，看起来不少呢，尤其是你很可能对汇编语言只是个半吊子。但也不是不可能。我常用的招是到程序的最后面，我们知道它返回时 EAX 必须等于 0，看看是什么完成了这项工作以及是什么阻止了它发生，然后再回头用我们的方法。向下滚动直到你看到函数的 RETN 指令：

这里，我们可以看到，我们肯定是想要在函数返回前避免 401510 处的指令将 EAX 的值设置为 1。你可以看到有一个红色箭头指向该行（译者注：不是作者加的箭头，是那个细线的小箭头，在指令的边上），所以该跳转也需要被干掉。现在如果我们向上看看，我们能够看到 EAX 被设置为 0 的地方，也能看到函数底部将其返回的路径：

如果我们看看 4014FB 那行，EAX 将会被置 0（对自身做 XOR 操作），跳转指令将会跳过 401510 处的坏消息指令，相关的执行流将会返回 EAX 值为 0😀。现在我们跟一跟我们看到的第一个跳转（也就是会跳到 401510 处 MOV EAX,1 这个坏指令的跳转），看看从哪跳过来的：



40147C 就是那个坏跳转。我们想要阻止它跳，否则我们肯定会得到坏消息。好的，我们现在已经有了关于这段程序的基本的知识，对于级别二的破解我们就到这里，现在来打个补丁确保 EAX 总是返回 0。你会怎么做呢？我准备将它留给你来做（这是本章结尾的作业🙂）。放心好了，我会给你答案的....。不过你要明白这个级别的补丁已经开始时的补丁要好很多，一是我们只打了一个补丁，二是如果这段程序被应用中的其他部分调用的话，我们仍然能够获得好消息😀。

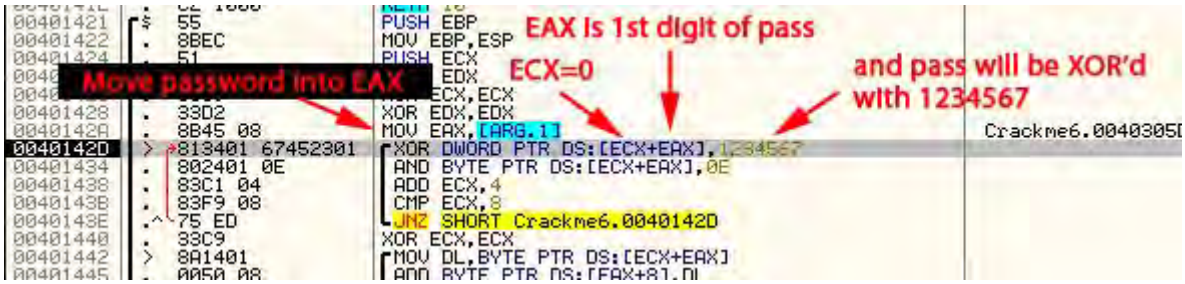
现在，先停一会，考虑考虑你怎么来打这个补丁。记住，EAX 必须返回 0。我让你做的原因是，有很多很多 NOOB 补丁可以完成这个任务，我想让你开始像一个逆向工程师那样思考！如果你需要提示，那就看看结尾的作业那一块。如果你能够解决，那你就是一个真正的 NOOB!!!

当你完成时，就准备转到更详细的分析，继续阅读吧.....。

四、步入到级别三

我知道你仍然是一个初学者，不过我还是想让你尝尝更深层次补丁的感觉。如果你还没有准备好，或者完全失败了，也别气馁。这里只是给你一个想法。我们会在将来的教程中学习这一块的所有东西。你可能会问，更加深入代码的目的是什么，应用程序中调用这段程序的每一个地方都会被打上补丁吗？好吧，对于新手来说，假如有不同程度的注册会怎么样，比如“Private”、“Corporate”，“Enterprise”....。程序可能会根据内部的逻辑来做决定。另一个你想要更深入学习的原因是，为它做一个 keygen。你需要理解代码才能做。现在，咱们开

始打一个 SKILLED 级别的补丁。回到上面程序（译者注：这里的程序是指那个验证函数，本章大部分都是这个意思，读者应自己做分别）的起始处，实验一下：

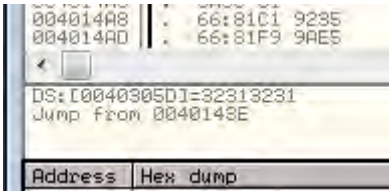


首先，那有一些典型的寄存器的压栈操作以及在栈中为本地变量开辟空间的操作。ECX 和 EDX 中的值被压栈，然后我们就可以在不用覆盖这些寄存器的情况下使用它们（函数返回时会将这些值出栈以将它们还原。译者注：这就是传说中的堆栈平衡，脱壳中 ESP 定律的原理）。然后我们就到了 40142A，这里将栈中（我们输入的密码的地址）的本地变量拷贝到 EAX 中。如果你看寄存器窗口就会发现 EAX 的值是地址 40305D，也就是我们密码所在的地址。下一行代码：

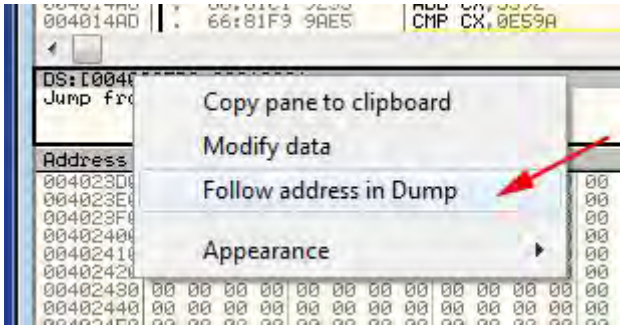
```
XOR DWORD PTR DS:[ECX+EAX], 1234567
```

该行的意思是，将 ECX（它的值是 0）和我们的密码首地址（密码存储在 40305D，记不记得？）相加，然后从该位置取 DWORD（4 字节）数据与十六进制的 1234567 进行 XOR 操作。因为 ECX 是 0，将其与我们的密码首地址进行相加不会有任何的影响，所以我们从密码的第一个数字的地址开始处理就行。简单点来说，这行代码的意思是“取密码的前四个字节与 1234567 进行 XOR 操作，将新值存储到内存中与我们密码同样的位置”。

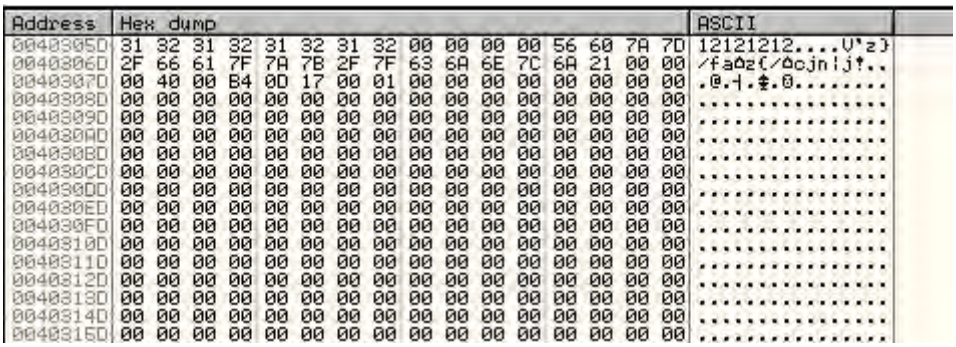
我们可以观察到这一过程。首先，要确保我们依然暂停在 40142D 那行，看数据窗口的上面那一块，它会告诉你地址 ECX+EDX（40305D）是什么，以及它存储的值是什么（32313231），用 ASCII 码表示就是“2121”（要记得数据存储序列😀）：



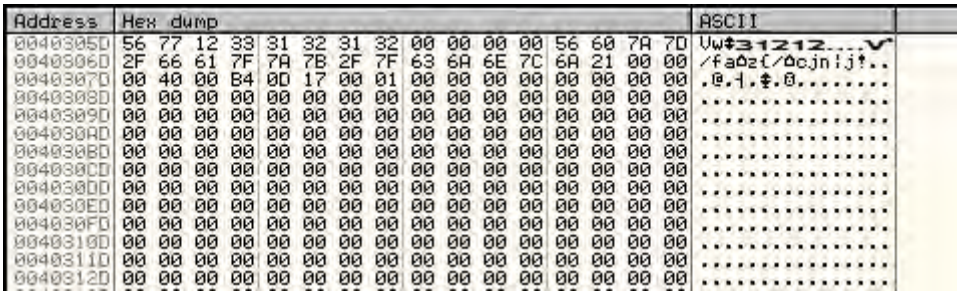
现在选中“DS:[0040305D]=32313231”这行，右键选择“Follow in dump（数据窗口中跟随）”，然后我们就能看到我们密码当前存储的实际内存内容：



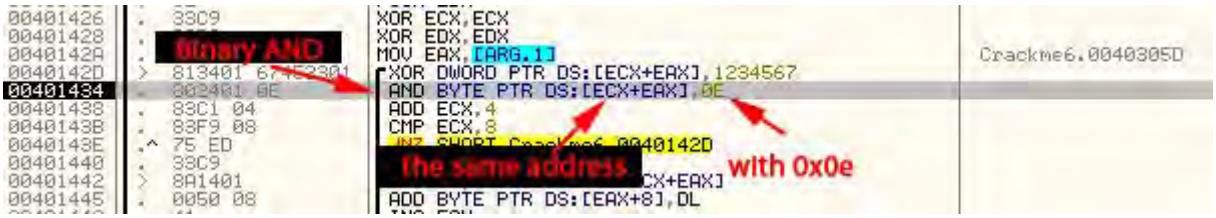
现在，数据窗口显示的就是从 40305D 开始的内存内容。前面 8 个字节就是我们的密码。记住，我们当前所在的行正准备取该地址的前四个字节（31，32，31，32），然后与 0x1234567 进行 XOR 操作，之后再将结果存回到该内存区：



咱们继续，单步步过一次，然后你就会看到我们密码的前四个字节已经变了，和 0x1234567 进行 XOR 的结果：



好，继续下一行代码：

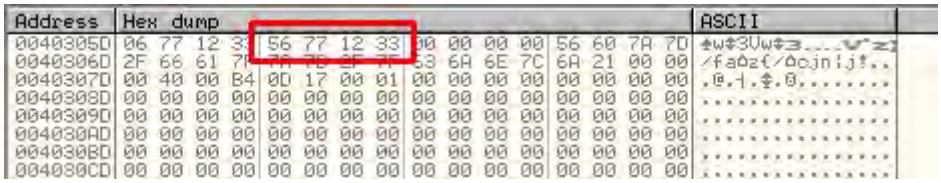


该行是 AND BYTE PTR DS:[ECX+EAX], 0E。我们已经知道了 ECX+EDX 的结果是 40305D，也就是我们以前密码的地址。现在，我们准备按 BYTE 与 0x0E 进行 AND 操作，并将结果存回该地址。这意味在，我们存储在 40305D 的以前的密码的第一个数字（译者注：这里我感觉作者的表述不太准确，因为存储在 40305D 的是 XOR 后的数据，早不是我们输入的 12121212 了，大家要注意，仔细观察数据区。）在与 0E 进行 AND 操作后再存回第一个位置。看看数据窗口的帮助区域已经显示出来了：

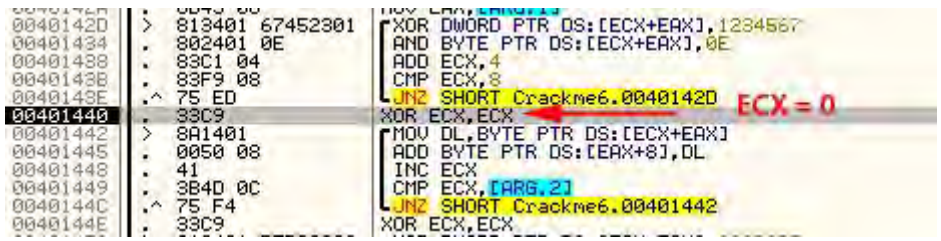
它告诉我们将受到影响的地址是 40305D，该地址的（当前）值是 56。继续，单步执行一次，你会发现第一个数字又变了：

现在咱们知道了 0x56 与 0x0E 进行 AND 操作的结果是 0x06。咱们继续跋涉这段困难的代码：

ECX 增加了 4（指向下面的四个字节），并和 8 进行比较。意思是这个循环会运行两次，第一次 ECX 等于 4，第二次等于 8，然后跳出循环。这意味着我们总共处理了 8 个字节。所以第二次循环时，我们将影响第二个 4 字节，也就是将它们与 0x1234567 进行 AND 操作。你在单步运行时，注意观察第二个 4 字节：



它们也会被修改。第五字节也会被再次修改，因为它将与 0x0E 进行 AND 操作。循环结束后，下一条指令也就是 401440 处的指令仅仅是将 ECX 值置 0：

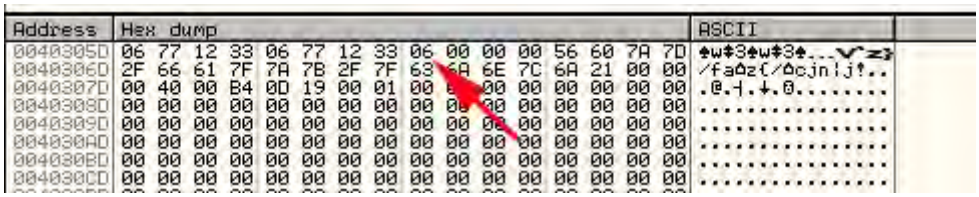


现在咱们来看看接下来的几条指令：

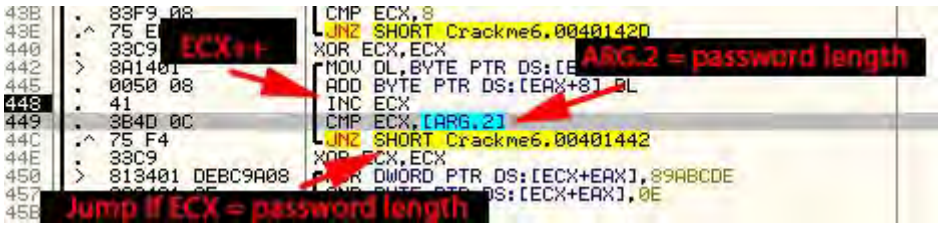
首先，我们将我们（旧）密码的第一个（新的）字节拷贝到 DL 中（因为 ECX 再次被置 0，所以我们正在处理的是第一个数字，也就是 EAX 当前指向的数字）。如果你看寄存器窗口，你会发现第一个字节（0x06）在 EDX 寄存器中：

然后我们将 DL 中的数字与 EAX+8 中的内容相加，也就是 EAX 的第八个字节，并将结果存回第八个字节：

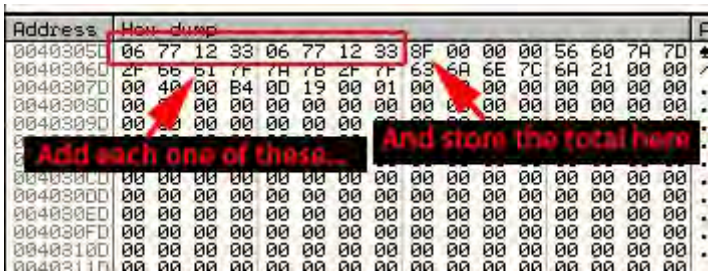
这里，我们能够看到那个字节被修改了：



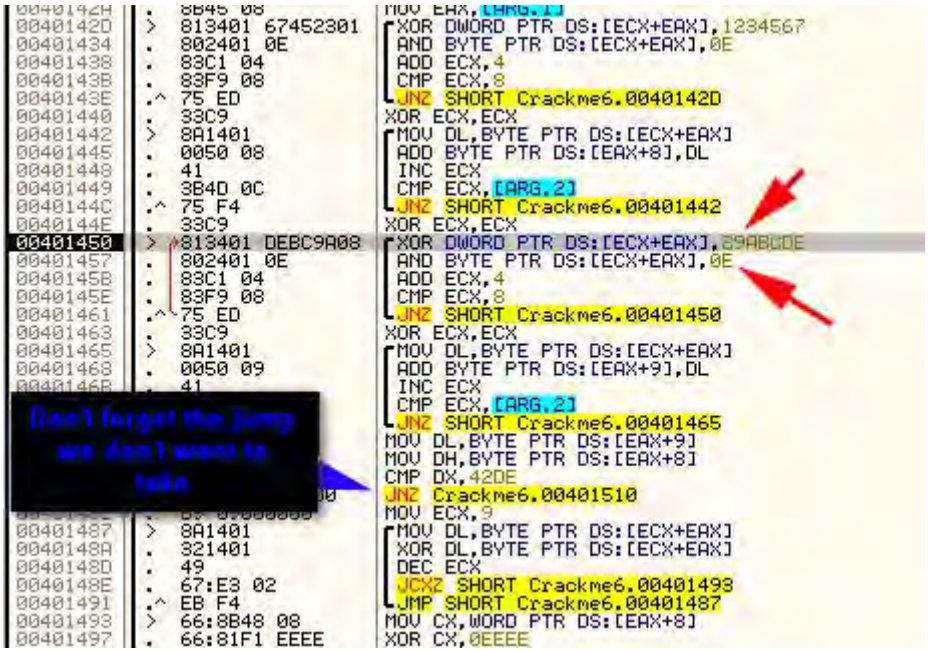
它是将 buffer 中的第一个字节（6）与第八个字节（0）相加，结果得 6（译者注：不知道读者注意到没有，上面的图片中箭头明明指的是第九个字节好不好，那是不是作者弄错了呢？我觉得是有问题的，因为如果按从 0 开始数，那么第一个字节就应该是 77，如果不从 0 开始数，那么第八个字节就应该是 33。那该怎么办呢？其实不用管第几个了，我们知道定位到该字节是按 EAX+8 的结果算的，因为 EAX 的值是 40305D，所以 40305D+8=403065。40305D 就是 06 那个字节，我们往右边数，数到 403065，刚好就是箭头指的那个 06）。如果我们密码的长度大于 8，这就会让我们密码的第一个字节与我们密码的后面的那个数字相加，不过因为我们的密码只有 8 位，所以这块内存被设为 0。接下来我们给 ECX 加 1（因此转移到下一个字节），将其与密码的长度进行比较。这只是查看我们是否已到达结尾。如果没有，就跳转到循环的开头再执行一次。这基本上意味在，我们将循环遍历密码的所有数字，每个数字相加然后将结果存储在第八个内存位置。现在我们明白了为什么密码只能是 11 个数字，所有空间加上 0 终止符一共只能接受 11 个字符。



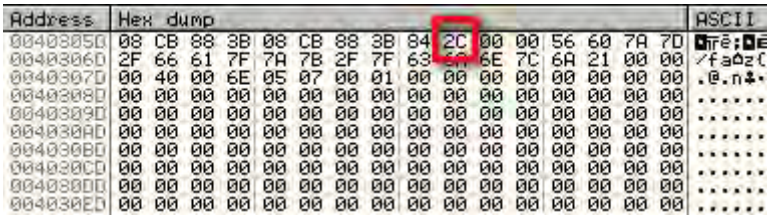
你单步执行这个循环，就可以观察到内存的变化：

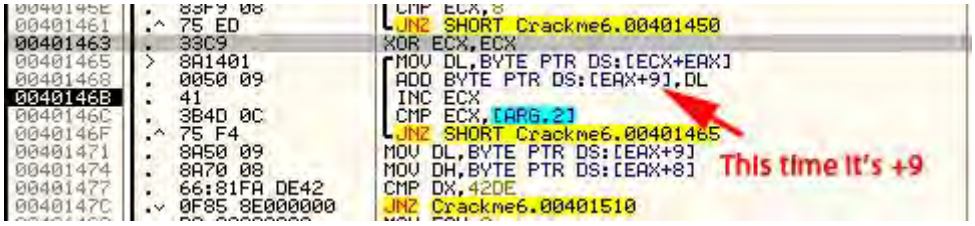


在循环结束后，我们再一次将 ECX 设置为 0，并进入了与第一个循环相似的循环中，这次将每个四字节数据与 0x89ABCDE 进行 XOR 操作。



它也将所有的字节相加，再将结果保存在第九字节。这个循环将会一直执行，直到 ARG.2 等于 0 为止。ARG.2 是我们密码的长度（还记不记得它是调用该函数前第二个被压入堆栈的？）所以，这些指令将会执行 8 次，密码中的每个数字一次。在执行完这段代码后，你就会看到最终的结果：





运行程序并观察所有的情况是非常非常的重要，因为这能让整个过程更加清晰很多。花点时间来理解每一行，看看它将做什么，它准备将结果存储在什么地方。你会发现，它其实不像听起来的那样难:)。别忘了，我们将要在 40147C 处的跳转做出我们的选择。下面我们总结下我们所做的:

1、我们将我们的密码的每一个四字节值与 0x1234567 进行 XOR 操作，再将结果覆盖回我们的密码。

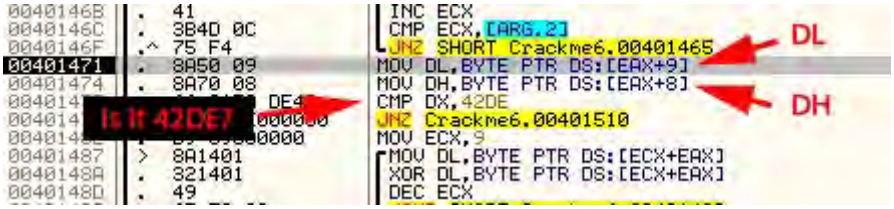
2、第一个字节与 0x0E 进行 AND 操作，第五个字节也是一样。

3、然后我们将所有的字节值加起来，将结果存储在第八字节。

4、然后，我们再将 buffer 中的每个四字节值与 0x89ABCDEF 进行 XOR 操作，再将结果存进这个 buffer。

5、我们再一次将 buffer 中的内容相加，将结果存储在第九个内存位置。

我们已将执行了此 crackme 保护机制魔法的大部分（*啧啧*）。现在我们将载入这两个值（buffer 内存内容的求和），一个在 EAX+8，另一个在 EAX+9，分别载入到 DL、DH，本例中 EDX 的结果就是 842C。然后，我们将这两个值与 42DE 进行比较:



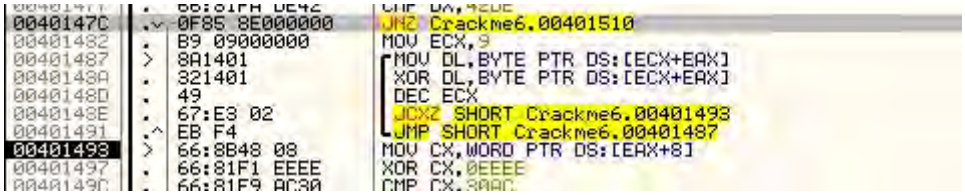
为啥是 42DE 呢？好吧，这很可能是一个硬编码的密码。你思考下，如果你有一个特殊密码，用它来进行整个的 XOR 和 AND 操作，将得到魔数 42DE。我们的例子中，EDX 等于 842C:

我们没有输入那个魔术密码，所以我将实现该跳转，跳到坏消息代码那里:

```
00401474  .  8A70 08          MOV DH,BYTE PTR DS:[EAX+8]
00401477  .  66:81FA DE42     CMP DX,42DE
0040147C  .v 0F85 8E000000    JNZ Crackme6.00401510
00401482  .  B9 09000000      MOV ECX,9
00401487  >  8A1401           MOV DL,BYTE PTR DS:[ECX+EAX]
0040148A  .  321401           XOR DL,BYTE PTR DS:[ECX+EAX]
0040148D  .  49               DEC ECX
0040148E  .  67:E3 02         JCXZ SHORT Crackme6.00401493
00401491  .^ EB F4            JMP SHORT Crackme6.00401487
00401493  >  66:8B48 08       MOV CX,WORD PTR DS:[EAX+8]
00401497  .  66:81F1 EEEE     XOR CX,0EEEE
0040149C  .  66:81F9 AC30     CMP CX,30AC
004014A1  .v 75 64            JNZ SHORT Crackme6.00401507
004014A3  .  8A08             MOV CL,BYTE PTR DS:[EAX]
004014A5  .  8A68 01          MOV CH,BYTE PTR DS:[EAX+1]
004014A8  .  66:81C1 9235     ADD CX,3592
004014AD  .  66:81F9 9AE5     CMP CX,0E59A
004014B2  .v 75 49            JNZ SHORT Crackme6.004014FD
004014B4  .  8138 08B0817A    CMP DWORD PTR DS:[EAX],7A81B008
004014BA  .v 75 4B            JNZ SHORT Crackme6.00401507
004014BC  .  66:33C9          XOR CX,CX
004014BF  >  80F2 0A          XOR DL,0A
004014C2  .  83C1 04          ADD ECX,4
004014C5  .  83F9 0C          CMP ECX,0C
004014C8  .^ 7E F5            JLE SHORT Crackme6.004014BF
004014CA  .  8178 04 02BF8D38 CMP DWORD PTR DS:[EAX+4],388DBF02
004014D1  .v 75 3D            JNZ SHORT Crackme6.00401510
004014D3  .  8ACA             MOV CL,DL
004014D5  .  32D1             XOR DL,CL
004014D7  .  8AD1             MOV DL,CL
004014D9  .  32CA             XOR CL,DL
004014DB  .  8A48 05          MOV CL,BYTE PTR DS:[EAX+5]
```

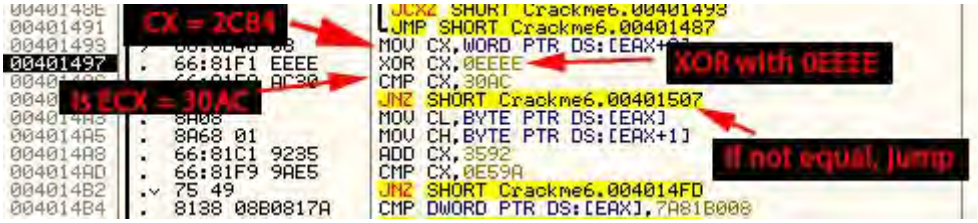
当然，除非我们给 O11y 帮点小忙：

因为我们不空降，所以 EAX 的值不会被置 1，并且该函数会立即终止。下一步，我们将 ECX 值置为 9，以便于我们访问 buffer 的第九个数字，将第九处内存位置的内容拷贝到 DL 中（这里是 0x2C），对其自身做 XOR 操作（让其等于 0），ECX 减 1 以指向前一个位置，这样做 9 次：



```
0040147C  .v 0F85 8E000000    JNZ Crackme6.00401510
00401482  .  B9 09000000      MOV ECX,9
00401487  >  8A1401           MOV DL,BYTE PTR DS:[ECX+EAX]
0040148A  .  321401           XOR DL,BYTE PTR DS:[ECX+EAX]
0040148D  .  49               DEC ECX
0040148E  .  67:E3 02         JCXZ SHORT Crackme6.00401493
00401491  .^ EB F4            JMP SHORT Crackme6.00401487
00401493  >  66:8B48 08       MOV CX,WORD PTR DS:[EAX+8]
00401497  .  66:81F1 EEEE     XOR CX,0EEEE
```

你可能有点疑惑，这没有改变 buffer 中的任何东西呀，那这个函数有什么意义呢？好吧，咱俩都被迷惑了。看起来它所做的一切都是让 DL 一次又一次的等于 0，这看起来几乎就是代码中一个圈套（或者是一个错误😃）。总而言之，不管这个代码运行还是不允许，这都没有什么不同，所以它就是死代码。我们现在来到一组短点的代码，基本上是将 EAX 与 30AC 进行比较：



首先，它将我们前面求的和存在 ECX 中（第九处内存位置是 0x2C，第八处内存位置是 0x84），将其与 0xEEEE 进行 XOR 操作，再与 30AC 进行比较。因为 ECX 不等于 30AC，所以我们将跳转：

```
Registers (FPU)
EAX 0040305D Crac
ECX 0000C26A
EDX 00008400
EBX 00000001
ESP 0018FA9C
```

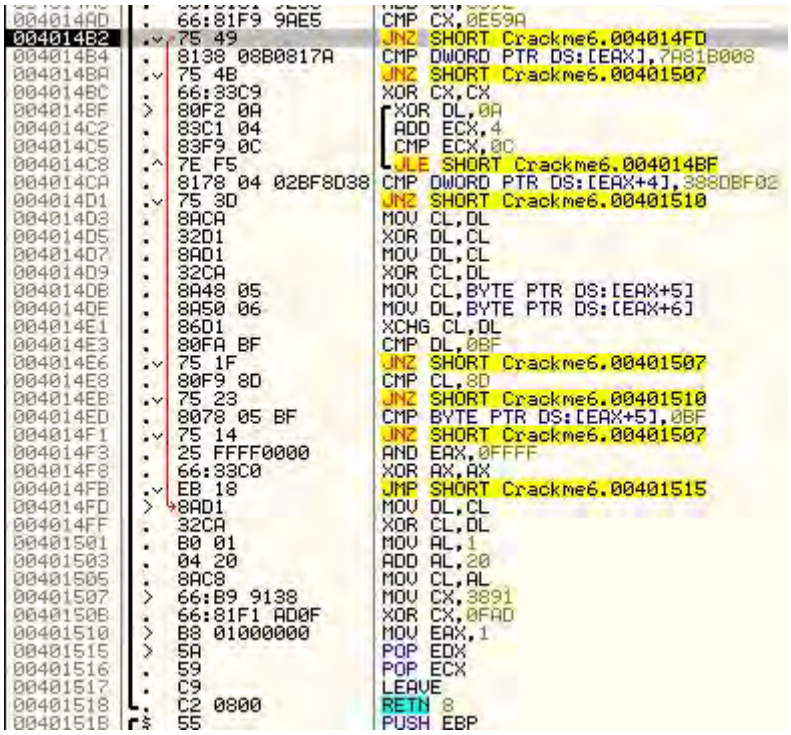
跳转到那里，ECX 再次被置为 1:

```
00401493  >  66:8B48 08        MOV CX,WORD PTR DS:[EAX+8]
00401497  .  66:81F1 EEEE      XOR CX,0EEEE
0040149C  .  66:81F9 AC30      CMP CX,30AC
004014A1  .v 75 64             JNZ SHORT Crackme6.00401507
004014A3  .  8A08              MOV CL,BYTE PTR DS:[EAX]
004014A5  .  8A68 01           MOV CH,BYTE PTR DS:[EAX+1]
004014A8  .  66:81C1 9235      ADD CX,3592
004014AD  .  66:81F9 9AE5      CMP CX,0E59A
004014B2  .v 75 49             JNZ SHORT Crackme6.004014FD
004014B4  .  8138 08B0817A     CMP DWORD PTR DS:[EAX],7A81B008
004014BA  .v 75 4B             JNZ SHORT Crackme6.00401507
004014BC  .  66:33C9           XOR CX,CX
004014BF  >  80F2 0A           XOR DL,0A
004014C2  .  83C1 04           ADD ECX,4
004014C5  .  83F9 0C           CMP ECX,0C
004014C8  .^ 7E F5             JLE SHORT Crackme6.004014BF
004014CA  .  8178 04 02BF8D38  CMP DWORD PTR DS:[EAX+4],388DBF02
004014D1  .v 75 3D             JNZ SHORT Crackme6.00401510
004014D3  .  8ACA              MOV CL,DL
004014D5  .  32D1              XOR DL,CL
004014D7  .  8AD1              MOV DL,CL
004014D9  .  32CA              XOR CL,DL
004014DB  .  8A48 05           MOV CL,BYTE PTR DS:[EAX+5]
004014DE  .  8A50 06           MOV DL,BYTE PTR DS:[EAX+6]
004014E1  .  86D1              XCHG CL,DL
004014E3  .  80FA BF           CMP DL,0BF
004014E6  .v 75 1F             JNZ SHORT Crackme6.00401507
004014E8  .  80F9 8D           CMP CL,8D
004014EB  .v 75 23             JNZ SHORT Crackme6.00401510
004014ED  .  8078 05 BF        CMP BYTE PTR DS:[EAX+5],0BF
004014F1  .v 75 14             JNZ SHORT Crackme6.00401507
004014F3  .  25 FFFF0000       AND EAX,0FFFF
004014F8  .  66:33C0           XOR AX,AX
004014FB  .v EB 18             JMP SHORT Crackme6.00401515
004014FD  >  8AD1              MOV DL,CL
004014FF  .  32CA              XOR CL,DL
00401501  .  B0 01             MOV AL,1
00401503  .  04 20             ADD AL,20
00401505  .  8AC8              MOV CL,AL
00401507  >  66:B9 9138        MOV CX,3891
0040150B  .  66:81F1 AD0F      XOR CX,0FAD
00401510  >  B8 01000000       MOV EAX,1
00401515  >  5A                POP EDX
00401516  .  59                POP ECX
00401517  .  C9                LEAVE
00401518  L. C2 0800           RETN 8
```

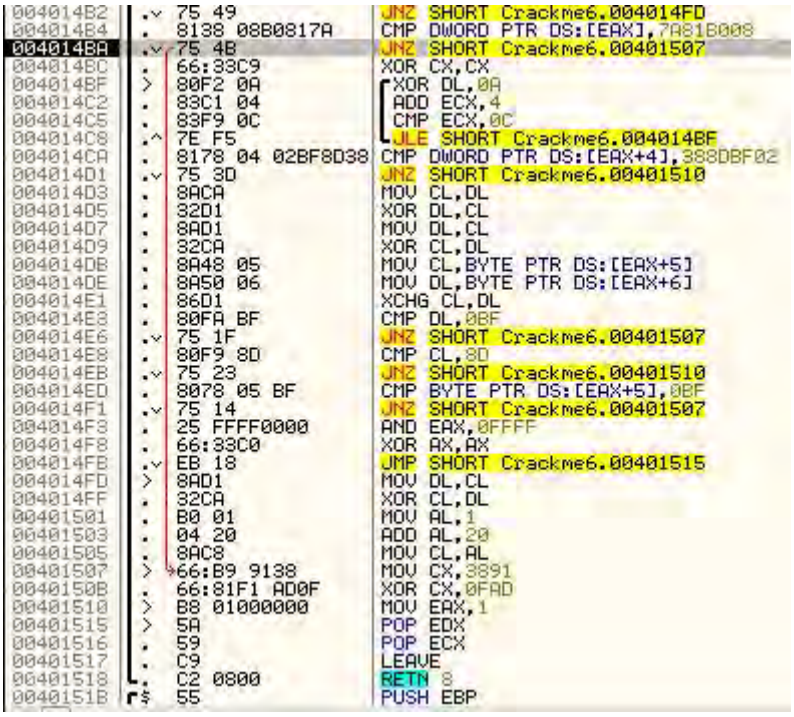
这基本上就是第二个密码检测点了。一个没有多少经验的逆向工程师（或刚好将 他/她 给难住了）很可能立即就将这个 JNZ 给打了补丁，原因是上面将我们转换的密码与 0x42DE 进行了比较。他们可能没有花时间来分析其他代码，认为这个补丁就够了。不幸的是，这个补丁明显不够，因为应用程序对我们的密码计算出来的值做了其他更多的操作，并且如果与新值不匹配的话就跳转。该方法多次被用来作为一种检测技术，检测是否有人尝试给应用程序打补丁：在没有任何补丁的情况下，如果我们的密码通过了检测并通过了第一个 JNZ，那我们也应该能通过第二个。如果没有，那么我们就知道有人给第一个打了补丁，所以我们就知道有人修改了代码。许多情况下，第二个跳转会跳到完全不同的代码块，有些看起来令人难以置信的复杂，但是实际上又什么都没做，最后就终止了。这是企图让逆向工程师做一些徒劳无功的事，让攻克保护机制变的更难。这不是我们想要的，所以我们设置 0 标志位，然后继续，我们遇到了下面两行代码:

```
004014A1  .v 75 64             JNZ SHORT Crackme6.00401507
004014A3  .  8A08              MOV CL,BYTE PTR DS:[EAX]      08
004014A5  .  8A68 01           MOV CH,BYTE PTR DS:[EAX+1]
004014A8  .  66:81C1 9235      ADD CX,3592                   CB
004014AD  .  66:81F9 9AE5      CMP CX,0E59A
004014B2  .v 75 49             JNZ SHORT Crackme6.004014FD
004014B4  .  8138 08B0817A     CMP DWORD PTR DS:[EAX],7A81B008
004014BA  .v 75 4B             JNZ SHORT Crackme6.00401507
```

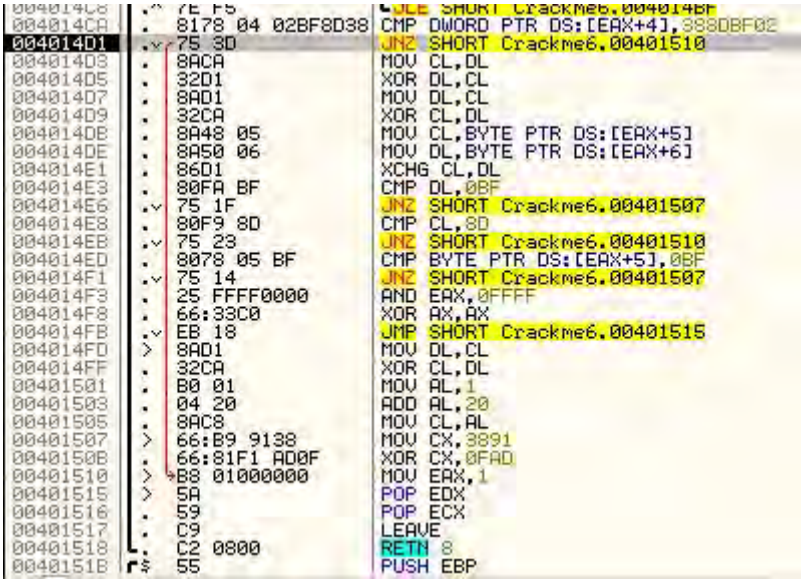
这是将我们密码 buffer 的第一和第二个内存内容拷贝到 CL 和 CH，本例中是让 ECX 等于 CB08。与 3592（十六进制）相加后再与 E59A 进行比较。如果它不等于该值就跳转:



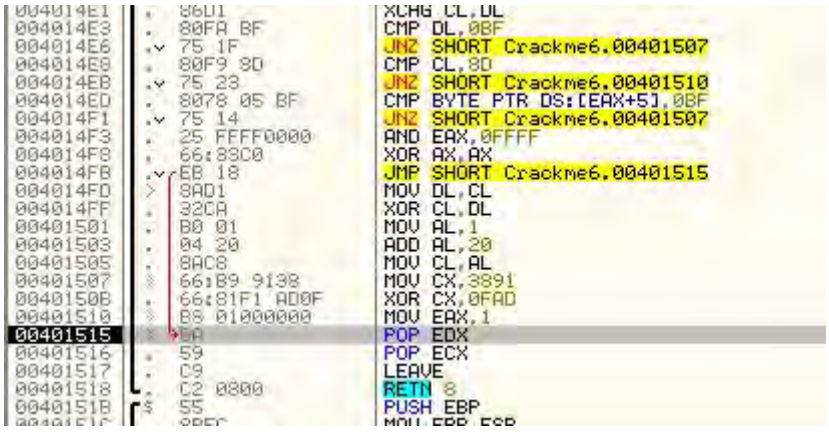
这个和上面做的事情是一样的。完成了另一个检测，以确保我们是合法的到达此处。我们明显不想让这个跳转实现，所以我们通过修改 0 标志位再次的帮了 O11y 的忙。然后我们顺利通过了另一个检测，这个是从 4014A3 到 4014AD。通过修改 0 标志位，我们也跳过了这个 JNZ，最终来到了这里:



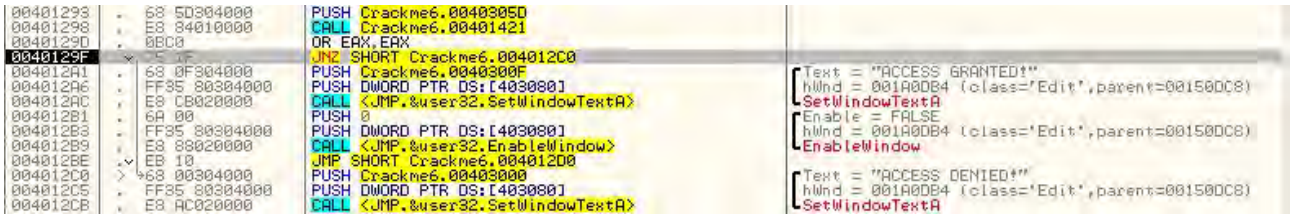
第一行代码 CMP DWORD PTR DS:[EAX], 7A81B008 做了另一个检测。在对密码做完了所有的操作以后 ，最后第一个四字节等于 7A81B008。如果不是，我们就会跳到坏消息那:



所以还是 0 标志位来帮 O11y 一把，然后我们就来到了另一个检测群（为什么不呢？），首先对接下来的几个字节做了一些操作，然后将其与 388DBF02 进行比较，并与各种内存中硬编码数字进行比较。这个在检测上有点矫枉过正了，不过我认为作者可能觉得检测越多就越能保护好 crackme。绕过所有的跳转，我们最后来到了我们想要的地方，就是那个 4014FB 的 JMP 指令:



如果我们单步通过 RETN，我们将来到熟悉的地方，不过这次有点儿不同:



注意这次我们来到了好消息这。这是因为我们组织应用程序将 EAX 设置为 1。

现在，你可能会认为“太棒了，我们在这个新的深入分析中，在级别二层面只用了一个补丁换来了 9 个（被设置 0 标志位的所有 JNZ）”，不过这却不是真正的情况。我不仅理解了它是如何工作的（并且对于将来的逆向挑战也赢得了大量经验），现在还能够打非常牢固的补丁，因为我们**知道**这个补丁无论在什么情况下都会起作用。有一点没有提，那就是找到这个软件的**真正**的密码其实不是很难，这样就绕过了任何需要打补丁的地方！这就是真正的逆向工程，它只能靠**大量的**练习。并且应用程序越难破解，你就越能够从代码中获取更多的细节。

再说一次，如果你失败了也不要担心。这次更多的只是提供一个相关方法的使用印象。我们将会再次的学习这些内容。同时呢，这里有一些...

五、作业

就是教程前面提到的，看你是否能够用 NOOB 技术给程序打补丁。这就意味着，找到一个方法步入对密码进行所有操作的 CALL，并找到一个绕过所有操作的方法。你不需要理解对密码做的所有操作，仅仅是找到一个让程序跳过它并且仍然能够得到好消息的方法。

如果你需要提示，请点击这里。

超级吊的加分题：你能够找到硬编码密码吗？

第十一章：用 NOOB 技术破解

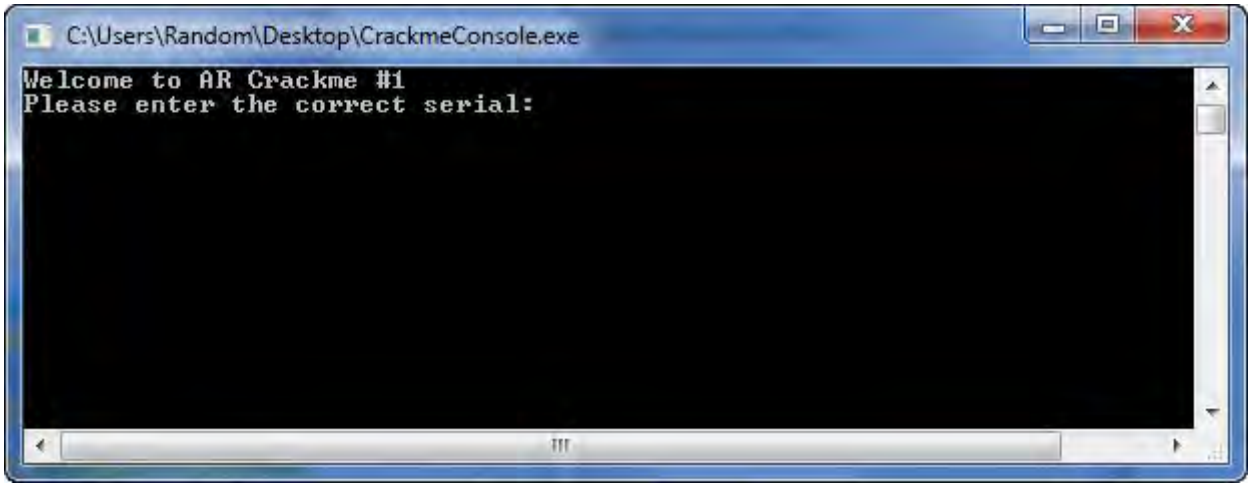
一、简介

本章我们将再次讨论补丁程序，不过比典型的单个“我们遇到的第一个补丁”要深入一点点。我们将从一个控制台程序开始，找到隐藏在其中的正确密码。教程的相关下载中有。除此之外，你只需要 OllyDbg。

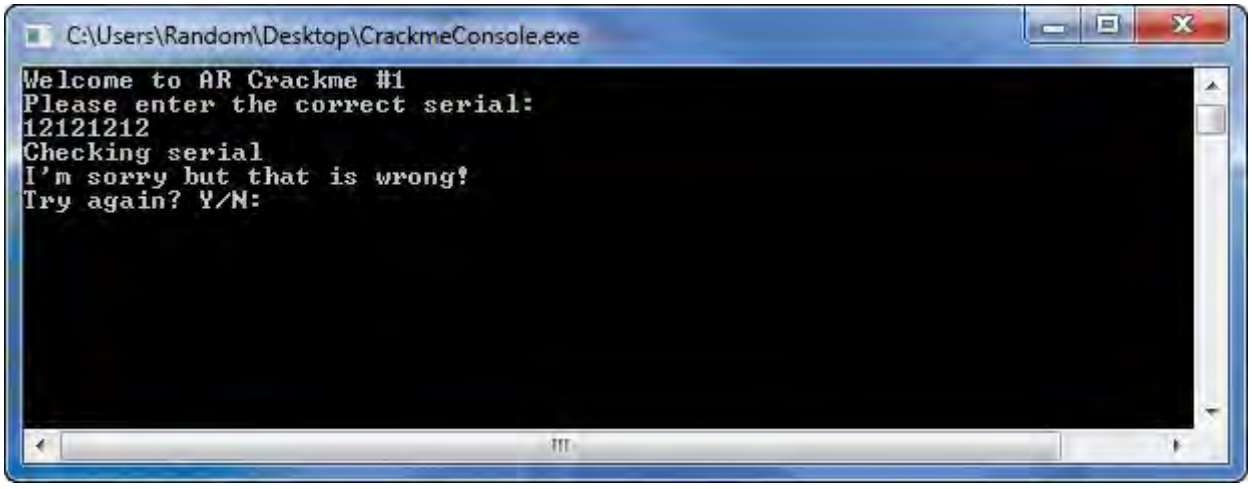
你可以在教程页下载相关文件及本文的 PDF 版。

那么，咱们开始吧....。

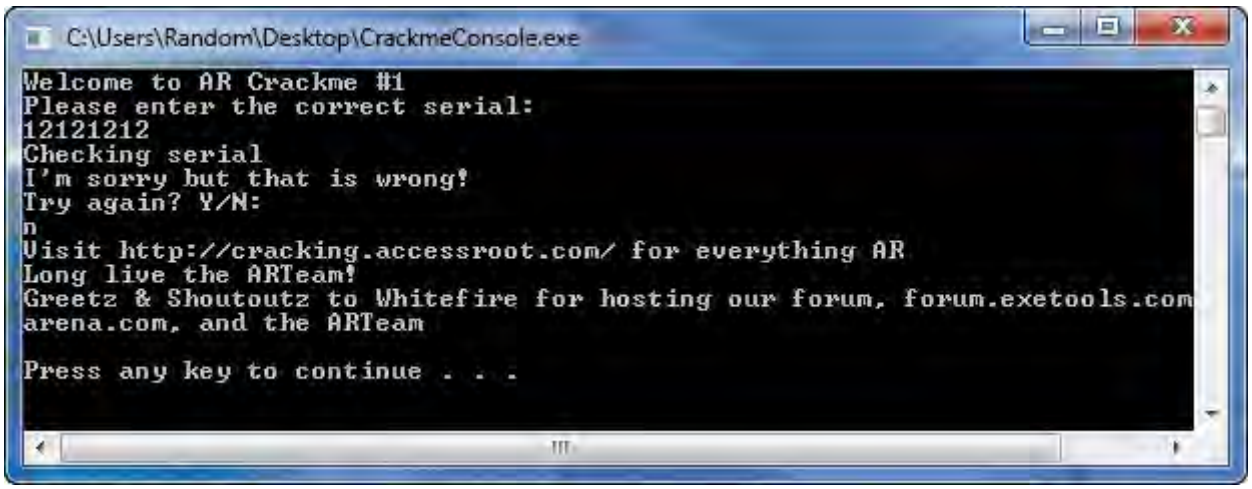
控制台程序是和其他 windows 下 32 位的程序一样。唯一的不同是它们不使用图形界面。除此之外，它们是一样的。此次的 crackme 叫 CrackmeConsole. exe。咱们来运行一下看看情况:



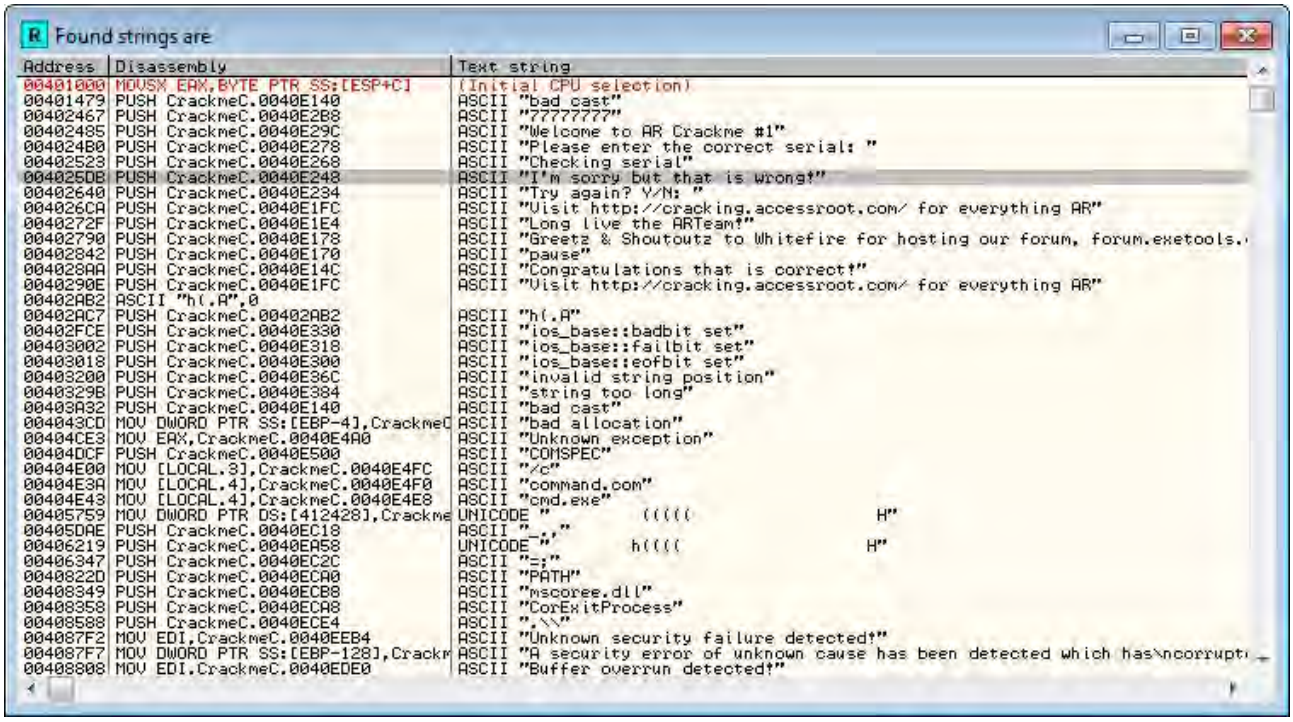
好，看起来挺简单得。咱们来随便输个密码:



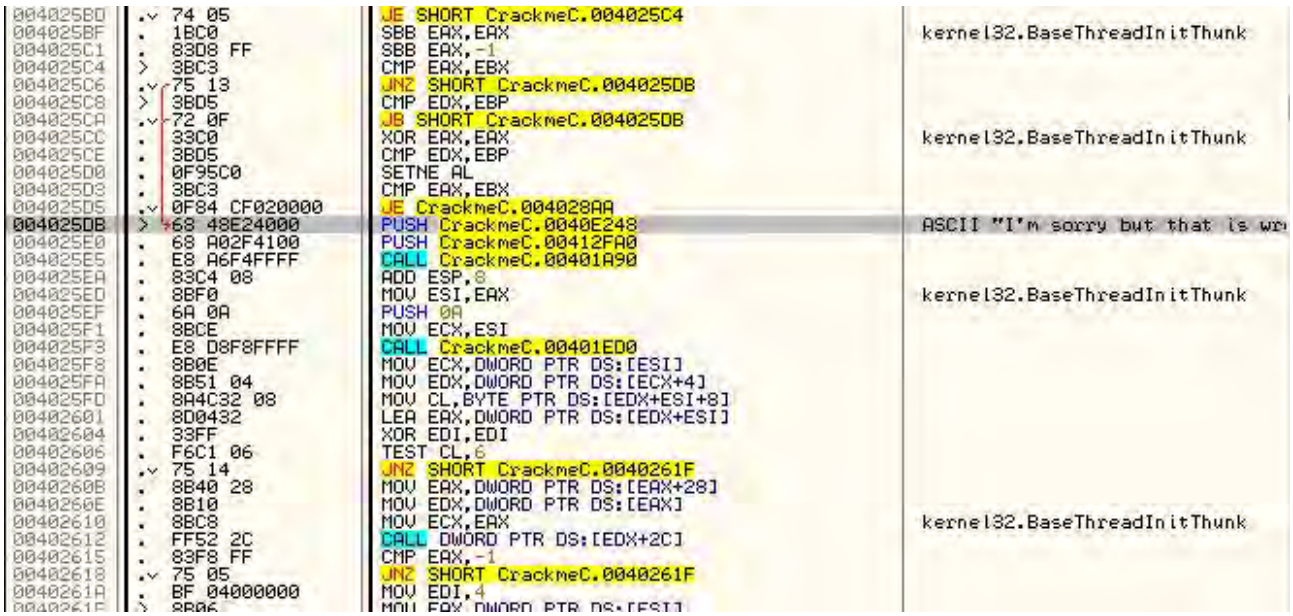
真失败！按“N”结束程序吧:



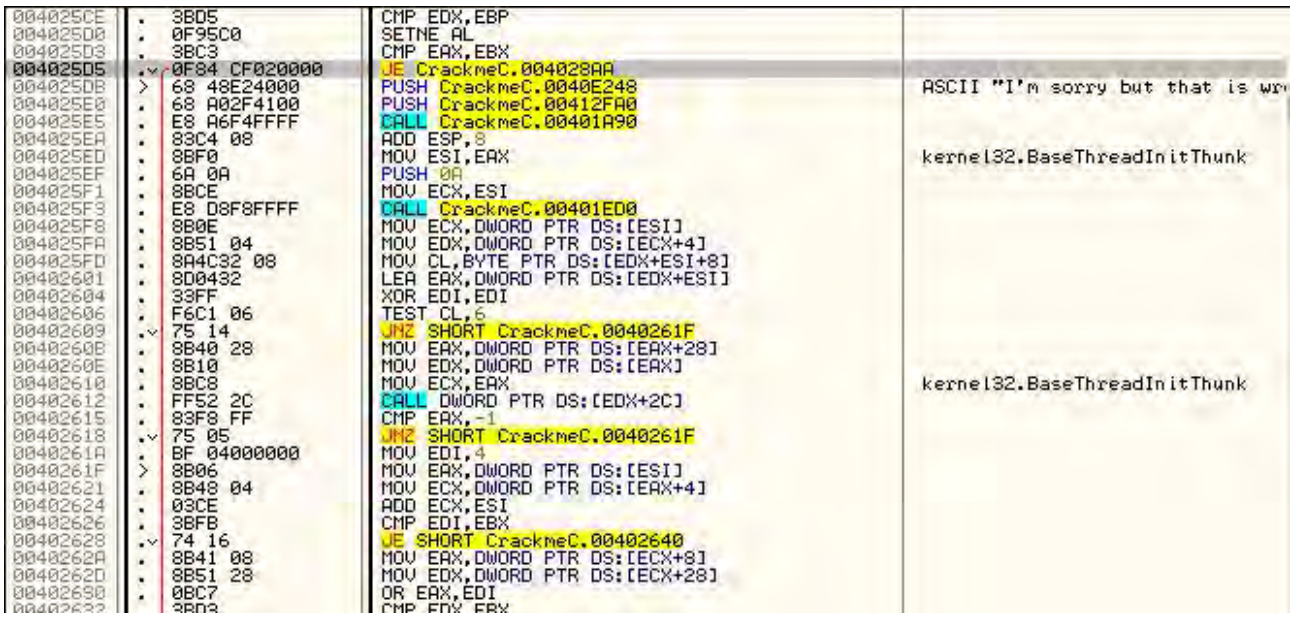
好吧，我觉得至少我们有了足够的信息来开始研究它。GO，Olly 载入应用。开始，首先搜索字符串：



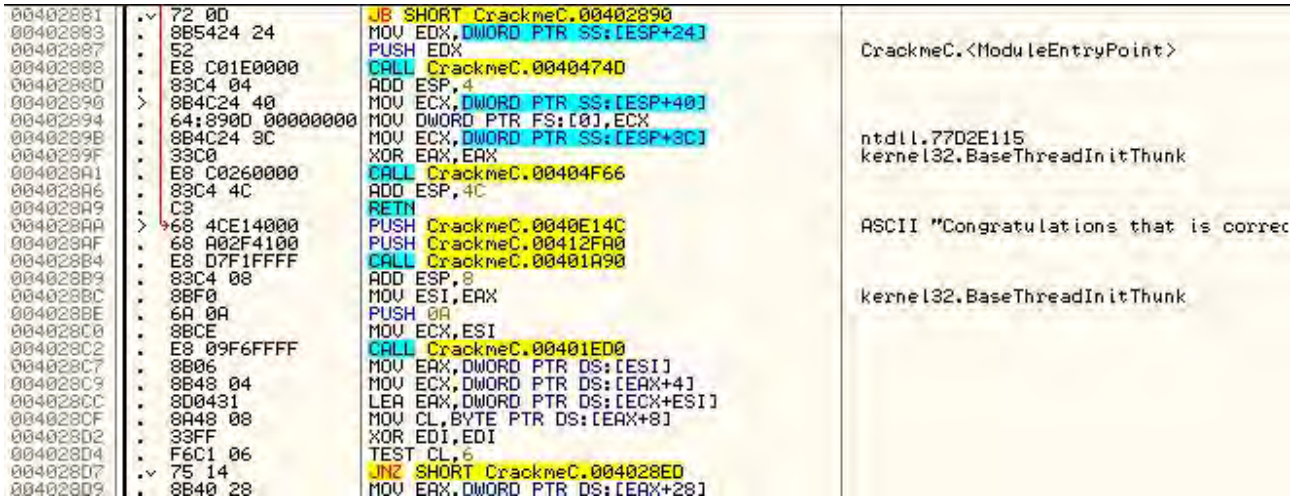
不是很难嘛！双击坏消息 “I’ m sorry, but that is wrong”，至少来到了正确的地方：



好，咱们研究研究这个。我们看到一个从 4025C6 来的跳转，有红色箭头标出来了。我们也注意到，如果 4025D5 的 JE 指令没实现的话，我们也会得到坏消息。咱们来看看如果这个跳转实现的话会怎么样。点它:

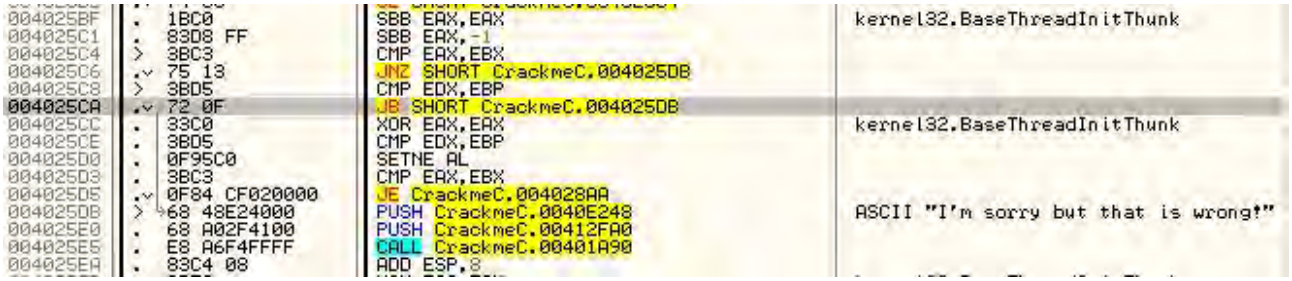


滚动到它指向的地方（在下面几页）:

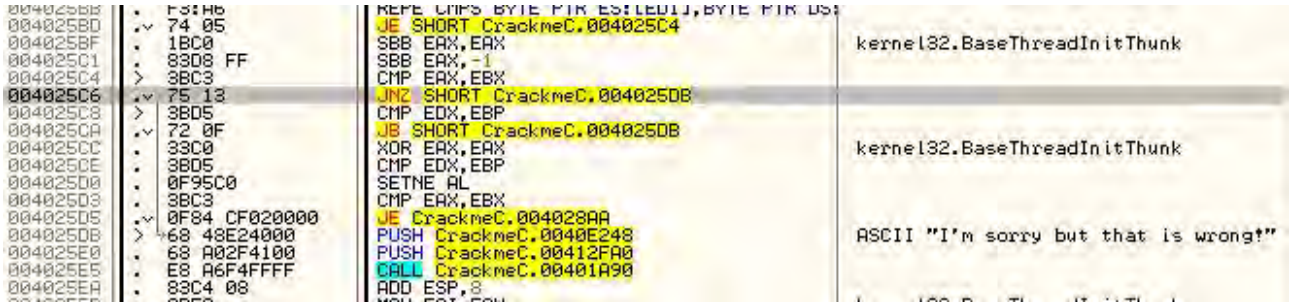


这看起来就是我们想走的路😀。咱们回到上面看看周围地方:

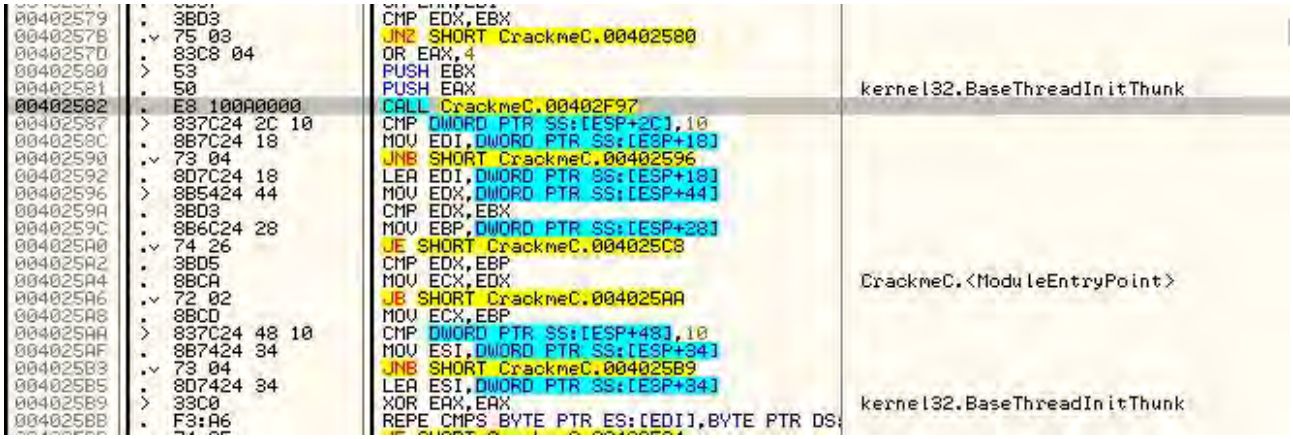
4025D5是调到好消息那的，这就是我们想要实现的跳转。咱们点一下另一个跳转看看它将跳到哪去....说不定前面也有个跳转可以跳到好消息那呢:



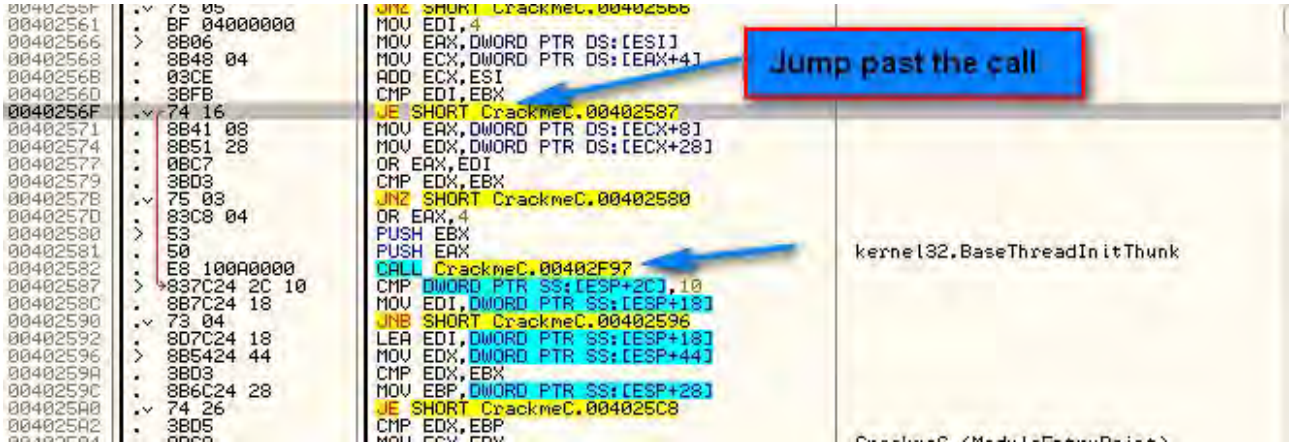
这个是到坏消息的:



这个也是，如果你接着点那些跳转指令，你会发现 4025D5 是唯一一个跳到好消息的跳转。所以基本上，我们要阻止所有跳到坏消息的跳转实现，强制跳到好消息的跳转成功跳转。如果我们接着往上滚动，就会在 402582 找到第一个 call/compare (调用/比较) 指令:



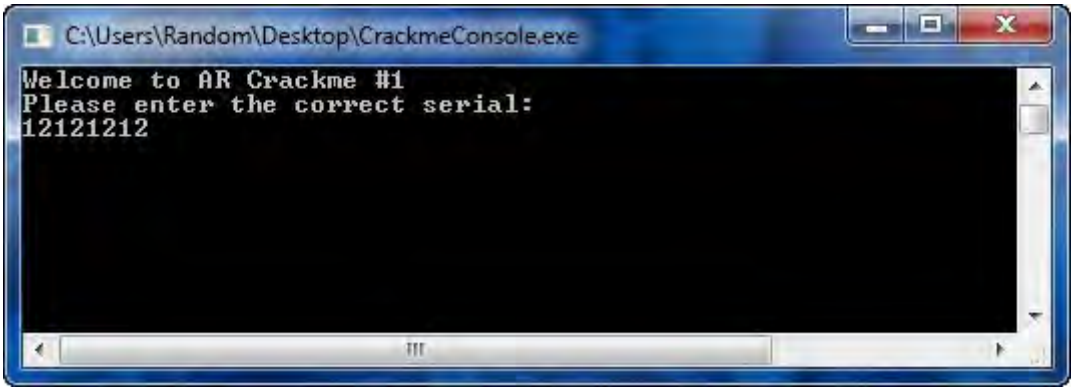
再往上滚动，就会发现有个跳转跳过了那个 CALL，但是仍然进行了比较:



这个行为不太正常，如果我们再往上滚动一点，就会发现另外一对 调用/比较 指令对。我在这两个 CALL 上都设置了 BP:

```
00402542  .  8B42 04        MOV EAX,DWORD PTR DS:[EDX+4]
00402545  .  8A4C30 08      MOV CL,BYTE PTR DS:[EAX+ESI+8]
00402549  .  03C6           ADD EAX,ESI
0040254B  .  33FF           XOR EDI,EDI
0040254D  .  F6C1 06        TEST CL,6
00402550  .v 75 14          JNZ SHORT CrackmeC.00402566
00402552  .  8B40 28        MOV EAX,DWORD PTR DS:[EAX+28]
00402555  .  8B10           MOV EDX,DWORD PTR DS:[EAX]
00402557  .  8BC8           MOV ECX,EAX                          kernel32.BaseThreadInitThunk
          .  FF52 2C        CALL DWORD PTR DS:[EDX+2C]
0040255C  .  83F8 FF        CMP EAX,-1
0040255F  .v 75 05          JNZ SHORT CrackmeC.00402566
00402561  .  BF 04000000    MOV EDI,4
00402566  >  8B06           MOV EAX,DWORD PTR DS:[ESI]
00402568  .  8B48 04        MOV ECX,DWORD PTR DS:[EAX+4]
0040256B  .  03CE           ADD ECX,ESI
0040256D  .  3BFB           CMP EDI,EBX
0040256F  .v 74 16          JE SHORT CrackmeC.00402587
00402571  .  8B41 08        MOV EAX,DWORD PTR DS:[ECX+8]
00402574  .  8B51 28        MOV EDX,DWORD PTR DS:[ECX+28]
00402577  .  0BC7           OR EAX,EDI
00402579  .  3BD3           CMP EDX,EBX
0040257B  .v 75 03          JNZ SHORT CrackmeC.00402580
0040257D  .  83C8 04        OR EAX,4
00402580  >  53             PUSH EBX
00402581  .  50             PUSH EAX                             kernel32.BaseThreadInitThunk
          .  E8 100A0000    CALL CrackmeC.00402F97
00402587  >  837C24 2C 10   CMP DWORD PTR SS:[ESP+2C],10
0040258C  .  8B7C24 18      MOV EDI,DWORD PTR SS:[ESP+18]
00402590  .v 73 04          JNB SHORT CrackmeC.00402596
00402592  .  8D7C24 18      LEA EDI,DWORD PTR SS:[ESP+18]
00402596  >  8B5424 44      MOV EDX,DWORD PTR SS:[ESP+44]
0040259A  .  3BD3           CMP EDX,EBX
0040259C  .  8B6C24 28      MOV EBP,DWORD PTR SS:[ESP+28]
```

好吧，咱们继续，在 Olly 中运行程序看看会发生什么。我将输入密码“12121212”：

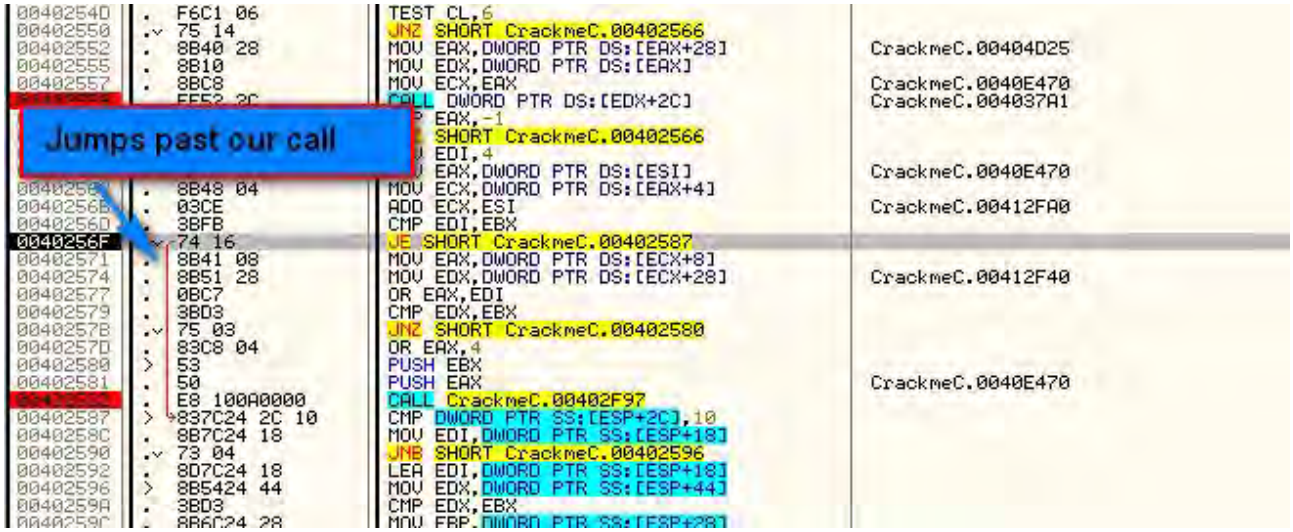


Olly 断在了第一个 CALL：

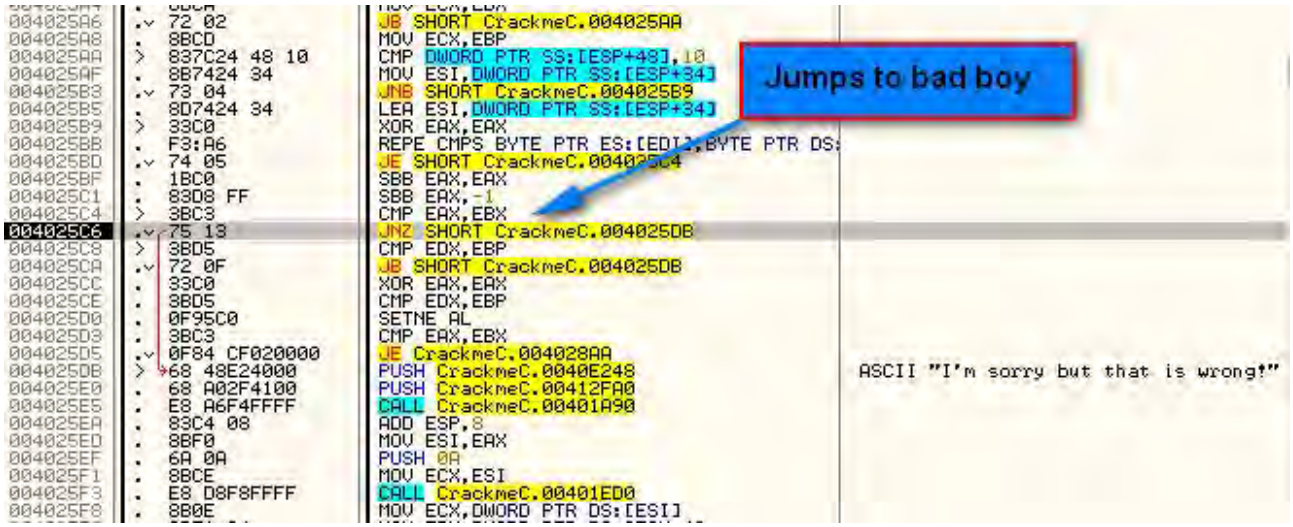
```
00402549  .  03C6           ADD EAX,ESI                          CrackmeC.00412FA0
0040254B  .  33FF           XOR EDI,EDI
0040254D  .  F6C1 06        TEST CL,6
00402550  .v 75 14          JNZ SHORT CrackmeC.00402566
00402552  .  8B40 28        MOV EAX,DWORD PTR DS:[EAX+28]
00402555  .  8B10           MOV EDX,DWORD PTR DS:[EAX]           CrackmeC.0040E408
00402557  .  8BC8           MOV ECX,EAX                          CrackmeC.00412F40
          .  FF52 2C        CALL DWORD PTR DS:[EDX+2C]           CrackmeC.004037A1
0040255C  .  83F8 FF        CMP EAX,-1
0040255F  .v 75 05          JNZ SHORT CrackmeC.00402566
00402561  .  BF 04000000    MOV EDI,4
00402566  >  8B06           MOV EAX,DWORD PTR DS:[ESI]           CrackmeC.0040E470
00402568  .  8B48 04        MOV ECX,DWORD PTR DS:[EAX+4]
0040256B  .  03CE           ADD ECX,ESI                          CrackmeC.00412FA0
0040256D  .  3BFB           CMP EDI,EBX
0040256F  .v 74 16          JE SHORT CrackmeC.00402587
00402571  .  8B41 08        MOV EAX,DWORD PTR DS:[ECX+8]
00402574  .  8B51 28        MOV EDX,DWORD PTR DS:[ECX+28]
00402577  .  0BC7           OR EAX,EDI
00402579  .  3BD3           CMP EDX,EBX
0040257B  .v 75 03          JNZ SHORT CrackmeC.00402580
0040257D  .  83C8 04        OR EAX,4
00402580  >  53             PUSH EBX
00402581  .  50             PUSH EAX                             CrackmeC.00412F40
          .  E8 100A0000    CALL CrackmeC.00402F97
00402587  >  837C24 2C 10   CMP DWORD PTR SS:[ESP+2C],10
0040258C  .  8B7C24 18      MOV EDI,DWORD PTR SS:[ESP+18]
00402590  .v 73 04          JNB SHORT CrackmeC.00402596
00402592  .  8D7C24 18      LEA EDI,DWORD PTR SS:[ESP+18]
00402596  >  8B5424 44      MOV EDX,DWORD PTR SS:[ESP+44]
0040259A  .  3BD3           CMP EDX,EBX
0040259C  .  8B6C24 28      MOV EBP,DWORD PTR SS:[ESP+28]
```

单步调试，注意 42056F 处的跳转跳过了第二个 CALL。嗯，这倒给了我们一个提示，第二个跳转可能不是校验密码的，不过有可能是某种验证程序，如

果我们的密码不符合某种规则，比如太短或者太长？不管是啥，咱们接着单步运行就行了：

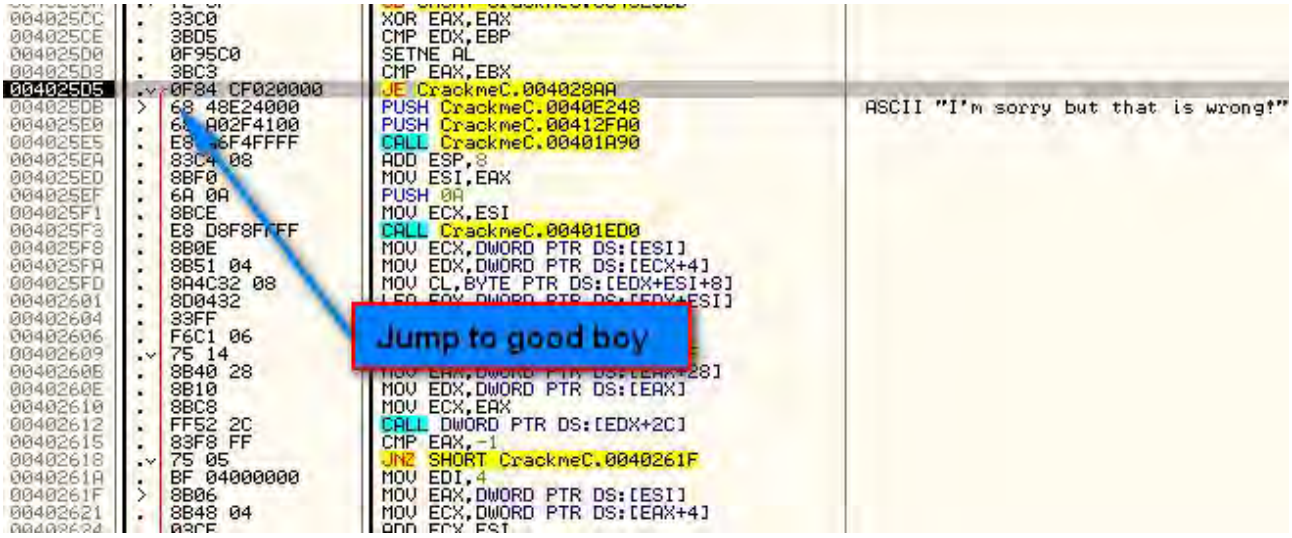


4025C6 这里，咱们看到了罪魁祸首了，就是它跳到了坏消息那：

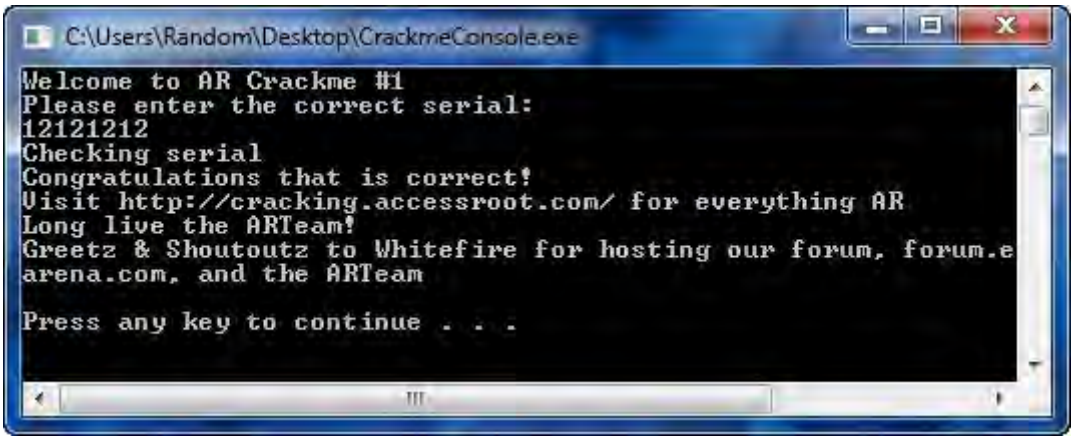


咱们设置下 0 标志位，看看会怎么样：

继续单步，终于和跳到好消息的跳转碰头了，注意它实现了：



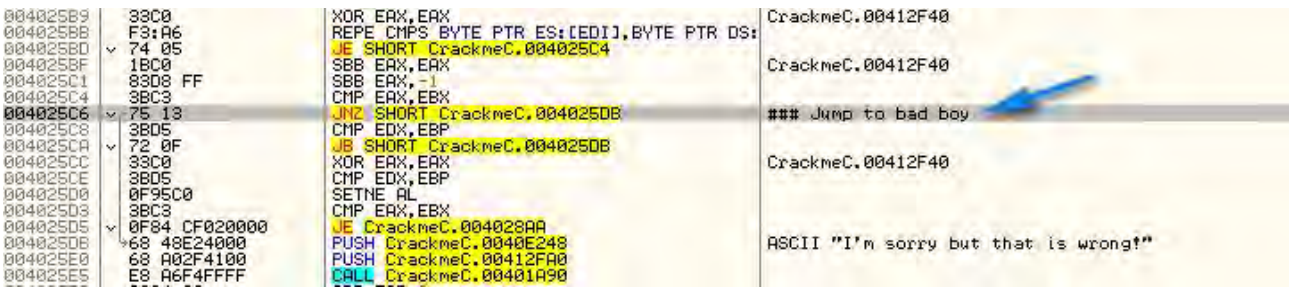
继续运行程序，我们发现我们已经找到了第一个潜在的补丁：



现在，给我们刚才设置 0 标志位的那个跳转打上补丁，这可能有用也可能不起作用。这很难说。如果我们的密码太短会怎样？太长呢？是不同于我们所输入的密码的（译者注：大概这个意思，我没弄明白作者啥意思。原文是 A different password than the one entered）。这个补丁不是一个非常好的补丁，因为我们真的不知道我们到底做了什么，我们只知道在这种情况下会起作用。

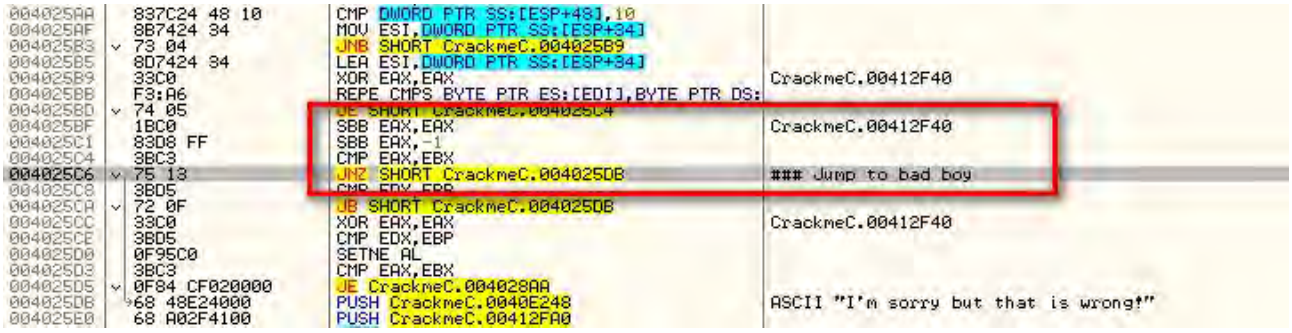
二、深入挖掘

咱们靠近点看看这段代码，用上一章我学到的级别，试试不那么 LAME 的方法。向上滚动到我们打过补丁的那个跳转，就是跳到坏消息的那个，咱们来试试看找出为什么我们没打补丁时它会跳转。注意，我已经在跳转那加了一个注释，这样后面比较容易记住（回想下，选中该行，按一下 “;” 来添加注释）：

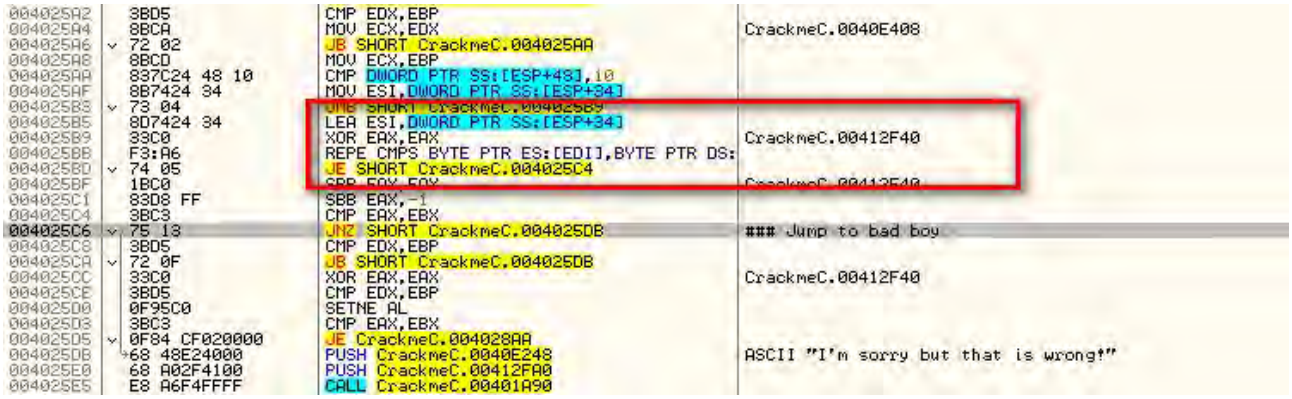


我们通常在注释前加上“###”以示区别，这样的话在将来，当用其他的工具来向我们显示注释的时候，就更容易找到我自己得注释，因为它们比较突出。当然你也可以按自己喜欢的方式做。

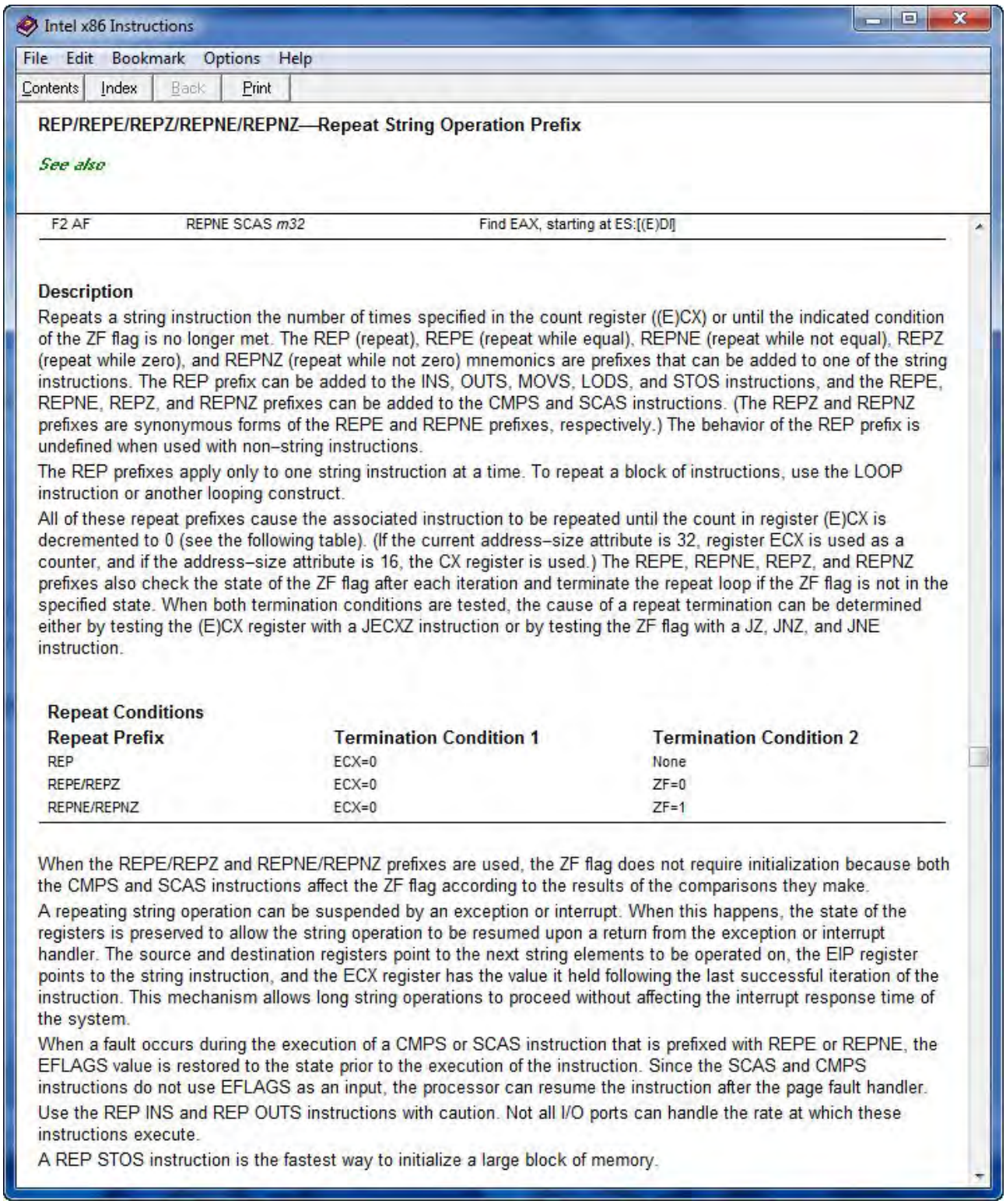
现在，咱们就来看看跳转的上面，看能否找到是什么让它跳转的。我在下面已经标记出了跳转上面的第一个区块:



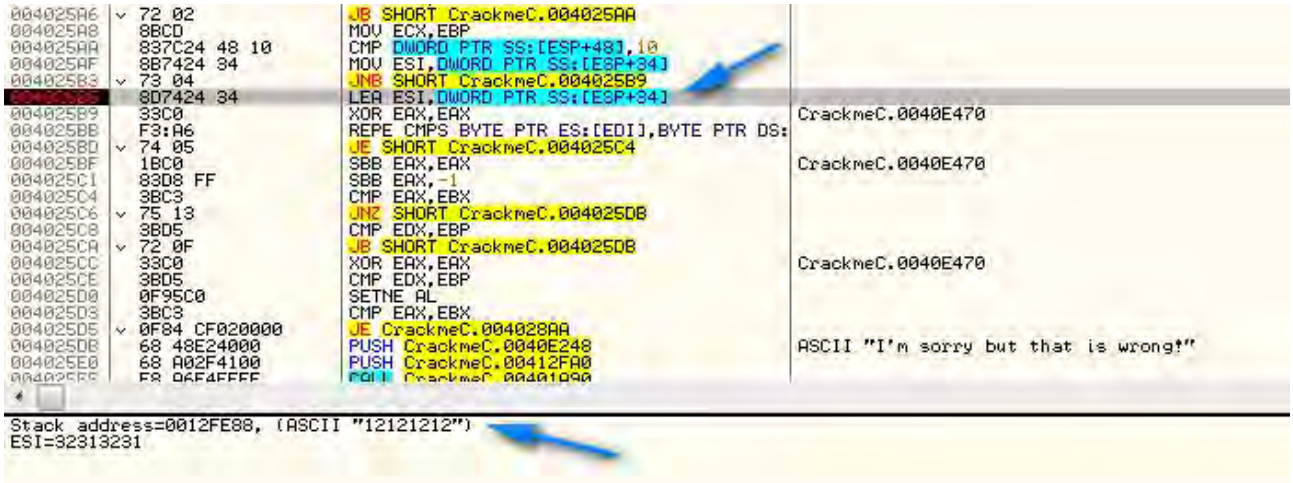
我们能看到有几个 SBB 指令和一个比较指令。对于我们来说，这里的这段代码并不真正有什么意义，因为我们不知道它是干啥的，所以咱们网上看下一个区，看看咱们能不能开始对它有所了解:



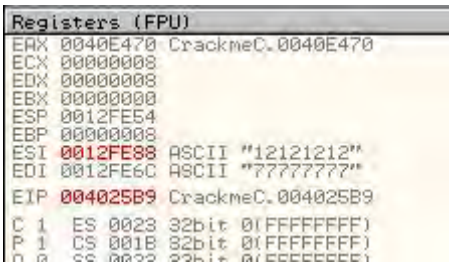
好，这里我们将会到达某个地方。可能你注意到第一个问题的是 REPE CMPS 指令。这是逆向工程的一个红色标志（译者注：原文是 This is a red flag in reverse engineering!，我不知道作者是啥意思，就直译了)! 咱们查查 REPE 看看是啥意思:



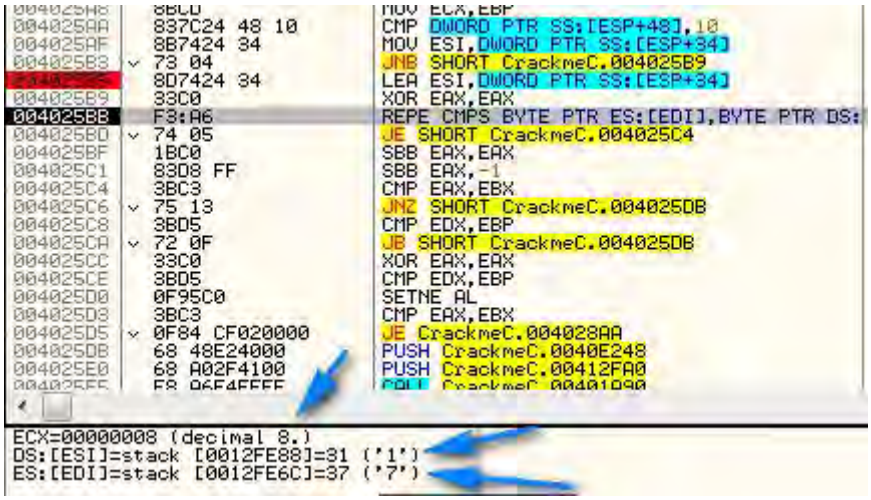
这个不是非常的清楚，不过如果你对汇编语言稍有经验的话，就知道 REPXX 语句像循环一样重复直至 ECX=0。REPXX 后面的指令，这里是 CMPS，就是重复的内容。放在一块的话，这个语句就是“当 0 标志位保持不变时，重复比较两个内存地址的内存，每循环一次就增加一次地址大小”。简而言之，就是“比较两个字符串”。在逆向工程领域，任何时候我们比较两个字符串，红色标志都应该消失。应用程序不会经常这么做，校验序列号/密码/注册码只是多次比较中的一个。咱们在该区块的第一行也就是 4025B5 处设置一个 BP，并重启应用。输入咱们的密码，然后 Olly 断了下来：



现在，注意第一条指令，LEA ESI，DWORD PTR SS: [ESP+34]，准备将一个栈中得有效地址载入到 ESI 中。SS: 表示堆栈，[ESP+34] 表示的是栈中的位置，本例中是 ESP 所指向位置前面的第 34 字节。LEA 指令意思是取地址，而不是取内容。如果我们看那个中间区域（就是蓝色箭头指向的地方），可以发现 SS: [ESP+34] 等于地址 0012FE88，在这个地址存储的是我们的 ASCII 形式的密码。单步步过该行，可以看到 ESI 被设置成我们的密码（当前是在栈上）:

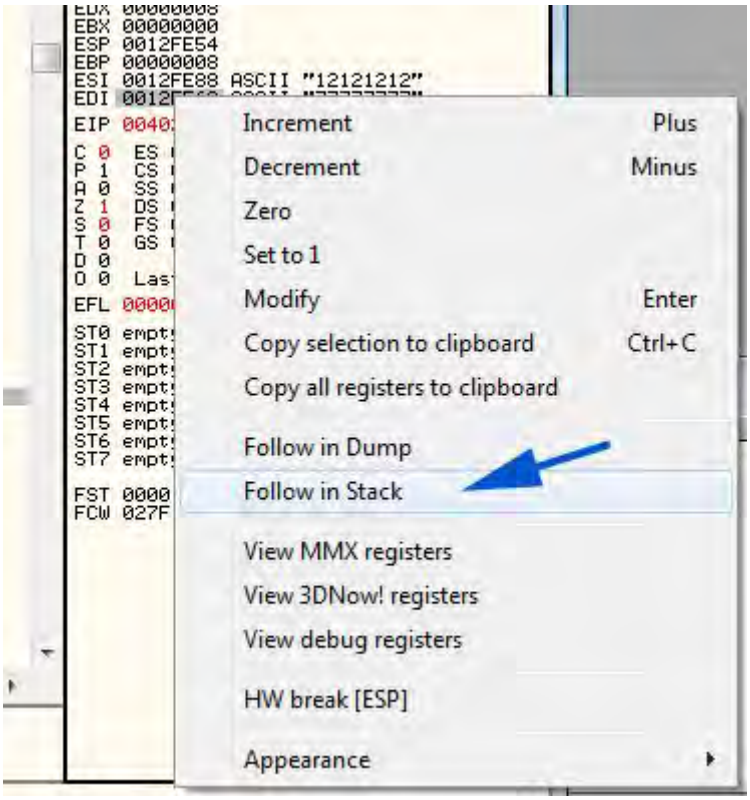


下一条指令将 EAX 设置为 0，然后就是 REPE 指令。本例中，是将存储在 ESI 中的地址的内容与存储在 EDI 中的地址中的内容进行比较:

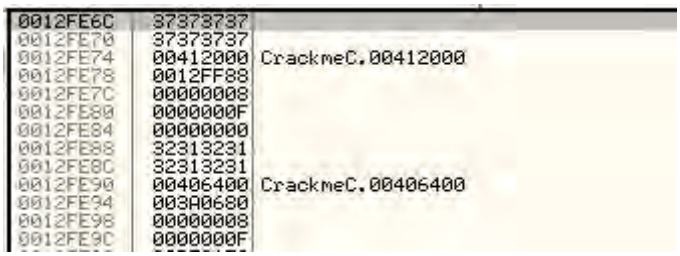


ECX 寄存器减一，比较就转到 ESI 和 EDI 的下一个内存位置，当 ECX=0 时循环结束。本例中，如果你往上看，会发现 ECX 被置为 8（就是我们密码的长度），所以该循环会遍历我们密码的 8 个数字，每一次将一个数字与 EDI 中相

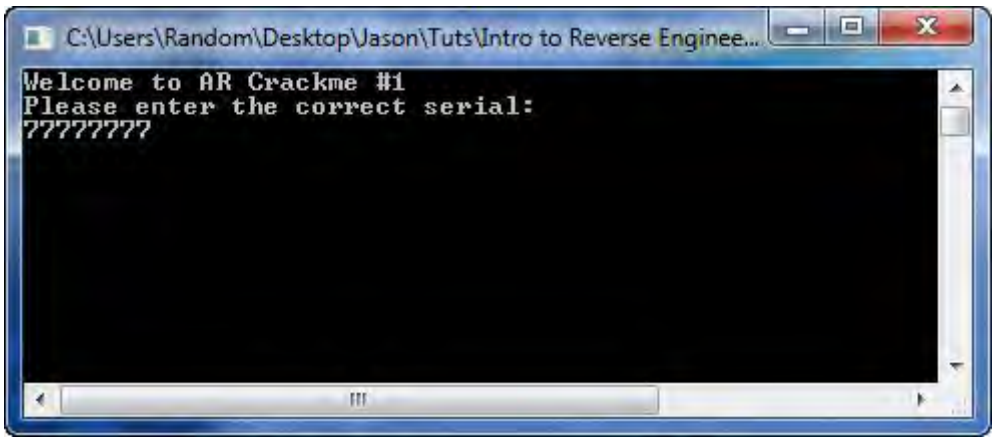
关的数字进行比较。不过，等等...，我们正和谁比较呢？如果我们再看看寄存器窗口，我们会发现 EDI 指向的是堆栈中的一个地址，其中存储着几个 ASCII 字符 7。咱们到堆栈中看看。点击挨着 EDI 的那个地址，在其上右键选择 “Follow in stack（堆栈中跟随）”：



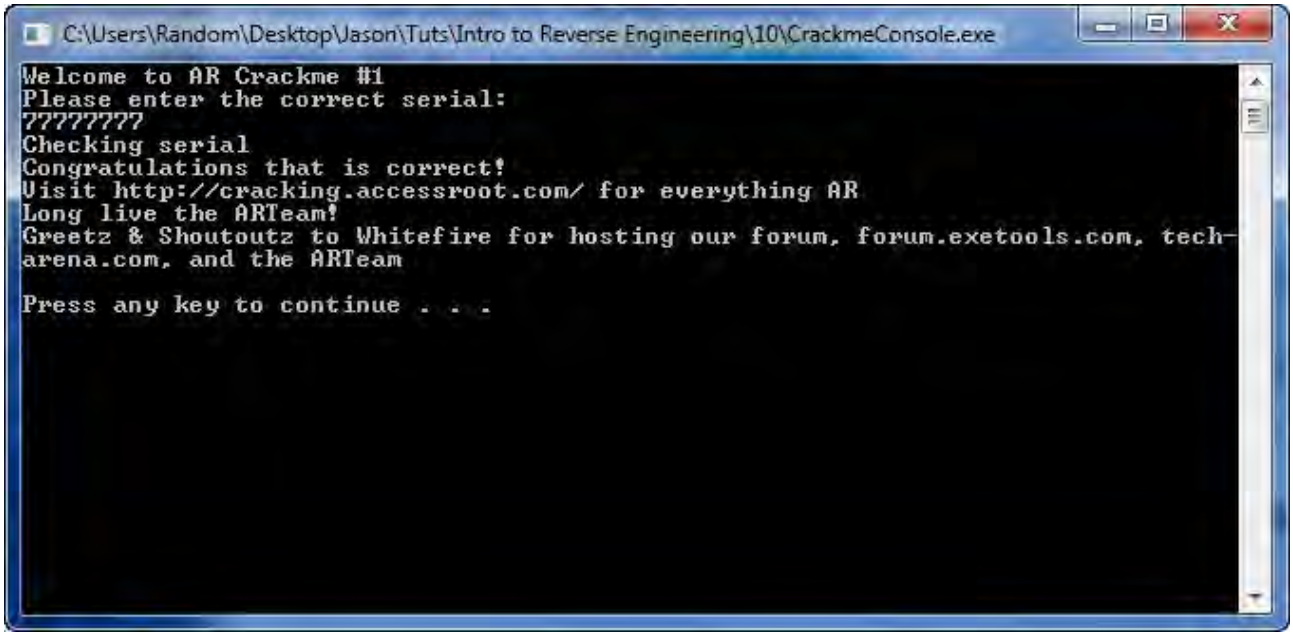
堆栈窗口立即就跳转到相关地址处，也就是 0012FE6C 处。在该地址（我们不能不注意到后面的也是一样）我们看到一串 “37”。查查 ASCII 码表就知道 37 就是 “7”，就是我们在寄存器窗口中看到的 EDI 寄存器中的内容：



好吧，不需要像外科医生那样就能够发现我们输入的密码正在和硬编码的全是 “7” 的字符串进行比较。堆栈中真切的只有 8 个 “7”（很走运，我们输入的密码正好和硬编码密码的长度相同😀）。这八个 “7” 与我们输入的密码一个一个的进行比较。如果所有的 8 个都相等（也就是等于 7），我们就会执行下一个跳转。嗯...，我们输入的密码被拿来和 8 个 “7” 进行比较。给我的感觉就是密码可能就是八个 “7”。咱们来重启应用试试看：



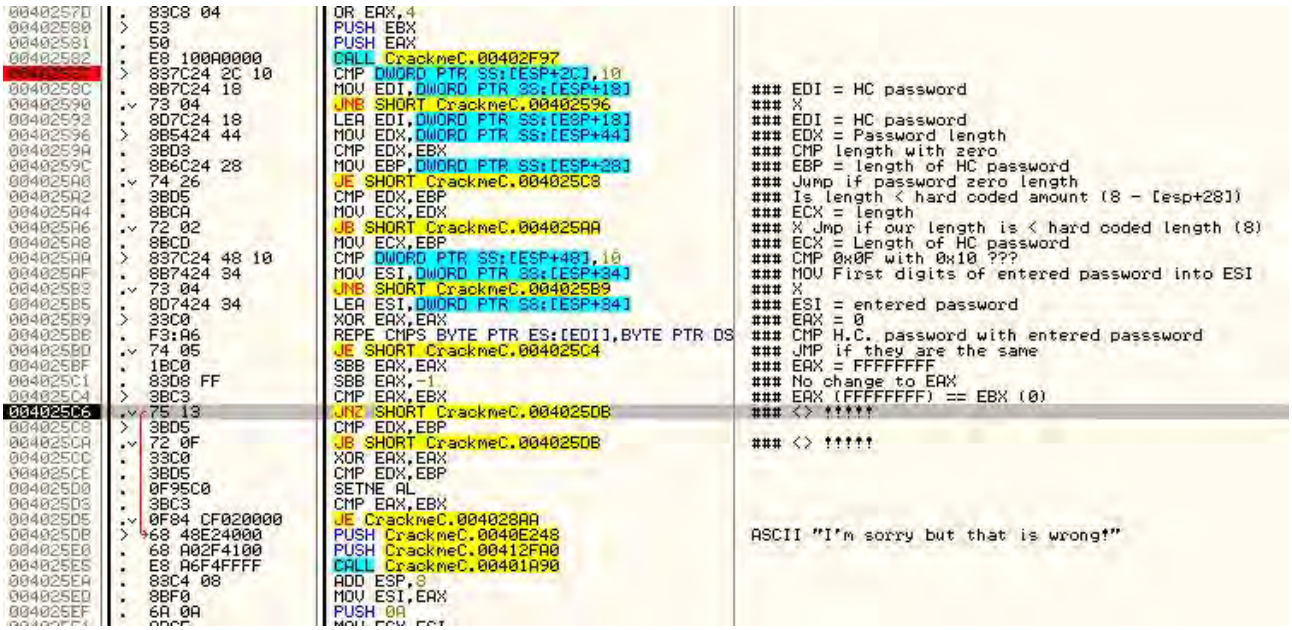
此处应该有掌声....



