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）。

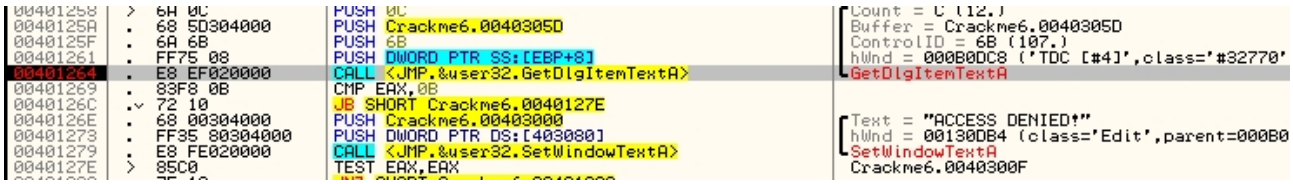
skill$（译者注：小标题中的第三个字母为 1，这里又变成 i，我不知道是不是作者故意的）的最高境界是，如果不能够从应用中提取代码，就必须自己编写代码来提供可用的序列号。意识是你必须完全理解程序是如何解密序列号并将其与你输入的进行对比。你必须自己写程序来完成相同的功能，仅在逆向领域中，有很多次是用汇编语言写的。

很明显，该方法的主要缺点是 skill$的复杂难懂。

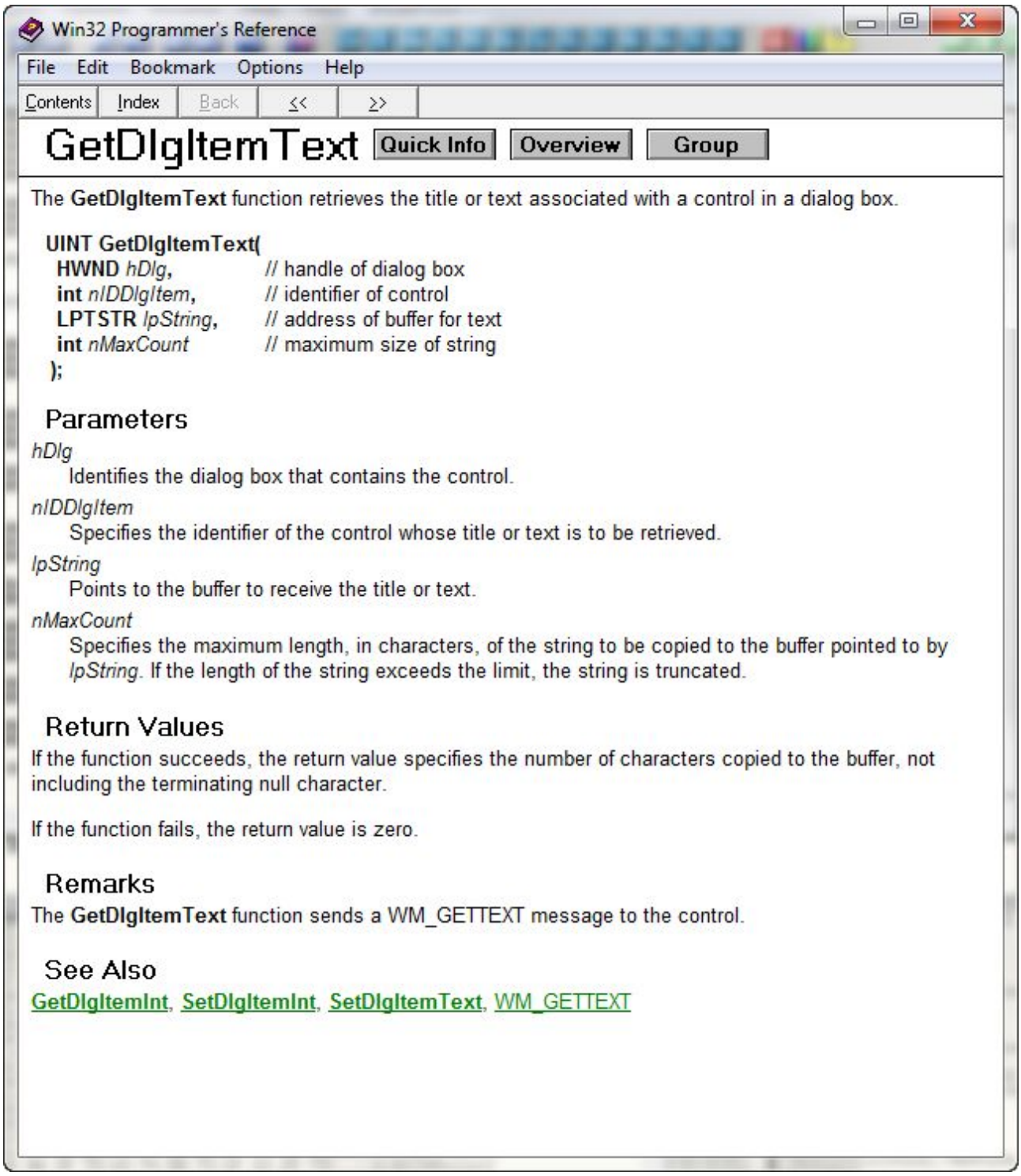
那么，鉴于我们对逆向工程级别的新的理解.....

