R4ndom's Tutorial #13: Cracking a Real Program

by R4ndom on Jul.12, 2012, under Beginner, Reverse Engineering, Tutorials

Introduction

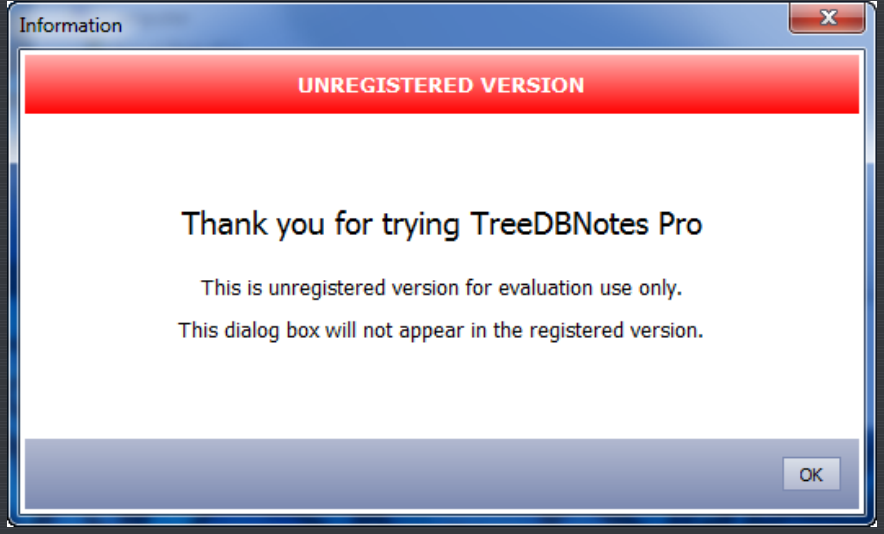
In this tutorial we are going to take off the training wheels and crack a real program. This program has a time restriction, and after this time, it will not work anymore. We are going to patch it to think it is registered. The target is included in this download (I am not stating the name of the program as the purpose of this tutorial is not to get a 'cracked' program but to learn how to do it.) Like all commercial programs, if you plan on using them, you really should consider buying it. People put a great deal of time into apps and they deserve to be compensated. In an attempt to not make this series about 'getting cracked software', I tried to get a program that no one would really want, so I downloaded this app, which had the least amount of downloads last week on Download.com. To be totally honest, after cracking the program in this tutorial, I liked it so much I paid for the registration and now use the app legitimately. Just goes to show you you can't judge an app by it's downloads.