我们拿到了😀。所以，在我们通常打补丁的地方的稍远处我们发现了密码，坦白的说这比给一个程序打补丁要好的多，因为我们不知道是真的打上了还是没有。相比 LAME 级别，这就是 NOOB 级别补丁的好处。

三、最后一件事

我只是想举个例子，是分析代码及对代码进行注释。不幸的是，在写教程时，你需要在相当深的层次上理解相关应用。下面是核心区块的图片，我在其中加了注释:



如你所见，很多都是对应用程序工作方式的理解。

第十二章：一个难啃的 NOOB 例子

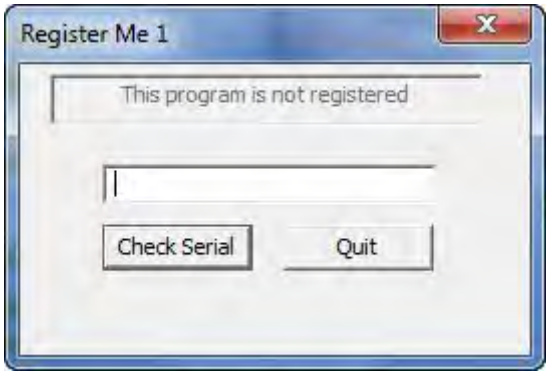
一、简介

本章我们将研究一个有点更具挑战性的程序。它叫 ReverseMe1，我写的。我也会讨论一个 Olly 的插件“ASCII 码表”。可以在工具页下载它。这个 ReverseMe 是用来说明为什么 LAME 补丁方式通常就是那么 lame（烂）的一个极好的例子。

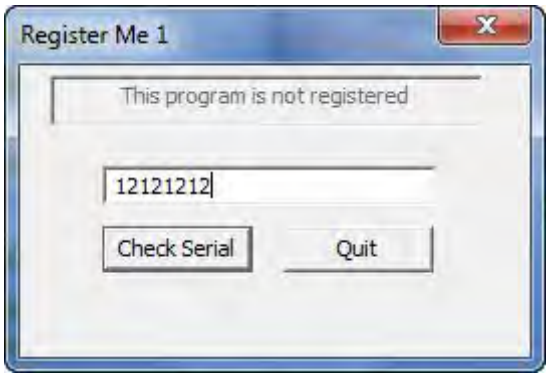
你可以在教程页下载相关文件及本文的 PDF 版。

二、准备开始

运行下程序看看:

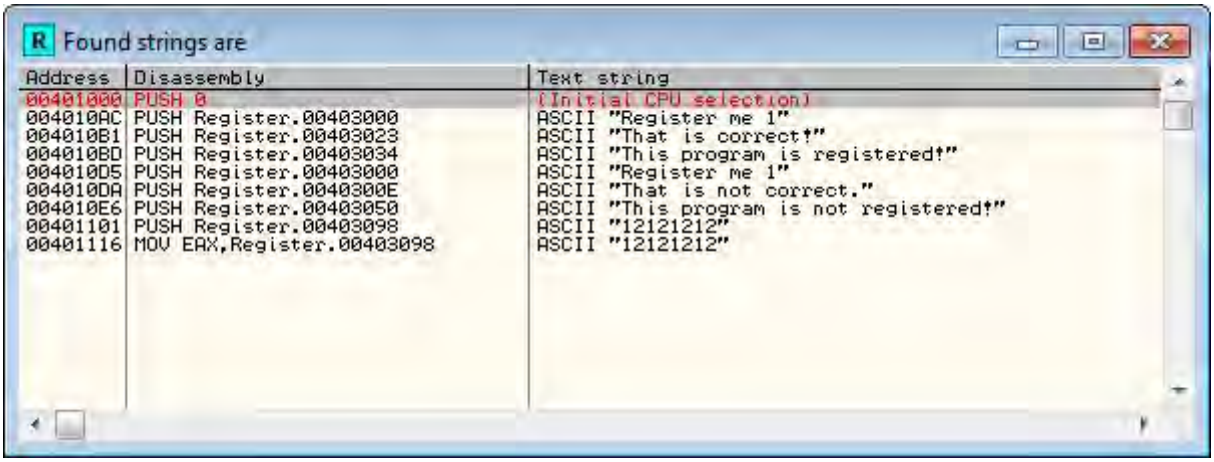


我们能看到它说还没有注册，需要序列号。那就给它一个:



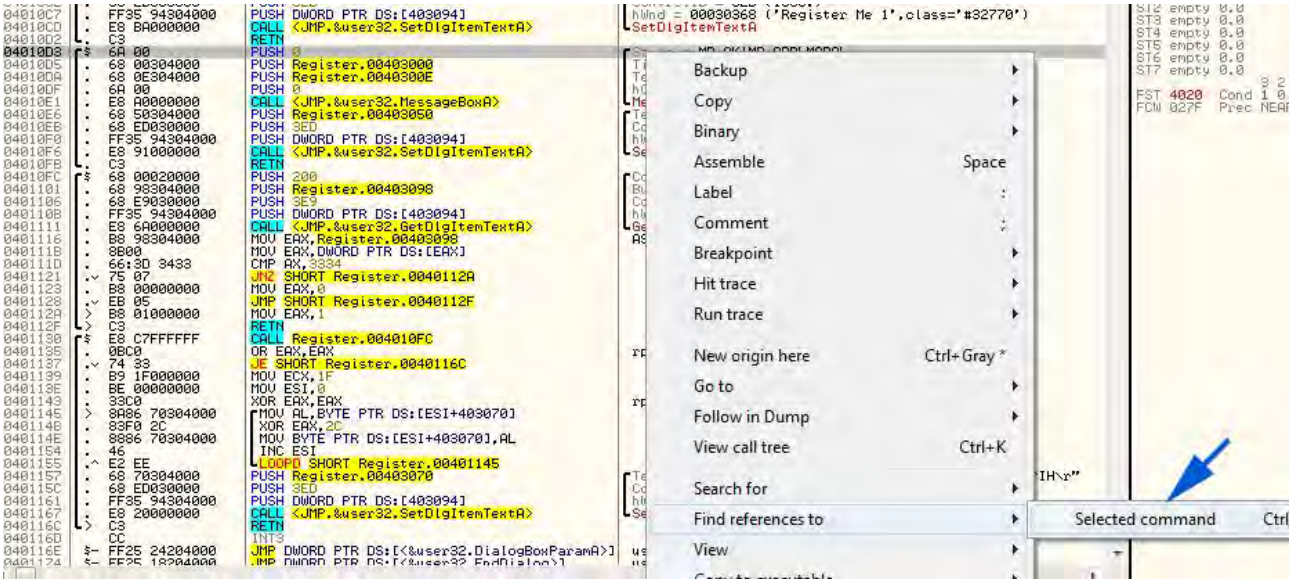
点“Check Serial”:

我们看到我们是错的（再一次）！Olly 载入应用，用咱们信得过的“搜索字符串”：

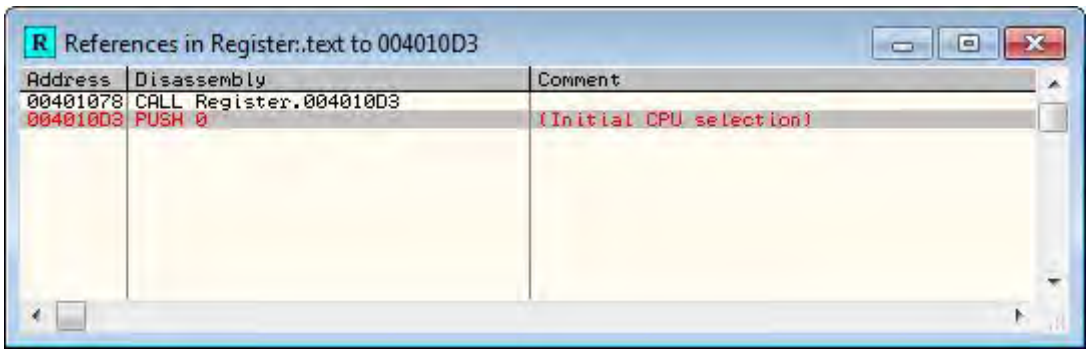


好哇，看起来前途光明呀。咱们来检查下“That is not correct”字符串：

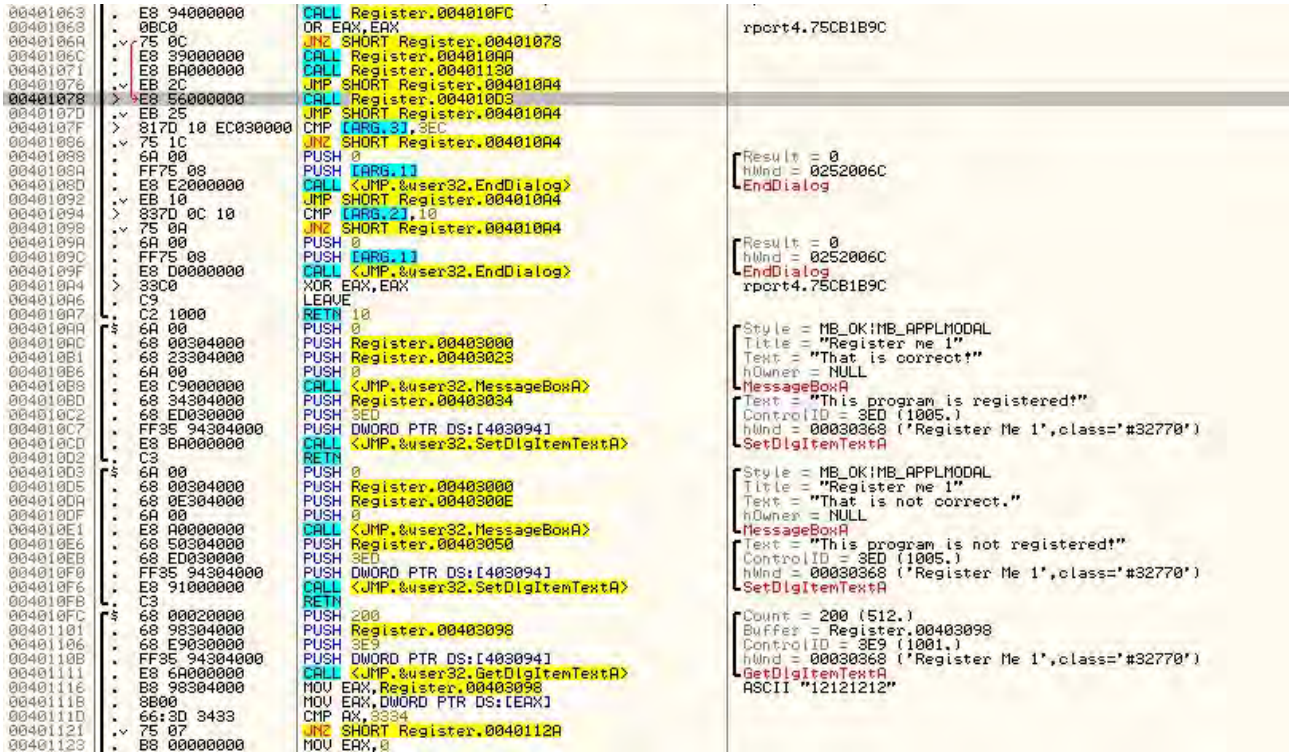
咱们来到了问题的核心。因为每一个都是单独的方法，我们需要看看哪里调用了它们，所以咱们要这么做：



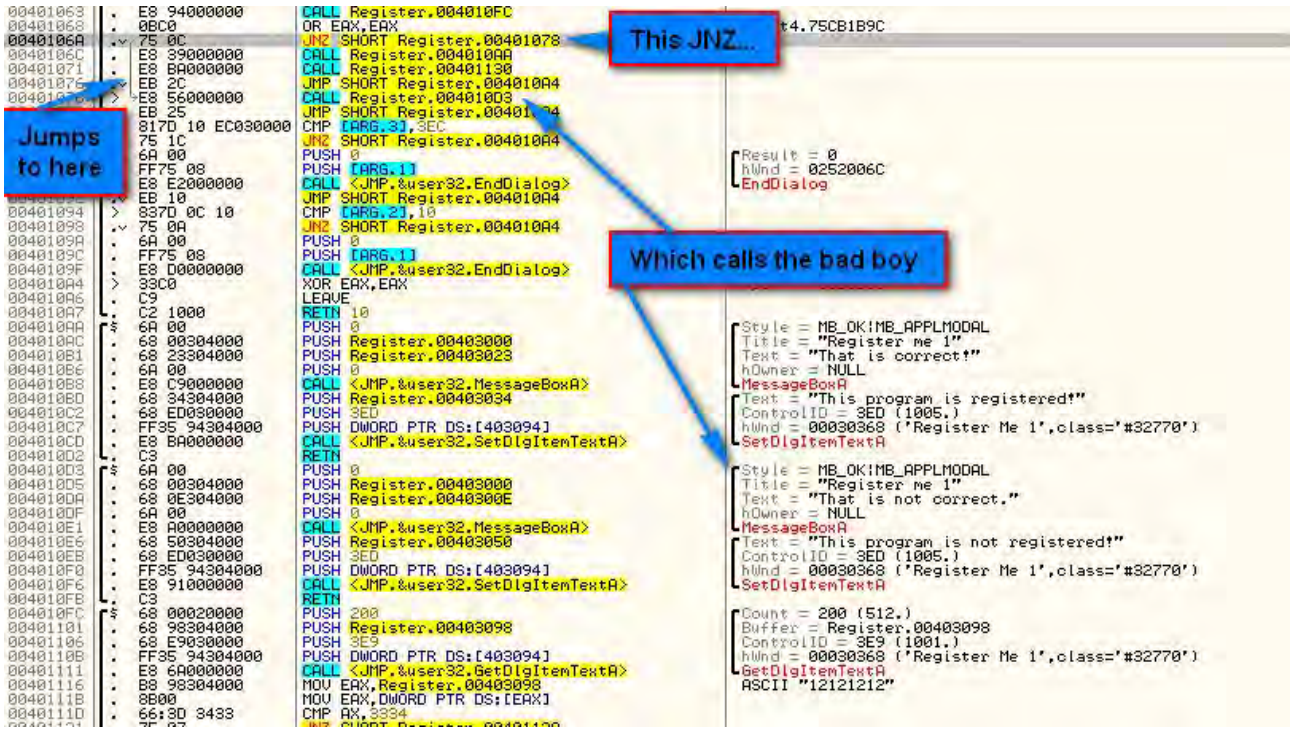
Olly 弹出了 References 窗口：



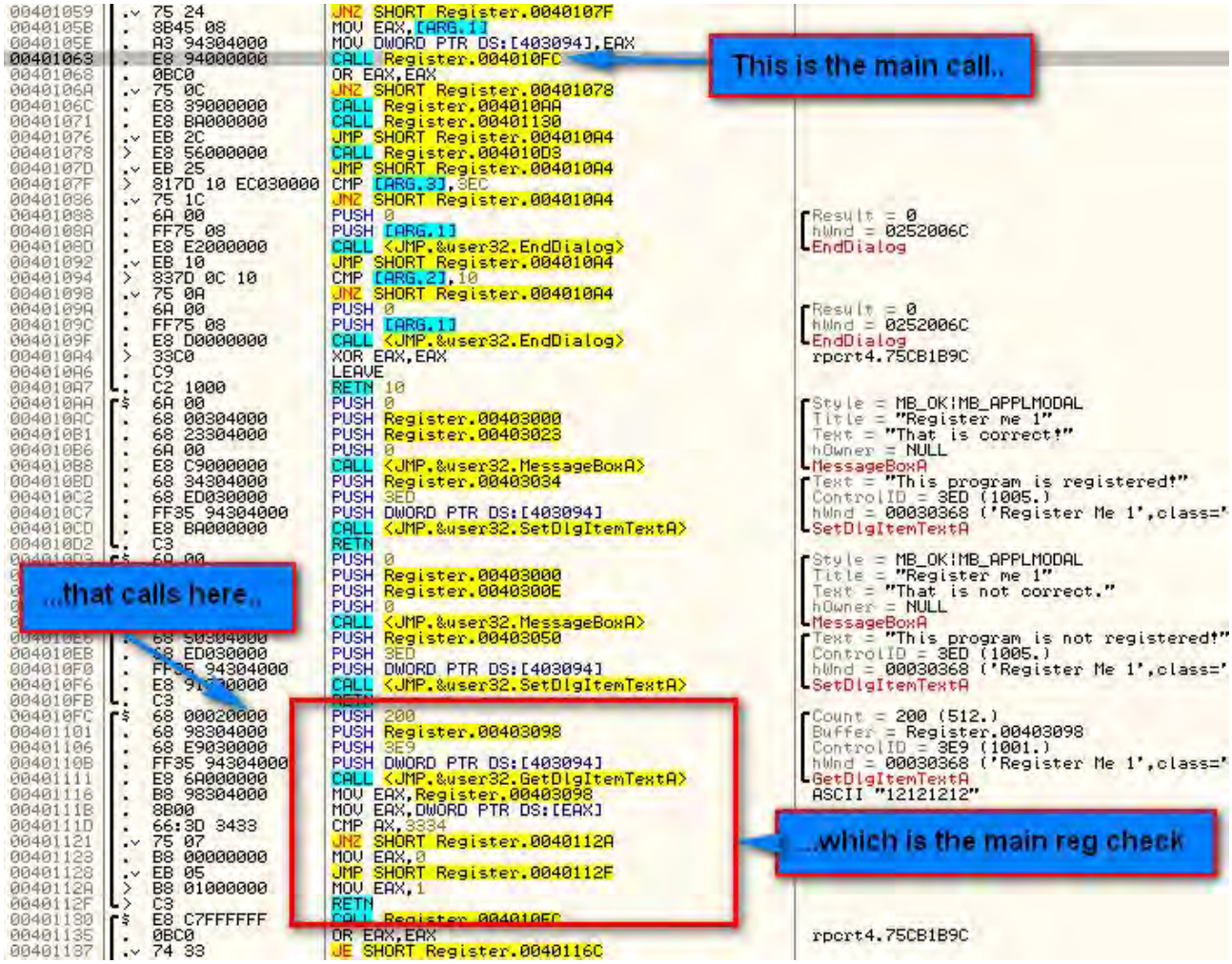
我们能够看到有一个对该函数的调用。咱们双击它，看看它是啥样的：



这里，我们能看到坏消息是在 401078 处调用的，并且我们马上就能看到 40106A 处有个跳转指令跳到这里：



向上滚几行，我们就能看到一个 CALL，用来检测 程序/比较/跳转 ，和我们前面看到的一样。从这里我们能够猜到，主要的检测程序是在 4010FC，401063 处调用了它。在返回后，EAX 寄存器被检测其值是否是 0，如果不是就跳到坏消息。

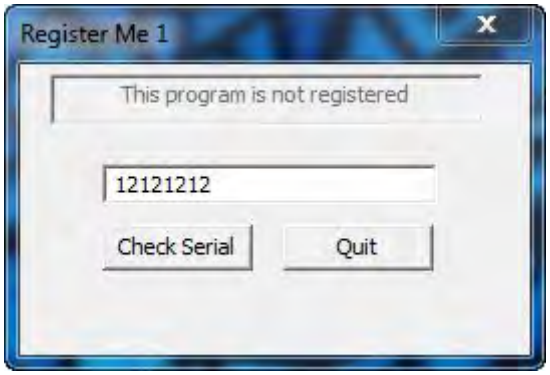


测试下我们的假设，在 40106A 处设置断点，然后重启应用。在输入一个序列号以后（我输入的还是“12121212”），我们断在了调用序列号校验的那个 CALL 后面的跳转处:

现在咱们帮 Olly 走正确的路，所以我们不能让跳转实现（直接到调用好消息的 CALL 那）:

点一下运行:

耶，so easy（妈妈再也不用担心我破不了了）!!点 OK:

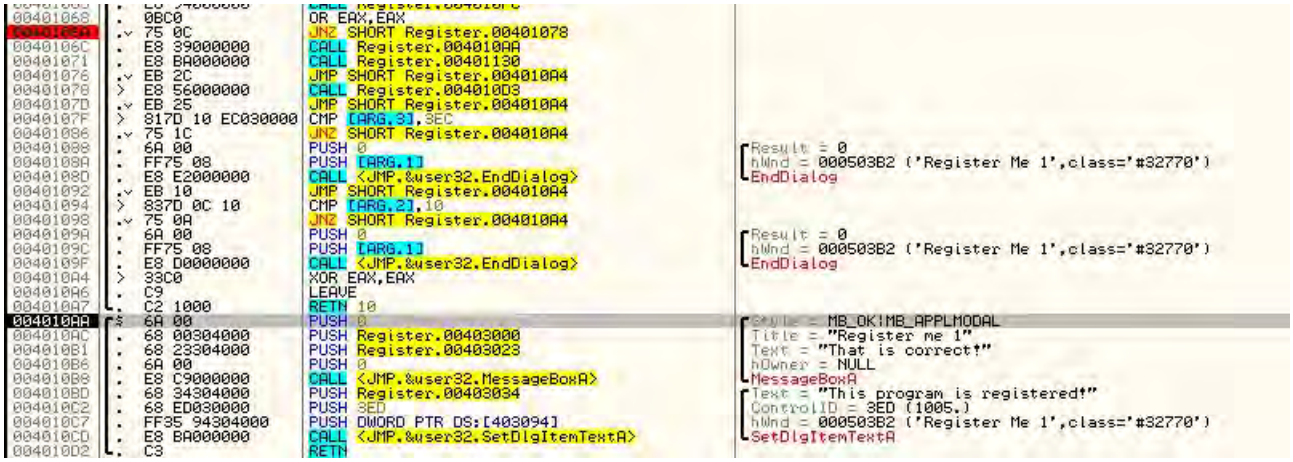


噢，艹%$@，这里他爸的发生了啥，你个阿西吧$$$%^#!!!!!很明显，我们的程序没有注册成功。这说明我们肯定错过了啥。

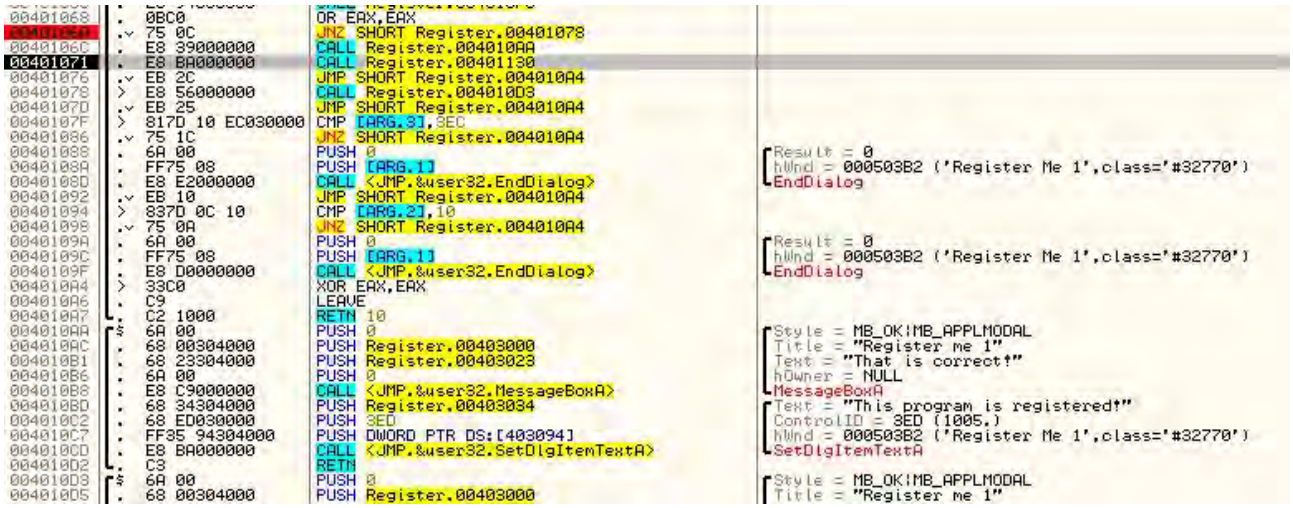
三、进一步分析

重启应用，输入序列号，让 Olly 再次断在 40106A:

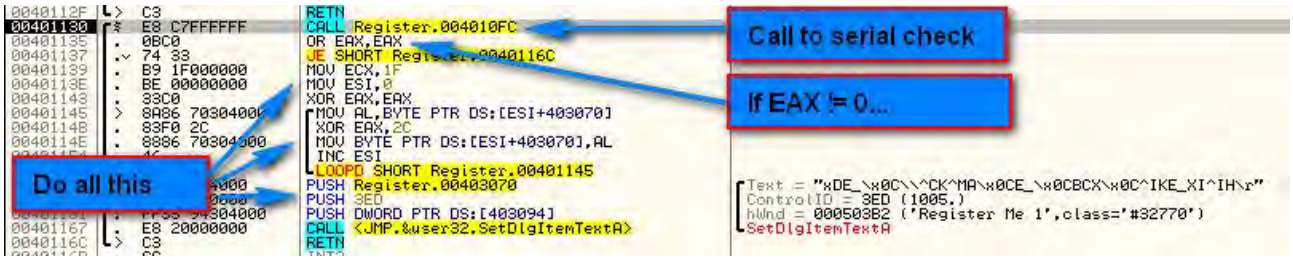
看看这个，如果我们阻止 Olly 跳到坏消息那，直接执行 40106C 的那个 CALL，就是调用 4010AA。沿着那条路往下走，我们能看到它是相当的标准：它弹出一个显示 “That is not correct” 的消息框，然后将主窗口的标签修改成 “This program is registered!”。



等等！一旦我们从那个 CALL 返回了，在 401071 还有另一个 CALL 等着咱：

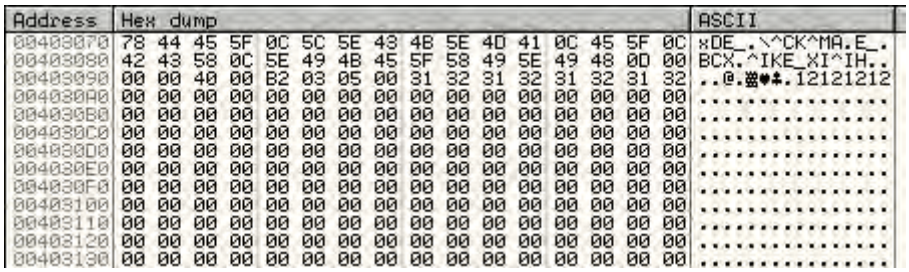


该 CALL 调用的是 401130，所以咱们看看那个子程序。首先，我们注意它调用了 SetDlgItemTextA，不过有一个看起来很奇怪的字符串。咱们来一行一行的执行。401130 有个 CALL 调用了 4010FC。往上看，我们看到这是一个序列号校验子程序。然后 EAX 自身做了 OR 操作看是否为 0，如果不是，它执行了许多看起来很怪异的玩意儿：



到目前为止，我们从这些收集到的信息是，在我们给程序打了补丁后它显示了好消息，然后另一个 CALL 执行了，在这个 CALL 里，又有一个 CALL 再次执行了序列号校验子程序，对结果做了同样的分析。这是一个备份检测点！现在我们来看看如果我们在这个备份检测点失败的话会怎样（这里我们是可以让它检测失败的，因为我们只给那个跳转打了补丁）：

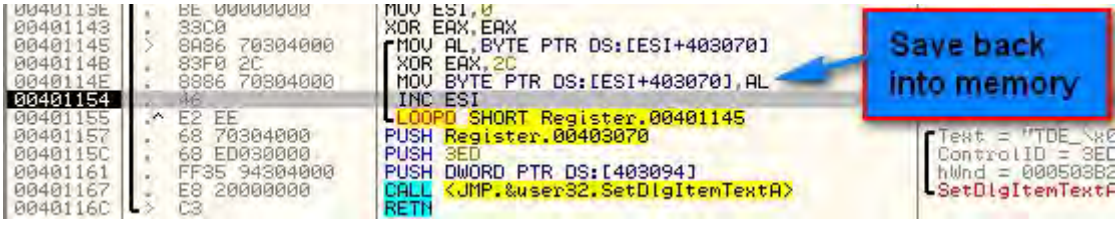
首先，ECX 被设置值为 1F（十进制是 31）***对不住了，被切掉了一点（译者注：指的是上面图片中 MOV ECX，1F 那行）***。然后 ESI 被赋值为 0，EAX 被清 0。然后就进入一个循环。咱们一步一步执行这个循环。第一行从 ESI+403070 拷贝了一个字节到 AL 寄存器中，我们知道 ESI 等于 0，所以地址实际上就是 403070。咱们看看内存中这个地址里是什么。右键并选择 Follow in dump->constant，或者就右键 dump 窗口，选择 goto 并输入地址 403070。



如果仔细看的话，就会发现这就是上面传给 SetDlgTextItemA 的字符串参数。所以它就是将那串奇怪的字符串的第一个字符拷贝到 AL 中。

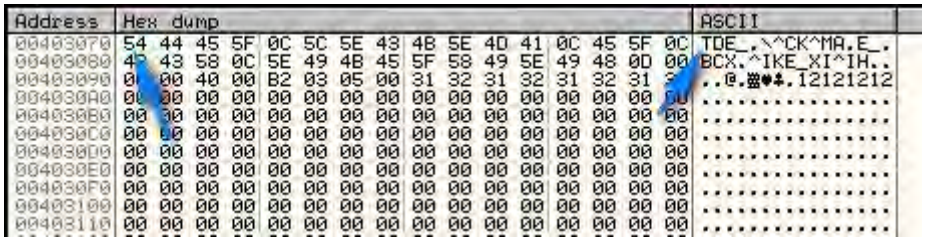
有件事你应该知道，许多汇编语言指令会按默认的使用方式使用某些寄存器，例如 ECX 被用来作为计数器，ESI 被用来作为源地址，EDI 被用来作为目的地址。本例中就是这样的。*

接下来，我们将该字符与 2C 进行 XOR，然后再将其存回原来的地址中：

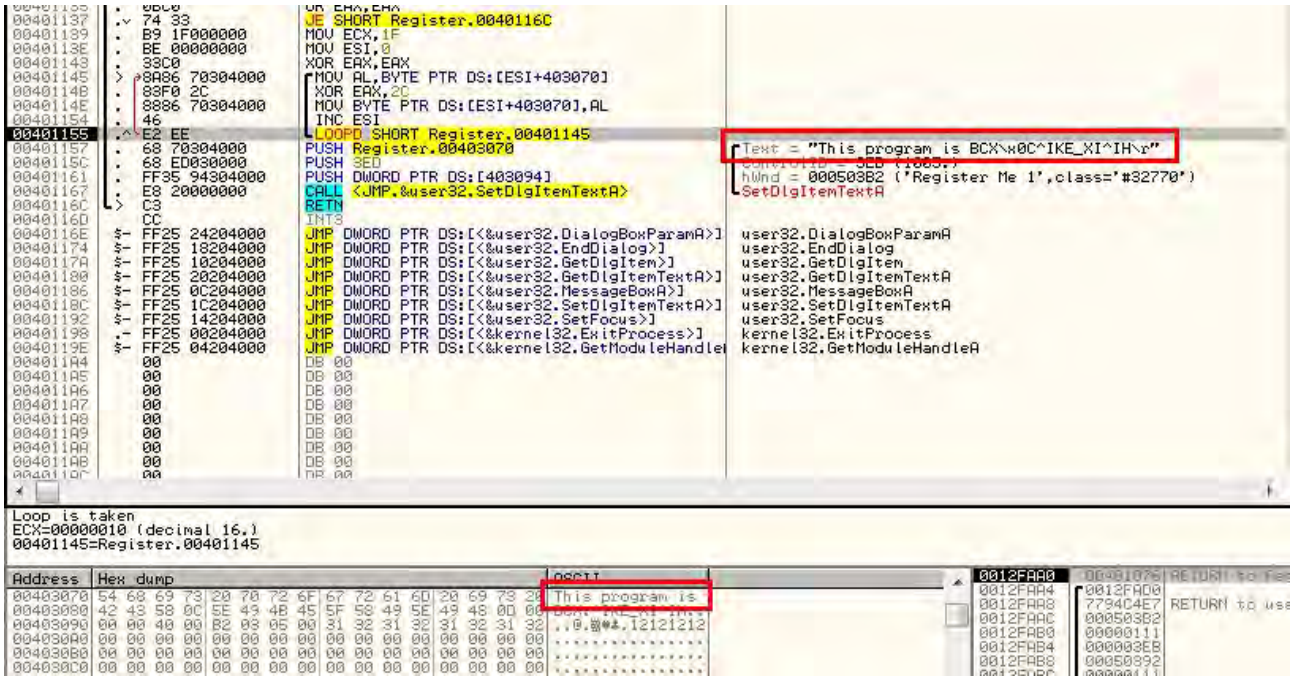


最后，给 ESI（源址寄存器）加 1，再做 LOOPD 操作。LOOPD 意思是 ECX 寄存器减 1，然后循环直至 ECX 为 0。也就是说，我们原来给 ECX 赋的值，十进制的 31，就是循环的次数。

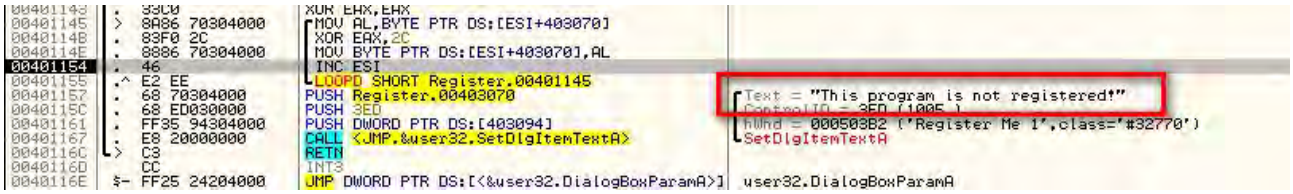
总的来看，该循环遍历奇怪字符串的每一个字符，将它们与 2C 进行 XOR 操作，再保存回原内存。这些操作将持续到 ECX 等于 0，或 31 次。单步执行一次 LOOPD 指令后回到顶部，然后看看数据窗口：



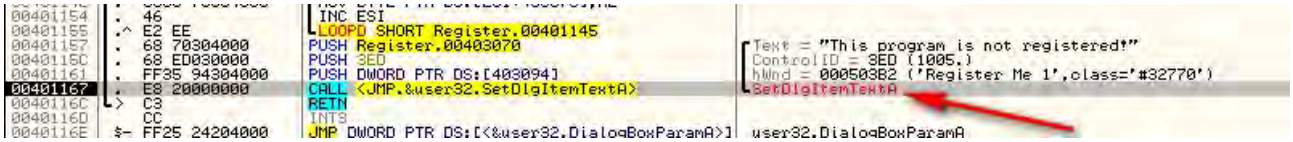
你会发现，字符串的第一个数字已经变了。原来的字符被执行 XOR 操作后变成了 “T”。如果你单步执行这个循环几次的话，会看到数据窗口中的字符串的变化。你也会发现传给 SetDlgTextItemA 的参数也变了：



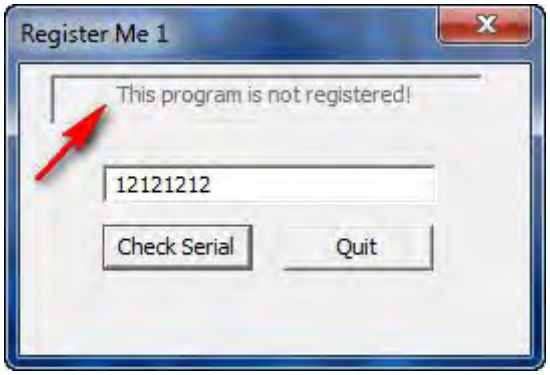
单步执行完这个循环，就会看到最后生成的消息，看起来相当属性呀，“This program is not registered!”。这和程序事实上还没有注册时主窗口中显示的消息是一样的：



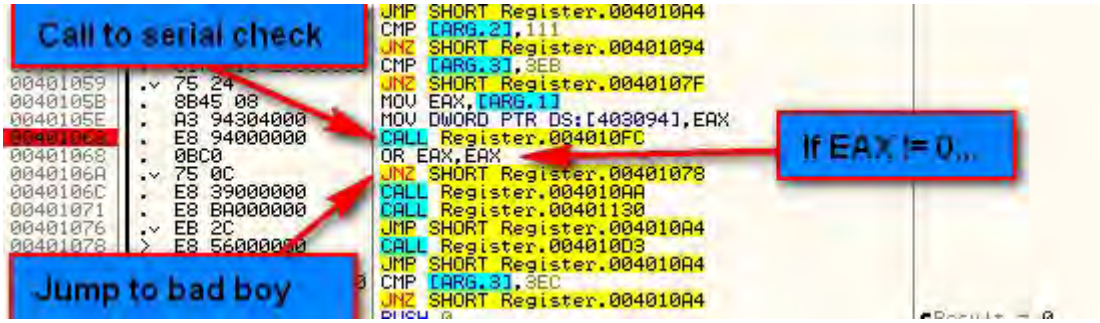
可以看到这个字符串变成了传递给 SetDlgTextItemA 的值，事实上用之前在那里的坏消息替换了已注册的好消息：



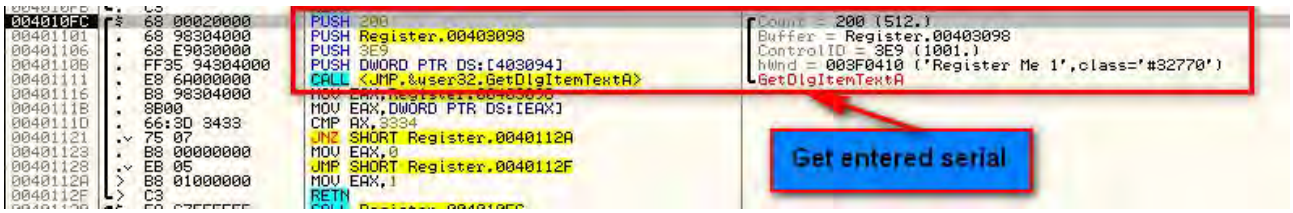
下面就是主窗口中显示的：



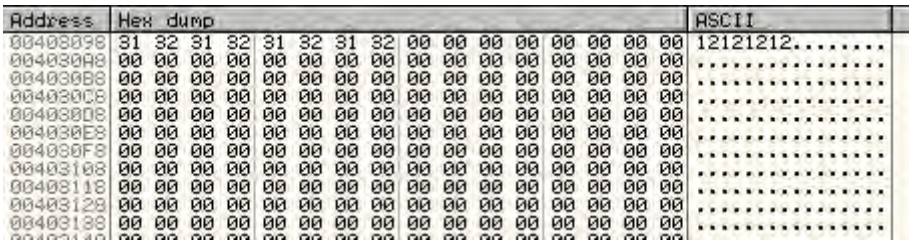
所以，现在我们知道了，给该应用打补丁的巧妙的方法是进入到序列号检测子程序，确保它总是返回正确的值，因为它不只是在第一次检测时被调用，而且在显示成功后再次被调用。再提醒你一下，序列号检测的相关 CALL 被调用，然对 eax 进行 0 测试。如果不是 0，就跳到坏消息，所以我们想让子程序返回 0！然后，序列号检测子程序再次被调用，如果它再次返回 0，那么我们的第二次检测就通过了：



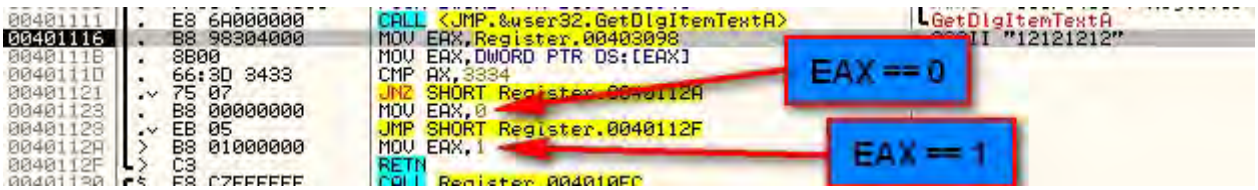
那么，咱们去序列号检测子程序那，看看能对它做些什么。子程序的开始调用了 GetDlgItemTextA，我们猜它就是获取我们输入的序列号。你可以在 401101（它指向的是放置文本的 buffer）的参数上右键，在数据窗口中跟随它：



我们单步步过 GetDlgItemTextA 指令后，就能在 buffer 中看到我们的序列号了：

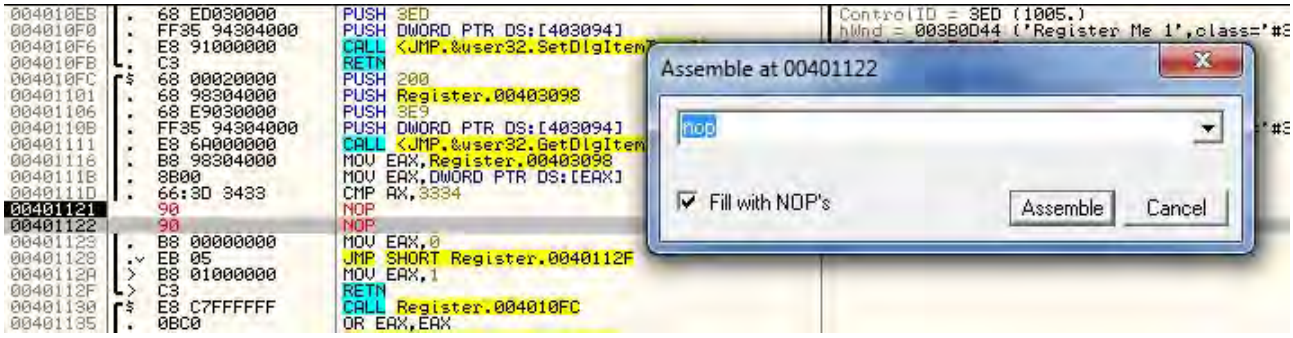


在它被保存到 buffer 后，该 buffer 的起始地址被拷贝到 EAX 中，随后该地址中的内容被拷贝到 EAX 中。就是将我们密码的前四个字节拷贝到 EAX 中。然后这几个字节与 3334 进行比较，如果不匹配，EAX 就被填充为 1（坏消息），否则就填充为 0（好消息）：



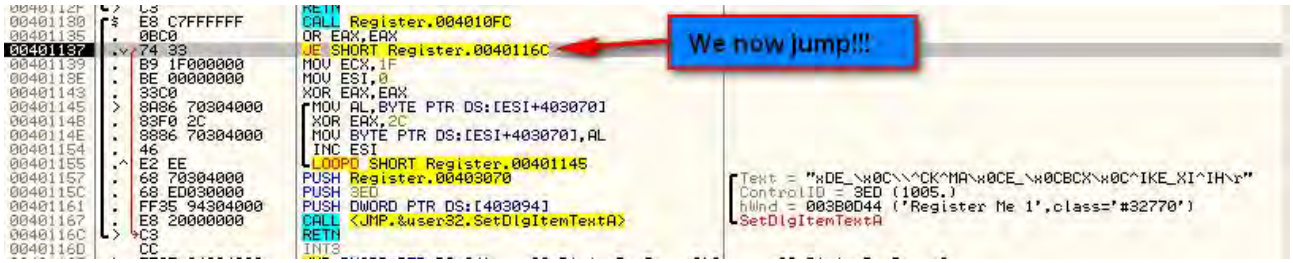
我们可以看到，做主要决定的是 401121 的 JNZ 指令：

这一行决定了在返回前，EAX 到底是 0 还是 1。所以我们要做的就是保证 EAX 总是等于 0:

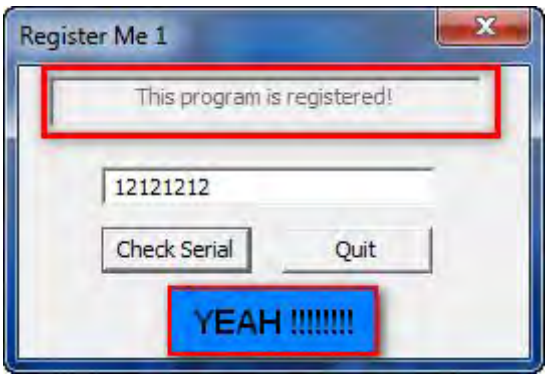


所以现在，代码将总是直接给 EAX 赋 0 值，然后直接跳转到返回处。运行下程序看看:

注意在对序列号检测子程序调用后，我们自然而然的就跳转到好消息那了:



在第二个检测点，我们也跳到了好消息那:

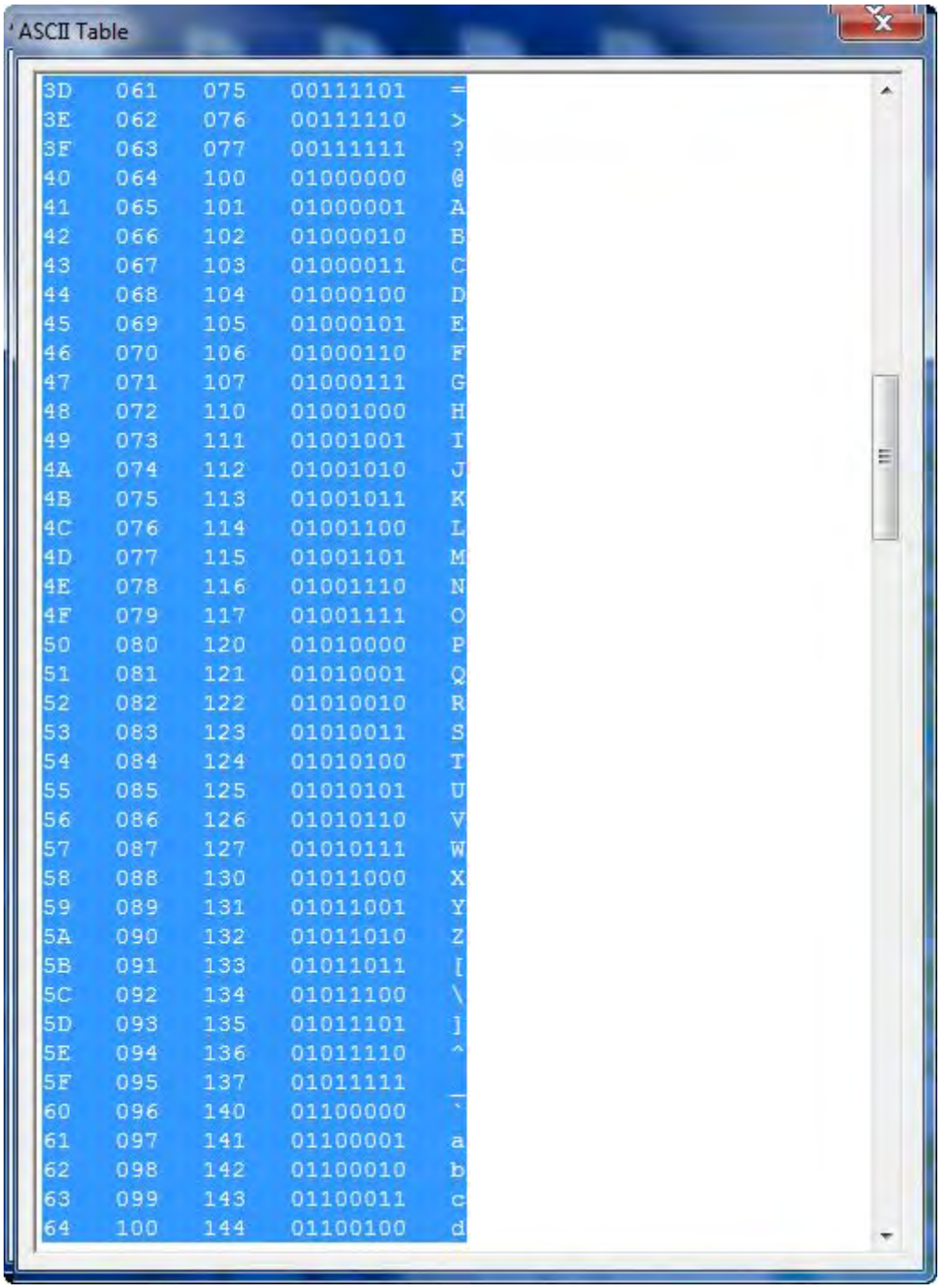


我们现在已经找到了一个注册该程序的补丁，无论你输入什么序列号都行  。

祝贺你！

四、ASCII 码表插件

你需要做的一件事是找出密码是什么（或对密码有什么样的要求）。给你些帮助，下载并安装“Ascii Table”插件，将其拷贝到插件目录。重启 Olly 后，选择“plugin”-> “Ascii table”就会显示一个表格。尽管它还有很多地方需要改进，不过它能让你快速查询 ASCII 值：



****如果有人想要主动更新或重做这个插件，我将永远感激。第一，那些文本不应该被选中，也不应该可编辑（我为什么要编辑 ASCII 码表？）。第二，让窗口大小可变真是件好事。如果有人做了，请告诉我，我欠你一辈子。****

第十三章：破解一个真正的程序

一、简介

本章我们打算不训练了，咱们来破解一个真正的程序。这个程序有个时间限制，过了这个时间，这个程序就不能用了。我们准备给它打补丁，让它认为是注册过的。目标文件在下载中有（我没有提及程序的名字，因为教程的目的不是为了拿到一个“破解版”程序，只是为了学习）。与所有的商业软件一样，如果你真的打算用它们，你真的应该考虑购买它。人们在软件中投入了大量的时间，他们应该得到补偿。为了不让这个系列教程成为关于“获得破解版软件”的东东，我试着找了一个没有人真想要的程序，所以我下载了这个软件，它是上周 Download. com 中拥有最少下载量的软件。作为一个完全诚实的人，在本章中破解了这个程序以后，我很喜欢这个程序，所以我买了一个注册码，现在我心安理得的用它（译者注：作者真是活雷锋，其实咱们都是搞技术的，或多或少都写过代码，尊重软件作者，为他们的劳动付费，其实就是尊重自己。实在不愿意花钱的，就用免费替代软件行了，我一般喜欢用开源免费软件。多说了几句哈）。只是告诉你，你不能通过下载量来判断一个应用。

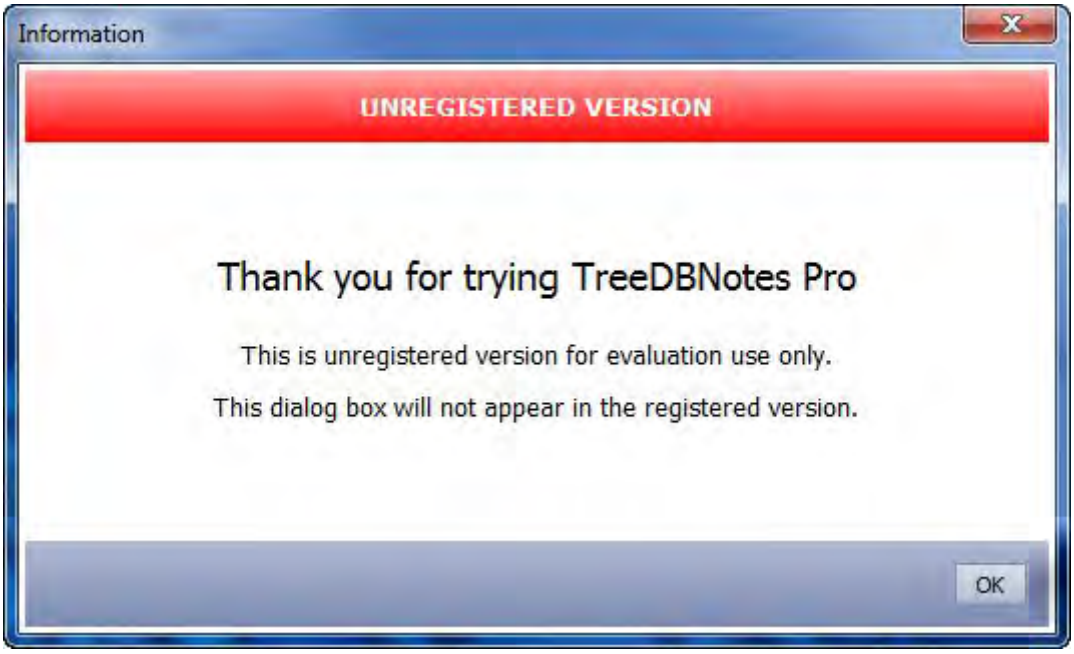
你可以在教程页下载相关文件及本文的 PDF 版。

好，咱们继续....

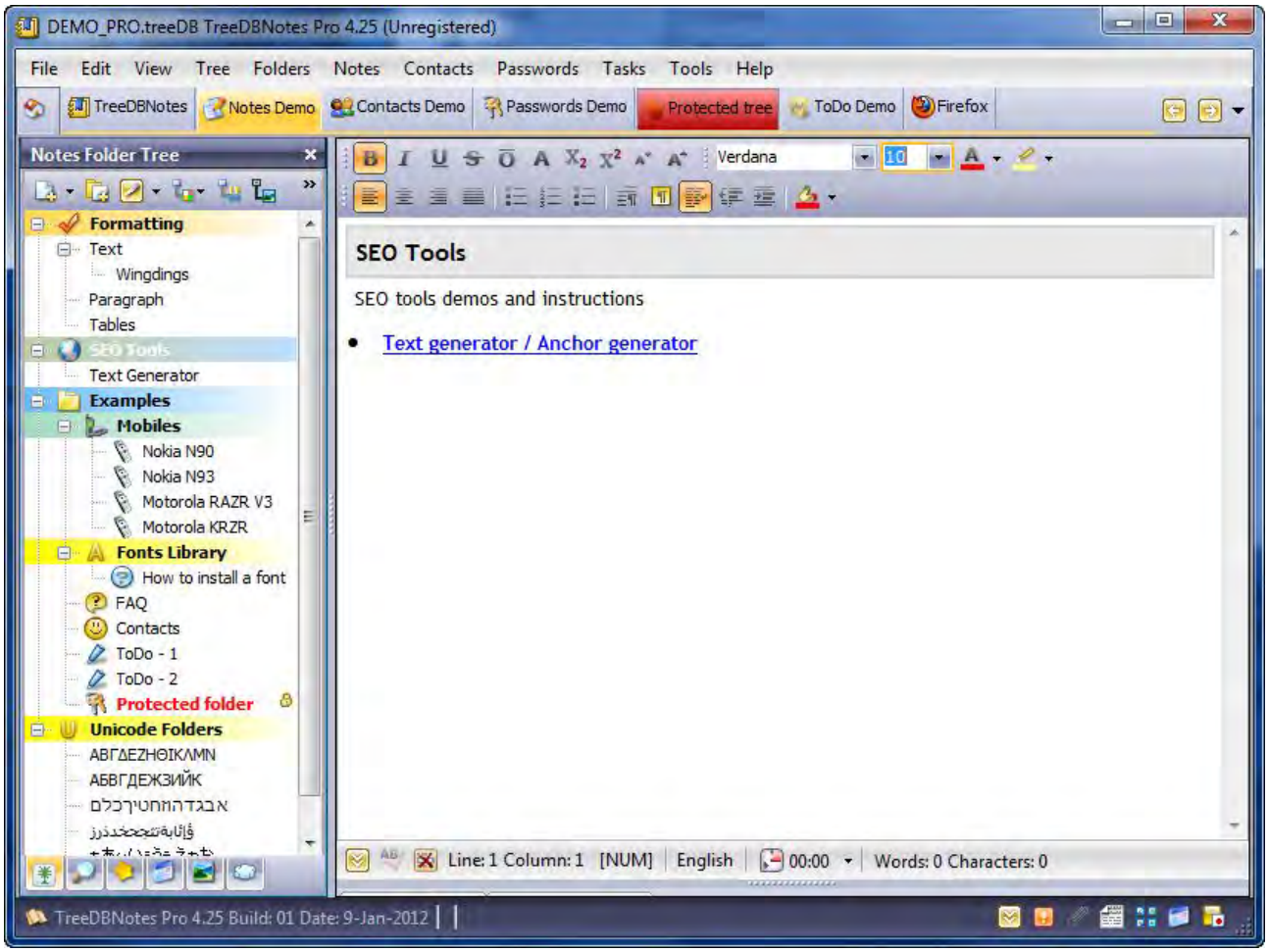
二、研究该应用

首先安装软件。安装完成后，会弹出下面这个窗口：

让“Run the app”保持勾选状态，看看会遇到什么：

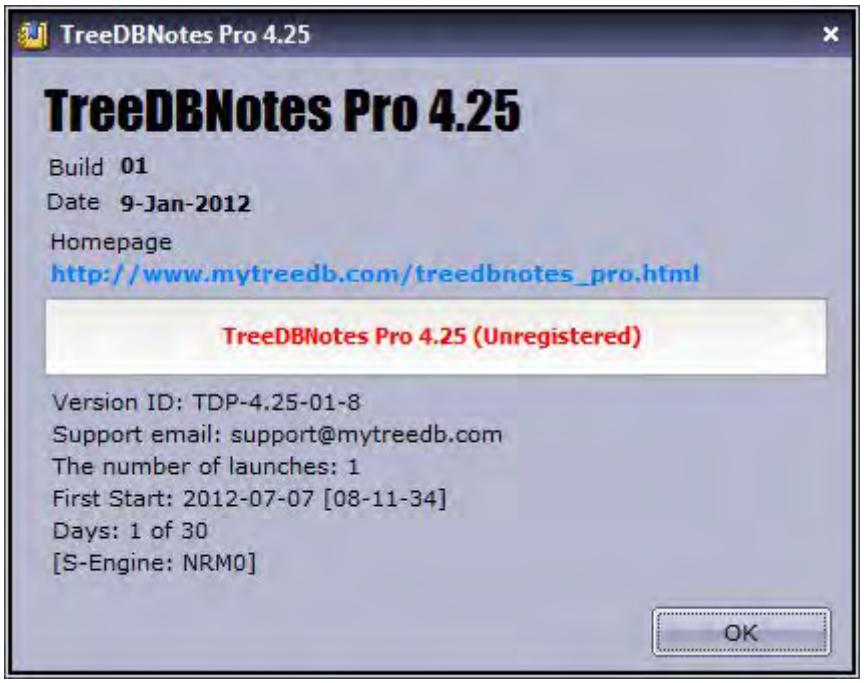


好吧，看起来不是很好啊。我们注意到这里有几个字符串可能会有帮助，“unregistered”、“evaluation”、“registered”等等。点 OK，然后弹出主界面：

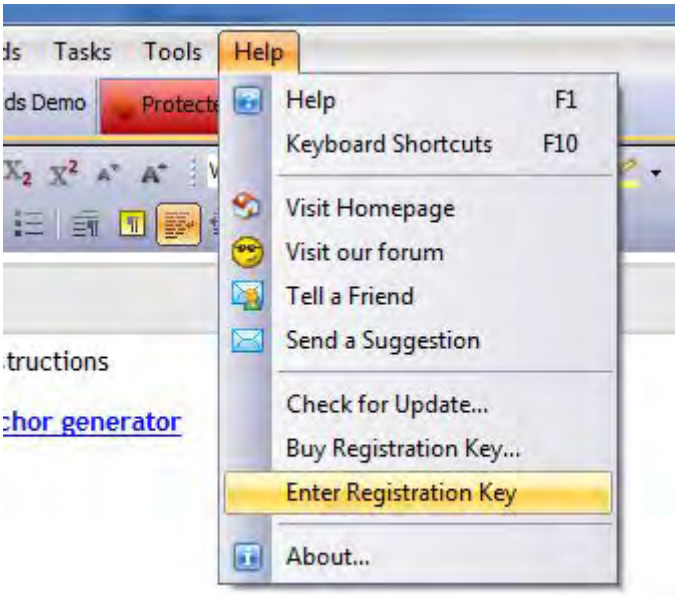


注意，标题栏中显示的是“unregistered”。通常，我注意一个程序的另一个地方就是它的关于对话框。它通常都包含有字符串，以及用于逆向的思路。

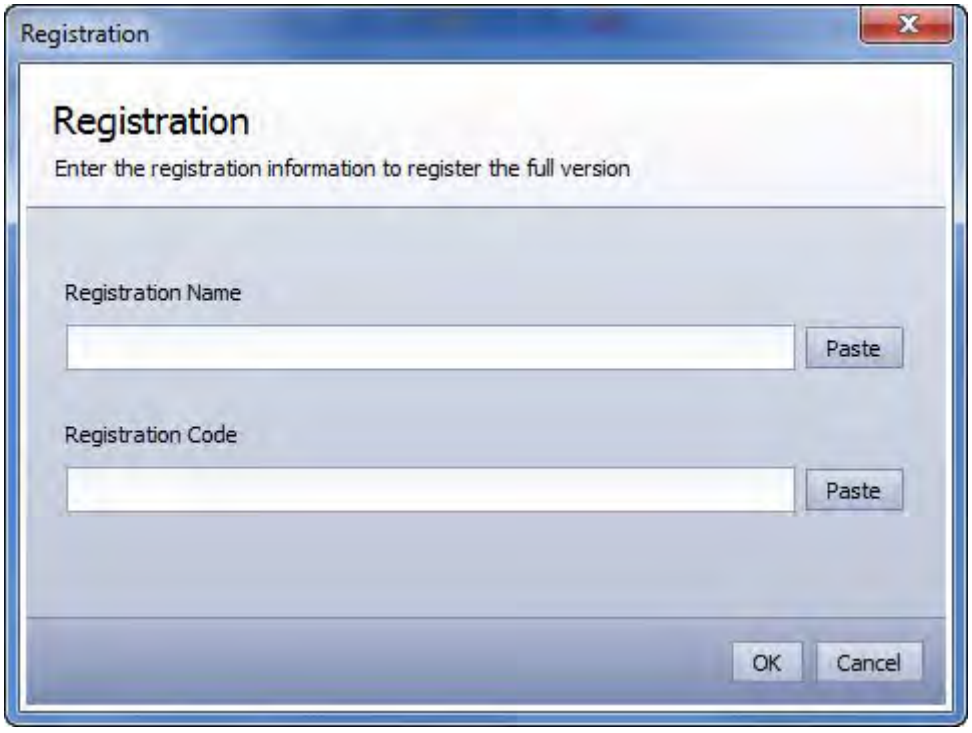
这时候，我们寻找关键字、可识别的方法调用，以及类似的东西。这样的工作你做的越多，就会有更多的线索。



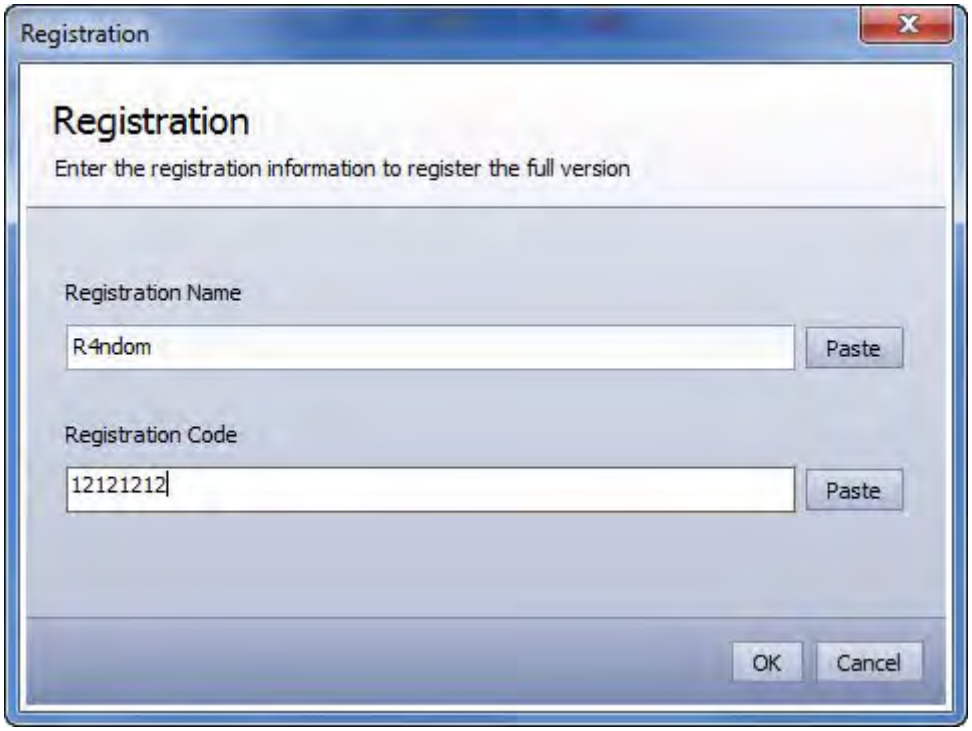
这里我们又看见了“unregistered”。我通常做的下一件事是，找找看有没有什么地方用来输入注册码。如果“搜索字符串”这招不好用的话，那么对于渗透来说这是一个好入手点：



下面是输入注册码的地方：



输入一个试试，看看什么情况:



点击 OK:

唉！我好像从来就没有输对过。好吧，对于我们当前搜集到的信息来说，我们有一个相当好的方法，Olly 载入程序：

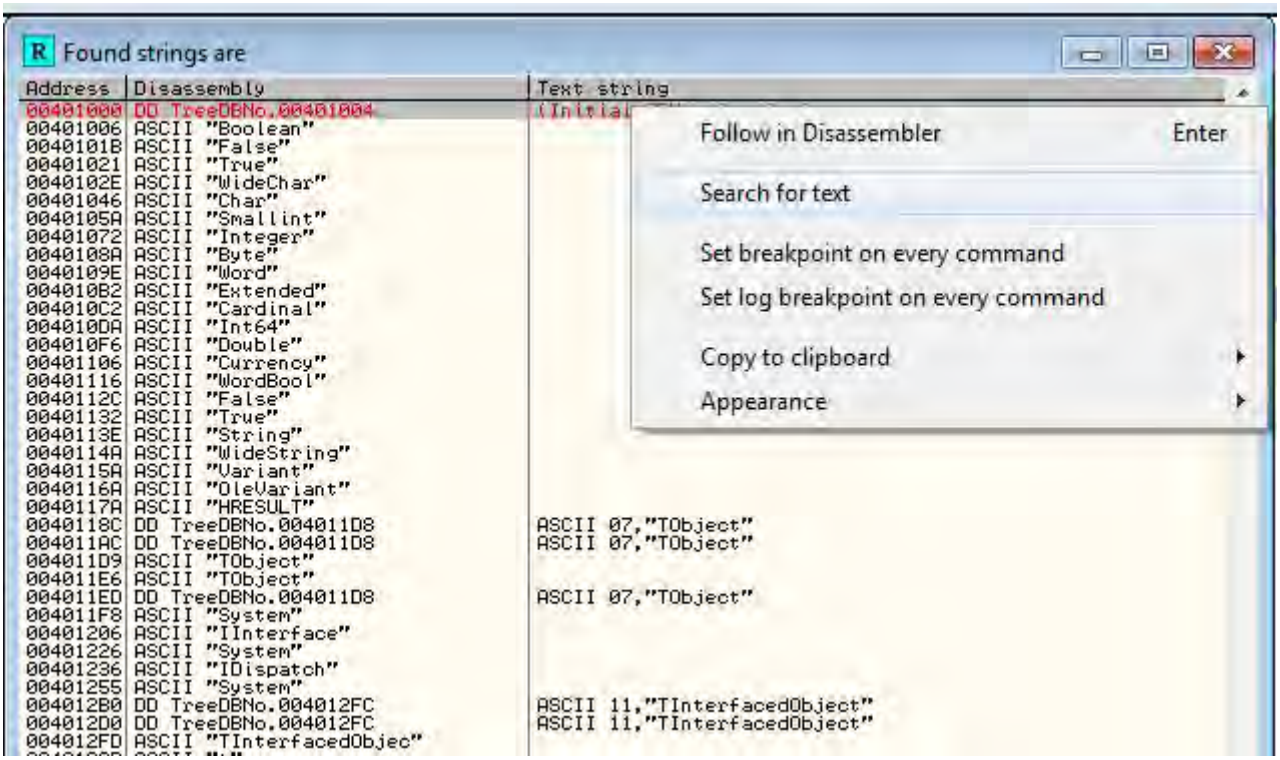
```
009F6098  $  55              PUSH EBP
009F6099  .  8BEC            MOV EBP,ESP
009F609B  .  83C4 F0         ADD ESP,-10
009F609E  .  53              PUSH EBX
009F609F  .  B8 204F9F00     MOV EAX,TreeDBNo.009F4F20
009F60A4  .  E8 CB16A1FF     CALL TreeDBNo.00407774
009F60A9  .  8B1D D006A200   MOV EBX,DWORD PTR DS:[A206D0]      TreeDBNo.00A21BF8
009F60AF  .  33C9            XOR ECX,ECX
009F60B1  .  B2 01           MOV DL,1
009F60B3  .  A1 7CE29700     MOV EAX,DWORD PTR DS:[97E27C]
009F60B8  .  E8 275EAAFF     CALL TreeDBNo.0049BEE4
009F60BD  .  8B15 5405A200   MOV EDX,DWORD PTR DS:[A20554]      TreeDBNo.00A26764
009F60C3  .  8902            MOV DWORD PTR DS:[EDX],EAX         kernel32.BaseThreadInit
009F60C5  .  A1 5405A200     MOV EAX,DWORD PTR DS:[A20554]
009F60CA  .  8B00            MOV EAX,DWORD PTR DS:[EAX]
009F60CC  .  E8 6BA1AAFF     CALL TreeDBNo.004A023C
009F60D1  .  A1 5405A200     MOV EAX,DWORD PTR DS:[A20554]
009F60D6  .  8B00            MOV EAX,DWORD PTR DS:[EAX]
009F60D8  .  8B10            MOV EDX,DWORD PTR DS:[EAX]
009F60DA  .  FF92 80000000   CALL DWORD PTR DS:[EDX+80]
009F60E0  .  8B03            MOV EAX,DWORD PTR DS:[EBX]
009F60E2  .  E8 45D8AAFF     CALL TreeDBNo.004A392C
009F60E7  .  8B03            MOV EAX,DWORD PTR DS:[EBX]
009F60E9  .  BA CC619F00     MOV EDX,TreeDBNo.009F61CC          ASCII "TreeDBNotes"
009F60EE  .  E8 31D4AAFF     CALL TreeDBNo.004A3524
009F60F3  .  8B0D 7C09A200   MOV ECX,DWORD PTR DS:[A2097C]      TreeDBNo.00A2632C
009F60F9  .  8B03            MOV EAX,DWORD PTR DS:[EBX]
009F60FB  .  8B15 D4AF8E00   MOV EDX,DWORD PTR DS:[8EAFD4]      TreeDBNo.008EB020
009F6101  .  E8 3ED8AAFF     CALL TreeDBNo.004A3944
009F6106  .  8B0D 0CFFA100   MOV ECX,DWORD PTR DS:[A1FF0C]      TreeDBNo.00A26874
009F610C  .  8B03            MOV EAX,DWORD PTR DS:[EBX]
009F610E  .  8B15 98229D00   MOV EDX,DWORD PTR DS:[9D2298]      TreeDBNo.009D22E4
009F6114  .  E8 2BD8AAFF     CALL TreeDBNo.004A3944
009F6119  .  8B0D 0000A200   MOV ECX,DWORD PTR DS:[A20000]      TreeDBNo.00A267A4
009F611F  .  8B03            MOV EAX,DWORD PTR DS:[EBX]
009F6121  .  8B15 D0999800   MOV EDX,DWORD PTR DS:[9899D0]      TreeDBNo.00989A1C
009F6127  .  E8 18D8AAFF     CALL TreeDBNo.004A3944
009F612C  .  8B0D F800A200   MOV ECX,DWORD PTR DS:[A200F8]      TreeDBNo.00A248F0
009F6132  .  8B03            MOV EAX,DWORD PTR DS:[EBX]
009F6134  .  8B15 58148B00   MOV EDX,DWORD PTR DS:[8B1458]      TreeDBNo.008B14A4
009F613A  .  E8 05D8AAFF     CALL TreeDBNo.004A3944
009F613F  .  8B0D 1403A200   MOV ECX,DWORD PTR DS:[A20314]      TreeDBNo.00A263E4
009F6145  .  8B03            MOV EAX,DWORD PTR DS:[EBX]
009F6147  .  8B15 28FB9100   MOV EDX,DWORD PTR DS:[91FB28]      TreeDBNo.0091FB74
009F614D  .  E8 F2D7AAFF     CALL TreeDBNo.004A3944
009F6152  .  8B0D 5802A200   MOV ECX,DWORD PTR DS:[A20258]      TreeDBNo.00A263F0
009F6158  .  8B03            MOV EAX,DWORD PTR DS:[EBX]
009F615A  .  8B15 B8019200   MOV EDX,DWORD PTR DS:[9201B8]      TreeDBNo.00920204
009F6160  .  E8 DFD7AAFF     CALL TreeDBNo.004A3944
009F6165  .  8B0D 5CFFA100   MOV ECX,DWORD PTR DS:[A1FF5C]      TreeDBNo.00A26360
009F616B  .  8B03            MOV EAX,DWORD PTR DS:[EBX]
009F616D  .  8B15 ACE78E00   MOV EDX,DWORD PTR DS:[8EE7AC]      TreeDBNo.008EE7F8
009F6173  .  E8 CCD7AAFF     CALL TreeDBNo.004A3944
009F6178  .  8B0D 2802A200   MOV ECX,DWORD PTR DS:[A20228]      TreeDBNo.00A26358
009F617E  .  8B03            MOV EAX,DWORD PTR DS:[EBX]
009F6180  .  8B15 A8E68E00   MOV EDX,DWORD PTR DS:[8EE6A8]      TreeDBNo.008EE6F4
009F6186  .  E8 B9D7AAFF     CALL TreeDBNo.004A3944
009F618B  .  8B0D 340BA200   MOV ECX,DWORD PTR DS:[A20B34]      TreeDBNo.00A2676C
009F6191  .  8B03            MOV EAX,DWORD PTR DS:[EBX]
009F6193  .  8B15 E8EA9700   MOV EDX,DWORD PTR DS:[97EAE8]      TreeDBNo.0097EB34
009F6199  .  E8 A6D7AAFF     CALL TreeDBNo.004A3944
009F619E  .  A1 0000A200     MOV EAX,DWORD PTR DS:[A20000]
009F61A3  .  8B00            MOV EAX,DWORD PTR DS:[EAX]
009F61A5  .  E8 1E71FAFF     CALL TreeDBNo.0099D2C8
009F61AA  .  A1 5405A200     MOV EAX,DWORD PTR DS:[A20554]
009F61AF  .  8B00            MOV EAX,DWORD PTR DS:[EAX]
009F61B1  .  E8 7EA0AAFF     CALL TreeDBNo.004A0234
009F61B6  .  8B03            MOV EAX,DWORD PTR DS:[EBX]
009F61B8  .  E8 07D8AAFF     CALL TreeDBNo.004A39C4
```

你可能已经注意到了，这看起来和我们已经见过的大部分应用有点不一样。看起来有辣么多的 CALL 指令，没有那些典型的 Windows 设置的玩意儿（像

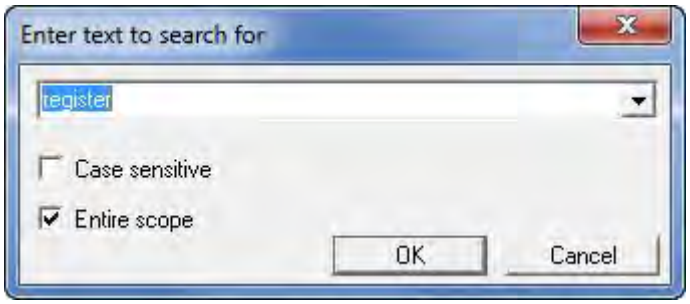
RegisterClass. . .)。这是一个好的标志，说明程序是用 Delphi 写的。Delphi 在程序中会使用大量的 CALL。我们可以通过运行一个 ID 程序来确定，不过我打算在后面的教程中讨论。也有一些专门的工具用来处理 Delphi 程序，不过幸运星是本章我们不需要用专用工具（虽然我们会接触到它们）。

三、寻找补丁

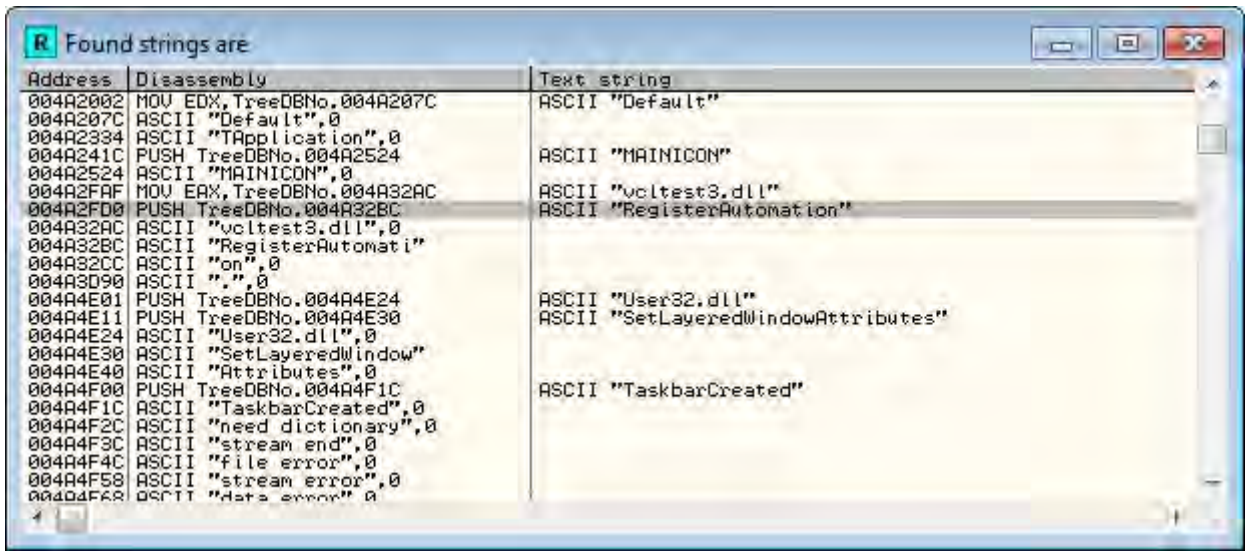
试试咱们的字符串搜索。右键，选择 “Search for” -> “All referenced text strings” ，将会弹出搜索窗口。滚动到顶部然后右键，选择 “Search for text”:



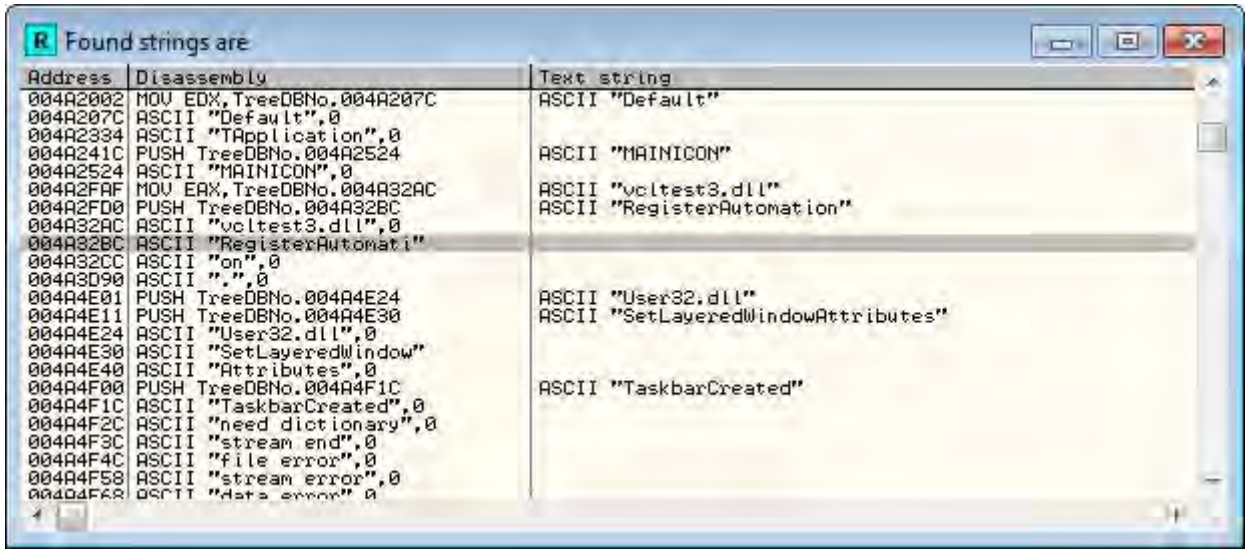
弹出文本搜索对话框。现在我们注意到 “registration” 和 “registered” 很早就用到了，所以咱们就搜它们。通常在这种情况下，因为是第一次搜索，我会搜 “regist”，因为包含了这两个单词，而且也从来没有让我失望过（我猜没有多少程序会使用单词 “register” ）。不要勾选 “Case sensitive”，选中 “Entire scope”，然后点 OK:



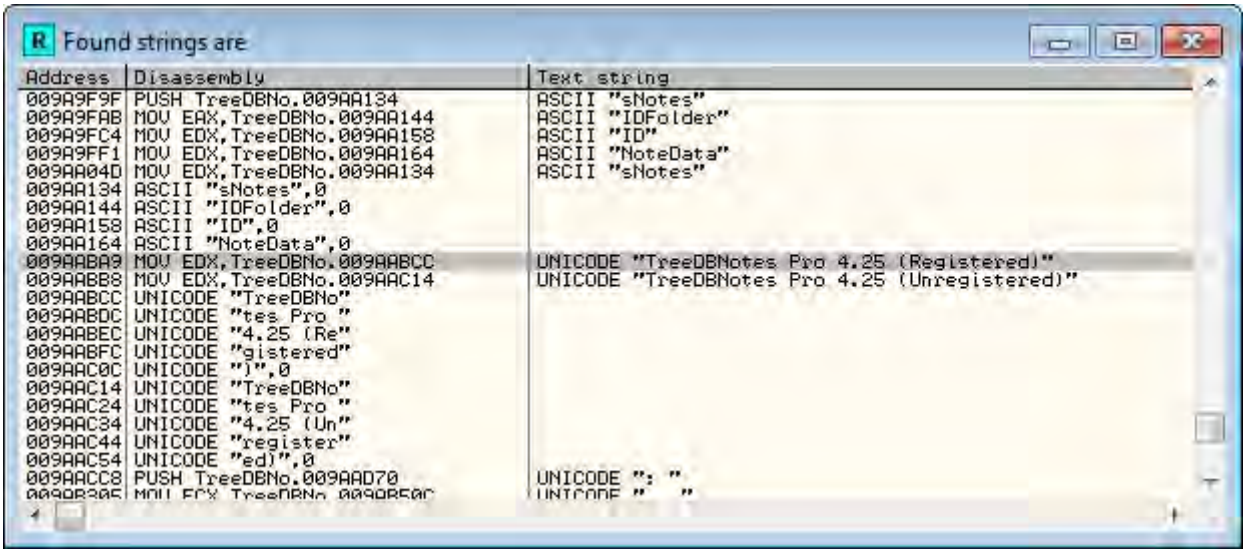
第一个命中的看起来没啥前途，按 CTRL+L 继续搜:



注意，这次找到的就是我们第一次搜到的。因为第一次命中的是在字符串被压到堆栈的地方，第二次才是字符串“RegisterAutomation”在内存中的真实的数据。因为在第二列中没有指令所以可以分辨出来，反而有个 ASCII 字样。你遇到的大多数字符串都有两个版本，一个是字符串被访问的地方，另一个就是字符串真正所在的地方:

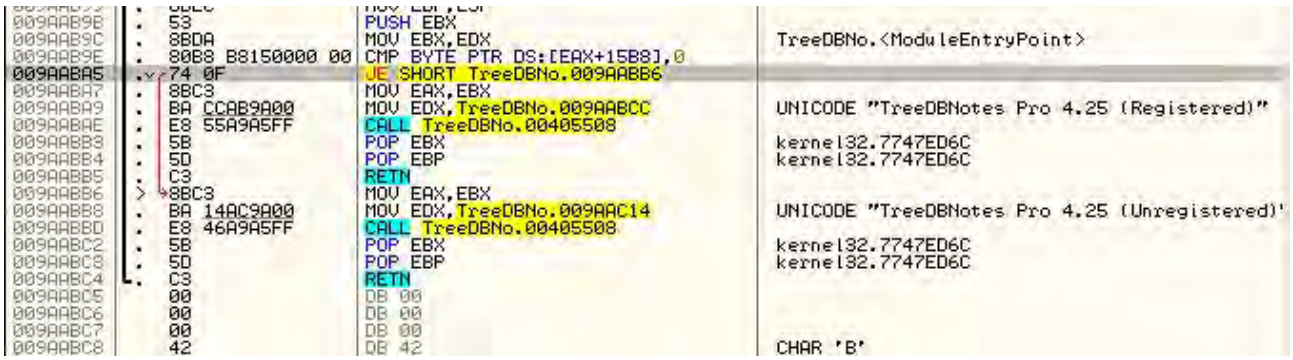


如果你再按一次 CTRL+L，我们会遇到另一个没前途的字符串。一直按 CTRL+L 直到来到下面这个地方:

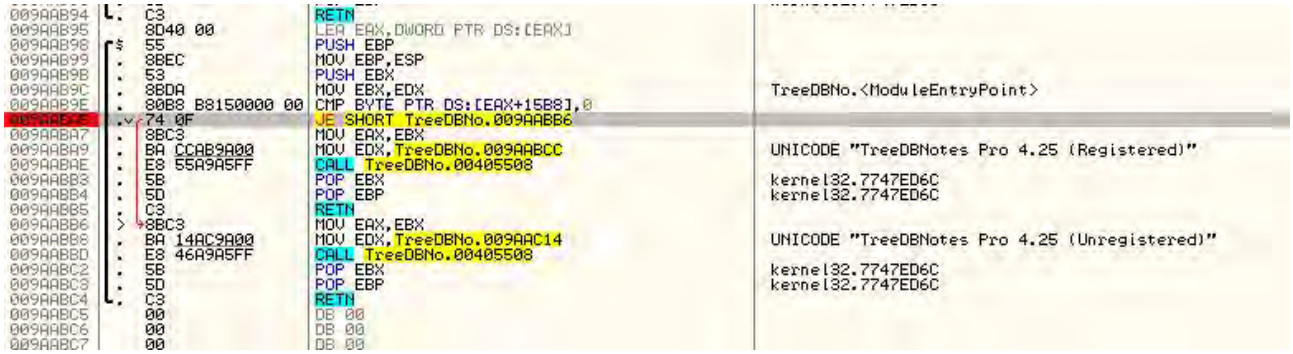


这回看起来好多了。它将会在程序启动过程中的某个时刻出现，它会检测我们有没有注册，然后根据检测的结果来决定窗口的标题栏显示注册还是没注册。这是我们开始干活的好地方。双击有“registered”的那行，咱们就会跳到相应的代码那:

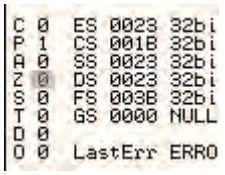
首先我们能看到字符串是在 9AABA9 那，还能看到字符串存储在内存的 9AABCC 处。第二，要注意到是两个字符串是在同一个方法中，在它们的上面有个一个条件跳转。点击 9AABA5 处的条件跳转:



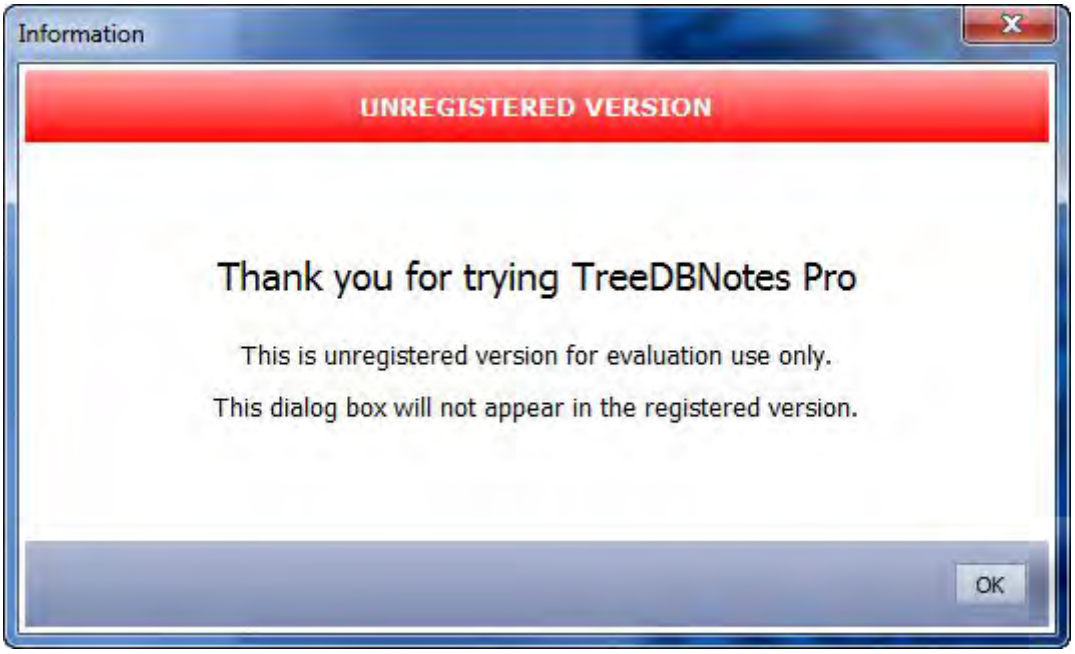
我们能够看到如果结果相等，就跳到“Unregistered”那里。很明显，不能让它跳。咱们在 JE 指令那设置一个 BP，启动应用:



Olly 就会断在那行，你会发现我们就要跳到坏消息那了。咱们得修改下:

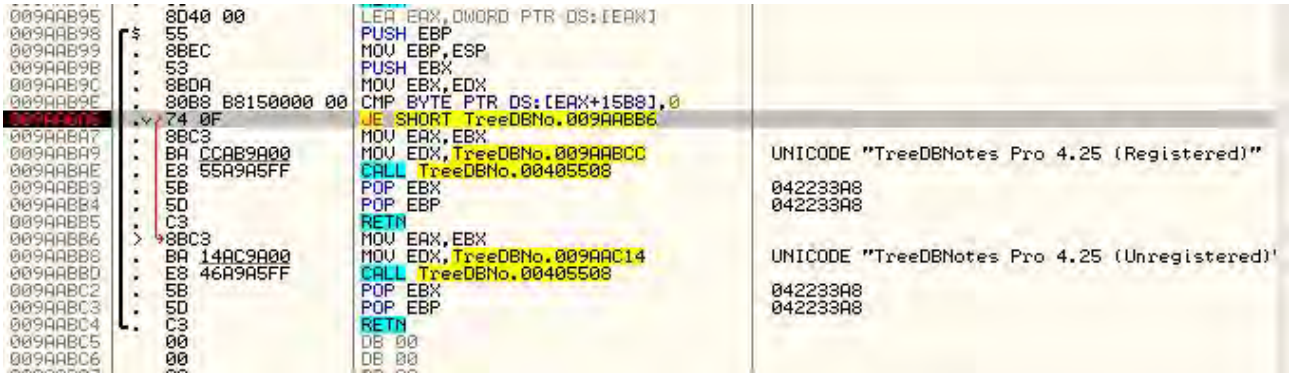


运行程序。Olly 会再次断在同一行，并准备跳到坏消息那。咱们再次就 0 标志位置 0，然后运行程序。又来了一遍，清除 0 标志位后，我们最终得到如下的反馈:



所以那样做是不起作用的。给那个检测点打补丁不会注册成功，如果你点 OK 并再一次将标志位置 0，你会发现主窗口的“unregistered”没有了：

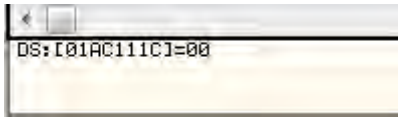
那么，我们知道至少没有跟踪错。我们准备做的是步入到下一“层”，做深入的研究。重启应用，然后断在了我们的断点处，咱们再多做些研究：



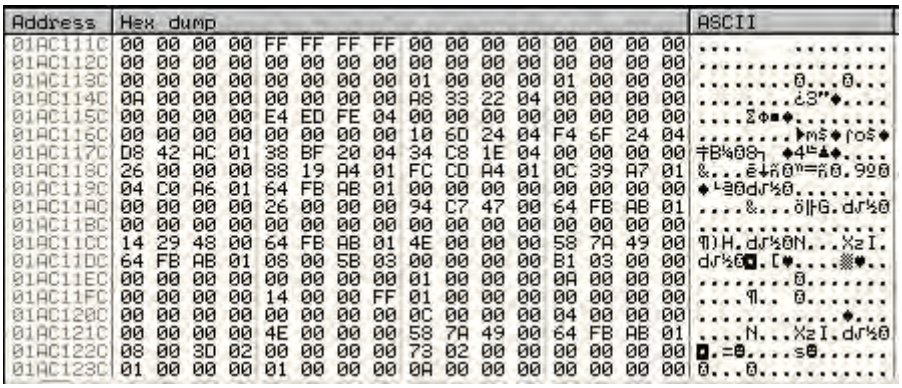
在比较的前面并没有 CALL，但是在 JE 指令的前面 9AAB9E 处有个比较：

CMP BYTE PTR DS: [EAX+15B8], 0

所以，这个比较的结果决定了我们是注册还是未注册。EAX+15B8 只是一个内存地址，在这里它是一个全局变量，因为是以 DS: 打头的。我们希望这是程序检测注册与否的唯一一个检测点。如果不是的话，我们就需要找出其他检测点的位置。点选比较指令，就可以看到 EAX+15B8i 是什么：



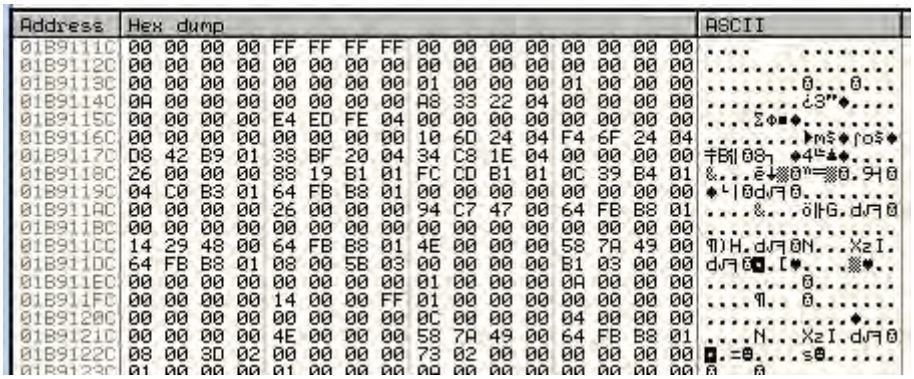
在地址上右键，选择 “Follow in dump”：



****你的地址几乎肯定和我的不一样。这个没事。跟着教程，将我的地址替换成你的地址，一样跑的好好的。****

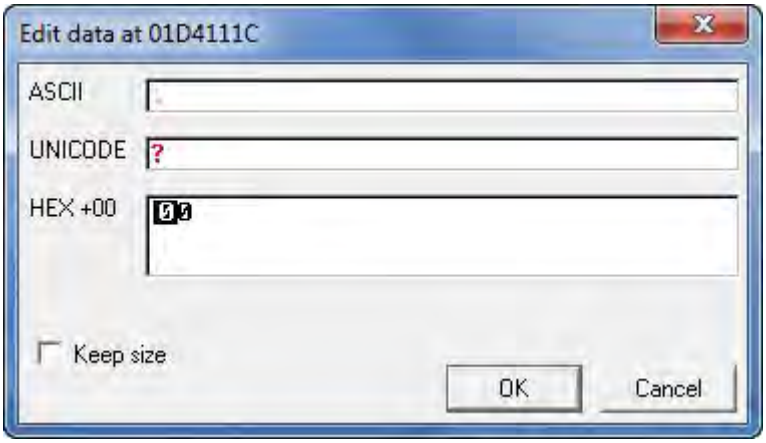
这里我们能看到该地址被用于检测注册与否，就是 1AC111C 处（至少我这里是这样）的第一个 00。意思是如果此内存位置的内容是除了 0 以外的任何数据，那么就假定我们已经注册。这也意味着，程序中可能有别的子程序检查该内存位置，这就是为什么主窗口显示 “Registered”，而程序的其他部分知道我们没有注册。因为我们只是在检查了内存内容后绕过了这个子程序的自然流程，其他子程序的检测却没有绕过。

首先，咱们把这个内存地址设置为非 0，那么我们知道至少这个子程序将会按照我们想要的方式工作。在比较那行（9AAB9E）设置一个断点，将其他断点删掉。重启应用后 Olly 就断下来了。在比较那行上右键，选择 “Follow in dump” -> “Memory location”，因为 Olly 会在我们重启应用的时候重置数据窗口。你可能已经注意到了，比较指令检查的内存地址这次变了：

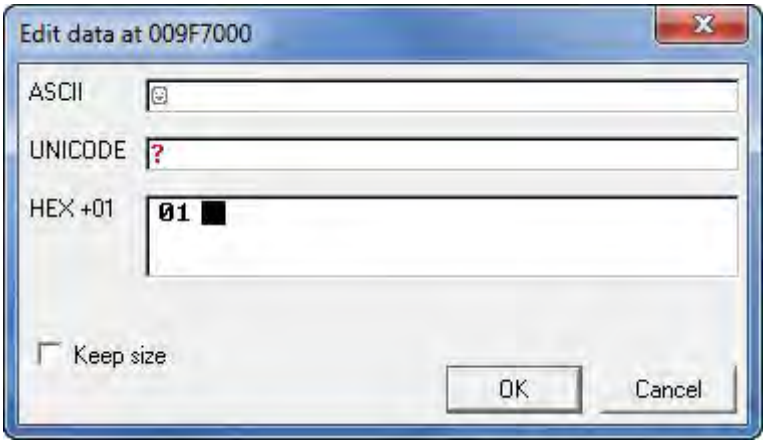


第一次是 1AC111C，现在是 01B9111C。你的和我的会不一样，你只需要注意第二次的就行，存储 已注册/未注册 标志的内存地址不同。

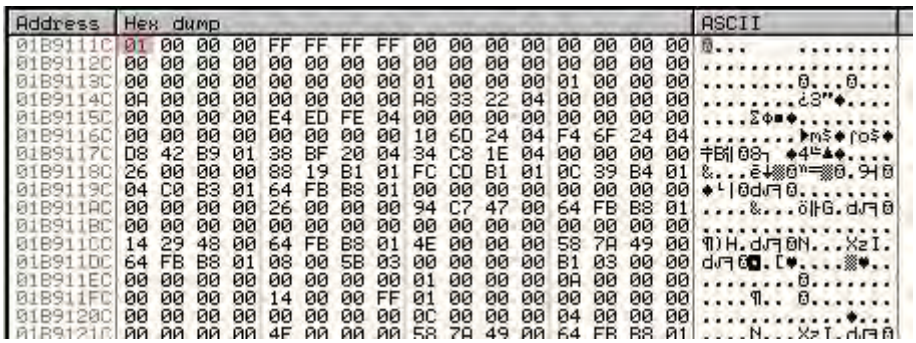
点击数据窗口中的“00”（在我的数据窗口中是 1B9111C），右键选择“Binary” -> “Edit”：