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:



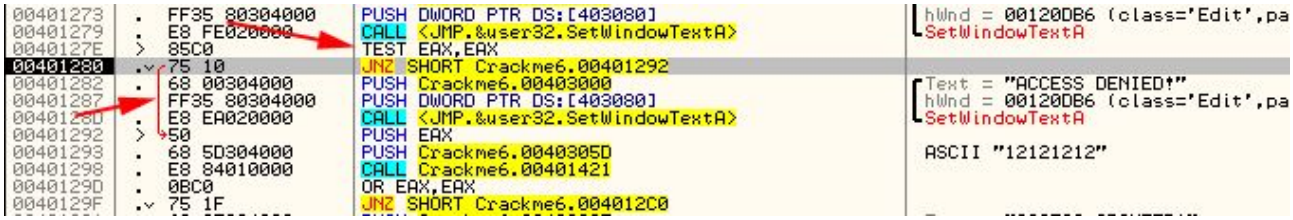
需要重点注意的是：其中一个参数是一个指向缓冲区的指针，该缓冲区是用来存储密码的（lpString）。返回值保存在 EAX 中，它保存的是字符串的长度：

> **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（Olly 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:

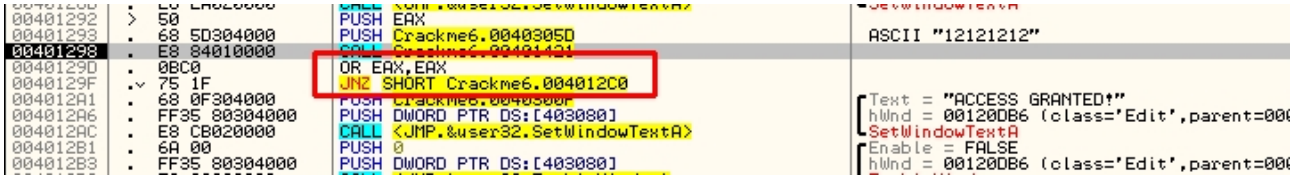
看吧!! 咱们已经了解了一些东西了，我们的密码最多只能有 11 个数字。因为我们的密码少于 11 个数字，所以咱们继续并让跳转实现。（如果你输入的密码大于 11 个数字，重启应用然后再输入一个小于 11 个数字的密码，再单步到我们所在的位置。）