Well, on with the show…

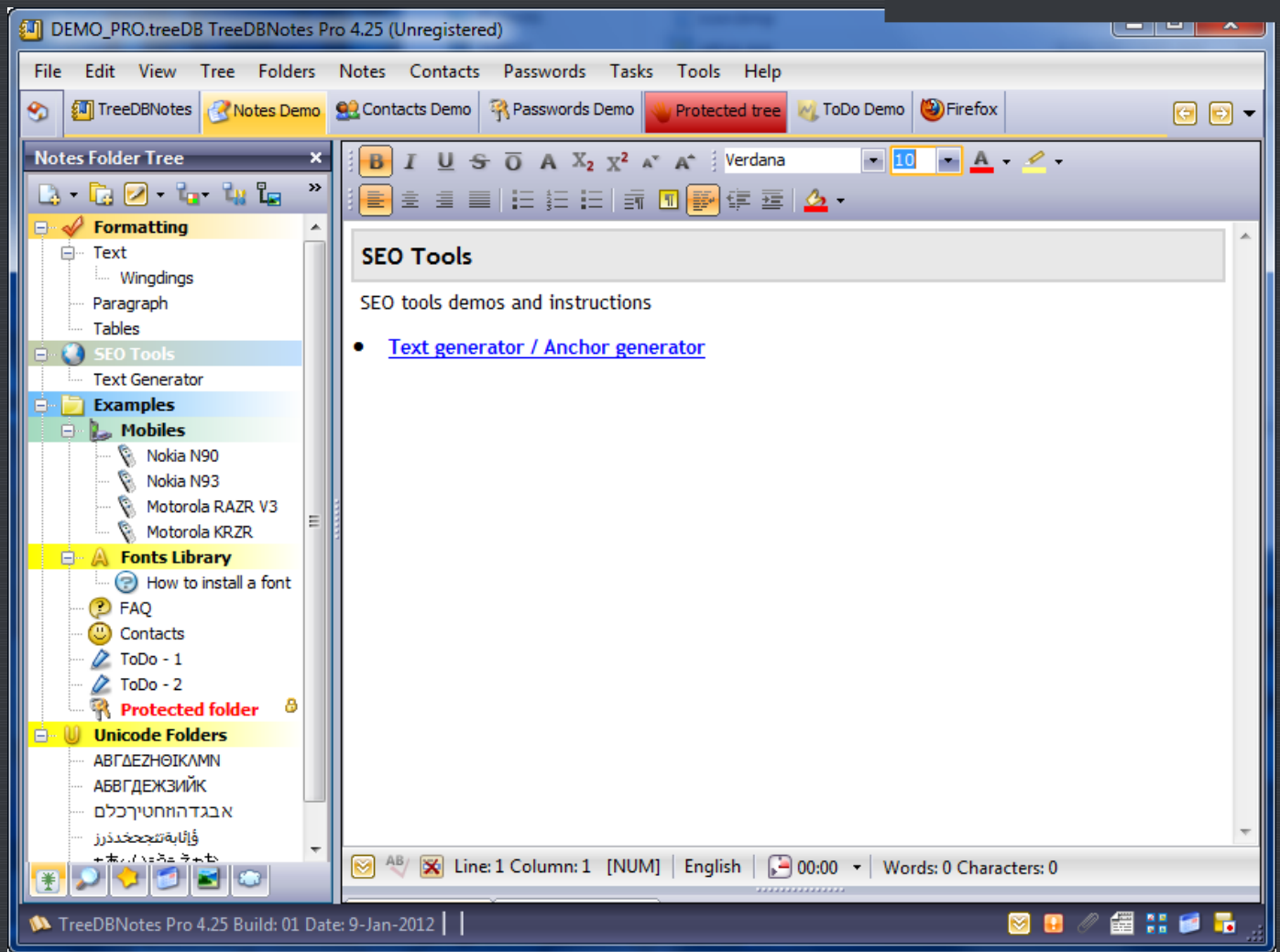
Studying the App

Go ahead and install the app. After completion, the following screen comes up:

Let's leave the "Run the app" checked and see what we're dealing with:

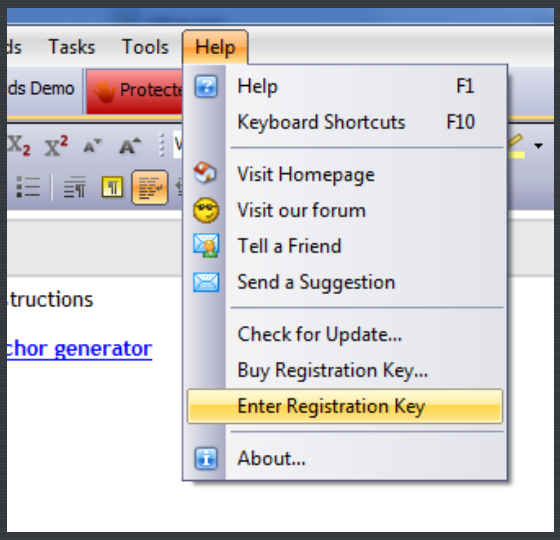


Well that's not very nice. Here we notice some strings that could be a potential help; "unregistered", "evaluation", "registered" etc. Click OK and we get to the main screen:

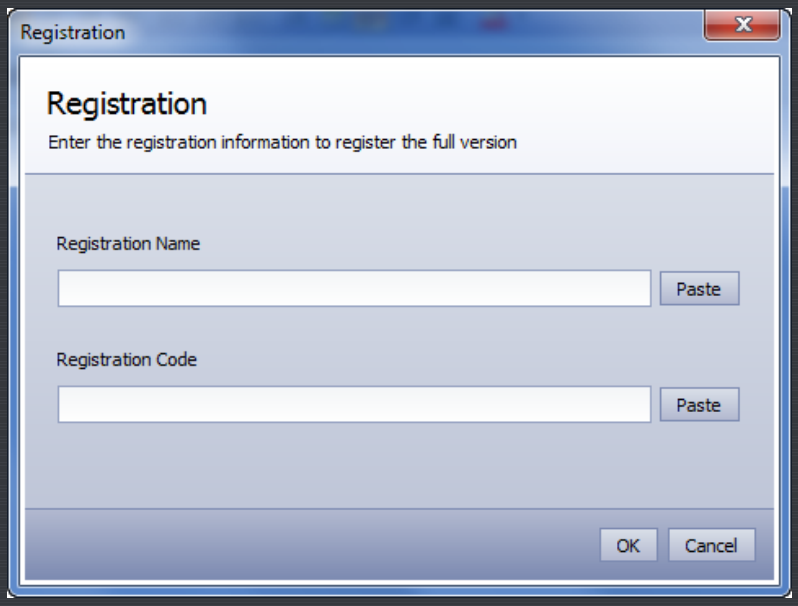


Notice it says "unregistered" at the top in the title bar. Usually, another place I look in an app is the about screen. A lot of times it will contain strings and or ideas for reversing. During this phase, we are looking for keywords, recognizable method calls, stuff like that. The more you do this the more clues will jump out at you:

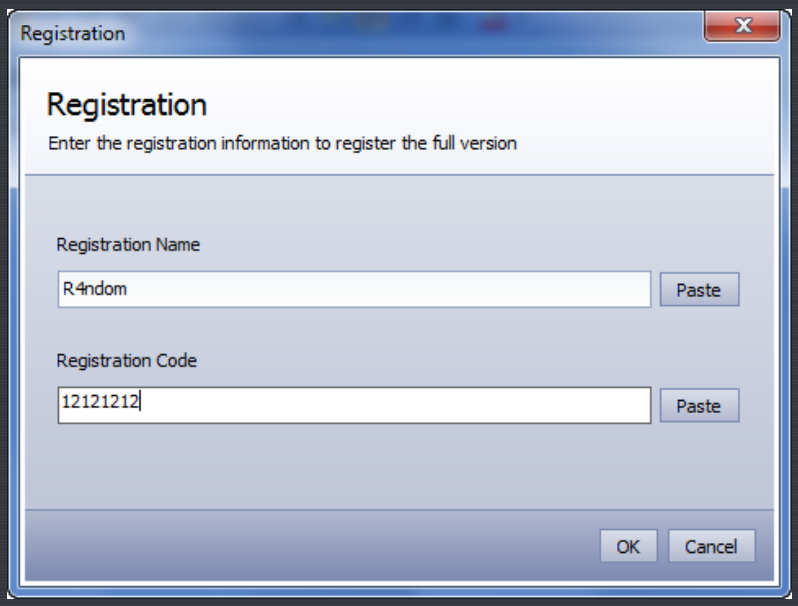
Here we see the word “unregistered” again. The next thing I usually look for is if there is a way to enter a registration code. This is a good starting point for penetration if the “search for strings” trick doesn’t work:



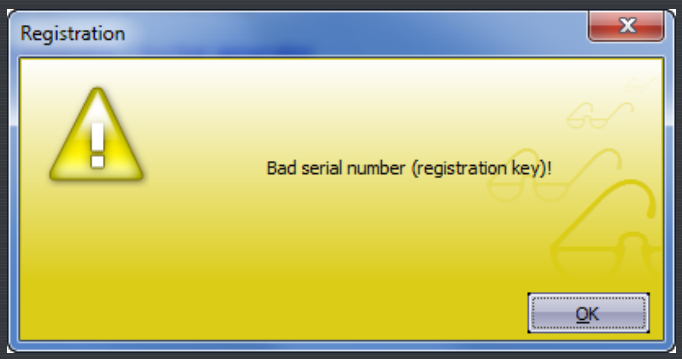
and here we see an option to enter a reg code:



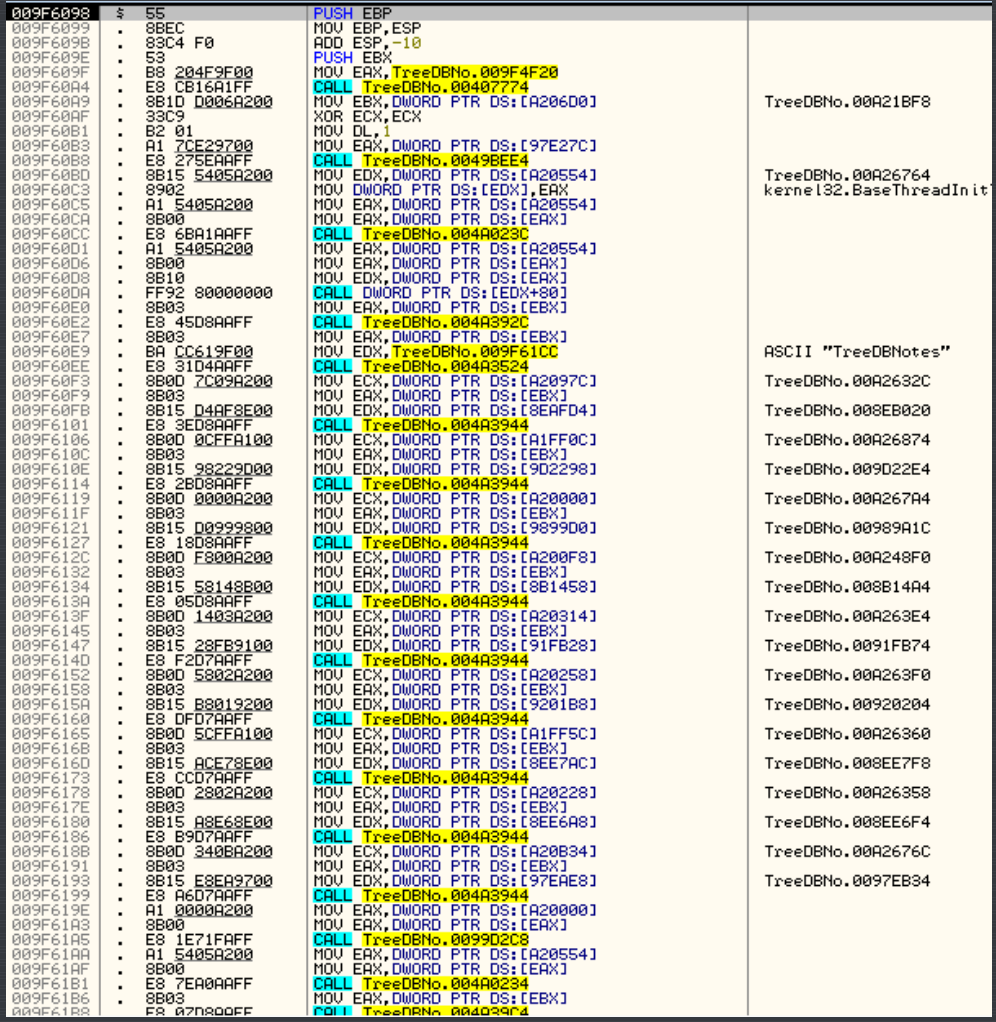
Let’s try one and see what happens:



Click OK:



Bummer. I never seem to get this part right . Alright, we have a pretty good idea as to what we have at our disposal, so let's load it up in Olly:

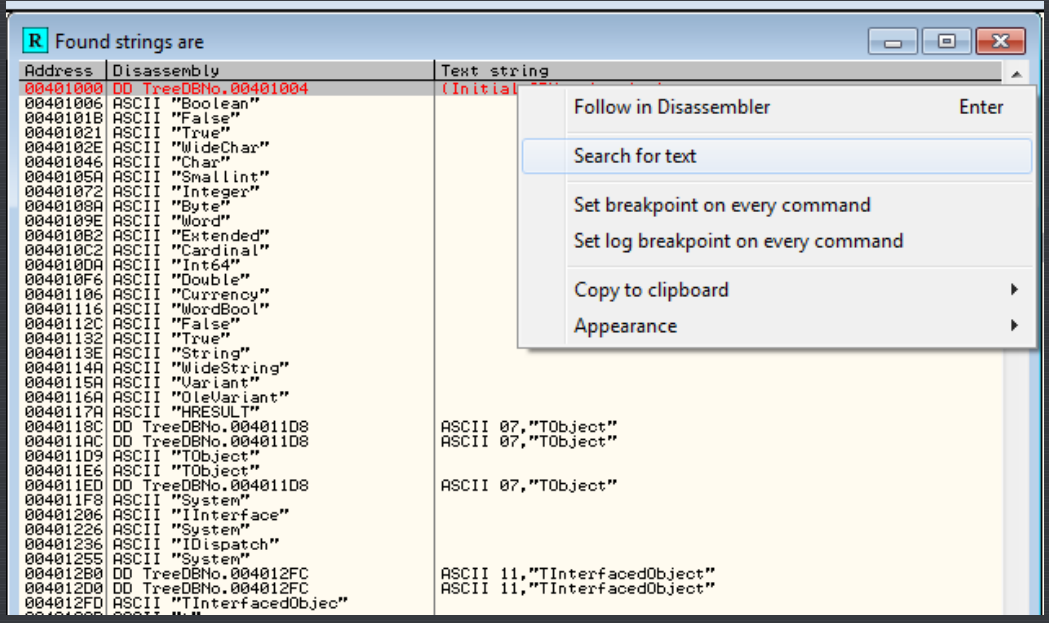


```
009F6098  $  55              PUSH EBP
009F6099  .  8BEC            MOV EBP,ESP
009F609B  .  83C4 F0         ADD ESP,-10
009F609E  .  53              PUSH EBX
009F609F  .  B8 204F9F00     MOV EAX,TreeDBNo.009F4F20
009F60A4  .  E8 CB16A1FF     CALL TreeDBNo.00407774
009F60A9  .  8B1D D006A200   MOV 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
```