咱们输入 01：

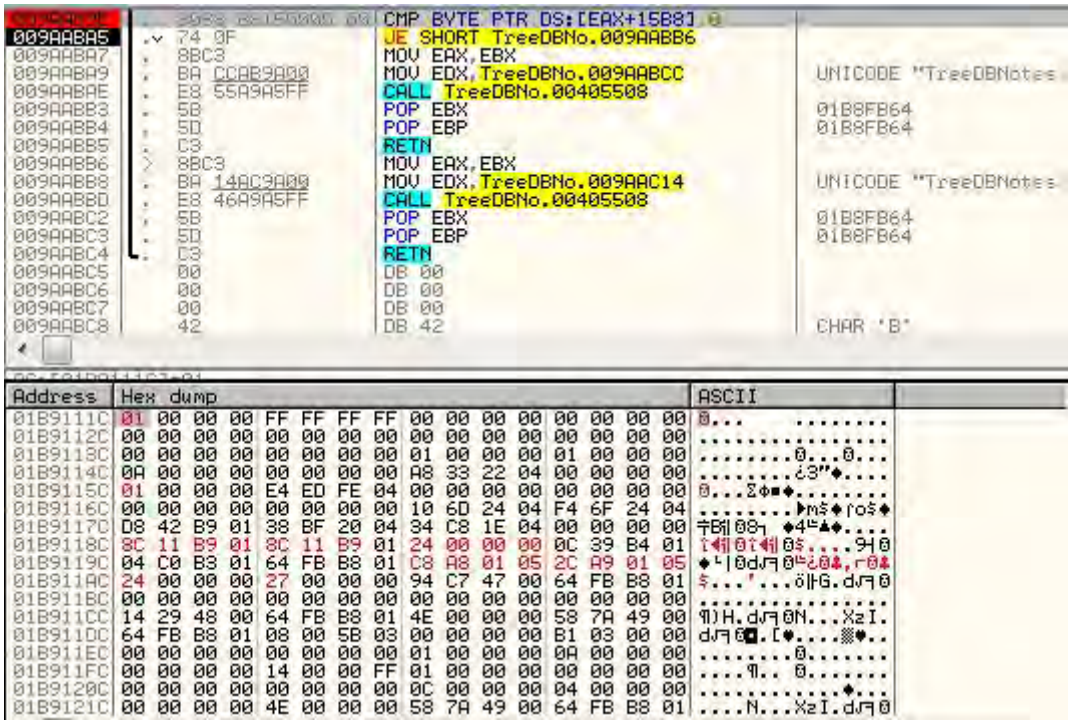


注意数据窗口中的内容已经更新了：

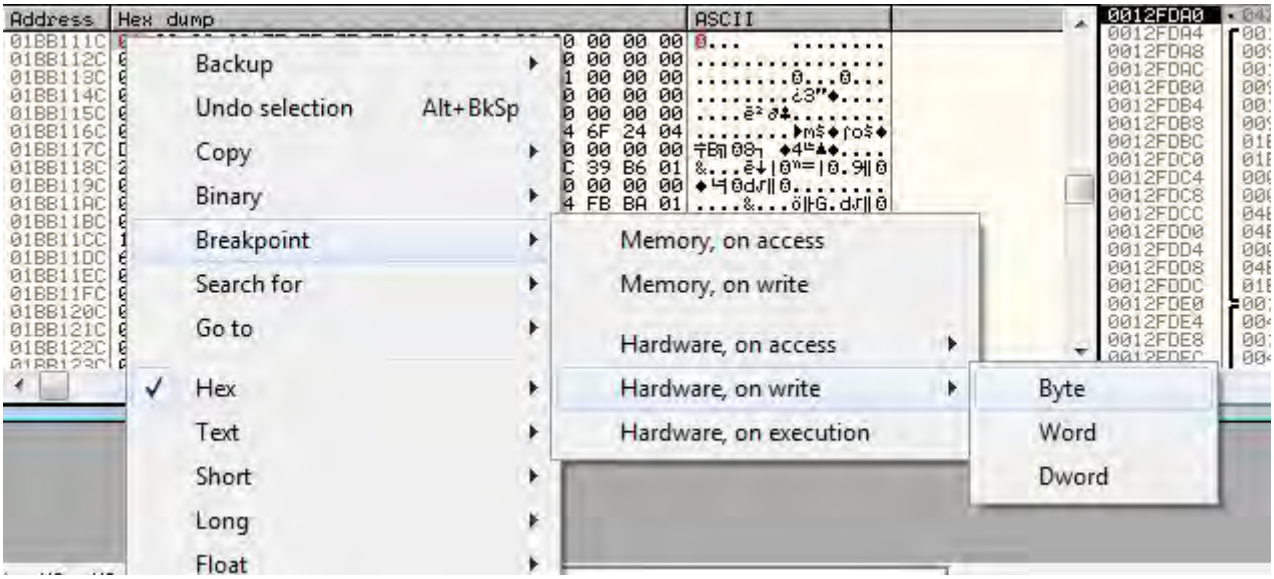


继续运行程序直到再一次断下来。你会发现内存中的内容又变回了 0，我们将再一次跳到坏消息那。这意味着程序的某个地方，做了第二次检测并将注册与否的标志重置为 0。我们需要做的就是找出在哪里重置的，确保不会再被重置。要这样做的话，在该内存位置设置一个硬件断点，当程序向该内存位置写数据时让 Olly 断下来。之所以选择“写”，是因为某个地方向该内存写了 0。

重启应用直到它断下来。右键比较指令，选择“Follow in dump”，因为 Olly 又重置了数据窗口。用二进制编辑方法将第一个内存位置修改为 01。注意它现在的地址又换了：

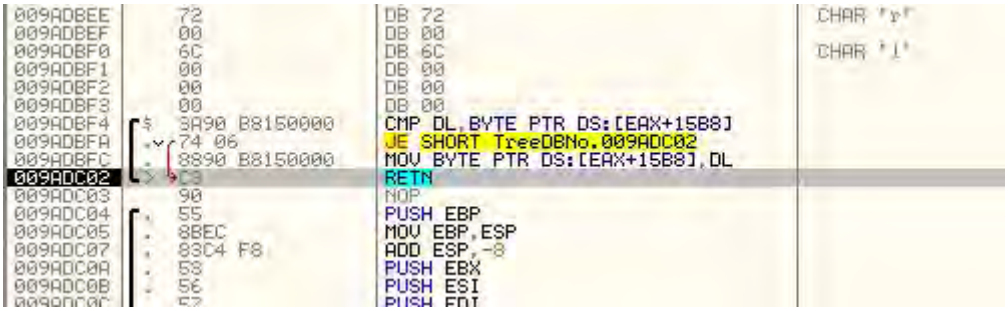


右键数据窗口中的第一个值，选择“Breakpoint” -> “Hardware, on write” -> “byte”：



在逆向一个程序时，我通常留在硬件断点，因为它们很难被应用检测到。我选择“byte”是因为我们想追踪的就一个字节。

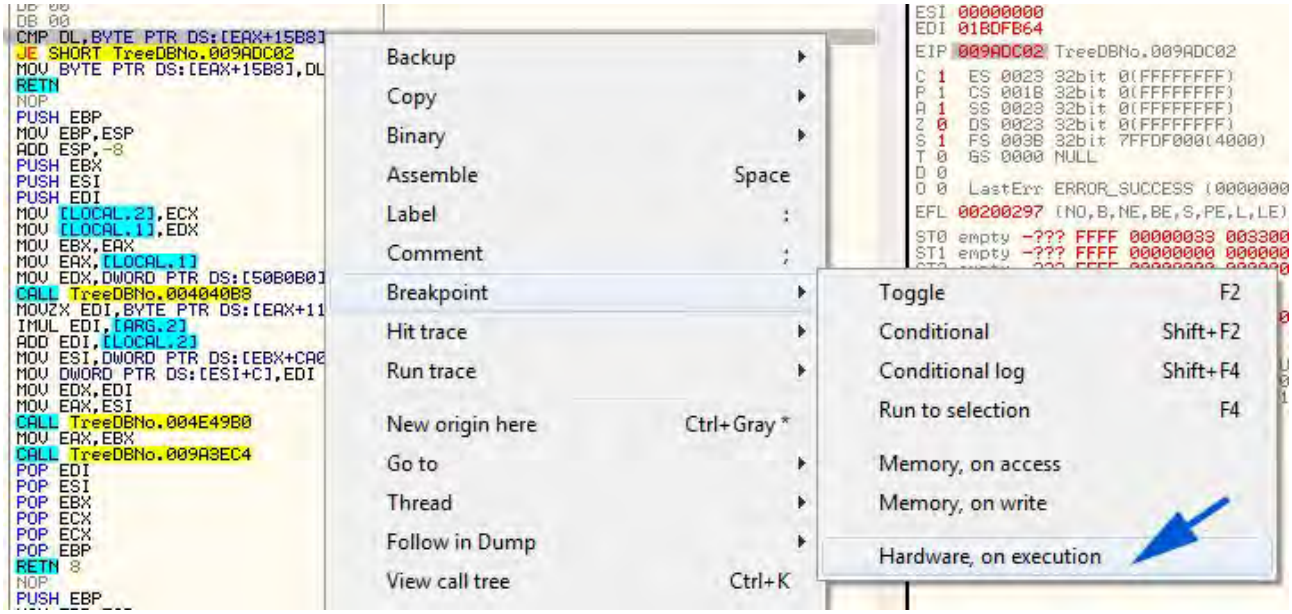
运行程序。Olly 会再次断在普通断点，你会发现我们输入的 01 仍然在那里，所以到目前为止还不错。再运行，Olly 会断在一个新地方：



如果你看 OllyDbg 左下角的话，会发现我们断在了硬件断点。

四、给程序打补丁

现在，咱们来研究研究这块代码。第一条指令是将 DL 与我们刚才编辑的内存内容进行比较，如果相等就跳到 9ADC02，然后就返回了。如果不相等，就将 DL 的内容存储到我们编辑的内存中。我们已经知道了 DL 等于 0，因为我们看到内存中的值从 01 变成了 00。所以这基本上就是另一个注册检测点，并且如果它检测失败就会将 已注册/未注册 标志置 0。如果成功，就什么都不做。现在咱们将硬件断点删掉，选择 “Debug” -> “Hardware breakpoints” （译者注：这里的 Debug 指的是菜单中的），将硬件断点删除。咱们在 9ADBF4 处设置另一个硬件断点，这样我们可以在该段代码运行前断下来：



你或许会纳闷，我为什么不在这里设置一个普通断点。因为我先试过了！Olly 根本就不会断下来好嘛！有几个愿意可能会导致该问题的发生：这段是多态代码，所以我们的 BP 丢了，程序检测到软件断点所以把它删了，断点在一个 Olly 不会自动追踪的区块....。不管怎样，就是这么个结果。我们需要设置硬件断点而不是软件断点。不保证硬件断点一定管用，因为软件有可能专门对它们进行检测。不过它是一个更可靠的设置断点的方法，所以通常来说还是比较好用的。

****在后面的章节中我们会更多的学习反调试技巧。****

重启应用，我们会再次断在新的硬件断点处:

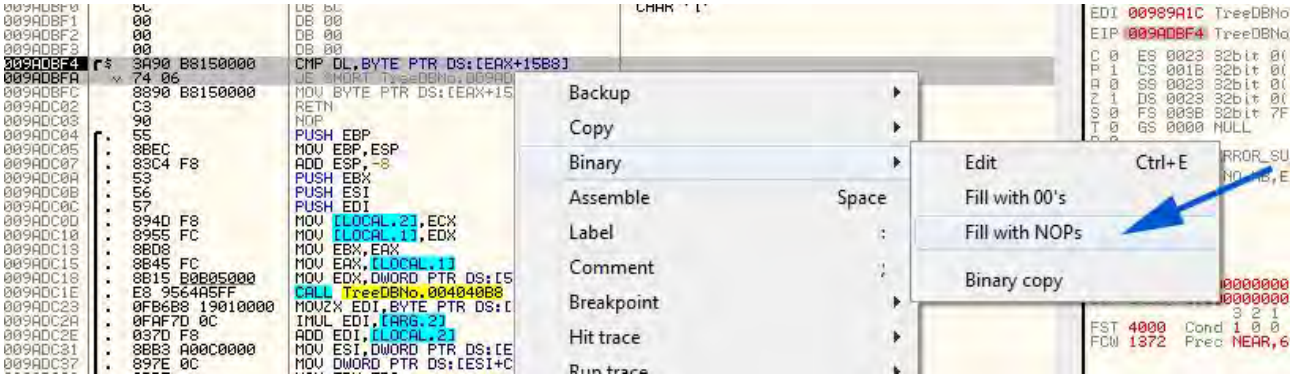
好，现在咱们来思考思考啊。这个子程序是在咱们原来的断点前面被调用。这个子程序检测我们是否注册，如果没有就将 [EAX+15B8] 地址处的内容设置为 0，如果注册了就置为 01（或者任何非 0 的数据）。然后我们原来的子程序被调用，就是那个在窗口标题中输出 “Registered” 或 “Unregistered” 的子程序，它也是根据内存中的数据是 0 还是 1 来决定输出。如果我们确保任何时候只要该子程序运行时那个内存位置中都是 1，那么任何其他子程序来检测内存中内容时看到的都只能是 1，也是就认为我们已经注册了。

如果我们只是将子程序修改成总是在内存中的合适位置放置 1 的话会怎么样？咱们来试试看。

下一个问题就是怎么做最简单。好，我们已经有了在 9ADBFC 处被用某些值（DL）填充的内存位置，所以我们只需要在上面的某个地方将 DL 改成 1。问题是将 DL 改成 1 需要在当前指令的长度上加一个字节，这样做会覆盖 RETN

语句。如果我们将比较/跳转指令替换成将 DL 置为 01 的指令怎么样。那样的话，在最后一行，DL 将被拷贝到我们的内存位置！下面就是我们的做法，选中比较/跳转那两行指令：

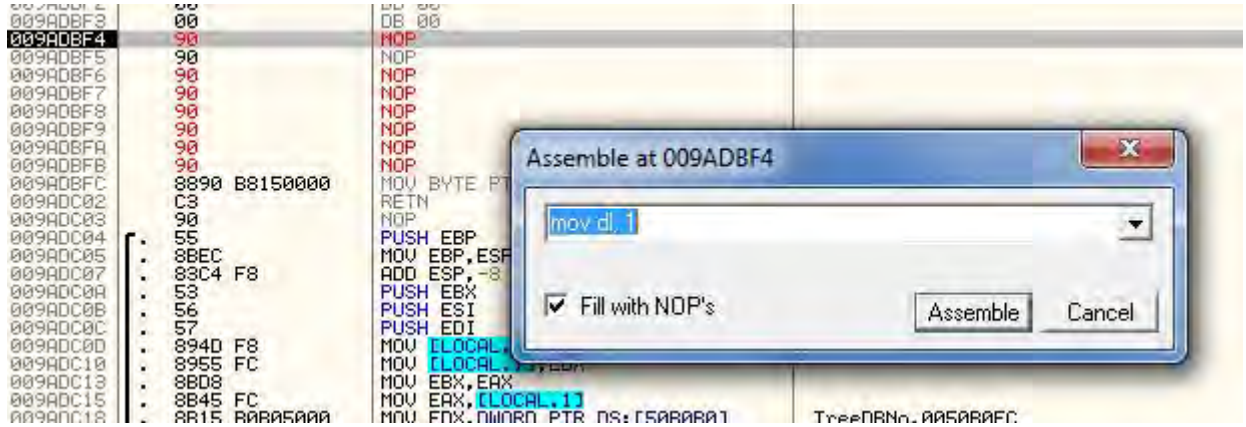
右键选择“Binary” -> “Fill with NOPs”：



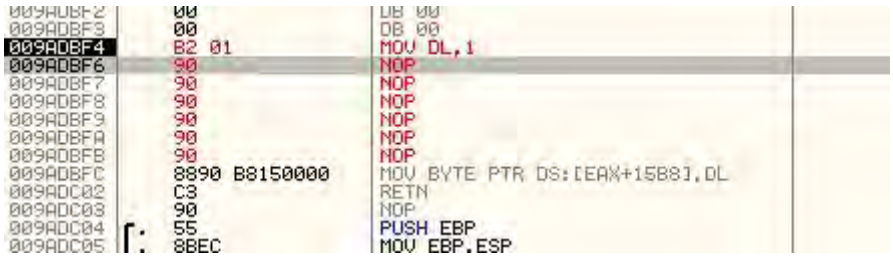
然后就像下面这样：

这一步不是必须的，不过这能让你更容易的看清自己正在干啥（译者注：这样可以防止不小心多添加或少添加字节）。

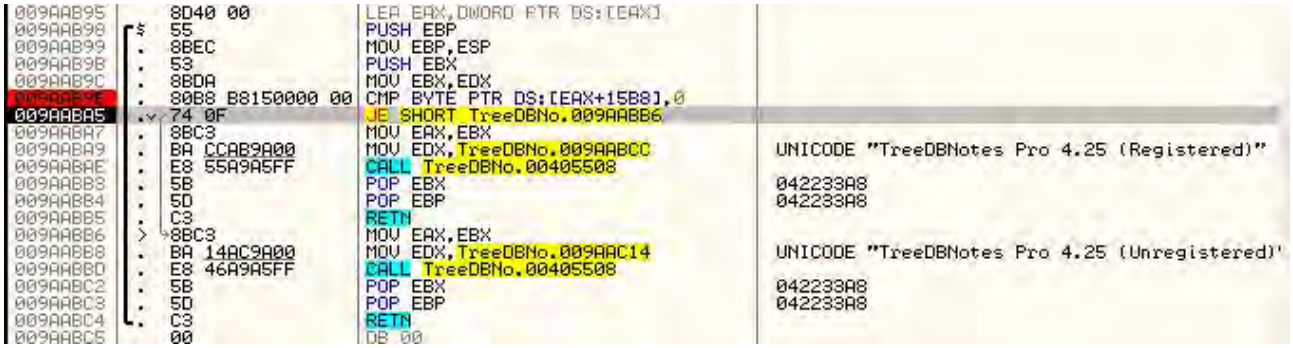
现在选中 9ADBF4 处的 NOP，按一下空格键。弹出汇编窗口，输入 MOV DL, 1:



先点 Assemble，然后点 Cancel。结果就像下面这样：

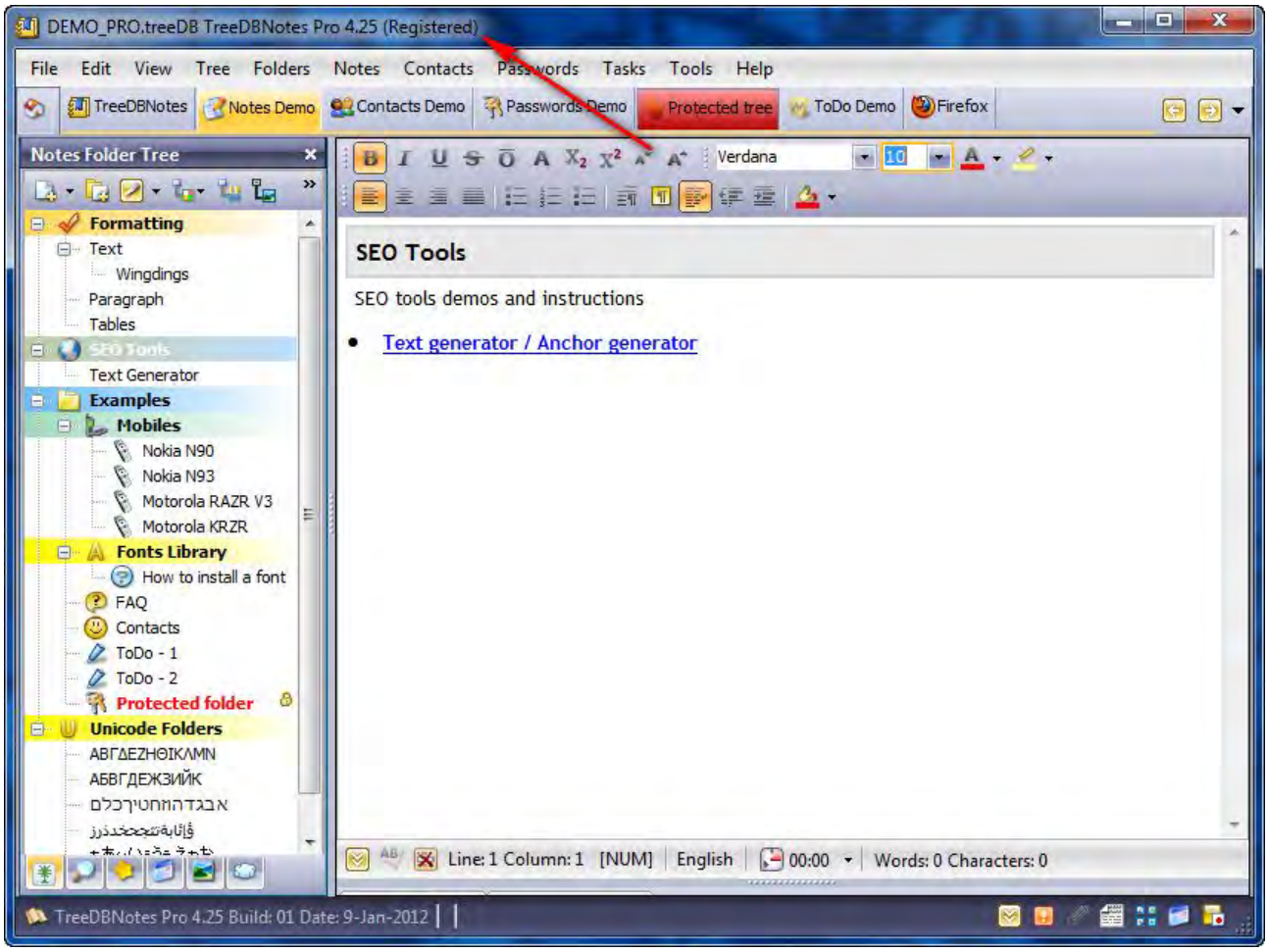


现在，无论什么时候调用这个子程序，内存中的那个标志位（译者注：这里的这个标志位和寄存器的标志位不一样啊，读者要能够分辨的了）都会被置 1 而不是 0。因为我们仍然暂停在子程序的第一行，所以你可以单步执行以观察 DL 被置 1 的过程，然后 1 就被存入内存中（你可能需要到数据窗口的合适的地址观察，因为 Olly 很可能再一次将数据窗口重置过）。现在运行程序，Olly 会断在我们原来的断点：



我们能看到我们将直达正确的字符串。继续运行，我们将断在我们修改过的注册检测子程序，它会如愿的放一个 01 到我们的内存地址中。这会回头再往前这样来回几次直到最后：

现在我们注册成功了!!! 继续运行程序（打开一个 demo 文件），Olly 会在注册子程序中断下来几次，不过每次它都会走正确的路。不久你就会看到主窗口：



你会看到我们仍然是已注册状态。点击显示关于对话框：

恭喜你！你已经成功的完成了第一次破解。

别忘了将它保存到磁盘。打开硬件断点窗口（“Debug” -> “Hardware breakpoints”），点断点边上的 Follow 按钮。然后我们就来到了我们打补丁的地方。选中所有我们修改过的行，右键选择 “Copy to executable”。在弹出的窗口中右键，选择 “Save to disk”。以原来的名字保存它。现在退出 Olly，运行程序体验它的全部，以及注册成功的骄傲与自豪!!!

第十四章：NAG 窗口（我不是在说你妈）[注①]

一、简介

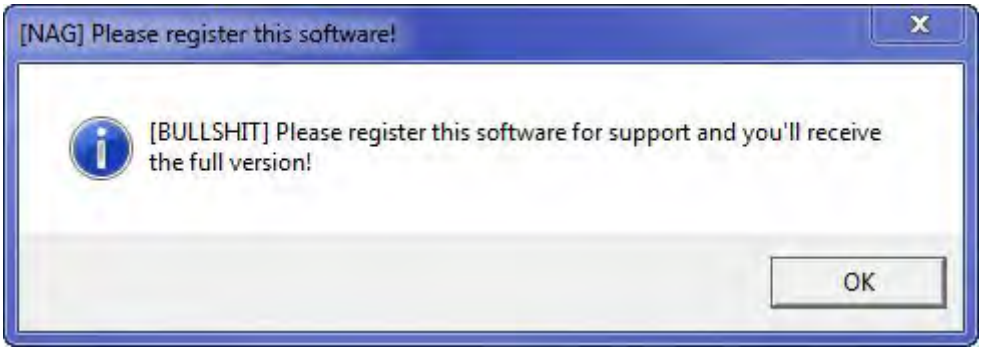
Nags，或者叫 Nag 窗口，是普通的消息对话框。它弹出来是提醒你你的试用结束了、你需要注册、关于访问网站的提醒....。基本上任何事它都要唠叨，而且还是不必要的（像大多数的 boss 一样）。许多免费软件之所以免费，是因为它们充满了 nag（广告、限时试用、重定向）。商业软件通常也有这些玩意儿，提醒你“你只剩下 18 天来使用此产品”等等。在逆向工程领域，除掉 nag 窗口是一个中心主题，有时候也提出了很多挑战。本章我们将会研究两个有 nag 的程序。我们将会绕过它们，之后它们就再也不会显示了，然后再打上补丁，这样它们就永远不会回来了。

我也会介绍一个新的 Olly 插件，叫 IDAFicator。它有许多特点和设置。你可以在工具页下载该插件。因为它有如此多的特性，我也在下载中包含了 IDAFicator 作者写的教程。我强烈推荐你看看教程，因为该插件有许多非常酷的特性。

你可以在教程页下载相关文件以及本文的 PDF 版。

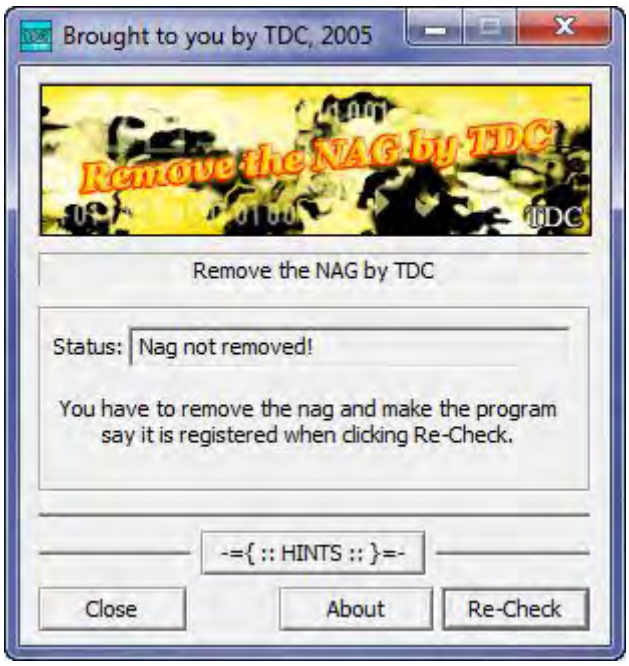
二、第一个应用程序

我们将要研究的第一个二进制文件是 Nag1. exe。程序一运行就会弹出 nag:

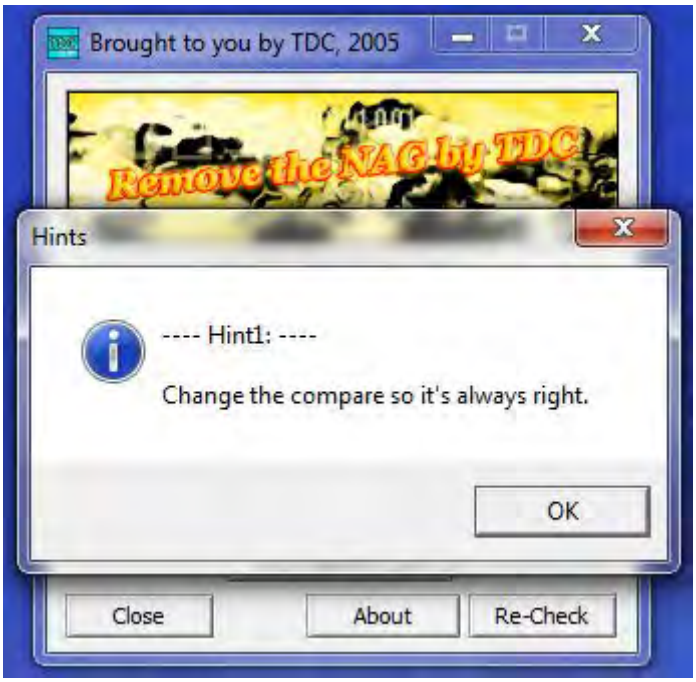


它明摆着告诉你这就是一个 cracker 写的。不管怎么样，点了 OK 就可以看到主窗口：

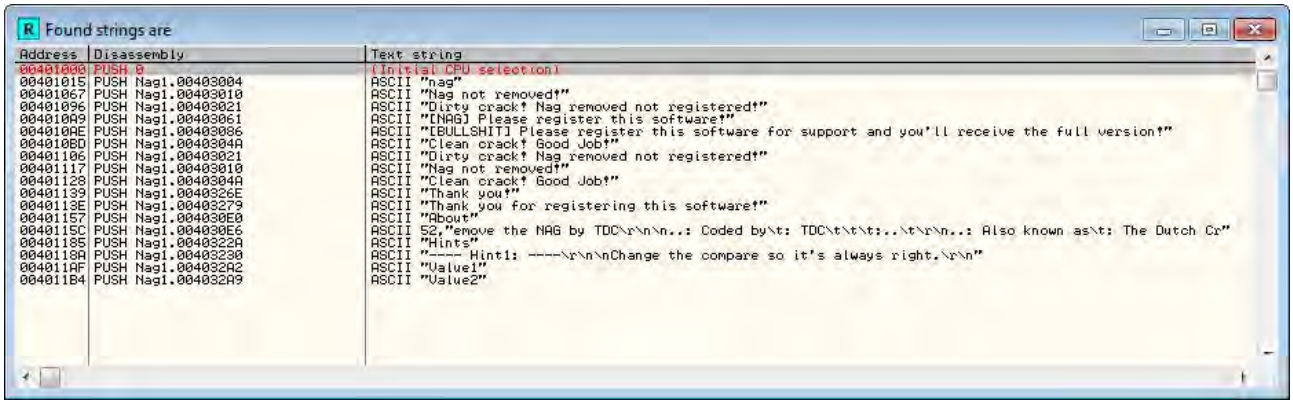
注①：标题中的“我不是在说你妈”可不是骂人的，因为 nag 有“唠叨”的意思，而英文的标题就是“Nags”，没有其他多余的字，所以你懂的。



注意，它说“Nag not removed!”。我情不自禁的就点了那个“Hints”按钮，然后给了一些非常详细的信息（译者注：我咱们没觉得很详细）:

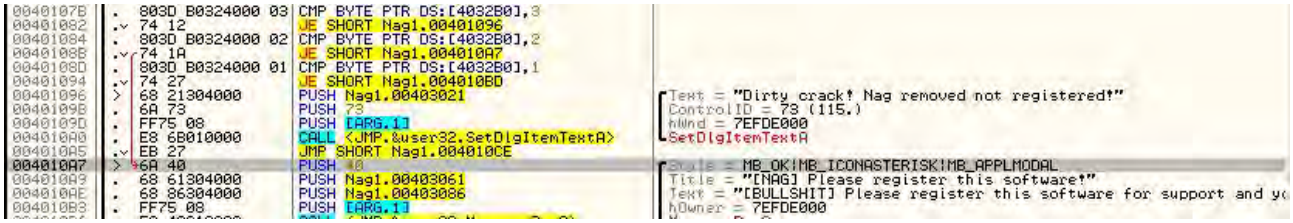


Gee，谢谢。Olly 载入应用，咱们试试老方法——搜索字符串:

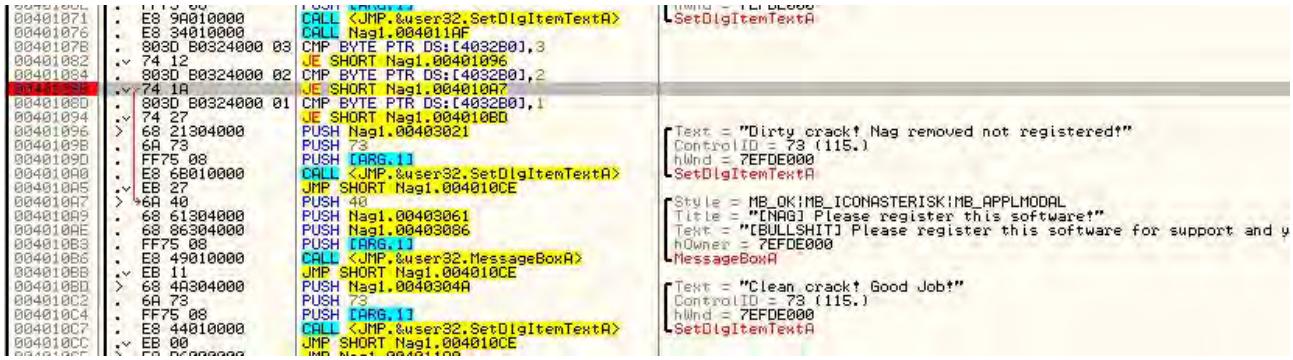


运气还不错。你可以在 4010AE 处看到 nag 窗口中的文本。双击它，咱们就跳到了 nag 窗口被创建的地方：

嗯，它上面有一个有趣的字符串，不过咱们现在先不管。咱们看看 4010A7 处，也就是调用 MessageBoxA 函数的第一行，看看哪里调用了它：



我们能看到 40108B 处的 JE 指令调用了它，而且刚好在一个比较指令的后面。好吧，这个场景我们已经很属性了😀。咱们在 JE 指令那设置一个 BP：

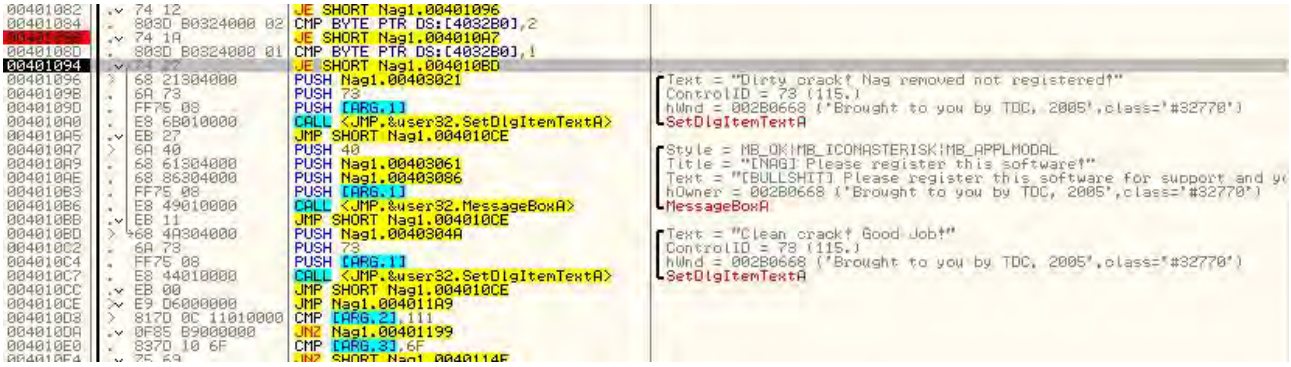


运行程序。然后我们断在了那个 BP，能够看到我们将跳转到 nag 窗口那，所以不能让它跳：

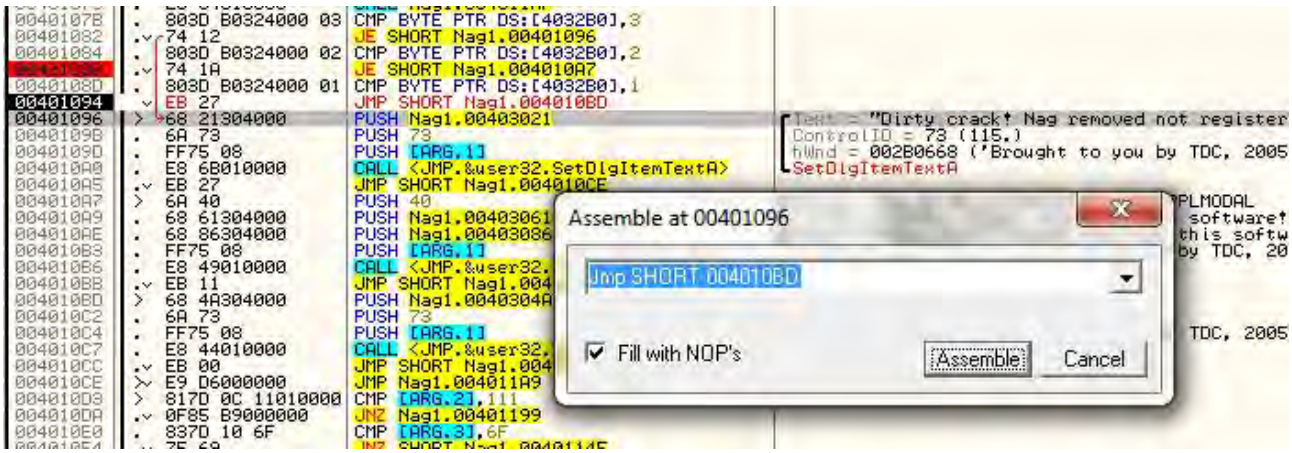
接着运行程序：

这就是“Dirty crack!”的出处，显然咱们的补丁打的还不够。重启应用，Olly 断在了 BP 那。再次将 0 标志位清零：

咱们单步执行两次到下一个跳转那。你可能已经猜出来了，这个跳转应该是跳到好消息那的，而不是到坏消息那：

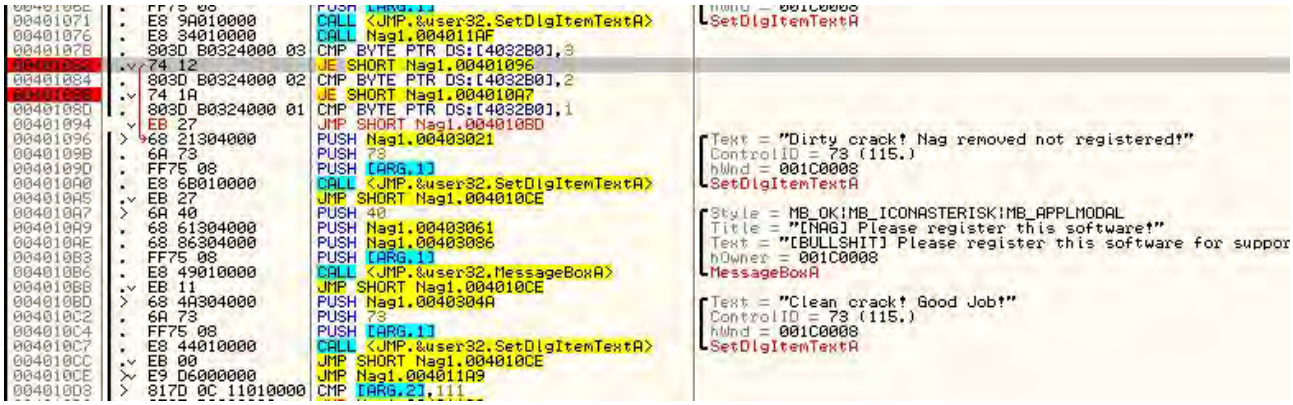


咱们打个补丁，让它直接跳：



运行程序，可以看到咱们弄对了：

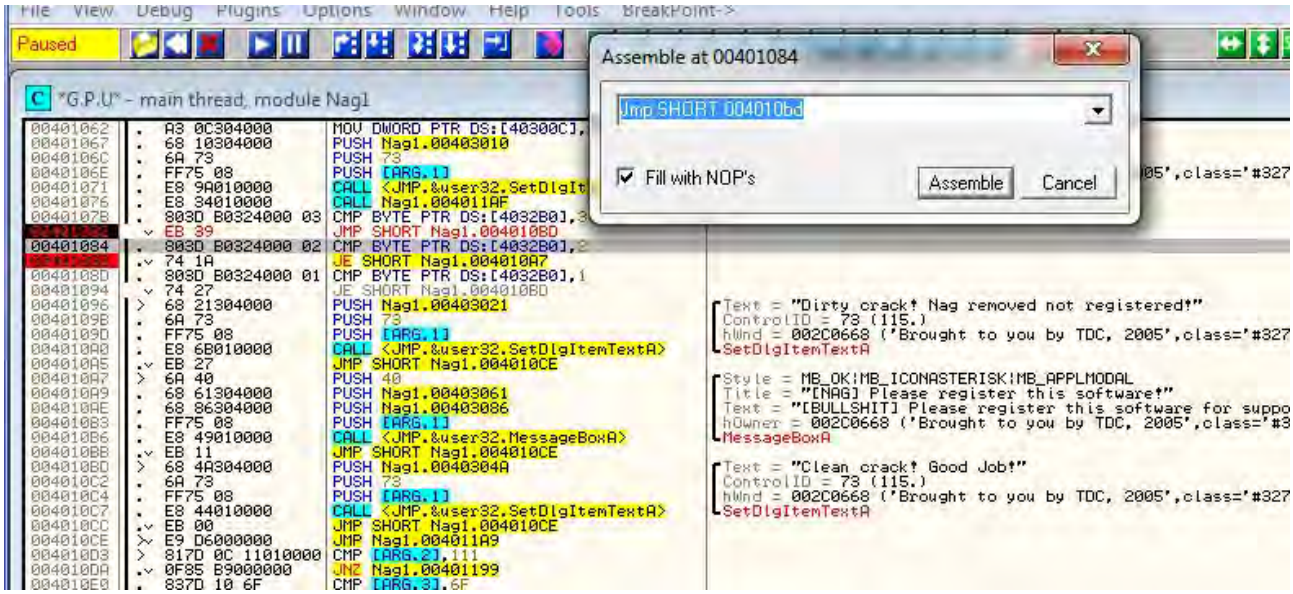
很明显，现在这个补丁就可以解决这个程序，咱们回到 40108B 那（咱们原来将 0 标志位清零的地方）给它打上补丁让它永远不会跳转。保存这两个补丁程序就会很好的运行。不过我也想向你展示（我以前提到过，如果我没有提到的话，那我应该提到的），通常总是有别的方法来给程序打补丁。重启应用，在我们的 BP 处断下来：



这些指令与下面这些（用高级语言表示）类似：

```
if (contents of 4032B0 == 3)
      jump  “Dirty Crack”
else if( contents of 4032B0 == 2)
      jump to  “Show Nag Screen
else if (contents of 4032B0 == 1)
      jump to Good Boy Msg
else
      Display  “Dirty Crack”
```

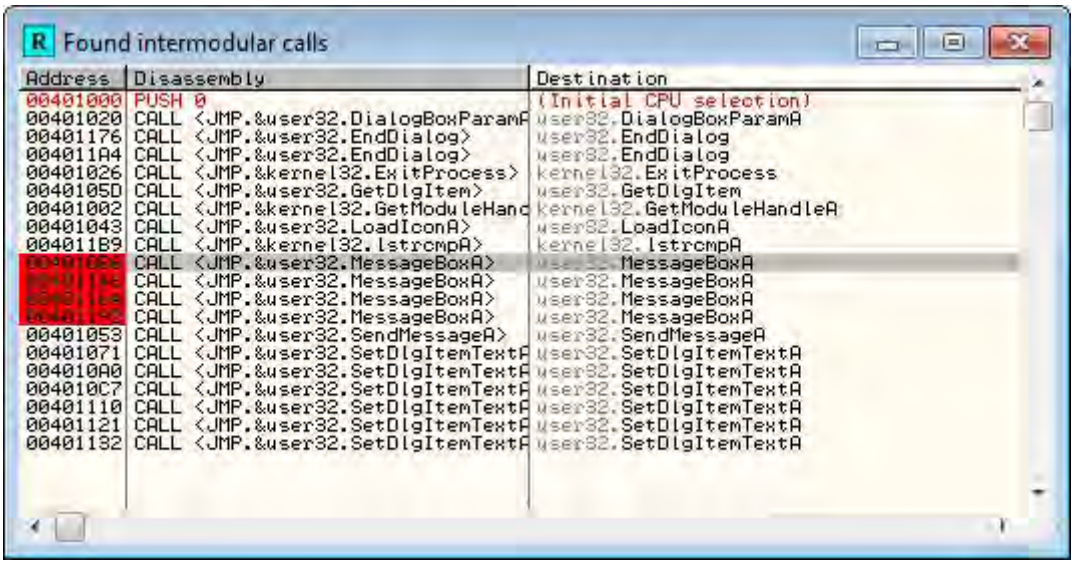
我们知道 nag 窗口默认是要显示的，4032B0 处内存总是等于 2，因为跳转得实现才行。如果我们跳过整个 if/then 语句，直接跳到好消息怎么样？所以如果我们将最开始的第一个跳转替换成跳到好消息的跳转，那么我们就只需要一个补丁就行。试试看：



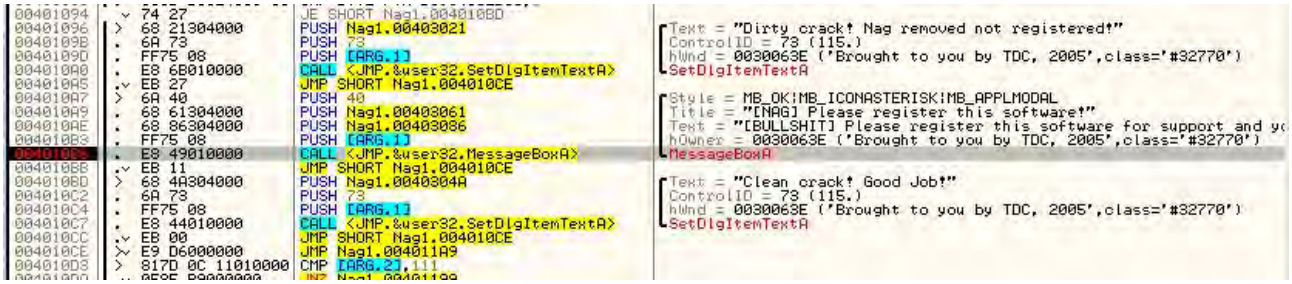
运行下程序：

可以看到结果是一样的。另外，可以思想下更加优雅的方法，“4032B0 中的内容总是等于 2，不过要显示好消息的话它就需要等于 1，那么为什么不在内存中就放一个 1 呢，那样的话就会一直显示好消息了呀？”你应该试试这个。重启应用，点一下数据窗口，转到 4032B0，用二进制编辑将它改成 1。起作用了没？

需要记住的另一件事是，总是有别的方法可以找到我们正在寻找的代码块。比如，如果本例中我们不能用字符串，我们就可以用 搜索模块间调用 （译者注：相关内容在第九章）:



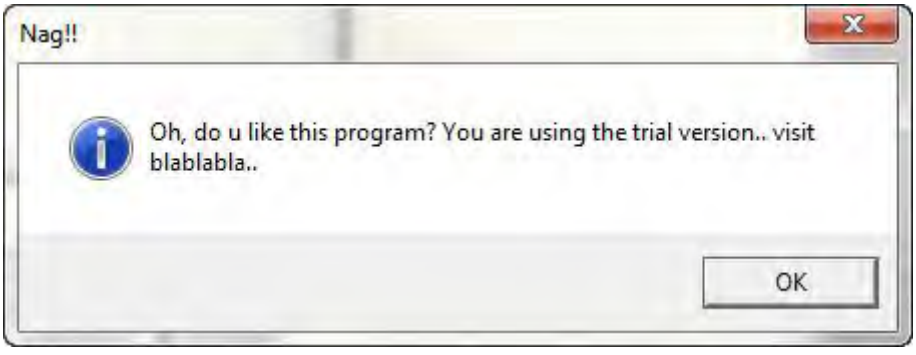
注意有四个对 MessageBoxA 的调用。右键其中一个，选择 “Place a breakpoint on every call to MessageBoxA”。当你运行程序时，在显示任何东西之前，我们会停在下面这行代码:



是不是很熟悉？它就是 nag 消息框!! 所以要记住总是有不只一种方法可以完成一些事情。不久我们将会学习其他的技术（像窗口消息处理），会给你更大包的技巧。

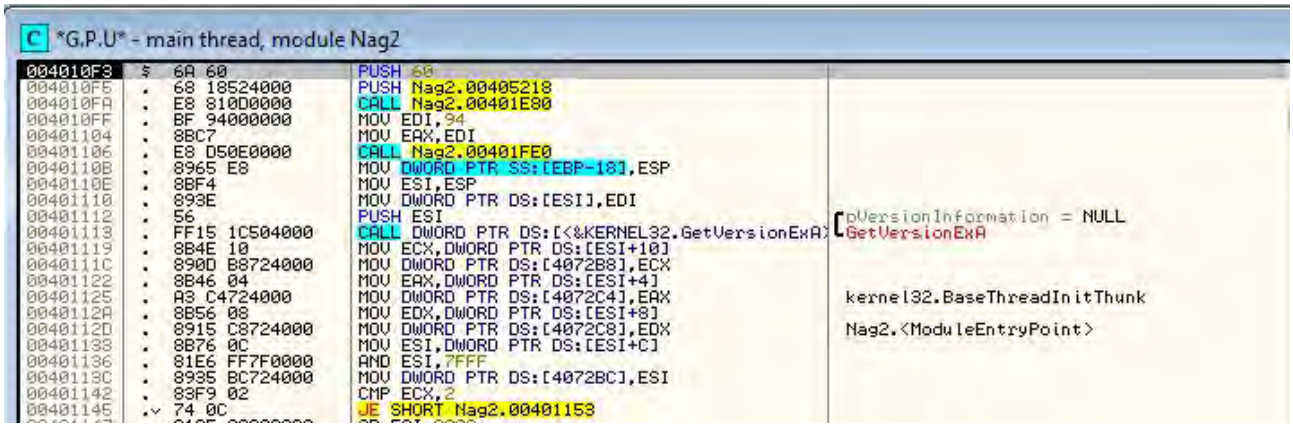
三、第二个应用程序

现在咱们来看看 Nag2. exe。看起来差不多，不过我们将用不同的方法来解决它。启动程序的时候，我们看到了意料中的 nag:



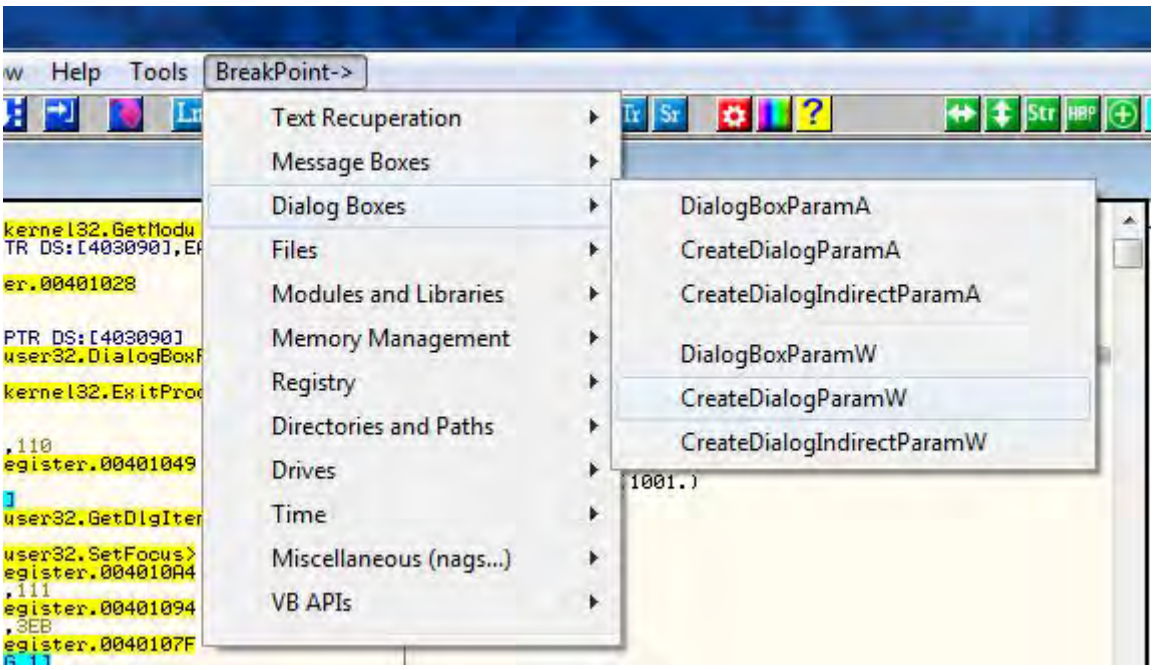
在点了 OK 后，我们看到了主窗口:

此时我关了程序，将其载入到 Olly 中：



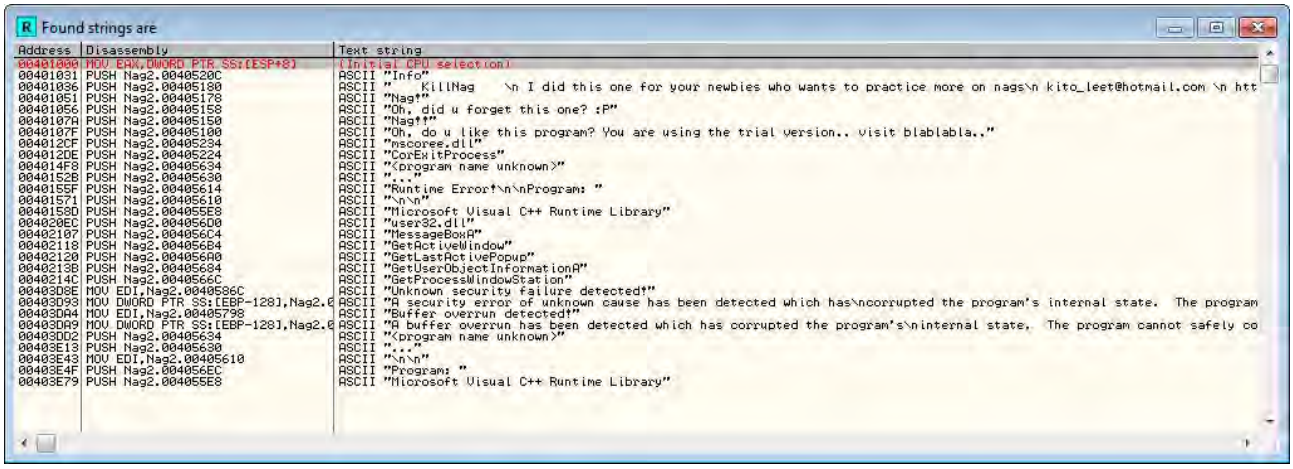
首先，咱们来看看有没有字符串。这里我想提的一件事是 IDAFicator 插件。在众多的添加功能中，它在程序顶部提供了一组按钮，让搜索字符串变得更加的简单。当点击字符串按钮（Str）时，它会显示 ASCII 和 Unicode 两种字符串，并自动的将光标移到顶部，这样你就不用自己滚动到顶部了。下面是那些按钮的样子：

第一个按钮（有左右箭头的那个）是向前和往后。比如，你点击一个 CALL，然后按一下 enter 键转到该 CALL，点一下第一个图标你会回到 CALL 指令那。右击你就会前进一步。第二个按钮会尝试找出当前函数的开头，右击则会尝试找到结尾。下一个就是字符串按钮。再下一个是硬件断点按钮。它会弹出一个很漂亮的对话框来显示你设置的所有硬件断点。非常的便利。十字图标打开你的应用程序所在的文件夹，列表图标会弹出一个对话框以便于你输入多行汇编代码，如果你打算修改 exe 文件的大部分代码时可以用这个。你会注意到一个叫“Breakpoints”的新的菜单项，它会弹出一个下拉菜单，里面是许多用到的 API，你可以自动对它们设置断点：

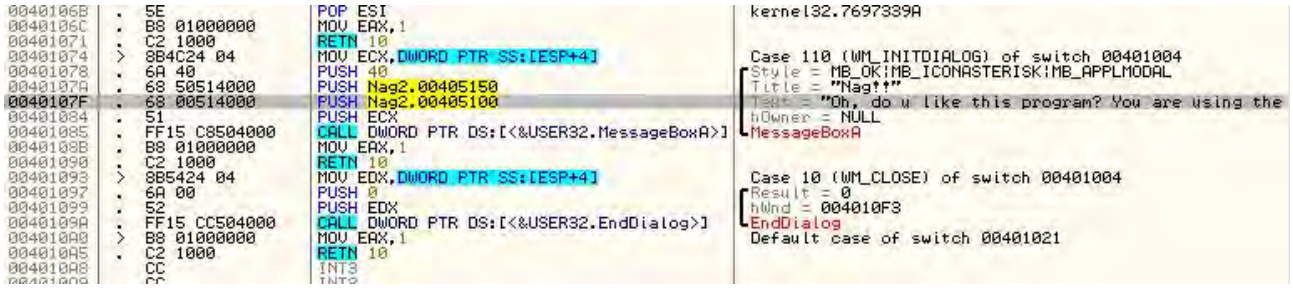


最后，有一个上下文菜单可以让你恢复隐藏的菜单，我们会在后面的章节中讨论。

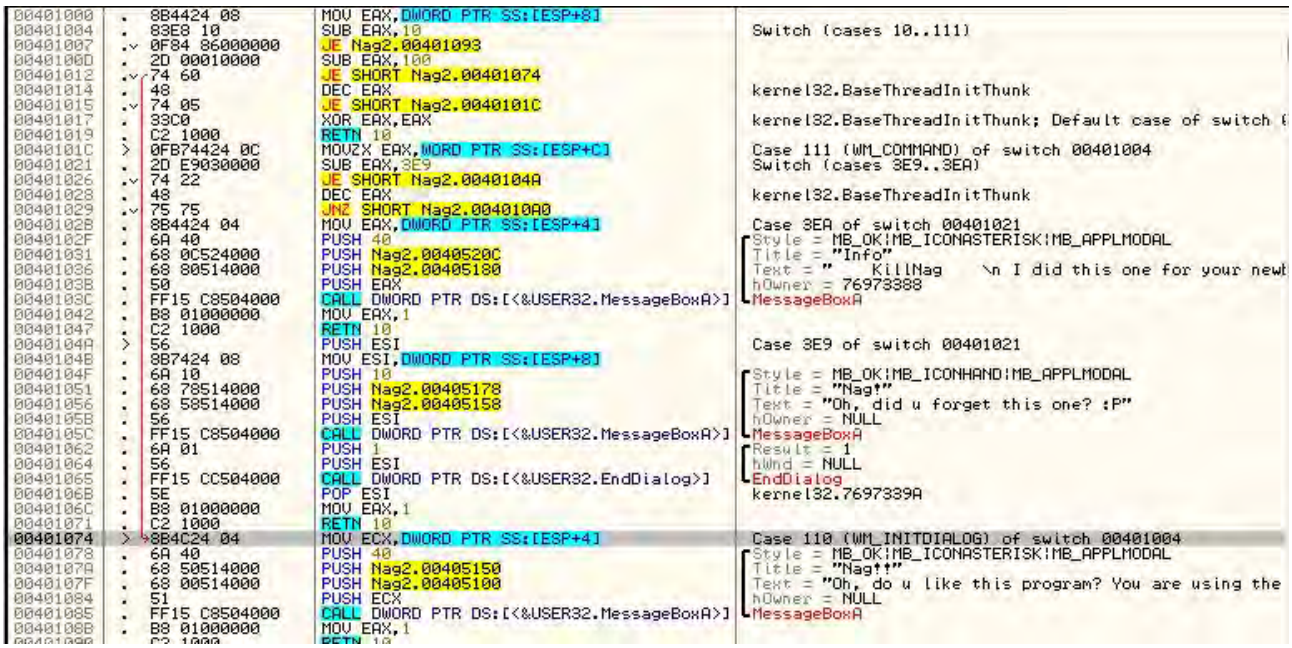
咱们继续，点击新的按钮栏中的字符串（“Str”）按钮：



在第七行，我们看到了 nag 上面的文本，双击该行：



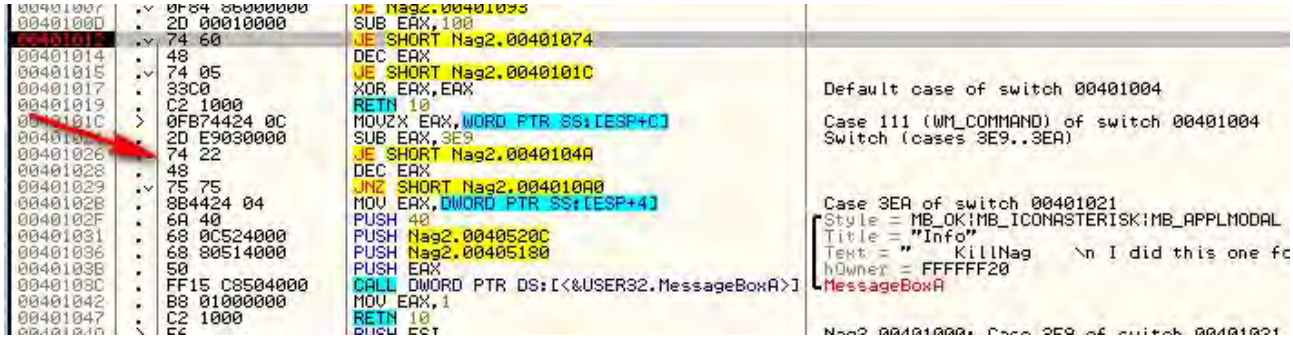
看到了 nag 的实现方法。是一个自包含的方法（在它的上面和下面各有一个 RETN 指令），所以我们知道它是在某个地方被调用的。点击 401074 那行，也就是该自包含方法的第一行，看看它是从哪被调用：



可以看到是被 401012 的 JE 指令调用。咱们在这里设置一个断点，运行程序:

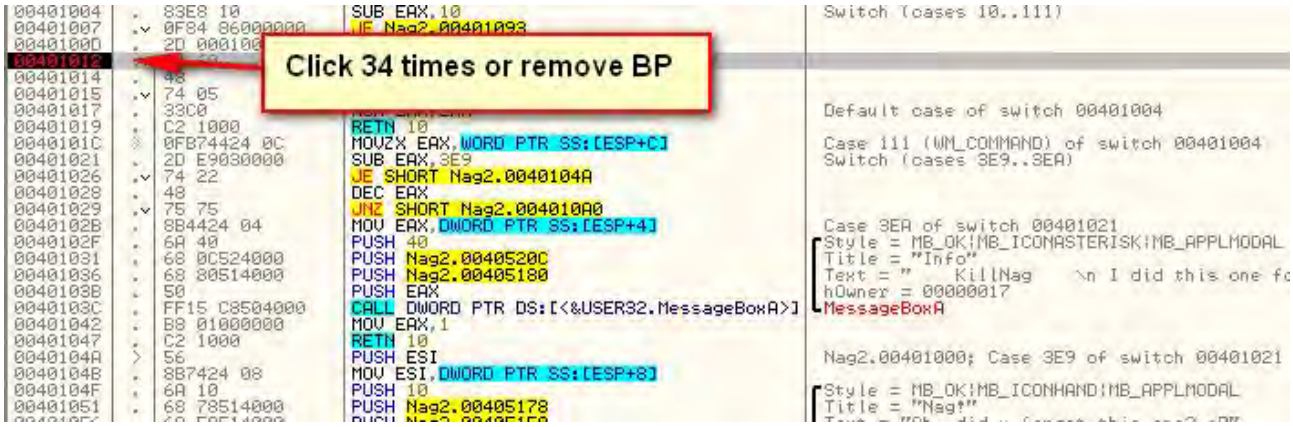
咱们断在了 JE 指令处。注意它没有调用我们的 nag 窗口代码。原来是我们刚好在 windows 的处理的中间。我们会在另一章深入讨论消息处理，不过目前我们只需要知道所有的 GUI Windows 程序都有一个消息处理程序，并且 Windows 通过它来发送各种消息。根据到达的消息，我们可以添加自己的代码来覆盖 Windows 的普通处理流程。例如，当我们点击窗口上的“X”时，Windows 就会通过消息处理程序发送一个消息说“嘿，用户想要关闭这个窗口”。我们可以让消息通过，这样的话Windows 就会处理它并关闭窗口，或者我们可以“捕获”这个消息，做任何我们想做的（可能弹出一个对话框说“你还未保存，确定退出？”）。

我们的断点刚好在这个的中间，所以已经来到的第一个消息与应用程序想要覆盖的以便于显示 nag 窗口的消息不匹配:



咱们继续，按 F9 运行程序，然后断在同一个 BP，不过这一次跳转会实现，将显示 nag 窗口。咱们让 Olly 别显示它：

现在，如果我们留着这个断点，通过这个消息处理程序将会发送超过 34 个消息。你可以留着这个 BP 然后运行 34 次（这种情况下，在某一刻你会看到有窗口出现，按钮被绘制等），你也可以删除断点然后就运行一次。这样的话，对 nag 窗口的调用就不会再次执行，所以删除断点再运行程序比较好：

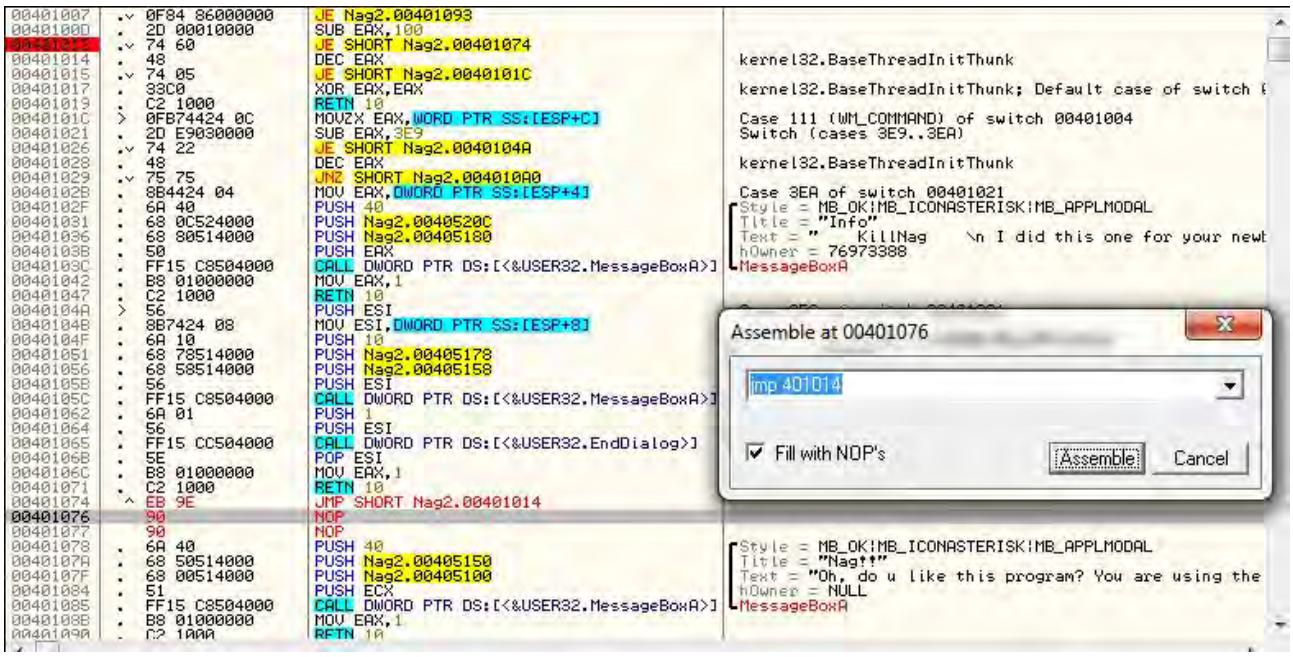


然后就看到了主窗口：

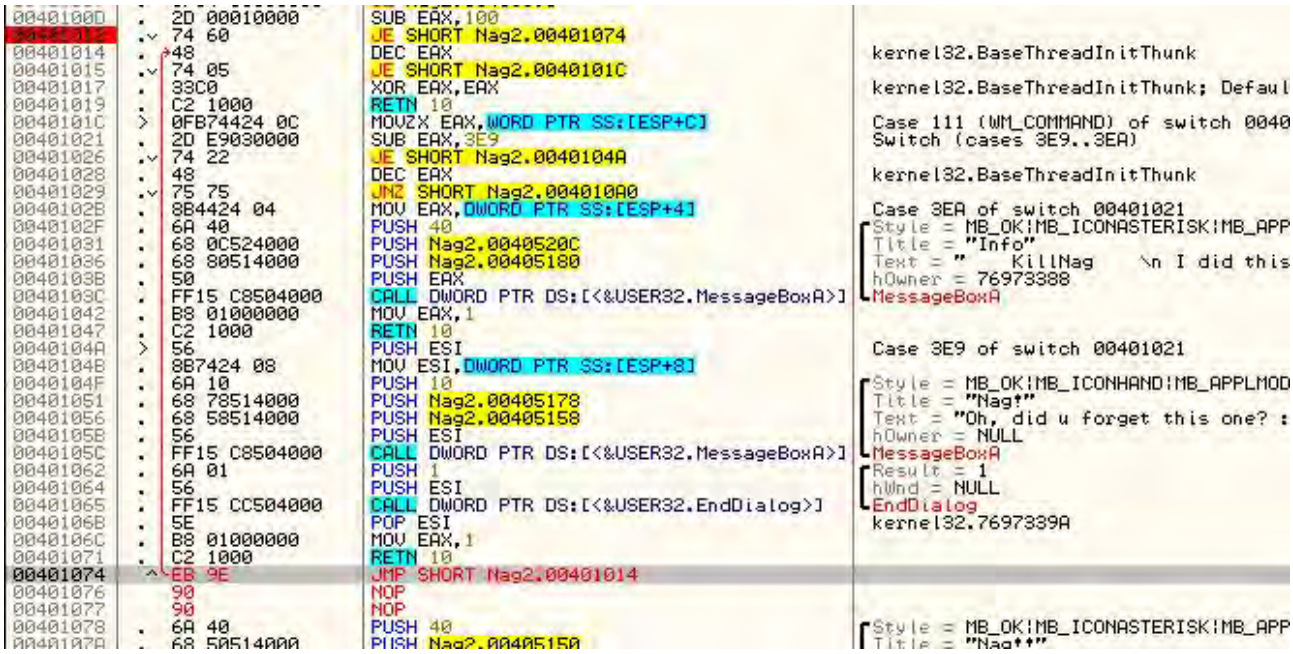
注意初始的 nag 窗口已经消失了。

四、给程序打补丁

通常我们所要做的是给跳转到 nag 的 JE 指令打补丁，将其 NOP 掉。这样的话，它就是再也不会跳了，不过我想告诉你另一个种方法也可以实现。我们知道当正确的消息来到消息处理程序时（这里指的是第二个消息），我们的 nag 代码将会被调用。好吧，如果我们允许跳到 nag 那，但是将 nag 改成再跳回去会怎么样？



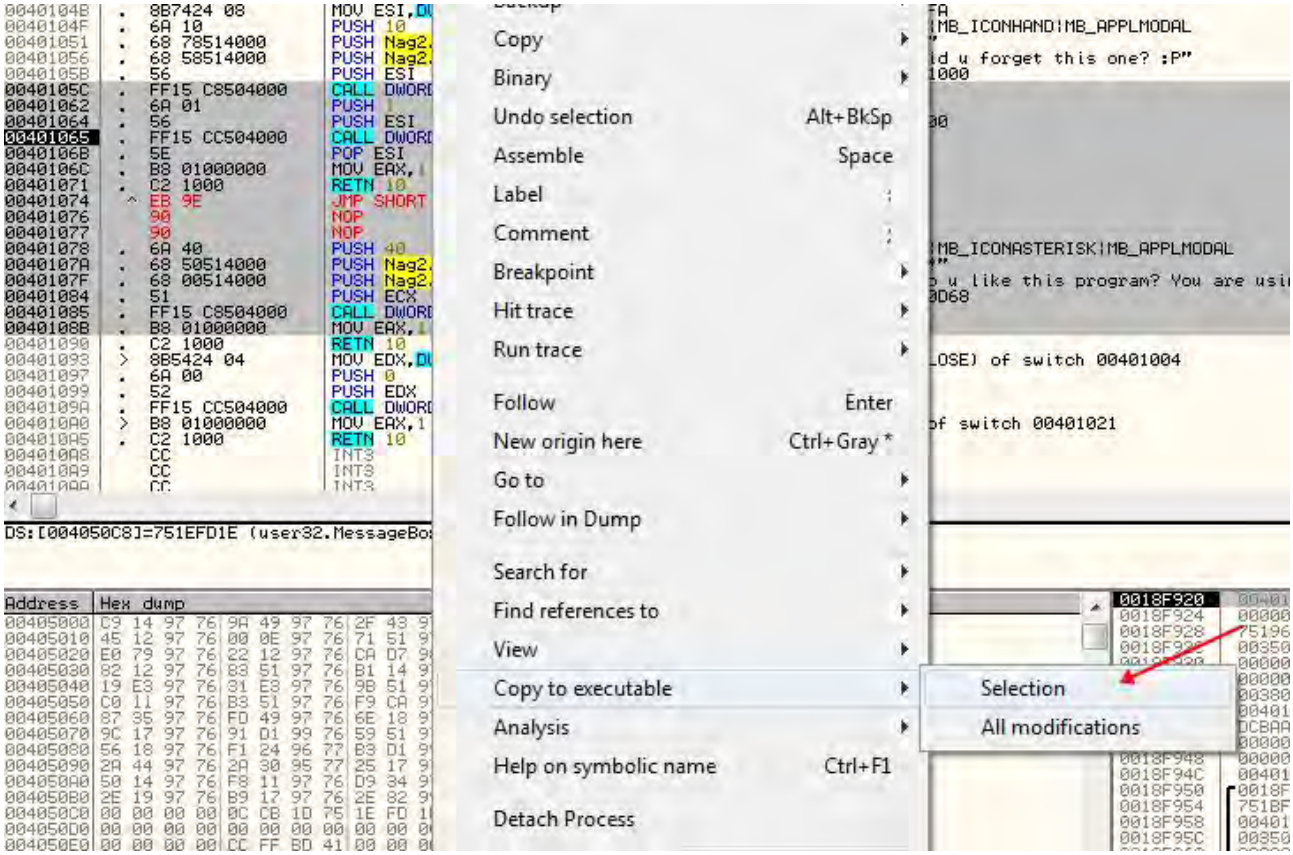
这里，将会跳转到 401074 的 nag 指令，但是我们让它又立即跳回了初始跳转的后面那行（401014）。基本上，我们的程序会跳转，然后又跳回到下一行：



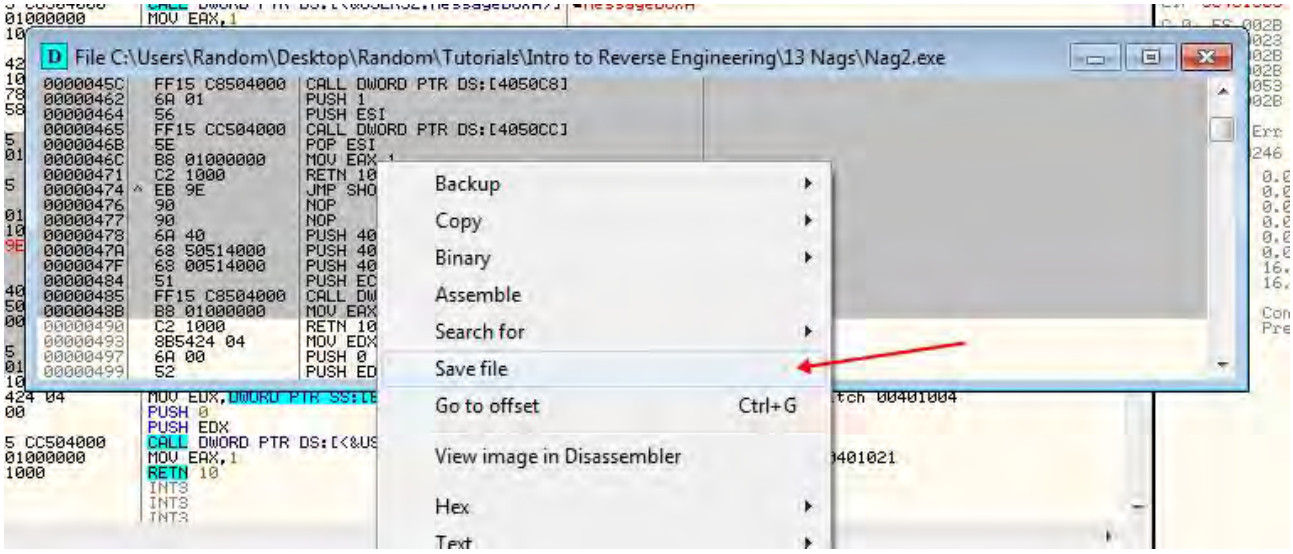
将 401012 处的 JE 指令 NOP 掉与添加一个跳回到 401074 的跳转真的没啥不同，不过我想让你开始注意到总是有多个打补丁的方法，有时候 NOP 掉一个 CALL 并不是最好的方法。记住，这个二进制文件是你的，你可以添加任何你想添加的代码，所以别害怕修改它，尤其是在学习的时候。

运行程序，可以看到 nag 窗口同样被绕过了：

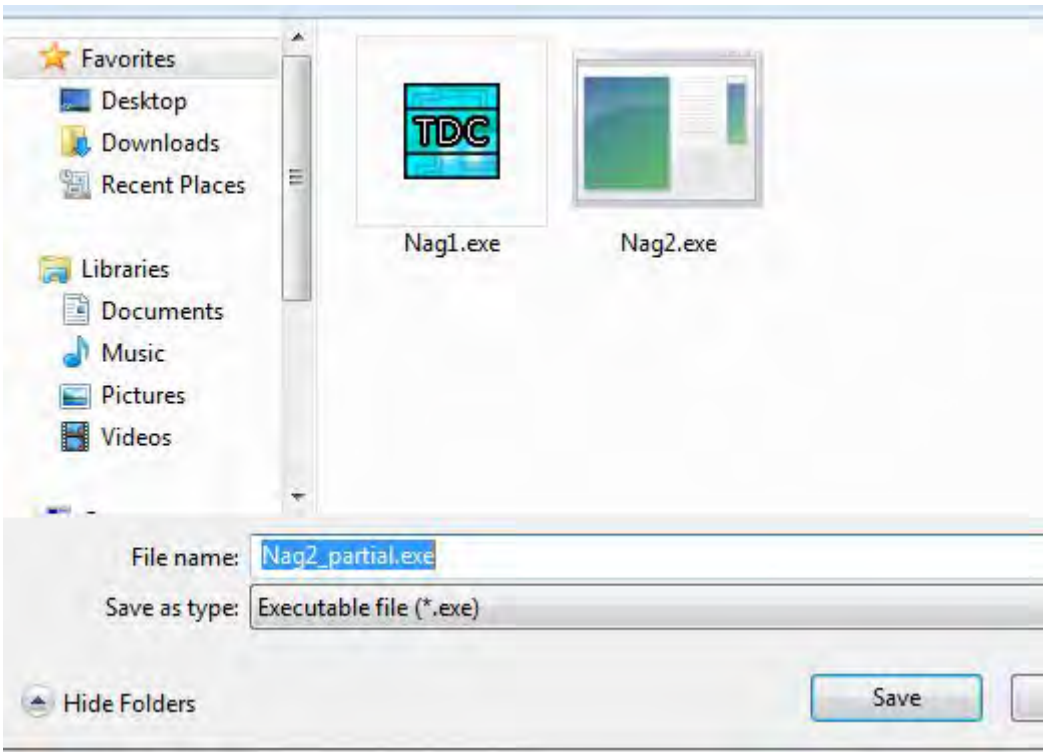
现在保存补丁。选中修改的代码（如果你选多了也没事。译者注：就是说你选中了那些没有修改的也没关系。），选择“Copy to executable” -> “Selection”：



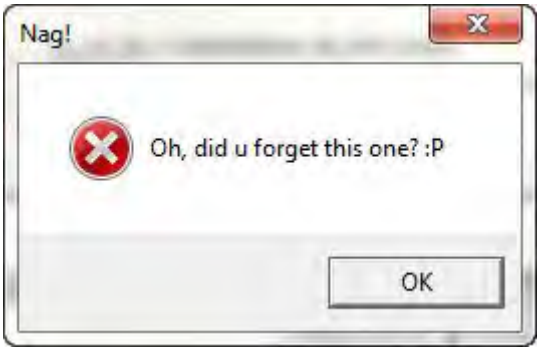
在新窗口上右键，选择“Save file”：



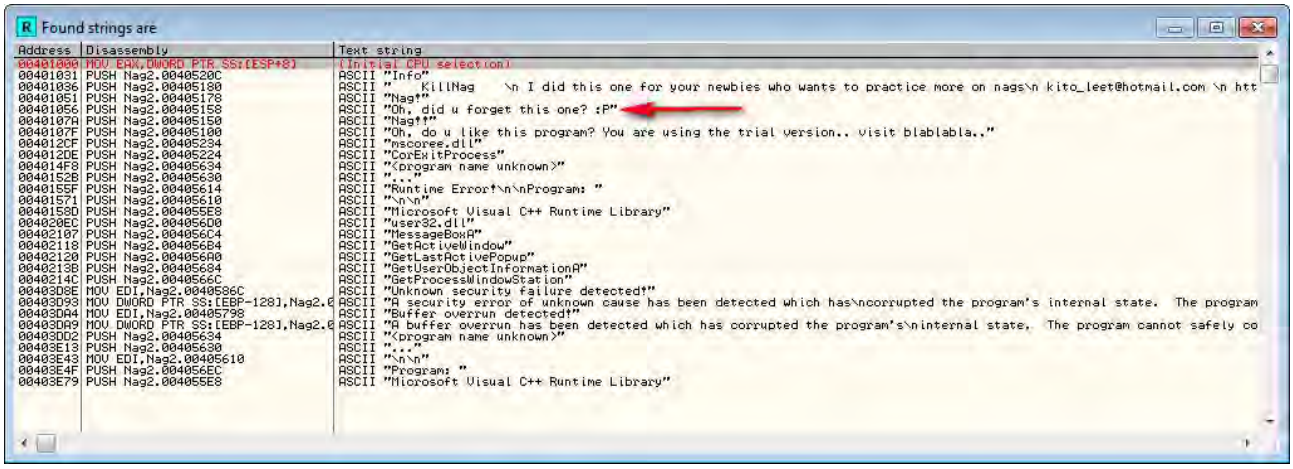
我将它保存为不同名字的文件，这里是 Nag2_partial. exe。等会你会明白我为什么把它取名为 partial:



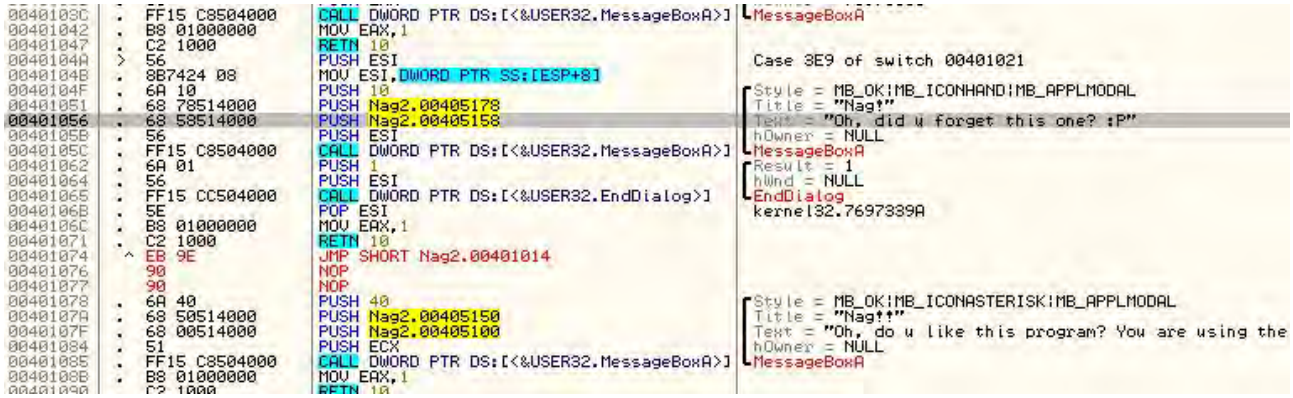
OK。咱们继续，Olly 载入这个打过补丁的程序试试看。我们直接跳到了主窗口，所以我们知道那个补丁起作用了。现在点击 exit，但是弹出了这个:



噢。我才这里作者是真的很确定（译者注：确定你没有干掉这个 nag）。咱们来找找这第二个 nag。回到字符串窗口那，可以发现这个 nag 的文本也在那：

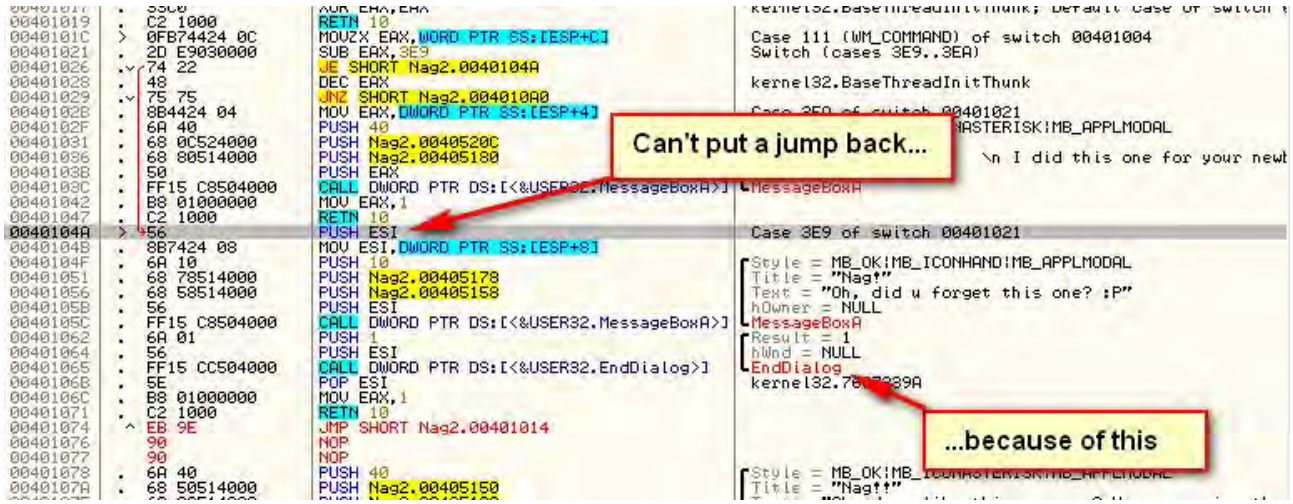


许多许多许多应用程序会这么干。它们启动的时候有个 nag，关闭了之后又来一个。大部分时候，你在搜索第一个 nag 的文本字符串时，你也应该自动找找其他的。在该文本上双击：

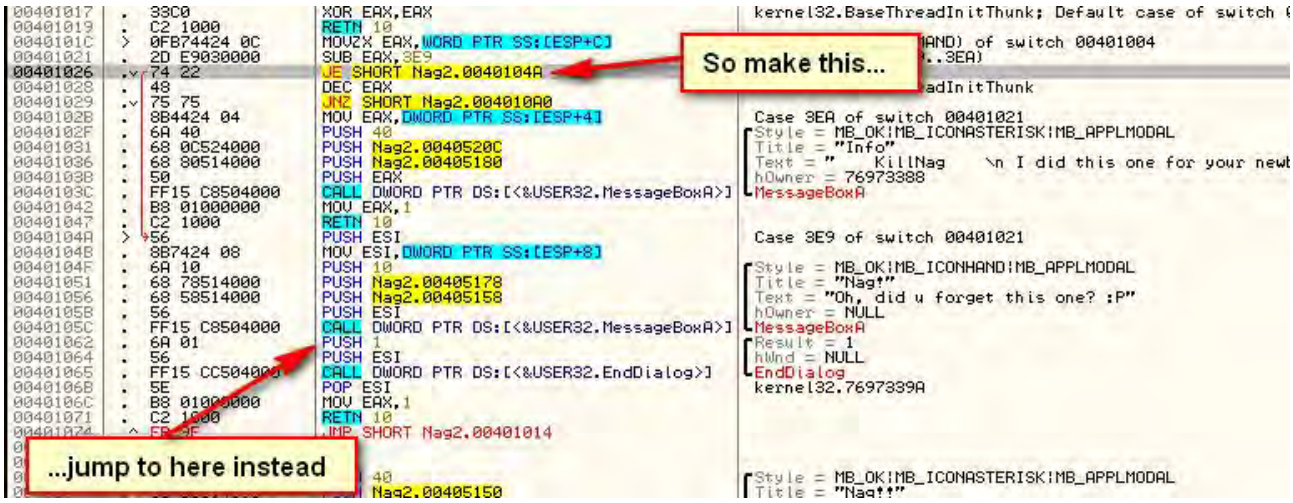


我们在这里看到了该 nag 的方法。点击该方法的第一行，我们能够看到第二个 nag 正是在第一个 nag 被调用之后调用的。但是它用了一个不同的消息来触发它（很可能是窗口销毁消息）。所以当这个消息到达时，发出用户选择 “Exit” 的信号，第二个 nag 就会被调用。

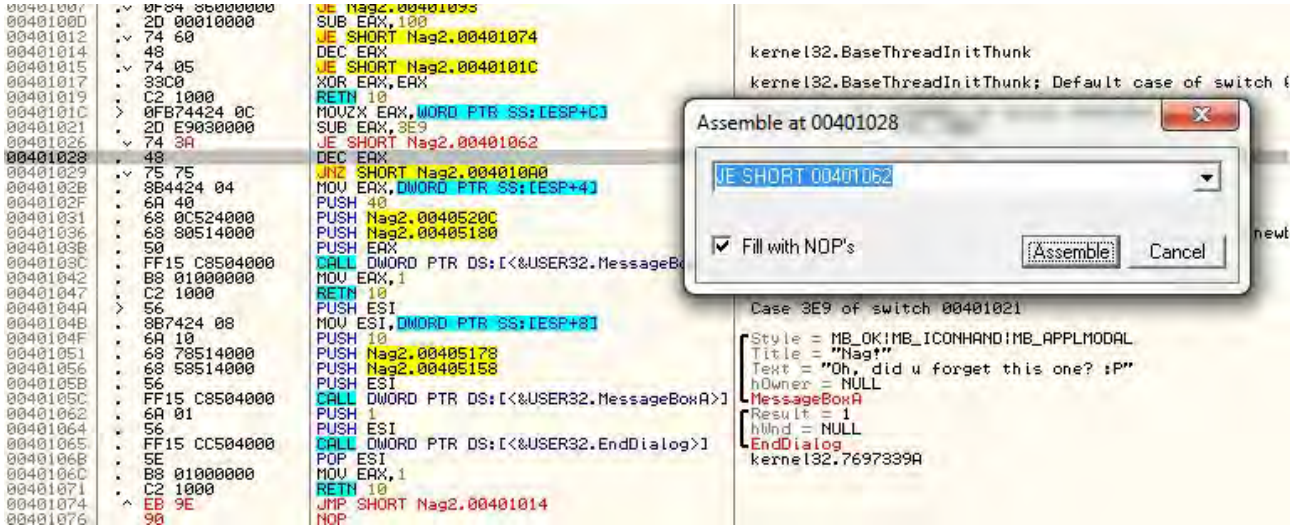
你可能首先会想到 “我们为什么不能像上一个那样，放置一个跳转让它再跳回去呢。” 好吧，你再仔细看看这个方法，我们能看到它调用了第二个 nag，但是然后它立即调用了 EndDialog。所以跳回去不会起作用，因为我们的对话框永远不会被关闭：



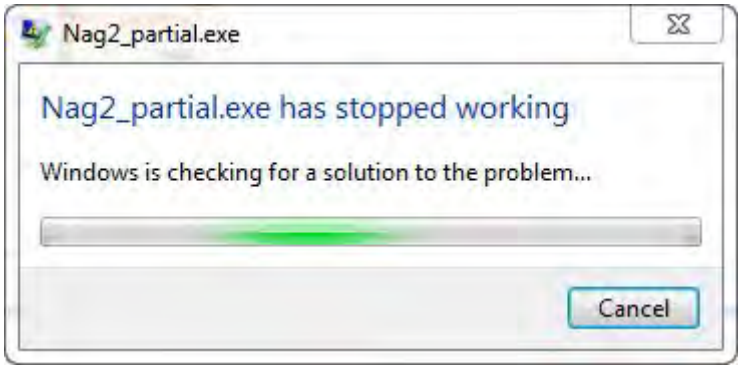
所以接下来你可能会想“咱们就把 401026 的 JE 指令修改成跳到 EndDialog，跳过显示 nag 的 MessageBoxA 的指令”。这个想法不错，咱们来试试：



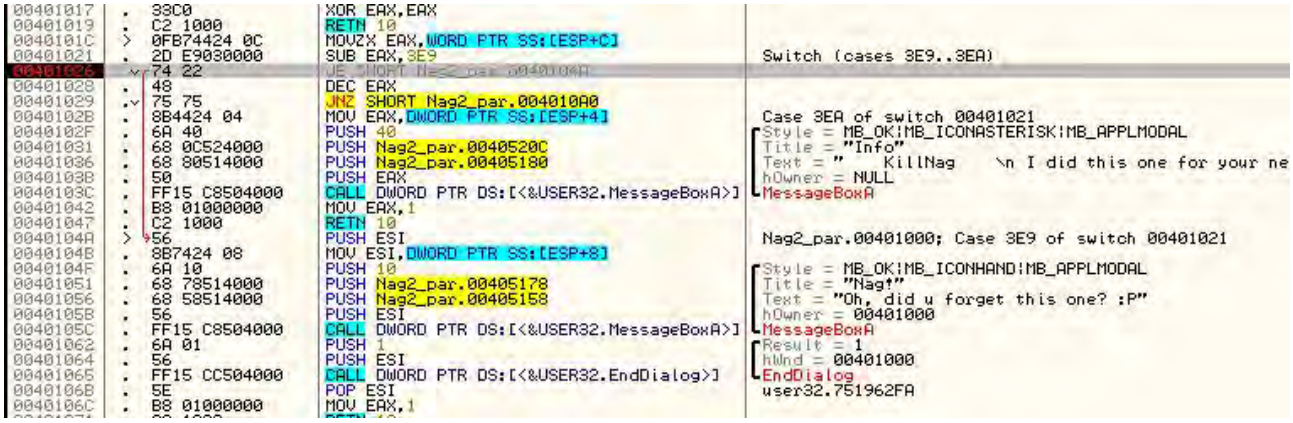
将 401026 的 JE 指令修改成跳转到 401062，也就是跳到 EndDialog 的第一行：



运行程序：

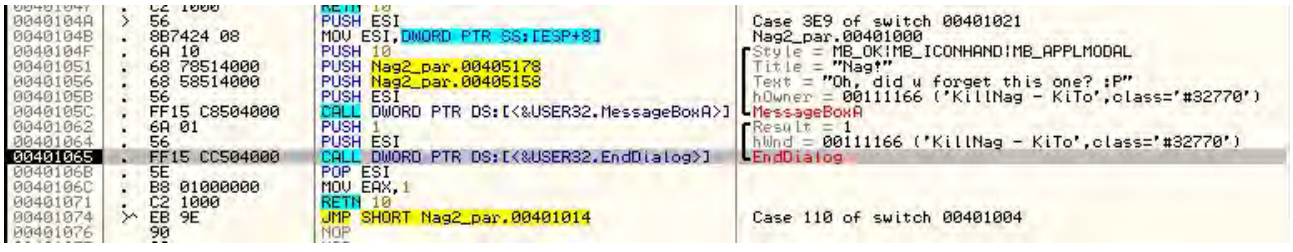


看起来前途灰暗呐。我们明显做错了什么。下面是我们准备做的：运行没有补丁的程序，单步执行它，看看是什么情况。然后再运行带补丁的程序，再看看两者之间有什么不同。重启应用并点击 “Exit”，我们会断在打补丁的地方（因为我们重启了应用，所以补丁消失了）：

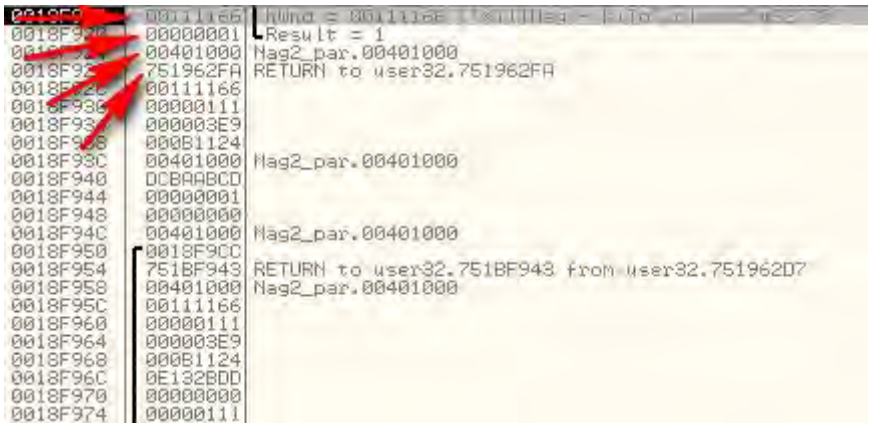


单步运行几行，当你单步步过对 MessageBoxA 的调用时，你就会看到 nag 窗口：

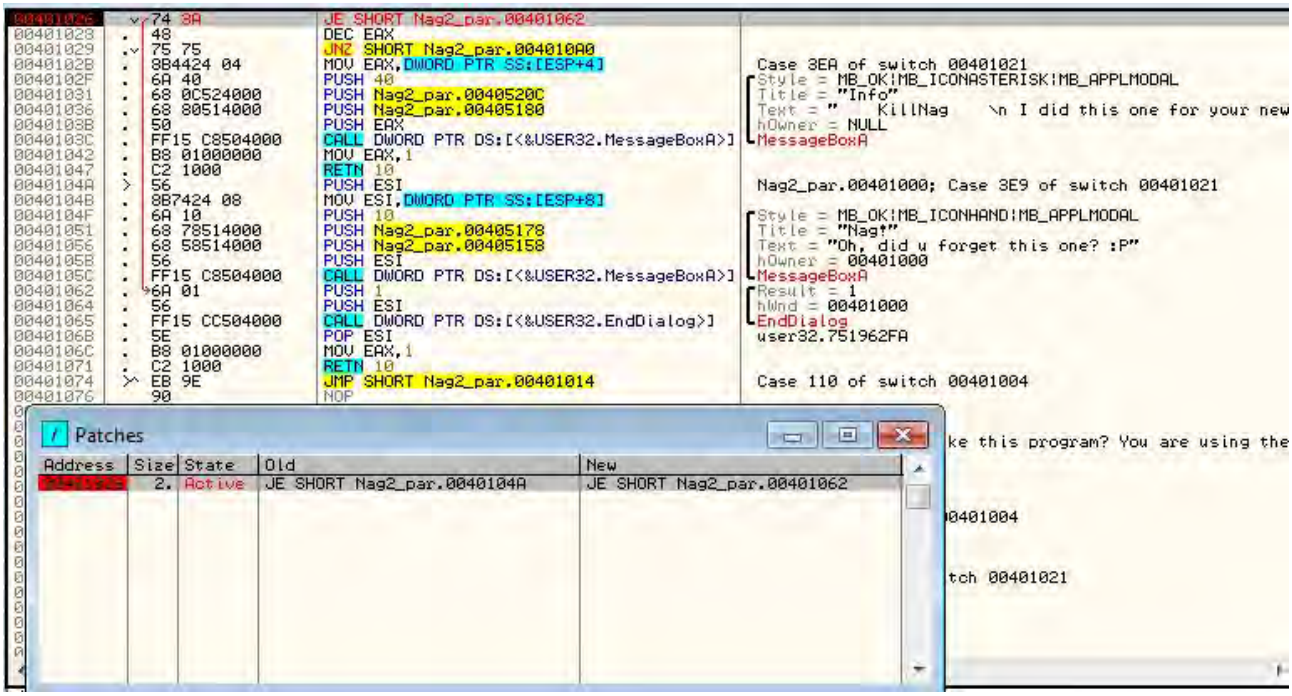
再单步两次，直到对 EndDialog 调用的那个 CALL：



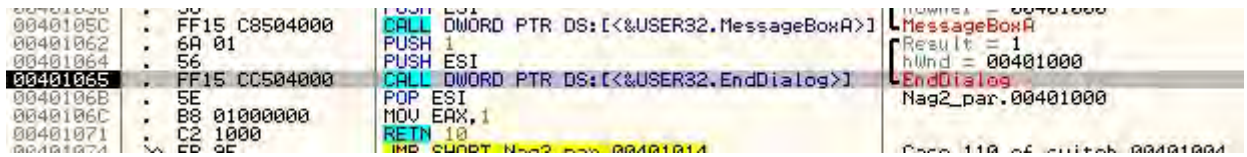
咱们来看看堆栈。在堆栈中我们看到四个项目：我们窗口的句柄、对话框的返回值、指向第一行代码（401000）的指针、返回到 user32 的地址。



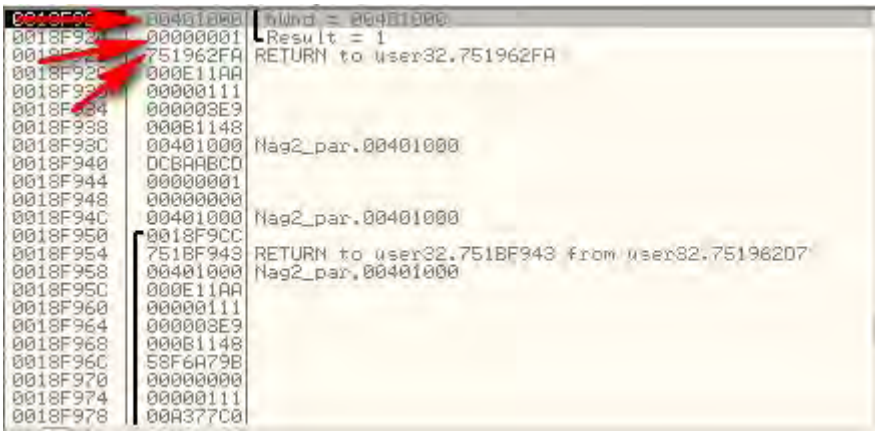
重启应用，当我们来到打补丁的地方后将其激活（打开补丁窗口，选中后按一下空格键）:



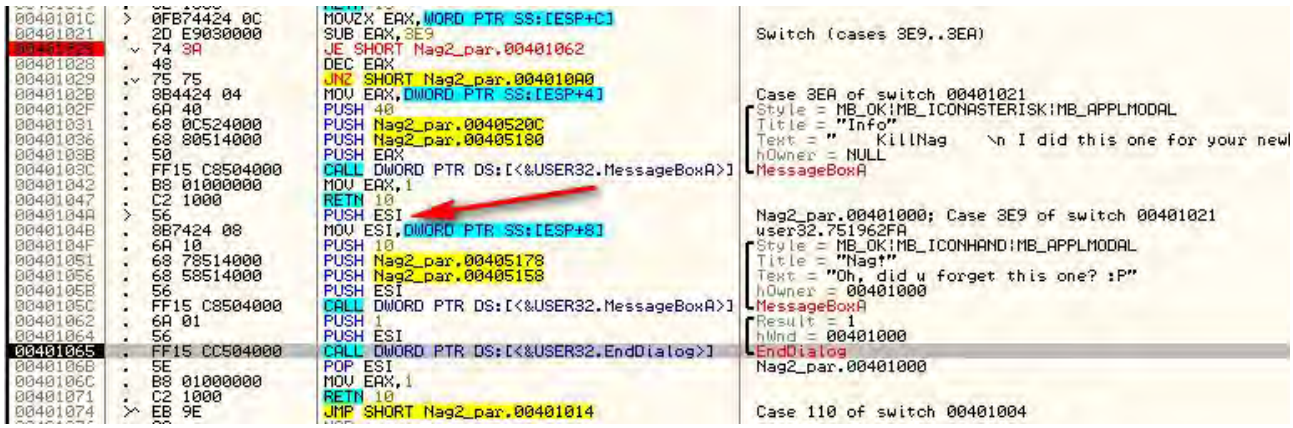
现在我们将跳过 nag 消息框。单步执行直到来到对 EndDialog 的调用处:



看看堆栈，有窗口的句柄、返回值、返回到 user32，但是没有了指向代码的第一行也就是 401000 的指针!!!