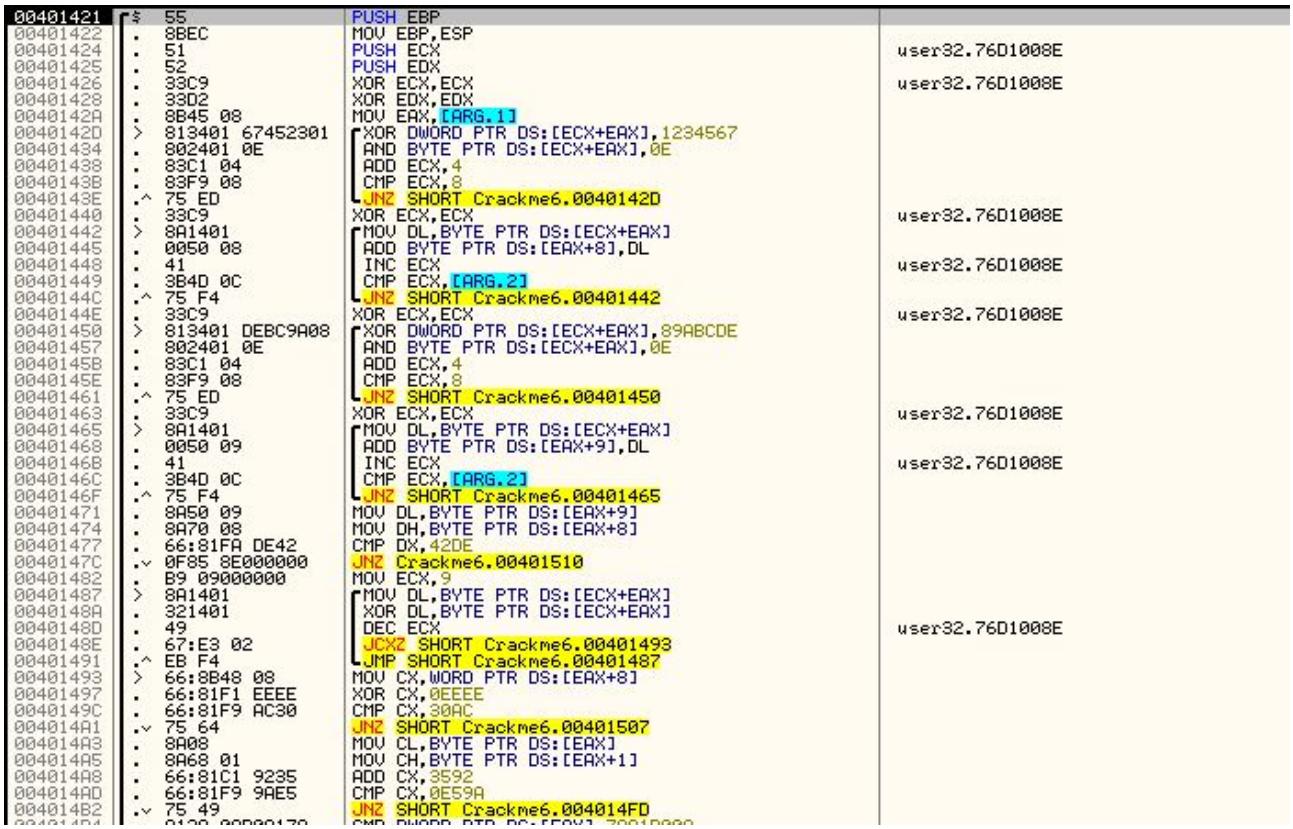
接下来你注意 EAX 的值，它仍然保存着密码的长度，并被测试是否为 0，如果不是 0 的话，就跳过第二个坏消息。那么现在我们知道了第一个坏消息是当我们的密码长度小于 11 时显示，第二个坏消息是在密码为空的时候显示。

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:

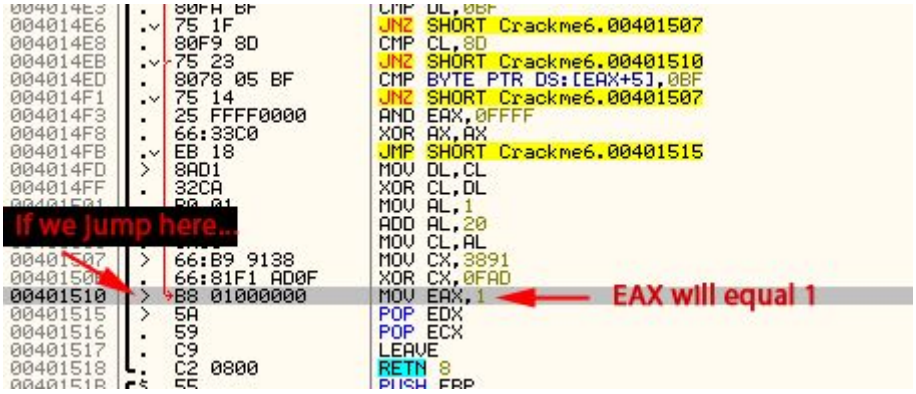
首先要注意的是（在地址 18FAB0 处）是长度（8）被压入栈，其次是在地址 18FAAC 处的地址 40305D 被压入栈，O11y 也向我们显示了“12121212”也就是我们的密码。现在我们知道了，我们的密码是存储在内存的 40305D 处。这一点在后面会很重要。后面，O11y 会将这两个值叫做 ARG.1 和 ARG.2，因为它们是传递给函数的参数。这两个值入栈以后，我们就可以调用 401298 处的主要注册程序了（我们之所以知道这个，是因为所有重要的 比较/跳转 指令组合的前面都有个 CALL，所以它的结果将决定我们是跳到好消息还是坏消息）:



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：

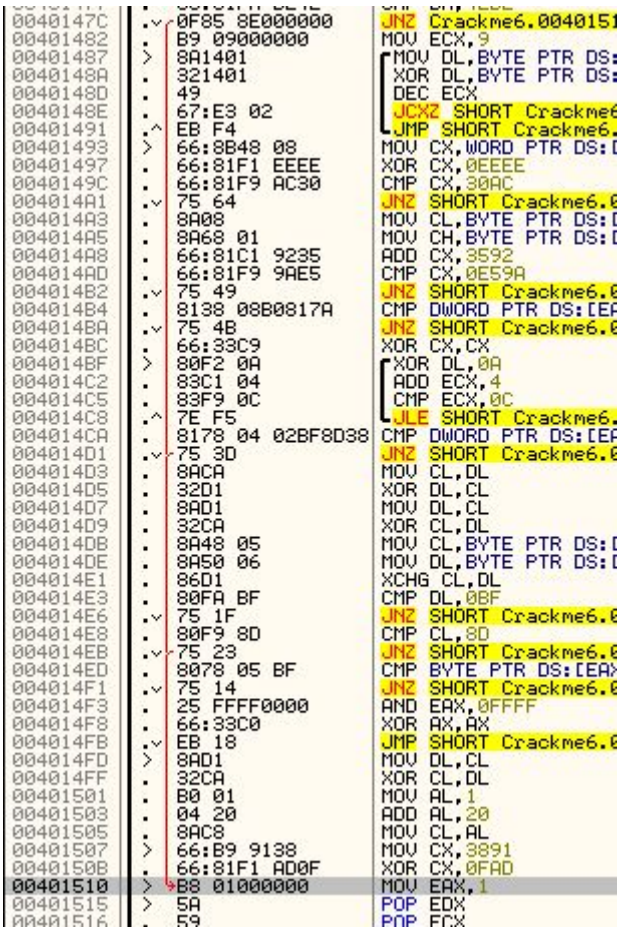


哇噢，看起来不少呢，尤其是你很可能对汇编语言只是个半吊子。但也不是不可能。我常用的招是到程序的最后面，我们知道它返回时 EAX 必须等于 0，看看是什么完成了这项工作以及是什么阻止了它发生，然后再回头用我们的方法。向下滚动直到你看到函数的 RETN 指令：



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

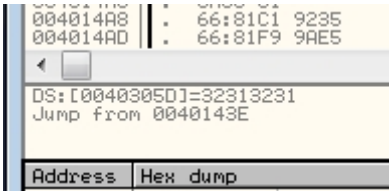
始打一个 SKILLED 级别的补丁。回到上面程序（译者注：这里的程序是指那个验证函数，本章大部分都是这个意思，读者应自己做分别）的起始处，实验一下：

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：

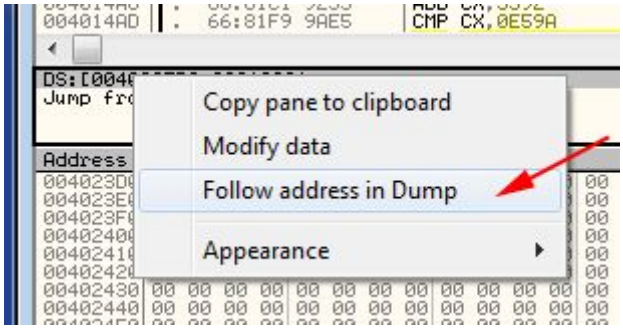
```
XOR DWORD PTR DS:[ECX+EAX], 1234567
```

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”。

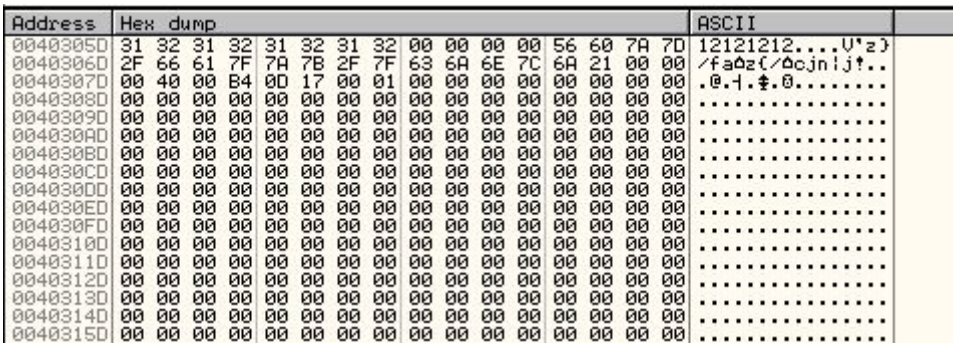
我们可以观察到这一过程。首先，要确保我们依然暂停在 40142D 那行，看数据窗口的上面那一块，它会告诉你地址 ECX+EDX（40305D）是什么，以及它存储的值是什么（32313231），用 ASCII 码表示就是“2121”（要记得数据存储序列😃）：



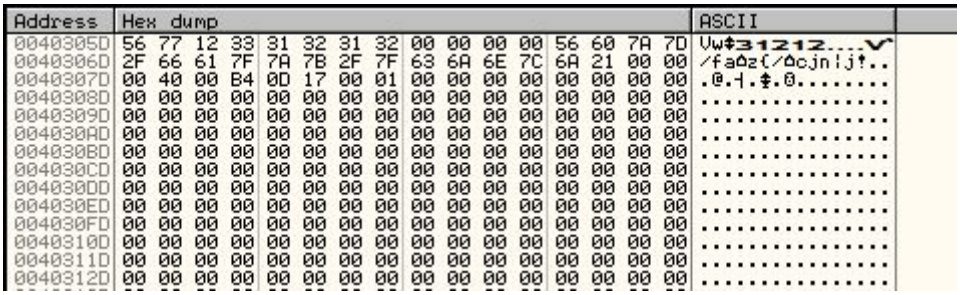
现在选中“DS:[0040305D]=32313231”这行，右键选择“Follow in dump（数据窗口中跟随）”，然后我们就能看到我们密码当前存储的实际内存内容：



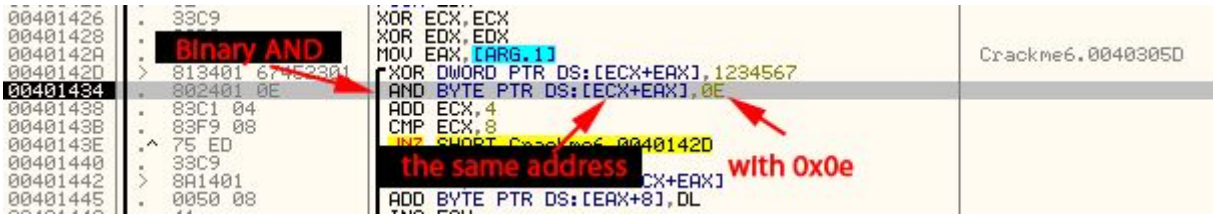
现在，数据窗口显示的就是从 40305D 开始的内存内容。前面 8 个字节就是我们的密码。记住，我们当前所在的行正准备取该地址的前四个字节（31，32，31，32），然后与 0x1234567 进行 XOR 操作，之后再将结果存回到该内存区：



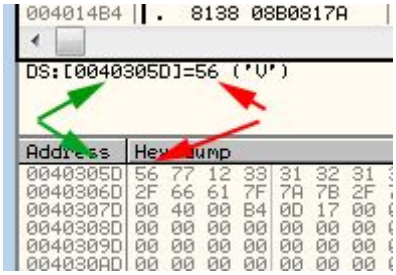
咱们继续，单步步过一次，然后你就会看到我们密码的前四个字节已经变了，和 0x1234567 进行 XOR 的结果：



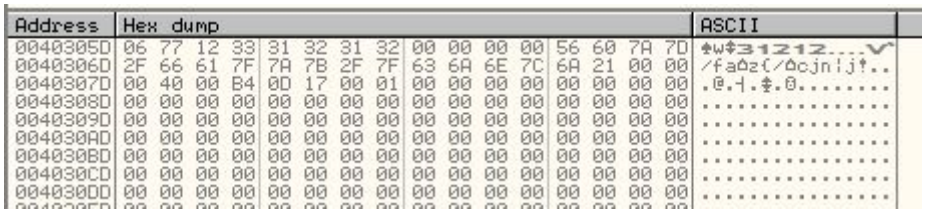
好，继续下一行代码：



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：

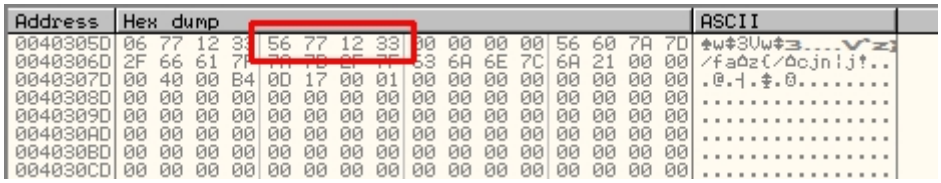


它告诉我们将受到影响的地址是 40305D，该地址的（当前）值是 56。继续，单步执行一次，你会发现第一个数字又变了：

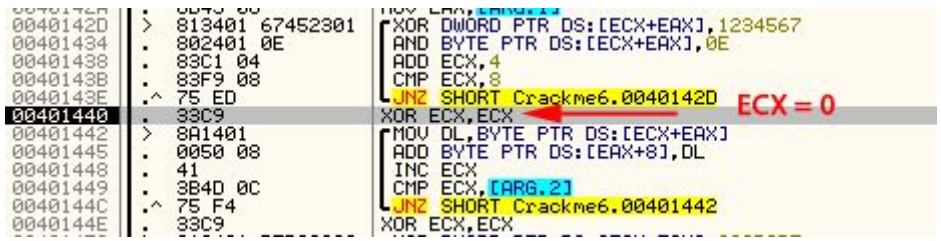


现在咱们知道了 0x56 与 0x0E 进行 AND 操作的结果是 0x06。咱们继续跋涉这段困难的代码：

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：

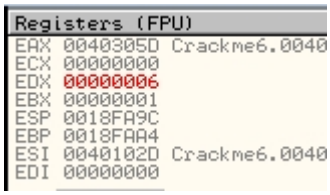


它们也会被修改。第五字节也会被再次修改，因为它将与 0x0E 进行 AND 操作。循环结束后，下一条指令也就是 401440 处的指令仅仅是将 ECX 值置 0：