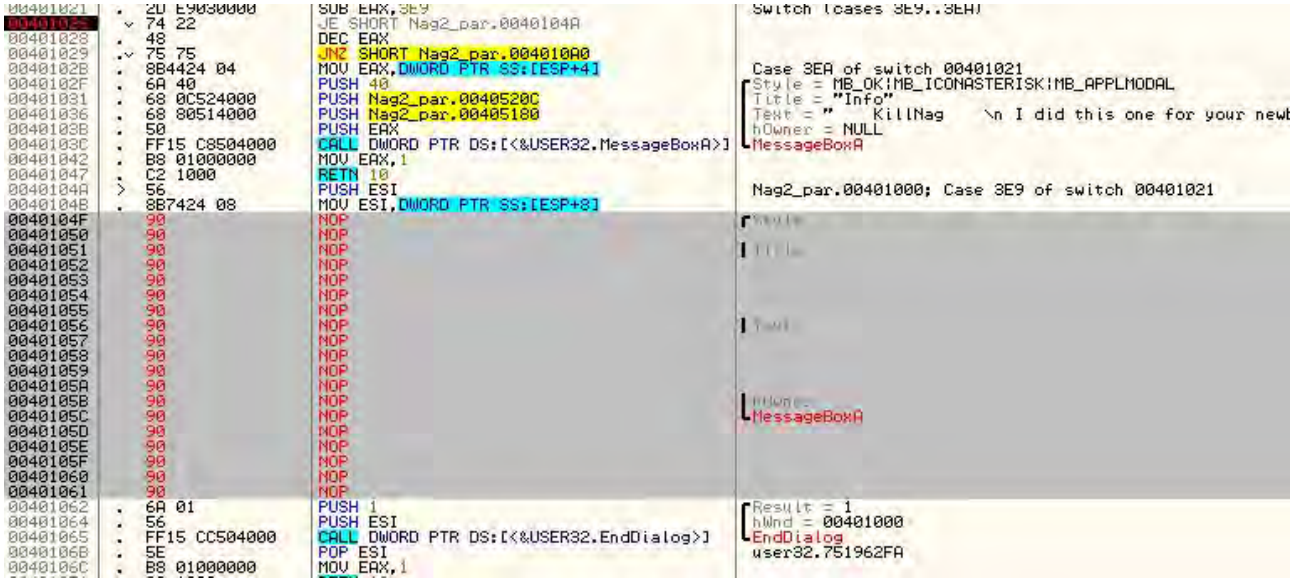


如果你向上滚动看看第二个 nag 的 CALL，你会发现在消息框被创建之前，ESI 被压栈了。这是一个指向我们代码的指针。在调用消息框前，它刚好被压入堆栈，虽然它也可以在这之后被压栈。所以我们丢失了一个重要的 push 操作，为了正确的运行 EndDialog 程序需要它。问题是，我们有一些想要的初始化代码，然后是一个我们不想要的对 nag 窗口的 CALL，之后又是一个我们想要的对 EndDialog 调用的 CALL:



好吧，那咱们就去除掉不想要的代码。选中 MessageBoxA 指令（从 40104F 到 40105C）然后右键，选择 “Binary” -> “fill with NOPs”:

然后，砰！再也没有对 nag 的 CALL 了:



现在当你在运行程序的时候，会发现程序可以正常关闭了。你可以保存补丁，并且没有 nag 窗口了。

第十五章：调用栈的使用

一、简介

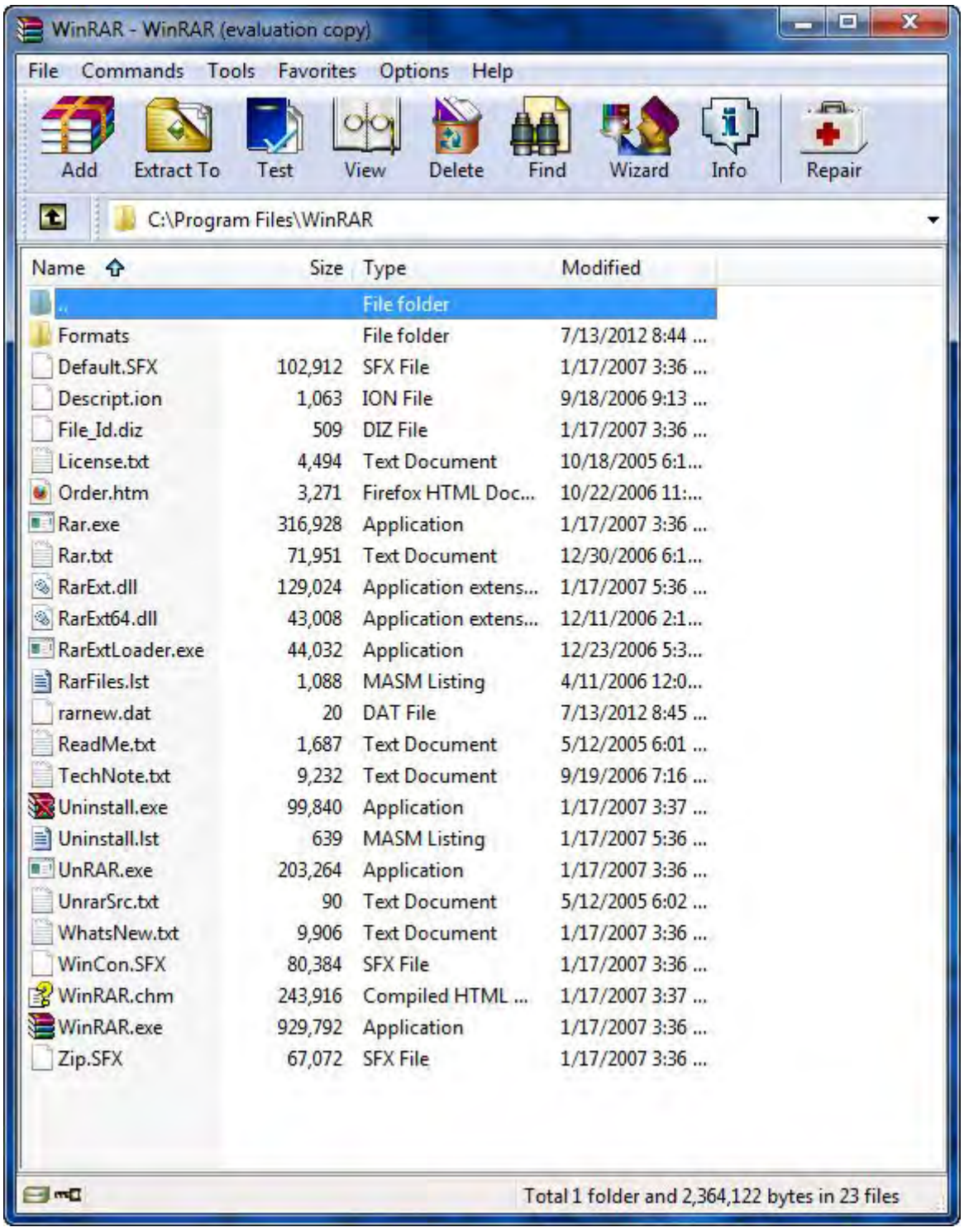
本章我们会删除一个“真正”的程序的 nag 窗口。为了试图帮助作者，因为他们花了大量的时间来创造这些应用，我试着挑出一个能将伤害降到最低的应用。这次，我用 Google 搜索了下“Cracked Software”，这个程序有着最高的点击率，包括教程、序列号、keygen，应有尽有。因为获得该软件的破解版是如此的简单，我想不管怎么获得它都不太可能有麻烦。不过我们请求你，如果你确实喜欢它，那就买它。

我也会添加一些技巧到咱们的逆向兵器库中。有一点需要注意，如果你是在 64 位 Windows7 下学习本章（像我一样），Olly1.1 版甚至我的版本，调用栈这招就不好用了。我的建议是，做我所做的：用 Olly2.0 版来学习新的技巧（取得正确的地址），然后再转到我所用的 Olly 来做其他的。或者就用 Olly2.0 版，它有很多很好的特性，它可以在 64 位操作系统下工作。（译者注：既然有了 Olly2.0，那我们为什么在平时用的时候还是以 1.1 的版本居多呢？因为 1.1 版的 Olly 拥有大量的插件，这给破解带来了很大的便利，而 2.0 版的插件则弱了很多。）

你可以在教程页下载本章的相关文件。

二、探究该应用

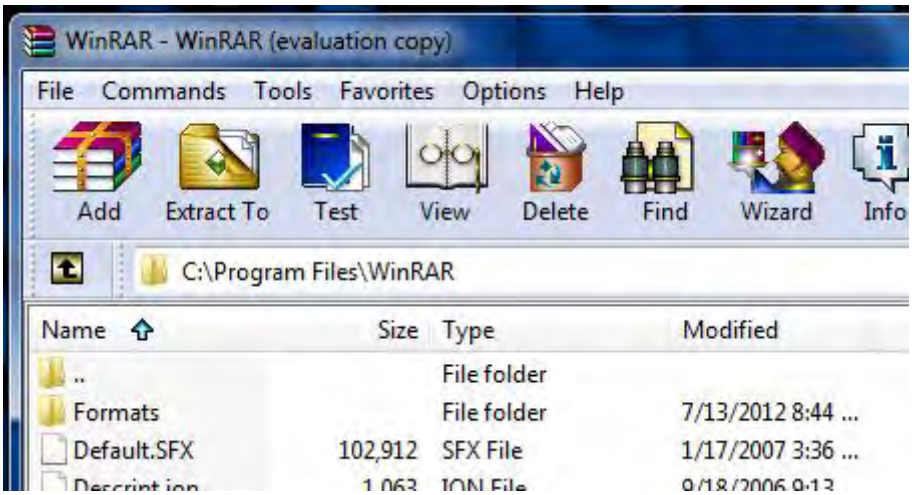
该程序有一个 40 天（可能和圣经有关？）的使用限制，40 天后会弹出一个 nag 窗口。相信我，根据以往的经验，它肯定很啰嗦。不幸的是，因为 nag 窗口 40 天（40 个晚上？-对不起。）内都不会出现，所以你有两个选择：你可以安装该应用，然后等上 40 天再阅读本章；或者你可以将系统时间设置成今天加上 41 天后练习本章，然后再将日期设置回今天。你要确保在练习本章前做了其中的一个，否则它不匹配。


WinRAR - WinRAR (evaluation copy)
File Commands Tools Favorites Options Help
Add Extract To Test View Delete Find Wizard Info Repair
C:\Program Files\WinRAR
Name Size Type Modified
.. File folder
Formats File folder 7/13/2012 8:44 ...
Default.SFX 102,912 SFX File 1/17/2007 3:36 ...
Descript.ion 1,063 ION File 9/18/2006 9:13 ...
File_Id.diz 509 DIZ File 1/17/2007 3:36 ...
License.txt 4,494 Text Document 10/18/2005 6:1...
Order.htm 3,271 Firefox HTML Doc... 10/22/2006 11:...
Rar.exe 316,928 Application 1/17/2007 3:36 ...
Rar.txt 71,951 Text Document 12/30/2006 6:1...
RarExt.dll 129,024 Application extens... 1/17/2007 5:36 ...
RarExt64.dll 43,008 Application extens... 12/11/2006 2:1...
RarExtLoader.exe 44,032 Application 12/23/2006 5:3...
RarFiles.lst 1,088 MASM Listing 4/11/2006 12:0...
rarnew.dat 20 DAT File 7/13/2012 8:45 ...
ReadMe.txt 1,687 Text Document 5/12/2005 6:01 ...
TechNote.txt 9,232 Text Document 9/19/2006 7:16 ...
Uninstall.exe 99,840 Application 1/17/2007 3:37 ...
Uninstall.lst 639 MASM Listing 1/17/2007 5:36 ...
UnRAR.exe 203,264 Application 1/17/2007 3:36 ...
UnrarSrc.txt 90 Text Document 5/12/2005 6:02 ...
WhatsNew.txt 9,906 Text Document 1/17/2007 3:36 ...
WinCon.SFX 80,384 SFX File 1/17/2007 3:36 ...
WinRAR.chm 243,916 Compiled HTML ... 1/17/2007 3:37 ...
WinRAR.exe 929,792 Application 1/17/2007 3:36 ...
Zip.SFX 67,072 SFX File 1/17/2007 3:36 ...
Total 1 folder and 2,364,122 bytes in 23 files

过了一会...。


Please purchase WinRAR license
Please note that WinRAR is not free software. After a 40 day trial period you must either buy a license or remove it from your computer.
Buy online
How to buy
Close
Help

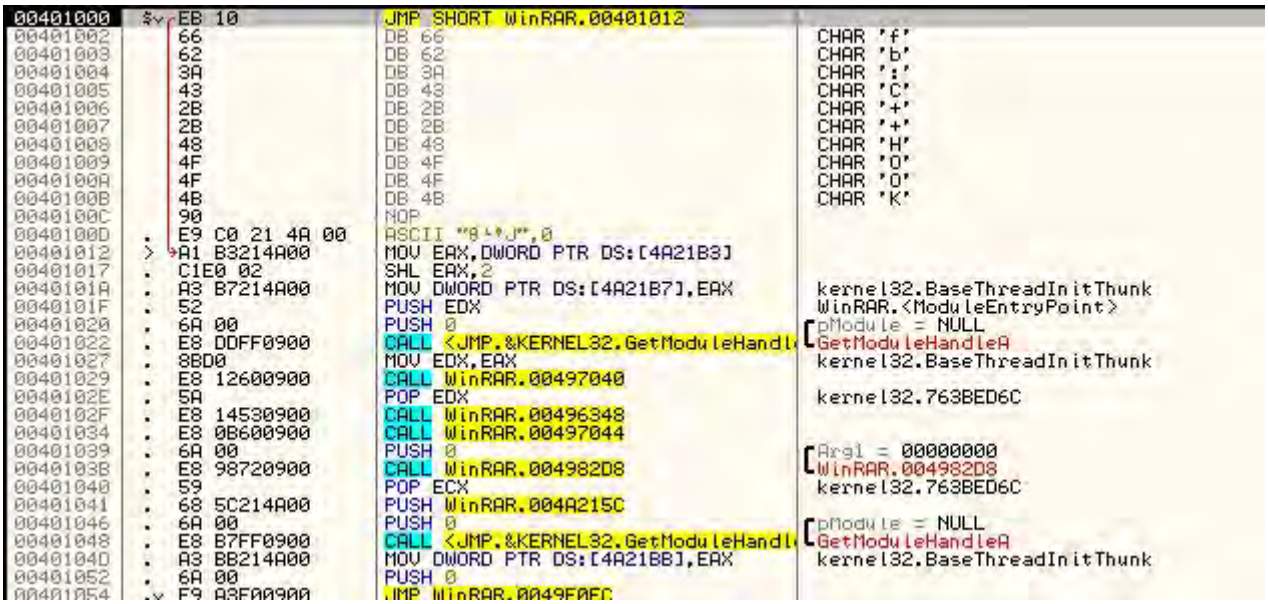
nag 窗口弹出来了。在使用的时候它出现了很多次。这真是很烦人。我们在顶部也可以看到 “evaluation copy”:



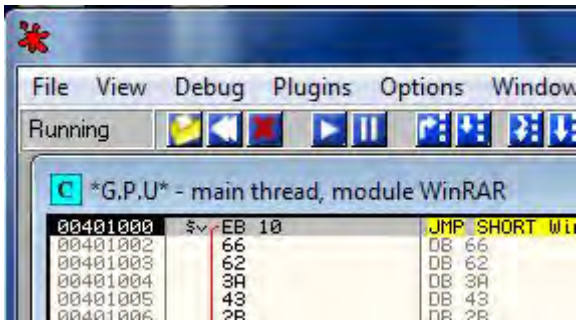
我打算介绍两种方法来获取相关注册码。

三、第一种方法

Olly 载入并启动程序:



启动应用后，等 nag 出现。它一出现（在关闭它以前），切换到 Olly，点击暂停按钮（靠近 play 的那个）:

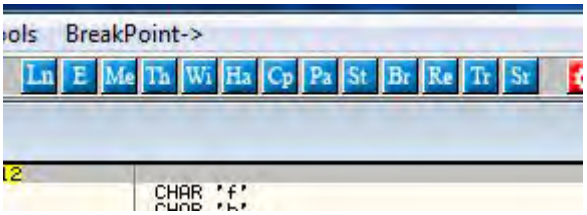


现在，我们要找出那个 nag 窗口是从哪来的，最后找出是谁让它显示的。我们当然也可以搜索字符串或模块间调用，不过我向你保证，这些技巧对于外面的大部分应用都不管用。所以咱们学习另外一个技巧...。

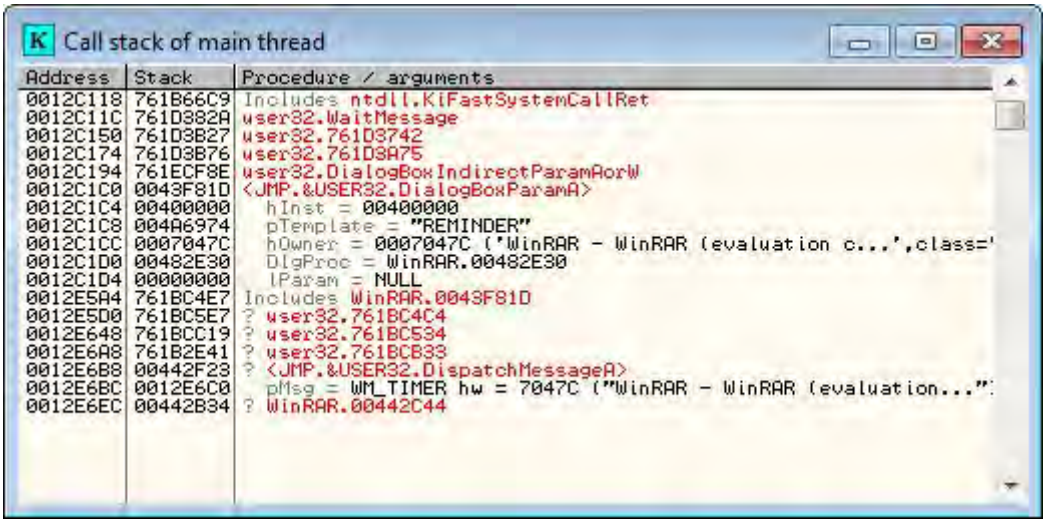
四、调用栈

调用栈是 Olly 尝试跟踪让我们到达某处位置的代码，从而试着找出哪个函数被调用。它也尝试向你显示被传递给函数的参数。所有的这些都可以通过右下角的“普通”的堆栈来完成，不过通过调用栈用来查看这些数据要更好用。要记住 Olly 在这方面不是很完美，你不能把这个窗口的所有东西都当做福音（糟糕，我又犯了同样的错误。译者注：我也不知道这句话啥意思。）。要做很多猜测。当然，有很多次，这个窗口是空的。通常是因为 Olly 完全糊涂了，或者是在逆向一个 VB 程序（VB 程序在调用函数的方式上和真正的程序不太一样）。

要查看调用栈，如果你用的是我的版本的话，点击“St”按钮:



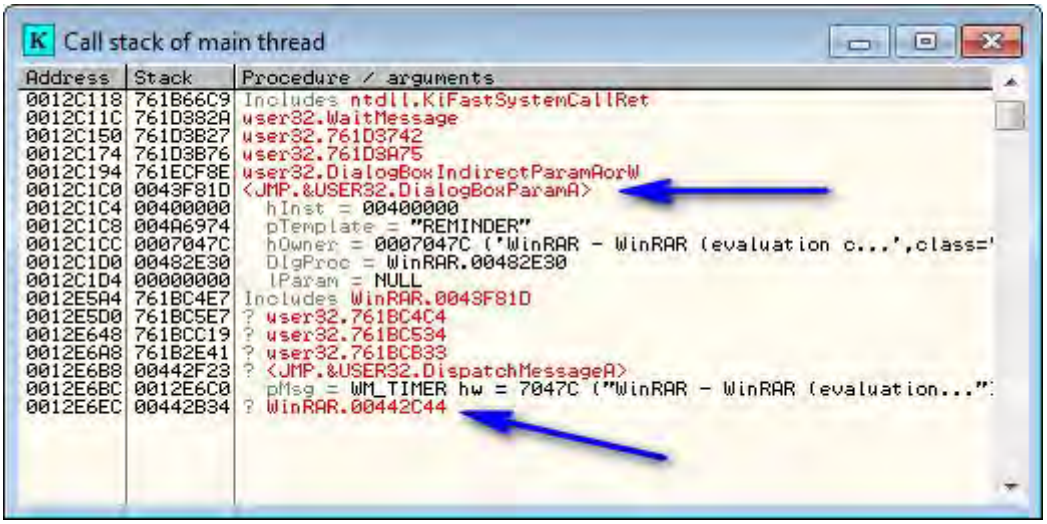
如果用的是原版的 Olly，点击工具栏中的“K”按钮。似乎“Call”这个词在作者的母语中是以“K”打头的:



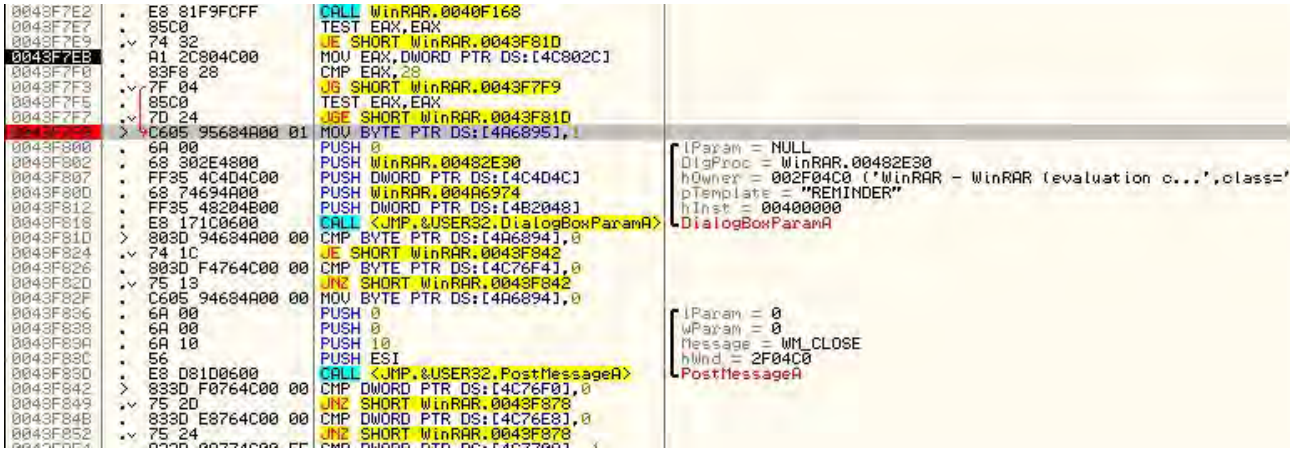
有几件事需要点出来...。最近的调用是在顶部，和堆栈类似。“Includes”意思是该指令在那个 CALL 中有涉及到，不过 Olly 不能够准确的知道是怎么回事。问号的意思是 Olly 对该行没有把握，所以你得为自己带盐（译者注：原话是“take it with a grain of salt”，意思是需要进行分析，要斟酌斟酌，不能全信。）。

在咱们的例子中，可以看到一个 ntdll 函数、几个 user32 函数、对 DialogBoxParamA 的带参数调用、又是几个对 user32 的调用，底部是对程序 WinRAR 本身函数的调用。下面是对这些内容的思考：WinRAR 在地址 442C44

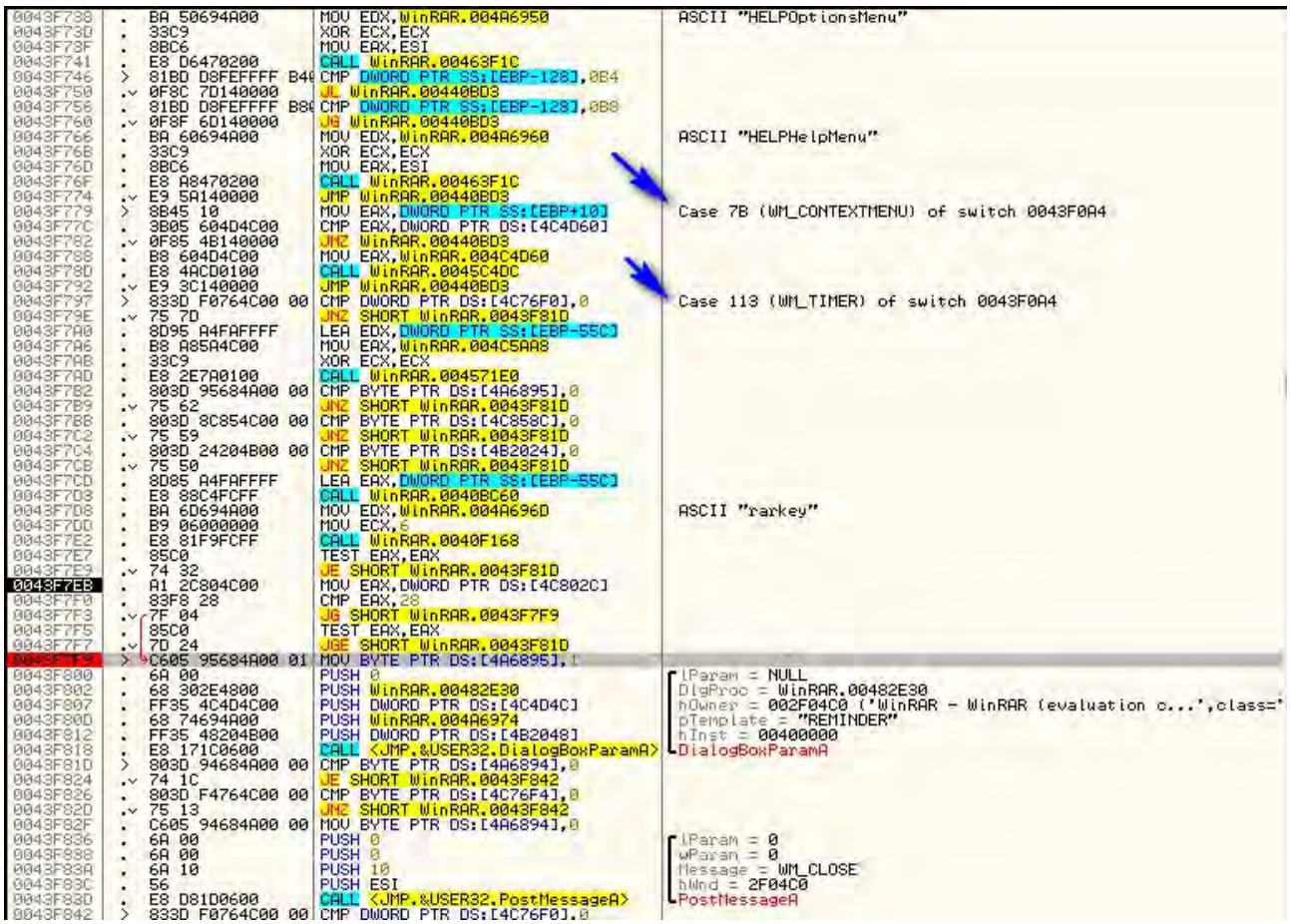
处调用了 DispatchMessageA，这里用一个消息来显示对话框。然后 User32 调用了 DialogBoxParamA 函数来显示对话框，标题是 “evaluation copy”，还有其他几个参数。然后 User32 显示对话框并等待我们的输入，它使用 WaitMessage 来做这个。



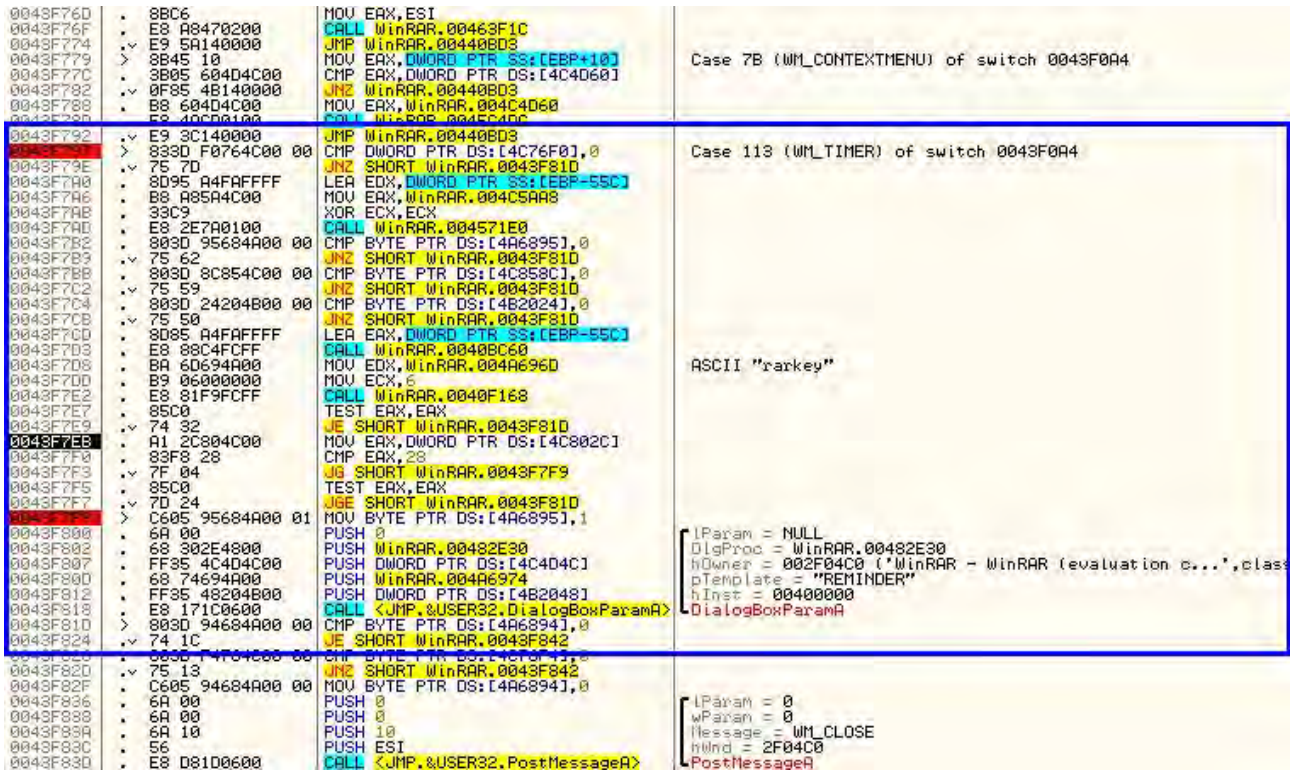
这个窗口里重要的是调用显示对话框的那个 CALL 以及应用自己的 CALL。通常使用调用栈时，从顶部开始，找到你感兴趣的可以用来找到代码区块的第一项。如果这个不好用，继续向下找，检查每一个函数调用，直到你“回到”代码足够的远，以找出决定该函数是否被调用的那个 比较/跳转 指令组合。通过双击那行，咱们来试试检查下对 DialogBoxParamA 调用的那个 CALL:



我们来到了调用 DialogBoxParamA 的地方。我在执行设置以及调用显示对话框这些指令的开始处设置了一个 BP。在它的上面，有几个条件跳转映入眼帘。如果你再向上滚动，你会发现有几个写着 “Case XX (WM_Something) of switch 0043F0A4” 的注释，这里的 XX 是一个十六进制数：

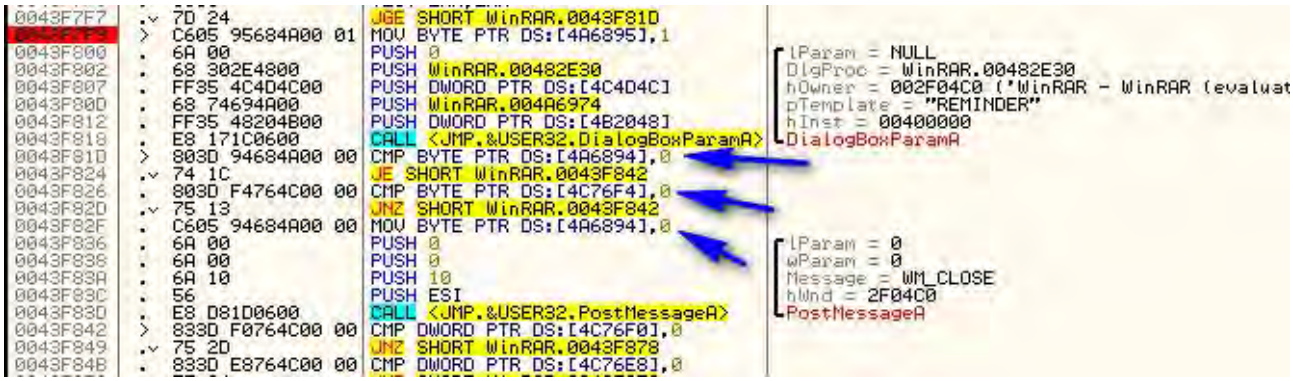


这是 Olly 显示 switch 语句的方式。如果你往上滚动，你会发现这真是一个相当大的 switch 语句。如果你有 Windows 编程经验，你可以认出“WM_SOMETHING”句子是 Windows 消息，你也可以认出这一整块代码是作为 Windows 消息的消息处理过程。如果你对这些一无所知也没关系，在后面的章节中我们会非常细致的讲解 windows 消息处理过程。目前来说，我们只对涉及到对话框调用的部分感兴趣。下面，我们可以看看这整个分支（case）:



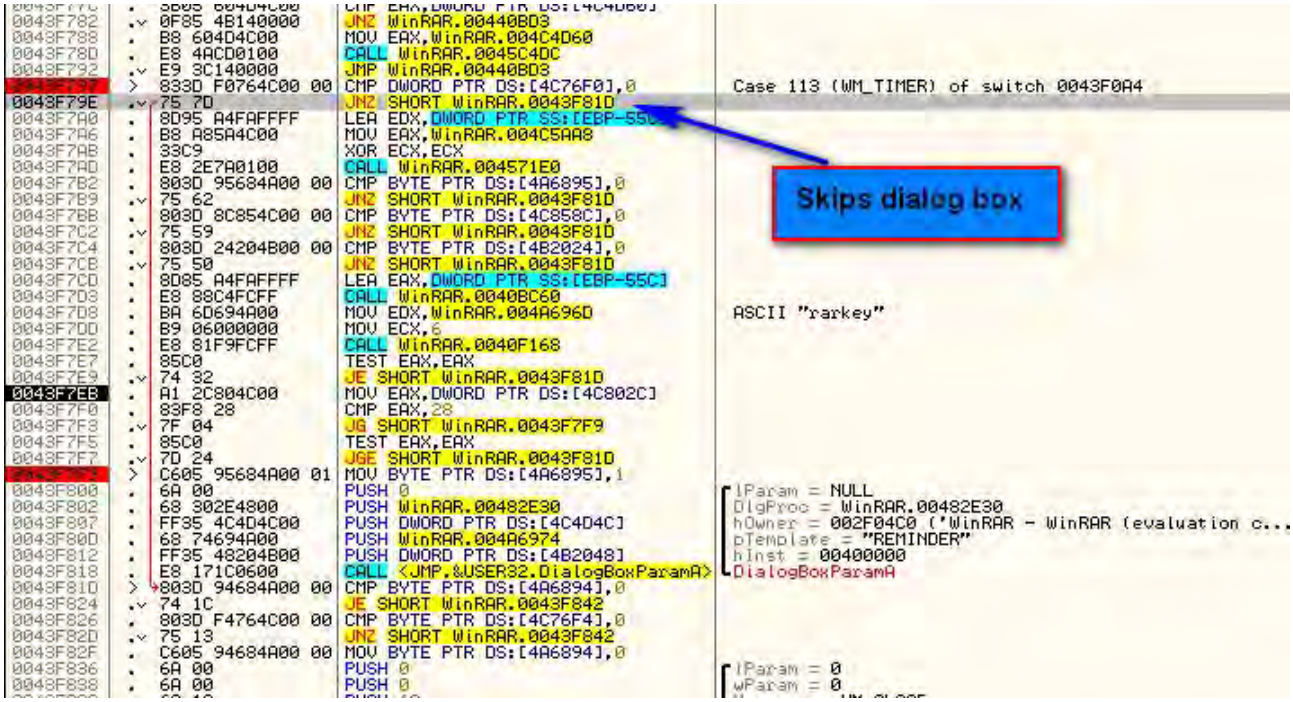
你会发现它是在处理 WM_TIMER 消息的区块。这很能说明一些问题。为什么我们的对话框是在一个对计时器超时的消息处理中？我们马上就会看到....

还要注意在对话框被调用后，有几个条件跳转和比较语句:

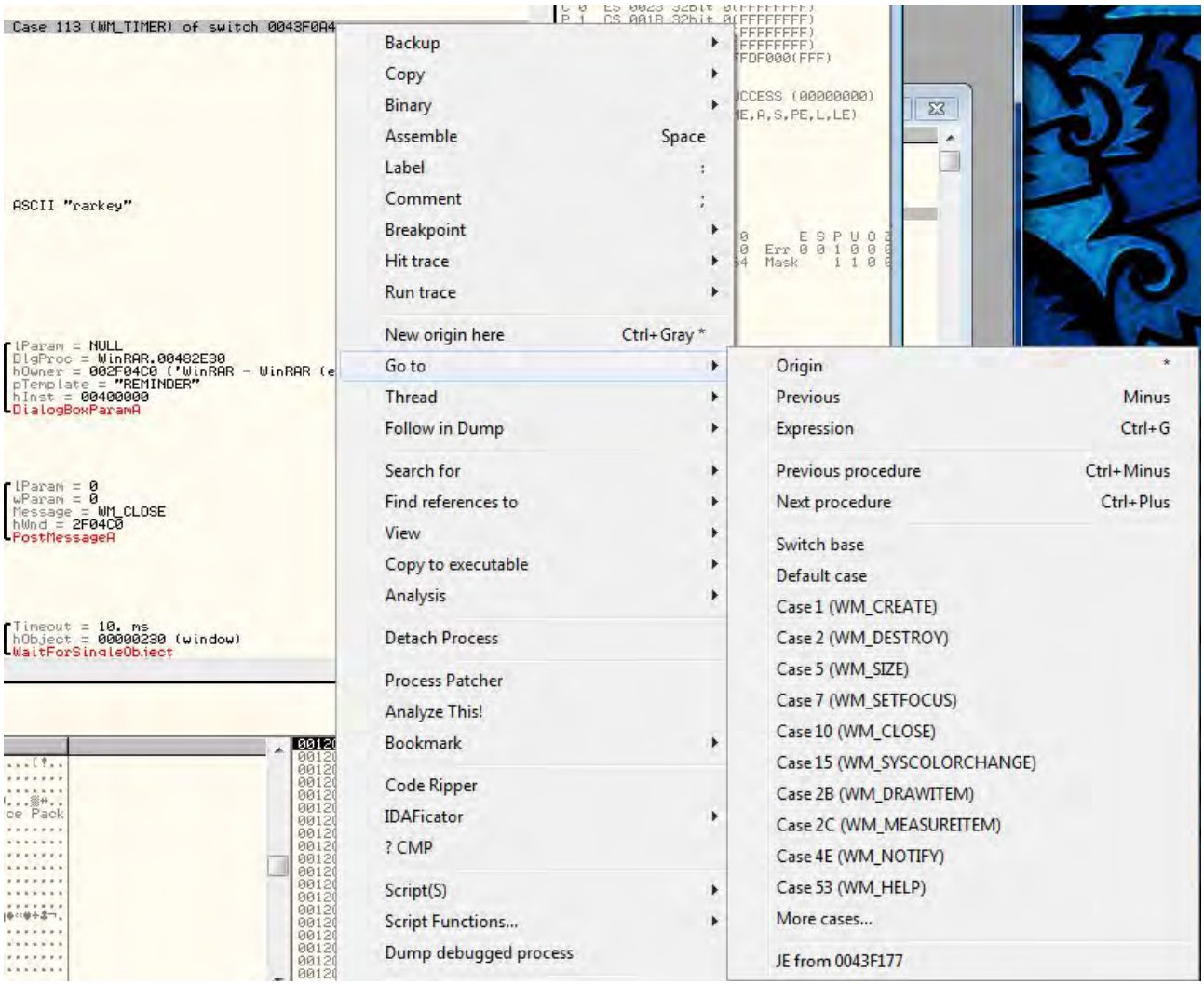


这些跳转执行的代码依赖于我们点了对话框中的什么。如果你点的是”Close“，它会跳到关闭窗口的代码等。

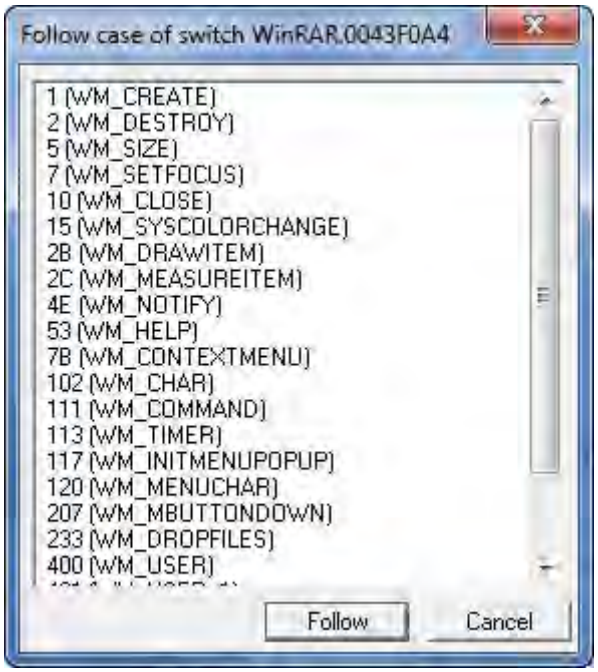
滚动到 switch/case 语句块的起始处，能够发现那里有一个初始的比较和跳转指令:



这个跳转跳过了打开 nag 窗口的那个 CALL(还有其他很多代码也被跳过)。咱们来看看这个初始的 比较/跳转 是啥。在该行上右键，也就是有“Case 113 (WM_TIMER)”的那行，选择“Goto”:



在弹出的下拉菜单中你可以看到，Olly 向我们显示了可以被这个 switch 语句处理的好几个 case。点击 “More cases. . . ”，会弹出一个对话框向我们显示全部的 case:

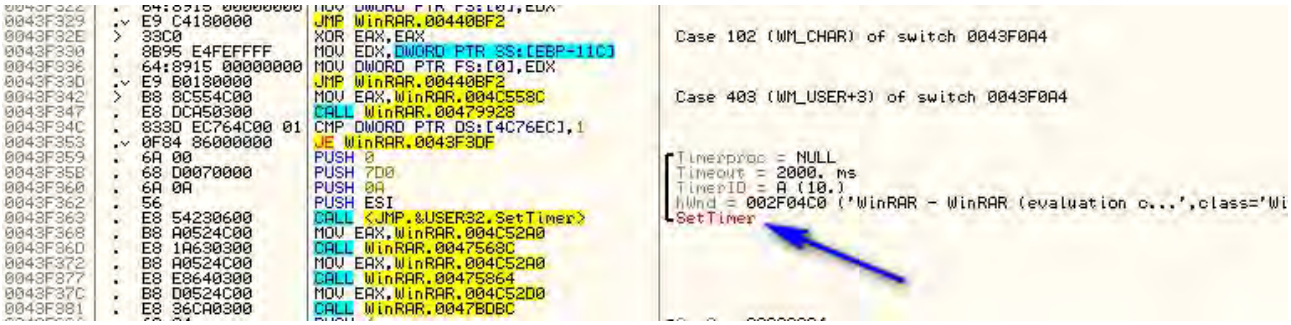


点击其中几个，然后点击 “Follow”，你会跳到处理相应 case 的代码。你会发现，所有这些 case 的开始都是一个 比较/跳转 组合。意思就是汇编语言处理 switch/case 语句，是作为巨大的 if/then 语句来处理的。有点像下面这样（用伪代码来表示）:

```
if (msg != WM_CREATE)
    jump to next if
Do WM_CREATE code
Jump to end
if (msg != WM_DESTROY)
    jump to next if
Do WM_DESTROY code
Jump to end
if (msg != WM_SIZE)
    jump to next if
Do WM_SIZE code
...
```

所以在每一个 case 的起始处，都要检测该 case 是不是用于处理特定的消息，如果不是就跳到下一个比较。如果是，就忽略跳转，直接转到处理消息的代码部分。

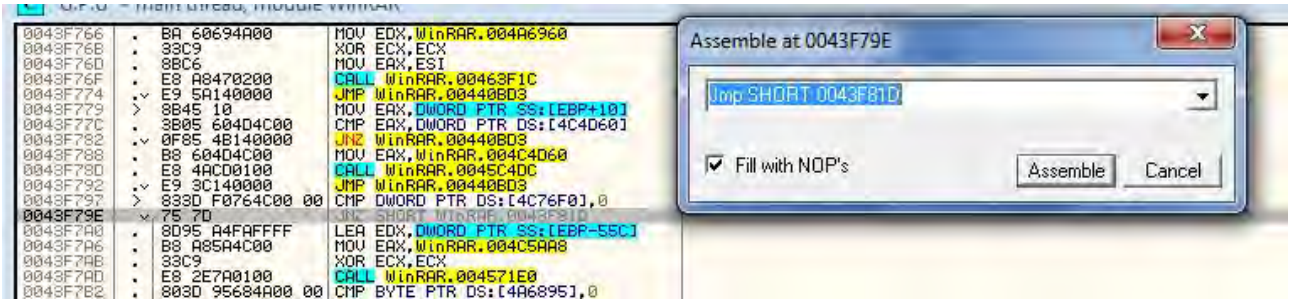
现在，因为我们的 case 是 WM_TIMER 消息，我们可以知道（通过在 Google 搜索 WM_TIMER 消息），这个是用来处理计时器超时时的消息。也就是说，在某个地方计时器必须被启动。向上滚动（多滚一点），我们看到了罪魁祸首：



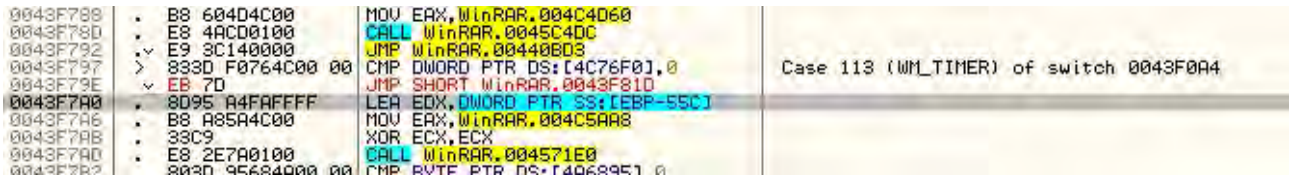
那么现在，我们可以猜测怎么来覆盖这个 nag 窗口了....

五、给程序打补丁

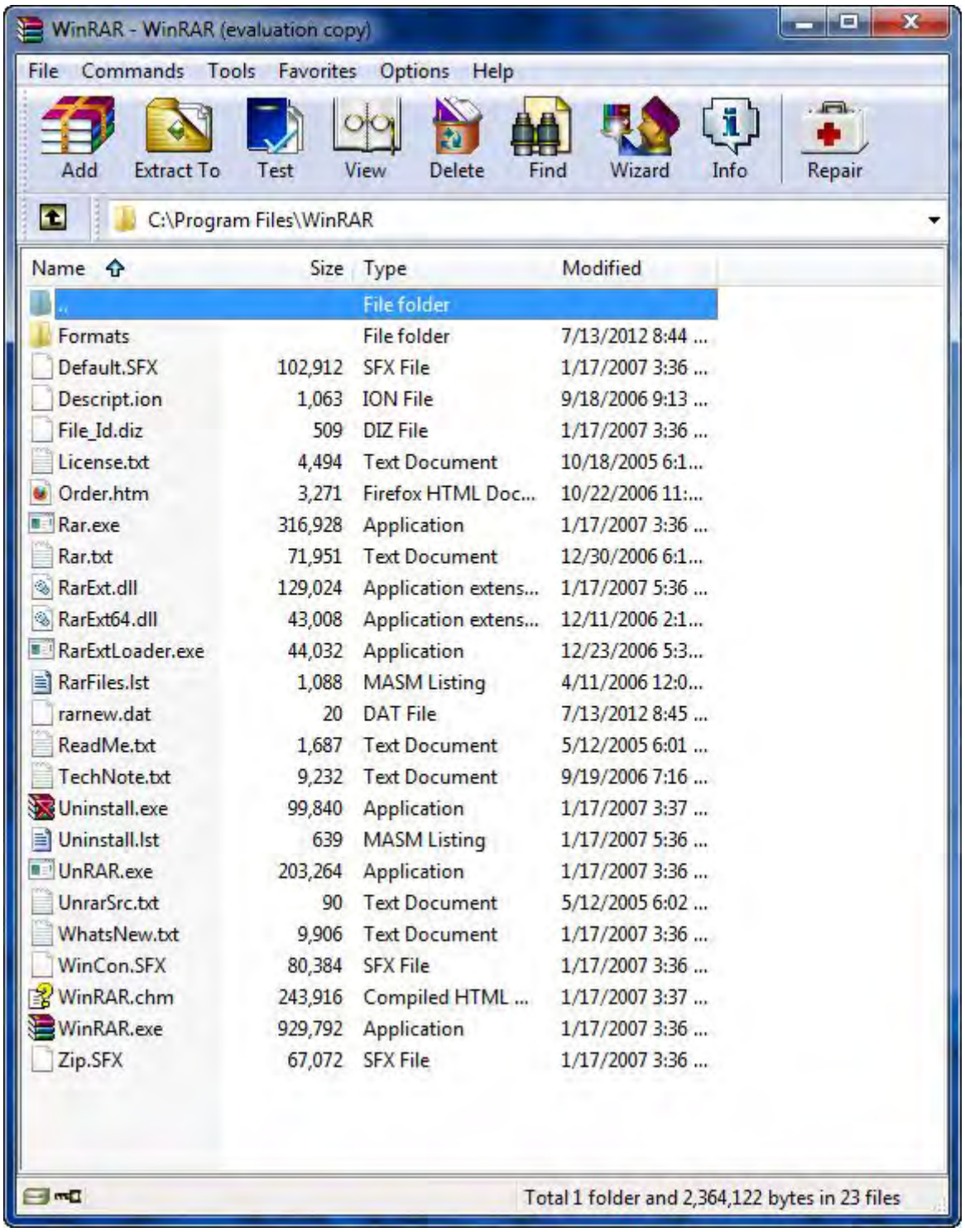
最简单的方法是，当计时器超时时，让消息处理过程什么都不做。做这个的最简单的方法是保证我们每一次都跳过这个 case。那转到该 case（113-WM_TIMER）的起始处，在这里它会检测是不是正确的 case，把它改成总是跳转：



下面就是打过补丁之后的样子：



现在，无论什么时候该消息过程得到了计时器超时的消息，它都会简单的忽略它。运行程序，过一会你会发现那个 nag 窗口再也不会出现了：



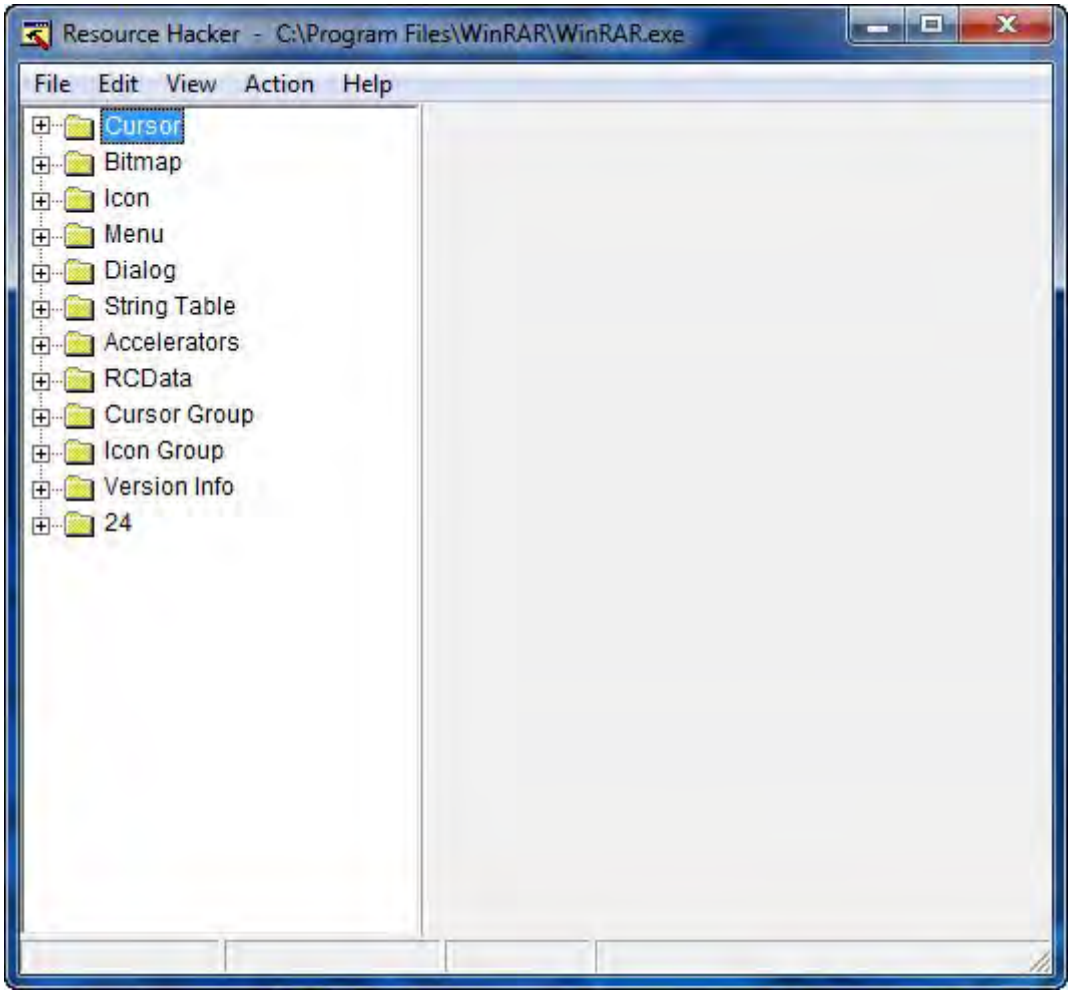
这个程序仍然会在标题栏显示“evaluation”，后面的章节我们会回到这个话题（修改这个有点复杂，你可以试试。这是学习的最好方法！）。不过目前，即使它显示“evaluation”，它也工作的很好并且永远不会过期。好吧，确切的说这不是真的，它仍然会过期，不过它却什么都不会做。确定你保存了补丁（和我们前面几章一样）以保存所做的修改。

六、第二种方法

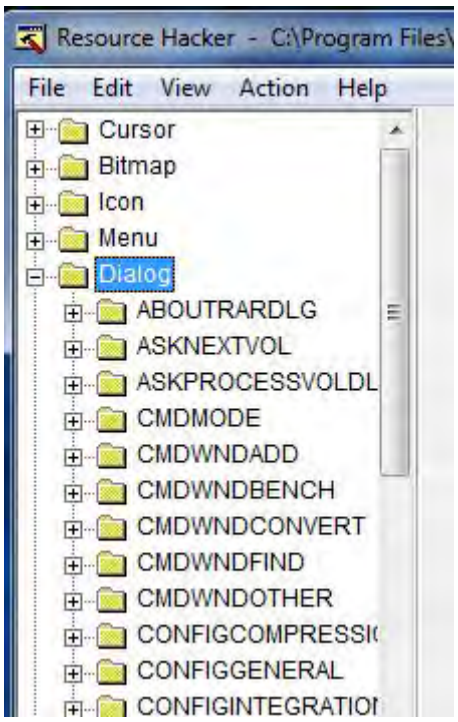
现在，我向你介绍我们能够用来找出对话框区块的另外一个方法（除了调用栈）。如果你还没有的话，先下载一个 Resource Hacker。你可以在工具页获

得它。Resource Hacker 可以让你查看和操纵一个 PE 文件内的资源。当我们讨论 PE 文件的构成时会更深入的讨论资源，不过就目前来说我们只需要知道应用程序所使用的任何资源（包括按钮、对话框、位图、图标、文本字符串）都存储在文件的独立区块，和代码是分开的。真正的，看看我所说的最好的方法是打开 Resource Hacker，载入几个程序看看。

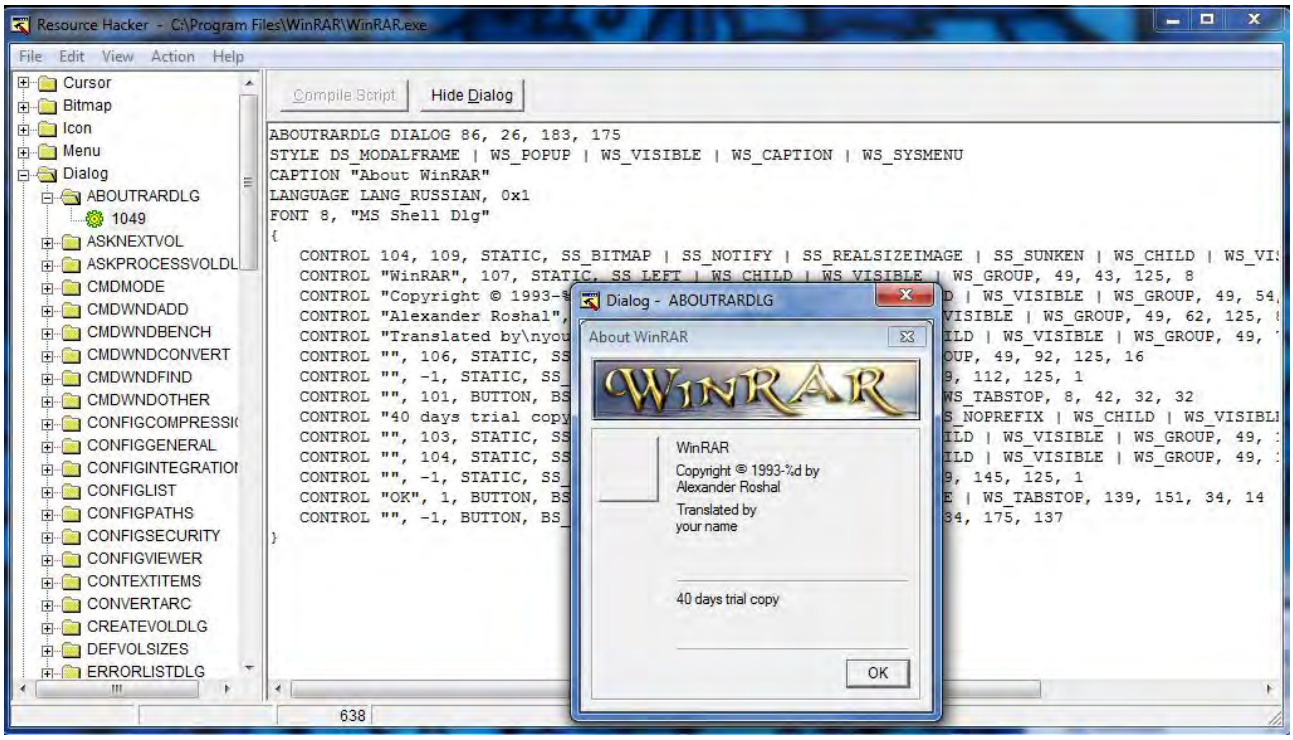
那么咱们就这么干....。打开 Resource Hacker，载入我们的应用：



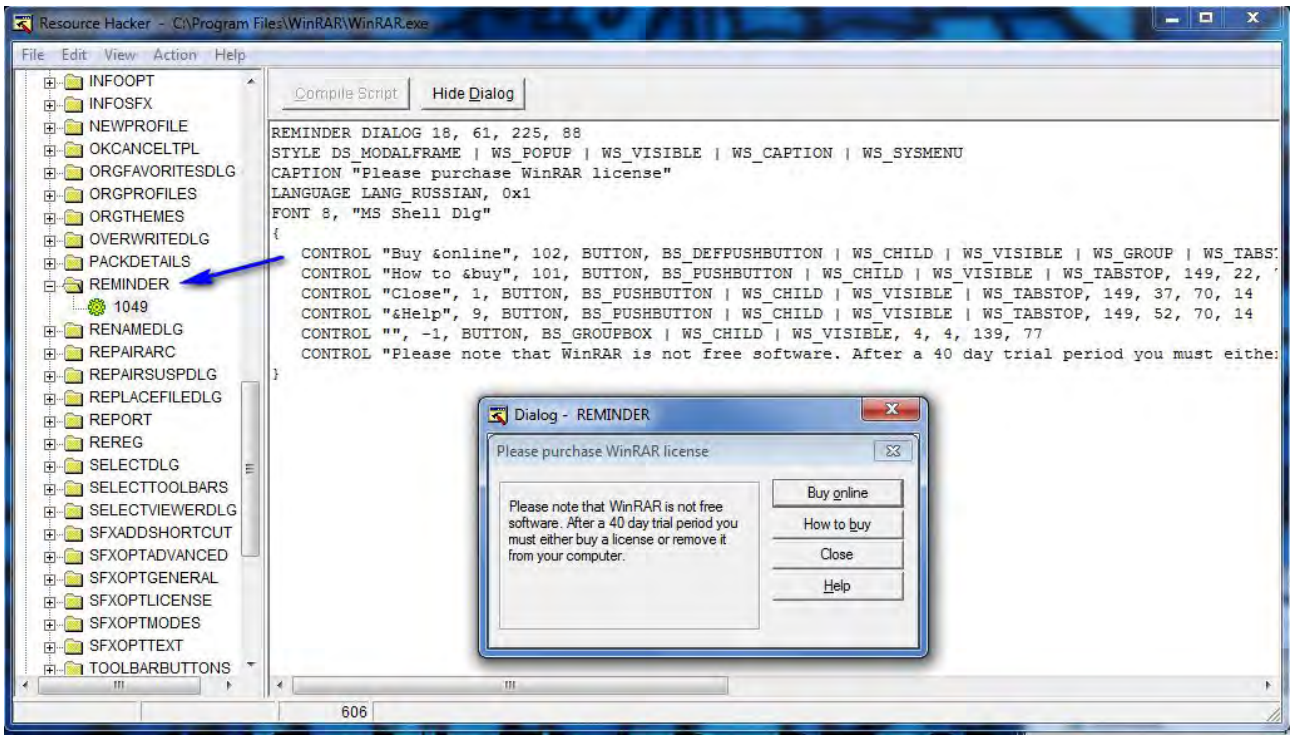
左边的树形列表显示了应用程序的各种资源。可以看到它包含有位图、图标、菜单，以及最重要的对话框：



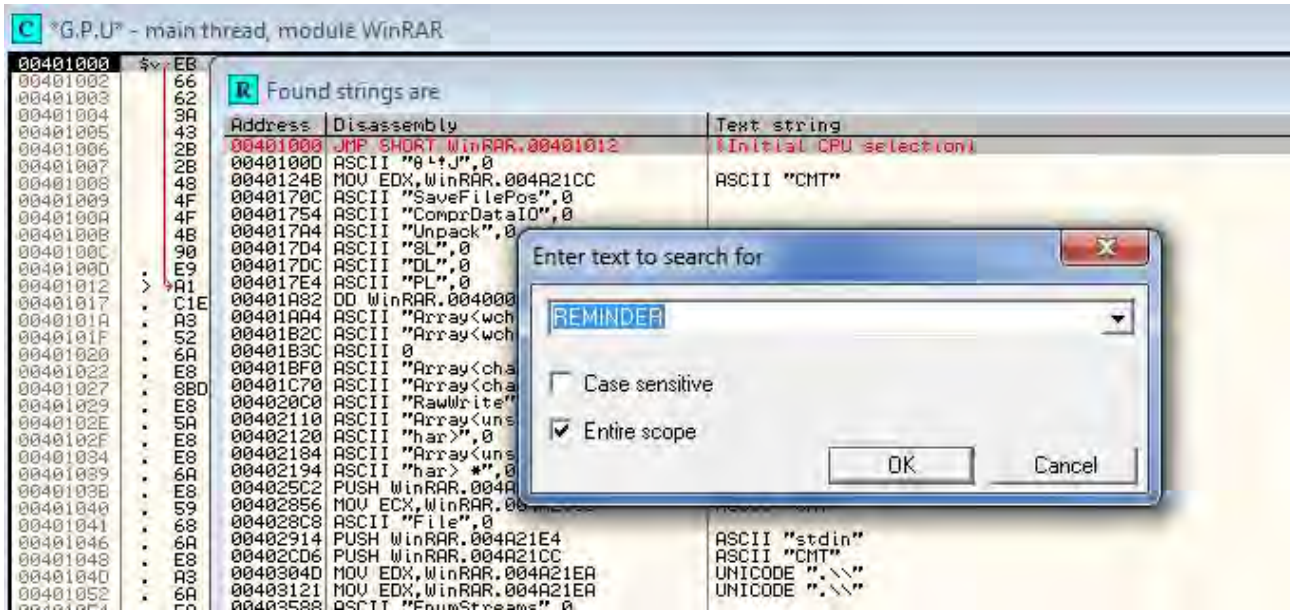
该应用有很多对话框。咱们继续，点击第一个，就是 ABOUTRARDIALOG:



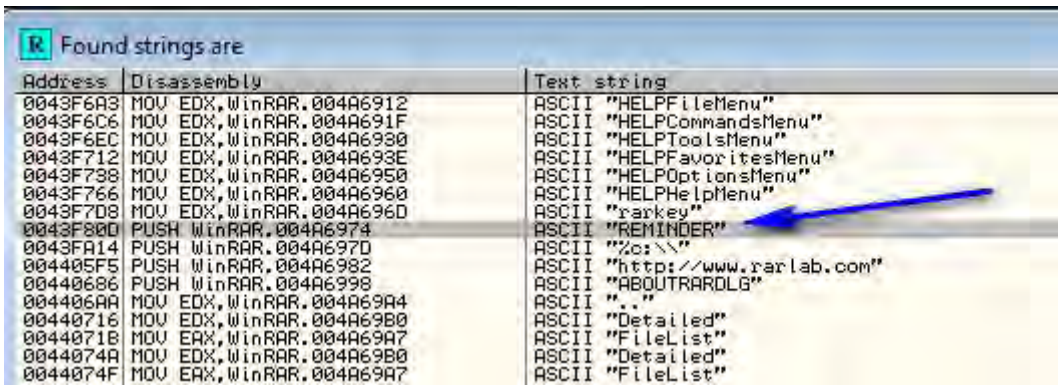
Resource Hacker 向我们显示了该对话框的相关数据，包括标题（显示在窗口标题栏上的），该对话框的相关按钮，以及它的各种设置。它也打开了一个窗口向我们准确的显示了该对话框的样子，这里的是关于对话框。在点了左侧一堆目录以后，就找到了我们想要的：



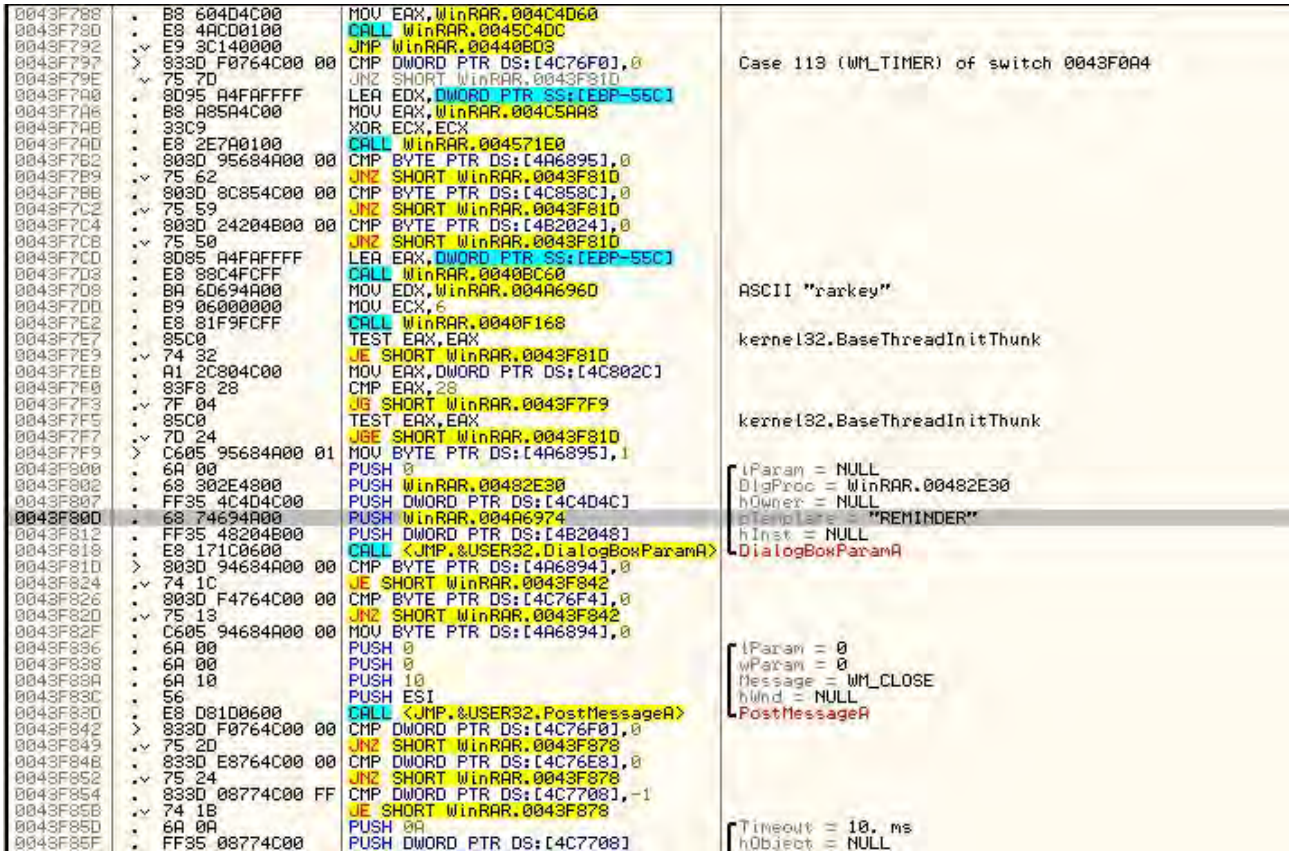
是不是看起来挺面熟的？注意对话框的名字是“REMINDER”。有时候 Windows 用名字来引用一个对话框，有时候用 ID。这里它用名字“REMINDER”。现在我们知道了所有我们需要的，Olly 载入应用，转到“search for strings”。咱们来搜索“REMINDER”：



咱们在列表中看到它了：



双击它，我们来到了与使用调用栈同样的区块:



事实上，你可以看到传递给 DialogBoxParamA 的其中一个参数就是“REMINDER”。如果资源是通过 ID 而不是名字来标示，我们可以通过右键反汇编窗口，选择“Search for” -> “Command”来找到它。然后在弹出的对话框中输入“PUSH xx”，xx 是资源的 ID（十六进制的）。这也会将你带至对话框的调用 CALL。

七、最后一件事

如果你看看众多的破解二进制文件的教程，会发现有一个方法是简单的通过 Resource Hacker 来删除对话框来实现的。本例中这是管用的，你再也不会看到 nag 窗口，但是这个方法不总是好用，因为它完全依赖于程序是如何处理丢失的资源的。之所以介绍这个简单的技术，是因为它总是值得试试的。

ps: 别忘了将你的日期改回去☺。

第十六章（上）：Windows 消息的处理

好，在干掉了两个病毒（一个是我身体上的，另一个是电脑的）以后，我最终还是上传了最新的教程。本章是其中的三分之一，所有三章处理的都是同一个 crackme（相当难的一个），是 Detten 写的 Crackme12。第一部分我们将学习 Windows 消息是怎么工作的。第二部分是关于自修改代码，该部分我们也会破解该应用。在第三及最后那部分我们介绍暴力破解。如你所猜测的，在第三部分我们将爆破这个二进制文件。每一部分都会继续前面部分的研究结果。

这个系列的三部分都比较有挑战性，不过我保证如果你花时间并自己动手实践，你会获得逆向领域中的很关键的知识。记住，如果有任何问题，就在论坛里随意发问。我也会在每一章的最后布置作业，让你为下一章做好准备。课后作业是真正学习的地方。

一如既往，你需要的相关文件可以在下载页下载。对于第一部分，下载中包括 crackme，以及一份 Windows 消息备忘单。

那么，事不宜迟，咱们开始吧....

一、Windows 消息简介

本章我们将会讨论 Windows 消息，以及处理它们的过程。几乎所有的程序，除了用 Visual Basic *唉*、.NET 或 java 写的程序以外，任务都是通过使用消息驱动回调过程来完成的。意思就是，与 DOS 时代程序不同，在 Windows 中你只需设置窗口，提供各种你想要显示的设置、位图、菜单项等，然后你再提供一个循环运行到程序结束。这个循环的唯一责任是从 Windows 接收“消息”，然后再将提发送到我们应用的回调函数。这些消息可以是任何东西，从移动鼠标到点击一个按钮，到点击“X”来关闭一个应用。当我们在做一个 Windows 程序时，我们要在 WinMain 过程中编写一个无限循环，以及一个无论消息什么时候到来都可以调用的地址。这个地址就是回调（函数的地址。译者注：这几个字是我补充的）。然后该循环用我们提供的地址将其接收到的消息发送给我们的回调函数，在回调函数中我们决定是否对特定的消息做处理，或只让 Windows 处理它。

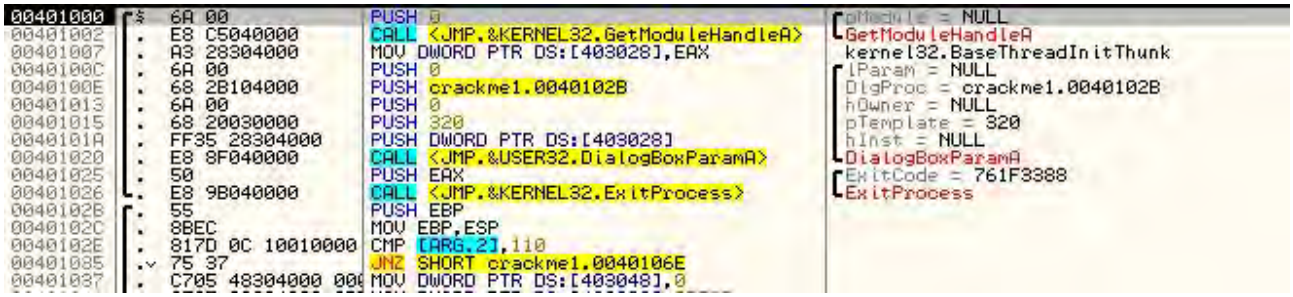
例如，我们想要显示一个带有 OK 按钮的警告消息框。我们只关心 OK 按钮被点击的消息。我们不关心用户是否移动了窗口（WM_MOVE 消息），或者是点击了窗口 OK 按钮以外的某处（WM_MOUSEBUTTONDOWN 消息）。不过当 OK 按钮被点击的消息传来的时候，那就是我们要做些什么的时候了。所有我们不想处理的消息，Windows 会为我们处理。对于我们想要处理的消息，我们只需重写相关消息的处理，做我们想做的。

设置窗口以及包含循环的主过程叫做 WinMain，如果是窗口的的话回调函数通常被叫做 WndProc，如果是对话框的话回调函数通常被叫做 DlgProc，虽然这些名字可以任何其他的东西。

我在下载中包含了一个所有 Windows 消息的指南，你在学习本章时应该打开看看。你可以在教程页下载到所有的文件。你也可以在工具页面下载 windows 消息备忘单。

二、载入应用

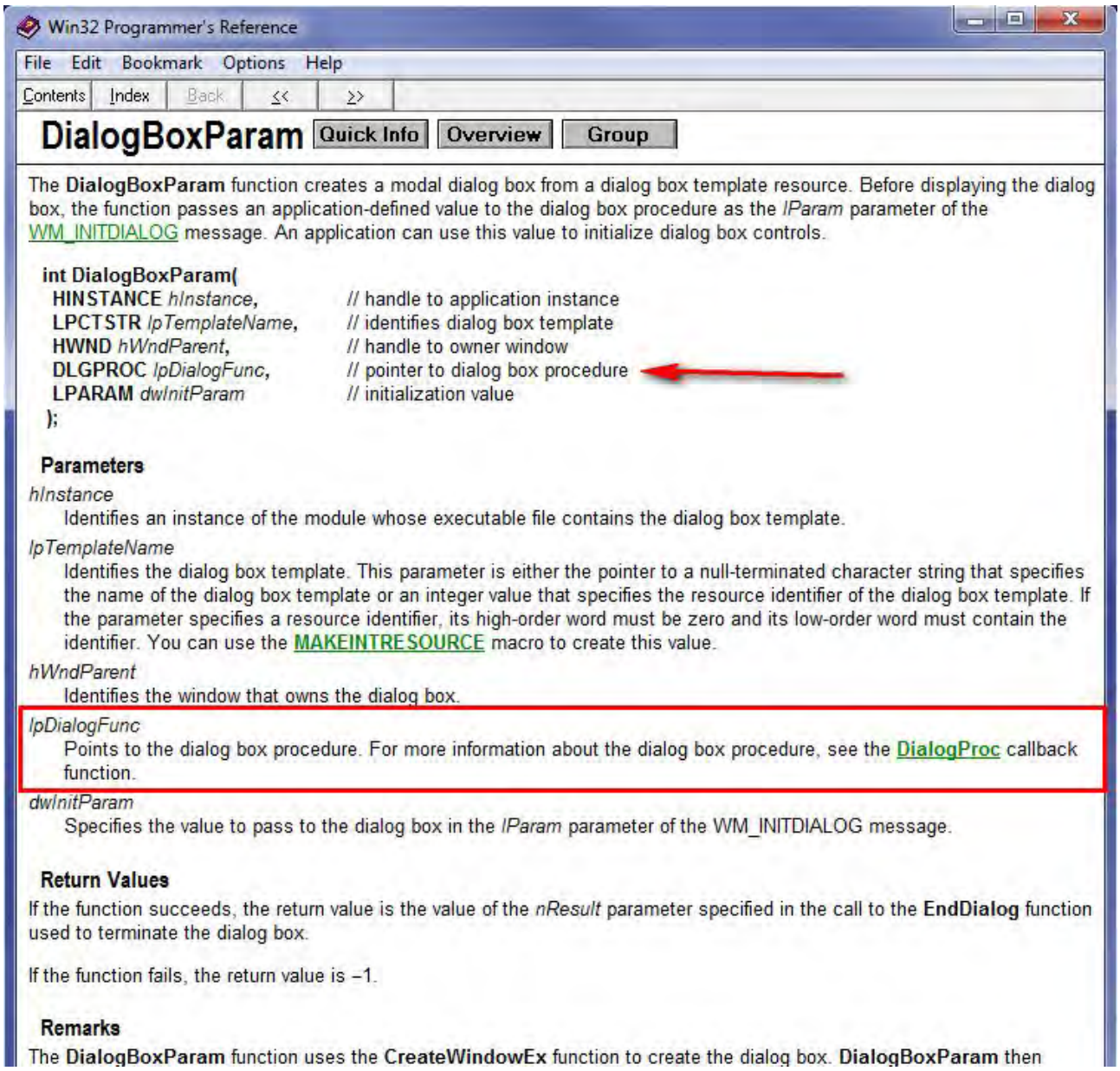
Olly 载入 Crackme12. exe，咱们来看看:



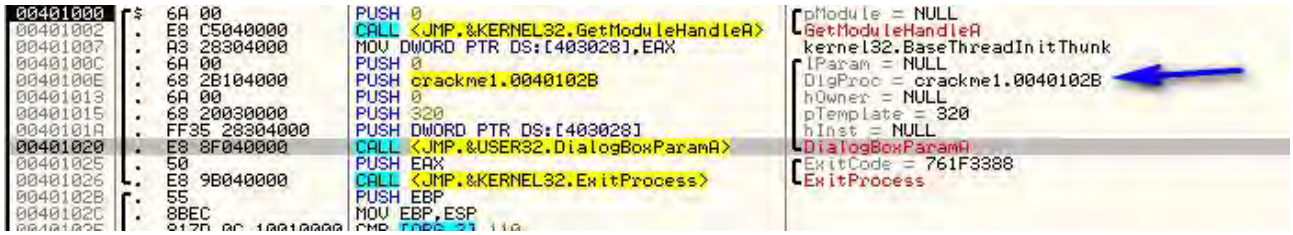
这个一个标准的应用程序，在使用一个对话框作为主窗口，看起来像是用 C 或 C++写的。

****如果程序使用的是常规窗口而不是对话框窗口，它看起来会不一样。参见下面的。****

注意参数是被压入堆栈，以及对 DialogBoxParamA 的调用。这个将对话框设置成程序的主窗口（而不是普通窗口，不过别太在意这些细节，这真的没什么关系）。咱们看看有关 DialogBoxParamA 的帮助怎么说:



对于我们的目的来说，这个 CALL 的最重要的东西是 DLGPROC 的地址。它是我们应用程序的回调函数的地址，用于处理所有的 windows 消息。回头看看反汇编代码，能够清晰的看到这个地址：



这里它是 40102B。咱们过去看看它长啥样。这将是....

三、主对话框消息处理回调函数

这里我们能够看到它的开头：

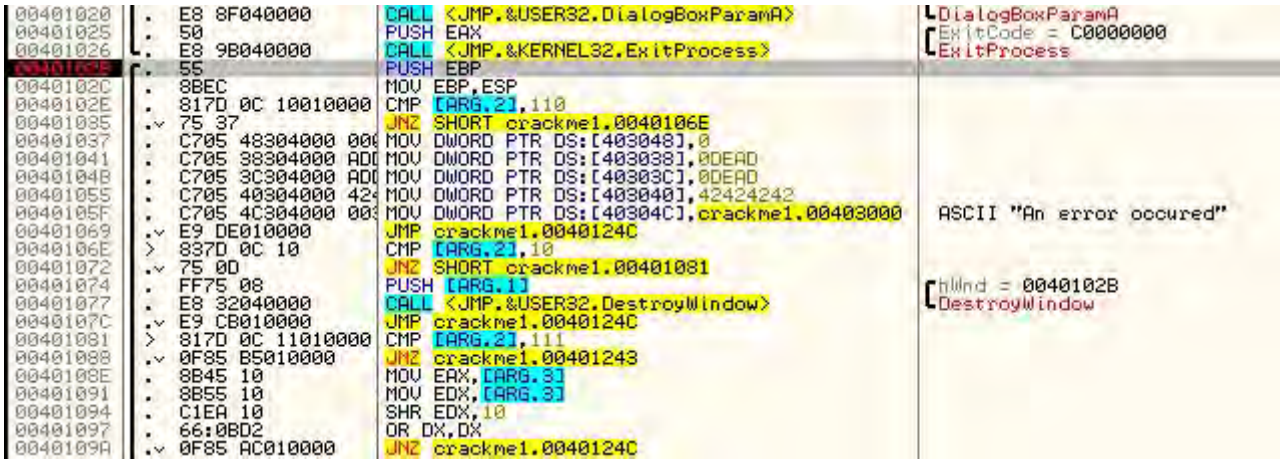
这是一个相当普通的 DlgProc。它通常就是一个真正的大的 switch 语句，虽然在汇编形式下，变成了一个真正的大的 if/then 语句。如果你通读了我的上一章，这个看起来应该比较熟悉，本例中它唯一的不同是，Olly 无法指出 case 标签（ie. Case 113 (WM_TIMER)）。

这个过程在这里有一个原因：是为了响应我们感兴趣的 windows 消息。如果你仔细看的话，你会看到一堆比较和跳转语句。这是将每一部分代码与 Windows 发送过来的消息 ID 进行核对。如果代码匹配了其中一个比较语句，该代码就会运行（译者注：感觉作者这几句怎么那么别扭呢）。否则，它就会尝试所有的比较，没有匹配的话，它就会被发送给 Windows，让 Windows 来处理。

咱们来深入的看看这个过程。继续运行程序：

至少可以说它是一个很奇怪的 crackme。咱们来把玩把玩。你会发现你可以持续点击按钮，不过什么都不会发生，虽然它有一个“clear”按钮。看起来它希望咱们输入一个指定的代码，如果我们不那么做，程序就什么都不做。

现在咱们在 DlgProc 代码的起始处也就是 40102B 设置一个 BP。重启应用，我们能观察到有消息来了：

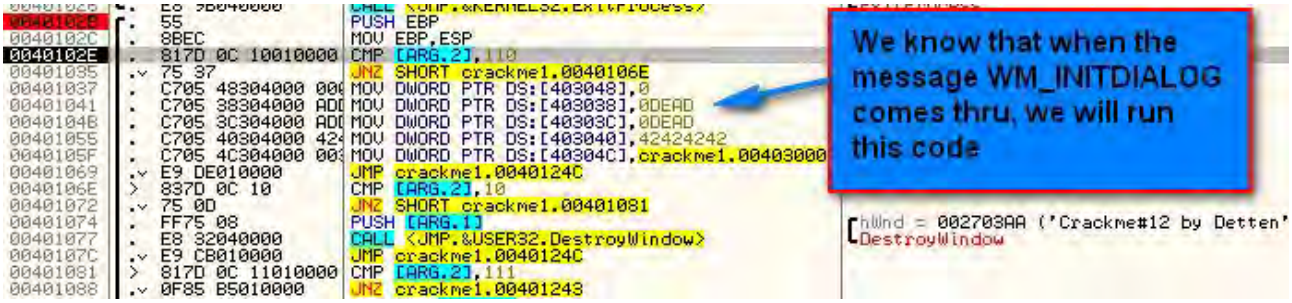


一启动应用，我们立马就断在了那个 BP。你会发现在我们开始进行第一次比较的地方有几条指令

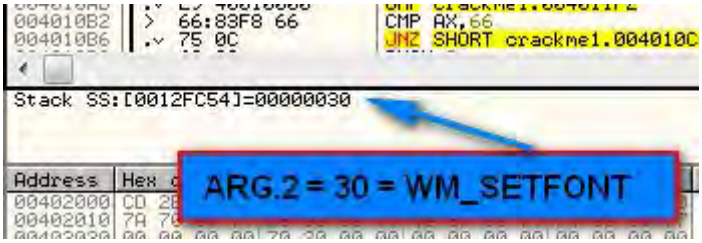
40102E　CMP［ARG. 2]，110

如果你在本章相关下载中的 Windows 消息备忘单中查询 ID 110 的话，就会发现 110 是 InitDialog 的编码：

这个消息给了我们的应用一个机会来初始化一下东西。如果你单步执行，并且消息是 INITDIALOG 的话，我们会直接到 401037 执行相关指令:

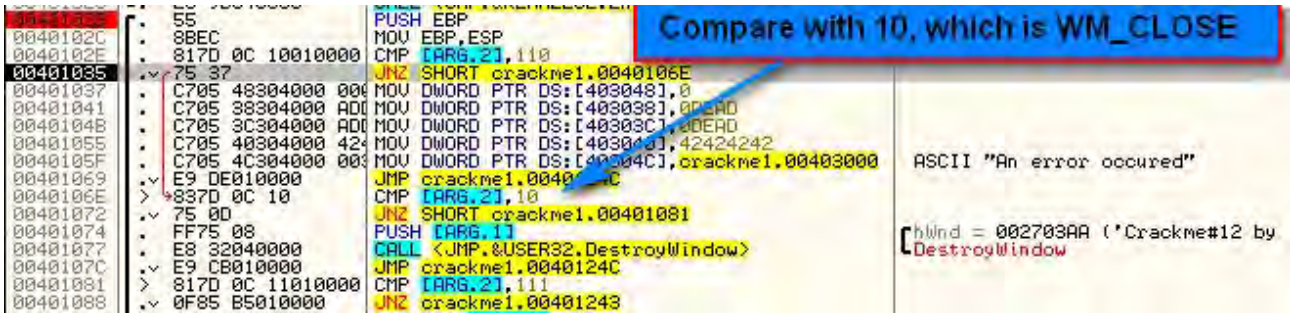


看看下面的信息区，可以看到 ARG. 2 不是 110 而是 30:

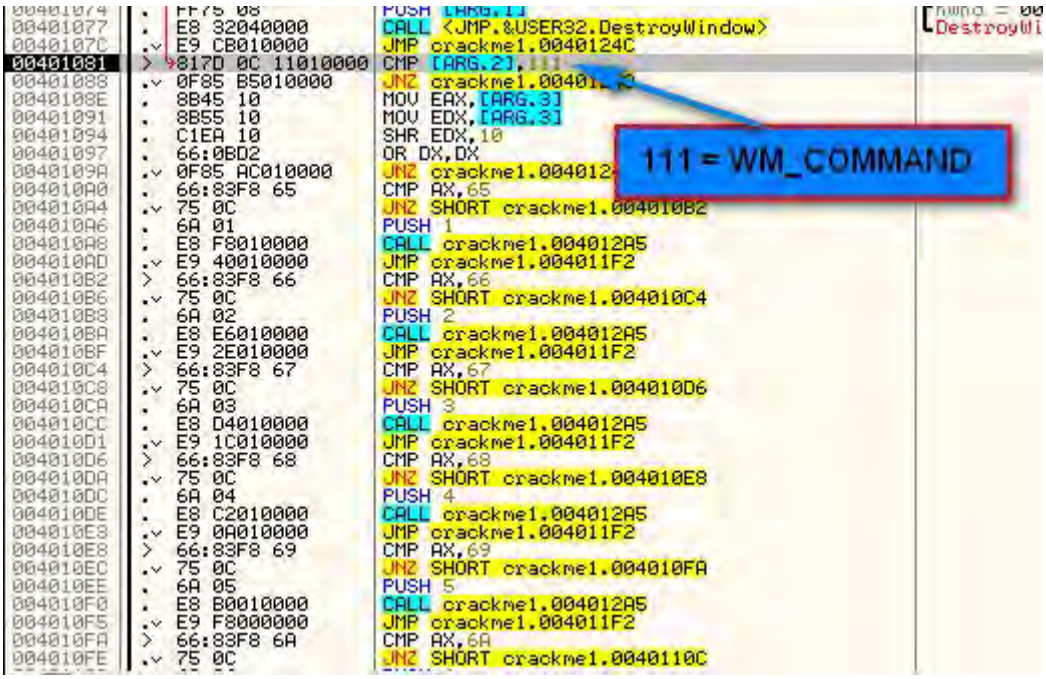


在我们的表中，30 是 set font（设置字体）消息。所以这是 Windows 发送的第一个消息。

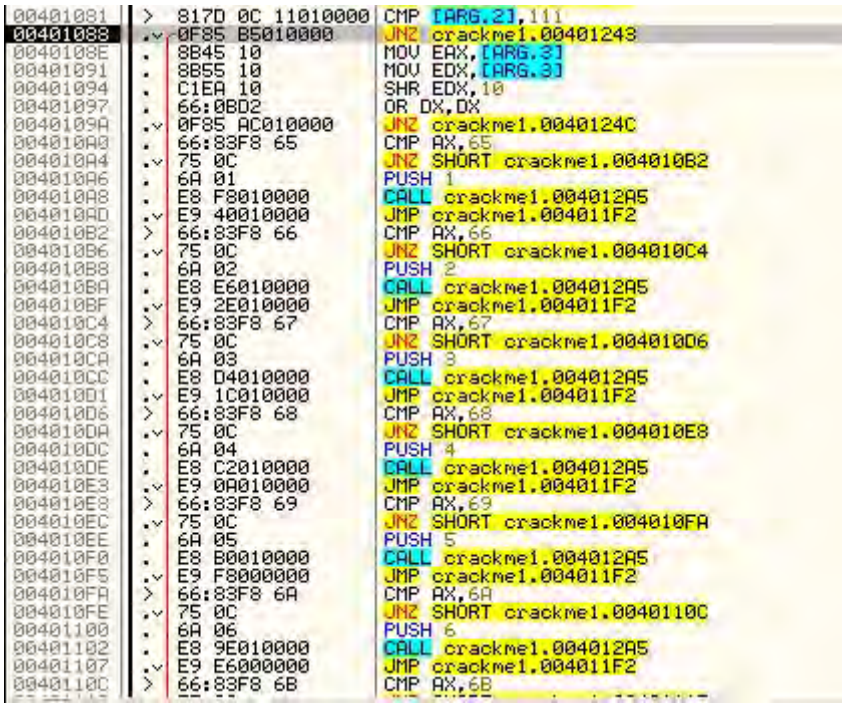
下一个是和 10 进行比较，在咱们的消息列表中是 WM_CLOSE:



所以当关闭按钮被点击时，这段代码就会被执行。下一个要比较的是 111，它是 WM_COMMAND:



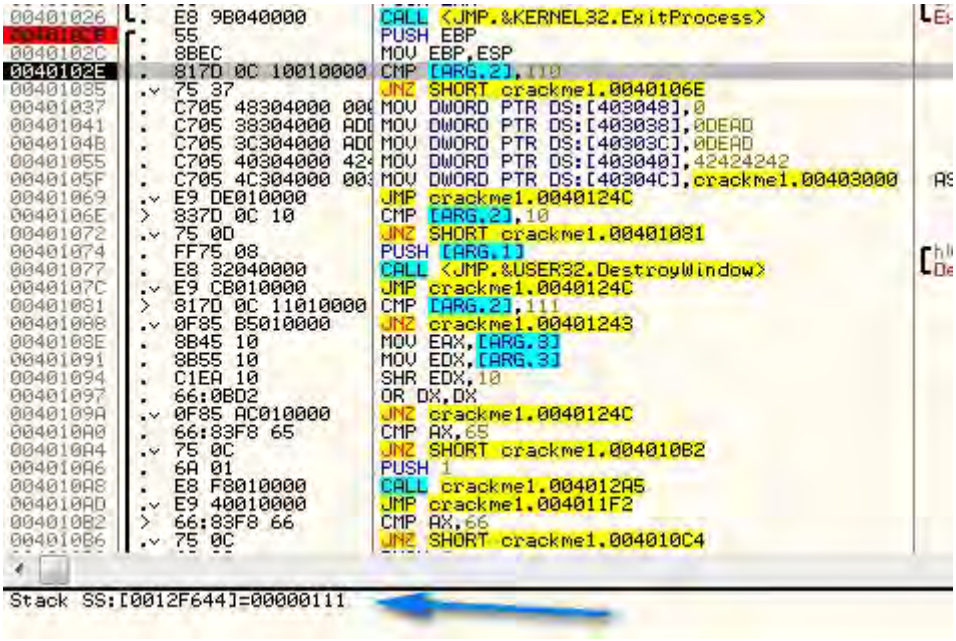
WM_COMMAND 包揽了好几种 Windows 消息，通常与资源相关联，例如点击按钮、选择菜单或点击工具栏中的图标。此外，对于一个 WM_COMMAND 消息来说，在 ARG. 3 中存储了一个整型数据，用来帮助弄清楚命令消息。例如，如果你点击一个按钮，就会传来一个 WM_COMMAND 消息，并且 ARG. 3 中有可能保存有按钮的 ID。如果你正在用一个徒手绘画程序画画，ARG. 3 可能保存有当前鼠标的 X 和 Y 坐标。



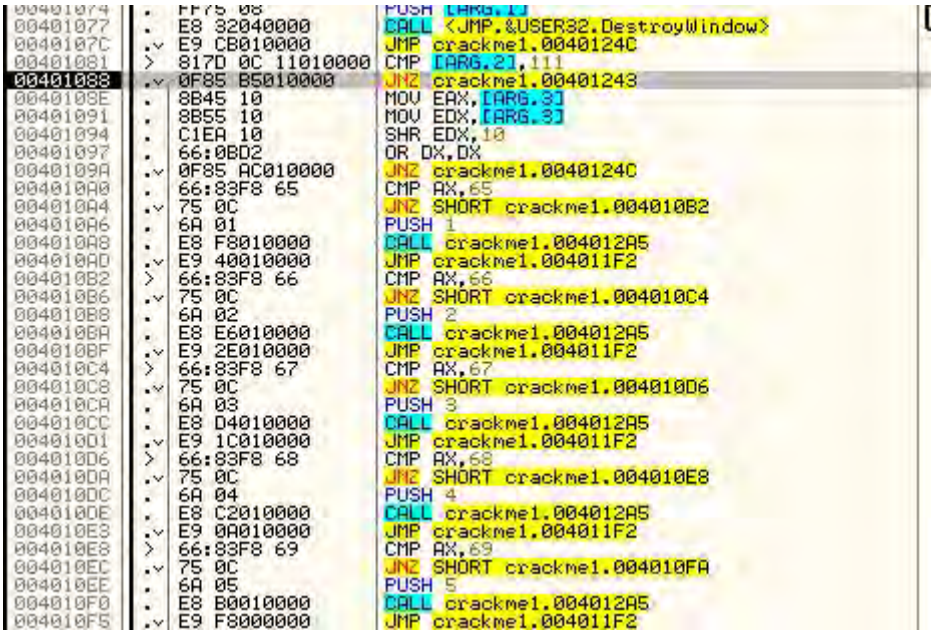
仔细看这个，能够发现过程处理的其他消息只剩 WM_COMMAND 了（真的，每一个 WM_COMMAND 都有可能是一个不同的“类型”）。如果你单步执行，就会发现对于当前的消息 QM_SETFONT，没有与之有关的代码可以执行，

只是在我们的过程的结尾返回了。这是在告知 Windows，我们希望 Windows 来处理这个消息，而不是由我们来处理:

再次点击运行，我们会断在下一个消息:

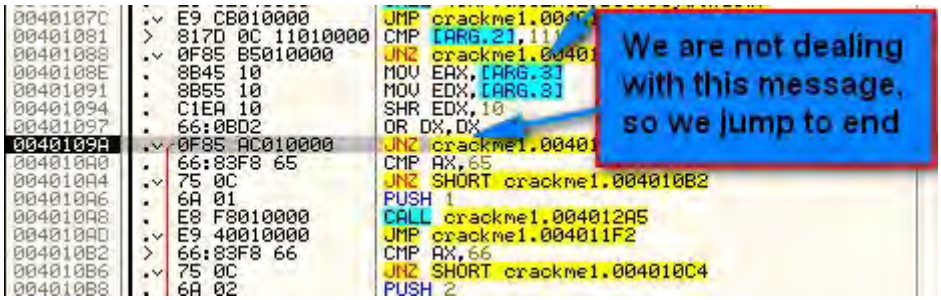


这回它是一个 WM_COMMAND 消息。向下单步执行到 401081 检测该消息的比较指令处，咱们再仔细看看 WM_COMMAND 的处理程序:



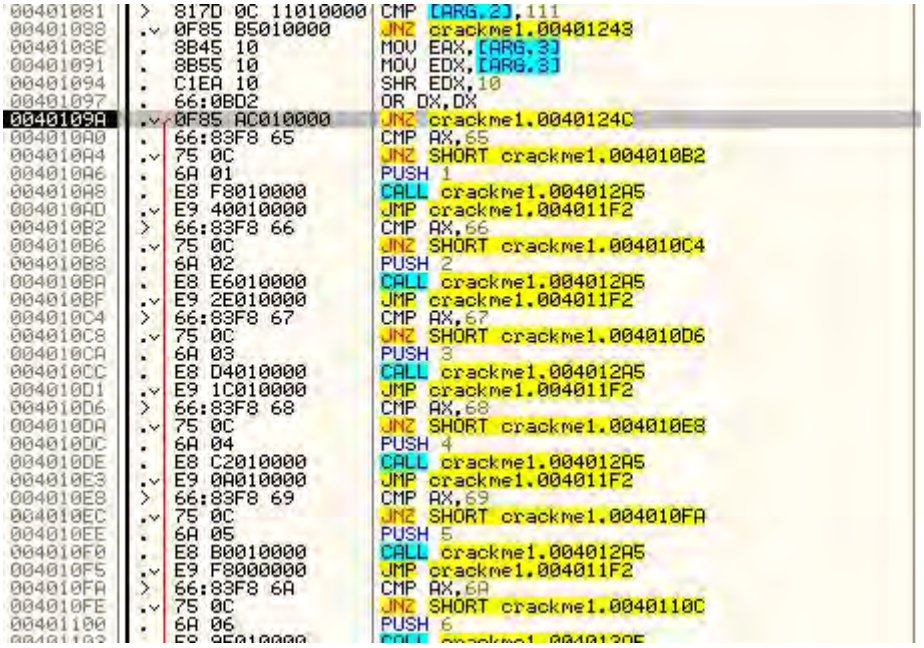
注意它将 ARG. 3 拷贝到 EAX 和 EDX。然后它对 EDX 完成了 16 位 SHR（右移位）操作。然后对该值做 OR 操作，如果它不是 0，就跳转。基本上这是

在检测参数的两个高位字节是否是 0（你正在读汇编语言的书，不是吗？）。EDX 的低位两字节保存了被影响到的资源 ID。本例中，它不是 0，所以我们跳过剩下的代码，然后从回调函数返回：



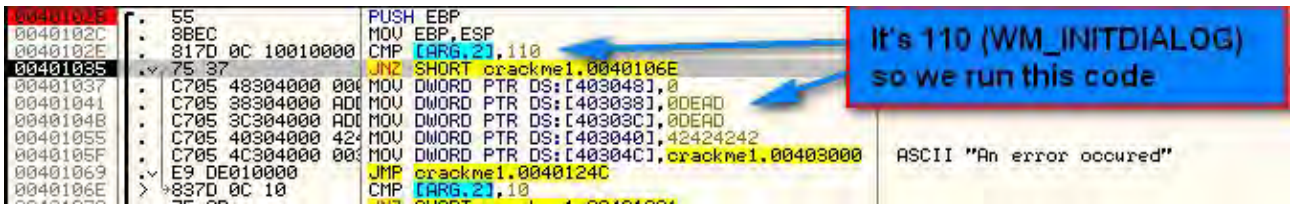
这里我们能看到正在处理的是 111，或者叫 WM_COMMAND 消息：

这里我们能看到那个跳转：

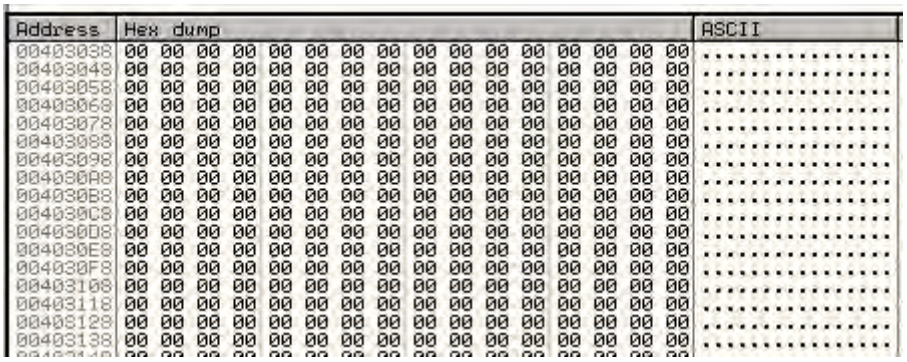


再次运行程序，我们有停在了我们设置的 BP。这回我们处理的是 WM_INITDIALOG 消息：

咱们运行对话框初始化部分的前面几行代码：

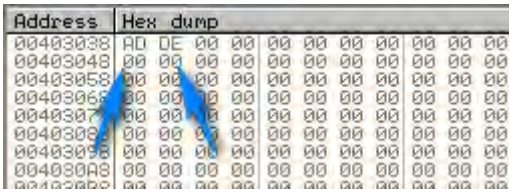


在这个特殊的 crackme 中，刚好这部分代码比较重要。咱们看到有几个整型被存进以 403038 为起始位置的内存中（颠倒顺序进行访问，403038 是最低位）。咱们先在数据窗口中看看:



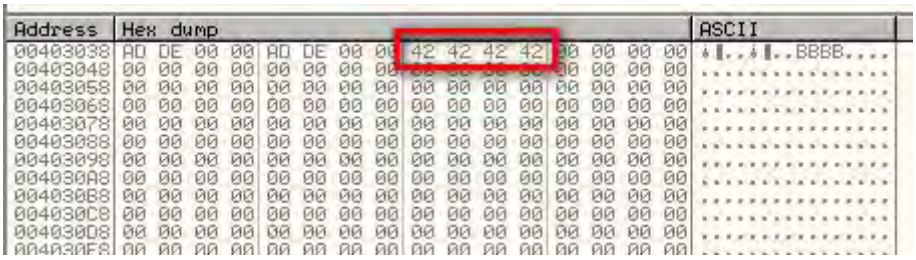
看，在咱们运行这几行前，它被初始化为 0。现在，单步步过第一个 MOV 指令，什么也没发生，不过一个 0 被拷贝到 403038 处。单步步过下一条指令，能够看到产生的效果:

可以看到 0x0DEAD 被拷贝到内存中（以小端序列的形式）:

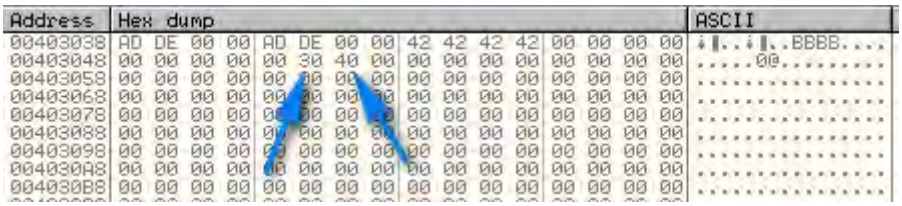


单步步过下一行，它做了同样的事情，不过是在地址 40303C 处:

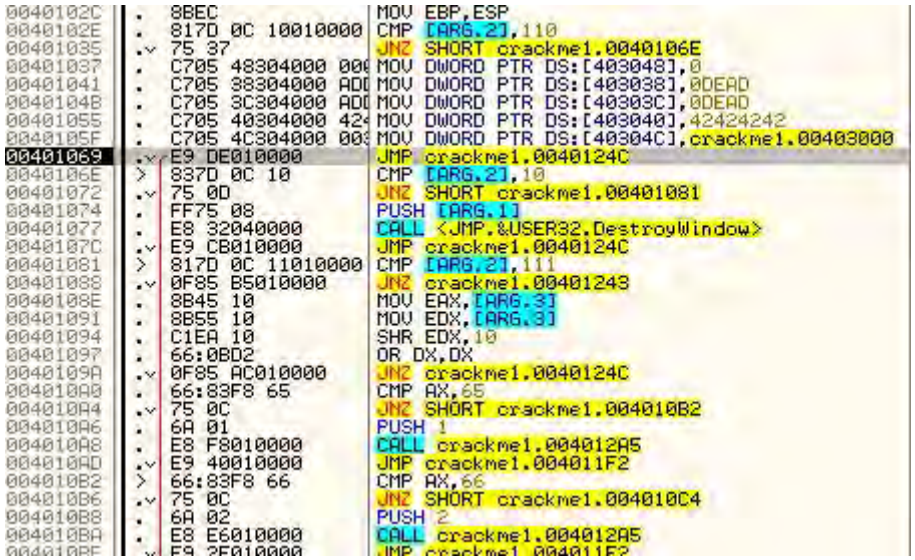
事实是，以十六进制的双字写入彻底的暴露它对该 crackme 是很重要的😃（译者注：为啥？）。接下来，在 403040 处值 42 被写入了 4 次。在 ASCII 数据区可以看到 42 的 ASCII 值是 “B”:



最后，整数 403000 被拷贝到 40304C。Olly 可以分辨出是一个指向以 403000 为起始位置的一段代码或数据（记住是小端序列）:

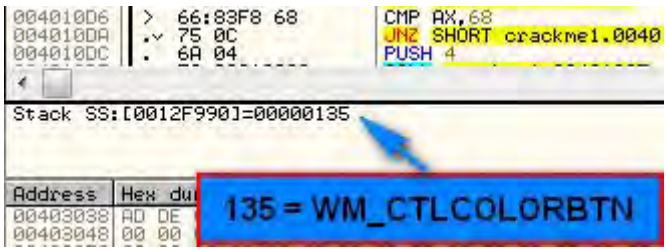


最终我们跳到了结尾并返回，等待发过来的下一个消息:

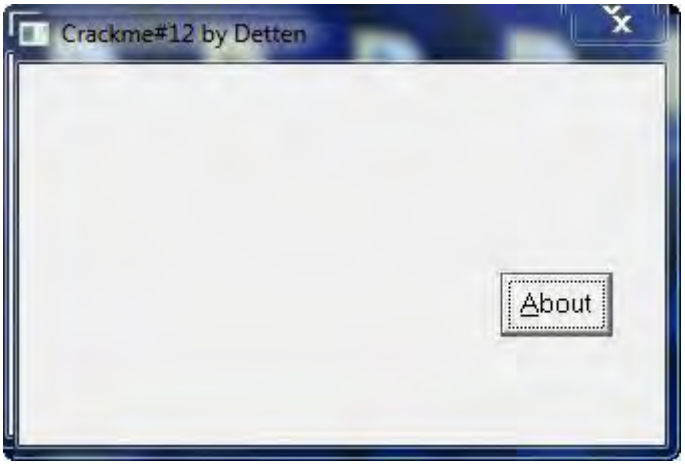


多点 F9 几次（10）你会看到主对话框窗口被创建出来:

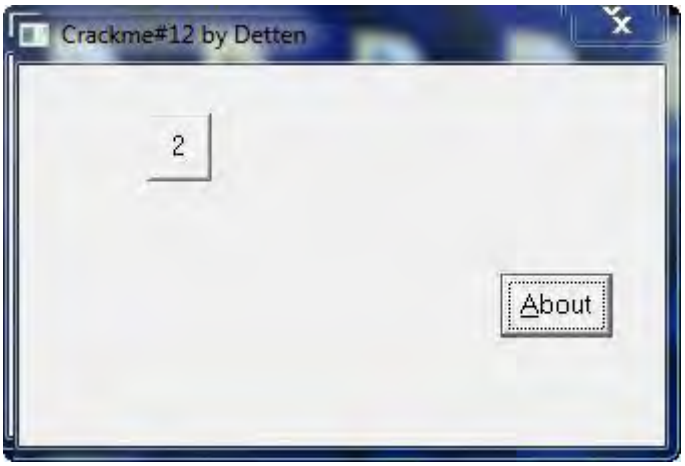
这地方非常的有意思，因为在你点击 F9 时，每点一次在对话框中就会有新的东西出现（大概运行 6 次以后），就会接收到一个消息来在对话框中画资源。下一个消息是 135，或者叫 WM_CTLCOLORBUTTON:



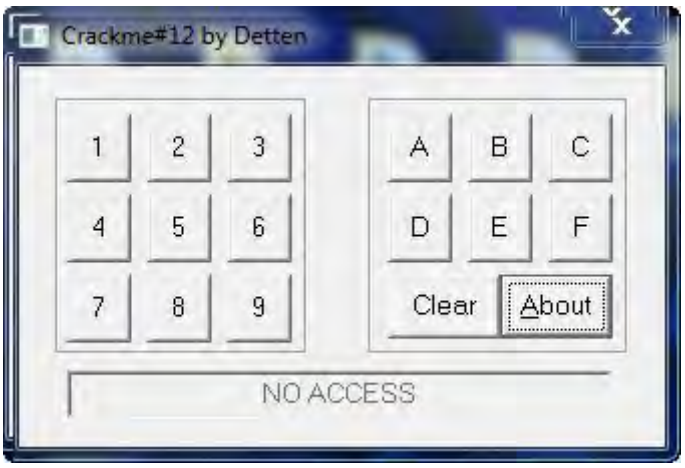
在窗口中画了一个按钮:



下一个是一个写着“2”的按钮:



这时候按 F9，你会真切的看到对话框被构建，一次一个按钮。观察到来的所有消息以及在表中查询它们是相当的有趣。你会看到有很多消息到来。如果有那个你不知道的，就 Google 搜索它，然后就会得到关于它的相关描述。直至最后，底部的 label 控件会被绘制出来，“No access”文本会被写入进去。整个窗口就快完成了。在窗口彻底完成前，我还得按 F9 大概 35 次:



那么现在你知道了一个对话框是如何构建的。设置对话框的基本设置，标题以及整体外观，传进一个回调函数的指针（地址），用来处理所有从 Windows 发送过来的消息。然后 Windows 发送一系列的消息，一次一个的发给回调函数，给我们机会让我们在我们收到所渴望的消息时运行代码。在对话框被构建完成后，Windows 就进入一个内部循环，就坐在那等我们干些什么。只要我们一有动作，一个带有已经发生的动作的 ID 的消息就会被发送给回调函数。然后我们就可以决定对该消息做些什么或者忽略它，让 Windows 来处理它。

你需要注意的最后一件事是，如果该应用是在 Olly 中运行的，只要在窗口上移动鼠标就会导致 Olly 暂停在处理新消息的消息处理过程的开始处。Windows 告诉我们的消息处理过程有鼠标在窗口上移过。基本上，你对对话框做的任何动作都会发送消息给消息处理过程。

四、作业

1、你能不能找出点击一个按钮后会发生什么，尤其是以 403038 为起始位置的内存内容方面。不同的按钮做的不一样吗？你能够理解代码正在修改这些内存位置了吗？

2、猜一猜密码有多长。

第十六章（中）：何谓自修改

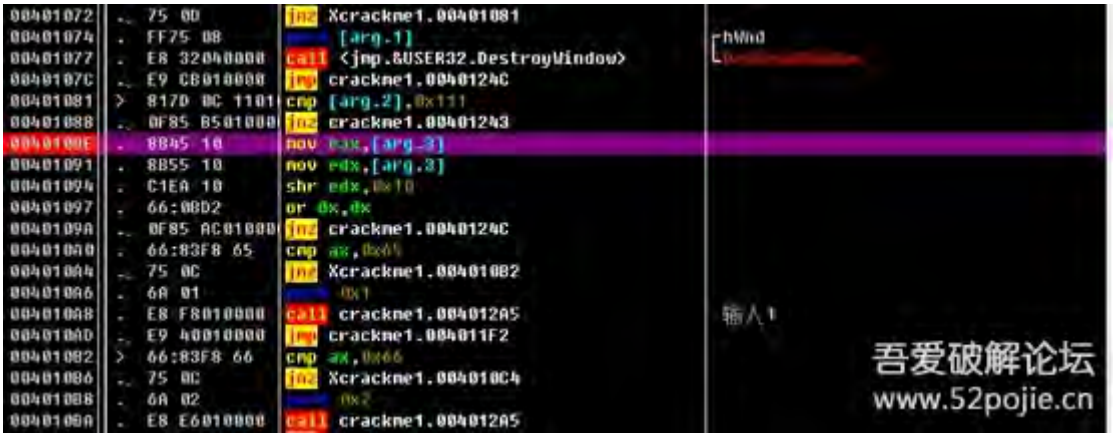
发起帖子的原因是YYSniper没有继续更新后面的文章，我只是按照我看完原版后的理解进行翻译，如果翻译的不对请见谅！并且我会进行程序分析后给出下面的理解。

致谢 YYSniper 提供开始的翻译教程。

转载请注明来自吾爱破解论坛@52PoJie.cn

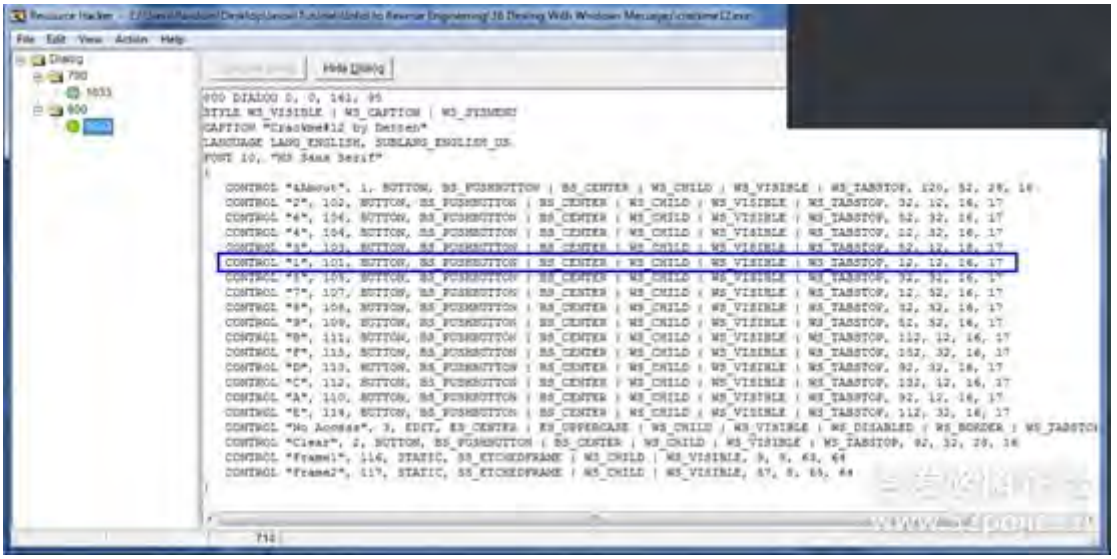
开始正文了：

现在，我们已经看到了基本的消息处理程序回调函数，让我们看看是否可以使用这个来破解这个crackme。我们可以看到，只有三个消息，这个应用程序处理；110（INITDIALOG），10（destroy_window），和111（WM_COMMAN）。任何其他消息都被忽略。我们已经经历了初始化对话框的代码，我们真的不在乎销毁窗口的代码，当我们关闭应用程序时才被调用。因此，任何值得注意的事情发生在wm_command节。所以让我们只停留在那一段Olly。删除任何旧的断点然后设置一个新的在地址40108e，或经过ID 111比较/跳：

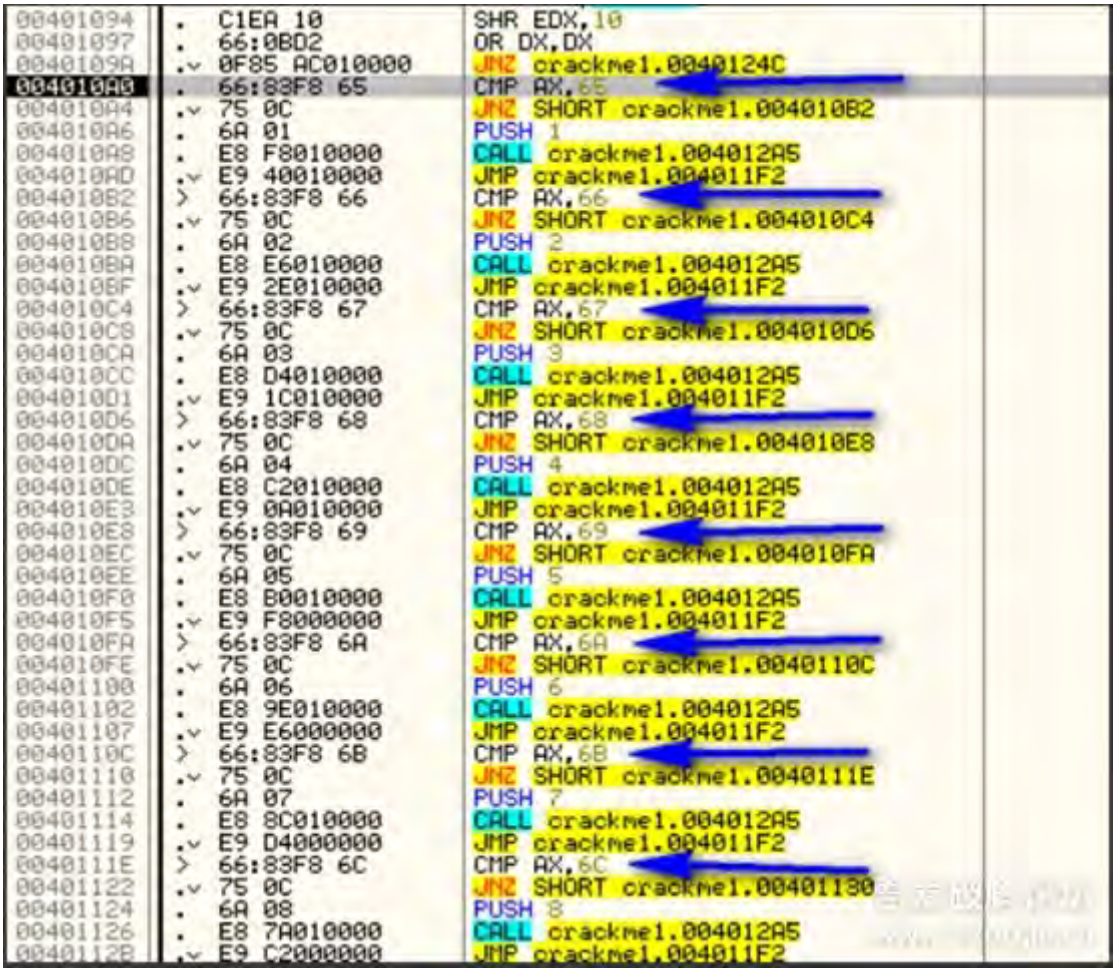


然后F9跑起来，这时打开程序，移动窗口点击窗口都不会被断下，因为这些消息已经被忽略，交给Windows处理了。好了，按下窗口中的“1”按钮，这时OD断在了我们设置的断点上，现在我们看看arg.3的值为0x65

如果我们用Resource Hacker打开Crackme，然后打开主窗口，我们会发现0x65对应的10进制为101，在Resource Hacker中显示的ID就是“1”按钮。



这个值进入到arg.3中，他只是这个按钮的ID，因此我们向下看两行会发现一个比较出现了，和这个ID进行比较，然后继续往下看，就会明白这里的比较和跳转就是与程序的按钮ID进行比较的



不管你输入的是1或者2，都会进入到同一个Call中，那么我们到这个Call去看看到底处理了什么。

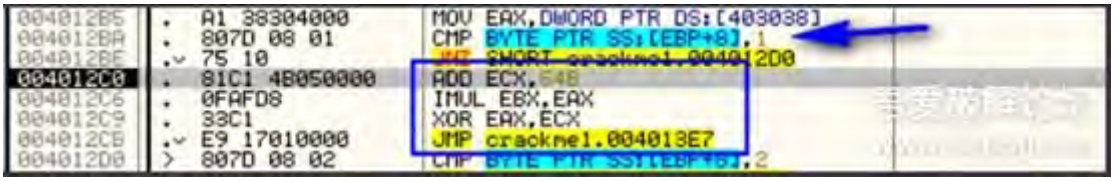
```
004012A5  r$  55          push ebp
004012A6  .   8BEC        mov ebp,esp
004012A8  .   60          pushad
004012A9  .   8B0D 4030400 mov ecx,dword ptr ds:[0x403040]
004012AF  .   8B1D 3C30400 mov ebx,dword ptr ds:[0x40303C]
004012B5  .   A1 38304000 mov eax,dword ptr ds:[0x403038]
004012BA  .   807D 08 01  cmp byte ptr ss:[ebp+0x8],0x1
004012BE  .v  75 10       jnz Xcrackme1.004012D0
004012C0  .   81C1 4B05000 add ecx,0x54B
004012C6  .   0FAFD8      imul ebx,eax
004012C9  .   33C1        xor eax,ecx
004012CB  .v  E9 17010000 jmp crackme1.004013E7
004012D0  >   807D 08 02  cmp byte ptr ss:[ebp+0x8],0x2
004012D4  .v  75 12       jnz Xcrackme1.004012E8
004012D6  .   81E9 3302000 sub ecx,0x233
004012DC  .   6BDB 14     imul ebx,ebx,0x14
004012DF  .   03C8        add ecx,eax
004012E1  .   23D8        and ebx,eax
004012E3  .v  E9 FF000000 jmp crackme1.004013E7
004012E8  >   807D 08 03  cmp byte ptr ss:[ebp+0x8],0x3
004012EC  .v  75 0F       jnz Xcrackme1.004012FD
004012EE  .   05 82050000 add eax,0x582
004012F3  .   6BC9 16     imul ecx,ecx,0x16
004012F6  .   33D8        xor ebx,eax
004012F8  .v  E9 EA000000 jmp crackme1.004013E7
004012FD  >   807D 08 04  cmp byte ptr ss:[ebp+0x8],0x4
00401301  .v  75 0F       jnz Xcrackme1.00401312
00401303  .   23C3        and eax,ebx
00401305  .   81EB 2212110 sub ebx,0x111222
```


现在我们来到了004012A5这个Call里面了，我们现在打开之前在初始化（WM_INITDIALOG）中的内存数据，数据地址为 00403038，然后我们在数据窗口中可以看到

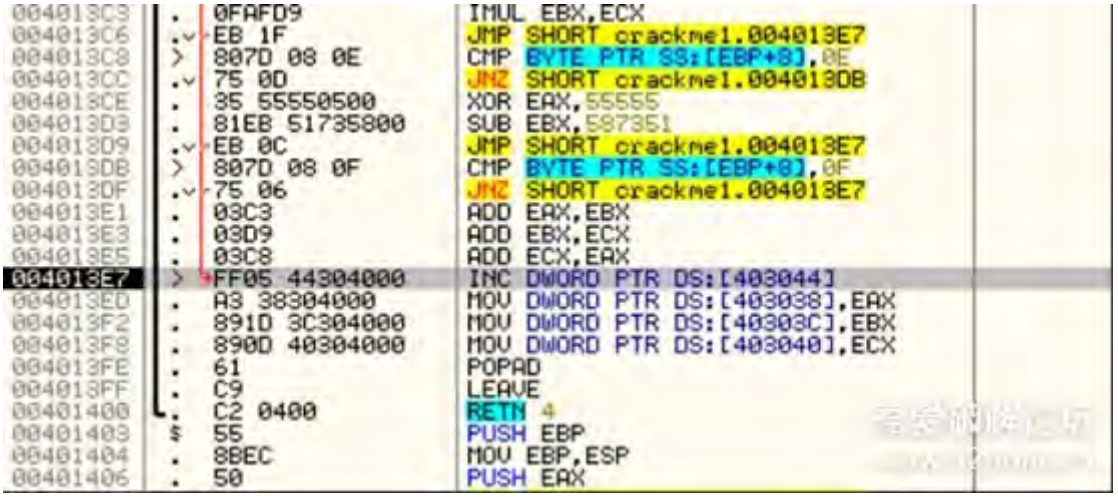
```
地址      HEX 数据                                          ASCII
00403038  AD DE 00 00 AD DE 00 00 42 42 42 42 00 00 00 00  ....BBBB....
00403048  00 00 00 00 00 30 40 00 00 00 00 00 00 00 00 00  .....0@.........
00403058  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403068  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```


地址中显示了初始化的数据：两次“DEAD”，已经4个“42”，代码中我们先把4个“42”移动到ecx，然后把两个“DEAD”分别移动到ebx和eax

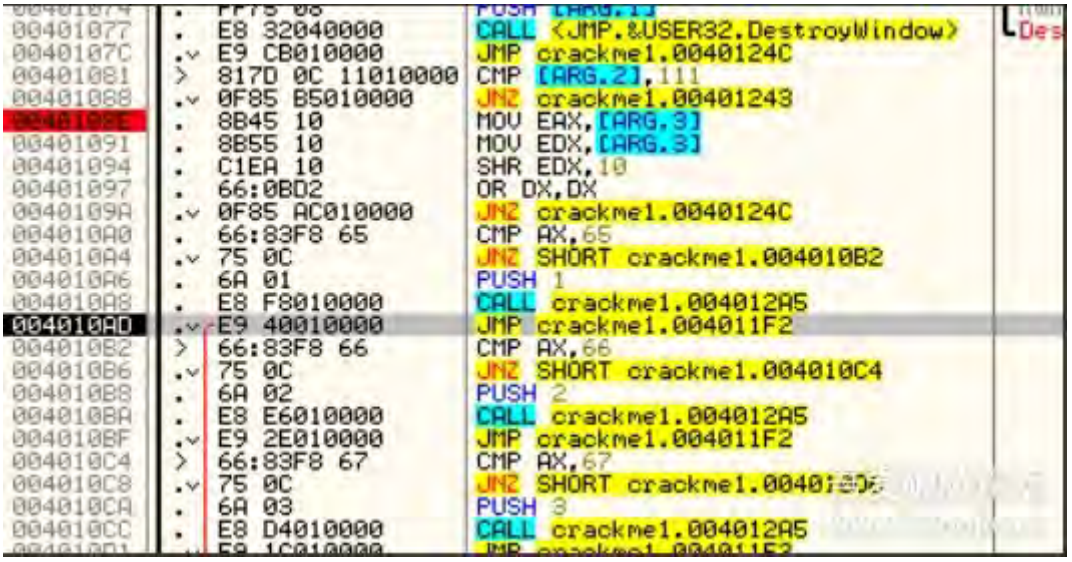
Next, we do a series of compares to findout which button we pushed based on the value that was pushedonto the stack. Here, SS:[EBP+8] is directly accessing this pushed value.Since we clicked the first button,we will perform the first set ofinstructions:接下来，我们做了一系列的比较，找出我们按下按钮后被压入栈的值。在这里，自从我们点击了第一个按钮SS：[EBP+8]直接访问此栈的值.我们将执行第一次操作：



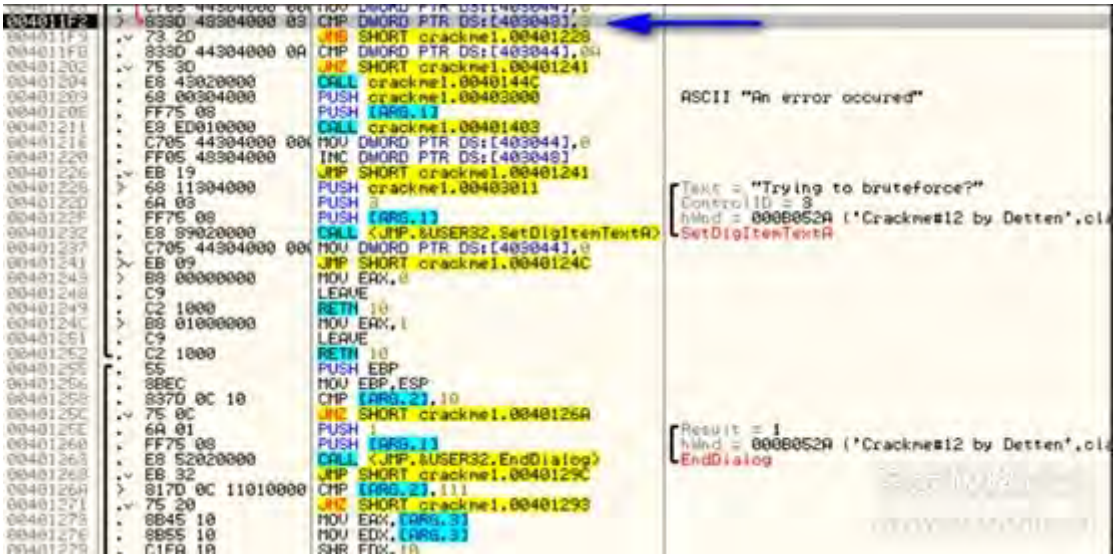
下面执行的操作如下；Ecx加上0x54B 后ecx为4242478D，然后EBX乘以EAX（0xDEAD x0xDEAD）后ebx为C1B080E9。最后我们进行XOR操作Ecx，然后跳转到4013E7.单步不过这个跳转我们来到：



我们来到算法的最后，如果你往回看，你会发现所有按钮按下后都会做一系列加、乘和XOR的处理只是不同的按钮各不相同，然后来到4013e7处。在这里我们把403044加一，把EAX移动到403038，把EBX移动到40303c，把ECX移动到403040，然后出栈返回到主函数中。



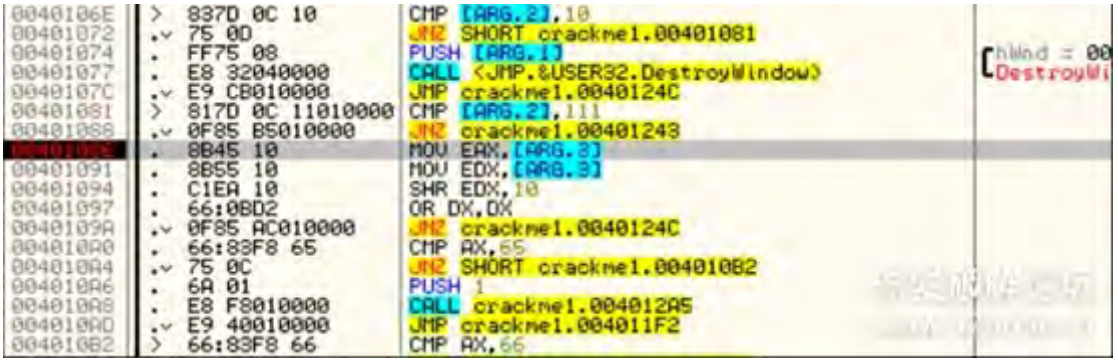
然后一个跳转直接跳到4011F2处：



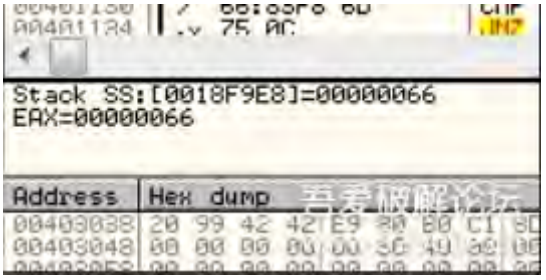
这里把内存地址403048（0）与3进行比较，然后再把内存地址403044和0xA比较，而403044是我们前面的

Call中写入的，我们暂时把这个值猜测为长度，如果403044不是0xA的话，我们会跳走。然后来到暴力破解信息处，同样的，如果403048等于3的话也会来到暴力破解信息处。（译者注：其实这里就已经能够猜到密码为10位，如果连续3次都输错就会出现暴力破解信息）。

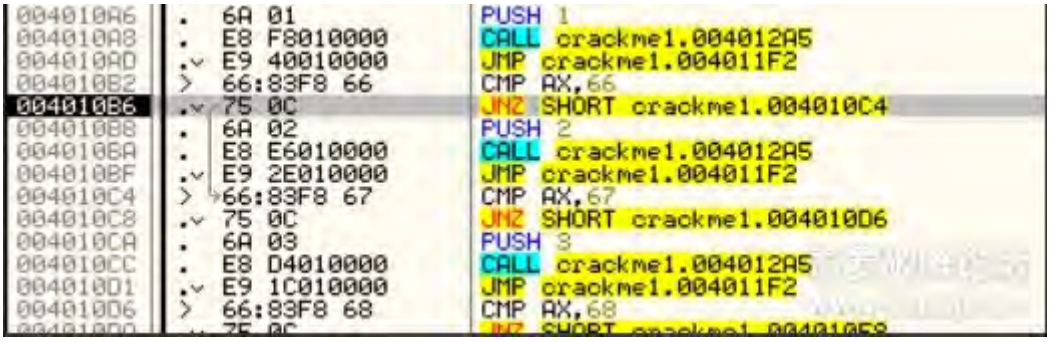
让我们继续按下“2”按钮，来到我们的断点位置：



参数arg.3为66



这与我们之前的推测是一致的

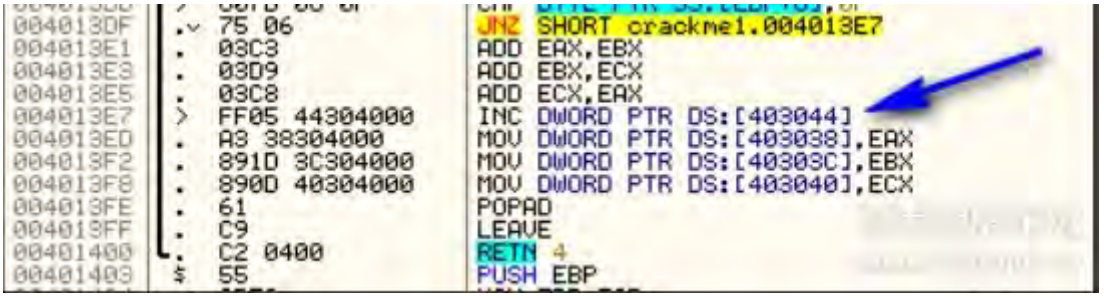


接下来的操作与按下“1”按钮一样来到了004012A5这个Call中，只是原来的“DEAD”和“42424242”不一样了，以为之前“1”进行过计算了。然后又来到

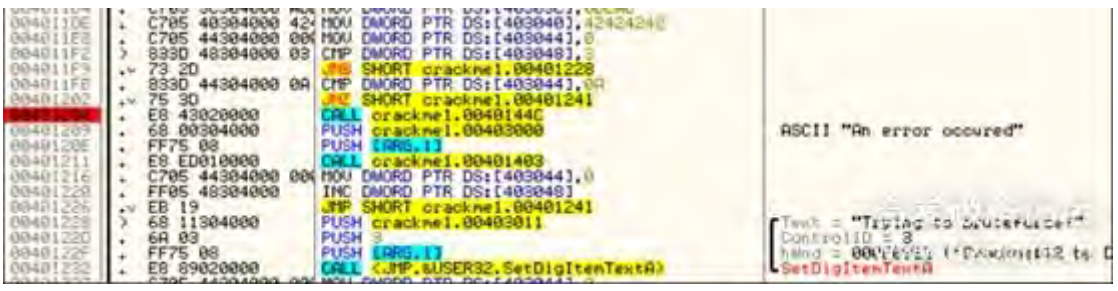
403048还是为0（这里作者问如果等于3了会跳到暴力破解信息中是什么原因，上面我的分析是连续3次都输错密码就会出现暴力破解），而403044已经变成2了，还是不做任何处理进行回到主窗口循环，一定是要我们做什么动作。（译者注：其实我们在分析的时候可以输入10次后按看403044和0xA比较后的代码，看做了什么）

正式破解Crackme

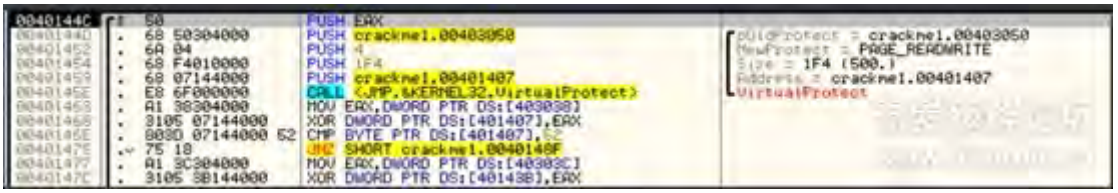
现在我们已经明白这个程序是怎么工作的了，让我们来打补丁，对于这个程序我们需要一点直觉。根据这个程序的流程，我们没有看到更多的比较和跳转跳到正常流程，.真的，我们唯一能看到的都是跳转到暴力破解，地址为4011F9。一个跳转是重4010B2到4011A6是“about”按钮的。 “Clean”的跳转会把DEAD和42424242重设到初始化内存中，在地址4011CA处，如果点击”about“然后跟踪代码，你会发现只是打开about窗口，然后回到主函数循环中。“clean”也是做同样的动作，因此还是会跳到暴力破解信息处。现在这里有一个小的直觉进来。每一次我们检查了地址403048，看看我们是否应该跳转到蛮力消息，内容是零和跳转从来没有采取。然而，在地址4011fb处比较403044地址值达到0x0a会跳转。我们也知道，每一次通过循环，403044的值的增加，所以我们可以假设这个计数器记录多少我们按下按钮：（译者注：其实这里我们之前就已经分析了）



当然，你的第一个想法是'是的'，但是代码导致错误信息！“。是这样吗？所有的代码是否加载指向一个消息的指针，该消息说有一个错误，但这显示吗？在代码中没有显示……或许根本不是这样的。这段代码看起来很可疑，所以让我们分析它。我们知道需要输入10次后才能会进入到错误信息。所以让我们在地址401204，设置断点然后删除之前的所有断点并重新启动应用程序（当然我们也可以数10次后断下^_^）：



好了，现在已经点击了10次按钮，并且断在了我们的断点处，让我们进去看看做了什么？



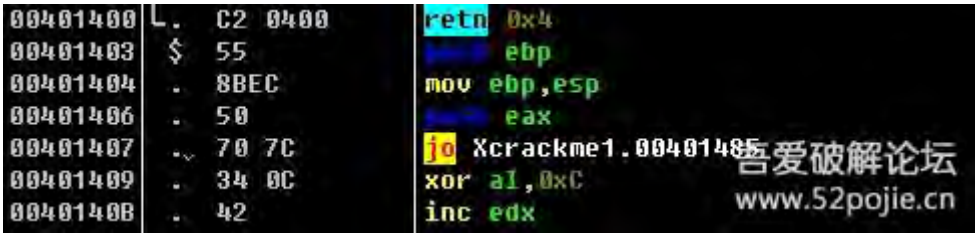
嗯，这里会先Call VirtualProtect，在读取完成VirtualProtect后基本可以确认是用来修改内存数据的. 举个例子，如果要修改一段内存，就要给这段内存赋予可写属性，使用VirtualProtect后就可以给这段内存赋予写属性。现在你可以去修改这段内存了，实际上是在这个领空上进行修改。在这段代码中是自修改代码把内存数据进行相关计算后写入内存然后call VirtralProtect后恢复原来的属性。看来，这个应用程序正在做类似的事情。对virtualprotect的最后一个参数是你想要改变属性的内存位置，和第三个参数是在你想改变的部分的字节长度。在这种情况下，我们可以看到，起始地址为401407，长度为0x1f4（500）。我们也可以看到，第二参数page_readwrite，使这段可写、可读。让我们看看这段地址，在401407开始，看看会发生什么变化

Hmmmmm. That looks really suspicious. Itdoesn’t look like code at all. Let’s keep going and see what theapp changes in this section of memory. Stepjust past the call to VirtualProtect:嗯，这一段确实有惊喜，看上去不像常规代码，让我们继续看看程序是怎么改变内存的。单步步过Call VirtualProtect：

现在，第一件事情是把403038的值mov到eax，然后与401407的值进行XOR，然后把结果存放到401407的地址中。等等，401407是我们上面设置了可写属性的，所以我们可以进行写操作，然而403038是重DEAD通过按下不同的按钮后进行不同的计算后的值。因此，这一系列的指令是改变，基于什么按钮，并在其中的顺序，

他们被按下的内存空间等等。走到地址为401475的JNZ跳转然后让我们看看地址401407：

（译者注：我的OD里面显示的和他不一样，我的是jo，而作者的是JECXZ，我百度了下解释为：JECXZ(ECX为 0 则跳转);jo则是溢出跳转



跳转相关的标志位：

11	10	9	8	7	6	5	4	3	2	1	0
OF	DF	IF	TF	SF	ZF		AF		PF		CF
溢出				符号	零	未用	辅助		奇偶		进位

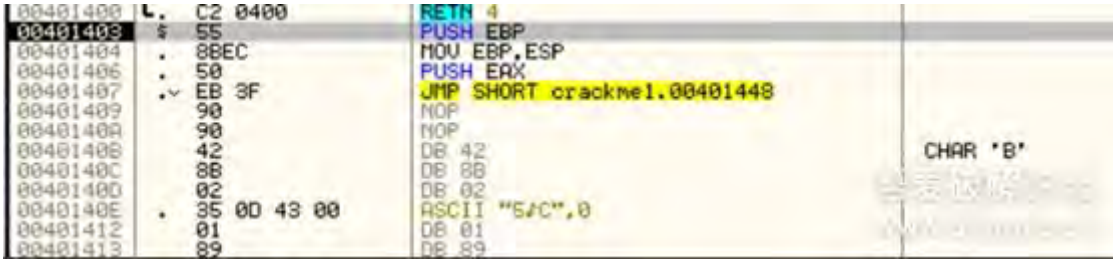
）你可以看到401407的内存数据已经被改变了，但是这里有个没见过的指令：JECXZ SHORT Crackme1.004013E5.程序加了一个条件跳转到自己的代码！（译者注：没明白什么意思）这方法是在内存位置修改操作码或者裸数据。现在回到我们的指令位置，下一个是用401407和0x52进行比较，如果不等则跳转到40148F。看上面的图片，我们可以看到401407处操作码是“E3”是不等于0x52的，因此我们跳走。这个跳转就是恢复之前VirtualProtect，把之前可写的内存恢复到只能执行不能写的状态：

but before this you may have noticed thatmemory location 401407 was XOR’ed again at address 40148F.

Looking again ataddress 401407 we see that it was changed again:在此之前你应该注意到了，内存401407和eax（40148F）做了Xor操作.再去看看401407你会发现数据有被修改了。

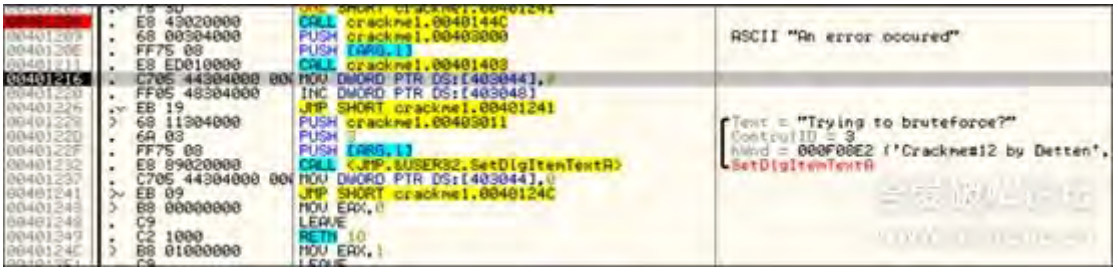
现在这里的JECXZ已经被JMP跳转替代了。事实上，程序改变这个内存两次，一次是JECXZ，另一次则是JMP（译者注：这里我把JECXZ替换为jo）。然后我们回到主窗口循环中：

我们放了一个值（F08E2（译者注：这个值与你按下的按键不同会不同，我的与作者的就不一样））到栈然后Call另外一个常规地址401403.步进这个Call我们看见了这个功能：



现在我们来到了程序修改内存的地方。我们之前就知道401407这里的这个跳转了，让我来到JMP看看会把我们带到那里去。

奇怪，他直接跳到了返回。所以看来这并没有真正做什么。我们现在回到主程序：

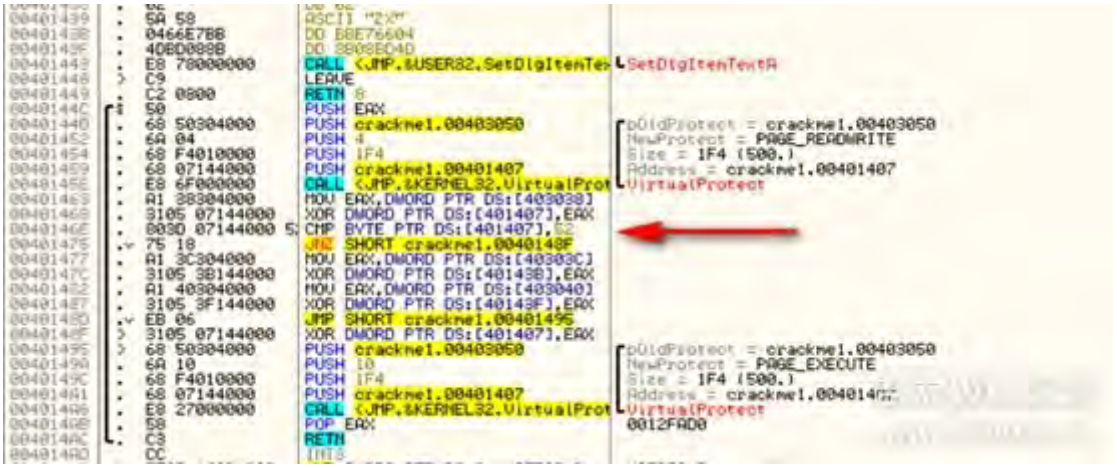


接下来是把计数器置零。然后把另一个计数器加1.现在我们明白暴力破解信息是怎么检查的了：如果我们输入10个密码错误超过3次，403048计数器就大于3就跳到暴力破解信息。如果想再试请继续。保存401204的断点存在，点击10个按钮：

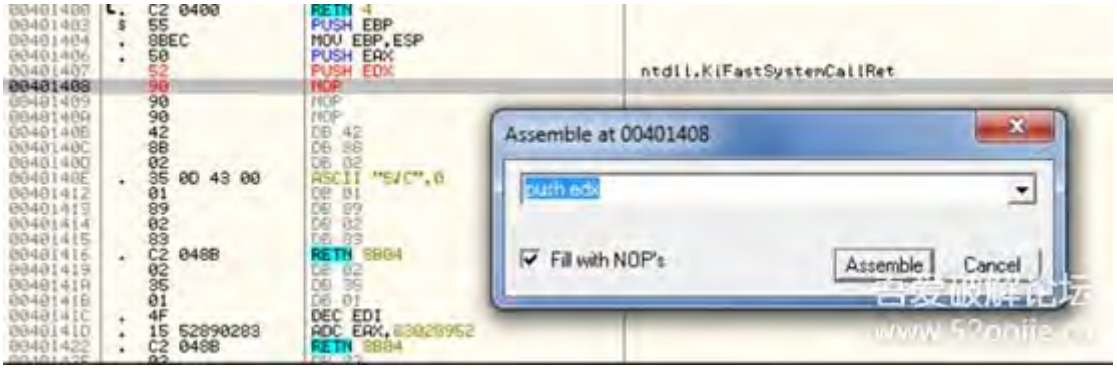
出现了暴力破解的信息。现在，确保401204的断点存在然后重启程序。以确保计数器被清零。

因此我们知道程序更多的信息：1、密码为10位数。（译者注：早前猜测）2) 如果连续三次错误，则会出现暴力破解信息，出发重新启动程序。3) 每一次按下按钮403038, 40303C和 403040内存中的数据都会被不同的

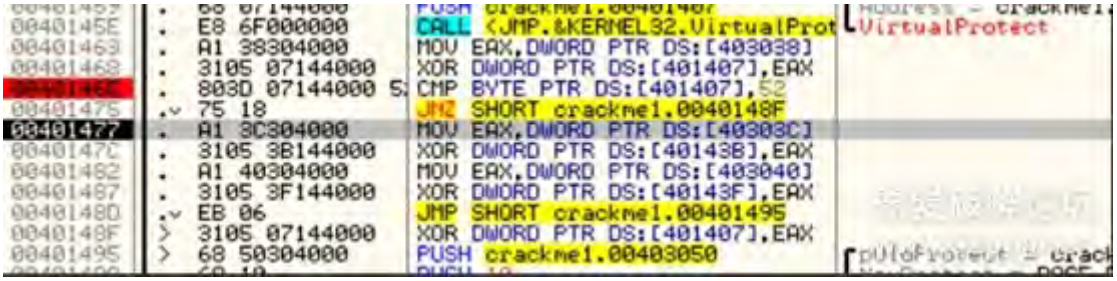
算法修改。每一个按钮的算法都不一样。4) 直到我们按了10次按钮后来到一个Call，进入后发现它会检查代码和改变跳转指令，在地址401407处。5) 如果密码不对，这个跳转就会生成并且就返回到我们的主窗口循环中。6)因此，输入正确的密码，必须改变这跳转到别的东西，或者跳到不同的内存位置，来到我们想去的地方，或者在这个内存的这个段修改代码以创造好消息，而不是创建一个跳转。这听起来似乎更合理，因为它只是简单地变成了一个跳转到一个新的位置，这个奇怪的，非功能性的代码是什么？知道了这一切，我们知道我们必须要在程序自修改前清空这个段，即从地址40144c。让我们再来看看这段



我们可以确认一件事就是在40146e处与0x52的比较是重要的。它基本上告诉我们，程序已作出的代码的变化是正确的变化。但是0x52操作码是什么意思呢？好吧我们百度下（作者使用Google），我们发现操作码0x52是“PUSH EDX”，如果不是呢,这就是Bug。如果我们强制设置这个指令为“PUSH EDX”看看会发生什么？试试看吧，在40146E处设置断点。然后来到401407处，并且修改操作码为0x52：

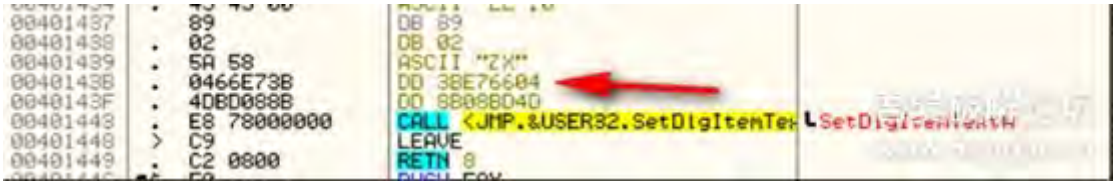


现在，单步走，我们能通过这个跳转

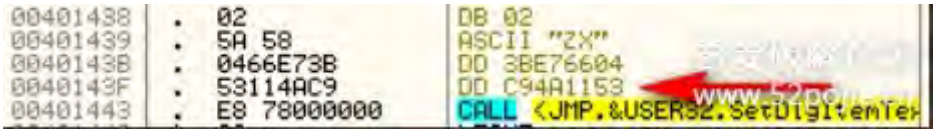


现在把40303C的移动到eax，然后把40143B与eax进行Xor，这时什么地址？我们看看

就像我们看到的一样，这就是我们被跳过的一个内存段的尾部。Xor后我们来到：

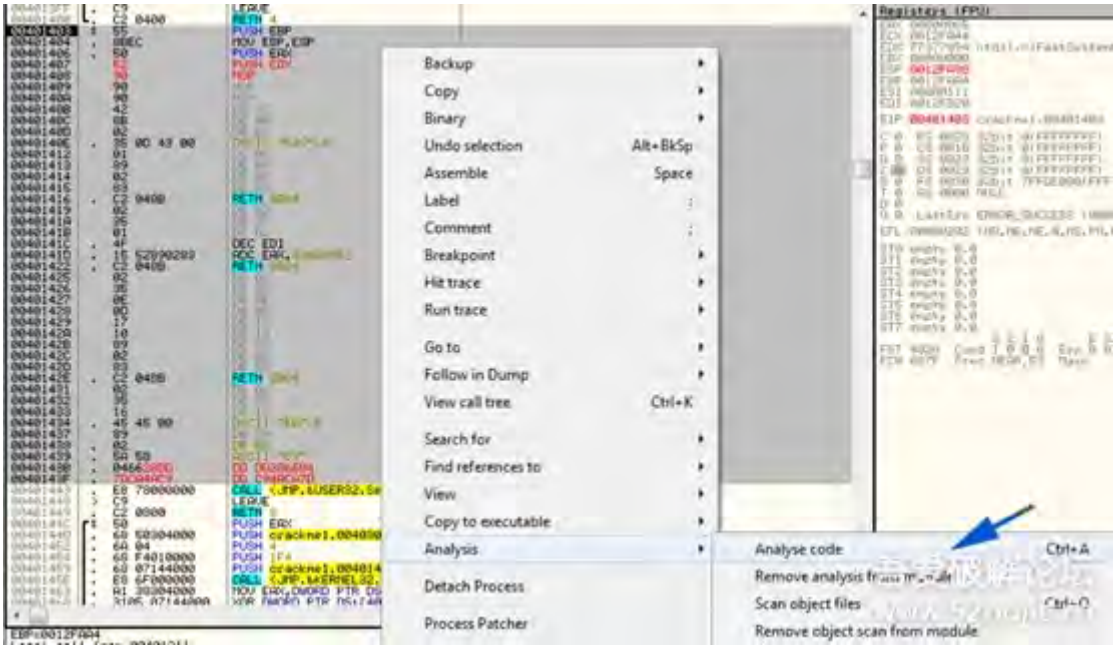


我们改变了40143B的值，下一步又修改了40143F的值。



这些位置并没有被改变成正确的代码，所以它并没有真正帮助我们，但看到这是应用程序改变的最后一件事，这是重要的。现在继续走，我们已经改了 PUSH EDX了，让我们看看这个内存段做了什么.返回主窗口循环后，来到401211地址处的Call。

我们来到了程序自修改的段了，从我们修改PUSH EDX处开始把。告诉OD代码已经被修改了，我们重新分析代码：

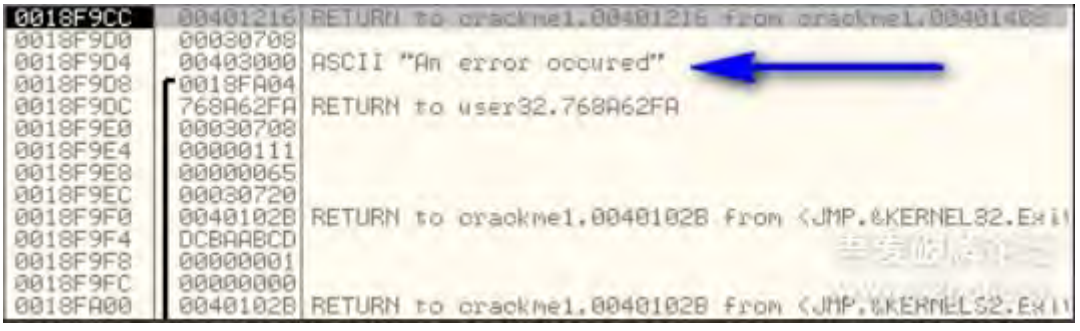


分析后看上去舒服了。

这就是我们常规的代码了，都做了什么呢？EDX+1把EDX移动到EAXXOR EAX，100430D，然后存放到EDX中，EDX+4，把EDX移动到EAX然后与52154F01进行XOR后放回EDX中，EDX+4，把EDX移动到EAX然后与10170D0E进行XOR后放回EDX中，EDX+4，把EDX移动到EAX然后与454516进行XOR后放回EDX中，出栈EDX，出栈EAX。AL+0x66

最后，我们都是不正确的解密从地址40143d几个内存位置。但是Call到SetDlgItemTextA并不是他们中的一个，这个指令没有改变的意义。一般前一个call到SetDlgItemTextA，我们已经看到，参数压栈，所以我们可以假设当我们输入正确的密码，从40143d 401442说明会包含几个推指令（可能是3个）。

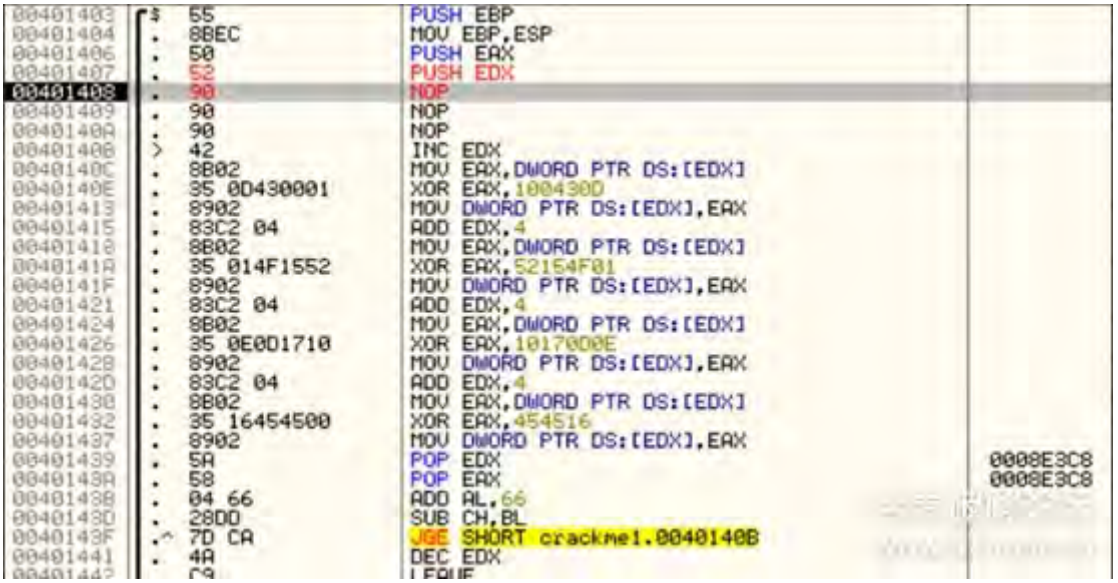
现在的大问题是EDX应该指向哪里？我们在这里有几个选择，应该指向哪里是经验告诉我们的。一个有经验的逆向工程师可能会记得那串“一个错误发生“和认为”我们从来没有使用该字符串。我们看到它只是一个诱饵，从未被使用过。也许这就是将解密…”。另一个提示，告诉我们，这是一个可行的解决方案是字符串被推到堆栈上，但从未使用过。为什么？这里是一个堆栈的图片时，我们进入这个代码：



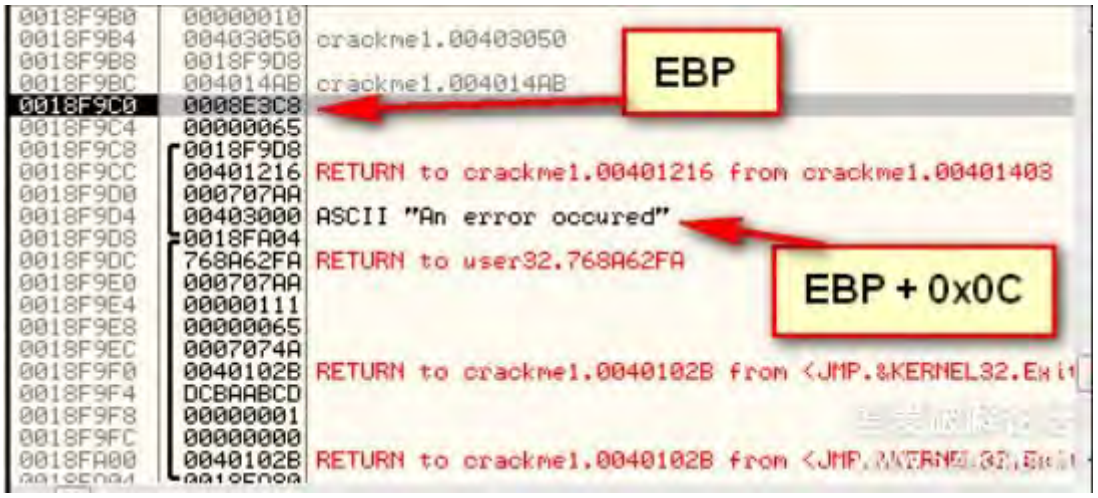
所以验证我们的猜测，我们希望EDX能指向这个字串。最简单的方法就是加载EDX时在内存中做偏移直到“An error occured”字符串放置，即地址403000。问题是，会占用太多字节。再看我们的代码，只有3个字节，也就是3个nop位置来修改EDX指向“An error occurred”字串，好了，保持头脑清醒，和记住堆栈上的字串，或许我们可以在堆栈上让EDX指向我们要的字串。

通常情况下，我们可以这样加载栈上的数据MOV EDX, [EBP + some_#] or MOV EDX, [EBP -some_#]

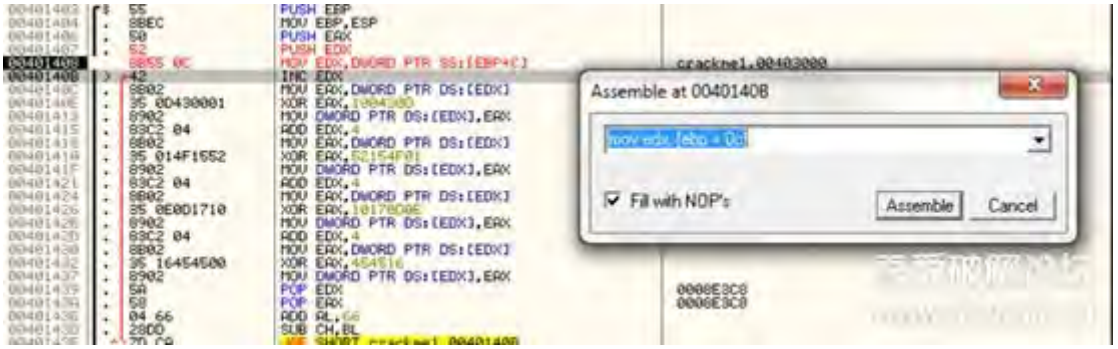
问题是那个才是我们要的偏移值？步过这两个指令，知道我们到达401408



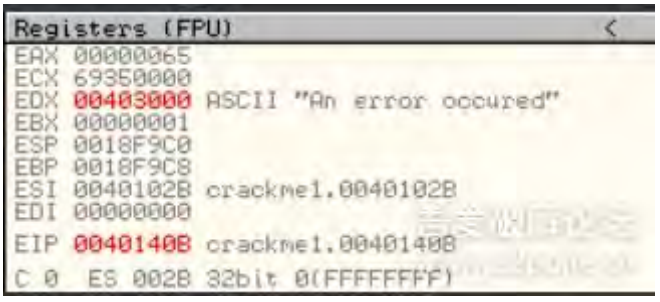
看暂存器中EBP指向的地址，字串比18f9c0高12字节。（栈是先进后出的，所以比他高）



因此我们的指针指向的位置应该是:MOV EDX, [EBP + 0x0C]让我们看看多少个字节被偏移了



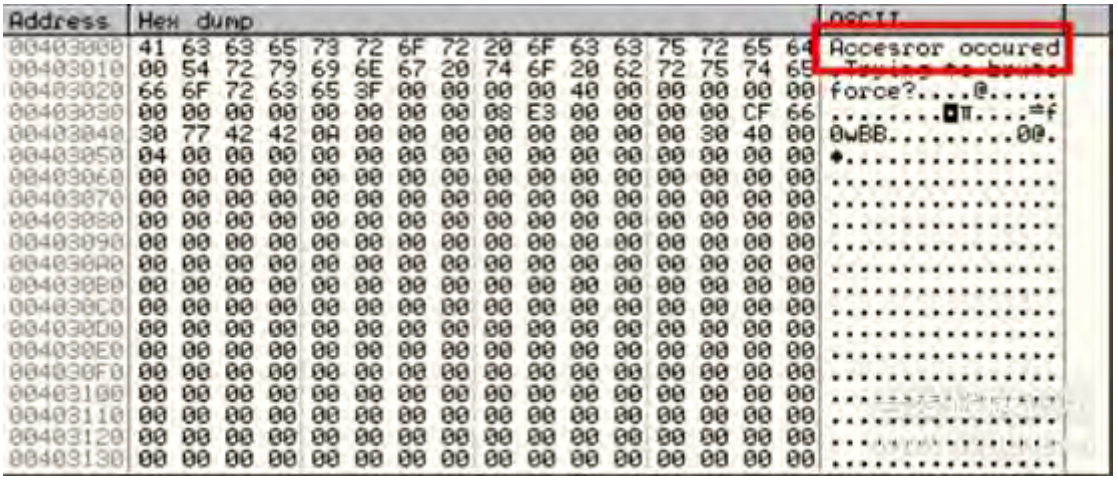
看上去正好。现在我们单步走看看会发生什. 首先在地址401408处, EDX 装载了我们要的字串。



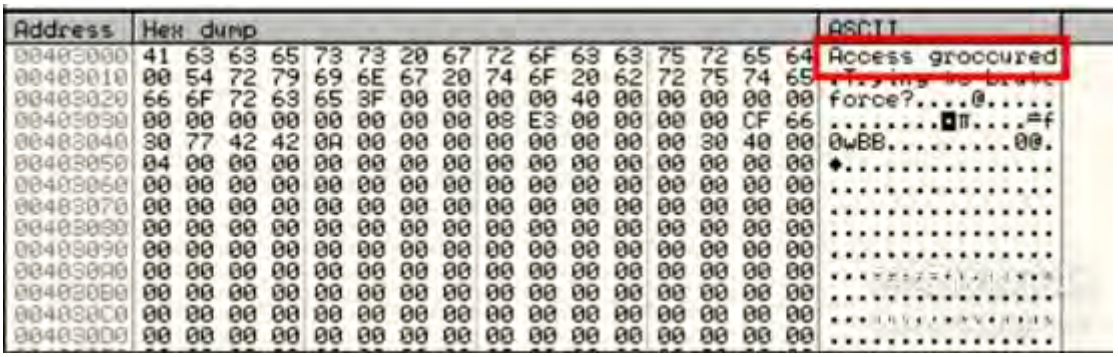
Edx + 1， 现在指向了第二个字符n，然后从n开始4字节移动到eax，然后eax和0x100430D进行Xor后保存到

Edx的刚刚移动的给eax的位置：移动前：

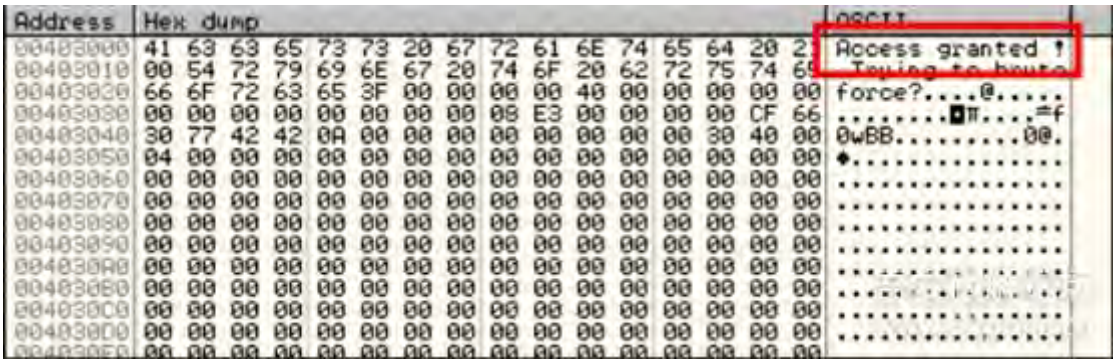
移动后



嗯嗯，字串已经被修改了。我们继续，然后edx偏移4字节后，装载到eax然后与0x52154f01进行Xor后保存到edx偏移后的地址。

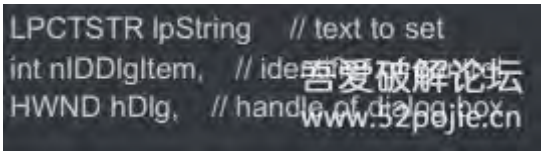


后续也是一样的操作，只是Xor的数据不一样，最后得到

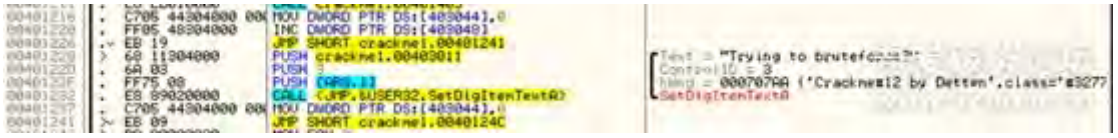


我们发现字串对了, 尽管我们还没有明白原理.只是知道正确的String是什么了. 问题是，既然我们输入了错误的

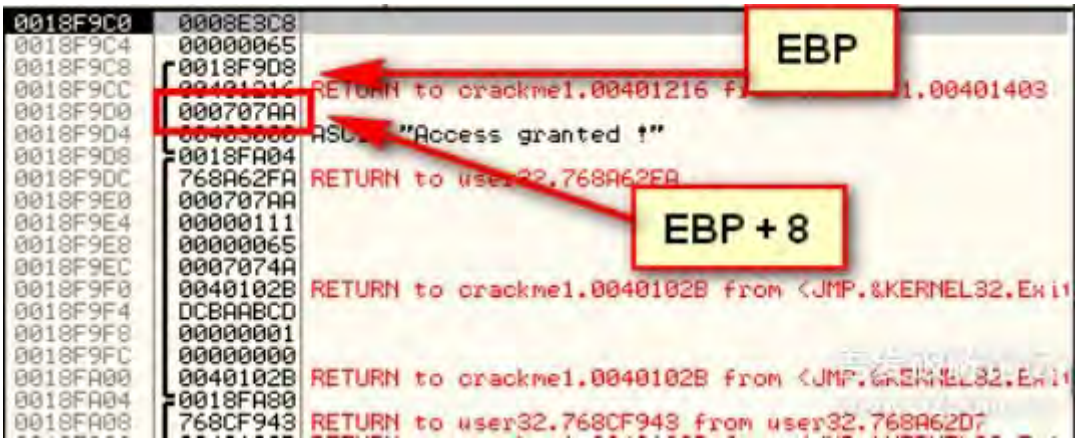
密码，最后的陈述是不正确的解密，我们的信息将不会被显示.我们必须要找到在调用SetDlgItemTextA时往栈中压入什么参数，查找SetDlgItemTextA的帮助文档，这里有3个参数：



第一个参数：简单，PUSH [EBP + 0x0c]就是我们的字串指针。、第二个和第三参数有点麻烦：但是幸运的是我没有参考. 因为这里有一个失败时调用SetDlgItemTextA的参数。



我们发现ControlID为3，Window Handle为707AA第二个参数：PUSH 3第三个参数有点麻烦，我们看下栈数据



第三个参数：PUSH [EBP + 8]好了，现在开始补丁吧

好了补丁完成了。（译者注：看作者的分析可能和我们的不一样，因为10次输入按键不同导致的。所以按照作者的思路还是可以正常破解的）作业：只用按一个按钮就显示成功信息。加分作业：启动就显示成功信息。
http://pan.baidu.com/s/1bEvlTG 密码：e5ve

其实我希望能找到一个高压缩的软件，被你们提醒了，就放上网盘链接了。

谢谢支持，有时间会继续做后面的翻译的，我也是新手，翻译是按照我理解的方式去做的，可能和原版不同。大家共同学习吧！

第十六课（下）：再谈暴力破解

先说说上篇文章最后留的作业这是我做的，可能不是最好的方法第一个是按一次按钮出现破解成功信息。这个简单，把代码中和0xA比较改为和0x1比较就OK了。第二个也不是很难，初始化显示成功在初始化数据“DEAD”“42424242”和错误信息后的跳转，直接跳到Call正确信息的地方。

注意：1、 需要把3次错误检查的地方Nop掉，否则点三次按钮还是会出现暴力破解信息。2、 点第二次的时候会回到一个错误发生字串，这里我还没想到要怎么处理，有思路的请提供想法，谢谢！（其实用奇偶的方法可以做，但是代码空间可能够）

以上就是作业的思路，有兴趣的再去试试看，可能你的方法比我好。欢迎提供不同的思路。

翻译都是我理解的方式进行描述，可能和原文不一致。本教程中文版只在吾爱破解论坛 首发。转载请注明来自吾爱破解论坛@52pojie.cn

正文开始了

暴力破解是一种方法，这中方法是你可以从程序中找到爆破点然后直达我们需要的地方，虽然通过常规的加密/解密你已经知道这个程序的输入和输出，但是你不知道他的解密过程，而是直接打了一个补丁就完事。而这种方法与通过输入用户名和序列号的方式是不同的。如果你以前下载过破解软件，是通过输入用户名和序列号来让程序工作的，它有可能是通过暴力破解来的。

这种工作方式是通过常规的加密/解密输入的用户名和序列号，你尝试不同的输入直到有一次匹配成功，例如：我们输入“12121212”，程序通过常规解密后得到“j6^^gD7-L”，我们使用不同的输入，得到的结果是不同的。我们就是要想办法知道“12121212”是怎么变成“j6^^gD7-L”的，然后让我们的输入的序列号能让程序启动，或者能够注册成功。

请记住，这中方法只适用于程序内部检查用户名和序列号，不适用于网络验证。

和大家说的一样，暴力破解不是难事，首先你至少知道一门编程语言这样就可以自己编写暴力破解的程序，此教程主要是汇编语言，因为要分析代码中的算法。作者自己也会两门其他编程语言____，因此你就可以在高级语言中做暴力破解算法（译者注:我只会C++初阶）

另一个就是要明白用户名和序列号是怎么变成输出的，这样做的原因是它减少了操作次数，所以我们必须尝试。如果我说我们必须在密码栏输入“SECRET“导致输出为”MESSAGE”。这有无限多的方法。但是如果我说把用户名做异或操作后的值很有价值，这样会减少很多方法。

破译密码

还记得前面的教程中我问你是否能破译密码吗，修改什么地方能出现成功，下面就是所有的程序代码：（译者注：这里后面的解释好像ecx和eax反了，我自己是吧eax当作a，ebx当作b，ecx当作c，然后我放上我的解释）

```
004012A9 mov ecx, dword_403040          ; 变量 'c'
004012AF mov ebx, dword_40303C          ; 变量 'b'
004012B5 mov eax, dword_403038          ; 变量 'a'
004012BA cmp [ebp+arg_0], 1             ; ***** Button 1
004012BE jnz short loc_4012D0
004012C0 add ecx, 54Bh                  ;c += 54Bh
```

```
004012C6 imul ebx, eax                  ; b *= a
004012C9 xor eax, ecx                   ; a^= c
004012CB jmp loc_4013E7
004012D0 cmp [ebp+arg_0], 2             ; ***** Button 2
004012D4 jnz short loc_4012E8
004012D6 sub ecx, 233h                  ; c -= 233h
004012DC imul ebx, 14h                  ; b *= 14h
004012DF add ecx, eax                   ; c += a
004012E1 and ebx, eax                   ; b &= a
004012E3 jmp loc_4013E7
004012E8 cmp [ebp+arg_0], 3             ; ***** Button 3
004012EC jnz short loc_4012FD
004012EE add eax, 582h                  ; a += 582h
004012F3 imul ecx, 16h                  ; c *= 16h
004012F6 xor ebx, eax                   ; b ^= a
004012F8 jmp loc_4013E7
004012FD cmp [ebp+arg_0], 4             ; ***** Button 4
00401301 jnz short loc_401312
00401303 and eax, ebx                   ; a &= b
00401305 sub ebx, 111222h               ; b -= 111222h
0040130B xor ecx, eax                   ; c ^= a
0040130D jmp loc_4013E7
00401312 cmp [ebp+arg_0], 5             ; ***** Button 5
00401316 jnz short loc_401324
00401318 cdq
00401319 idiv ecx                       ; a /= c, divisionrest --> (r)
0040131B sub ebx, edx                   ; b -= r
0040131D add eax, ecx                   ; a += c
0040131F jmp loc_4013E7
00401324 cmp [ebp+arg_0], 6             ; ***** Button 6
00401328 jnz short loc_401339
0040132A xor eax, ecx                   ; a ^= c
0040132C and ebx, eax                   ; b &= a
0040132E add ecx, 546879h               ; c += 546879h
00401334 jmp loc_4013E7
00401339 cmp [ebp+arg_0], 7             ; ***** Button 7
0040133D jnz short loc_401351
0040133F sub ecx, 25FF5h                ; c -= 25FF5h
00401345 xor ebx, ecx                   ; b ^= c
00401347 add eax, 401000h               ; a += 401000h
0040134C jmp loc_4013E7
00401351 cmp [ebp+arg_0], 8             ; ***** Button 8
00401355 jnz short loc_401367
00401357 xor eax, ecx                   ; a ^= c
00401359 imul ebx, 14h                  ; b *= 14h
0040135C add ecx, 12589h                ; c += 12589h
00401362 jmp loc_4013E7
00401367 cmp [ebp+arg_0], 9             ; ***** Button 9
0040136B jnz short loc_401378
0040136D sub eax, 542187h               ; a -= 542187h
00401372 sub ebx, eax                   ; b -= a
00401374 xor ecx, eax                   ; c ^= a
00401376 jmp short loc_4013E7
00401378 cmp [ebp+arg_0], 0Ah           ; ***** Button 10
0040137C jnz short loc_40138A
```

```
0040137E cdq
0040137F idiv ebx                       ; a /= b, division rest -->(r)
00401381 add ebx, edx                   ; b += r
00401383 imul eax, edx                  ; a *= r
00401386 xor ecx, edx                   ; c ^= r
00401388 jmp short loc_4013E7
0040138A cmp [ebp+arg_0], 0Bh           ; ***** Button 11
0040138E jnz short loc_4013A3
00401390 add ebx, 1234FEh               ; b += 1234FEh
00401396 add ecx, 2345DEh               ; c += 2345DEh
0040139C add eax, 9CA4439Bh             ; a += 9CA4439Bh
004013A1 jmp short loc_4013E7
004013A3 cmp [ebp+arg_0], 0Ch           ; ***** Button 12
004013A7 jnz short loc_4013B2
004013A9 xor eax, ebx                   ; a ^= b
004013AB sub ebx, ecx                   ; b -= c
004013AD imul ecx, 12h                  ; c *= 12h
004013B0 jmp short loc_4013E7
004013B2 cmp [ebp+arg_0], 0Dh           ; ***** Button 13
004013B6 jnz short loc_4013C8
004013B8 and eax, 12345678h             ; a &= 12345678h
004013BD sub ecx, 65875h                ; c -= 65875h
004013C3 imul ebx, ecx                  ; b *= c
004013C6 jmp short loc_4013E7
004013C8 cmp [ebp+arg_0], 0Eh           ; ***** Button 14
004013CC jnz short loc_4013DB
004013CE xor eax, 55555h                ; a ^= 55555h
004013D3 sub ebx, 587351h               ; b -= 587351h
004013D9 jmp short loc_4013E7
004013DB cmp [ebp+arg_0], 0Fh           ; ***** Button 15
004013DF jnz short loc_4013E7
004013E1 add eax, ebx                   ; a += b
004013E3 add ebx, ecx                   ; b += c
004013E5 add ecx, eax                   ; c += a
```

在他的教程中为我做了大部分工作 (当我发现我已经完成了三分之二后). ***

现在我们知道了每一个按钮都做了什么操作了。接下来我们需要的是输入和输出。这是我们已经知道的数据了，在代码自修改段中有官方（译者注：程序自己的算法，我们分析的代码）的算法，然后进行一些列合法的异或操作。具体就是与变量a、b、c进行异或后保存，然后第二个以后的数据都是和之前异或后的数据进行再次异或。

地址 401407的值EB 3F 90 90 与a 异或后为 528B550C（这个值是我们之前修改出来的）然后反向求出 a为 B9B4C59C地址 40143B的值04 66 E7 BB与b 异或后为FF 75 0C 6A 然后可反向求出b 就是直接与结果异或就OK了地址 40143F 的值 4D BD 08 8B与c 异或后为03 FF 75 08 然后可反向求出c

我们最终要做的是尝试修改的每一个组合，通过点击按钮模仿每一个尝试手动可能的组合，当我们做了10次按钮操作后，我们可以看到a\b\c中的值，这个就是正确的值。（译者注：不明白为什么是正确的值，是我们补丁后的程序？）

这个程序的作者提供了前两个值是7和9.给出的原因是，如果你的电脑比较慢的话，要把所有有可能的值都试一次的话要花费太多时间，在不知道前两位的情况下我用一台8核的电脑花了大约1小时才破译出密码。知道前两位的情况下只花了大约1分钟。通常在破解程序时我们不会有任何的提示（当然），我在破译程序中有包含两

个已知的数。

下面是C语言写的破译程序

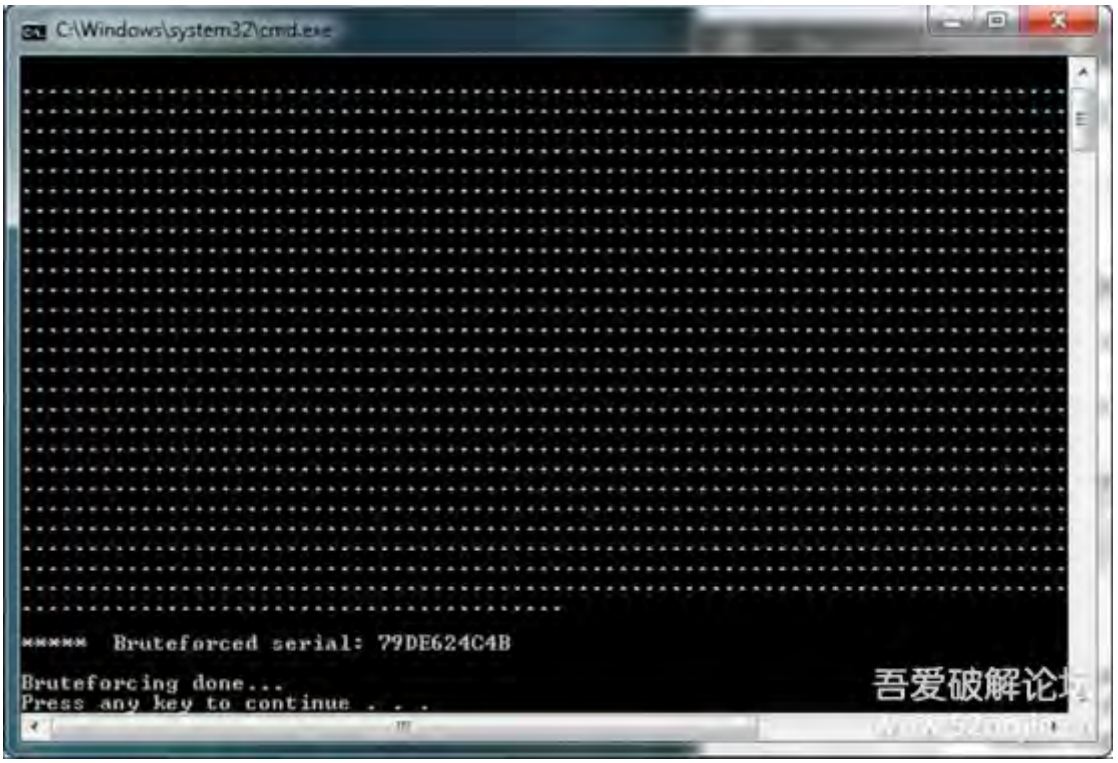
```
#include <iostream>
using  namespace   std;

void brute(  void  )
{
   char  finalAsciiSerial[11] =  "";
   int       i, varA, varB, varC,tempVar, tempSerial[10];

   // we know the first number is '7'
   for   (tempSerial[0] = 7;tempSerial[0] <= 7; tempSerial[0]++)
    {
     // and we know the second number is '9'
     for   (tempSerial[1] = 9;tempSerial[1] <= 9; tempSerial[1]++)
     {
     for   (tempSerial[2] = 1;tempSerial[2] <= 15; tempSerial[2]++)
     {
      for   (tempSerial[3] = 1;tempSerial[3] <= 15; tempSerial[3]++)
      {
        for   (tempSerial[4] = 1;tempSerial[4] <= 15; tempSerial[4]++)
         {
        cout <<  ".";       // Update display
        for   (tempSerial[5] = 1;tempSerial[5] <= 15; tempSerial[5]++)
        {
          for   (tempSerial[6] = 1;tempSerial[6] <= 15; tempSerial[6]++)
          {
          for   (tempSerial[7] = 1;tempSerial[7] <= 15; tempSerial[7]++)
          {
            for   (tempSerial[8] = 1;tempSerial[8] <= 15; tempSerial[8]++)
            {
            for   (tempSerial[9] = 1;tempSerial[9] <= 15; tempSerial[9]++)
            {
                // Reset variables
                varA = 0xDEAD;
                varB = 0xDEAD;
                varC = 0x42424242;

                // Apply each digit
                for   (i = 0; i < 10; i++)
                {
                     switch  (tempSerial)
                     {
                     case  1:
                         varC += 0x54B;
                         varB *= varA;
                         varA ^= varC;
                         break ;

                     case  2:
                         varC = varC - 0x233 +varA;
                         varB = (varB * 0x14)& varA;
```

```
    break ;

case  3:
    varA += 0x582;
    varC *= 0x16;
    varB ^= varA;
    break ;

case  4:
    varA &= varB;
    varB -= 0x111222;
    varC ^= varA;
    break ;

case  5:
    if  (varC != 0)         // Watch divide by zero!
    {
       varB -= (varA %varC);
       varA /= varC;
       varA += varC;
    }
    break ;

case  6:
    varA ^= varC;
    varB &= varA;
    varC += 0x546879;
    break ;

case  7:
    varC -= 0x25FF5;
    varB ^= varC;
    varA += 0x401000;
    break ;

case  8:
    varA ^= varC;
    varB *= 0x14;
    varC += 0x12589;
    break ;

case  9:
    varA -= 0x542187;
    varB -= varA;
    varC ^= varA;
    break ;

case  10:
    if  (varB != 0)         // Watch divide by zero!
    {
       tempVar = varA %varB;
       varA /= varB;
       varB += tempVar;
       varA *= tempVar;
       varC ^= tempVar;
```

```
                    }
                    break ;

                case  11:
                    varB += 0x1234FE;
                    varC += 0x2345DE;
                    varA += 0x9CA4439B;
                    break ;

                case  12:
                    varA ^= varB;
                    varB -= varC;
                    varC *= 0x12;
                    break ;

                case  13:
                    varA &=0x12345678;
                    varC -= 0x65875;
                    varB *= varC;
                    break ;

                case  14:
                    varA ^= 0x55555;
                    varB -= 0x587351;
                    break ;

                case  15:
                    varA += varB;
                    varB += varC;
                    varC += varA;
                    break ;
                }
            }

            // stop if serial equals propervalues
            if  ((varA == 0x9CC5B4B9)
                            &&(varB == 0xD1EB13FB)
                            &&(varC == 0x837D424E))
            {
                // Convert to ASCII
                for   (i = 0; i < 10; i++)
                {
                    if  (tempSerial <= 9)
                    {
                       finalAsciiSerial= tempSerial + 0x30;
                    }
                    else
                    {
                       finalAsciiSerial = tempSerial +0x37;
                    }
                }
                cout <<  "\n\n*****   Bruteforced serial: ";
                                    cout<< finalAsciiSerial << "\n";
                return;
}}}}}}}}}}
```