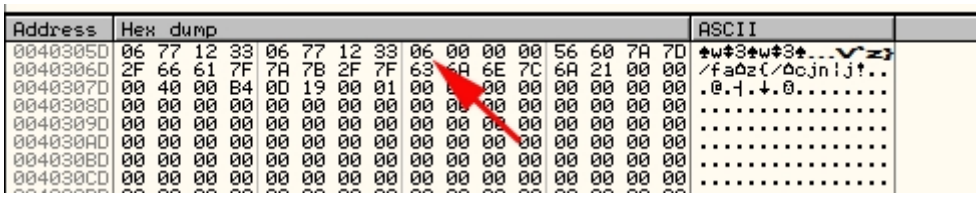
现在咱们来看看接下来的几条指令：

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：

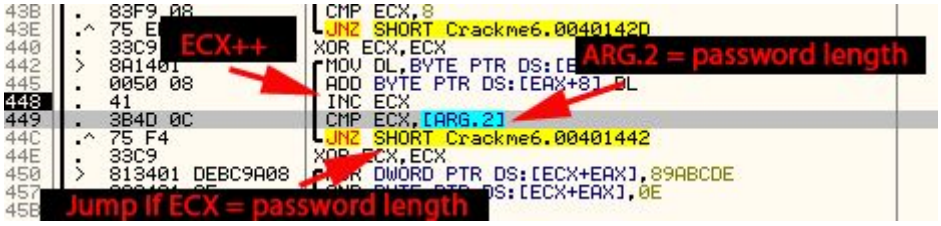


然后我们将 DL 中的数字与 EAX+8 中的内容相加，也就是 EAX 的第八个字节，并将结果存回第八个字节：

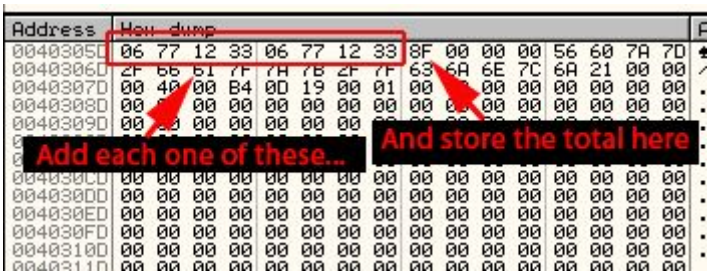
这里，我们能够看到那个字节被修改了：



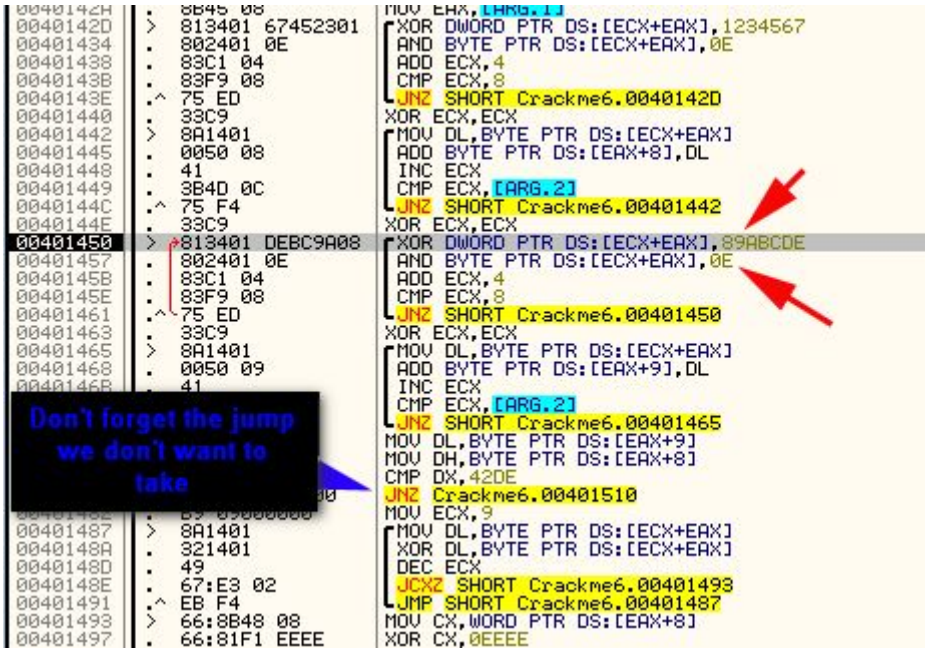
它是将 buffer 中的第一个字节（6）与第八个字节（0）相加，结果得 6（译者注：不知道读者注意到没有，上面的图片中箭头明明指的是第九个字节好不好，那是不是作者弄错了呢？我觉得是有问题的，因为如果按从 0 开始数，那么第一个字节就应该是 77，如果不从 0 开始数，那么第八个字节就应该是 33。那该怎么办呢？其实不用管第几个了，我们知道定位到该字节是按 EAX+8 的结果算的，因为 EAX 的值是 40305D，所以 40305D+8=403065。40305D 就是 06 那个字节，我们往右边数，数到 403065，刚好就是箭头指的那个 06）。如果我们密码的长度大于 8，这就会让我们密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。



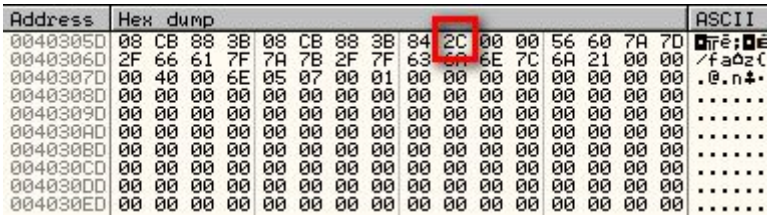
你单步执行这个循环，就可以观察到内存的变化：

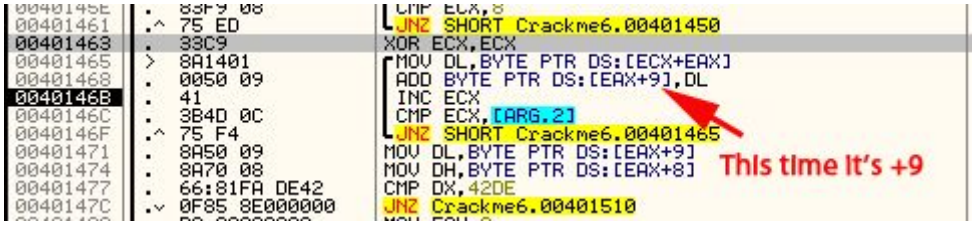


在循环结束后，我们再一次将 ECX 设置为 0，并进入了与第一个循环相似的循环中，这次将每个四字节数据与 0x89ABCDE 进行 XOR 操作。



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。

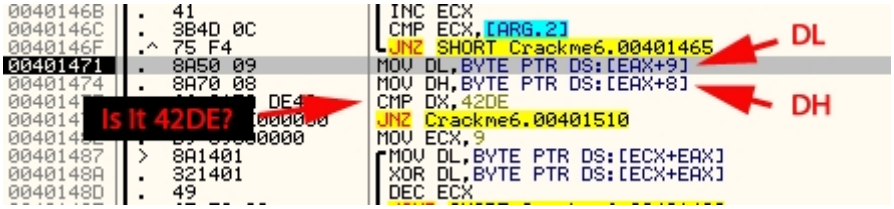
2、第一个字节与 0x0E 进行 AND 操作，第五个字节也是一样。

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:

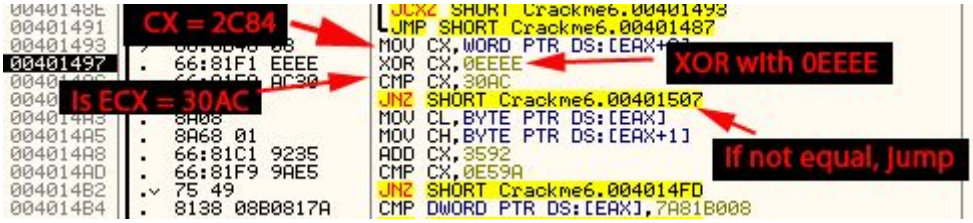
```
00401474  .  8A70 08          MOV DH,BYTE PTR DS:[EAX+8]
00401477  .  66:81FA DE42     CMP DX,42DE
0040147C  .v 0F85 8E000000    JNZ Crackme6.00401510
00401482  .  B9 09000000      MOV ECX,9
00401487  >  8A1401           MOV DL,BYTE PTR DS:[ECX+EAX]
0040148A  .  321401           XOR DL,BYTE PTR DS:[ECX+EAX]
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
```