```
}
int  main()
{
   cout <<  "Bruteforcerby R4ndom\n\n";
   brute();
   cout << "\nBruteforcing done...\n";
   return  0;
}
```

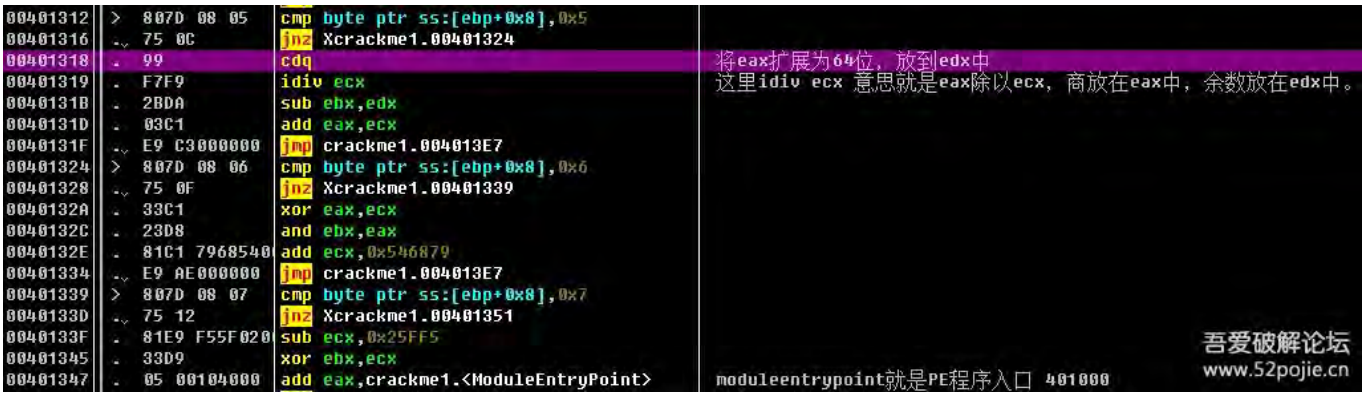
首先，建立我们的变量a\b\c，然后我们知道第一个和第二个是7和9，后面的是在1-15之间，然后我插入了一个“点”字串在控制台中输出，我不太喜欢程序没有任何反应，能看到解密的动作，证明程序没有当掉。接下来，我们执行的变量的修改取决于哪个键被按下，就像我们在上面显示的一样。当输入10个数据时（因为长度是10位），我们会检查这三个变量，看它们是否与我们程序中比较的数据一致，如果一致，我们则停下来，把这个数据转换为ASCII码，然后把它显示在程序上。如果不一致则继续进行下一条数据。下面是控制台破译过程和结果：



让我输入破译后的密码看看程序怎么运行

我们现在已经破译了这个程序。

译者注：算法中要注意5和7里面，5和A是一样的东西



由于工作关系，后面的文章可能会比较晚才进行翻译，因为我要正在破解了，才能写出我理解的流程，才能在这里告诉大家。

第十七章（上）：如何应对 Delphi 二进制代码（上）

翻译都是我理解的方式进行描述，可能和原文不一致。

本教程中文版只在吾爱破解论坛 首发。

转载请注明来自吾爱破解论坛@52pojie.cn

正文开始了

本次教程中，我们将讨论用Delphi写的程序是怎么工作的。Delphi写出的程序和其他语言写出的程序有很大不同。你会说有很多个调用（不仅仅是一个经典程序）我们将讨论其他技术。

本教程需要下载逆向的两个软件，一个是DeDe，一个是ExeinfoPE，均可在本教程中下载。

当然还是需要用到OD的，请自行下载。

Delphi

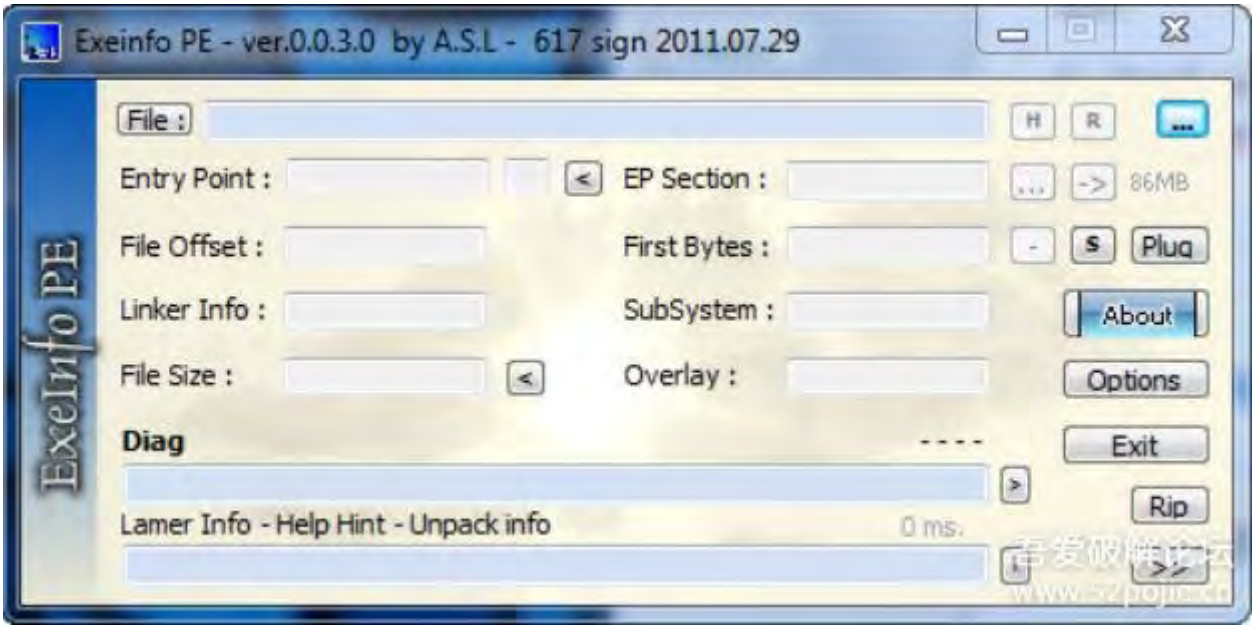
大多数程序都包含许多窗体，基本的窗口和对话框。你使用窗体进行绘制，如添加一个按钮，对话框等，这是你喜欢的方式。你唯一要做的就是告诉Delphi编译器你希望程序做什么样的动作，例如：当你按下按钮后，希望打开一个打开文件的对话框。被教程中你就是要告诉Delphi编译器通过你的代码打开一个对话框，只是简单的代码就可以做到。

这些窗体，以及他们所关联的一切（字串，大小，颜色）都保持到资源中，看上去像C++程序，但是实际上有很大不同。一个有趣的现象是，Delphi通过名称来控制这些资源，也就是说你所调用的特定资源的名称被硬编码编译到可执行文件中的名称。名称看上去就是资源。这样有好处也有坏处，好处是你可以很容易的找到这些名称与资源的管理，坏处是他们都存在同一个地方，在逻辑上看不到正常的代码。所以要找到按钮关联的资源会比较麻烦。

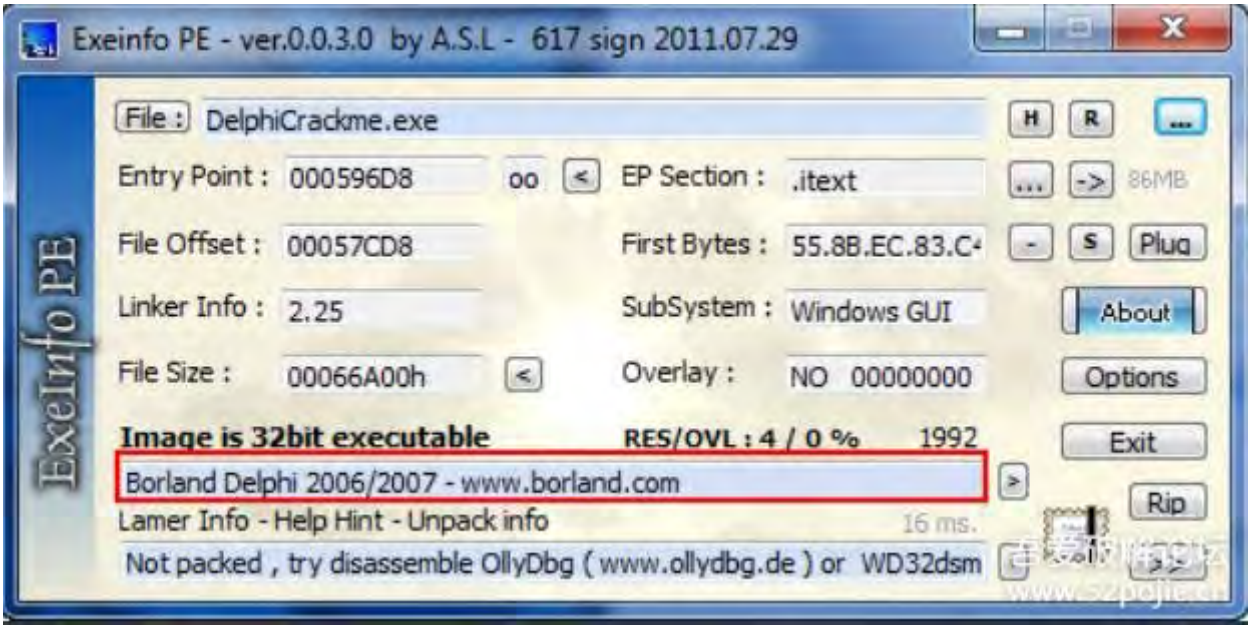
多数程序以分析你就知道，这是C++的，Delphi写的程序在幕后有很多不同，代码看上去就和以前的不同，这就是麻烦的一个原因。

第一个目标

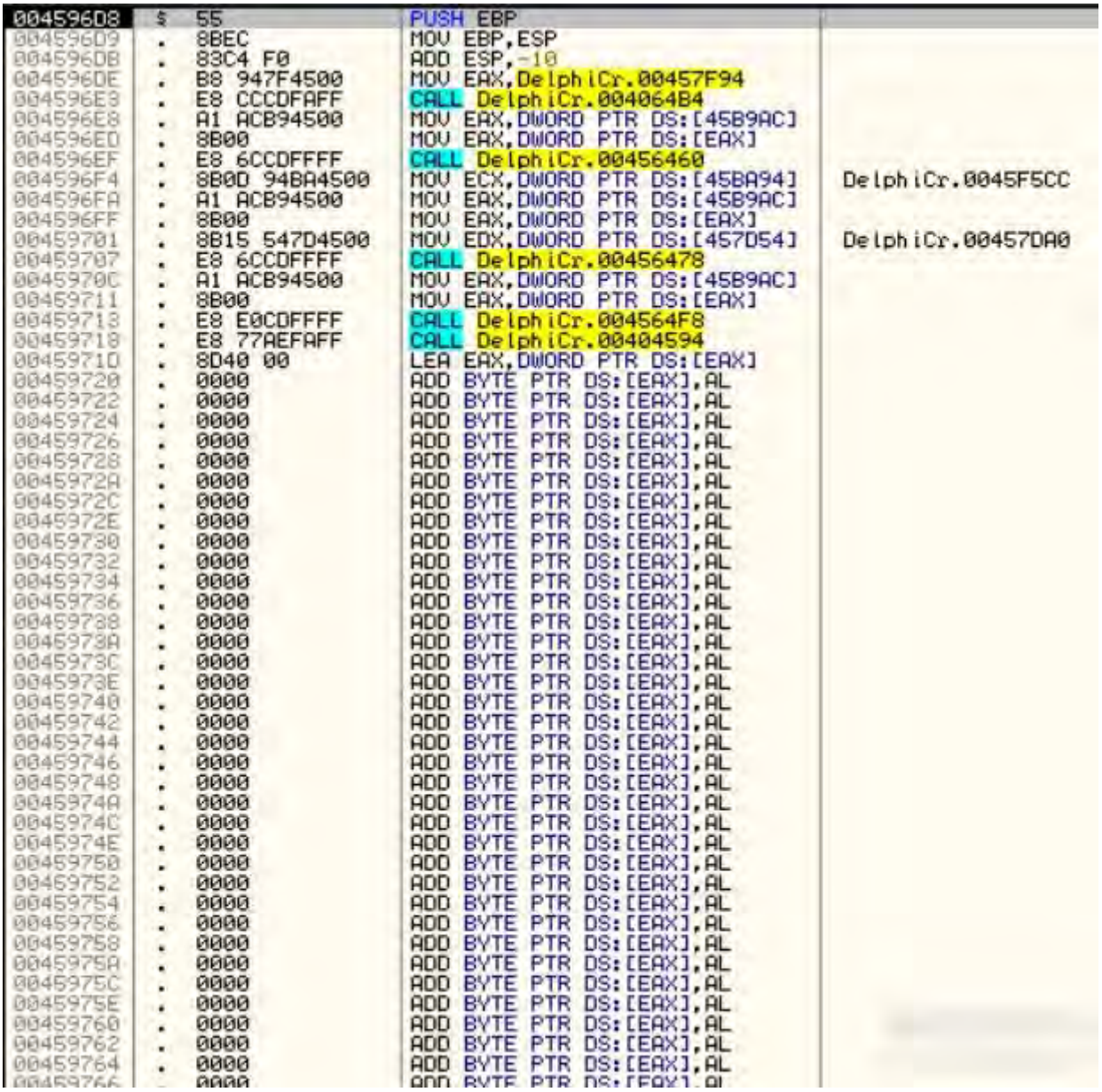
你的第一个问题会问，我们怎么知道这是一个Delphi编译的程序？多数的解释都只说了大概，这是我们就需要使用软件来确认了。打开ExeInfoPE。这个程序可以查看程序是否带壳，带来什么壳，如果不带壳我们就知道是用什么语言来写的了。当你第一次使用ExeInfoPE时，会出现下面的图像：



然后打开我们要查看的程序，DelphiCrackme.exe，然后就可以看见如下图：



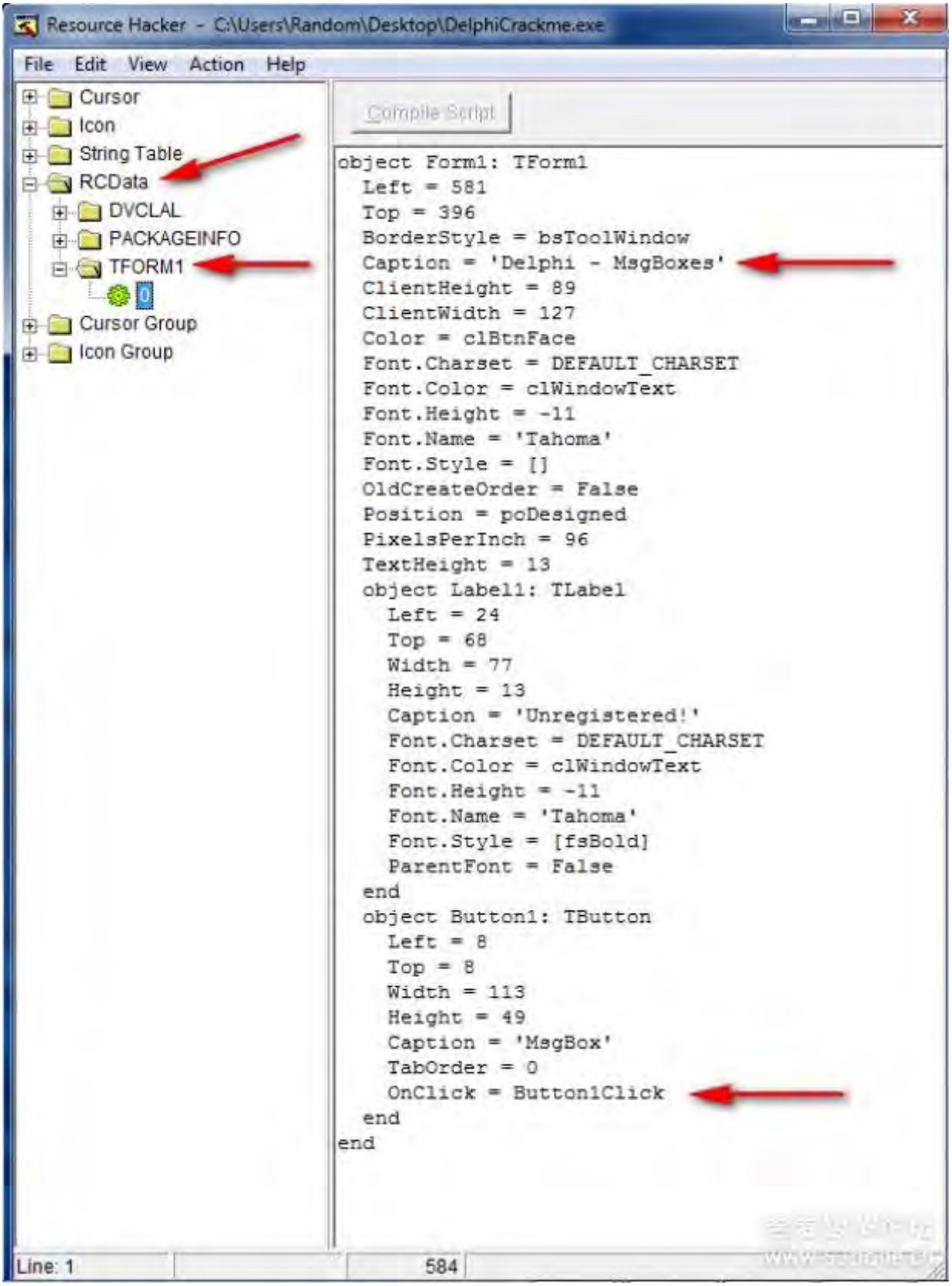
我们看到了该程序是由Delphi写的。也可以看到这个程序是没有加壳的。把这个放一边，使用OD加载该程序，你会看到：



你会发现这和我们以往的程序不太一样。

查看Delphi程序的资源

资源是一个最大的区别，作为一个逆向工程师，这时就把程序使用Resource hacker打开看看吧。这时就发现和以前的不同，多了一个叫RCDATA的新文件夹，点开看看



一般来说，最重要的子文件夹（资源部分）是TFORM段。在Delphi程序中。这些是窗口和对话框，在这个特殊的Crackme中，只有一个TFORM。在Resource Hacker打开TFORM1下面带小花图样，里面的信息告诉我们这个程序的所有数据。比如尺寸，颜色和位置，标题，办好多少个按钮等等。

通常情况下我们先看到标题栏，它告诉我们在窗口中显示的信息。在本教程中，“Delphi-MsgBosxes”是最要的，在程序中可能会调用TFORM1、TFORM2…很难知道哪一个窗体调用TFORM1.看着文字进行分析，例如，标题会告诉我们是否已经注册，关于则给出关于信息。

最后，对我们来说，最重要的在底部的按钮对象。重要的原因是按钮被按下后使用我们的用户名和序列号进行

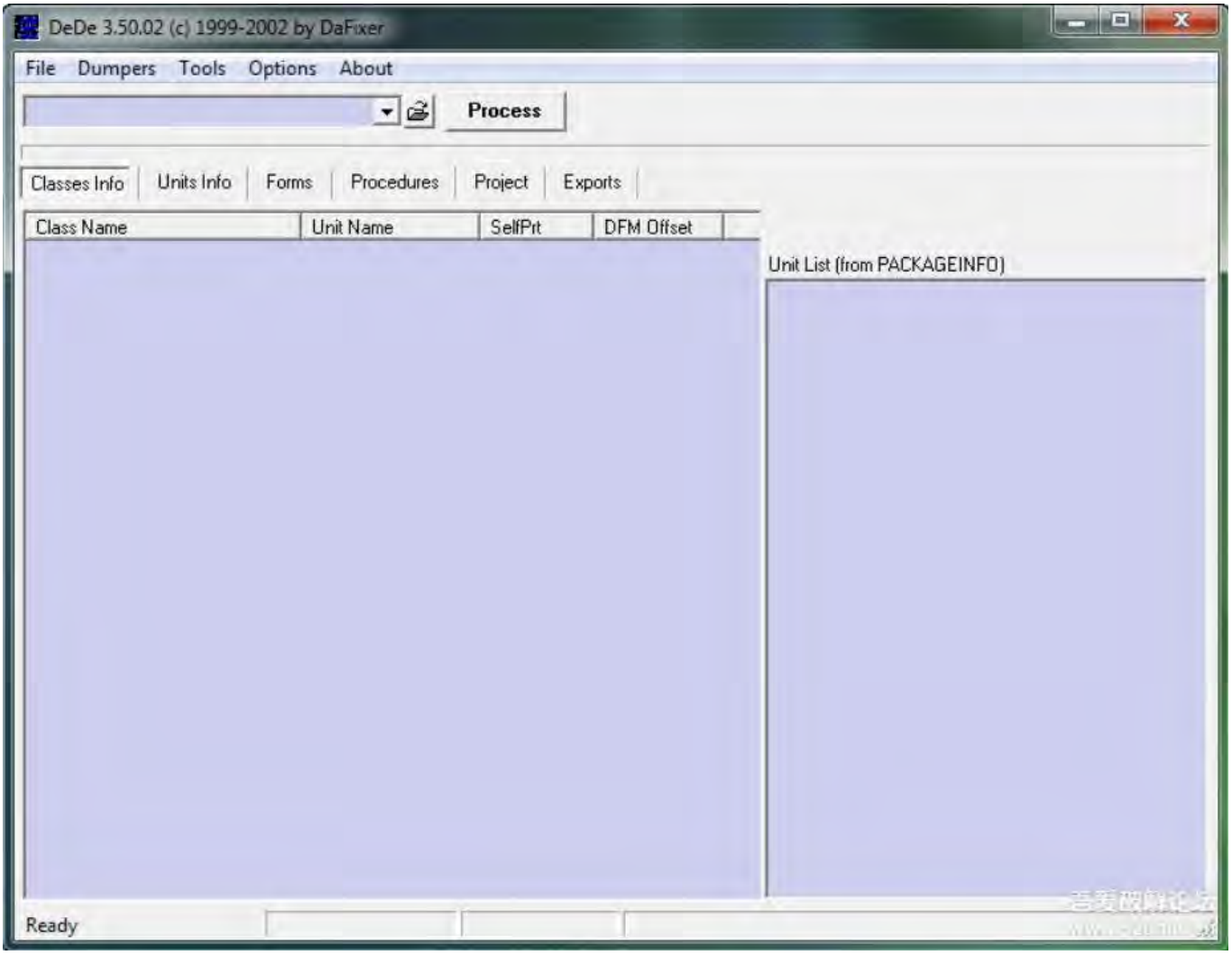
注册，并确认是否注册成功。当按钮被按下时，我们要知道程序做了什么动作。本例中，按钮是“Button1Click”。像前面说的一样，Delphi是通过ASCII与按钮进行关联的，因此当程序按下这个按钮后就会按照“Button1Click”的方法进行动作。

通过Resource hacker得到的信息中，我们知道这个程序只有一个按钮，标题是“Delphi-MsgBoxex”，按钮的句柄为“Button1Click‘。

让我们使用一个对于Delphi程序很重要的软件。

使用DeDe

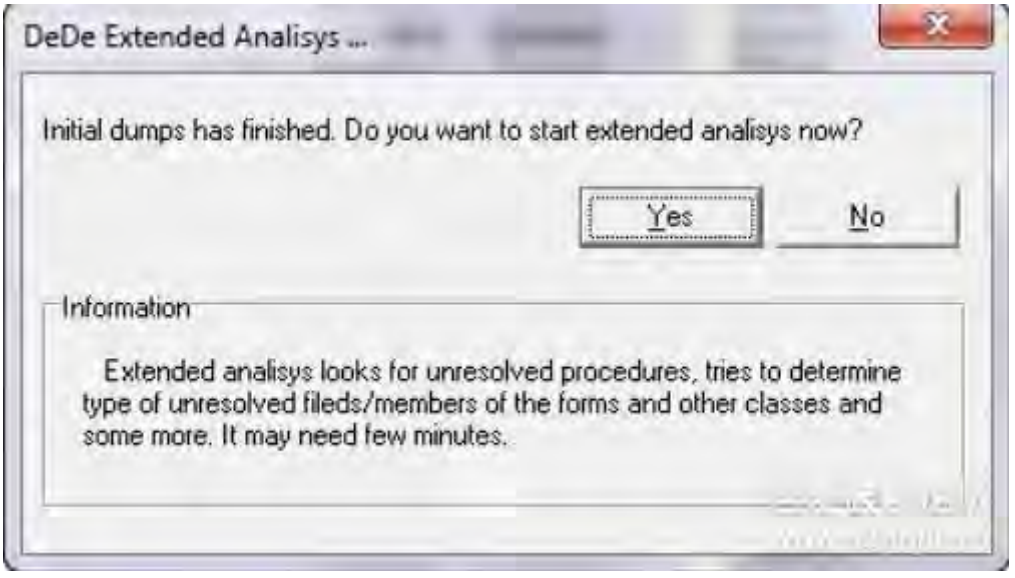
DeDe打开一个Delphi编译后的程序，然后完整的呈现出来。显示出所有我们已经知道的资源信息和所有的调用，实现方法的地址，实现方法的名称，如果你愿意它还可以反编译，当然也可以修改它。让我们打开DeDe，然后加载Crackme。



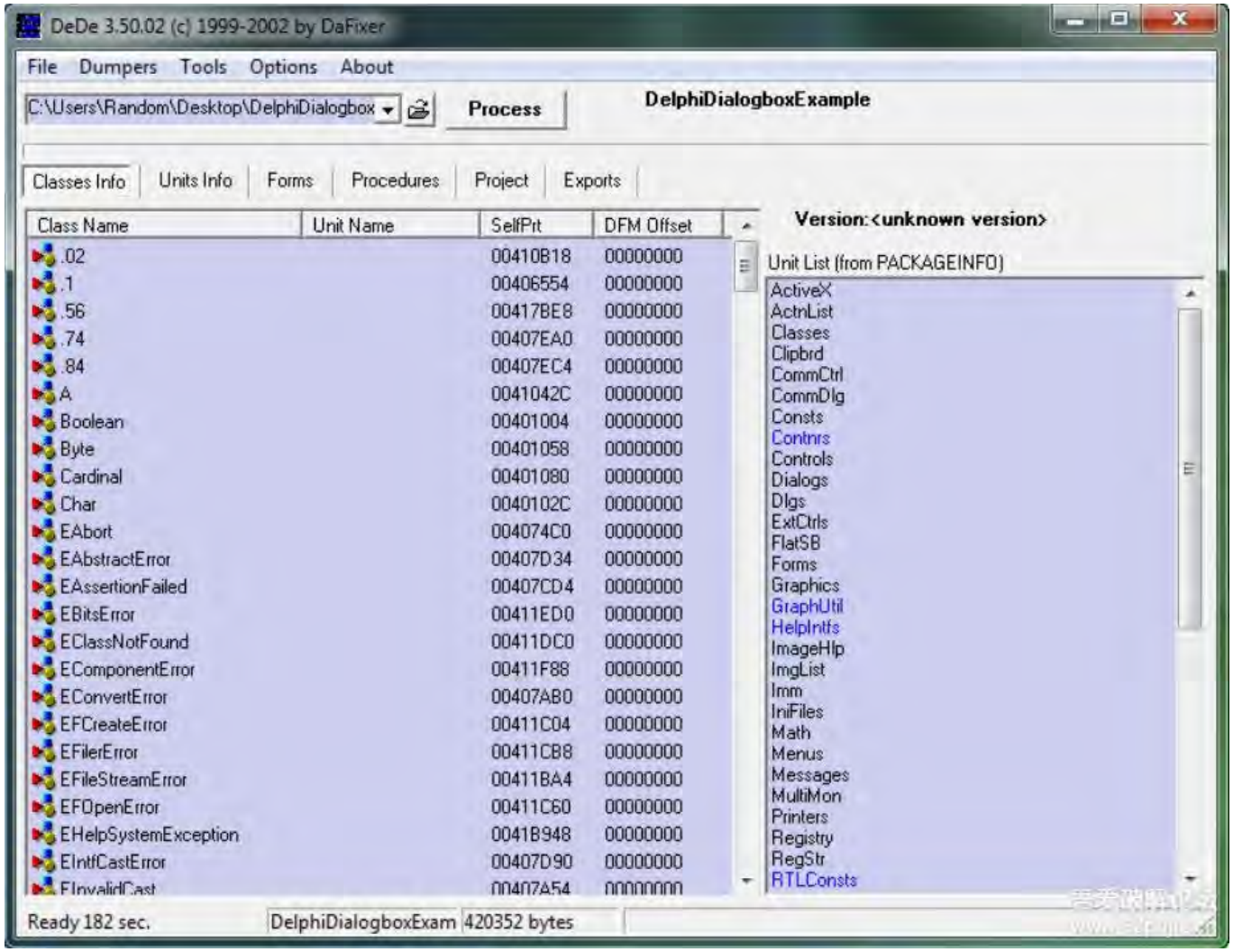
首先，我要打开我们要逆向的程序。有两种方式，一个是通过程序的按钮找到我们要逆向的程序，另外就是直接把程序拖进DeDe，然后选择“YES“开始反编译程序。此时DeDe会弹出信息框询问是否已经加载。

同时我们的程序也跑起来啦

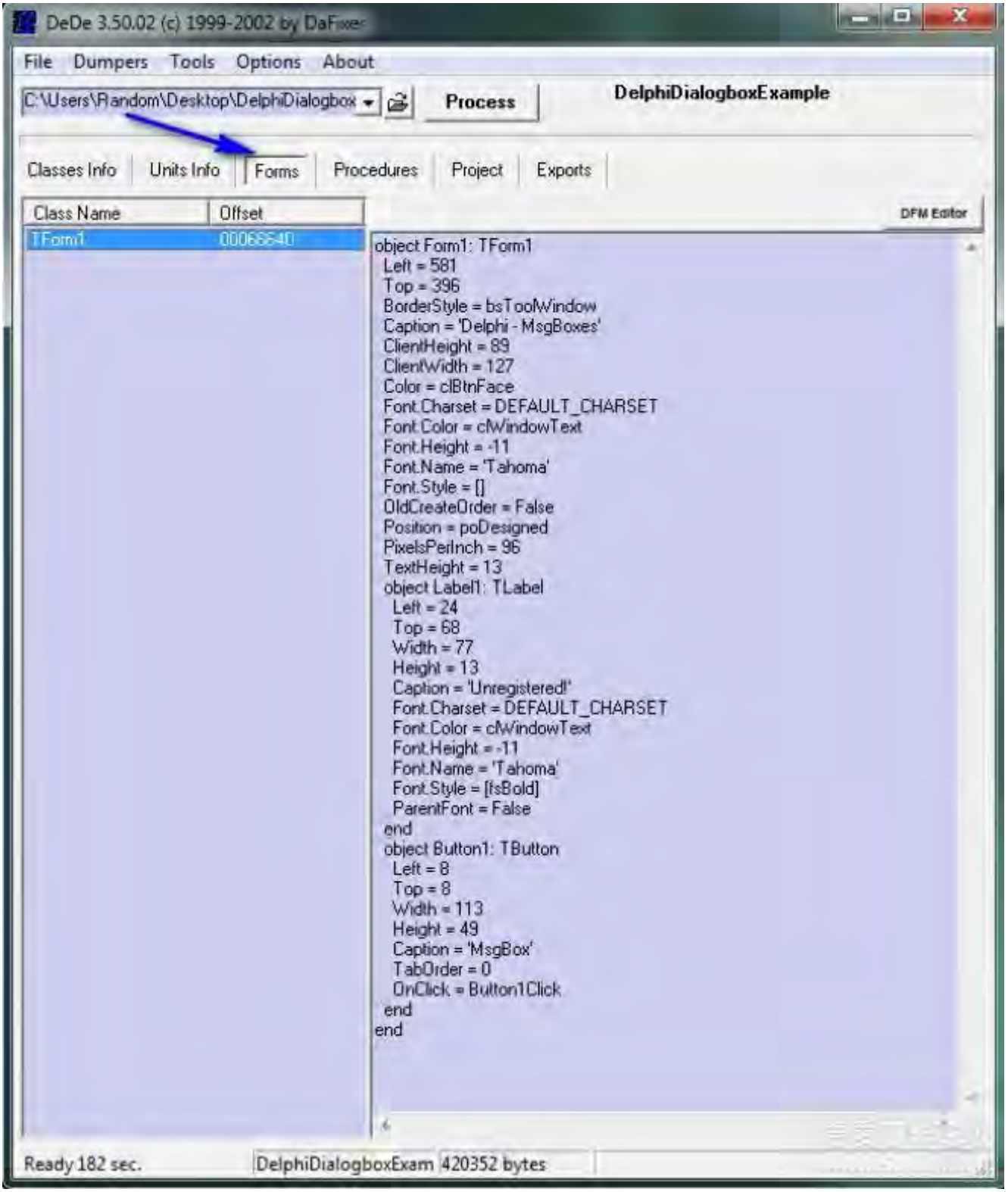
有时，DeDe会弹出一个信息框询问程序是否已经完全加载完成，然后继续进行分析。本例中只需要点击OK按钮，允许DeDe进行分析。DeDe将会关闭程序，然后询问是否使用更多的方式进行分析。



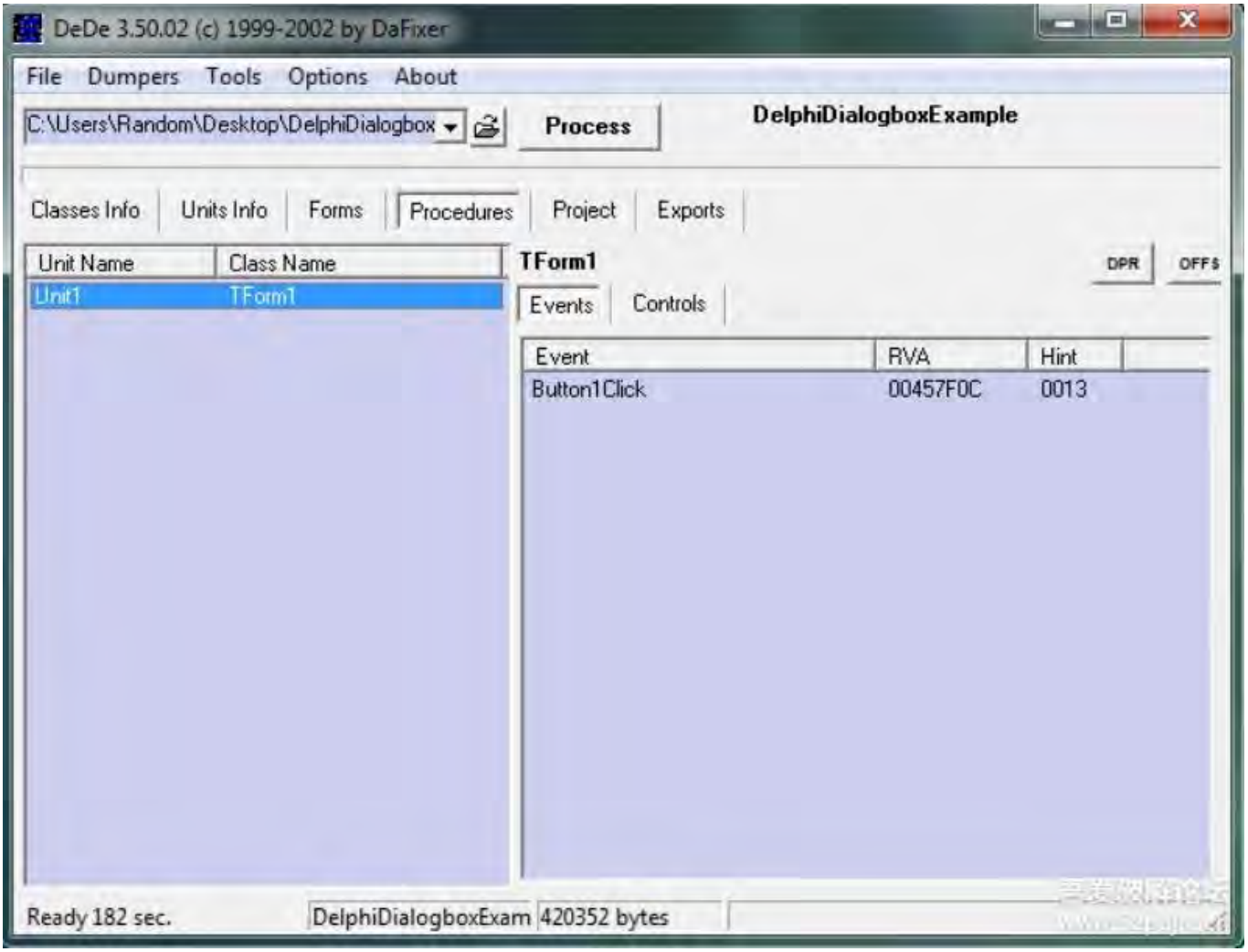
我通常选择yes，然后不再弹窗。然后DeDe分析完程序：



DeDe默认显示类信息，我们通过“ClassInfo“表格可以看到，如果需要可以进行排序，但是我们需要的”FORMS“表：



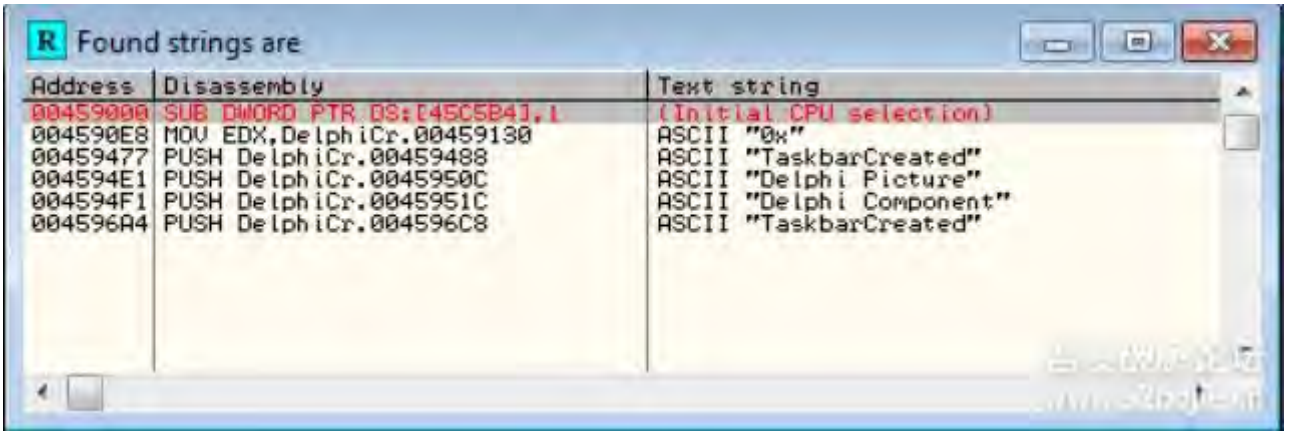
这里，我们看到的和Resource Hacker一样。我这样做是为了以后可以直接略过Resource Hacker，直接使用DeDe。现在点击“Procedures”表。这在DeDe中是很重要的。



这里DeDe显示了TFORM1的实现方法和地址。这是一个简单的程序，只有一个按钮，因此只有一个CallBack。现在我们记住这个地址“457F0C”，显示用OD打开程序，看看我们能做什么。

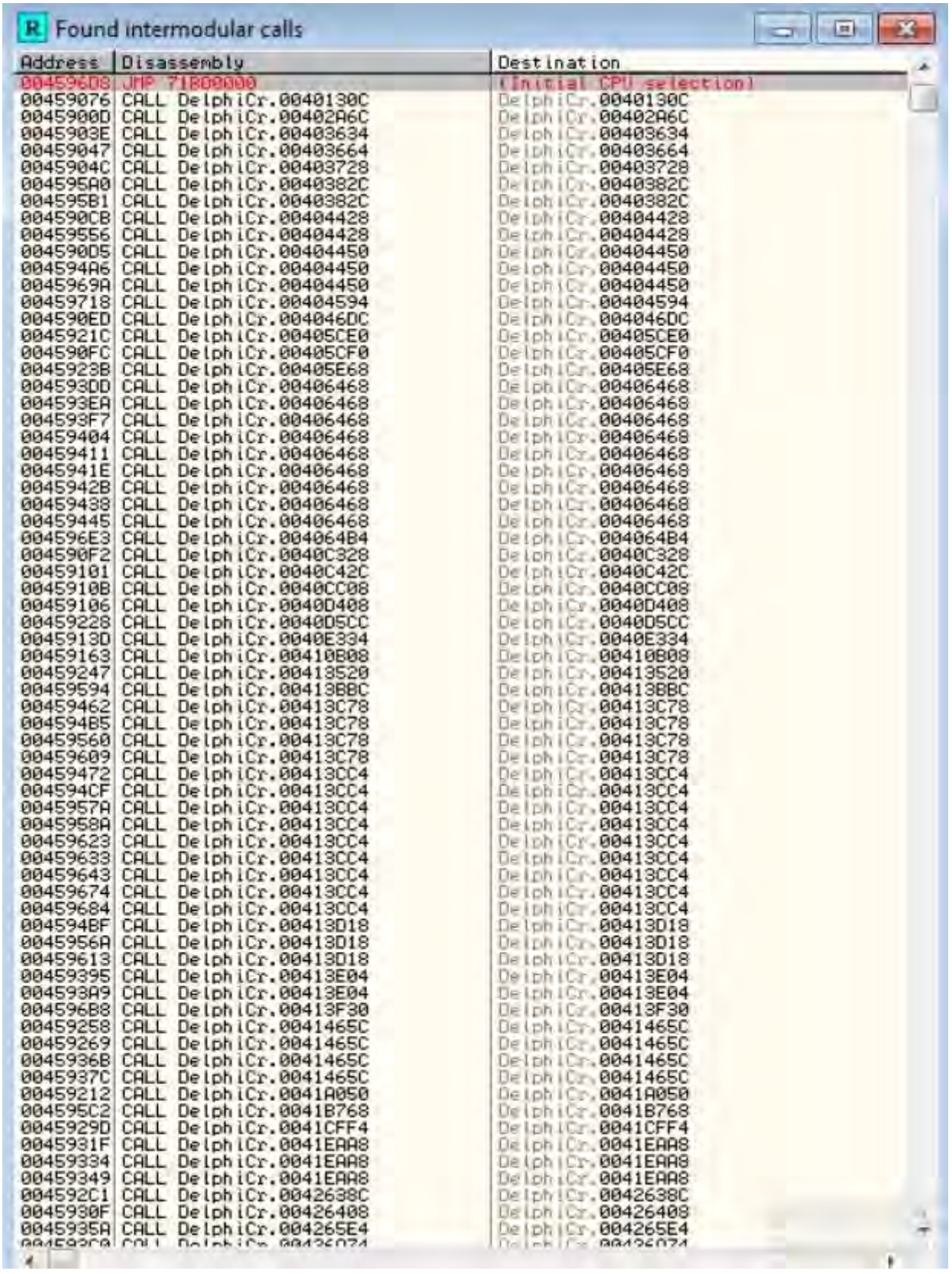
找到补丁位置

如果进行字串搜索能看见：



（译者注：其实我刚开始使用字串搜索能找到，也可以打补丁，运行正常，不明白作者什么意思）

找到的Call在Delphi中都是无用的。（译者注：不明白）



然后看到一大堆的Call

通常情况下，我们停在错误的地方，然后打补丁使程序进入到对的地方，而在Delphi中则不容易找到，需要进入大概15个call之后来到真正代码的地方。

但是如果我们直接去到DeDe中的Call的位置，我们可以看到