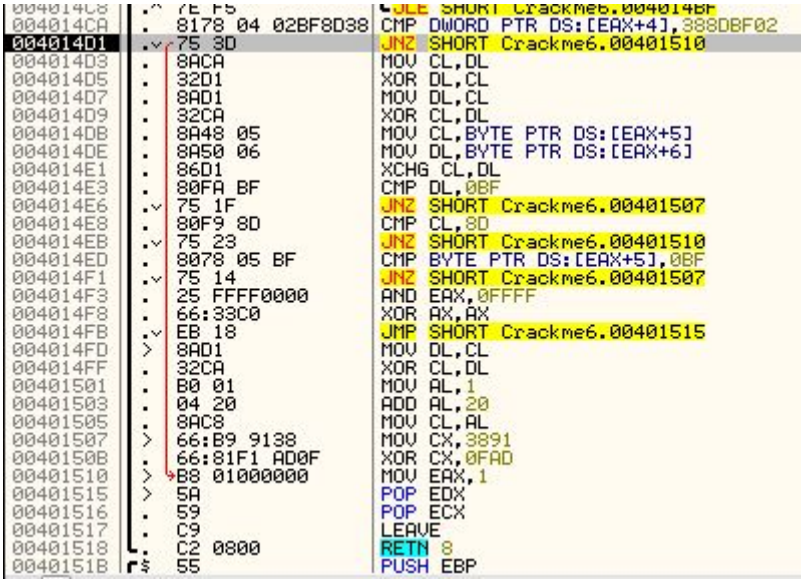
这是将我们密码 buffer 的第一和第二个内存内容拷贝到 CL 和 CH，本例中是让 ECX 等于 CB08。与 3592（十六进制）相加后再与 E59A 进行比较。如果它不等于该值就跳转:

```
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
0040151B  r$ 55              PUSH EBP
```

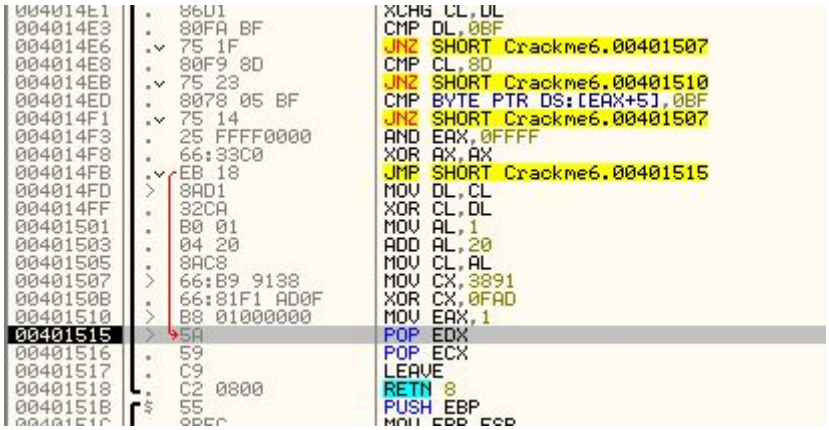
这个和上面做的事情是一样的。完成了另一个检测，以确保我们是合法的到达此处。我们明显不想让这个跳转实现，所以我们通过修改 0 标志位再次的帮了 O11y 的忙。然后我们顺利通过了另一个检测，这个是从 4014A3 到 4014AD。通过修改 0 标志位，我们也跳过了这个 JNZ，最终来到了这里:

```
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
0040151B  r$ 55              PUSH EBP
```

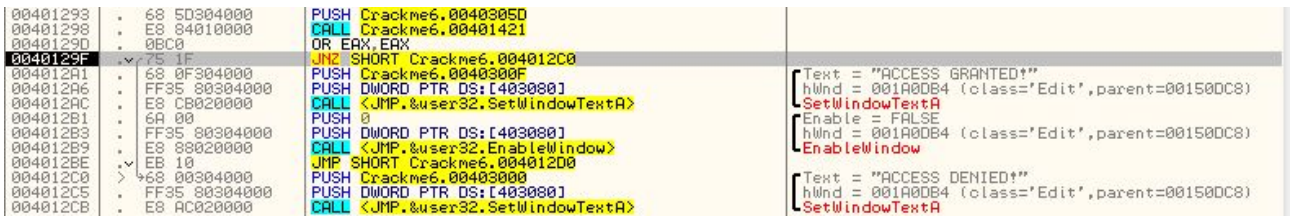
第一行代码 CMP DWORD PTR DS:[EAX], 7A81B008 做了另一个检测。在对密码做完了所有的操作以后 ，最后第一个四字节等于 7A81B008。如果不是，我们就会跳到坏消息那:



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？