```
00457F0C  ?  53              PUSH EBX
00457F0D  .  8BD8            MOV EBX,EAX
00457F0F  .  33C0            XOR EAX,EAX
00457F11  .  3C 01           CMP AL,1
00457F13  .v 75 1C           JNZ SHORT DelphiCr.00457F31
00457F15  .  B8 487F4500     MOV EAX,DelphiCr.00457F48        ASCII "Registered!"
00457F1A  .  E8 5144FDFF     CALL DelphiCr.0042C370
00457F1F  .  BA 487F4500     MOV EDX,DelphiCr.00457F48        ASCII "Registered!"
00457F24  .  8B83 64030000   MOV EAX,DWORD PTR DS:[EBX+364]
00457F2A  .  E8 451DFEFF     CALL DelphiCr.00439C74
00457F2F  .  5B              POP EBX                          DelphiCr.0043B79A
00457F30  .  C3              RETN
00457F31  >  B8 5C7F4500     MOV EAX,DelphiCr.00457F5C        ASCII "Unregistered!"
00457F36  .  E8 3544FDFF     CALL DelphiCr.0042C370
00457F3B  .  5B              POP EBX                          DelphiCr.0043B79A
00457F3C  .  C3              RETN
00457F3D     00              DB 00
00457F3E     00              DB 00
00457F3F     00              DB 00
00457F40  .  FFFFFFFF        DD FFFFFFFF
00457F44  .  0B000000        DD 0000000B
00457F48  .  52 65 67 69 73 7 ASCII "Registered!",0
00457F54  .  FFFFFFFF        DD FFFFFFFF
00457F58  .  0D000000        DD 0000000D
00457F5C  .  55 6E 72 65 67 6 ASCII "Unregistered!",0
00457F6A     00              DB 00
00457F6B     00              DB 00
```

哈哈，这里就是我们要的地方啊，让我们设断点跑起来看看。

注意，这个程序提示我们没注册，标题为“Delphi-MsgBoxes”。只有一个按钮，与我们之前的分析一致。来到我们的断点处看看

现在只需要改一下就可以来到正确的地方。

（译者注：这次程序实际不需要用到DeDe和ExeInfoPE，直接用查找字串参考就可以高低，只是作者希望我们使用DeDe和ExeInfoPE来分析Delphi程序）

第二个程序也是一样的方法，主要是使用DeDe和ExeInfoPE来处理Delphi程序。后面的等有时间再进行翻译。

第十七章（下）：如何应对 Delphi 二进制代码（下）

翻译都是我理解的方式进行描述，可能和原文不一致。

本教程中文版只在吾爱破解论坛 首发。

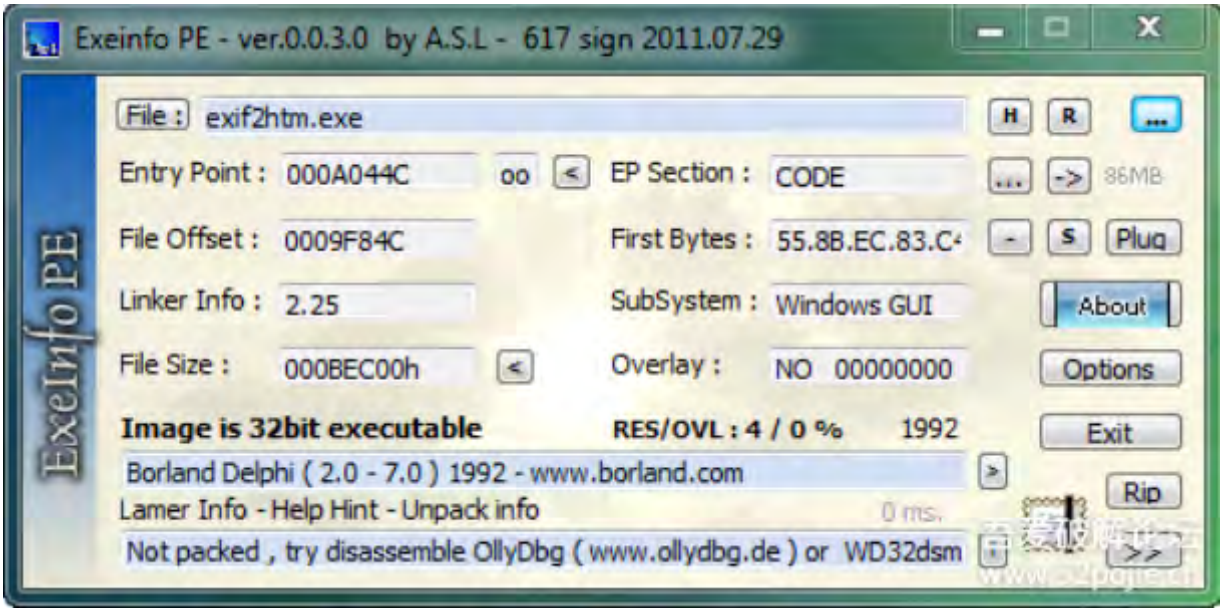
转载请注明来自吾爱破解论坛@52pojie.cn

正文开始

接上章的第二个程序破解。

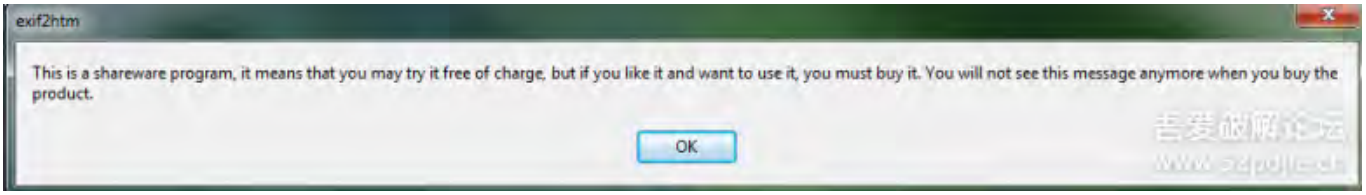
如果是从第一章一直看过来的朋友，一些简单的解释我这里就不再描述，你懂的。

使用ExeinfoPE确定程序是否加壳，什么语言写的

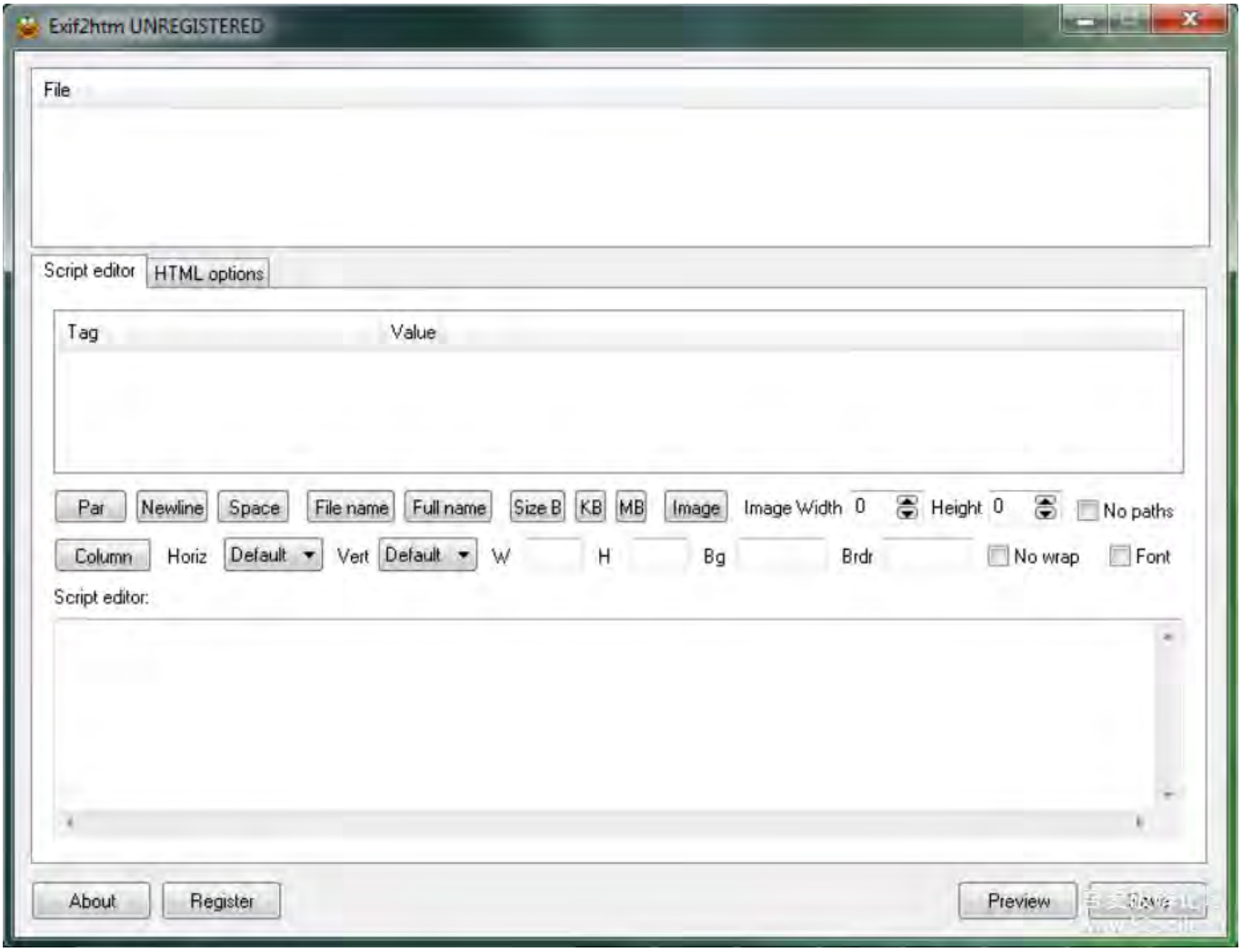


发现没有壳，Delphi写的。

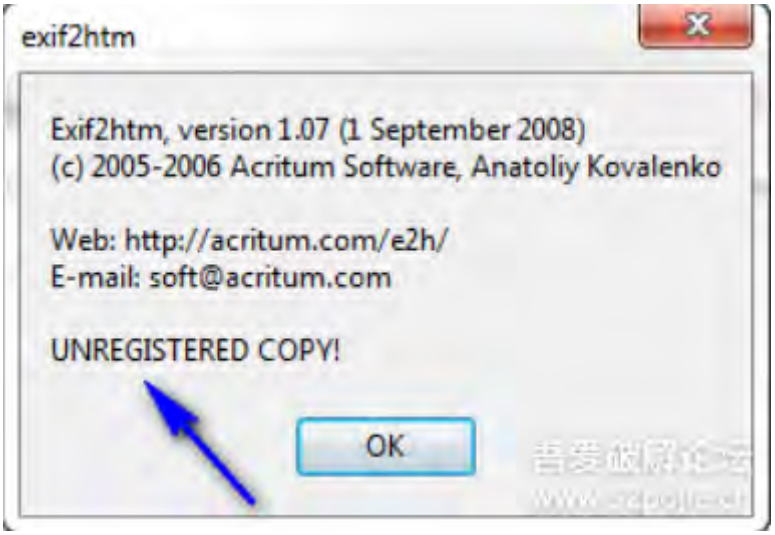
运行程序，弹出



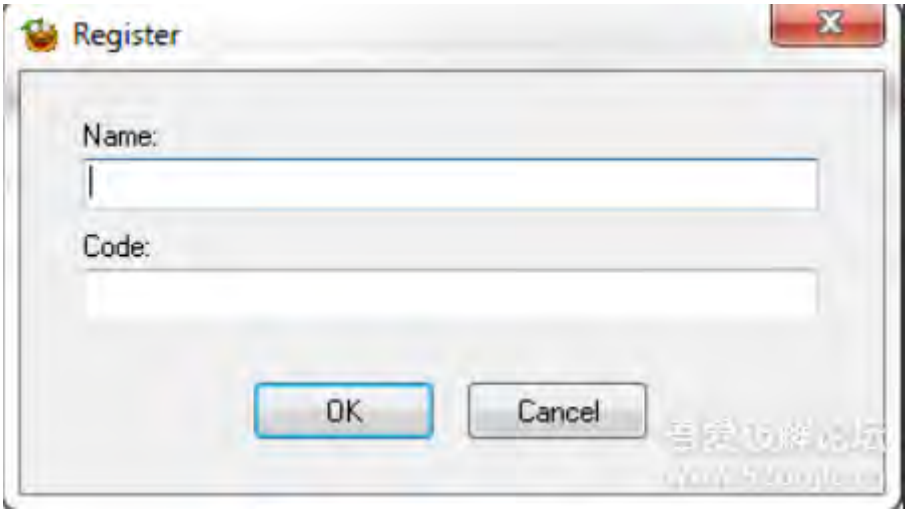
点击OK后出现



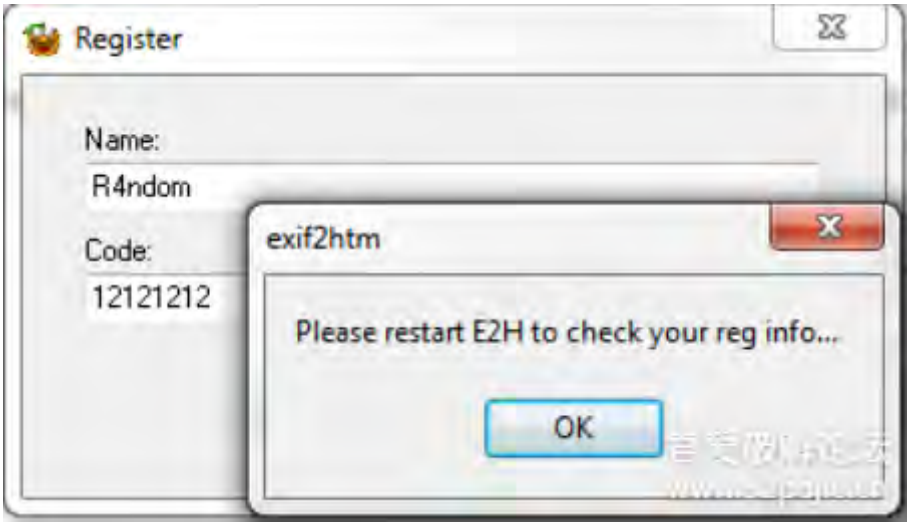
我们可以看见标题是未注册的，然后有一个注册按钮和关于按钮，点击关于后出现



我们发现是未注册的副本，点击注册按钮后出现

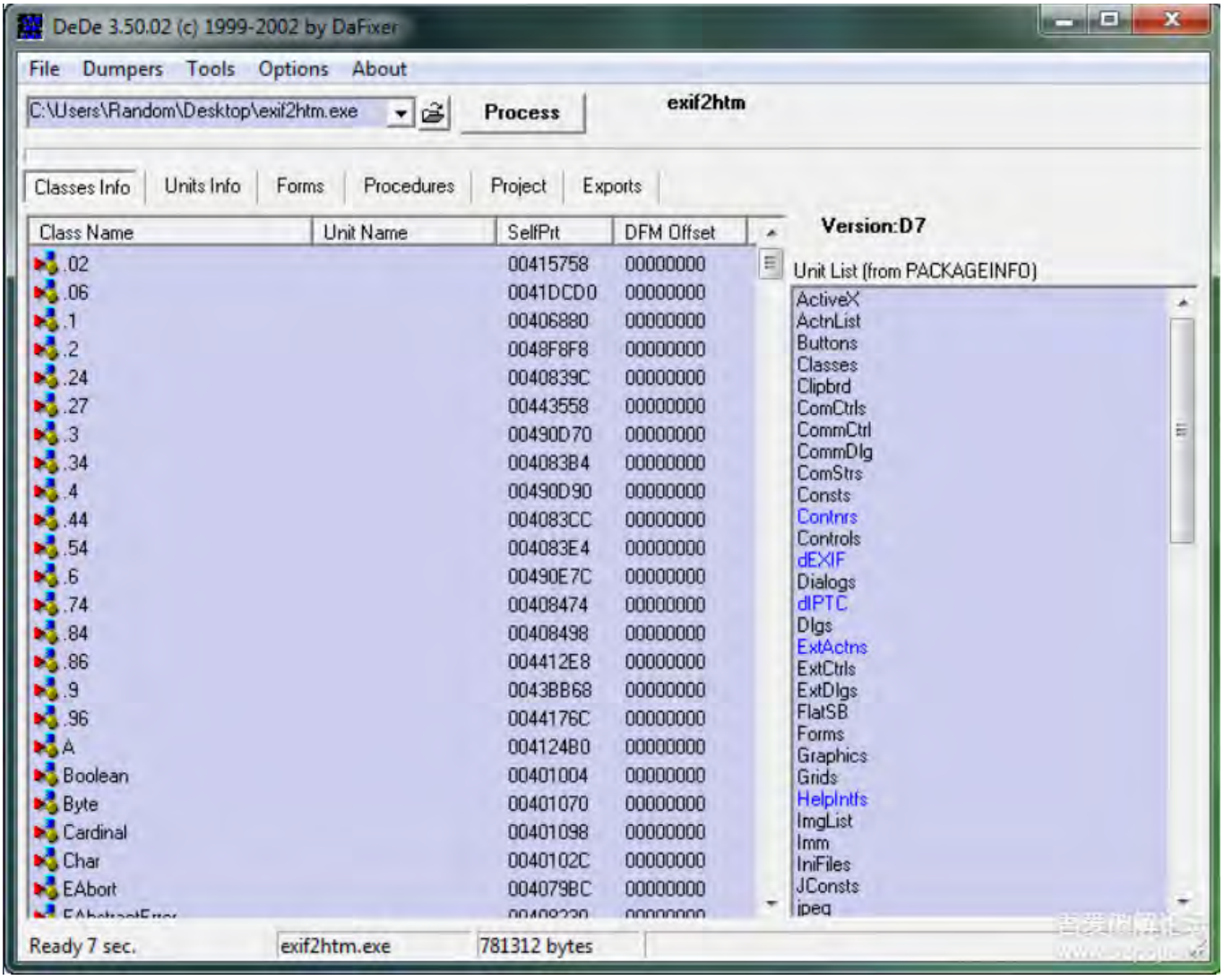


出入用户名和密码后出现

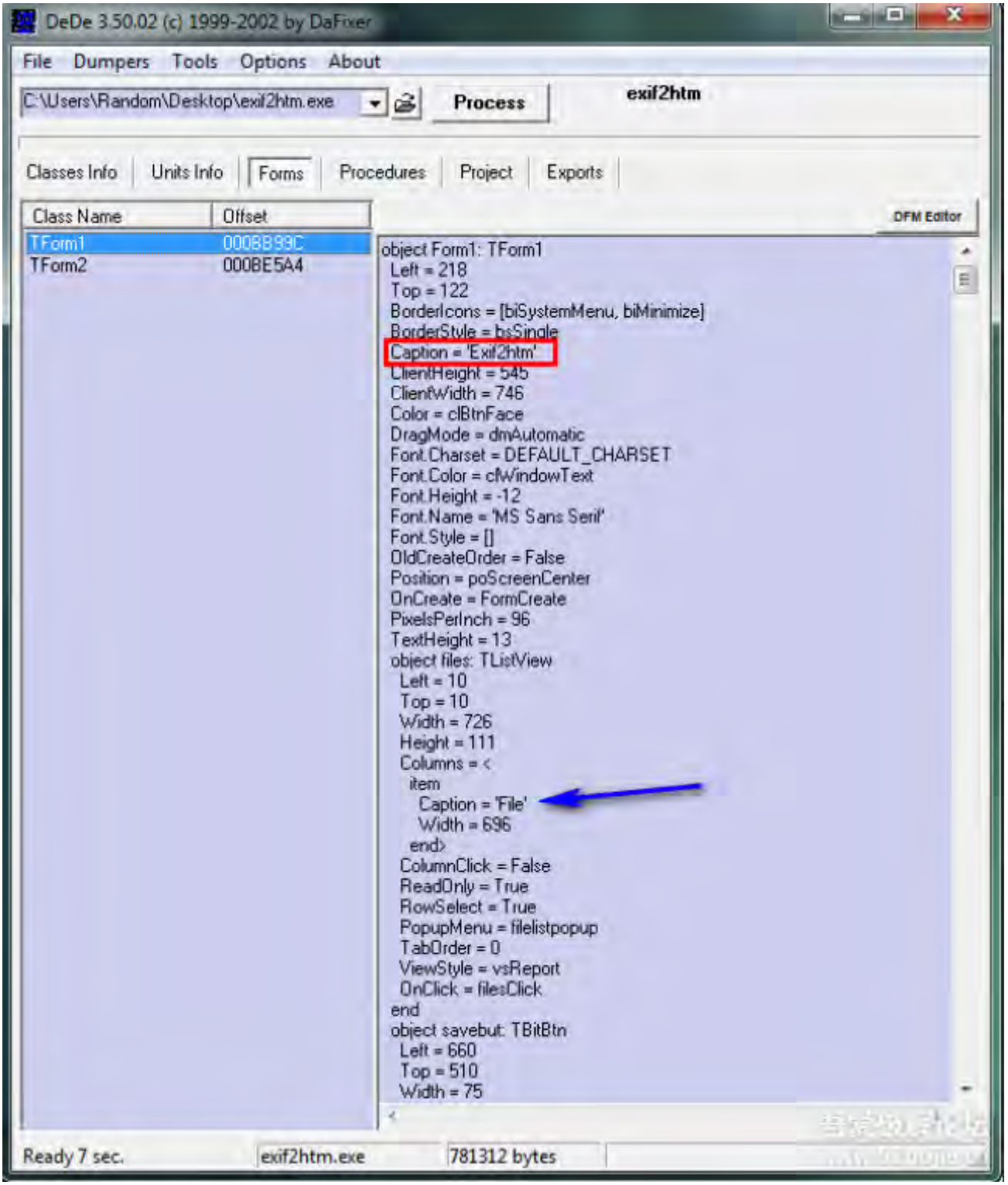


好了，到这里我们知道程序的运行大致流程了。

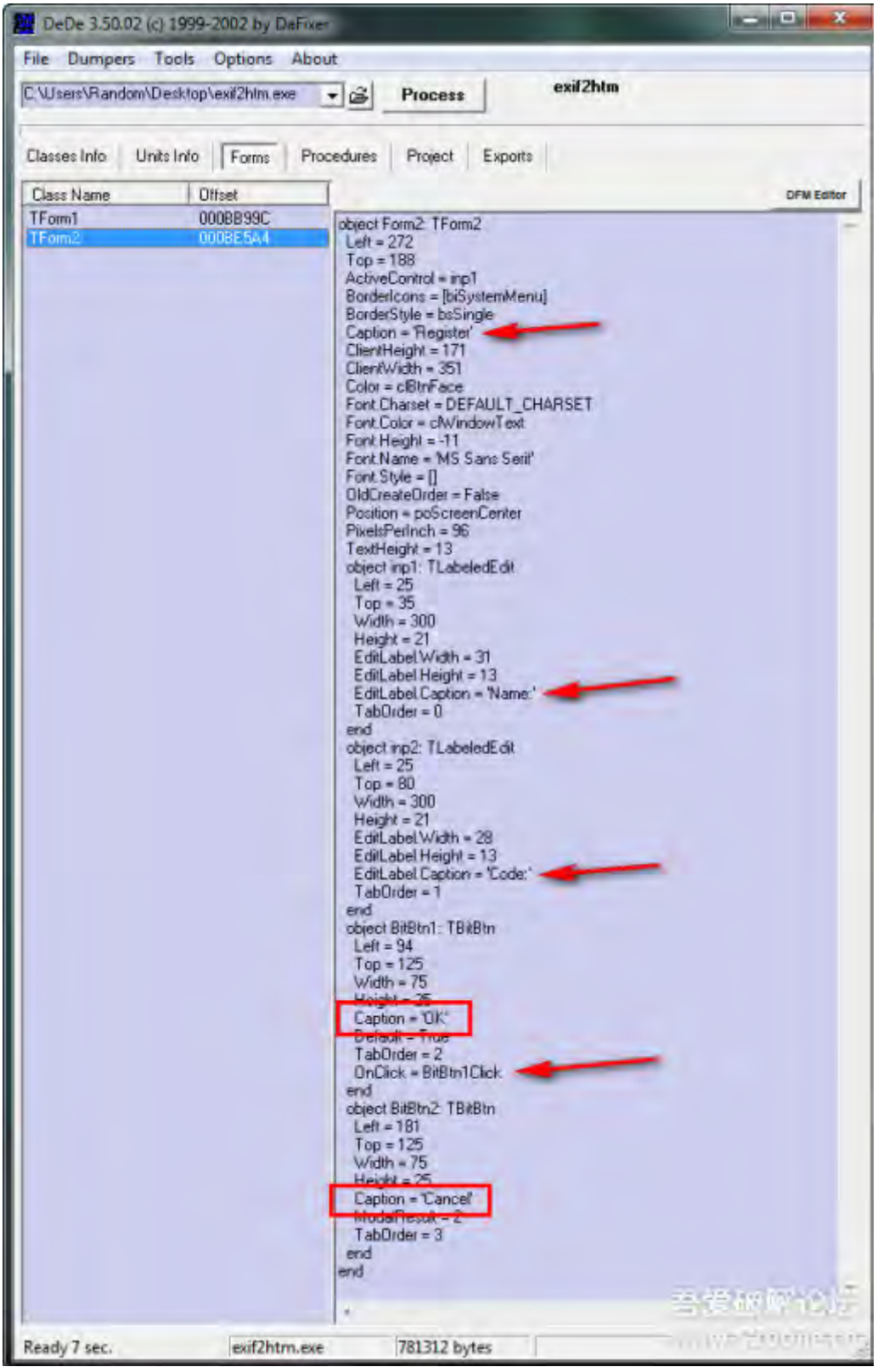
打开DeDe分析程序



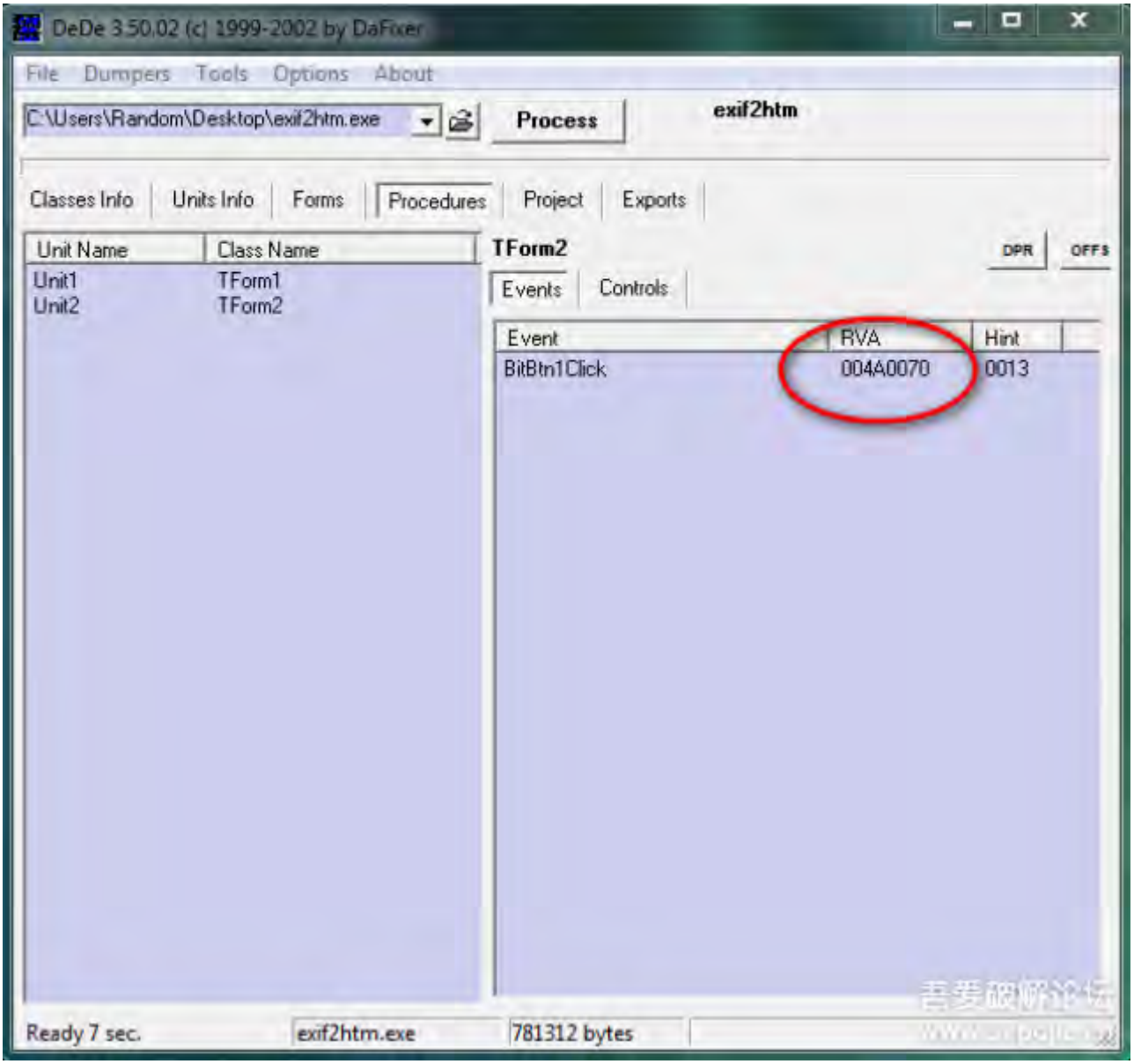
点击Procedures



发现有两个FORM，分别查看，发现1是主窗口，2是注册界面

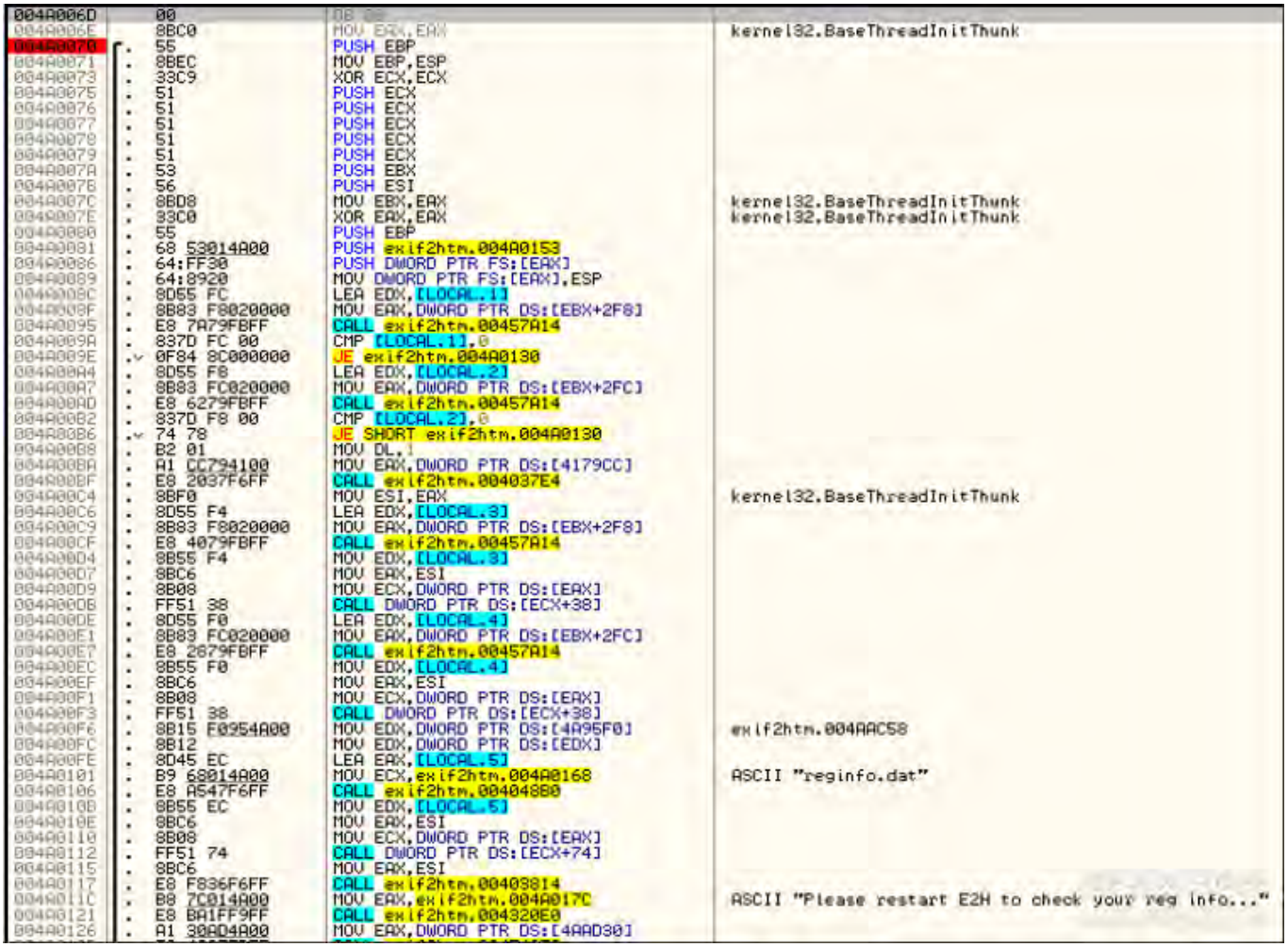


然后查看注册按钮的地址

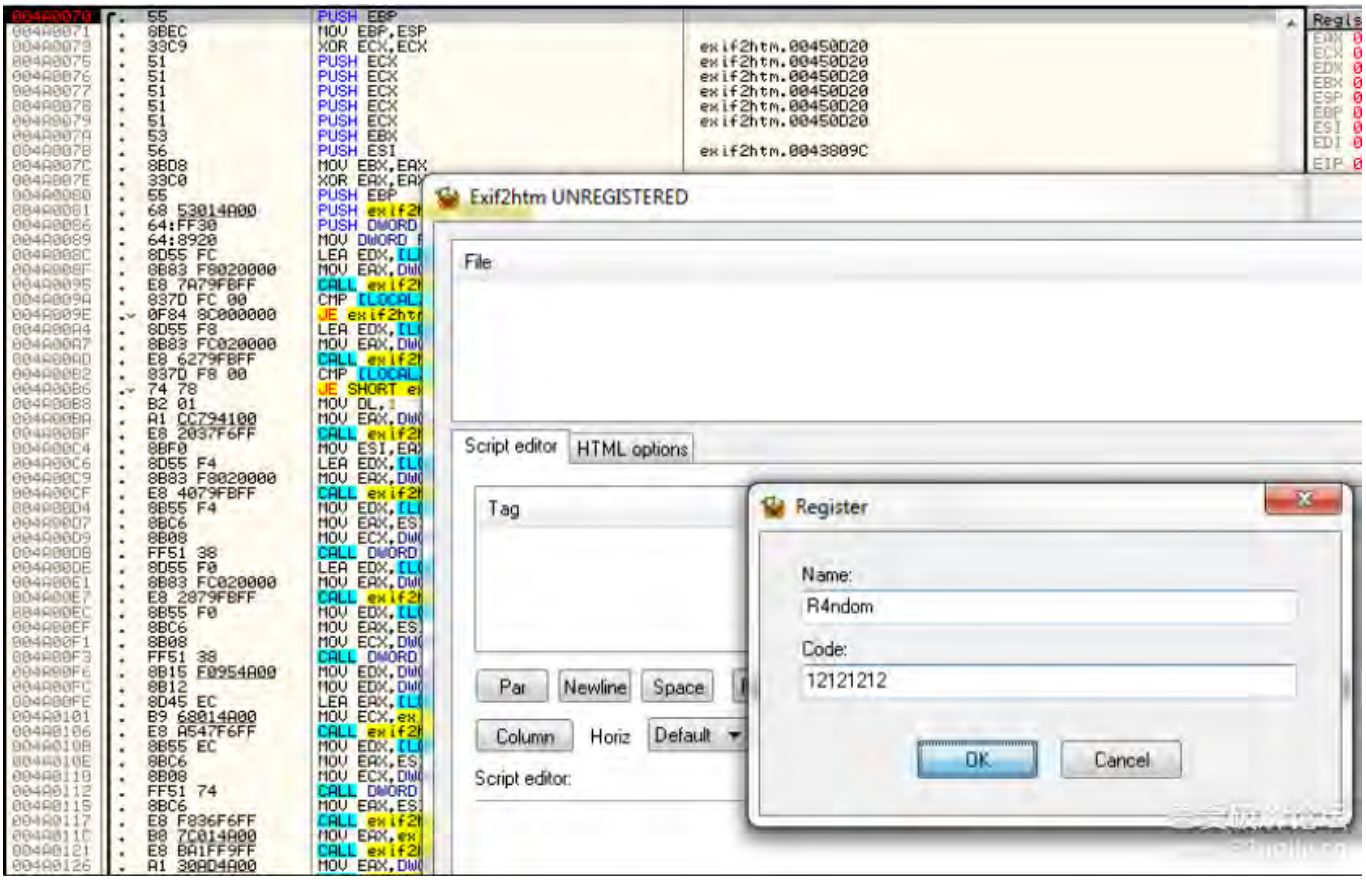


然后使用OD来到4A0070的位置看看

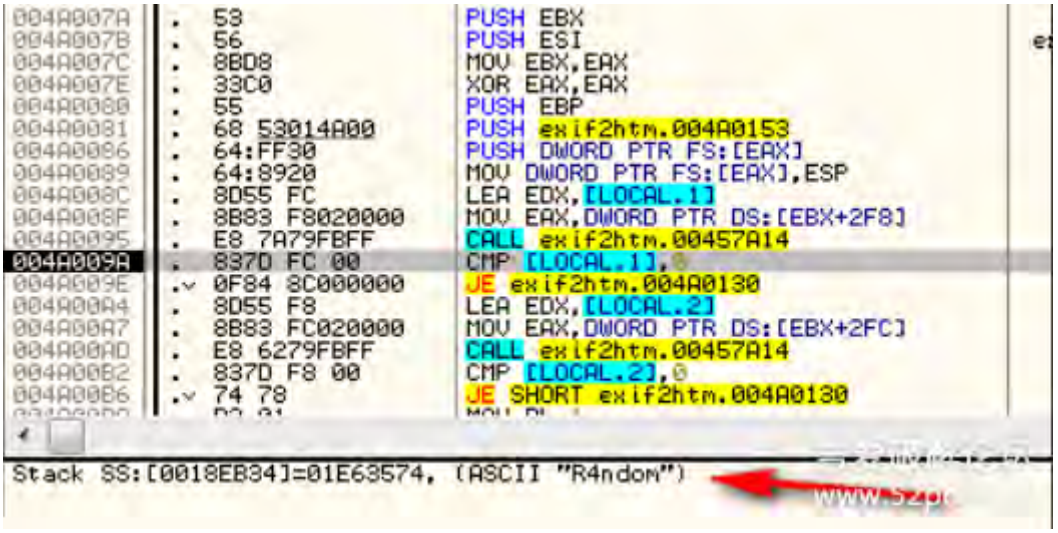
设置断点，让程序跑起来



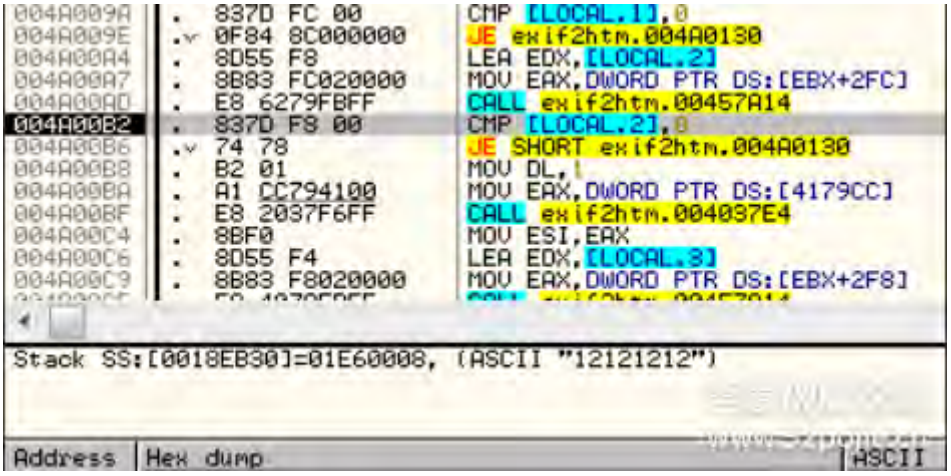
点击注册，输入信息后点击OK，OD断在了我们的断点处



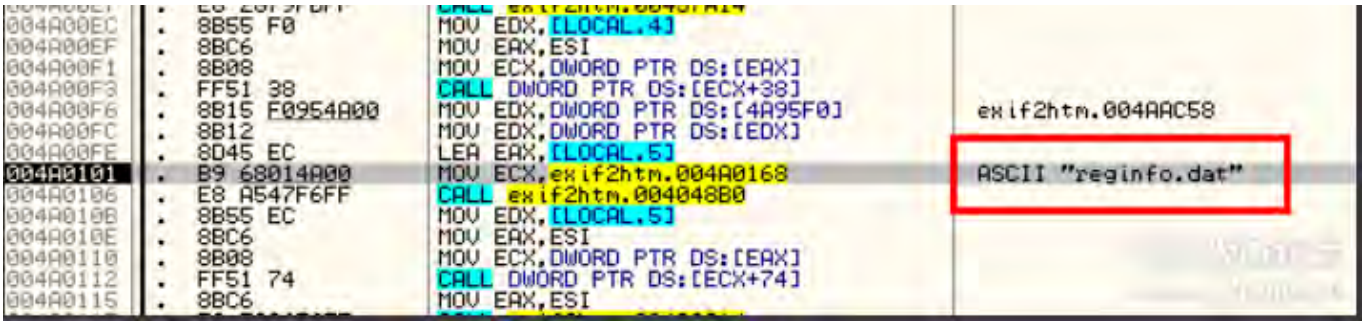
现在我们用OD进行分析，当运行到4a0095处时，我们能看到栈中显示的是我们的用户名，如果你已经步过了就看不到了



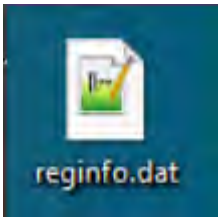
这里对用户名做了什么，99%是做检查，以此作为是否注册的依据，然后继续走，来到4a00B2处，在信息窗口可以看到我们输入的密码或者序列号



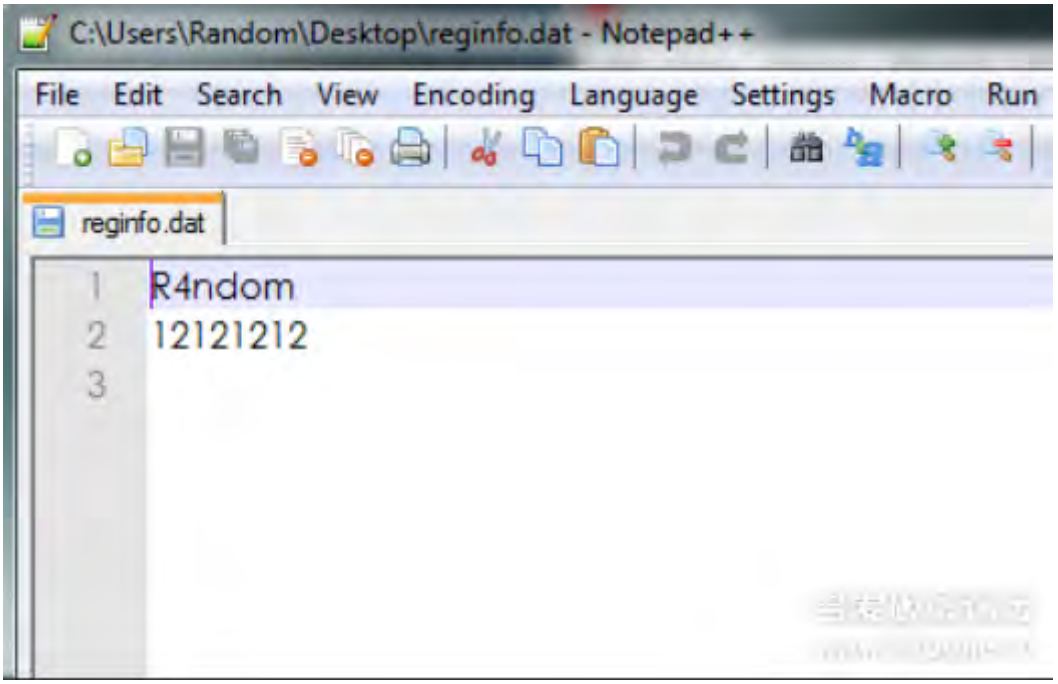
与用户名呼叫同一个call，然后一直来到4A0101处



这里看上去很重要，一个文件被创建了，步过两行来到4A010B处，发现在当前文件路径下创建了一个Reginfo.dat文件。然后步过4A0112这个Call后，文件被创建了

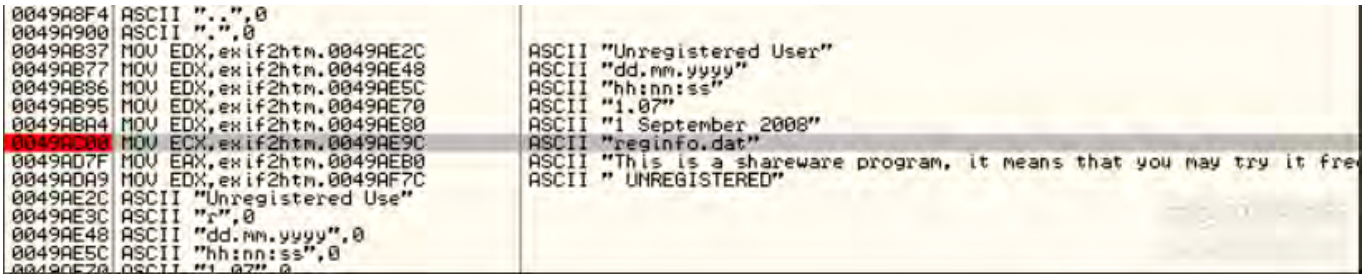


使用txt文件打开这个文件

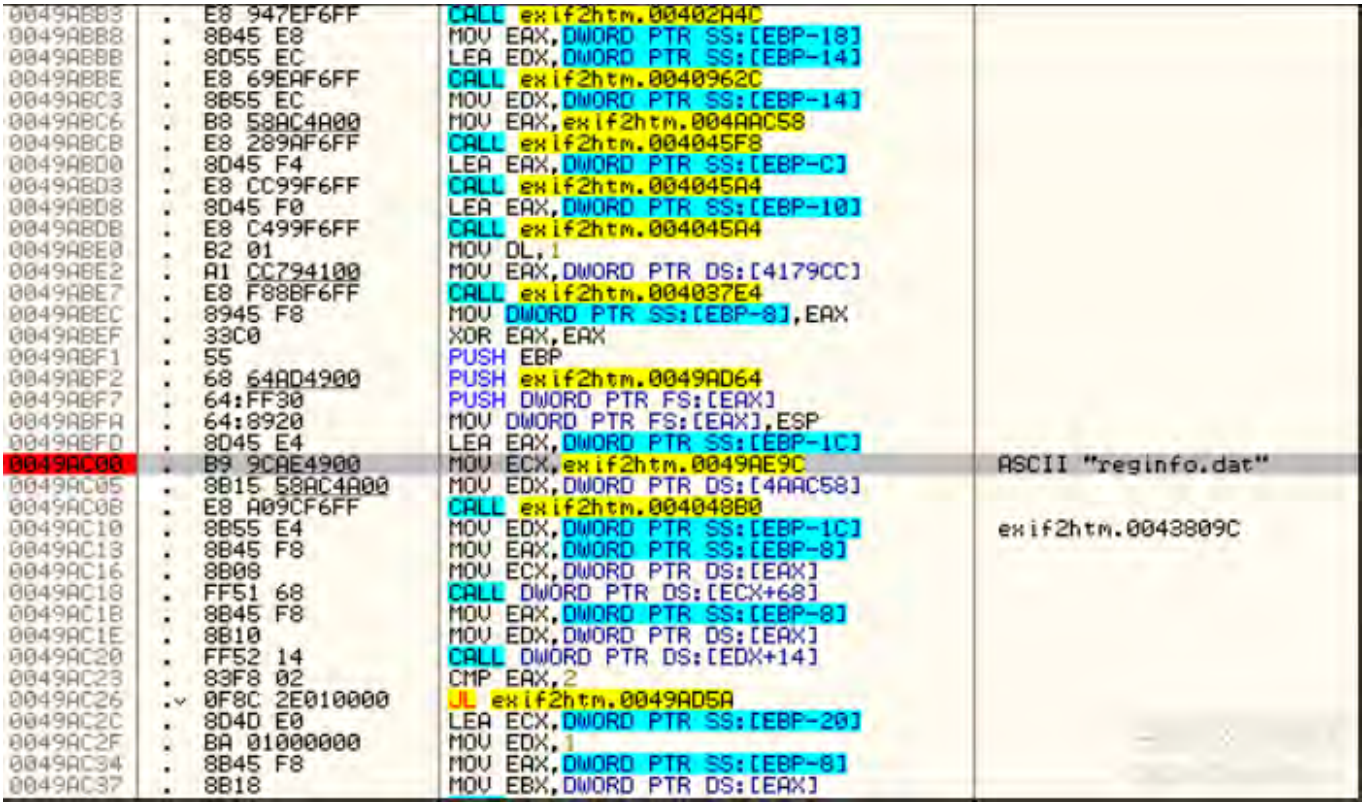


保存的是我们输入的信息

现在在OD中查找字串RegInfo.dat，能找到

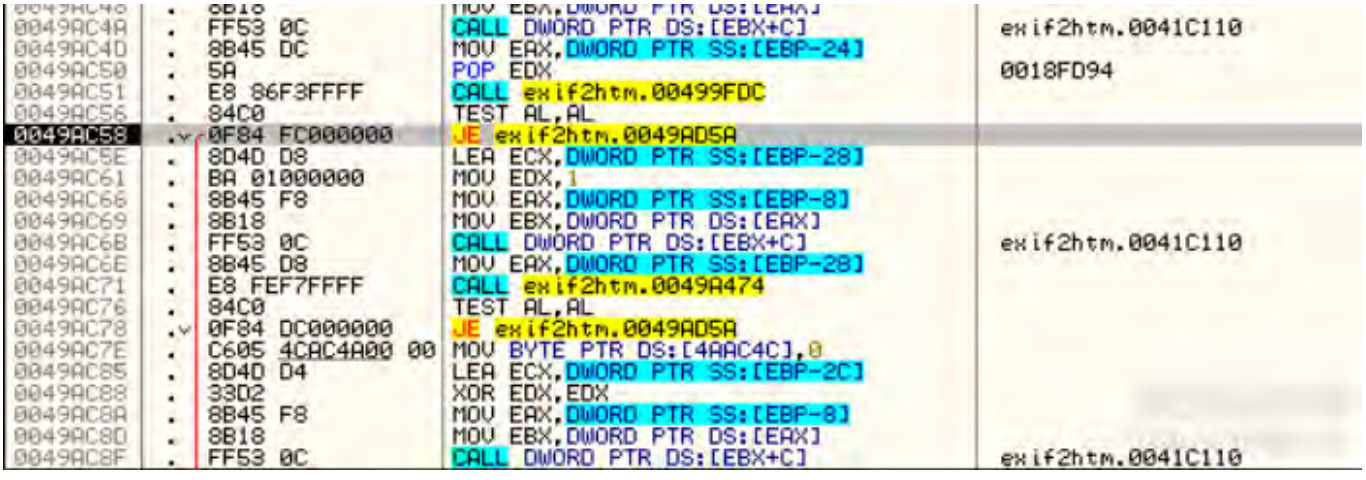


在这里设置断点，重新运行程序，段了这里49AC00

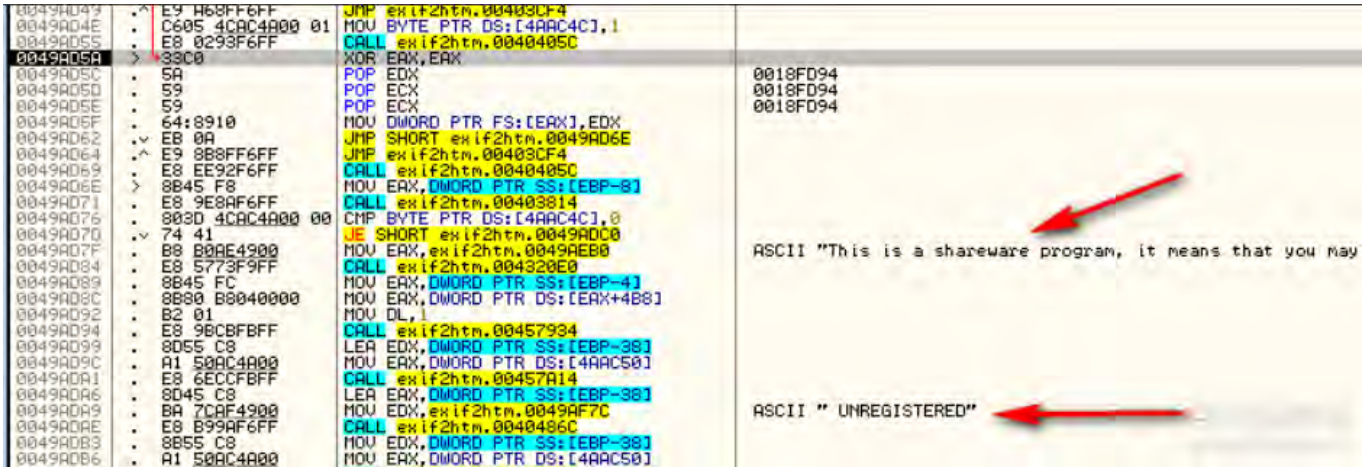


滚动代码上下看看，看是否能发现有用的信息。

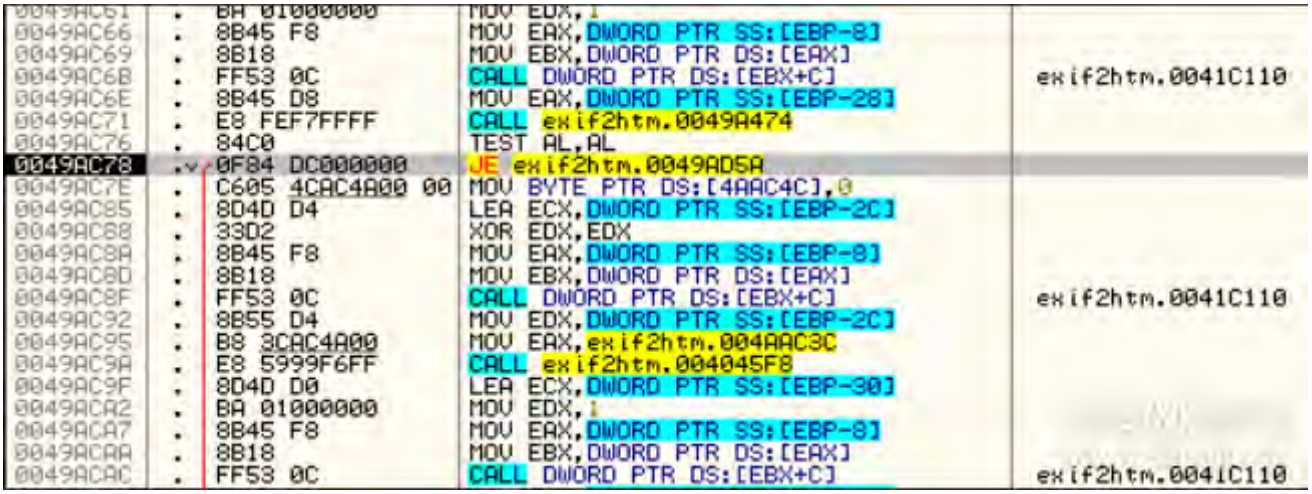
然后继续单步执行，来到49AC58



这里跳走了，看看跳到哪里去了

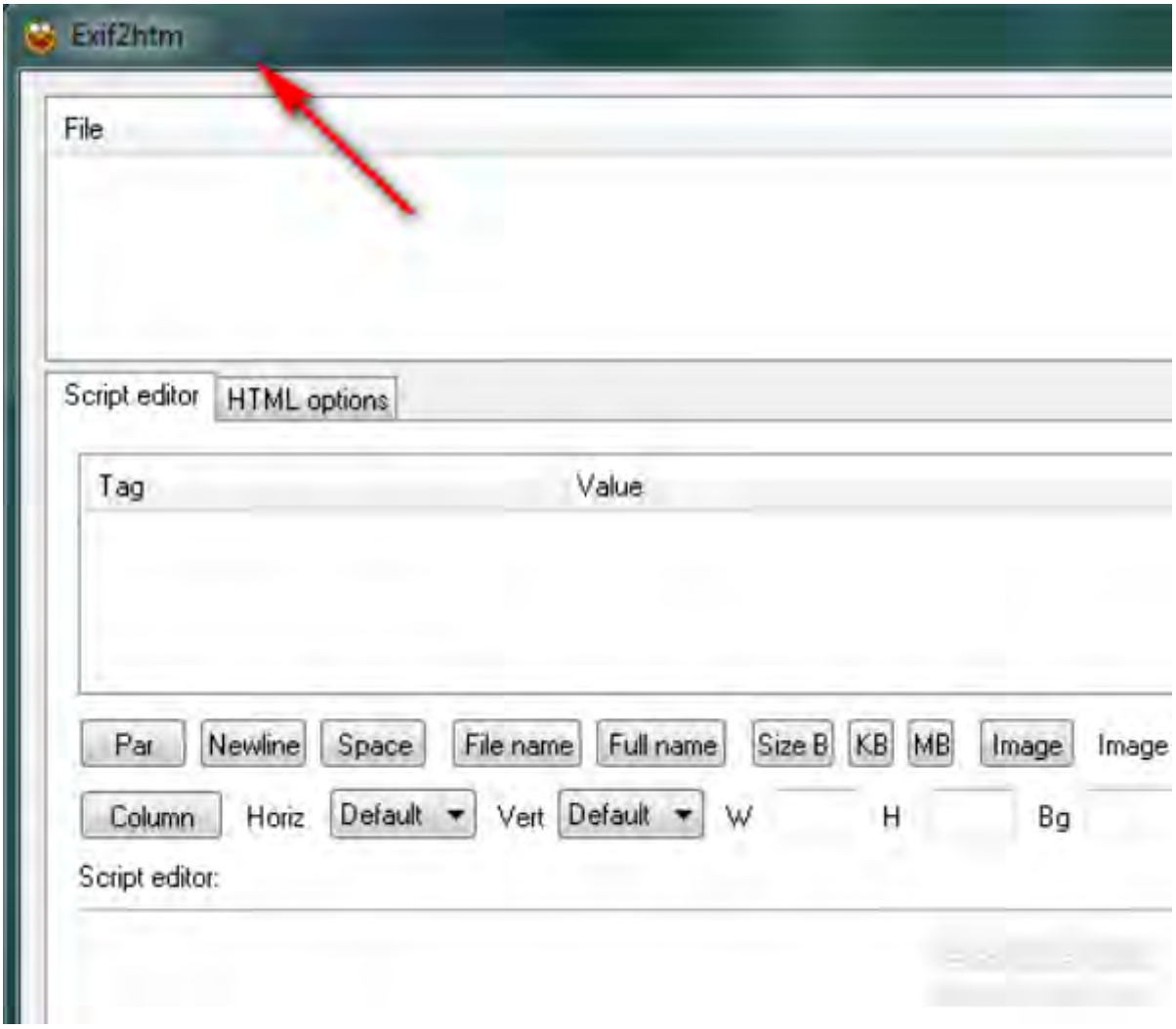


这里看上去不太好，不是我们想要的地方，我们知道怎么改让它不跳走，继续，来到49AC78



这里还有一个跳转，看看到哪里的，和刚刚那个一样，跳到了不好的地方。继续不跳转一直走就会发现，启动程序的弹窗没有了，标题上的未注册也没有了

现在我们能让这个程序使用任何用户名和密码进行注册了，我们成功了



到这里第十七章结束，作者的目的是教我们使用DeDe和ExeInfoPE这两个软件来分析Delphi写的程序，这样对新手来说比较容易。

当然我在没有看教程的时候就使用查找字串参考也能找到入口点，按照以往的教程知道怎么打补丁了。但是还是按照作者的意图进行转述，作者描述的比较罗嗦一点。我就直接精简了，因为如果从第一章一直看下来的朋友一定明白我说的是什么。

后续章节会抽时间继续，直到完成。

第十八章：时间限制与硬件断点之间的博弈

翻译都是我理解的方式进行描述，可能和原文不一致。

本教程中文版只在吾爱破解论坛 首发。

转载请注明来自吾爱破解论坛@52pojie.cn

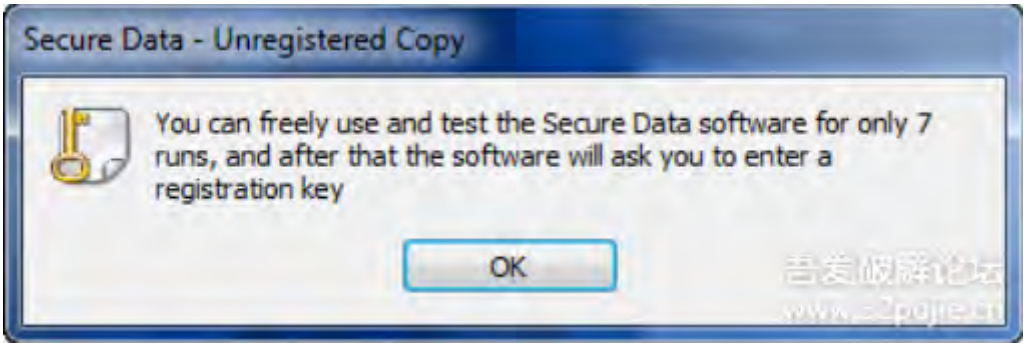
正文开始

试用时间和内存断点

解释：使用时间通常来说是一个程序提供一段时间让用户使用，超过这段时间后程序将不再工作或者减少功能工作，一般会给出30天的试用。当一个逆向工程师在破解这一类的程序时会去找到注册码，如果我们比较懒的话，就找到程序中的试用时间，让这个时间无限话。

教程中，为了保护作者和软件，我下载的软件在CNET网站上的下载次数只有2次，估计是作者已经不再提供维护和支持了，本程序的原名为：Secrre Data-Hide a File into an image.exe，而我将它改为：SecureData.exe。而且不含附带的DLL文件，因此该程序并不能真正的工作。但是足够本教程的教学。

打开程序后会弹出窗口



显示了可以试用的次数，点击OK后来到主界面

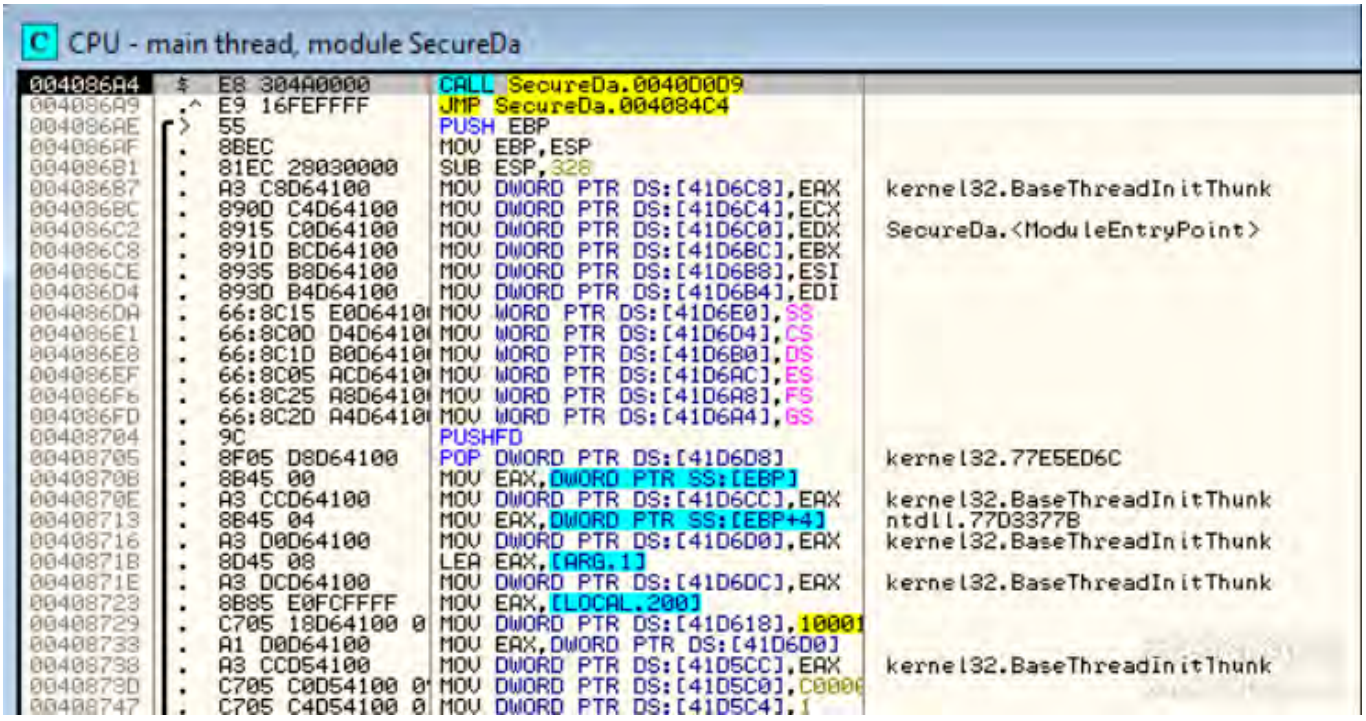

Secure Data - Hide a file into an image
Features
Hide File
Get Hidden
About
SECURE DATA
Protect your data using a secure means
SECURE DATA hides any file type (doc, pdf, text, jpg, bmp, gif, zip, rar, mp3, avi, ...) into an image file. This software allows you to protect your critical and private data in a form that cannot be noticed by anyone.
It also uses a user-defined password when hiding and retrieving the file. The software encrypts (codes) your file before the hiding process to get you additional security, so your hidden data will remain secure even if it is analyzed by hackers or attackers. Secure Data is a powerful application for encryption and hiding processes and it was designed carefully to be fast and simple to use. Featrures:
1. Encrypting (coding) your file.
2. Hiding your encrypted file into an image file, so it cannot be noticed by anyone.
3. Usage of a user-defined password to hide and retrieve your file.
4. Decrypting (decoding) your hidden file at the retrieving process.
5. Saving your hidden file as an independent file at the retrieving process.
6. Fast and easy to use.
Note: after hiding your file into an image then please do NOT repaint, resize or alter the image that contains your hidden file using image processing or painting applications, because that will corrupt the hidden data.

点击About后

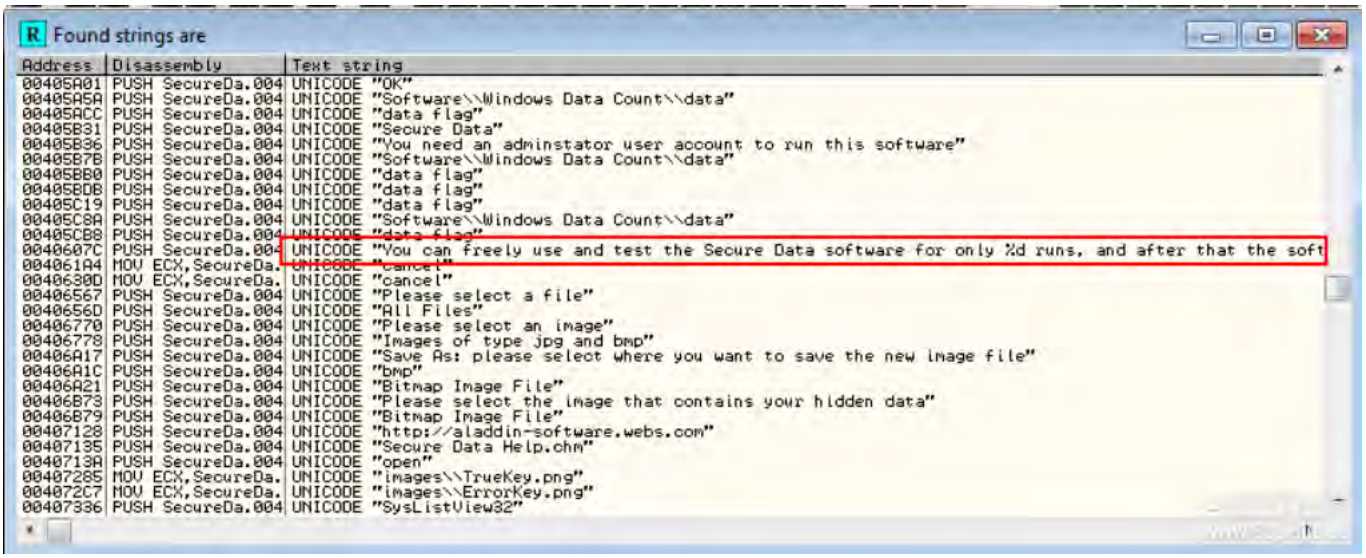
输入Key后，点击Register出现

好了，目前我们已经知道程序的大致流程了

让我们用OD来打开程序



搜索字串后，我们可以看到



我们没有看到和试用信息一样的字串，但是注意红色框

试用期

逆向工程师必须要知道的一个重要信息，在程序中必须要保存试用期的次数或者天数，这样来说试用期一定是保存在什么地方，通常情况下是保存到文件或者硬件中。

大多数时候，保存这个数据的地方很容易找到，只是逆向工程师会把自己往复杂的方向去想。最简单的方法就是看字串和搜索注册表或者文件路径。比如：

Software\\AppName\\Key

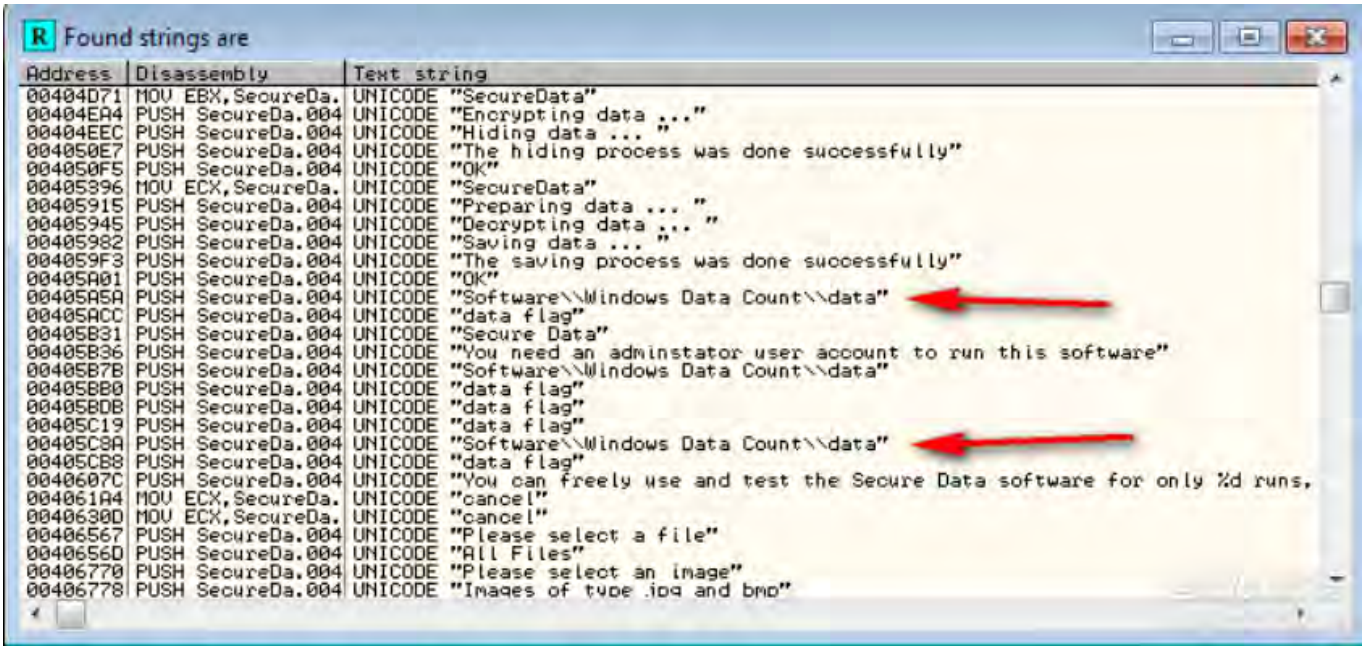
或者

AppName\DataFileName.ini or AppName\DataFileName.dat

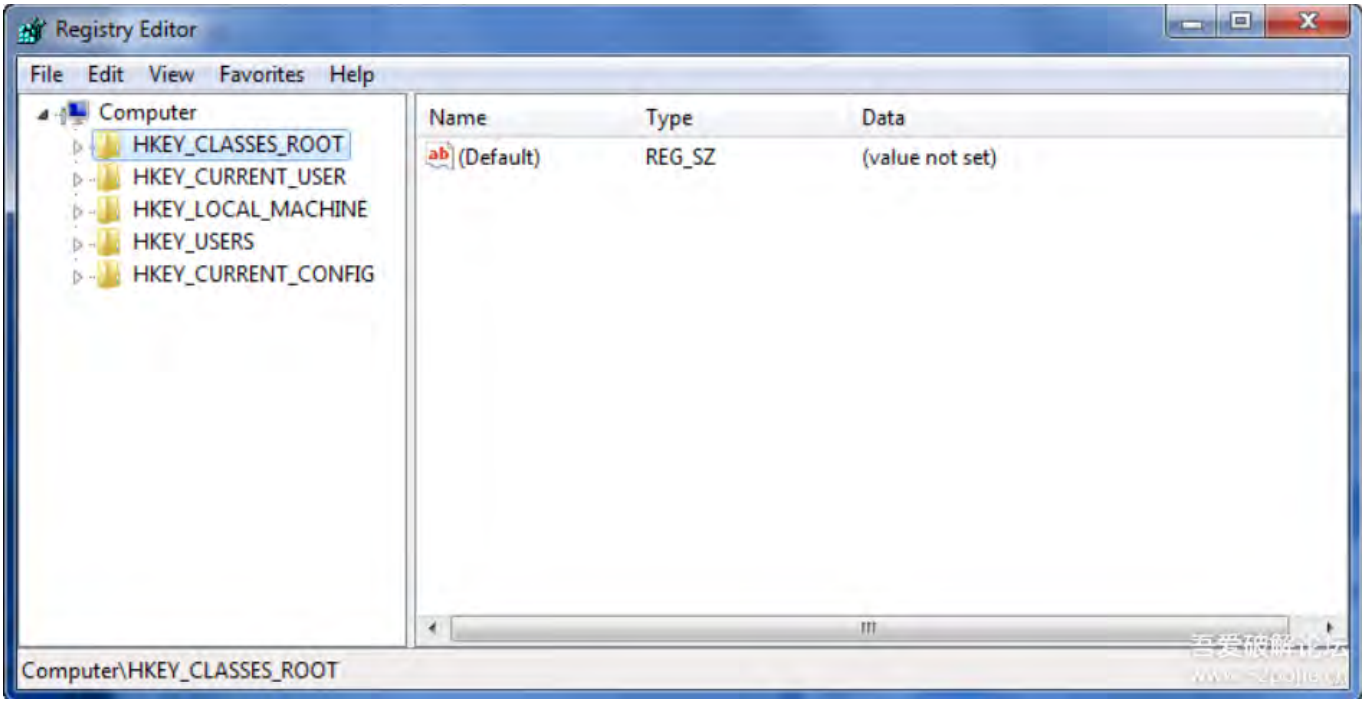
多数会采用引用window 系统变量，如%WINDOWS%，这表示在Windows目录。

当然，如果程序不是很大，你还可以一个Call一个Call的看。或者注意一下windows API，CreateFileExA，RegSetValueExA等等，确保数据是保存到文件还是保存到注册表中。而本程序使用了2套做法，在硬盘上创建了文件并且将它隐藏。

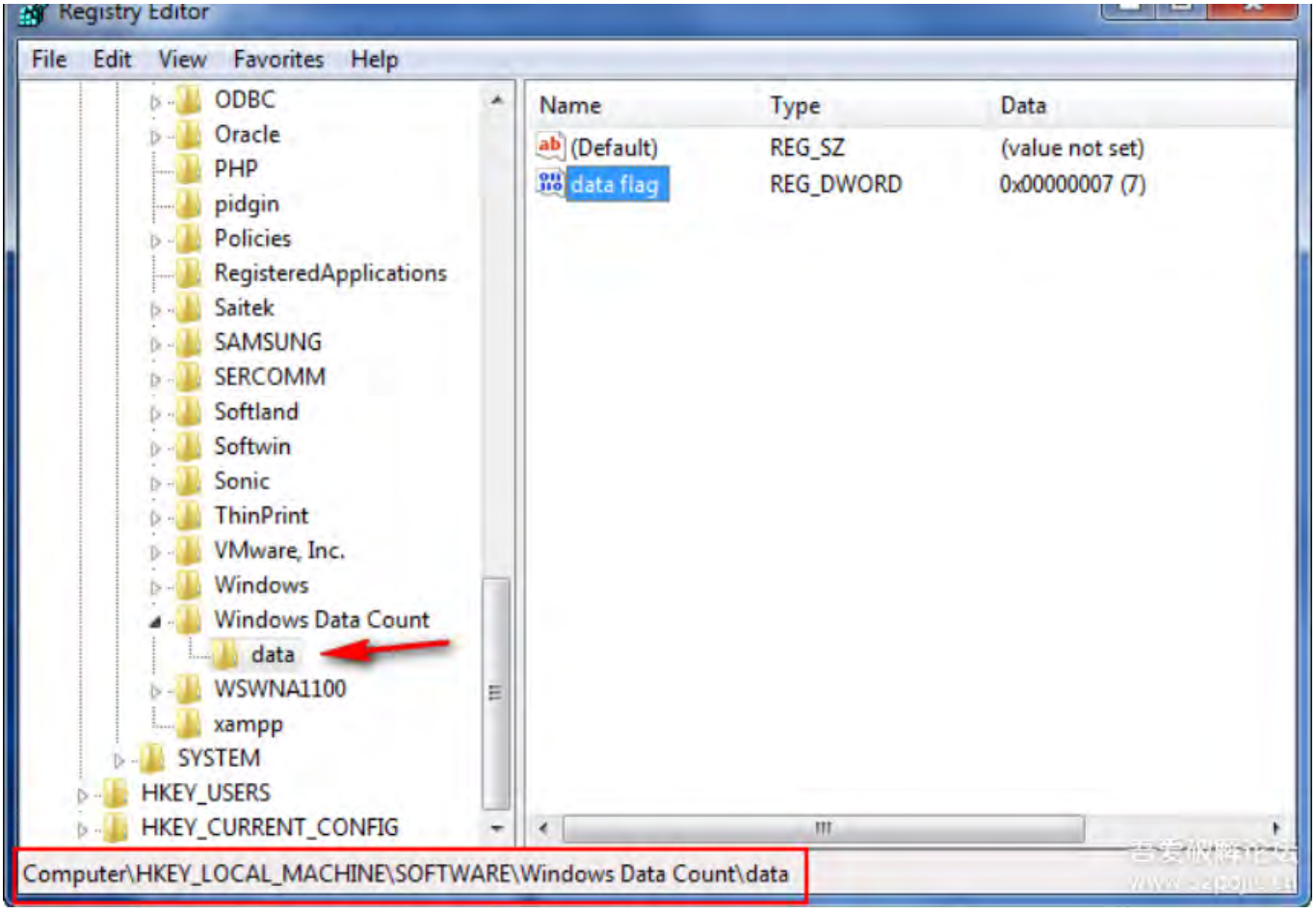
看会字串搜索结果



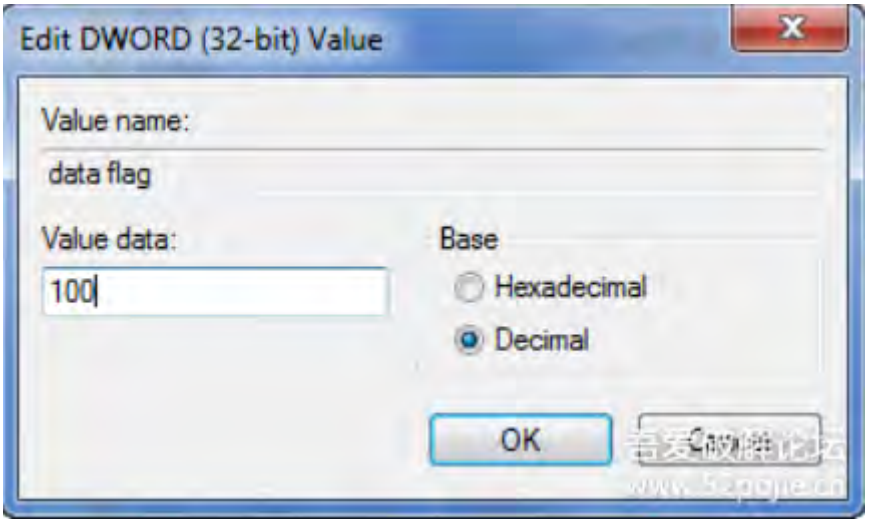
这里不会讲解注册表的树形结构，我们直接打开注册表，使用Window +R键，输入Regedit，打开注册表



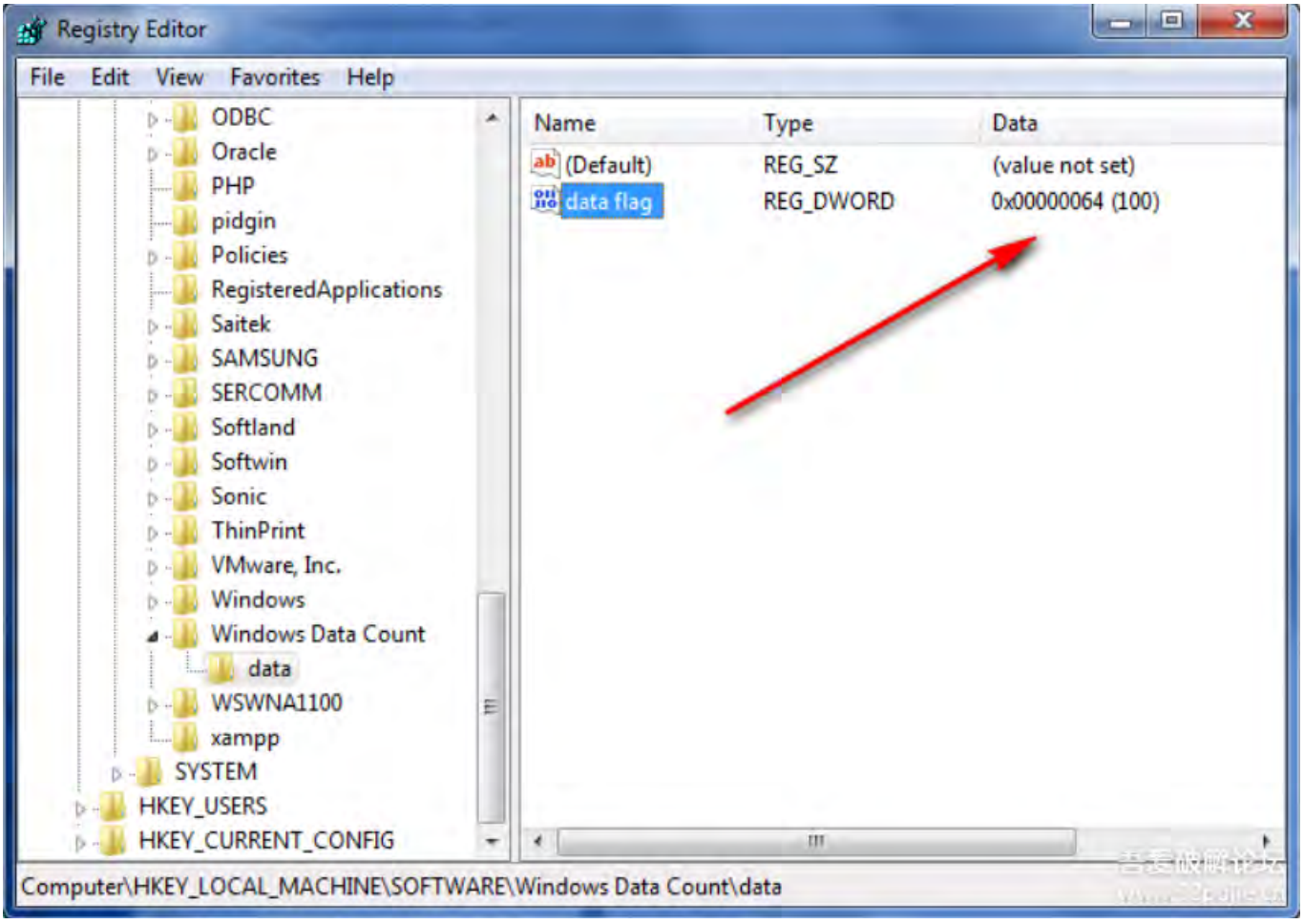
然后打开HEKY_LOCAL_MACHINE，找到字串Windows Data Count，然后可以看到



我们看到Data Flag显示为7，我们把这个值改为100



然后保存后显示

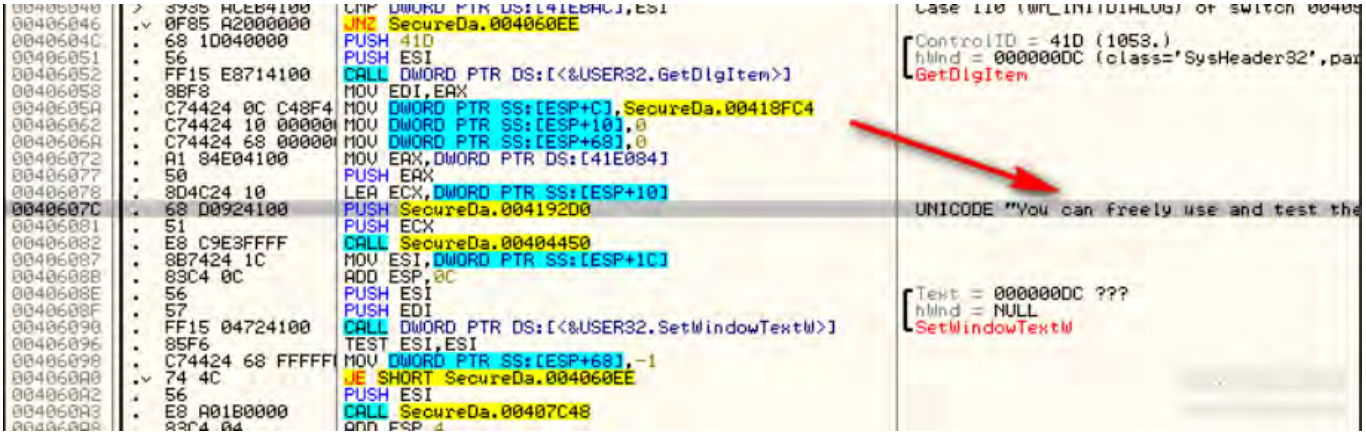


现在重新打开程序

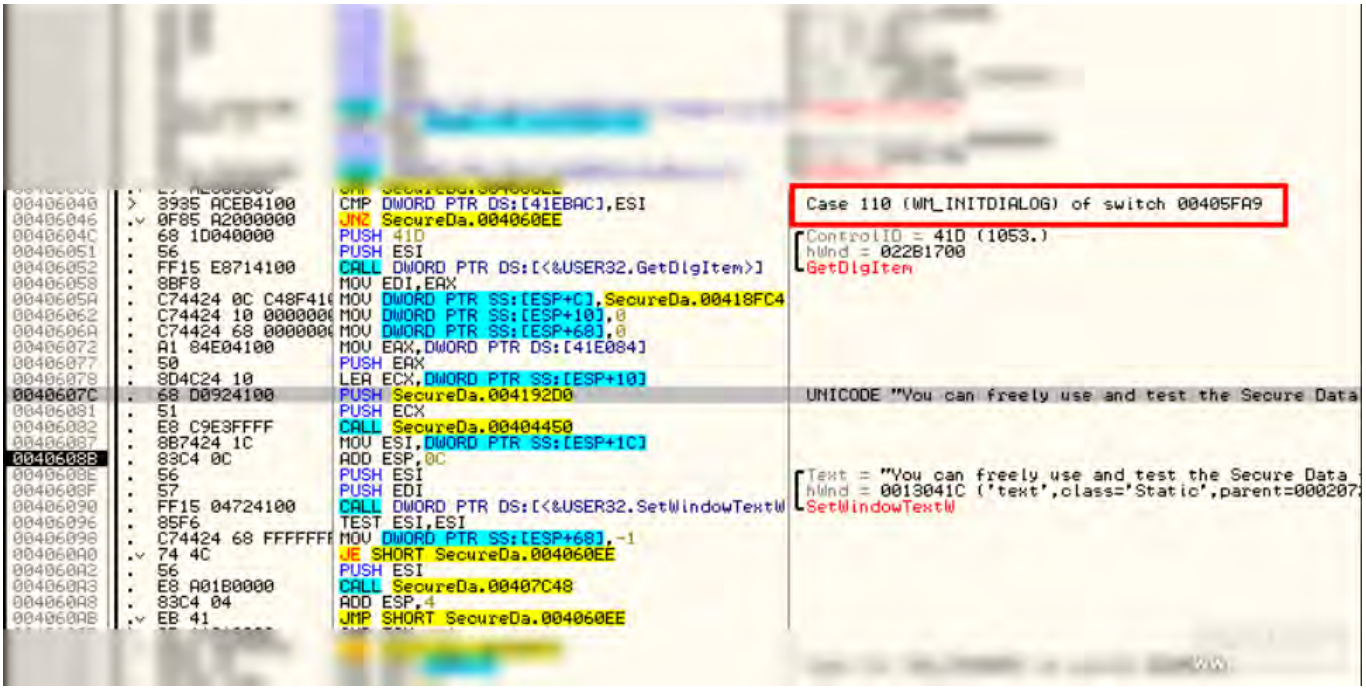
看上去修改成功了，这比想象中容易

分析程序

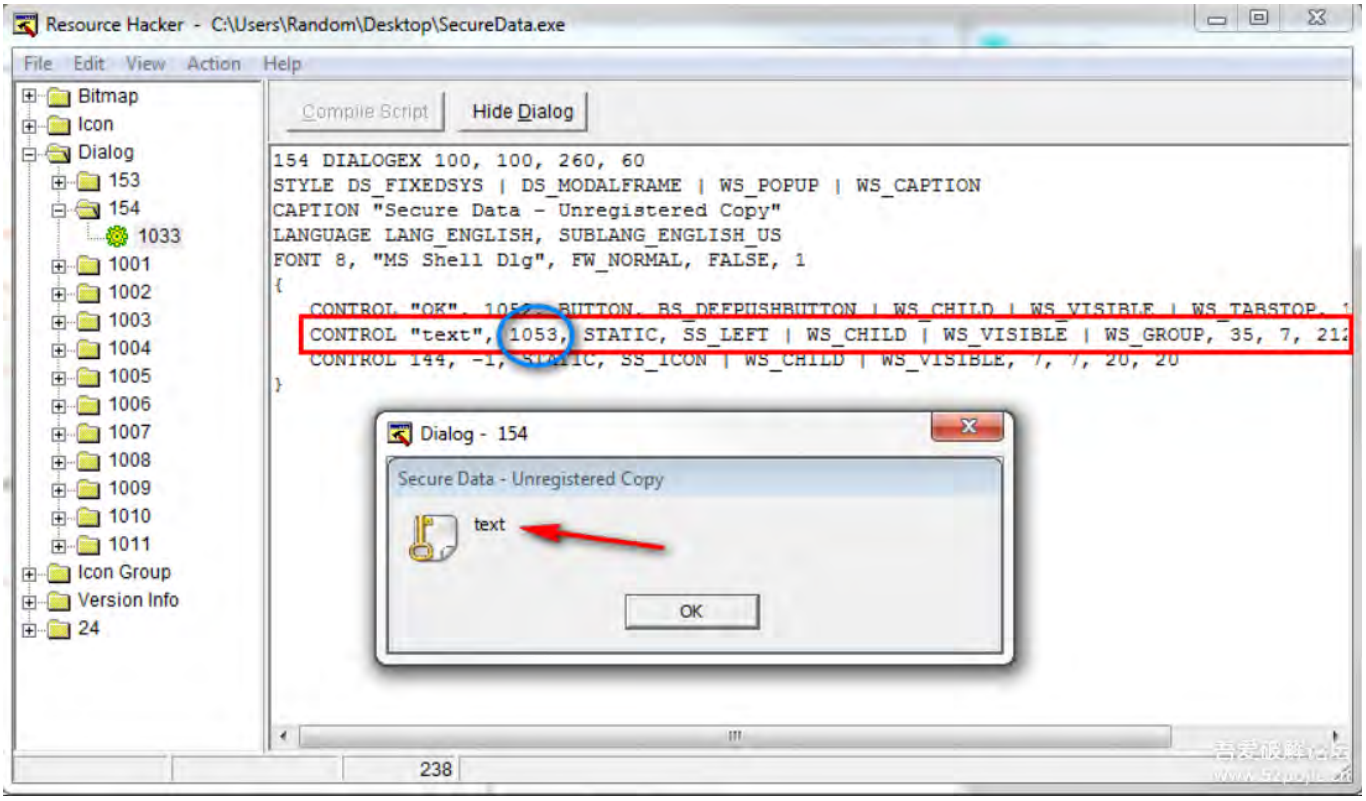
另一种方式就是我们通过修改程序代码达到同样的效果。这样做的好处就是不用每次快到时间就去修改注册表。记住，当我们运行程序的时候请记住还剩下的次数，让我们去看看这个地方程序到底做了什么，首先这个字串开始于406078



这是一大段程序，我们可以看到这是windows程序的初始化消息

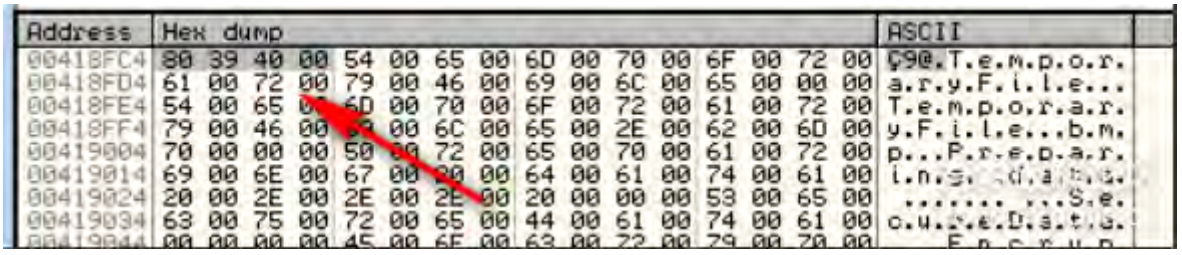


这里先把一个ID为0x41D的窗口句柄作为GetDlgItem的参数，让我们用Resource Hacker看看资源



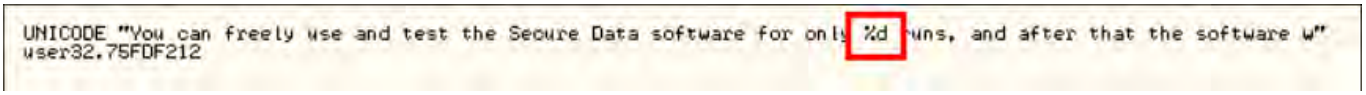
查看GetDlgItem的用法后，我们把断点设在40604C处，然后重启程序，来到我们的断点处，单步分析，发现GetDlgItem的返回值保存到EDI中。

下一条是把内存418FC4的数据放到ESP+C中，跟踪这个地址我们看到数据是403980.如果你继续跟踪就会发现这是一个Callback。我们假设这是一个对话框。



再往下两行，把内存41E084保存到eax中，而这个值为8，这个值和试用次数是匹配的。这就是我们要找的。

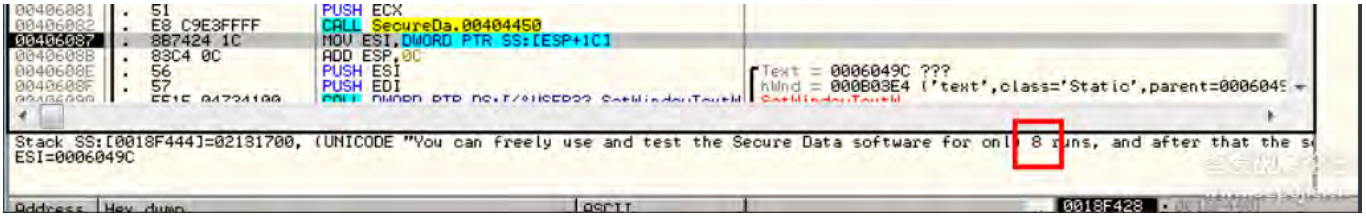
最后，我们在去看看字串



这个看上去是C语言中格式化，如果你熟悉C/C++语言的话。

printf(“My IQ is a whopping %d”, 18);

我们就看看这个%d是在那里被赋值的，来到



然后程序跑起来，我们看到试用信息，点击ok按钮后来到主界面。

打补丁

你先想到的是为什么不直接在地址406072的地方把试用次数改大，比如

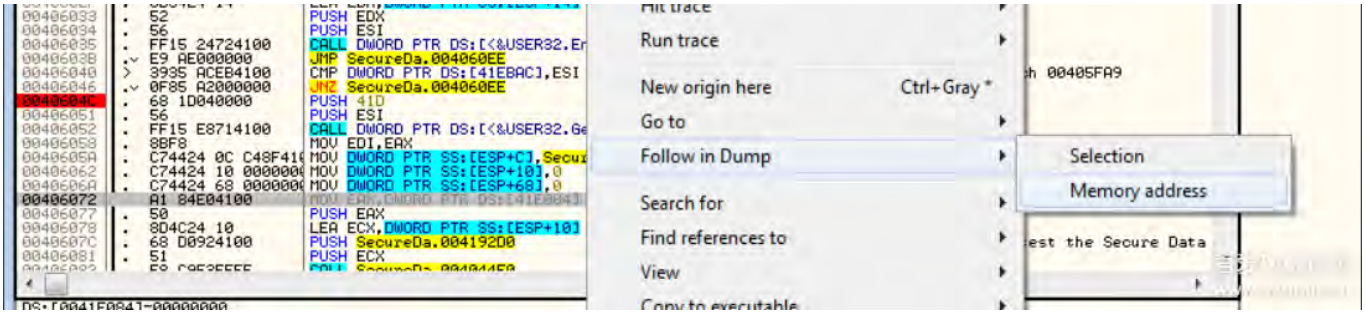
MOV EAX, DWORD PTR DS:[41E084]

改为：

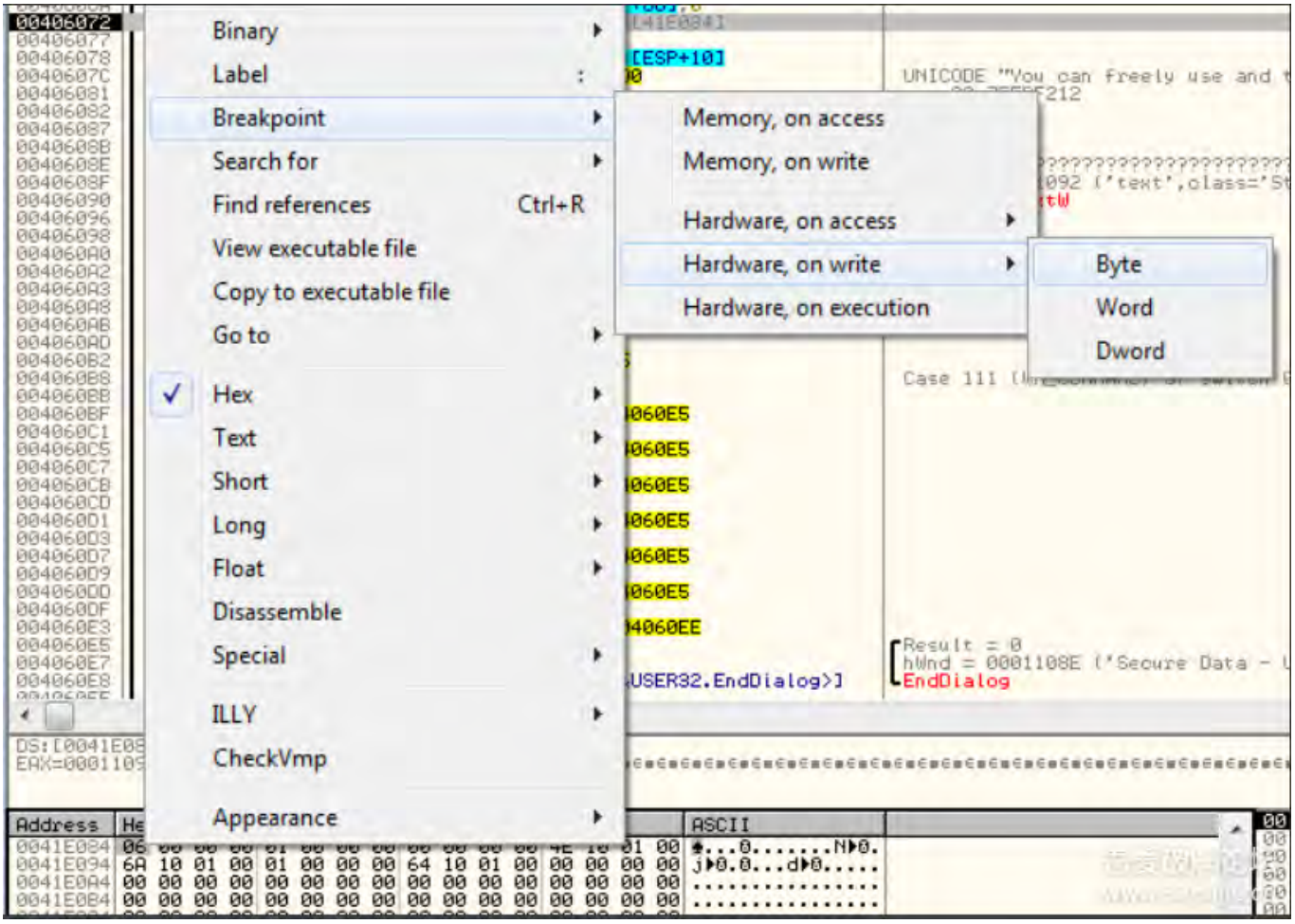
MOV EAX, 99

但是不能这样做，这样做虽然程序会显示试用次数很大，但是其他地方会检测到数据小于1时，程序将不再工作，所以我们必须要找到程序去改变这个值的地方，在程序修改后我们再进行打补丁（修改）。

所以我们要在这里设置硬件断点



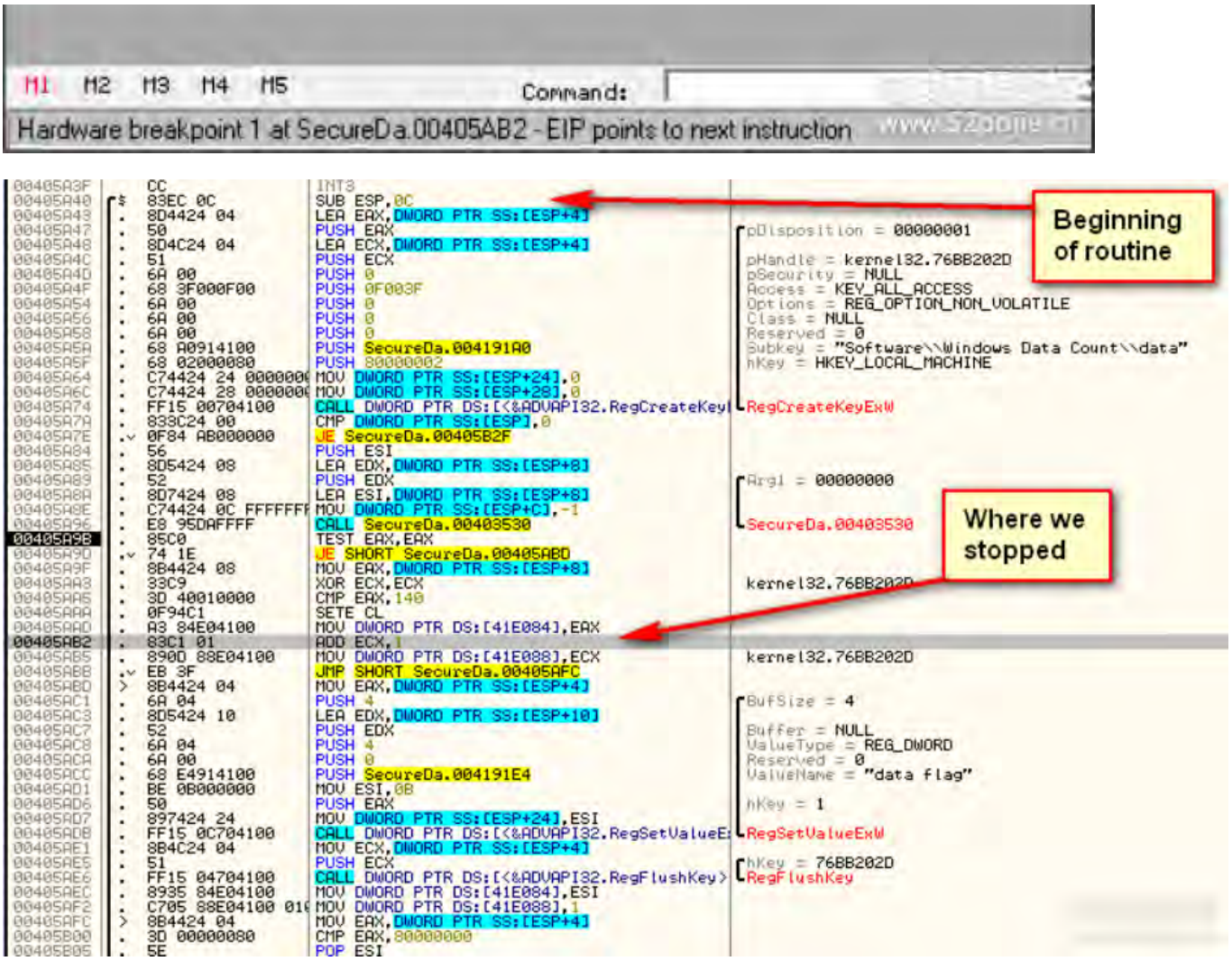
就是这个内存被写入的时候断下来



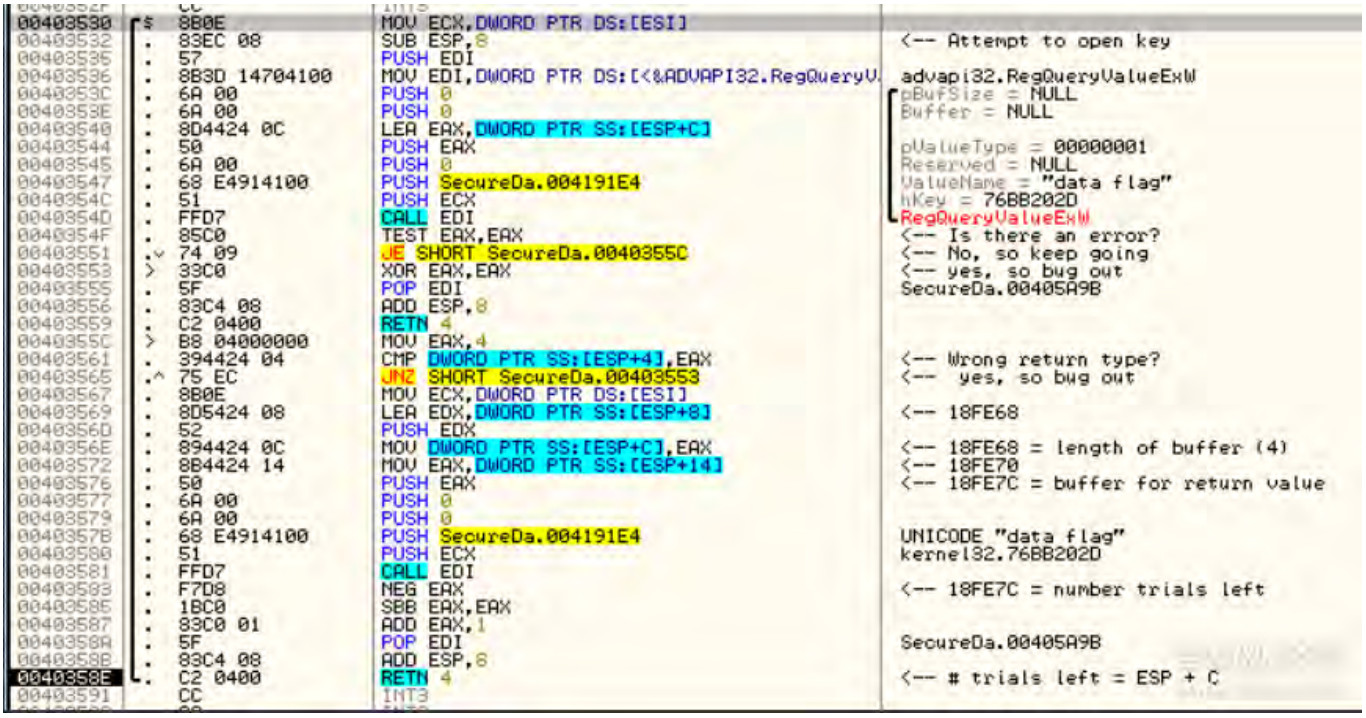
现在我们只剩下6次试用次数了，我们设置了硬件断点，不论在任何地方对这个地址执行写操作，我们都会断

下来。

重启程序，OD会断在我们设置的硬件断点处

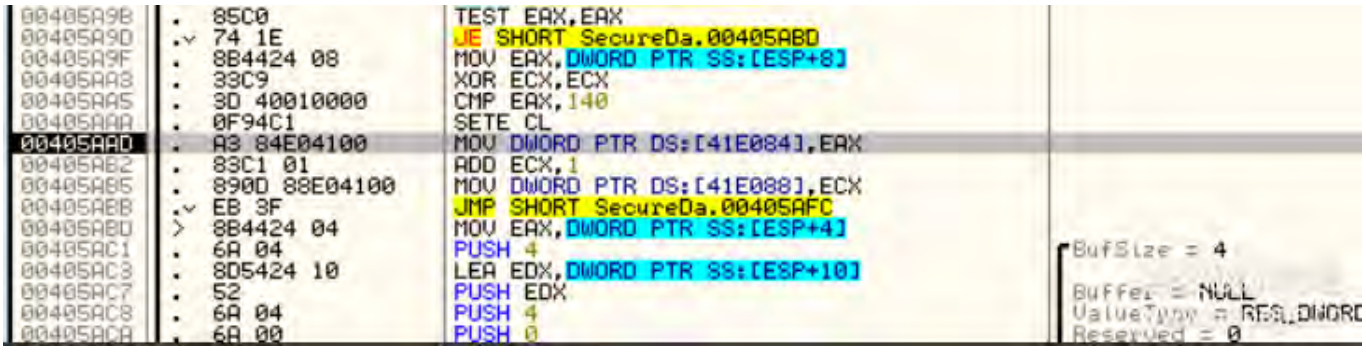


看图片顶部，我们可以看到注册表不是创建一个键值，而是打开。然后判断是否能打开，如果不能打开则跳到坏消息处。这就说明需要管理员进行操作，如果不出错405A96的call就会把值读取出来保存到ESP+C中，如果有第二个则保存到ESP+8



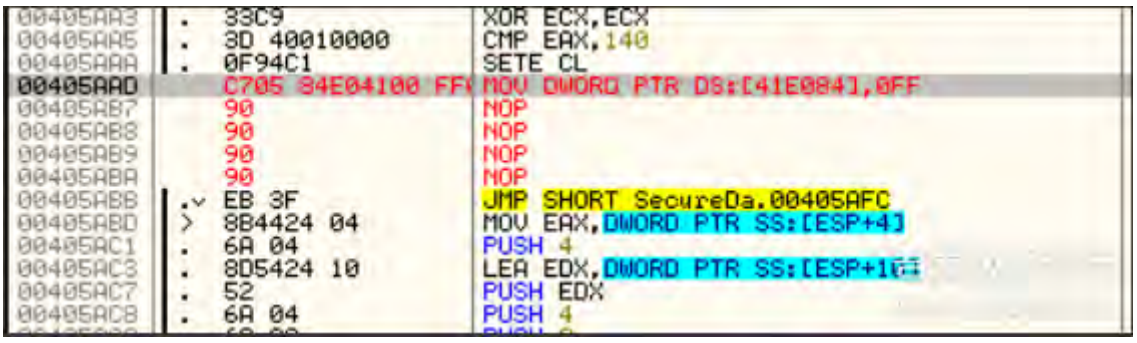
我已经分析了代码并给了注释，建议你自己分析代码

我们发现这个值（6）已经搬到了405AAD内存里面了

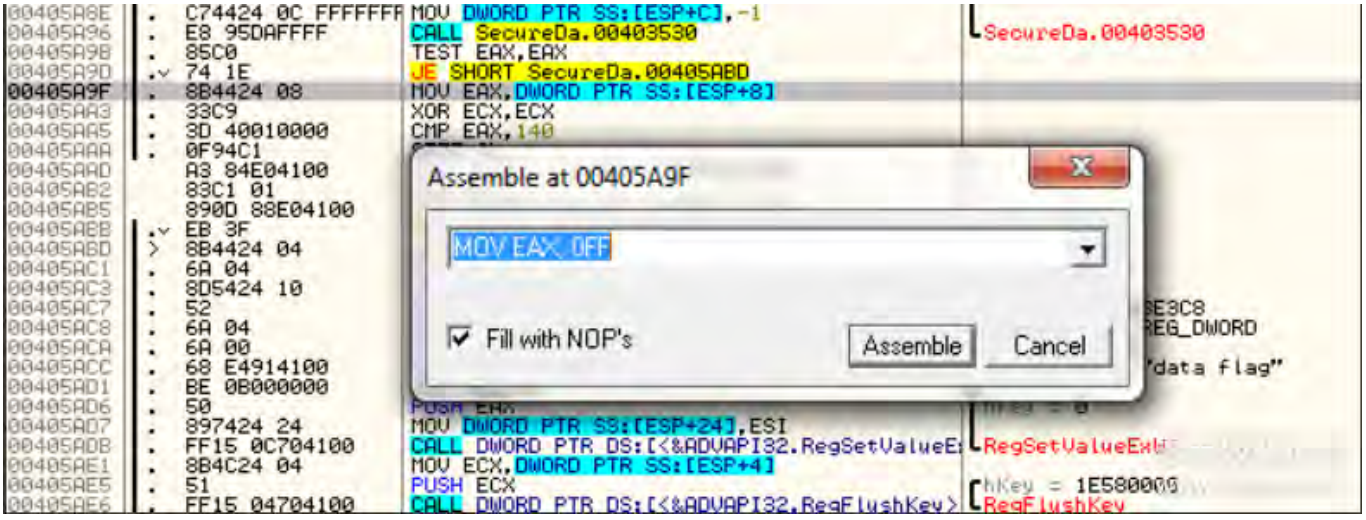


最后，我们在检查一下其他值，然后关闭注册表控制。

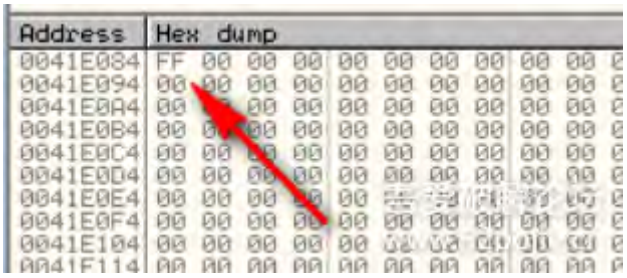
那么问题来了，哪个位置是做补丁最好的地方呢？回看代码，发现在405ADD被赋值后就没做过任何变化了。如果我们把405AAD修改为MOV DWORD PTR DS:[41E084], FF,这会导致程序代码不正常。因为会被删除下两条命令。



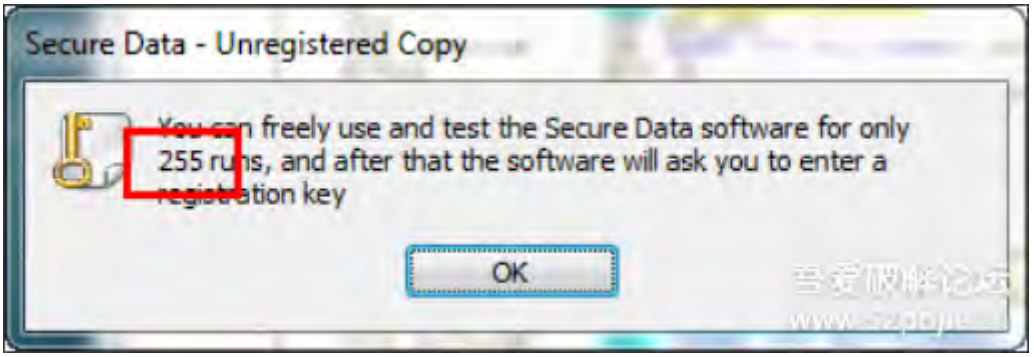
那么就在前面405A9F处进行修改吧



然后我们可以看到步过这个命令后，内存41E084的值变成FF了



然后跑起来，试用信息为：



现在我们的程序不管打开多少次，这里总显示为255了。

虽然说去掉NAG弹窗，直接进行破解是最好的，但是我会结束这次的教程。不去做破解的原因有2，1是通过逆向，你知道程序工作原理以及设计程序时避免这样的检测。2有时你不能破解一个程序，对你来说是下一个好事。

第十九章：打造补丁生成器

翻译都是我理解的方式进行描述，可能和原文不一致。

本教程中文版只在吾爱破解论坛 首发。

转载请注明来自吾爱破解论坛@52pojie.cn

正文开始

此次教程主要讲述补丁程序，补丁程序就是在程序中找到了补丁位置以及补丁方式，不修改程序，通过制作补丁程序来达到我们想要的效果。（注册或者显示好消息等），逆向工程师就会把这些补丁放到另一个副本中。通常情况下补丁程序是个小程序，是用来修改原始程序的，当运行补丁程序后，会去按照你制作的补丁对原始程序进行补丁，然后原始程序就能按照补丁后的效果运行了。

举个例子，现在你在网上下载了一个有试用期的程序，当试用结束时，你找到的补丁并且应用后，试用信息没有了。通过这次的教程我会教你怎么制作补丁程序，然后使用补丁程序精准的打上补丁，这样你就可以把这个补丁程序放到网上，其他用户则只需要到软件官方下载程序，使用你的补丁程序进行打补丁后，程序就能正常使用了。

另一种类似补丁程序是把程序进行加载，在没有打补丁的程序上是不能进行加载的。

此次教程中我们进行补丁程序的名字叫Saturday Night Creackme。这个程序进行破解是很简单的，我将用dUP2进行补丁程序制作，还需使用到CFF Explorer。当然最后我会提供这两个程序。

打开程序

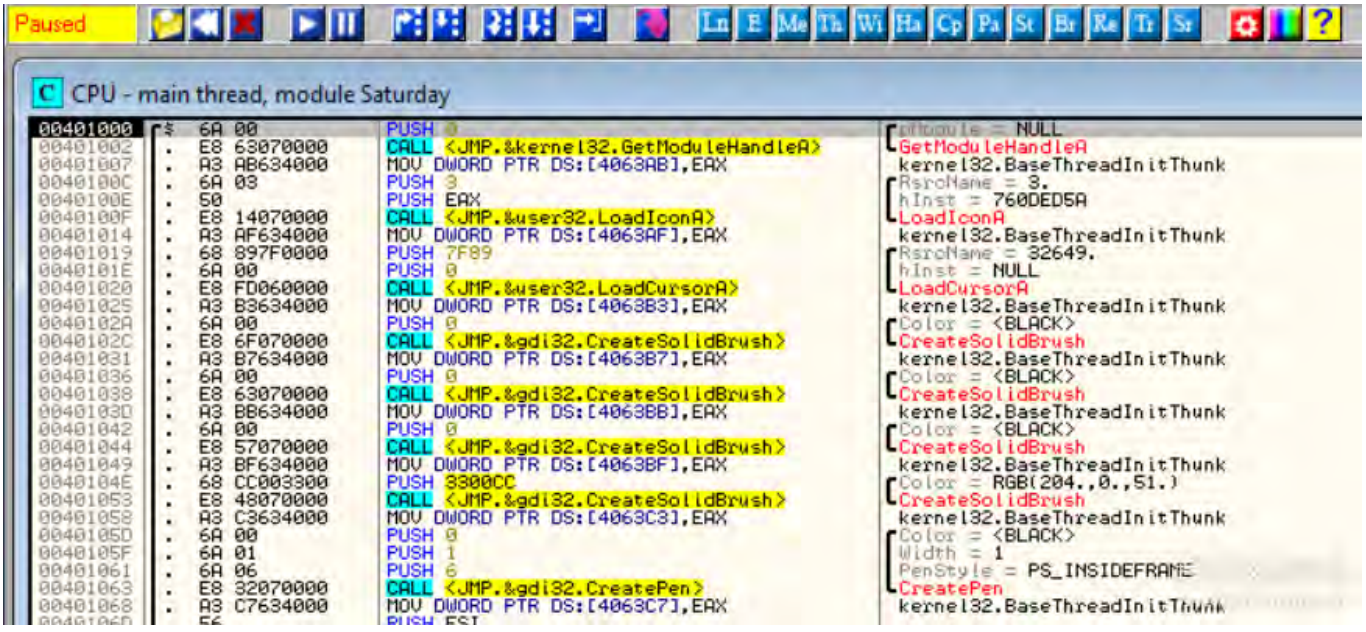
当我们打开程序后，我们可以看到一个彩色图像

如果有音箱，请打开，程序会进行音乐播放。

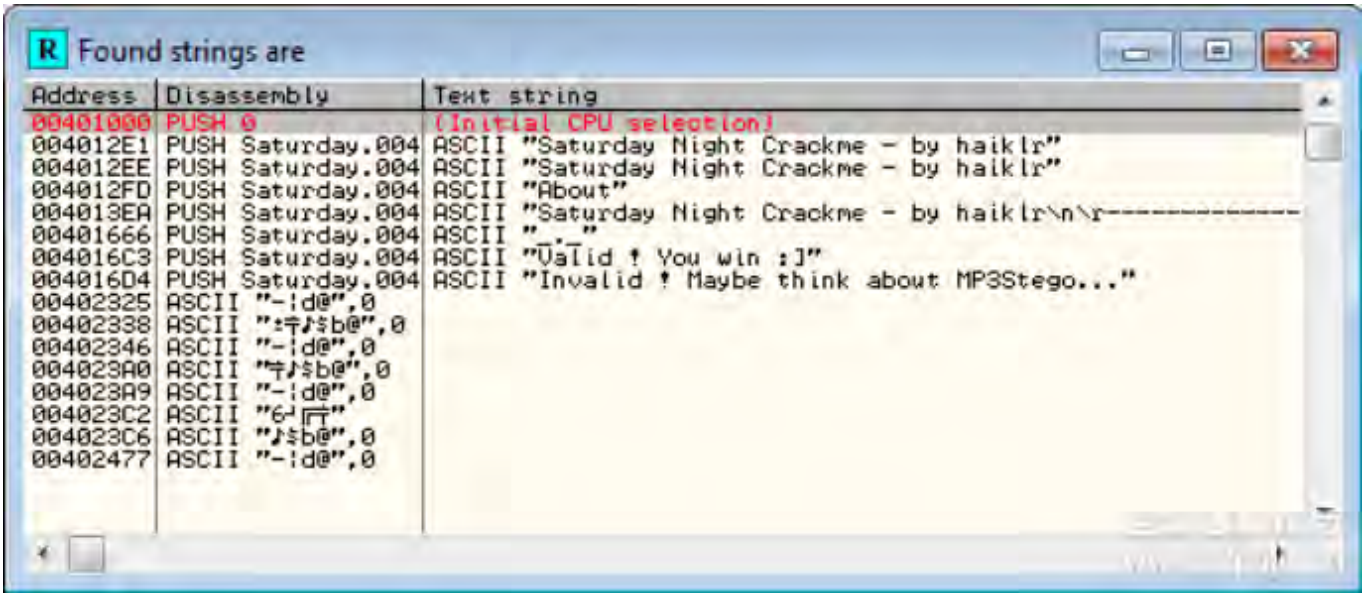
输入注册码

然后点击 dance！，我们可以看到

对程序进行补丁，使用OD打开

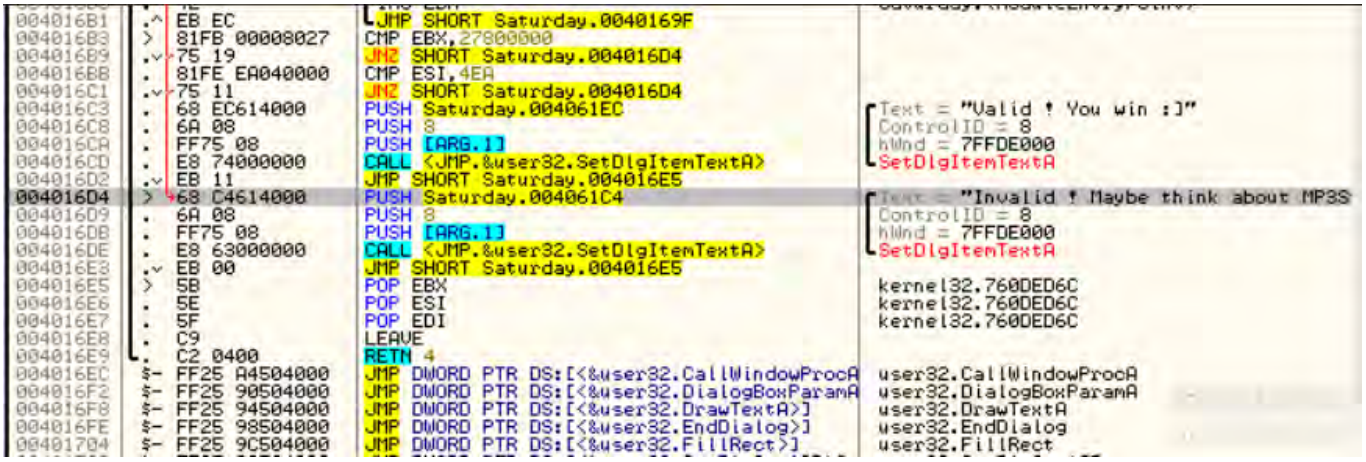


查找字串参考

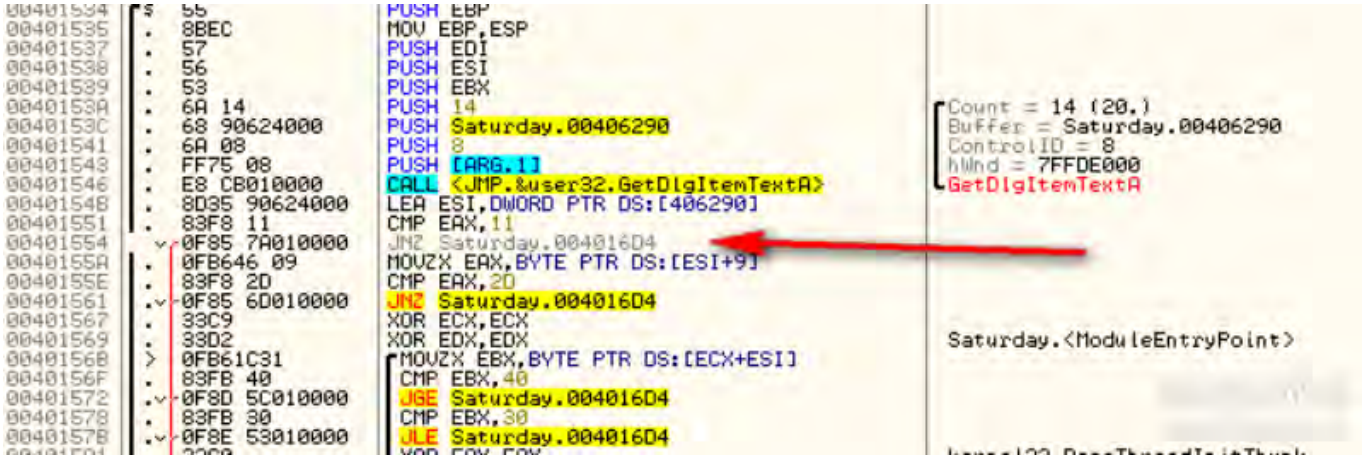


可以看到好消息和坏消息

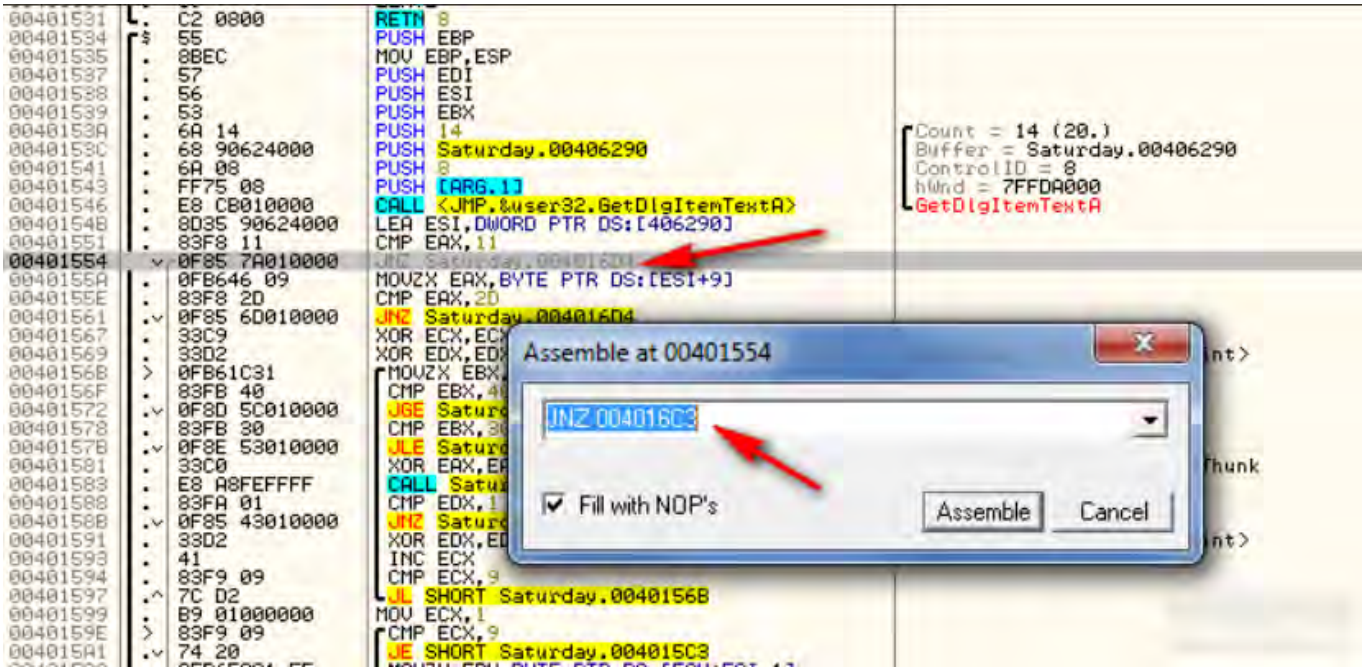
看下从什么地方来到了坏消息处。



找到这个跳转



在这个跳转上设置一个断点，让程序跑起来，断下来，后面的请自己分析，我这里直接跳转到好消息4016C3处。



应用这个补丁，让程序跑起来，我们成功了

好了，我们要使用dUP2进行补丁程序制作。

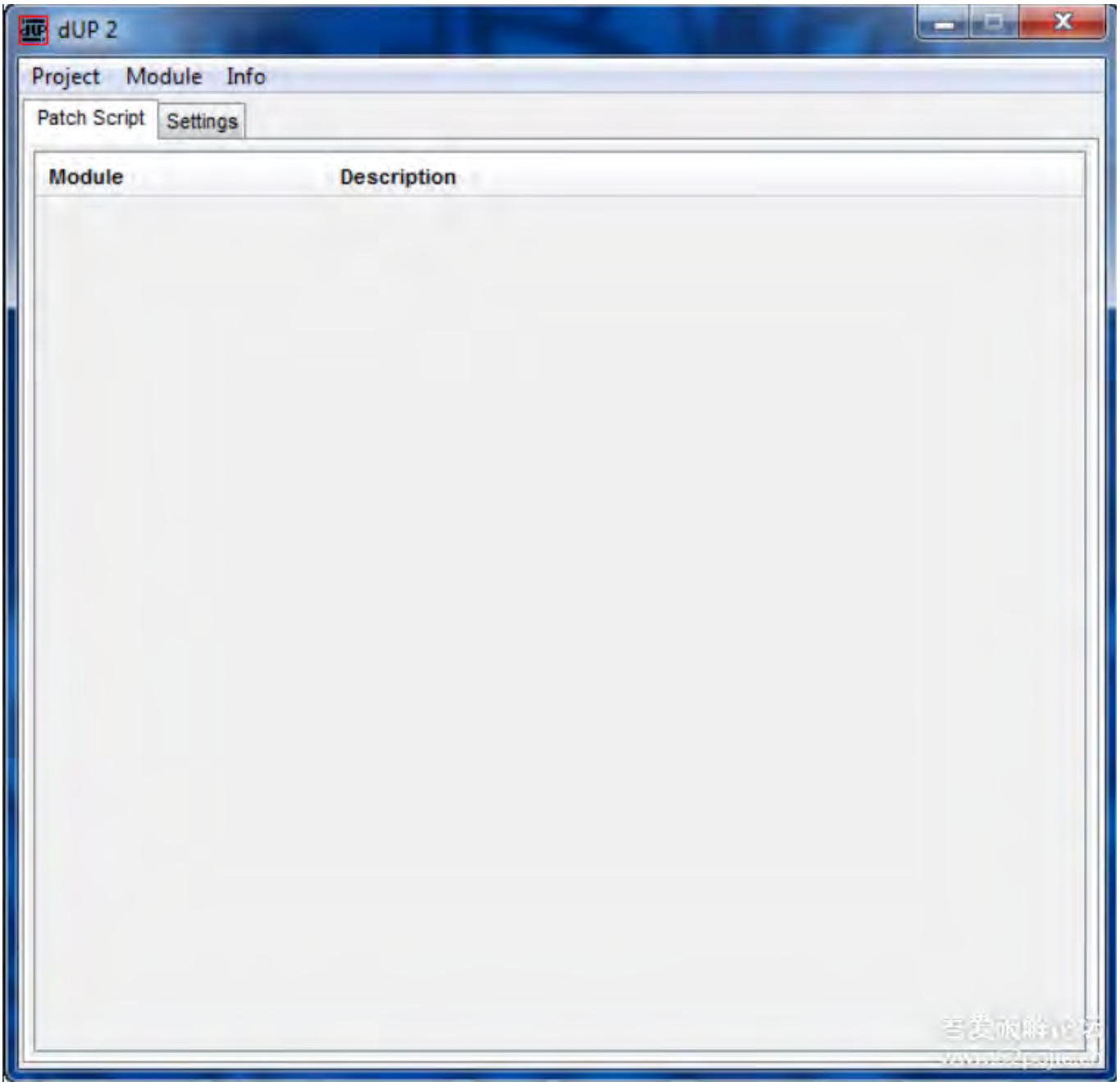
dUP2和其他补丁程序不一样，它只有两种方式进行补丁，1是偏移补丁，2是搜索和替换补丁。偏移补丁：你要知道补丁相对程序的偏移地址，然后进行补丁。使用OD可以很容易找到补丁位置，然后进行补丁。所以我要找到在OD中补丁位置与文件起始地址的偏移值，使用dUP2进行补丁。

第二种方式，搜索和替换补丁，你要知道补丁位置的命令码，对它进行搜索然后替换为补丁后的命令码，然后生成补丁程序。

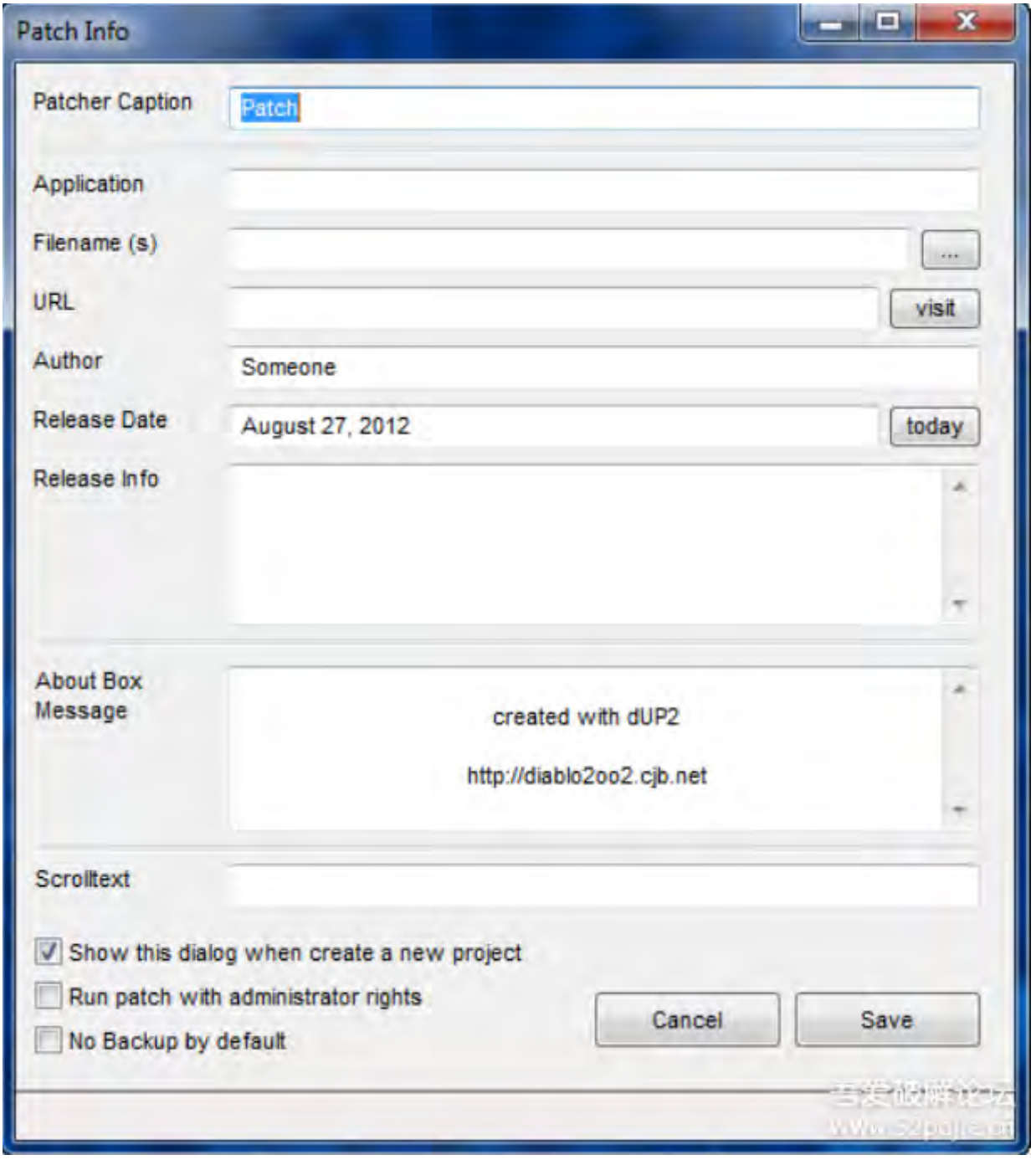
dUP2也可以对注册表进行补丁，像上一章中修改注册表，每隔255次进行一次补丁，就像生产了一个密钥，如果你经常使用软件的话255次很快就会结束。

最后，dUP2可以生成自定义皮肤，这样看上去程序界面比较美观。

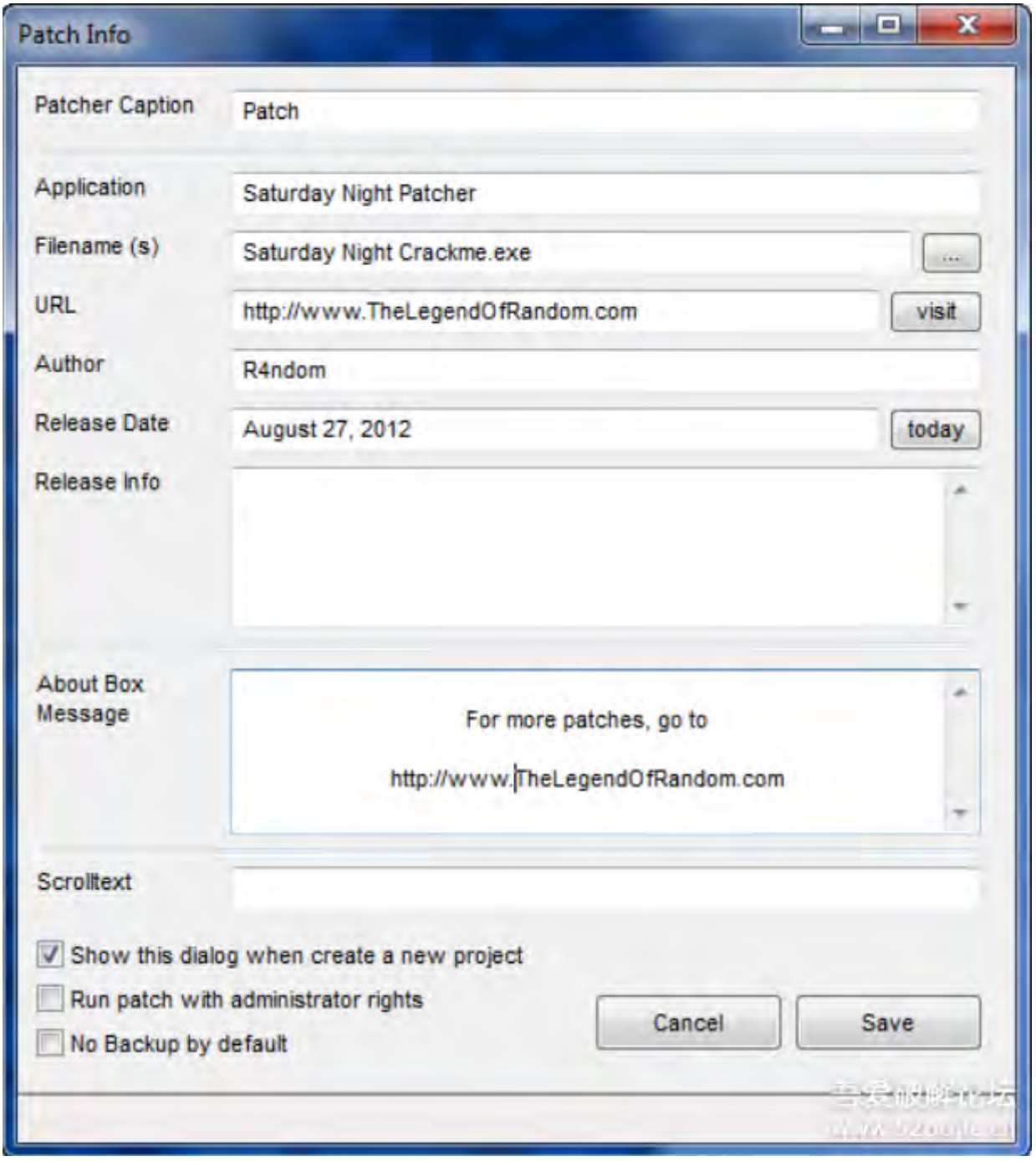
启动dUP2，我们可以看到如下界面：



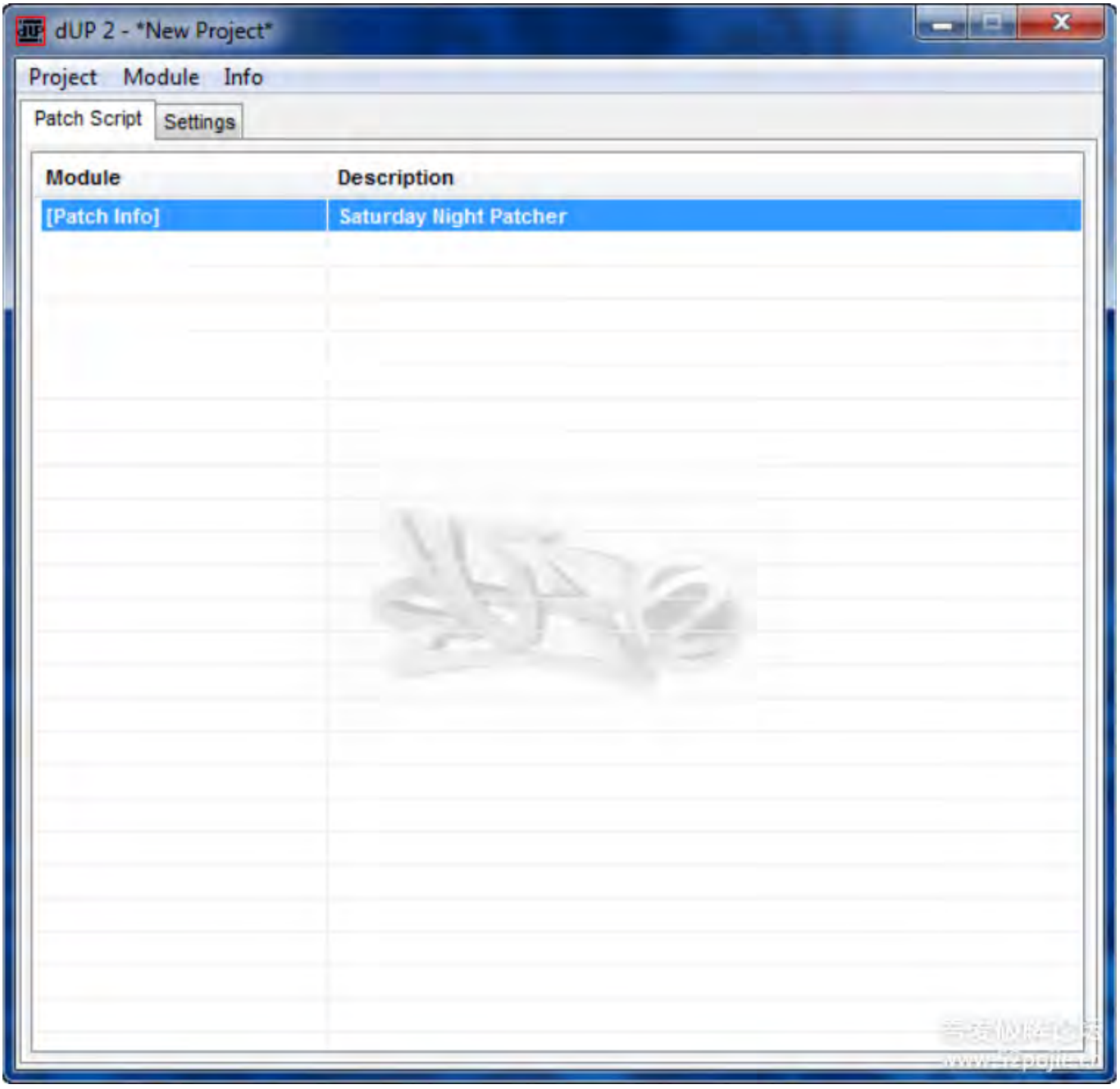
选择project->new：



这里我们可以输入一些信息，比如标题，作者，哪个程序的补丁等，填写完成后

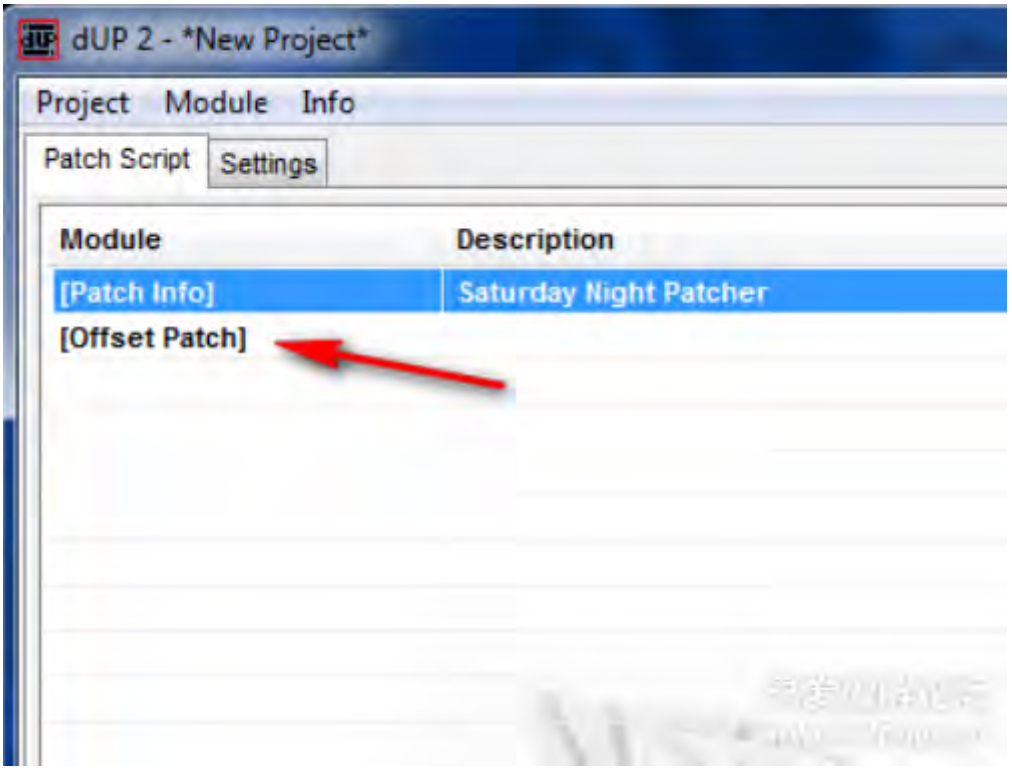

Patch Info
Patcher Caption
Patch
Application
Saturday Night Patcher
Filename (s)
Saturday Night Crackme.exe
...
URL
http://www.TheLegendOfRandom.com
visit
Author
R4ndom
Release Date
August 27, 2012
today
Release Info
About Box Message
For more patches, go to
http://www.TheLegendOfRandom.com
Scrolltext
Show this dialog when create a new project
Run patch with administrator rights
No Backup by default
Cancel
Save

点击save后来到

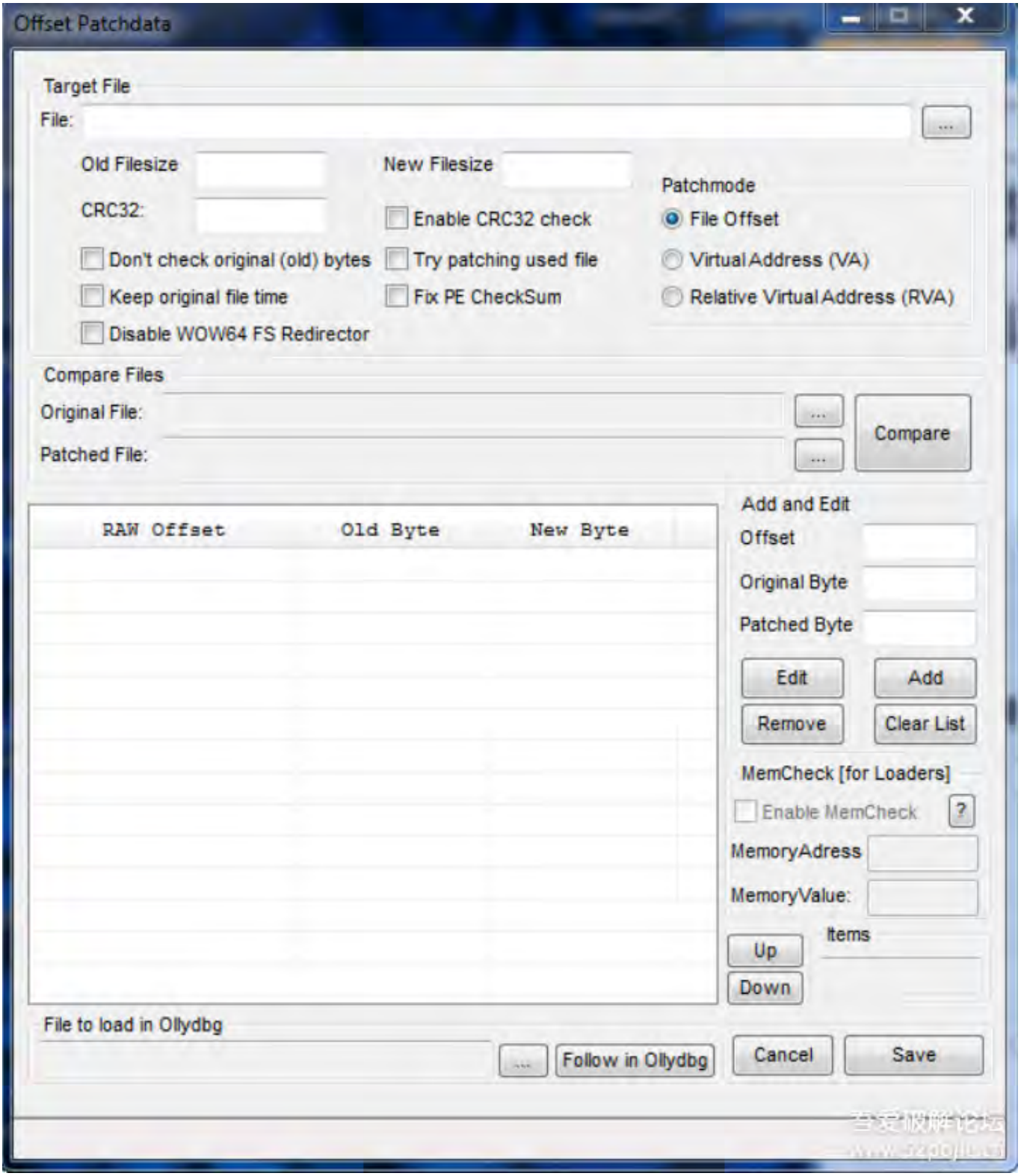


通常到这一步我会把这个工程保存下来以便后面使用，这样就不用再次填写上面的信息了。

现在我们要做补丁。右击后选择 Add 然后 Offset_Patch，界面会多出现一行

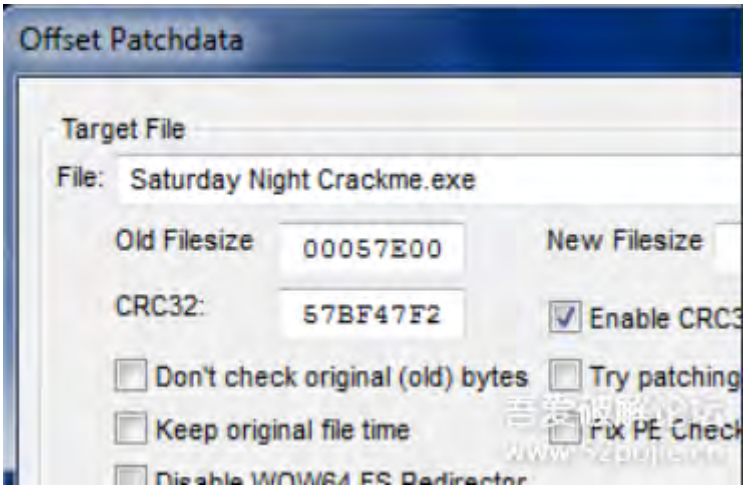

dUP 2 - *New Project*
Project
Module
Info
Patch Script
Settings
Module
Description
[Patch Info]
Saturday Night Patcher
[Offset Patch]

双击后打开：



这个界面就显示了补丁信息，我们点击... 按钮找到要补丁的程序

补丁模式可以选择Virtual Addresses 或者 RVA，而我们使用默认。

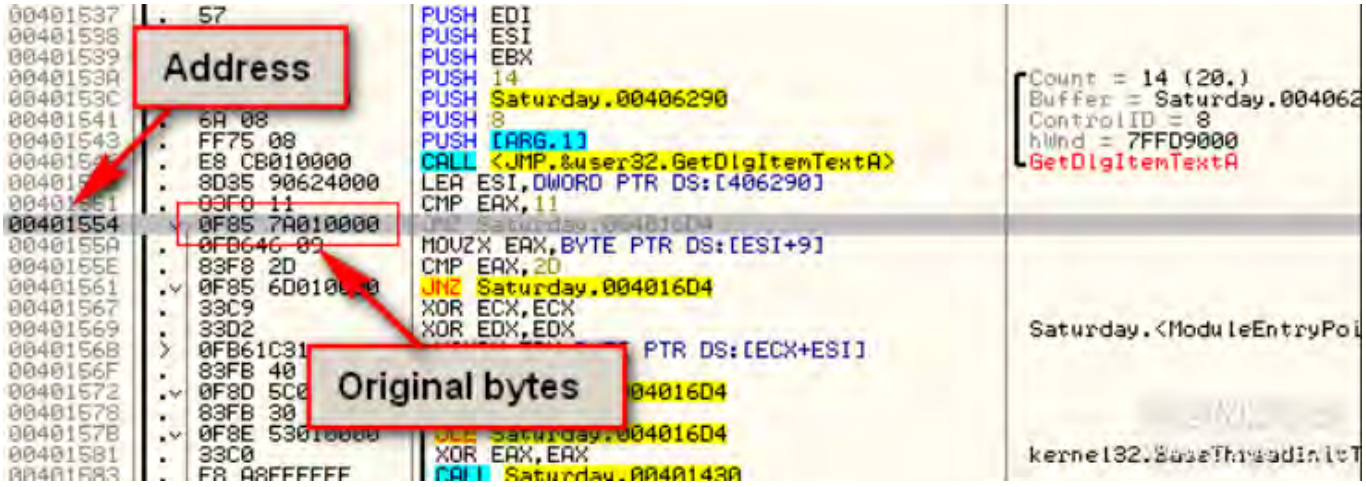


对比文件可以把远程程序和已经补丁后的程序进行对比，找到不一样的地方。

主要修改Add 和 Edit组。这就是我们填写偏移地址，原始数据和补丁数据的地方。

制作补丁程序

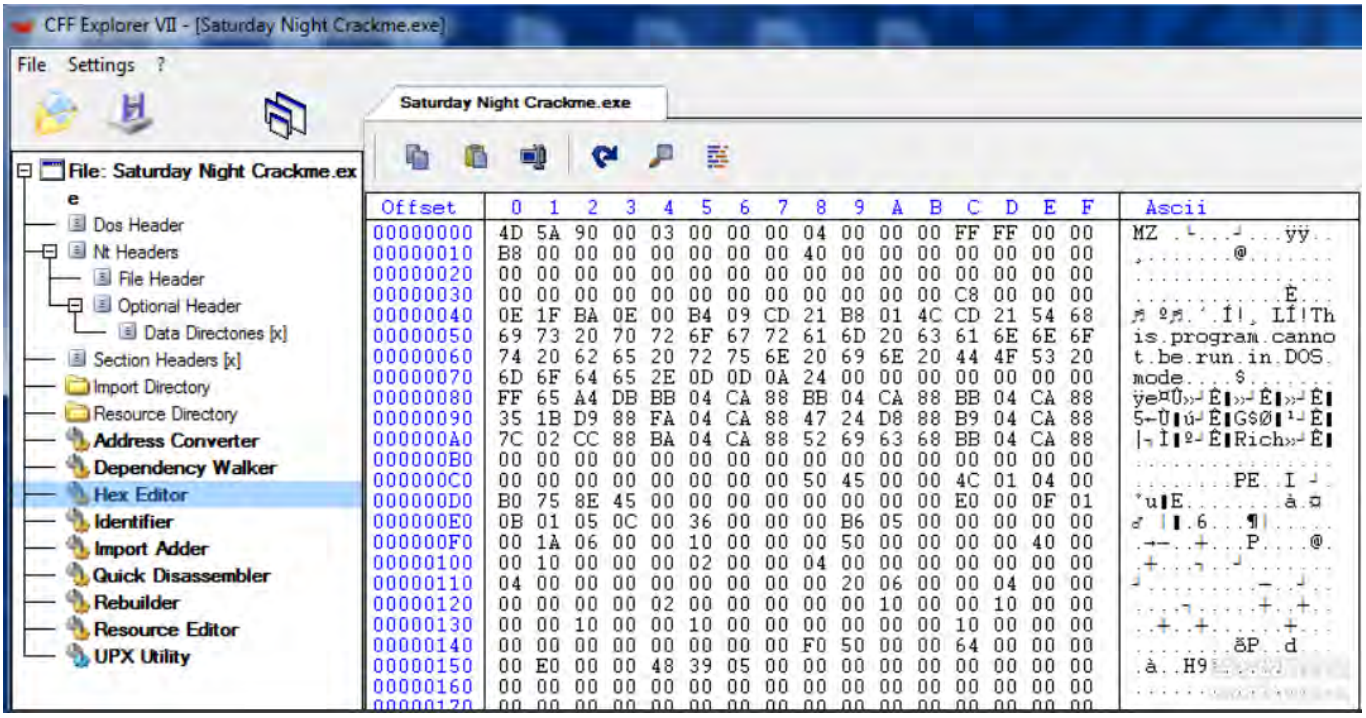
第一件事是找到地址并记下，原始值和补丁值。重新使用OD打开程序，来到401554处，记住原始的命令码



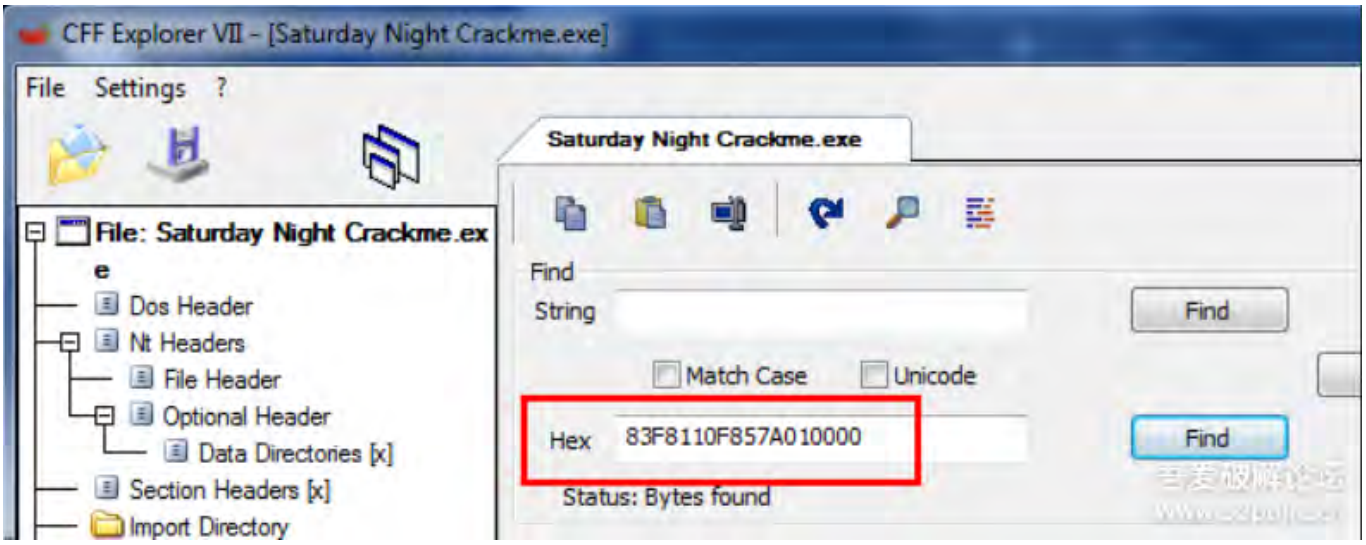
然后应用我们的补丁后看看变化

我们可以看到7A变成了69，然后我们把这一行和前一行的命令码记住：83f8110f857A010000.

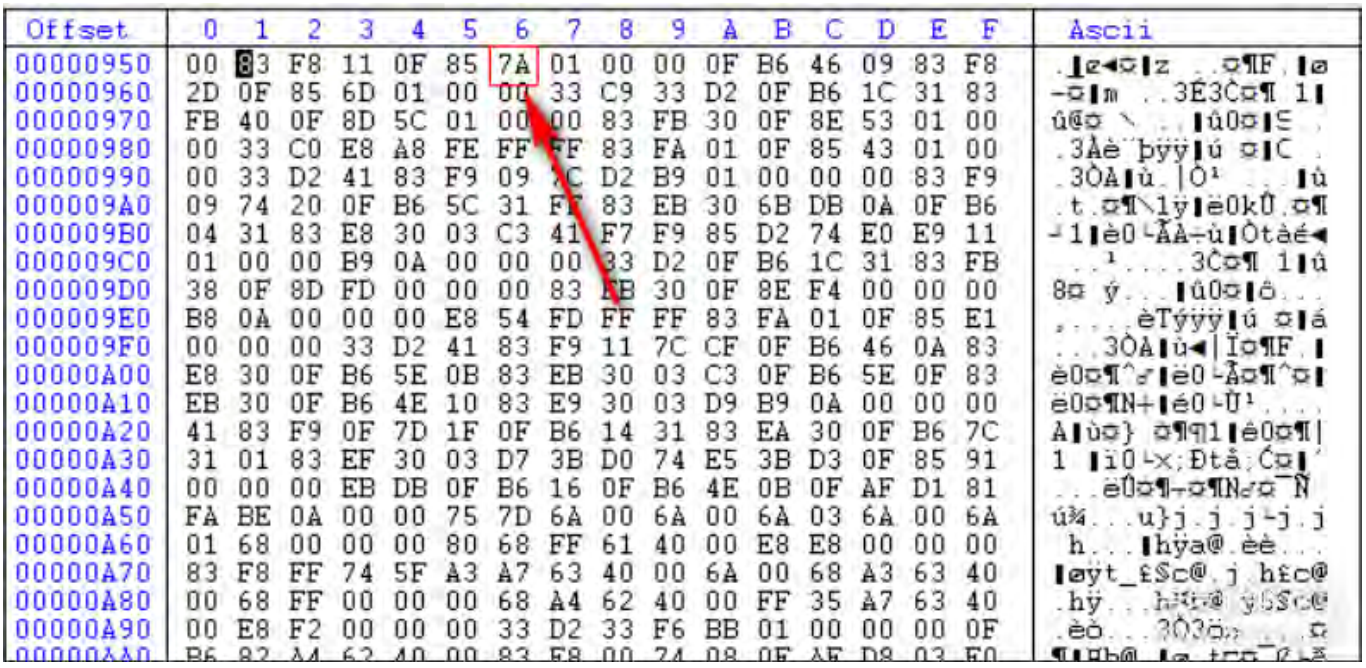
现在使用CFF Explorer打开程序，点击Hex Editor



点击放大镜进行搜索刚刚记下的命令码

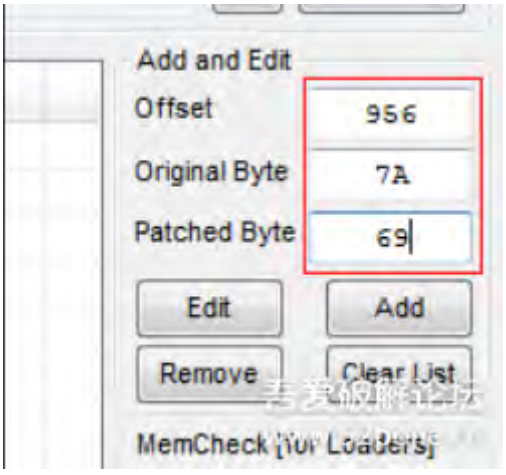


点击 Find 后可以找到

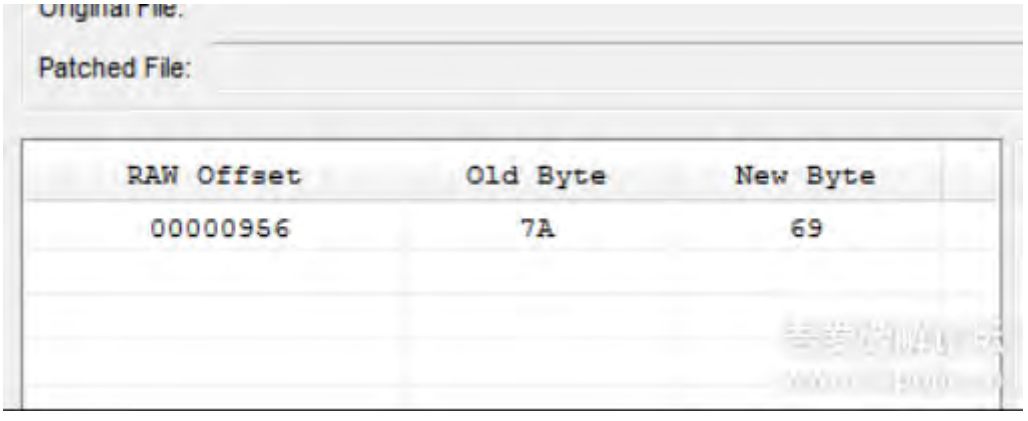


Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000950	00	83	F8	11	0F	85	7A	01	00	00	0F	B6	46	09	83	F8	.ƒø◄¤‖z ..¤¶F.‖ø
00000960	2D	0F	85	6D	01	00	00	33	C9	33	D2	0F	B6	1C	31	83	-¤‖m ..3É3Ò¤¶ 1‖
00000970	FB	40	0F	8D	5C	01	00	00	83	FB	30	0F	8E	53	01	00	û@¤ \ ..‖û0¤‖S ..
00000980	00	33	C0	E8	A8	FE	FF	FF	83	FA	01	0F	85	43	01	00	.3Àè ¨þÿÿ‖ú ¤‖C ..
00000990	00	33	D2	41	83	F9	09	7C	D2	B9	01	00	00	00	83	F9	.3ÒA‖ù .|Ò¹ ...‖ù
000009A0	09	74	20	0F	B6	5C	31	FF	83	EB	30	6B	DB	0A	0F	B6	.t ¤¶\1ÿ‖ë0kÛ.¤¶
000009B0	04	31	83	E8	30	03	C3	41	F7	F9	85	D2	74	E0	E9	11	♦1‖è0♥ÃA÷ù‖Òtàé◄
000009C0	01	00	00	B9	0A	00	00	00	33	D2	0F	B6	1C	31	83	FB	☺...¹....3Ò¤¶ 1‖û
000009D0	38	0F	8D	FD	00	00	00	83	EB	30	0F	8E	F4	00	00	00	8¤ ý...‖û0¤‖ô...
000009E0	B8	0A	00	00	00	E8	54	FD	FF	FF	83	FA	01	0F	85	E1	,....èTýÿÿ‖ú ¤‖á
000009F0	00	00	00	33	D2	41	83	F9	11	7C	CF	0F	B6	46	0A	83	...3ÒA‖ù◄|Ï¤¶F.‖
00000A00	E8	30	0F	B6	5E	0B	83	EB	30	03	C3	0F	B6	5E	0F	83	è0¤¶^♂‖ë0♥Ã¤¶^¤‖
00000A10	EB	30	0F	B6	4E	10	83	E9	30	03	D9	B9	0A	00	00	00	ë0¤¶N►‖é0♥Ù¹....
00000A20	41	83	F9	0F	7D	1F	0F	B6	14	31	83	EA	30	0F	B6	7C	A‖ù¤} ¤¶¶1‖ê0¤¶|
00000A30	31	01	83	EF	30	03	D7	3B	D0	74	E5	3B	D3	0F	85	91	1 ‖ï0♥×;Ðtå;Ó¤‖'
00000A40	00	00	00	EB	DB	0F	B6	16	0F	B6	4E	0B	0F	AF	D1	81	...ëÛ¤¶▬¤¶N♂¤¯Ñ
00000A50	FA	BE	0A	00	00	75	7D	6A	00	6A	00	6A	03	6A	00	6A	ú¾...u}j.j.j♥j.j
00000A60	01	68	00	00	00	80	68	FF	61	40	00	E8	E8	00	00	00	☺h...‖hÿa@.èè...
00000A70	83	F8	FF	74	5F	A3	A7	63	40	00	6A	00	68	A3	63	40	‖øÿt_£§c@.j.h£c@
00000A80	00	68	FF	00	00	00	68	A4	62	40	00	FF	35	A7	63	40	.hÿ...h¤b@.ÿ5§c@
00000A90	00	E8	F2	00	00	00	33	D2	33	F6	BB	01	00	00	00	0F	.èò...3Ò3ö»....
00000AA0	B6	82	A4	63	40	00	83	F8	00	74	08	0F	AF	D8	03	F0	¶‚¤c@.‖ø.t▭¯Ø♥ð

我们的7A偏移了956个字节，现在回到dUP2进行修改

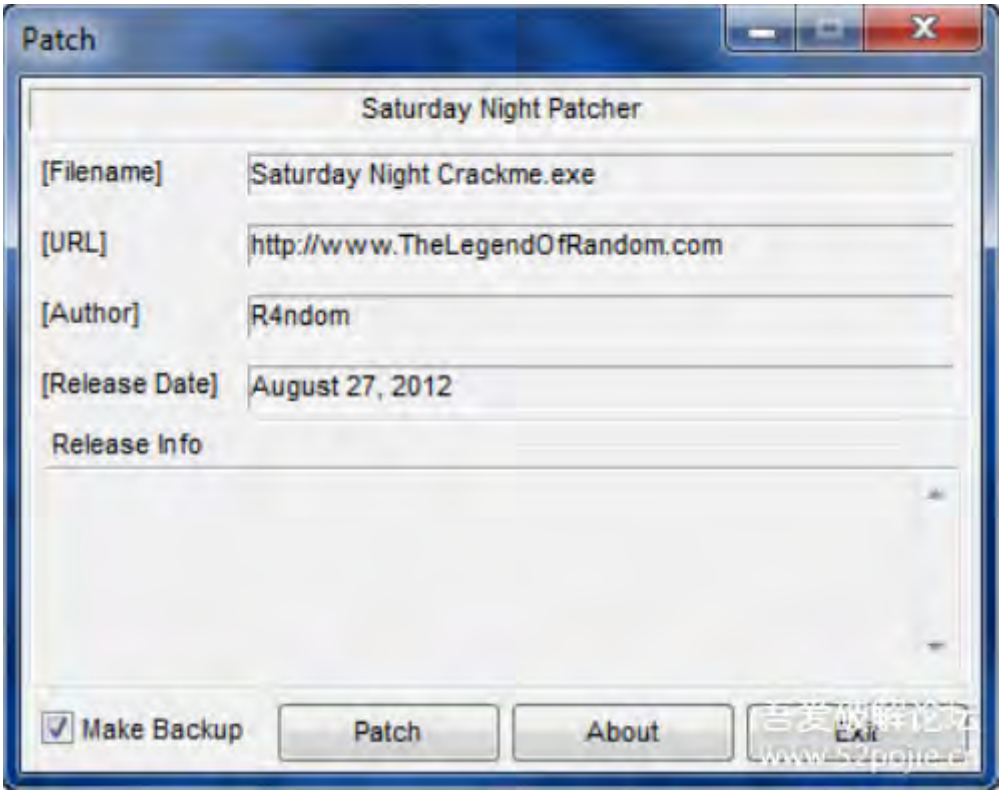


然后点击Add 后来到

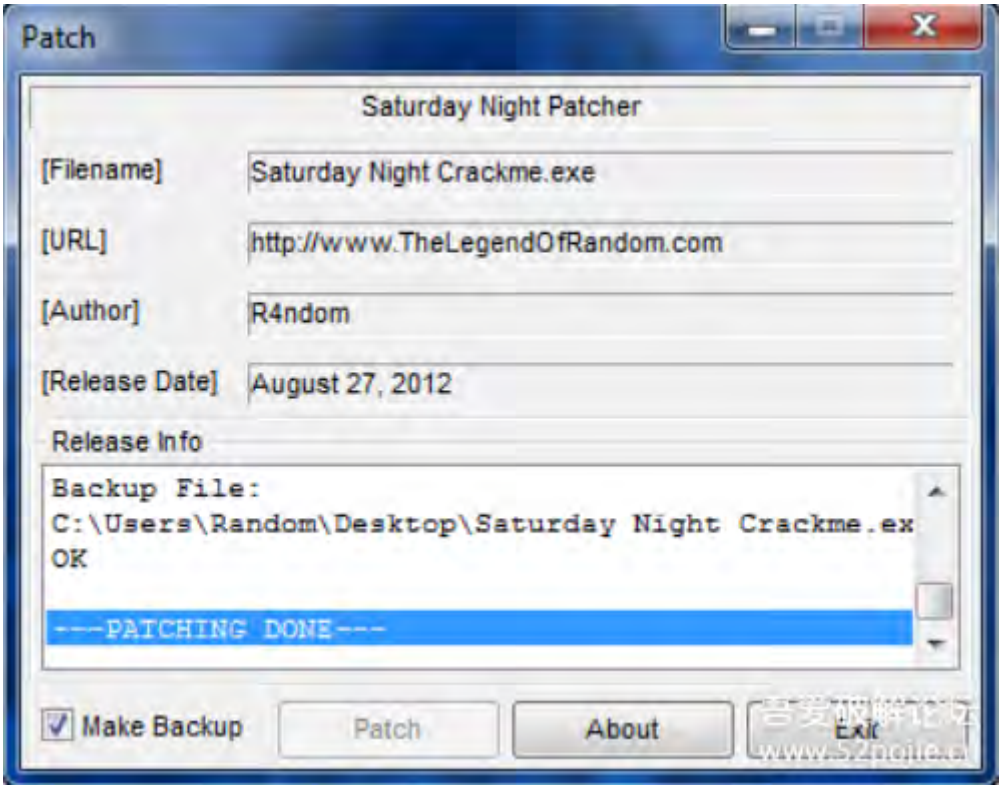


点击 Save 后就把补丁保存到我们的程序里面了

现在选择 Project ->Create Patch后，会在程序的同级目录生成一个Patch.exe，运行这个Patch后



点击Patch按钮后

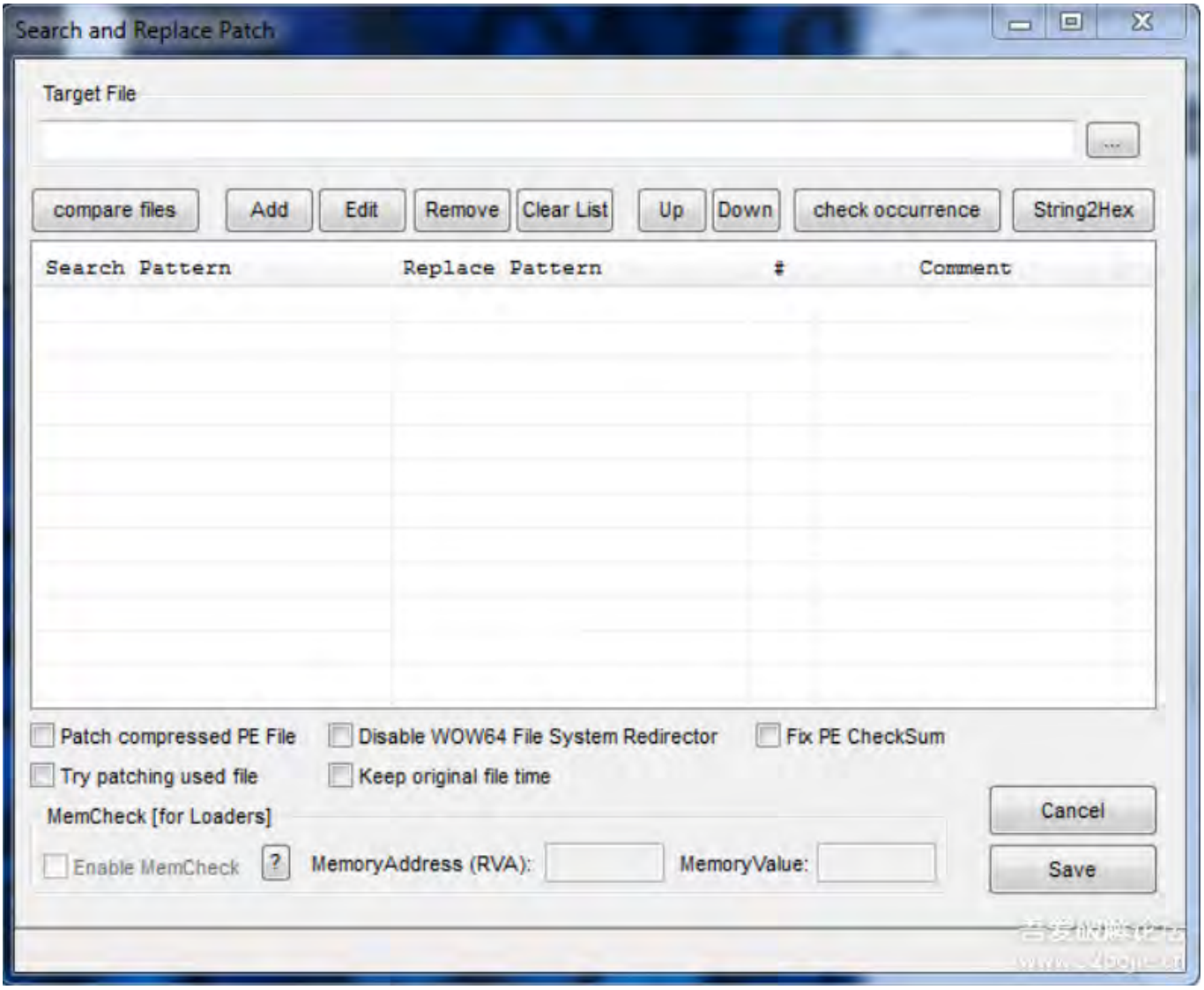


我们可以看到Patch成功的信息。然后运行程序，显示成功了

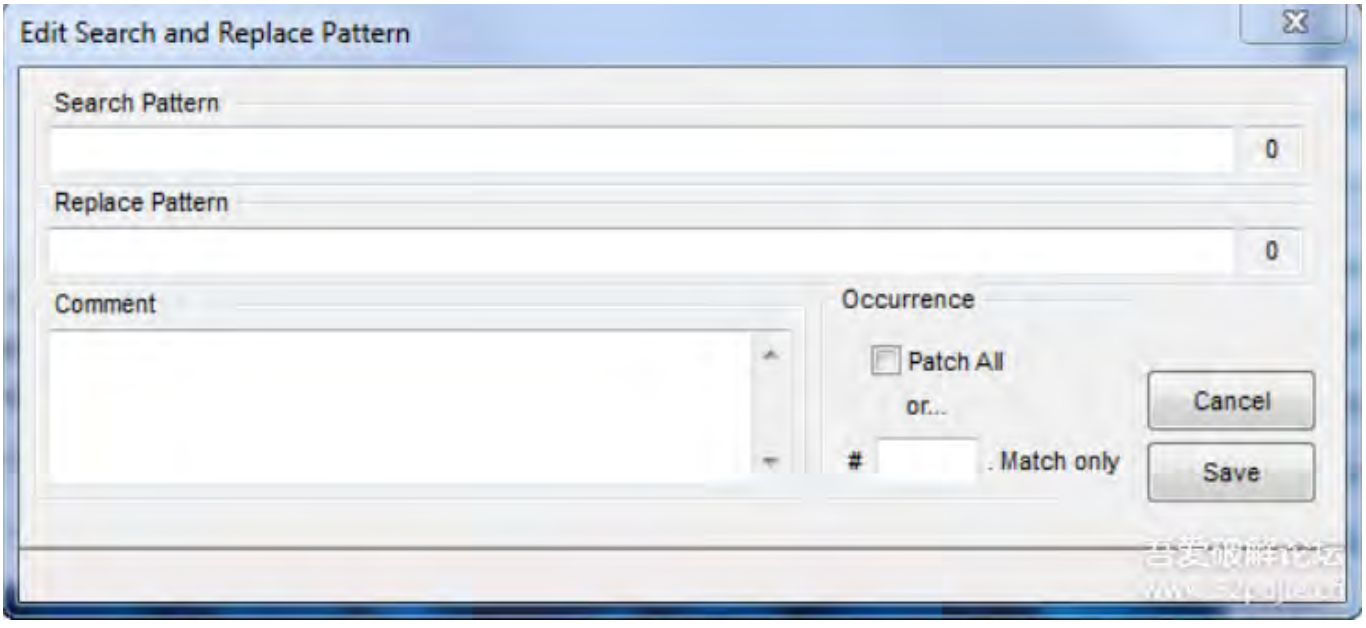
注意patch程序必须和原始程序放在同一目录才有效。

使用搜索和替换方式

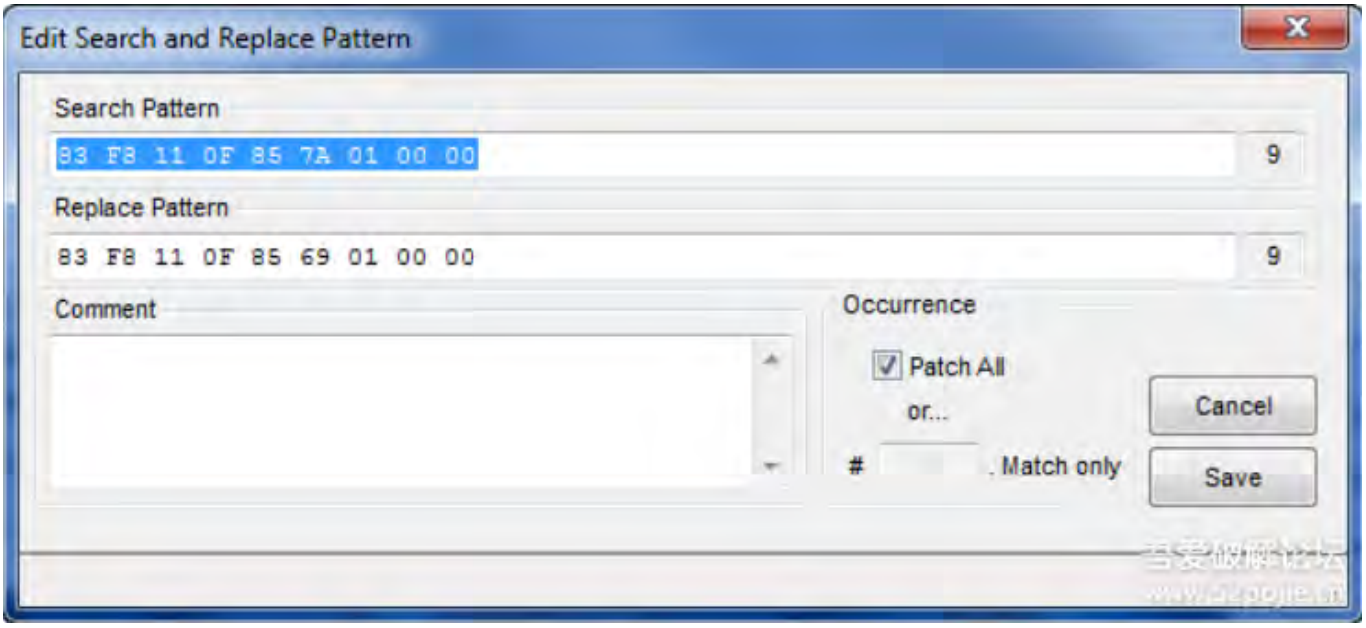
打开刚刚保存的dUP2工程，右击选择 Add->Search and replace Patch,双击后出现：



点击 Add 按钮

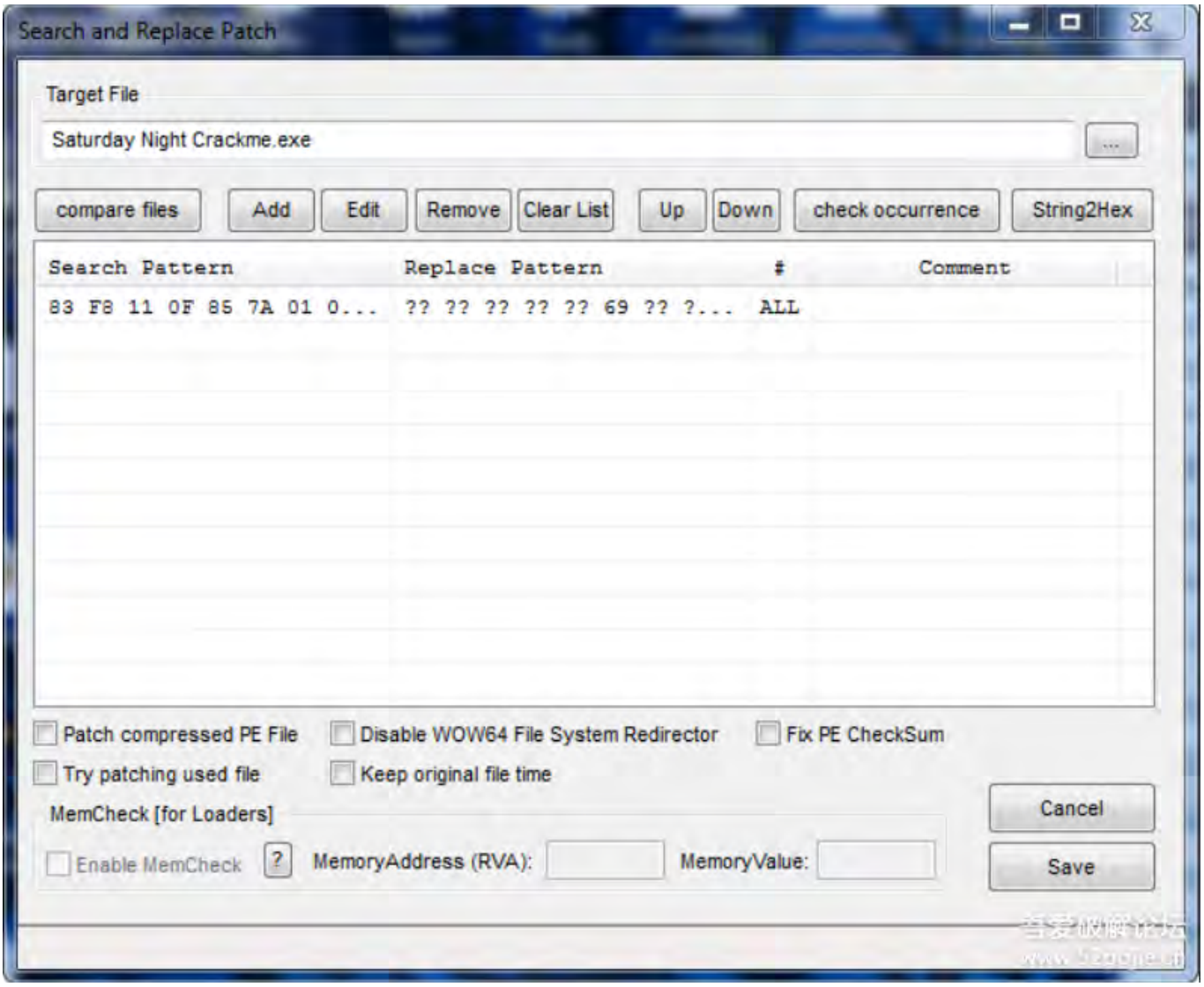


现在把刚刚的原始命令码和补丁命令码填到相应位置

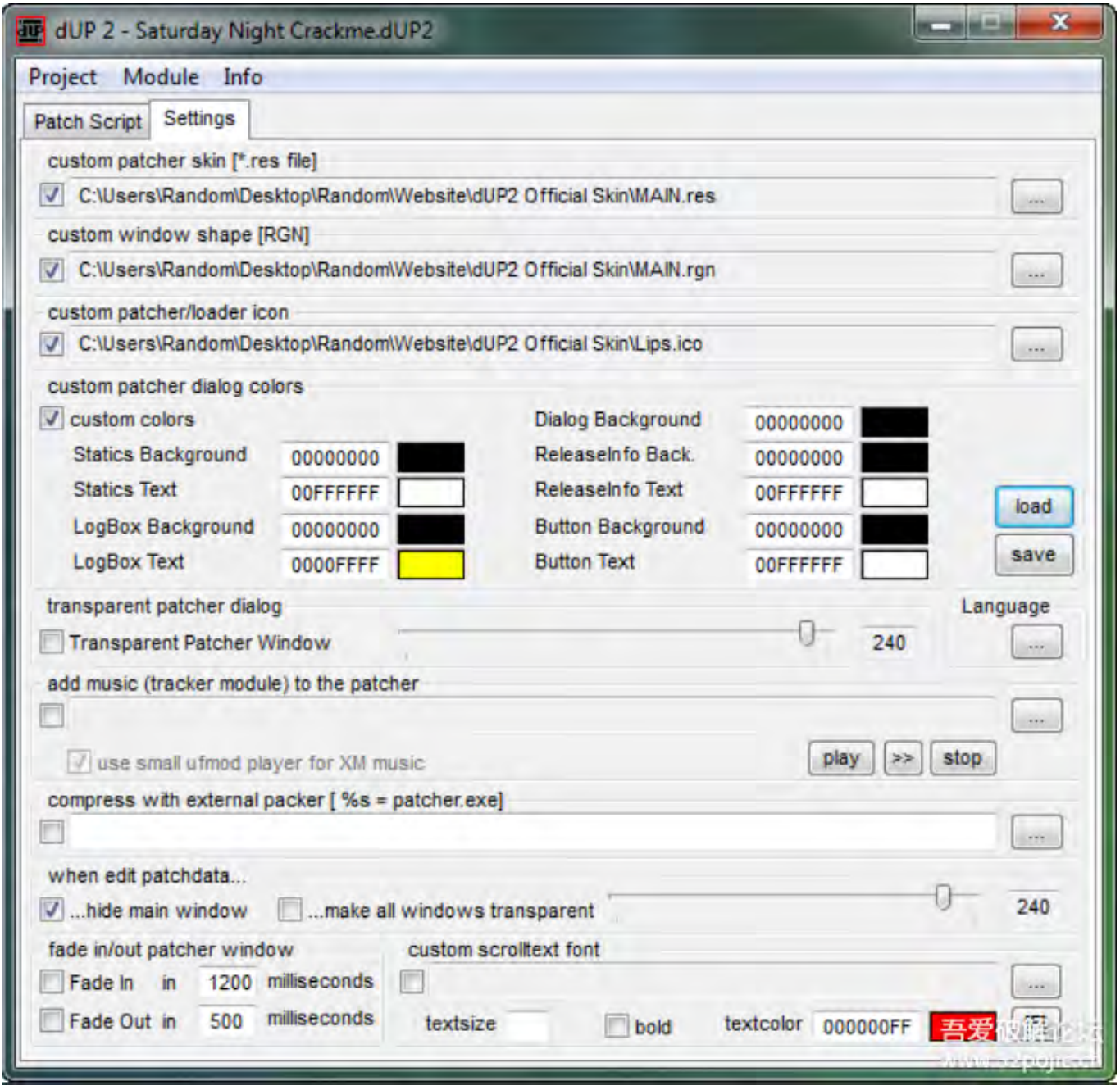


选择 patch All，然后保存

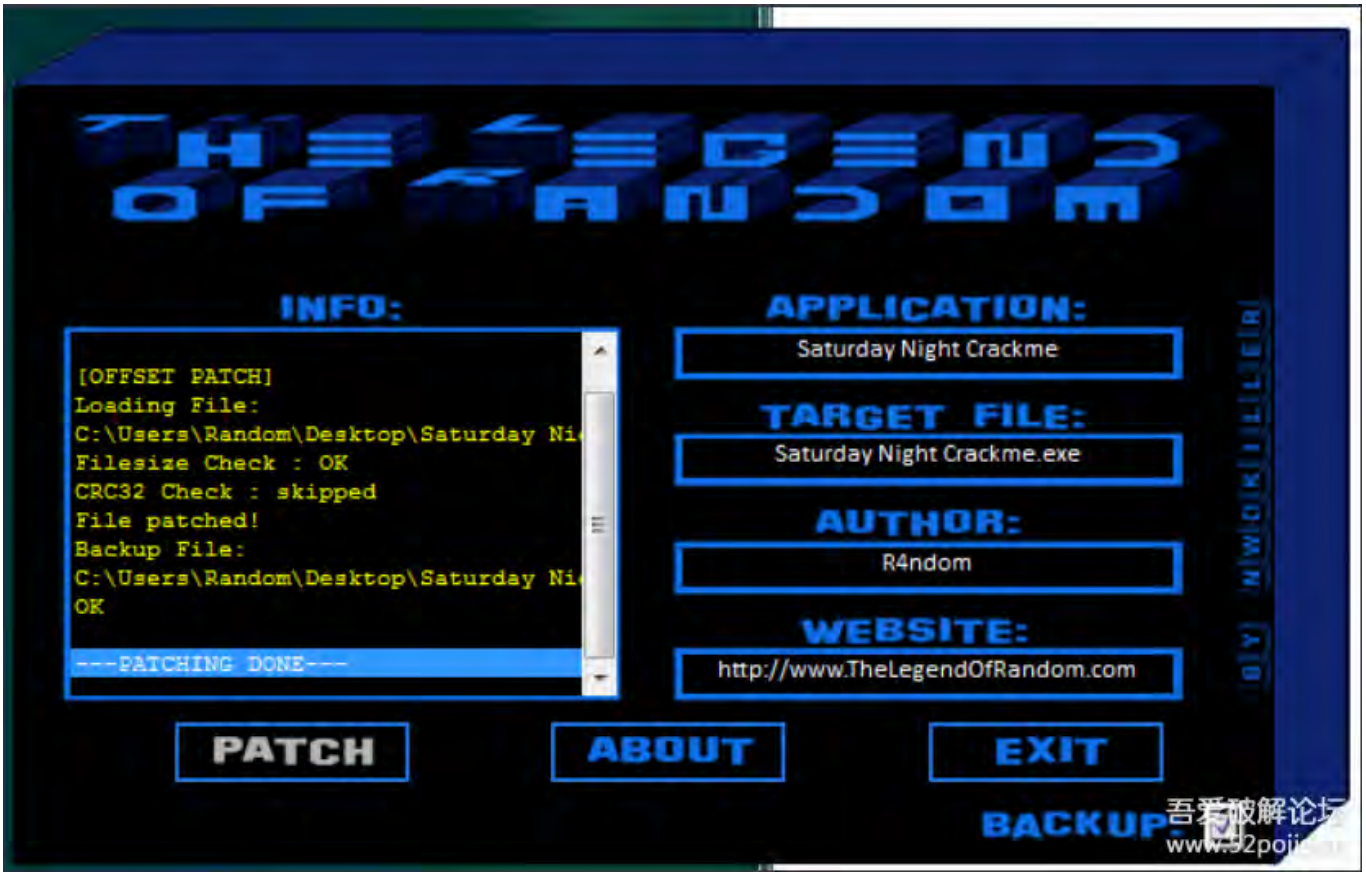
（译者注：注意这里一个字节空一位否则保存不了）



再次点击Save来到主界面。但是现在我们要进行界面修改，附件中有SSECS cUP skin.zip



按照我上面的设置然后进行ProjectàCreate Patch，就生成了新的补丁程序，界面如下；


THE LEGEND OF RANDOM
INFO:
[OFFSET PATCH]
Loading File:
C:\Users\Random\Desktop\Saturday Ni
Filesize Check : OK
CRC32 Check : skipped
File patched!
Backup File:
C:\Users\Random\Desktop\Saturday Ni
OK
---PATCHING DONE---
APPLICATION:
Saturday Night Crackme
TARGET FILE:
Saturday Night Crackme.exe
AUTHOR:
R4ndom
WEBSITE:
http://www.TheLegendOfRandom.com
PATCH
ABOUT
EXIT
BACKUP:
[0Y] [WORKILLER]

第二十章（上）：玩转 VB 程序 - 第一部分

翻译都是我理解的方式进行描述，可能和原文不一致。

本教程中文版只在吾爱破解论坛 首发。

转载请注明来自吾爱破解论坛@52pojie.cn

正文开始

本章主要讲解VB编写的程序要怎么破解，因为这是一个大工程，所以我分为两个章节进行说明。

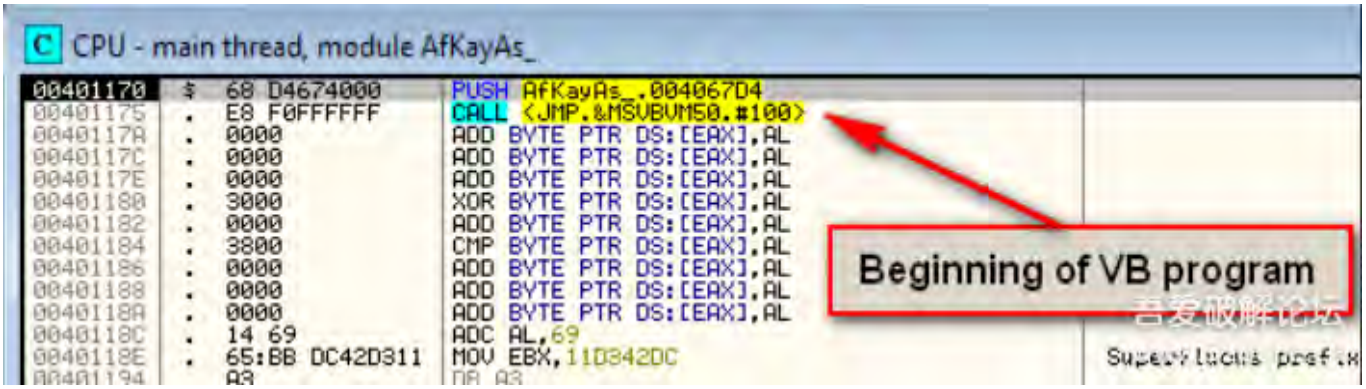
我们来看下我所提供的两个VB程序，被教程中的所有文件都会提供下载。

介绍VB编程语言与其他编程语言的区别，这个大家自行百度。

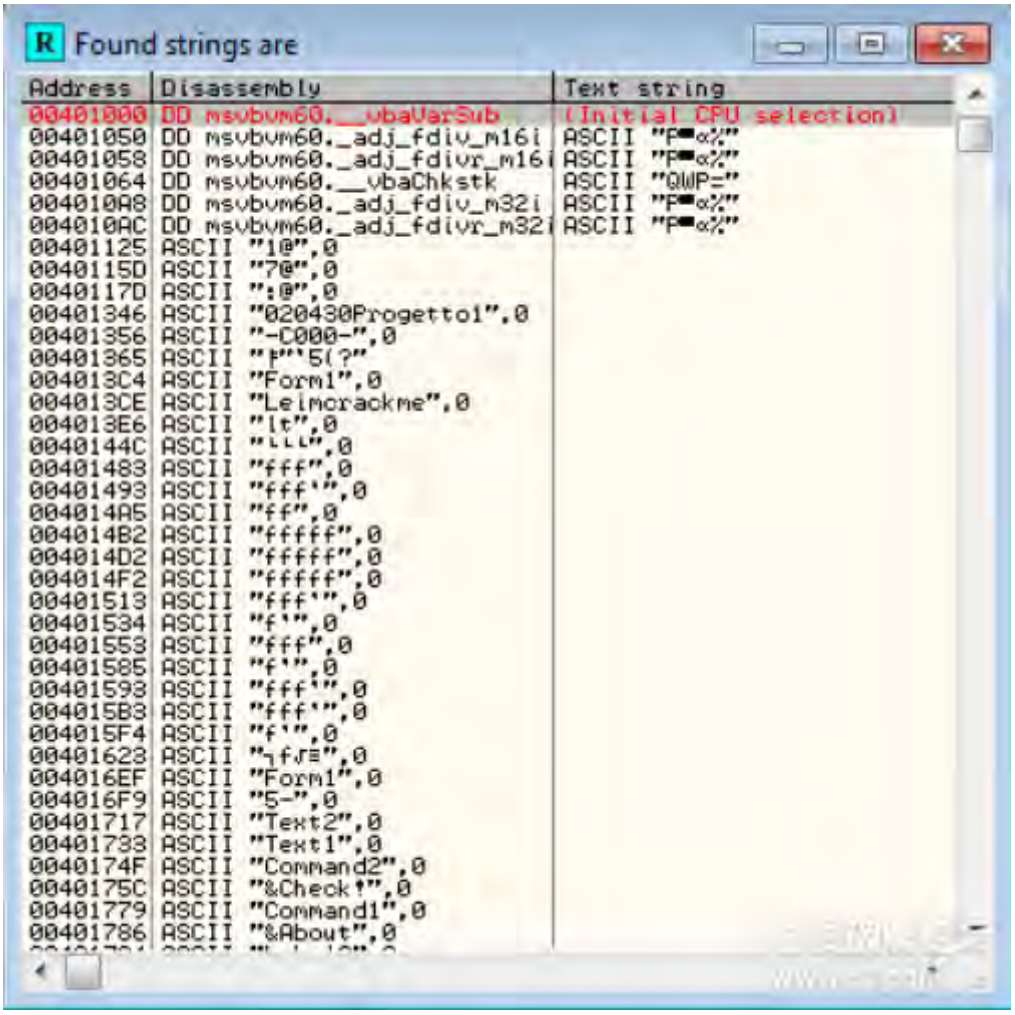
主要说明VB语言可以转换为p_code，这个语言可以跨平台移植，在Mac、Linux上都可以正常调用（如果是VB写的Dll）。

教程开始：

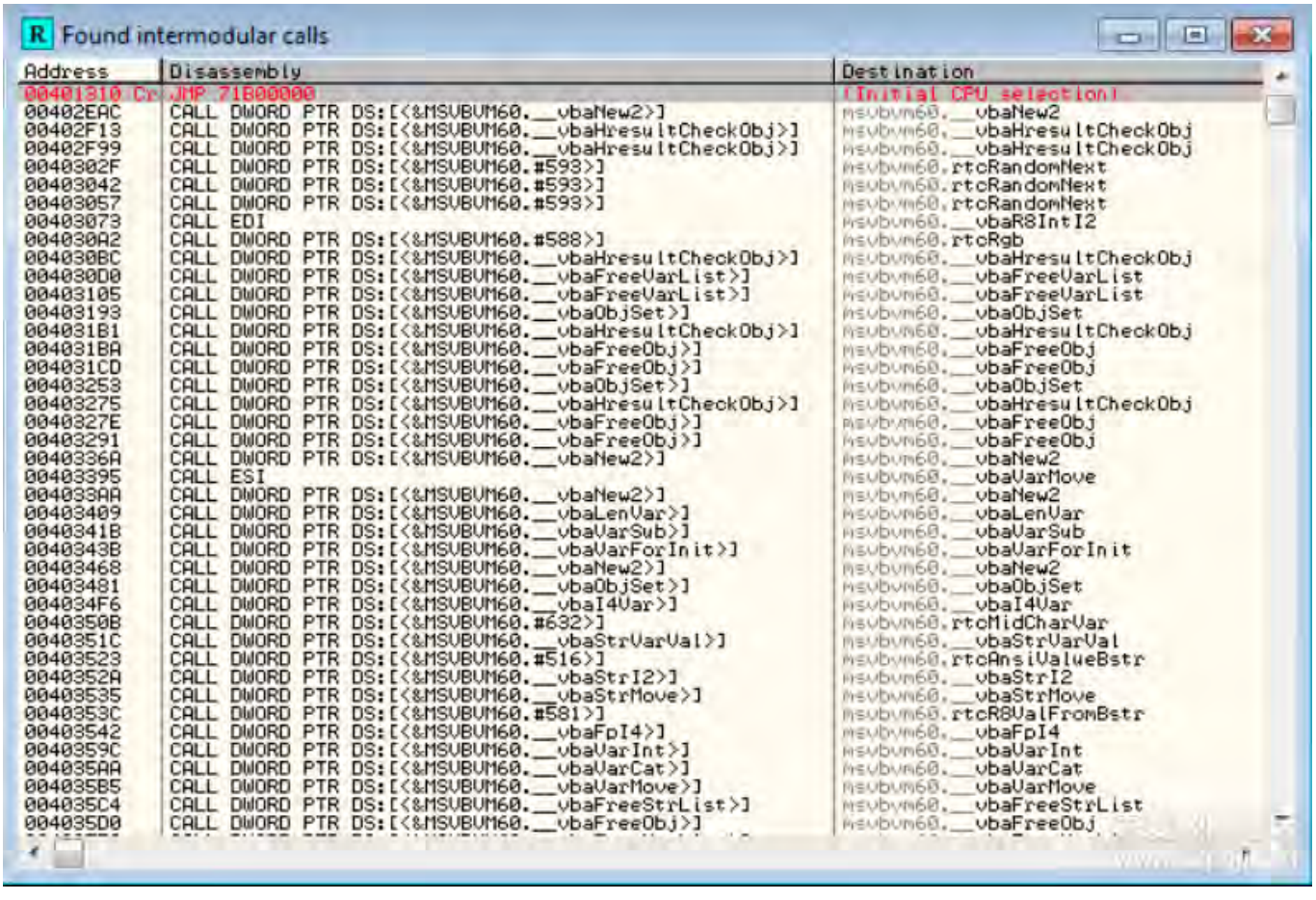
当你第一次使用OD打开一个VB编写的程序时，你会马上看到VB调用（MSVBVM50）DLL的Call，然后停在这里等待事件响应，因此这样就会让逆向程序员感觉有所不同。第一件事是你会发现call的栈中信息很少；因为多数程序的运行时间都是放在DLL中，而我们不关心DLL，只关心程序的callback、方法、时间。



另一个不同是字串表是查不到有用的东西的，因为多数的消息窗口和其他windows控件都是放在资源文件中，OD不会显示出来，不像逆向C\C++程序时可用查找字串参考。



另外一个比较麻烦的地方是，VB调用RegisterWindowEx和MessageBox是在自己的DLL中调用，我们不能下API断点。



通过点击这些方法，能看到VB和我们之前遇到的程序不一样的地方

```
004031FE    90              NOP
004031FF    90              NOP
00403200  > 55              PUSH EBP
00403201  . 8BEC            MOV EBP,ESP
00403203  . 83EC 0C         SUB ESP,0C
00403206  . 68 A6114000     PUSH <JMP.&MSVBVM60.__vbaExceptHandler>          SE handler installation
0040320B  . 64:A1 00000000  MOV EAX,DWORD PTR FS:[0]
00403211  . 50              PUSH EAX                                         kernel32.BaseThreadInitThunk
00403212  . 64:8925 00000000 MOV DWORD PTR FS:[0],ESP
00403219  . 83EC 14         SUB ESP,14
0040321C  . 53              PUSH EBX
0040321D  . 56              PUSH ESI
0040321E  . 57              PUSH EDI
0040321F  . 8965 F4         MOV DWORD PTR SS:[EBP-C],ESP
00403222  . C745 F8 30114000 MOV DWORD PTR SS:[EBP-8],Crackme_.00401130
00403229  . 8B75 08         MOV ESI,DWORD PTR SS:[EBP+8]
0040322C  . 8BC6            MOV EAX,ESI
0040322E  . 83E0 01         AND EAX,1
00403231  . 8945 FC         MOV DWORD PTR SS:[EBP-4],EAX                     kernel32.BaseThreadInitThunk
00403234  . 83E6 FE         AND ESI,FFFFFFFE
00403237  . 56              PUSH ESI
00403238  . 8975 08         MOV DWORD PTR SS:[EBP+8],ESI
0040323B  . 8B0E            MOV ECX,DWORD PTR DS:[ESI]
0040323D  . FF51 04         CALL DWORD PTR DS:[ECX+4]
00403240  . 8B16            MOV EDX,DWORD PTR DS:[ESI]
00403242  . 33FF            XOR EDI,EDI
00403244  . 56              PUSH ESI
00403245  . 897D E8         MOV DWORD PTR SS:[EBP-18],EDI
00403248  . FF92 FC020000   CALL DWORD PTR DS:[EDX+2FC]
0040324E  . 50              PUSH EAX                                         kernel32.BaseThreadInitThunk
0040324F  . 8D45 E8         LEA EAX,DWORD PTR SS:[EBP-18]
00403252  . 50              PUSH EAX                                         kernel32.BaseThreadInitThunk
00403253  . FF15 4C104000   CALL DWORD PTR DS:[<&MSVBVM60.__vbaObjSet>]      msvbvm60.__vbaObjSet
00403259  . 8BF0            MOV ESI,EAX                                      kernel32.BaseThreadInitThunk
0040325B  . 68 FF000000     PUSH 0FF
00403260  . 56              PUSH ESI
00403261  . 8B0E            MOV ECX,DWORD PTR DS:[ESI]
00403263  . FF51 6C         CALL DWORD PTR DS:[ECX+6C]
00403266  . 3BC7            CMP EAX,EDI
00403268  . DBE2            FCLEX
0040326A  .v 7D 0F          JGE SHORT Crackme_.0040327B
0040326C  . 6A 6C           PUSH 6C
0040326E  . 68 48274000     PUSH Crackme_.00402748
00403273  . 56              PUSH ESI
00403274  . 50              PUSH EAX                                         kernel32.BaseThreadInitThunk
00403275  . FF15 34104000   CALL DWORD PTR DS:[<&MSVBVM60.__vbaHresultCheckObj>] msvbvm60.__vbaHresultCheckObj
0040327B  > 8D4D E8         LEA ECX,DWORD PTR SS:[EBP-18]
0040327E  . FF15 EC104000   CALL DWORD PTR DS:[<&MSVBVM60.__vbaFreeObj>]     msvbvm60.__vbaFreeObj
00403284  . 897D FC         MOV DWORD PTR SS:[EBP-4],EDI
00403287  . 68 99324000     PUSH Crackme_.00403299
0040328C  .v EB 0A          JMP SHORT Crackme_.00403298
0040328E  . 8D4D E8         LEA ECX,DWORD PTR SS:[EBP-18]
00403291  . FF15 EC104000   CALL DWORD PTR DS:[<&MSVBVM60.__vbaFreeObj>]     msvbvm60.__vbaFreeObj
00403297  . C3              RETN
00403298  > C3              RETN                                             RET used as a jump to 00403299
00403299  > 8B45 08         MOV EAX,DWORD PTR SS:[EBP+8]
0040329C  . 50              PUSH EAX                                         kernel32.BaseThreadInitThunk
0040329D  . 8B10            MOV EDX,DWORD PTR DS:[EAX]
0040329F  . FF52 08         CALL DWORD PTR DS:[EDX+8]
004032A2  . 8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]
004032A5  . 8B4D EC         MOV ECX,DWORD PTR SS:[EBP-14]
004032A8  . 5F              POP EDI                                          kernel32.751B339A
004032A9  . 5E              POP ESI                                          kernel32.751B339A
004032AA  . 64:890D 00000000 MOV DWORD PTR FS:[0],ECX
```

你可看到，这里没有有用的API，没有有用的字串。

在看代码前，我们把用p-code编译CrackmeVB1.exe加载到OD，然后把代码窗口拉到最上面，看看VB程序的架构。我们可以看到DLL的很多功能

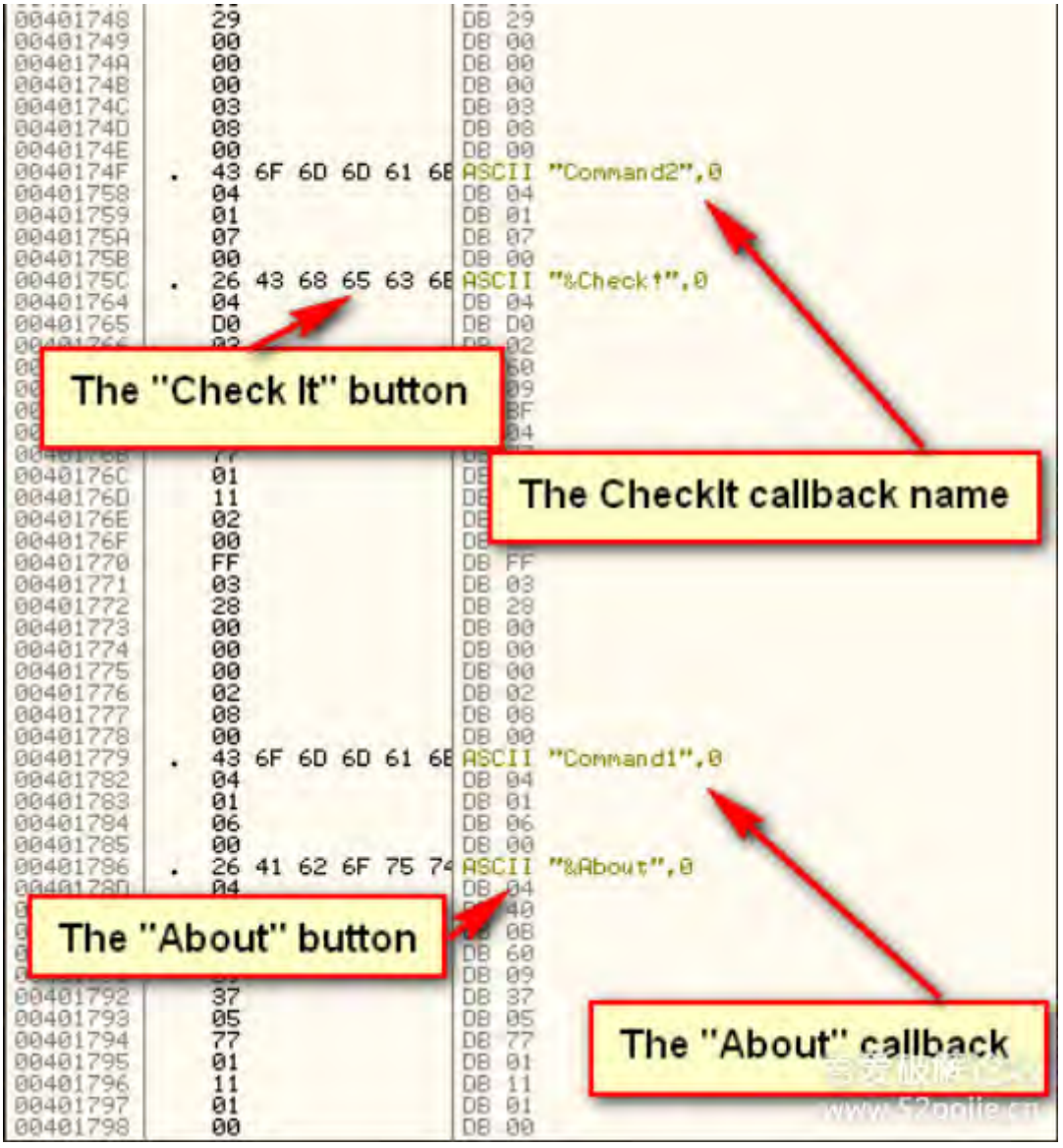
```
00401000  .  EA77A472    DD msvbvm60.__vbaVarSub
00401004  .  0705A272    DD msvbvm60.__vbaStrI2
00401008  .  8693A372    DD msvbvm60._CIcos
0040100C  .  F909A372    DD msvbvm60._adj_fptan
00401010  .  EE6AA472    DD msvbvm60.__vbaVarMove
00401014  .  3168A472    DD msvbvm60.__vbaFreeVar
00401018  .  9B6AA272    DD msvbvm60.__vbaLenBstr
0040101C  .  8DCCA172    DD msvbvm60.rtcRgb
00401020  .  6272A472    DD msvbvm60.__vbaFreeVarList
00401024  .  BA02A372    DD msvbvm60._adj_fdiv_m64
00401028  .  C39FA172    DD msvbvm60.__vbaFreeObjList
0040102C  .  B770A272    DD msvbvm60.rtcAnsiValueBstr
00401030  .  4109A372    DD msvbvm60._adj_fprem1
00401034  .  74A2A172    DD msvbvm60.__vbaHresultCheckObj
00401038  .  AB6AA272    DD msvbvm60.__vbaLenVar
0040103C  .  6E02A372    DD msvbvm60._adj_fdiv_m32
00401040  .  CC93A472    DD msvbvm60.__vbaVarForInit
00401044  .  05CDA172    DD msvbvm60.rtcRandomNext
00401048  .  32D1A172    DD msvbvm60.rtcMsgBox
0040104C  .  F19FA172    DD msvbvm60.__vbaObjSet
00401050  .  0603A372    DD msvbvm60._adj_fdiv_m16i
00401054  .  08A0A172    DD msvbvm60.__vbaObjSetAddref
00401058  .  0604A372    DD msvbvm60._adj_fdivr_m16i
0040105C  .  EE94A372    DD msvbvm60._CIsin
00401060  .  2F70A272    DD msvbvm60.rtcMidCharVar
00401064  .  EA62A372    DD msvbvm60.__vbaChkstk
00401068  .  749BA072    DD msvbvm60.EVENT_SINK_AddRef
0040106C  .  F697A472    DD msvbvm60.__vbaVarTstEq
00401070  .  F609A372    DD msvbvm60._adj_fpatan
00401074  .  879BA072    DD msvbvm60.EVENT_SINK_Release
00401078  .  9395A372    DD msvbvm60._CIsqrt
0040107C  .  859AA072    DD msvbvm60.EVENT_SINK_QueryInterface
00401080  .  6076A472    DD msvbvm60.__vbaVarMul
00401084  .  DF47A272    DD msvbvm60.__vbaExceptHandler
00401088  .  8906A372    DD msvbvm60._adj_fprem
0040108C  .  BA03A372    DD msvbvm60._adj_fdivr_m64
00401090  .  1375A472    DD msvbvm60.__vbaFPException
00401094  .  4819A272    DD msvbvm60.__vbaStrVarVal
00401098  .  7D69A272    DD msvbvm60.__vbaVarCat
0040109C  .  2B94A372    DD msvbvm60._CIlog
004010A0  .  37A2A172    DD msvbvm60.__vbaNew2
```

这就是运行领空中所有用到的API call了。

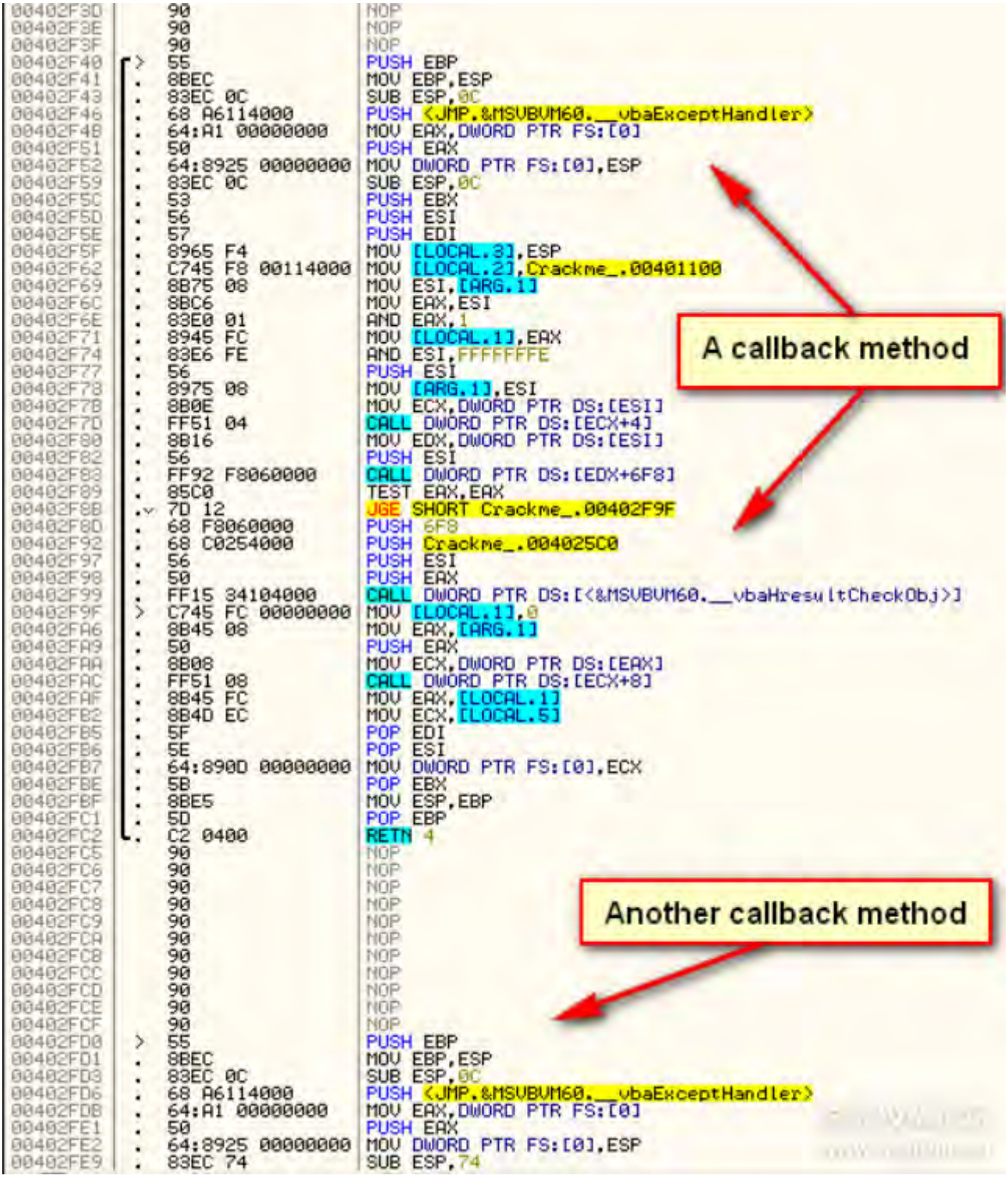
代码窗口往下拉一点点，我们可以看到跳转表。这和多数windows程序的一样。

```
0040119C     DF3C4000    DD Crackme_.00403CDF
004011A0  .- FF25 64104000  JMP DWORD PTR DS:[<&MSVBVM60.__vbaChkstk>]          msvbvm60.__vbaChkstk
004011A6  $- FF25 84104000  JMP DWORD PTR DS:[<&MSVBVM60.__vbaExceptHandler>]  msvbvm60.__vbaExceptHandler; Struc
004011AC  >- FF25 90104000  JMP DWORD PTR DS:[<&MSVBVM60.__vbaFPException>]    msvbvm60.__vbaFPException
004011B2  .- FF25 50104000  JMP DWORD PTR DS:[<&MSVBVM60._adj_fdiv_m16i>]      msvbvm60._adj_fdiv_m16i
004011B8  .- FF25 3C104000  JMP DWORD PTR DS:[<&MSVBVM60._adj_fdiv_m32>]       msvbvm60._adj_fdiv_m32
004011BE  .- FF25 A8104000  JMP DWORD PTR DS:[<&MSVBVM60._adj_fdiv_m32i>]      msvbvm60._adj_fdiv_m32i
004011C4  .- FF25 24104000  JMP DWORD PTR DS:[<&MSVBVM60._adj_fdiv_m64>]       msvbvm60._adj_fdiv_m64
004011CA  .- FF25 B8104000  JMP DWORD PTR DS:[<&MSVBVM60._adj_fdiv_r>]        msvbvm60._adj_fdiv_r
004011D0  .- FF25 58104000  JMP DWORD PTR DS:[<&MSVBVM60._adj_fdivr_m16i>]     msvbvm60._adj_fdivr_m16i
004011D6  .- FF25 B4104000  JMP DWORD PTR DS:[<&MSVBVM60._adj_fdivr_m32>]      msvbvm60._adj_fdivr_m32
004011DC  .- FF25 AC104000  JMP DWORD PTR DS:[<&MSVBVM60._adj_fdivr_m32i>]     msvbvm60._adj_fdivr_m32i
004011E2  .- FF25 8C104000  JMP DWORD PTR DS:[<&MSVBVM60._adj_fdivr_m64>]      msvbvm60._adj_fdivr_m64
004011E8  .- FF25 70104000  JMP DWORD PTR DS:[<&MSVBVM60._adj_fpatan>]         msvbvm60._adj_fpatan
004011EE  .- FF25 88104000  JMP DWORD PTR DS:[<&MSVBVM60._adj_fprem>]          msvbvm60._adj_fprem
004011F4  .- FF25 30104000  JMP DWORD PTR DS:[<&MSVBVM60._adj_fprem1>]         msvbvm60._adj_fprem1
004011FA  .- FF25 0C104000  JMP DWORD PTR DS:[<&MSVBVM60._adj_fptan>]         msvbvm60._adj_fptan
00401200  .- FF25 D4104000  JMP DWORD PTR DS:[<&MSVBVM60._CIatan>]            msvbvm60._CIatan
00401206  .- FF25 08104000  JMP DWORD PTR DS:[<&MSVBVM60._CIcos>]             msvbvm60._CIcos
0040120C  .- FF25 E8104000  JMP DWORD PTR DS:[<&MSVBVM60._CIexp>]             msvbvm60._CIexp
00401212  .- FF25 9C104000  JMP DWORD PTR DS:[<&MSVBVM60._CIlog>]             msvbvm60._CIlog
00401218  .- FF25 5C104000  JMP DWORD PTR DS:[<&MSVBVM60._CIsin>]             msvbvm60._CIsin
0040121E  .- FF25 78104000  JMP DWORD PTR DS:[<&MSVBVM60._CIsqrt>]            msvbvm60._CIsqrt
00401224  .- FF25 E0104000  JMP DWORD PTR DS:[<&MSVBVM60._CItan>]            msvbvm60._CItan
0040122A  .- FF25 DC104000  JMP DWORD PTR DS:[<&MSVBVM60._allmul>]           msvbvm60._allmul
00401230  .- FF25 EC104000  JMP DWORD PTR DS:[<&MSVBVM60.__vbaFreeObj>]      msvbvm60.__vbaFreeObj
00401236  .- FF25 4C104000  JMP DWORD PTR DS:[<&MSVBVM60.__vbaObjSet>]       msvbvm60.__vbaObjSet
0040123C  .- FF25 20104000  JMP DWORD PTR DS:[<&MSVBVM60.__vbaFreeVarList>]  msvbvm60.__vbaFreeVarList
00401242  .- FF25 D0104000  JMP DWORD PTR DS:[<&MSVBVM60.__vbaR8IntI2>]      msvbvm60.__vbaR8IntI2
00401248  .- FF25 1C104000  JMP DWORD PTR DS:[<&MSVBVM60.#588>]              msvbvm60.rtcRgb
0040124E  .- FF25 44104000  JMP DWORD PTR DS:[<&MSVBVM60.#593>]              msvbvm60.rtcRandomNext
00401254  .- FF25 34104000  JMP DWORD PTR DS:[<&MSVBVM60.__vbaHresultCheckObj>]  msvbvm60.__vbaHresultCheckObj
```

然后我们往下拉，来到VB程序的资源信息中，这里有包含按键、回调。一个比较值得注意的地方是，这里是直接使用回调名称进行回调的，比如程序上有个MyCallBackButton，在代码中可以直接看到这个名字。这样就可以在代码中找到callback的地方。



继续往下拉，就能发现callback的处理代码了。这里使用了常规的callback，和你看到的一样，这里没有说明是什么的callback，我们可以通过改变映射文件进行调用。



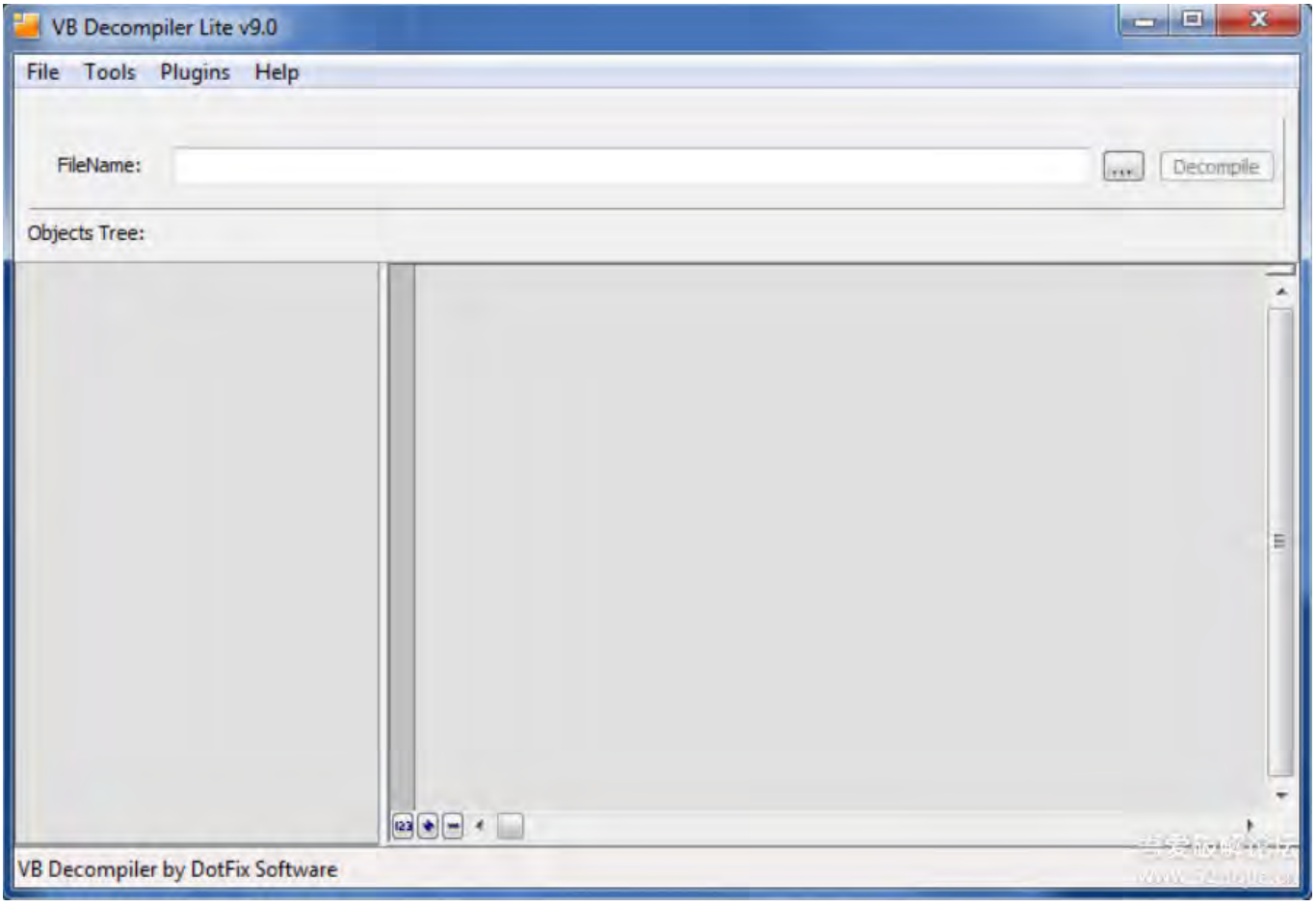
最后，我们来到导入表或者IAT。我们能得到更多有用的信息。

```
00403D12     9E                DB 9E
00403D13     9E                DB 9E
00403D14  .  3C3D0000          DD 00003D3C      Struct 'IMAGE_IMPORT_DESCRIPTOR'
00403D18  .  FFFFFFFF          DD FFFFFFFF
00403D1C  .  FFFFFFFF          DD FFFFFFFF
00403D20  .  343E0000          DD 00003E34
00403D24  .  00100000          DD 00001000
00403D28  .  00000000          DD 00000000      Struct 'IMAGE_IMPORT_DESCRIPTOR'
00403D2C  .  00000000          DD 00000000
00403D30  .  00000000          DD 00000000
00403D34  .  00000000          DD 00000000
00403D38  .  00000000          DD 00000000
00403D3C  .  423E0000          DD 00003E42      Import lookup table for 'MSVBVM60.DLL'
00403D40  .  503E0000          DD 00003E50
00403D44  .  5E3E0000          DD 00003E5E
00403D48  .  683E0000          DD 00003E68
00403D4C  .  763E0000          DD 00003E76
00403D50  .  863E0000          DD 00003E86
00403D54  .  963E0000          DD 00003E96
00403D58  .  4C020080          DD 8002024C
00403D5C  .  A63E0000          DD 00003EA6
00403D60  .  BA3E0000          DD 00003EBA
00403D64  .  CA3E0000          DD 00003ECA
00403D68  .  04020080          DD 80000204
00403D6C  .  DE3E0000          DD 00003EDE
00403D70  .  EC3E0000          DD 00003EEC
00403D74  .  043F0000          DD 00003F04
00403D78  .  123F0000          DD 00003F12
00403D7C  .  223F0000          DD 00003F22
00403D80  .  51020080          DD 80000251
00403D84  .  53020080          DD 80000253
00403D88  .  343F0000          DD 00003F34
00403D8C  .  423F0000          DD 00003F42
00403D90  .  543F0000          DD 00003F54
00403D94  .  683F0000          DD 00003F68
```

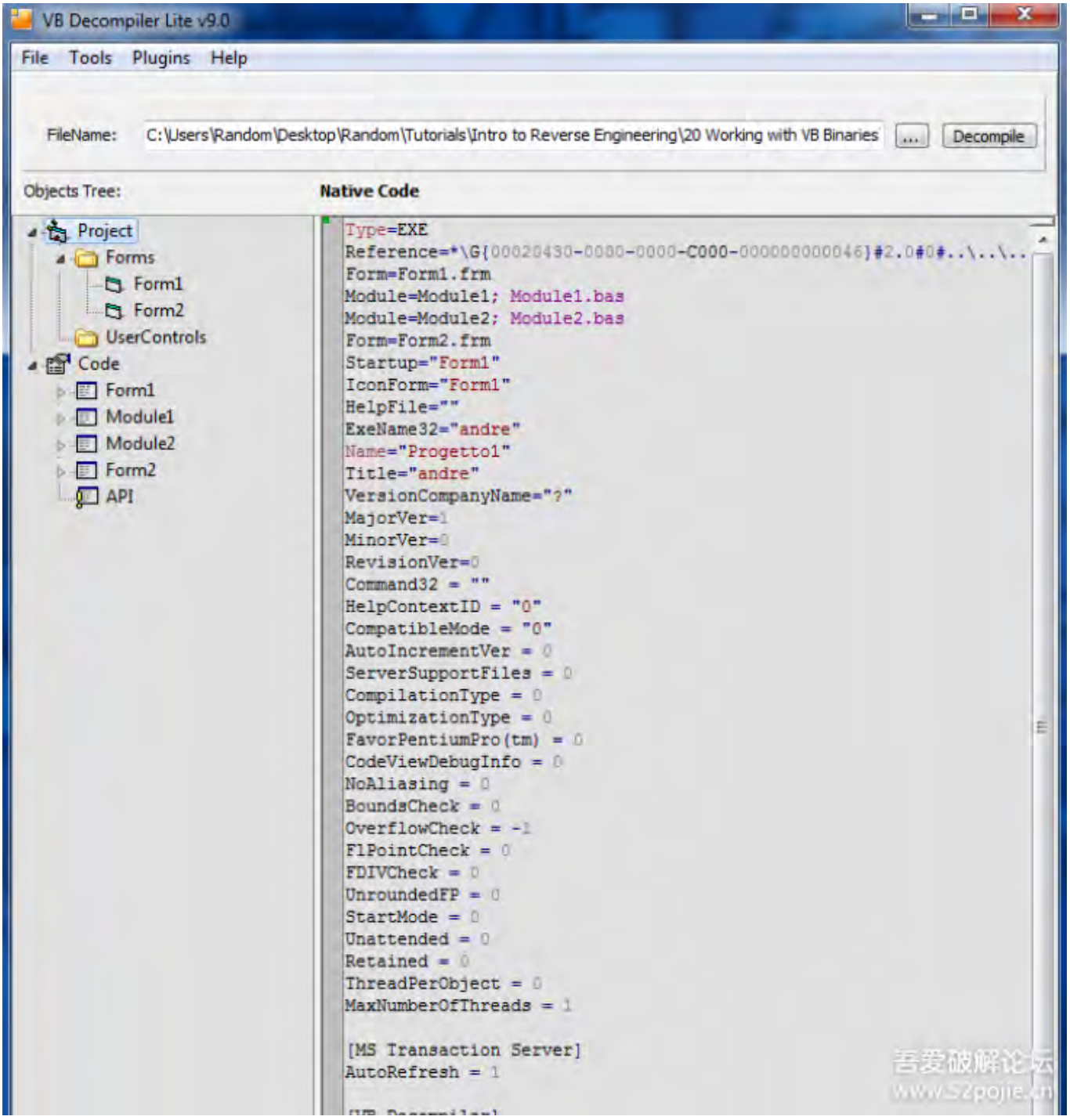
当然上面的信息并不能完成这次的破解。我们需要使用其他工具

VB Decompiler Lite

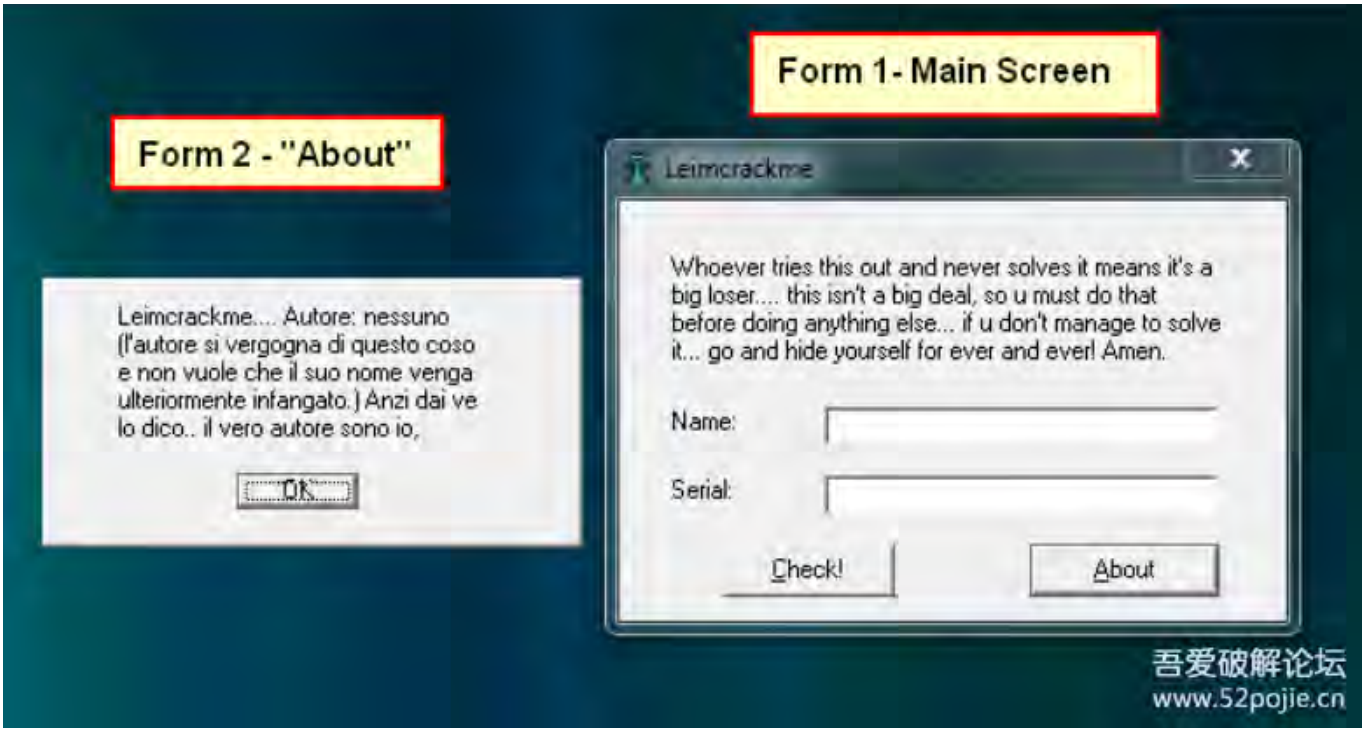
这个软件有两个版本，一个是Lite，一个是Pro。Lite版本是免费的，所以我们有附带该软件。VB Decompiler可以把p-code编译的VB程序进行反编译，反编译为VB语言。这样我们就可以看到VB程序的资源文件。好，现在打开VB Decompiler程序。



打开 CrackmeVB1.exe程序，然后点击Decompile按钮

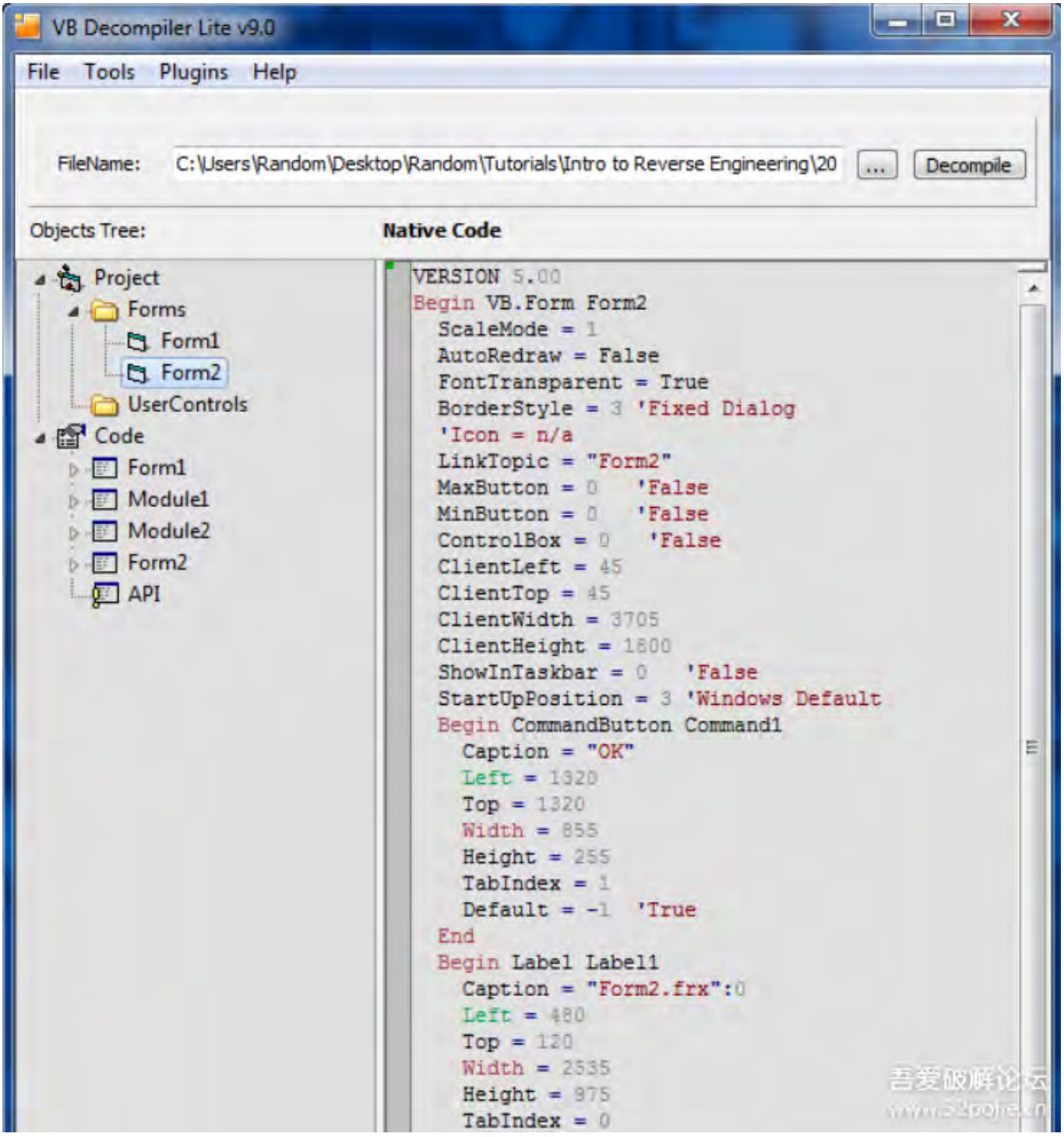


然后可以看到许多信息了，注意左边的Forms文件夹，这里有两个，所有的资源文件都在这里面，我们一个一个看，一个是主窗口，一个是关于窗口。



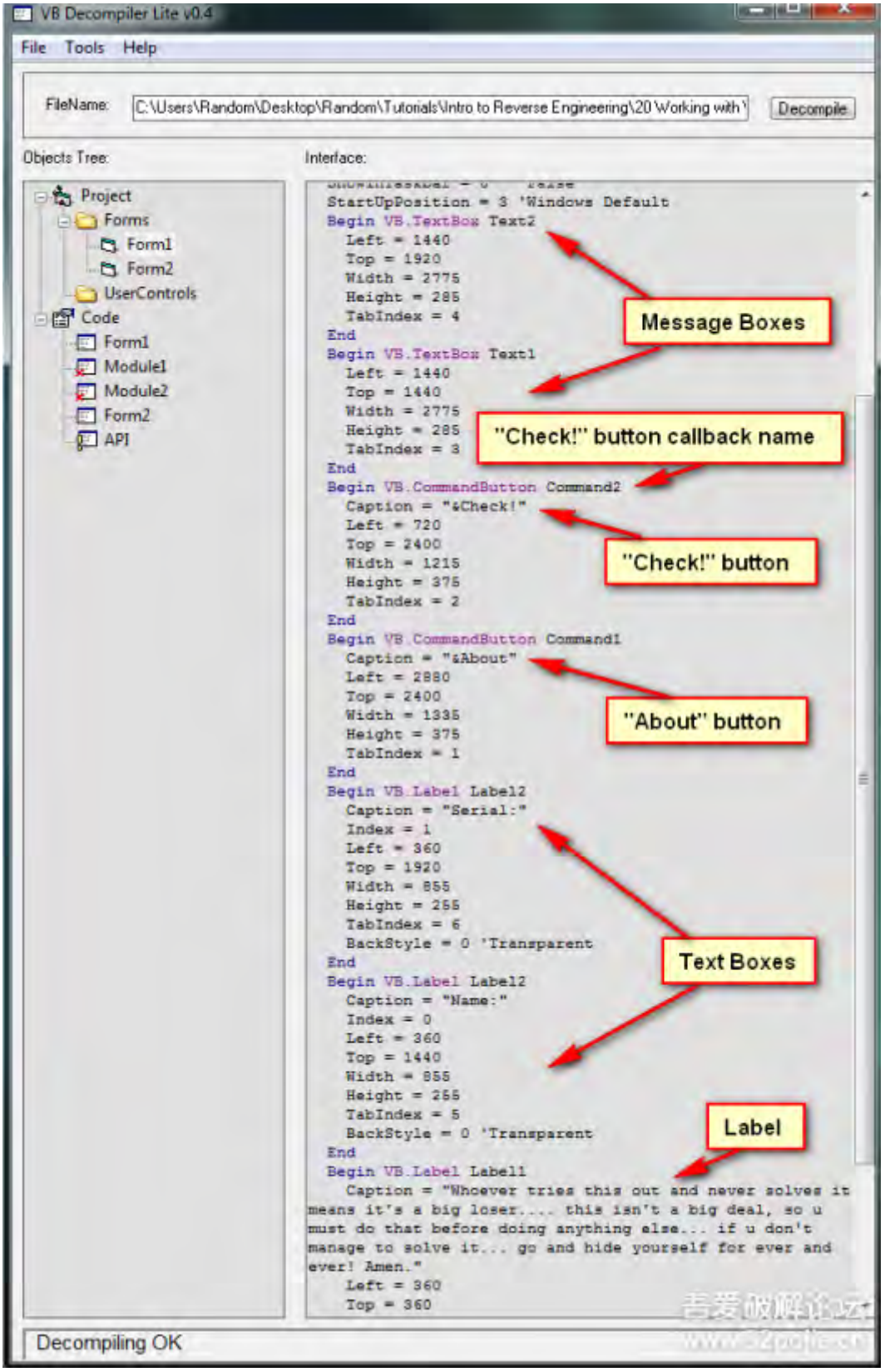
你启动程序的时候已经发现这里点击的About按钮后出现的是其他信息，而且about窗口上的OK按钮是不可点击的。如果你跟着我的教程走，你就会知道怎么来进行破解了。

双击Form2后，我们可以看到资源信息了

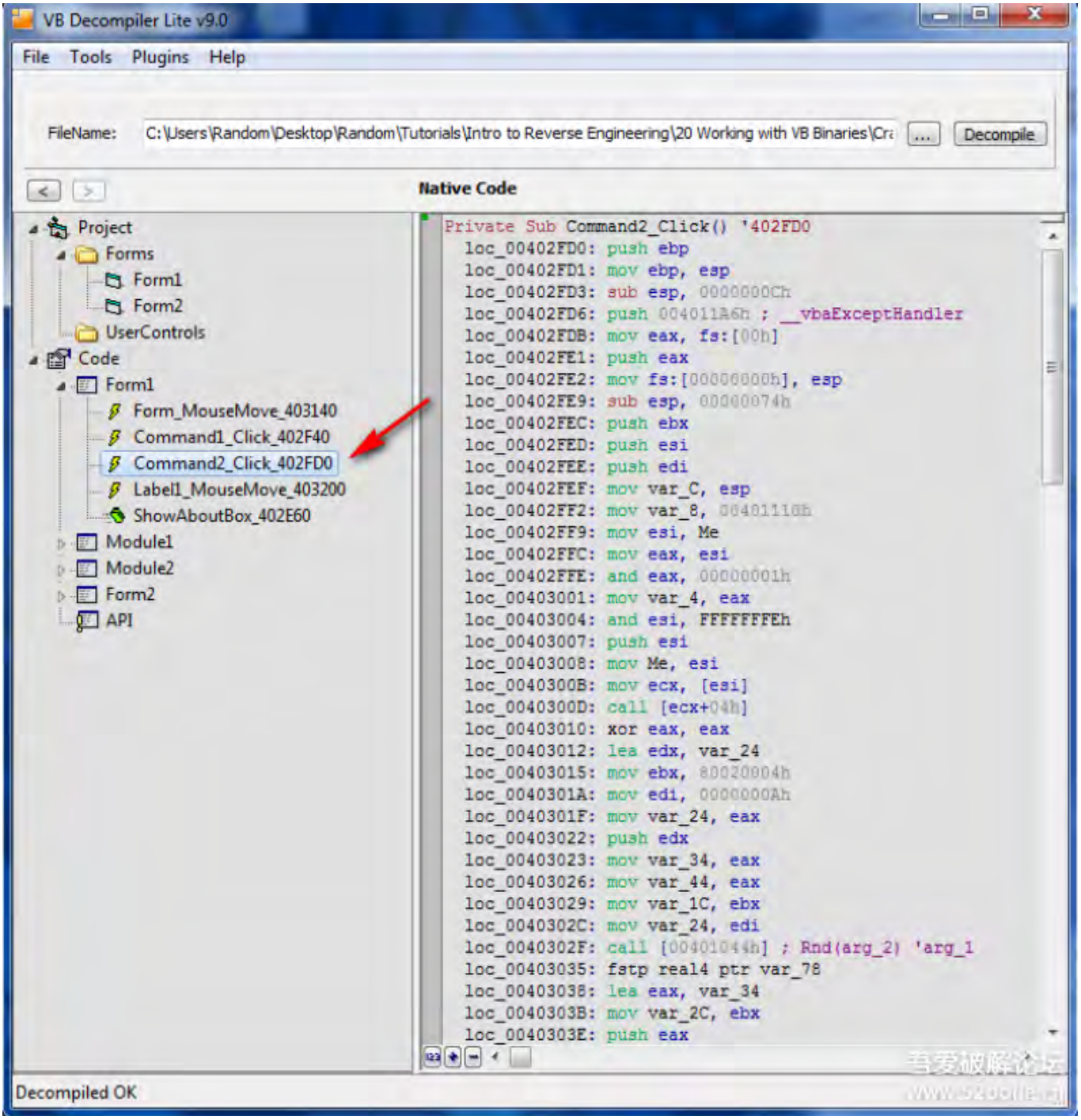


这里我们可以看到一个按钮的字串是OK，另外一个信息的是这里的callback被叫做Command1。

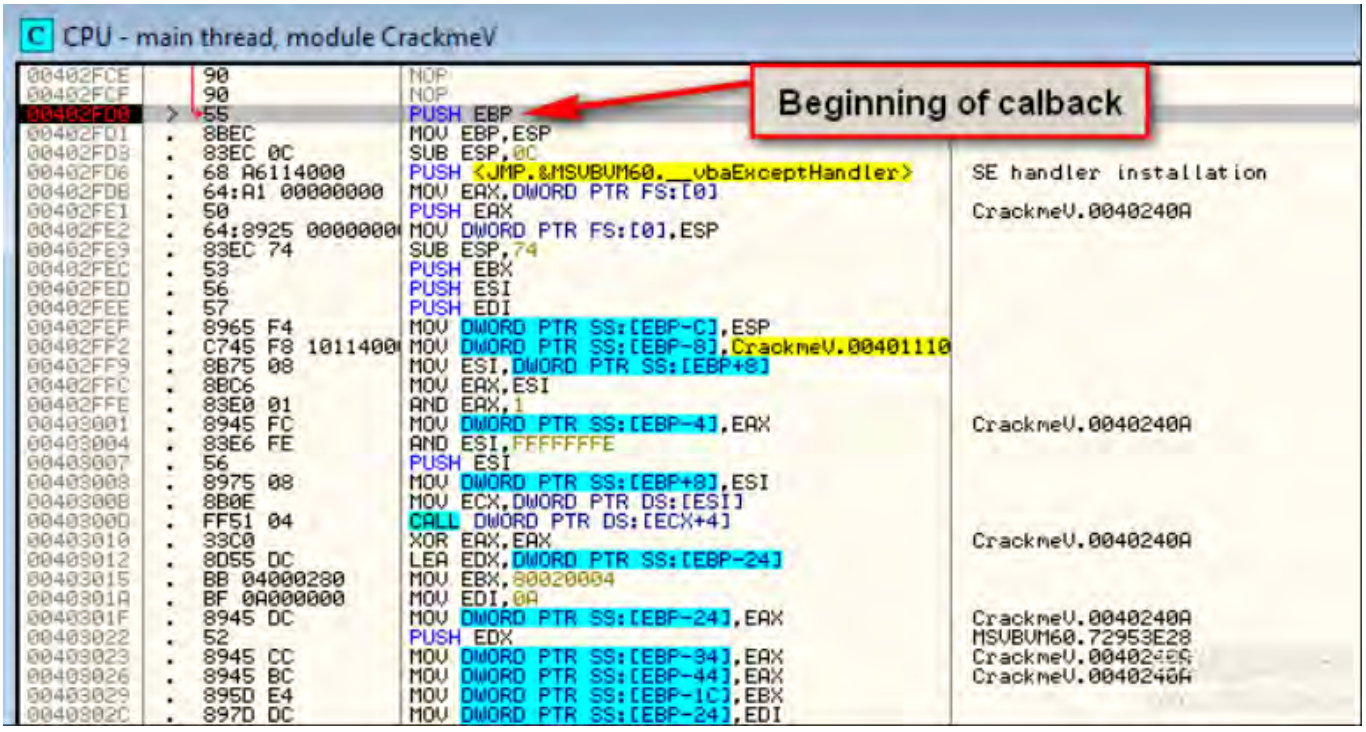
双击Form1，我们来到主窗口



现在我们在破解前知道了许多信息。一个重要的按钮叫 Check!，他的callback叫Command2，接着往下看，你可以看到Code中有对应的callback，而且知道偏移地址（Command2_Click_402FD0），如果你双击后就可以看到VB代码



现在我们知道Check按钮的偏移地址了，然后使用OD打开程序，来到402FD0处

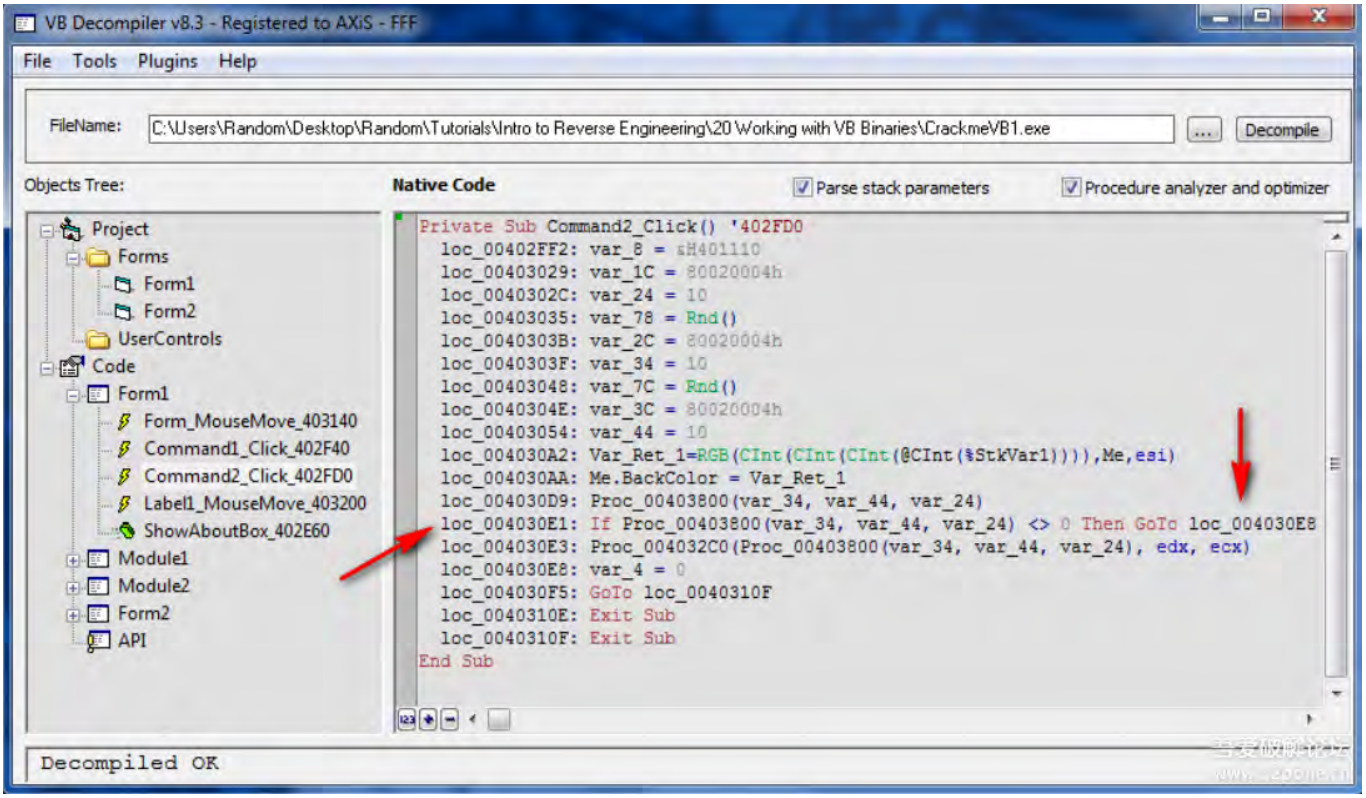


如果你在这里设了断点，运行程序，输入用户名和序列号，点击Check后就会停在这里了，后面的破解就很简单了。不再进行详细教程。

VB Decompiler Pro

我想告诉你们使用p-code进行查找callback，这样就必须使用VB Decompiler Pro，当然这个不是免费软件，所以不附带（译者注：我在52Pojie中找到，并附上），使用VB Decompiler Pro

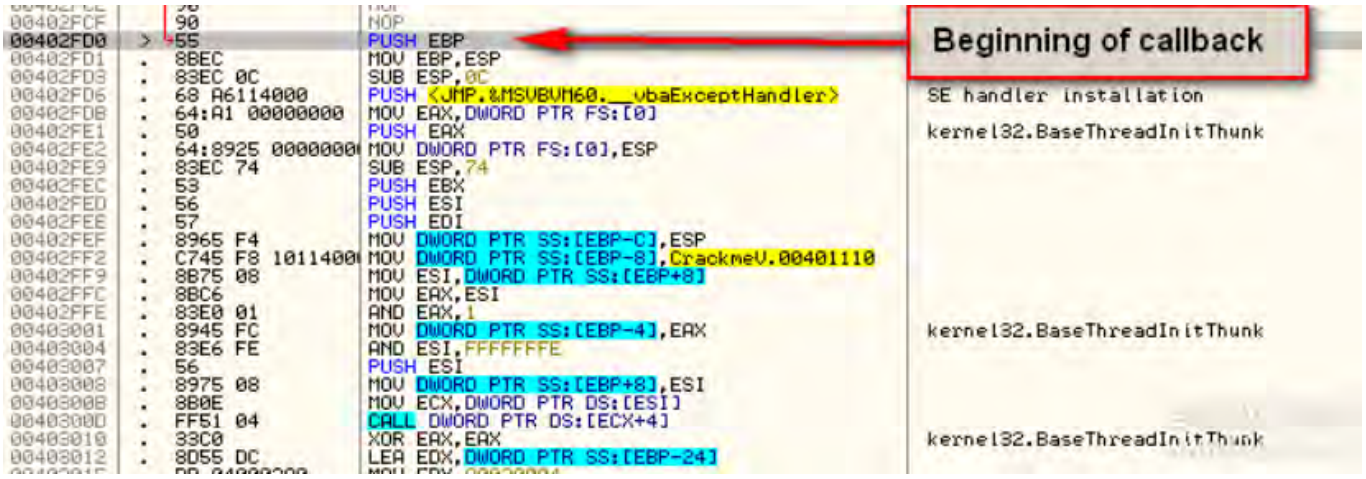
打开CrackmeVB1，然后反编译。



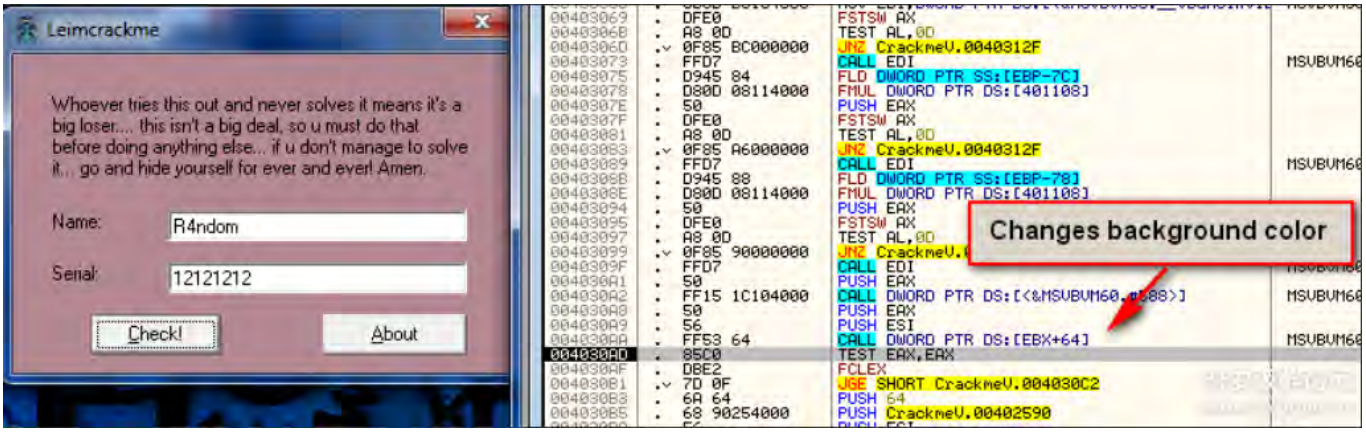
这里我们就能看到p-code方法和callback了，所有地址都显示了要做什么东西。

现在进行补丁

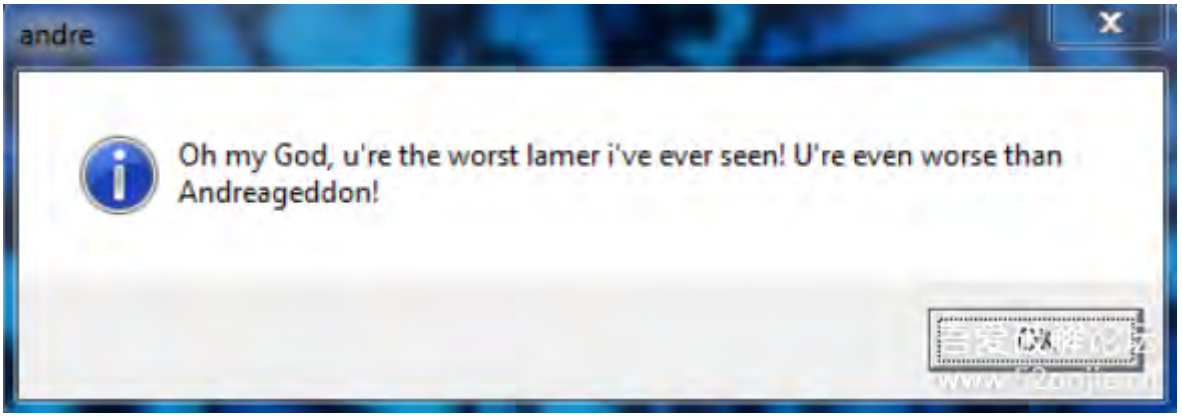
使用OD打开程序并来到402FD0处



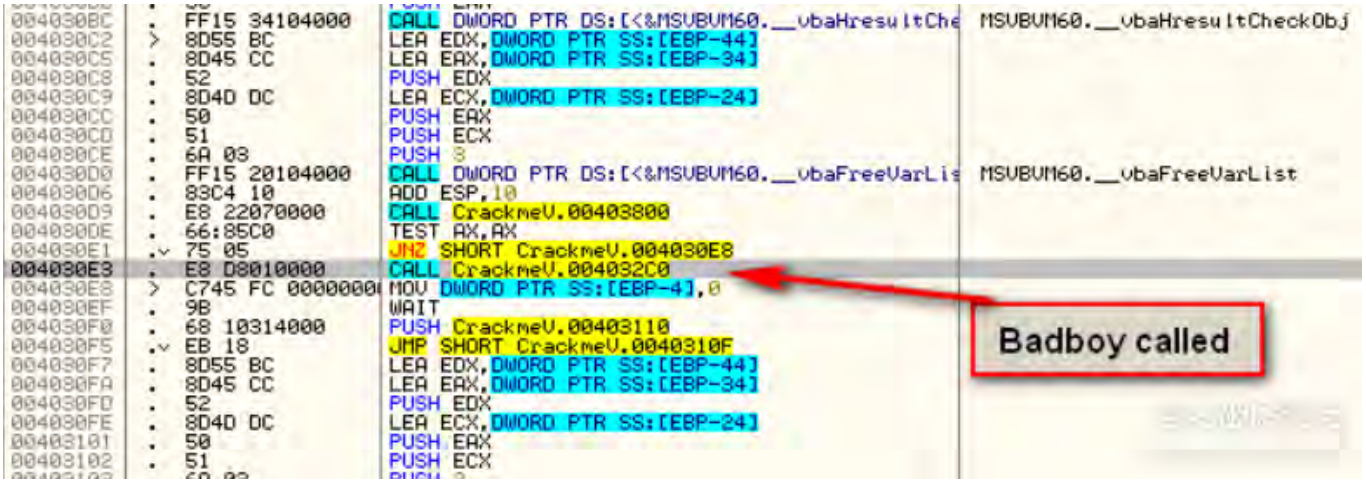
设置断点，然后跑起来，来到4030AA处，背景颜色变了，这是从p-code中知道的。



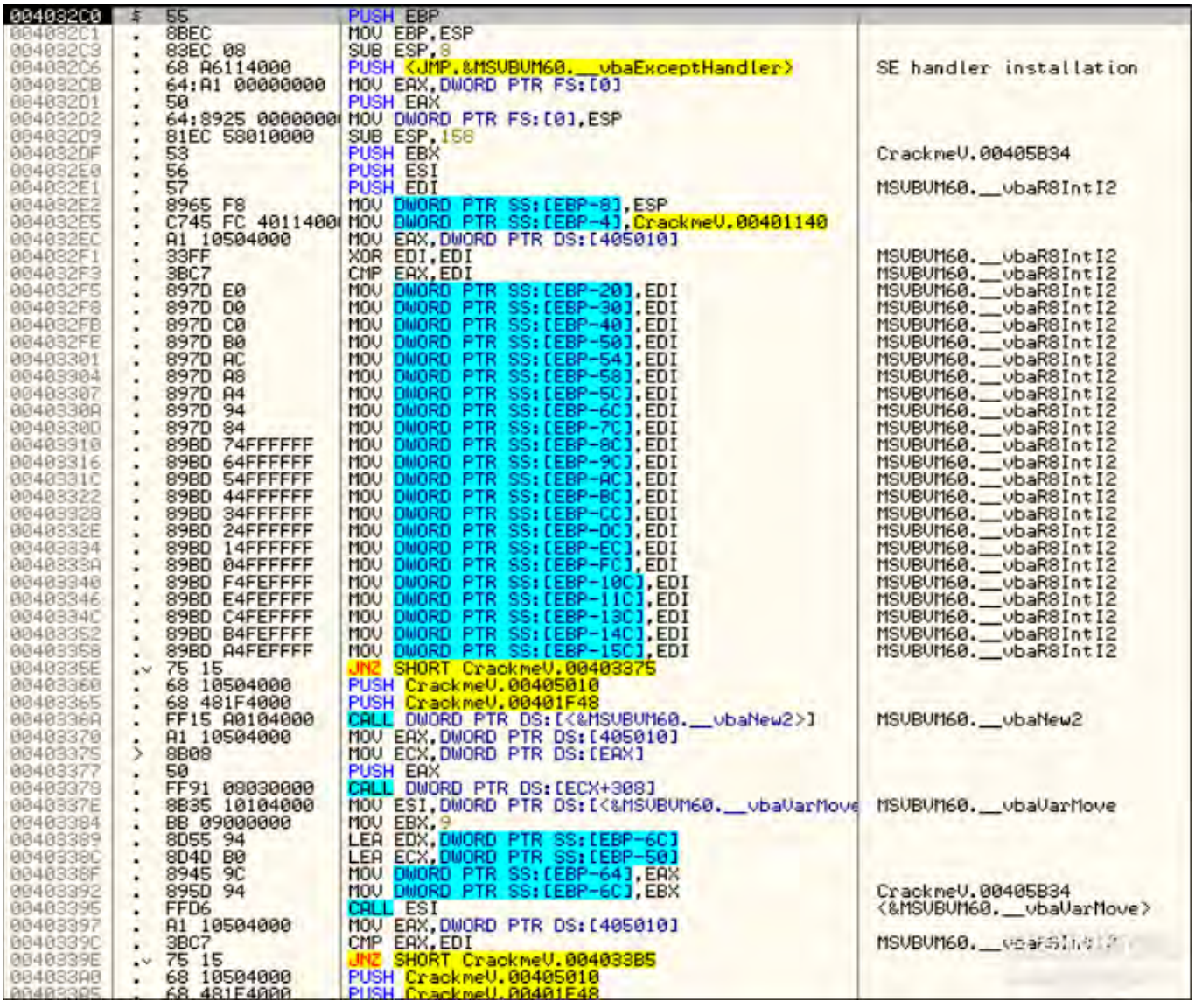
在4030E3处，弹窗坏消息。



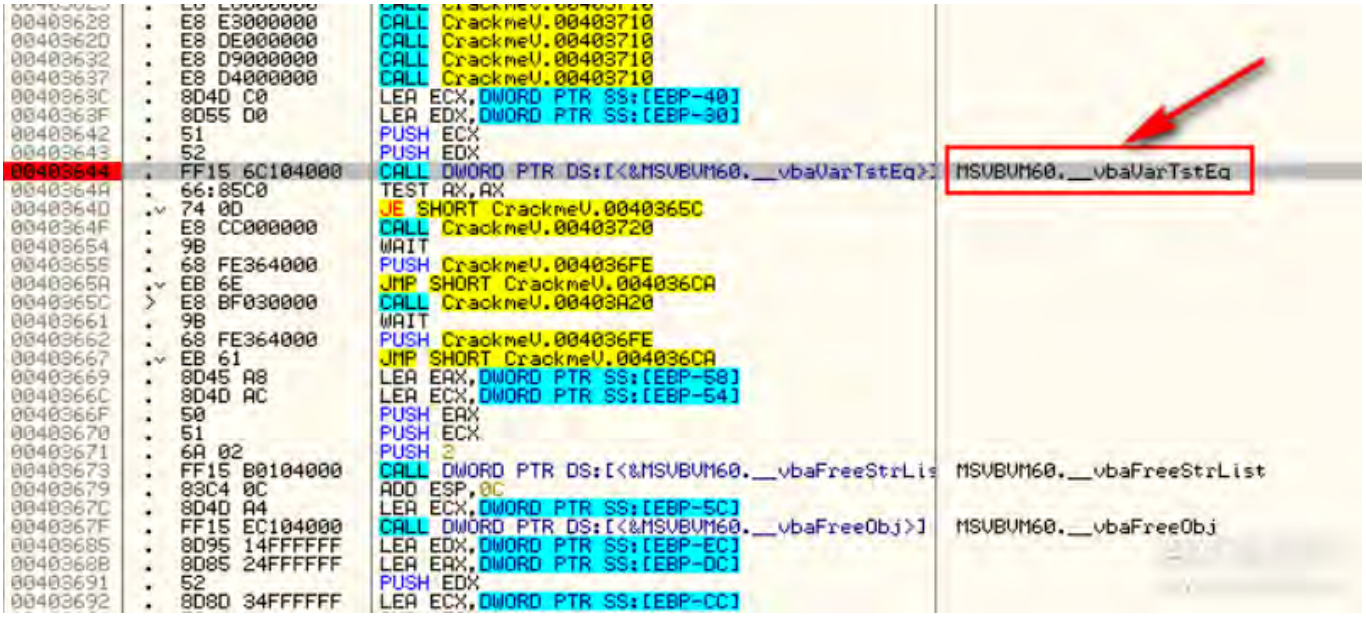
我们看这里的代码可以发现有比较跳转的代码



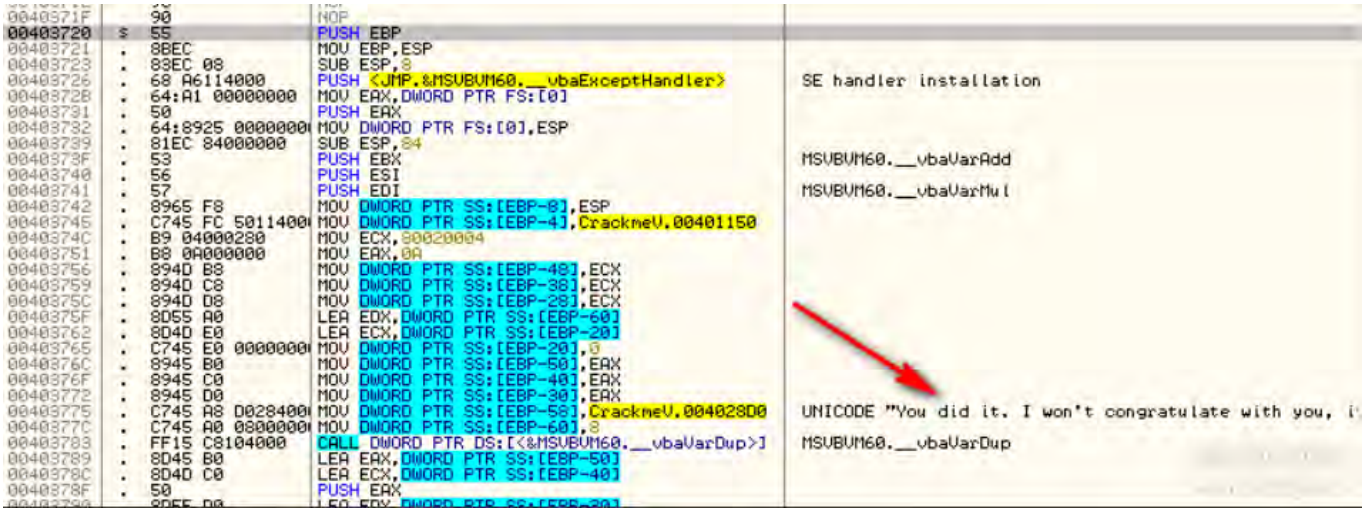
把断点设在4030E1处，重启程序，点击Check后，OD断下来，改变0标志位。这里不会跳出坏消息。这样来看就是4030C0这个Call中进行了判断和处理坏消息。我们把断点设在4030C0处，重启程序，然后一步一步调试。



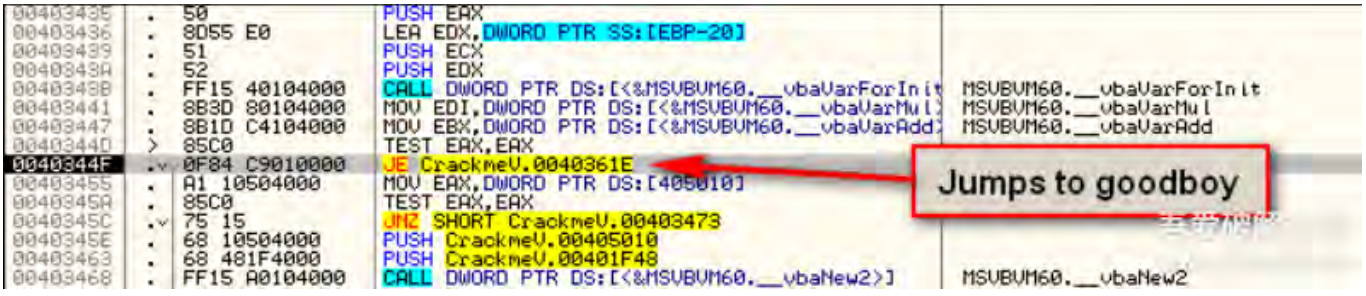
我们慢慢看，就能看到Call VB DLL的方法，下拉代码窗口可以看到4033644处



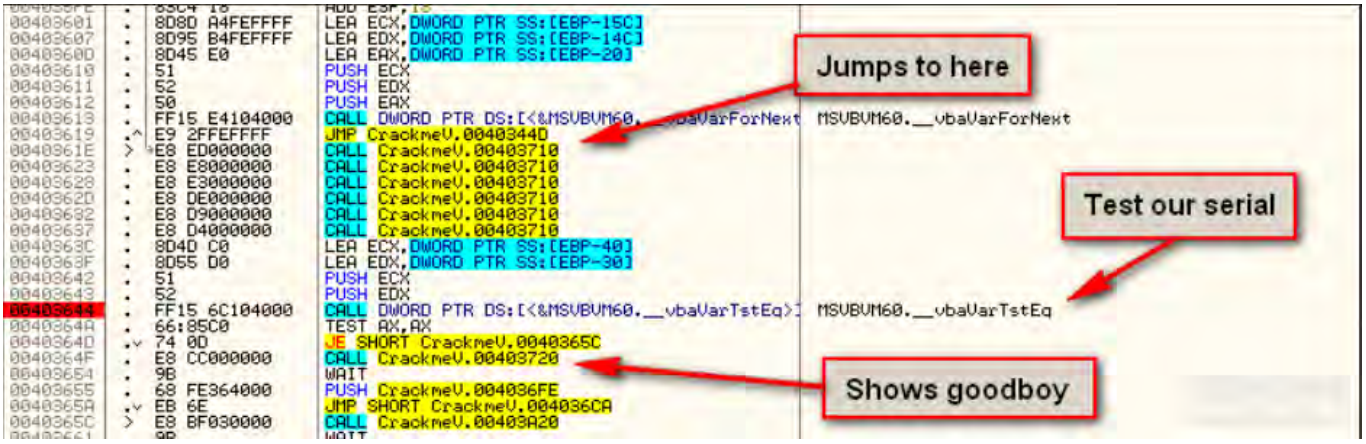
vbaVarTstEq看上去像strcmp。然后前面传来2个参数。然后在40364F处还有一个call，点进去看看



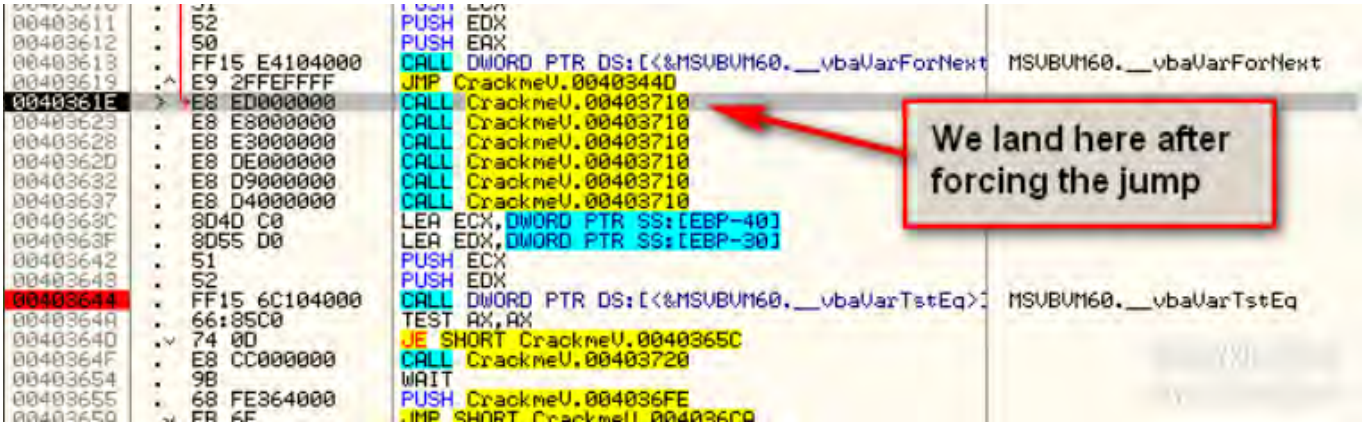
现在我们必须要跳转到403644处。这样就能得到好消息。所以我们修改40344F的JE跳转



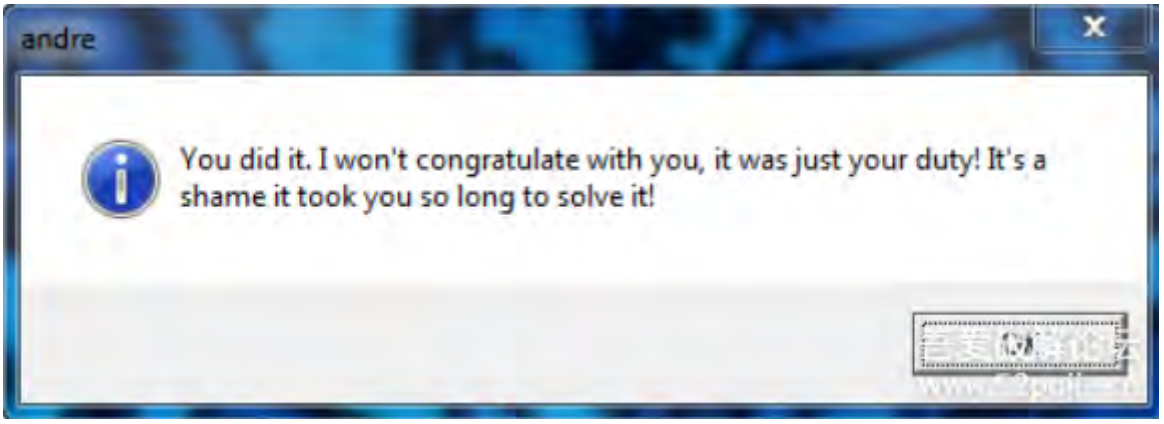
这就是我们希望代码走到的地方



现在把断点设到40344F，跑程序，修改0标志位，强制跳转到40361E处



然后来到40364D处后，我们得到了好消息，这就是破解这个程序的方法



有兴趣的可以去找到VB 调用API的部分与C/C++的API对比，其实很容易找到相同的东西。

附件其他VB程序请自己练习，这里就不在进行讲解了