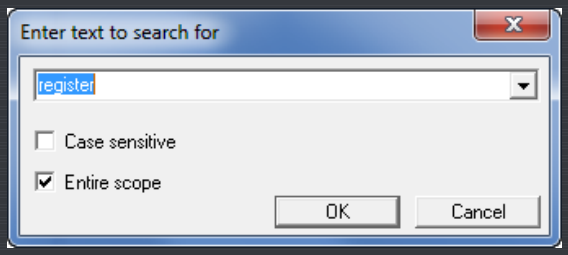
You may notice that this looks a little different than most of the apps we've looked at so far; there seem to be an awful lot of CALL instructions, without the typical Windows setup stuff (like RegisterClass...). This is a good sign that the program was written in Delphi. Delphi uses a TON of calls all over the place. We can tell for sure by running an ID program, but we'll get into that in a future tutorial. There are also specialized tools for dealing with Delphi programs, but fortunately we do not need to use them in this tutorial (we will get to them though 😀)

Finding the Patches

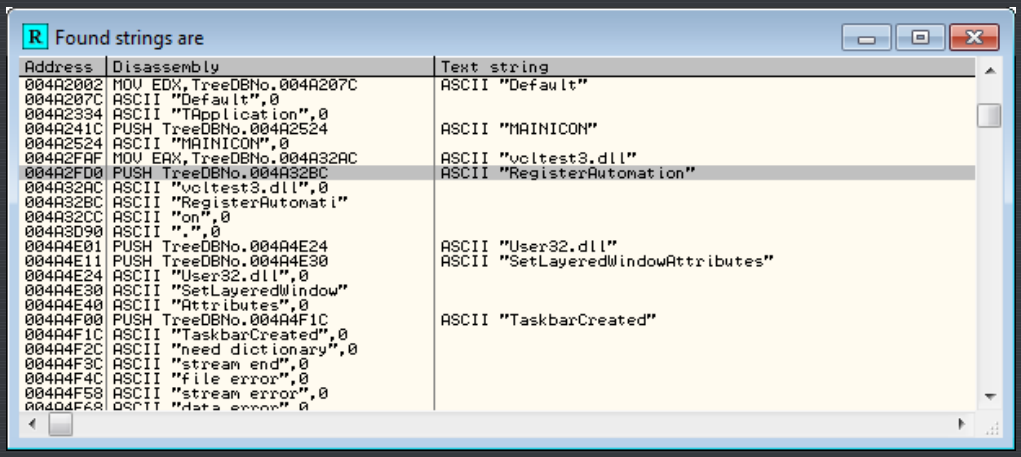
Let's try our string search. Right-click, choose "Search for" -> "All referenced text strings" and the search window will open. Scroll to the top and right click. Choose "Search for text":



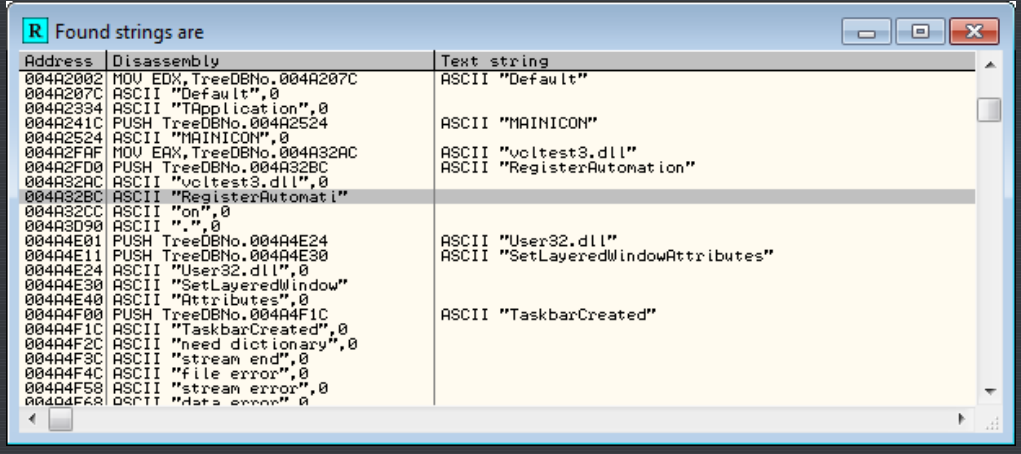
and the search for text window opens. Now, I noticed that the word “registration” and “registered” were used a lot earlier, so let’s search for them. Usually in this case, as my first search, I will search for “regist” as this covers both “Registration” and “registered”, and I’ve never gotten a false positive from this (I guess not a lot of programs use the word “registrar” in their programs 😀). Make sure “Case sensitive” is un-clicked and “Entire scope” IS clicked and hit OK:



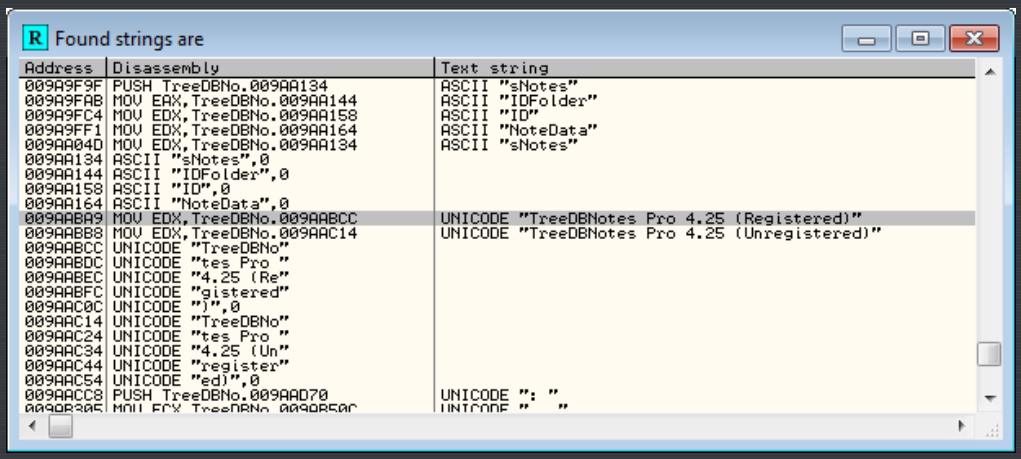
The first hit we get doesn’t seem to promising, so hit ctrl-L to go to the next occurrence:



Notice that this occurrence is just the actual data of the first hit we had. This is because the first hit was where the string “RegisterAutomation” was pushed on to the stack, and the second occurrence is the actual data in memory for the string “RegisterAutomation”. You can tell because there is no instruction for it in the second column, and instead it says ASCII. Most strings you come across will have two version of it, the one where the string is accesses, and one where the sting actually resides:



If you hey ctrl-L again, we will come to another not very promising looking string. Keep hitting ctrl-L until we come to the following:



Now that looks a lot better. It would appear that at some point in the programs starting up sequence, it checks if we art registered or not, and depending on the results, it fills the title bar of the window with either the registered or unregistered string. This is a good place to start. Double click on the “registered” version and we will jump to the code:

```
009AAB84  .  0F94C2            SETE DL
009AAB87  .  8B83 280F0000     MOV EAX,DWORD PTR DS:[EBX+F28]
009AAB8D  .  E8 9A3BAEFF       CALL TreeDBNo.0048E72C
009AAB92  .  5B                POP EBX                          kernel32.762BED6C
009AAB93  .  5D                POP EBP                          kernel32.762BED6C
009AAB94  L. C3                RETN
009AAB95     8D40 00           LEA EAX,DWORD PTR DS:[EAX]
009AAB98  r$ 55                PUSH EBP
009AAB99  .  8BEC              MOV EBP,ESP
009AAB9B  .  53                PUSH EBX
009AAB9C  .  8BDA              MOV EBX,EDX                      TreeDBNo.<ModuleEntryPoint>
009AAB9E  .  80B8 B8150000 00  CMP BYTE PTR DS:[EAX+15B8],0
009AABA5  .v 74 0F             JE SHORT TreeDBNo.009AABB6
009AABA7  .  8BC3              MOV EAX,EBX
009AABA9  .  BA CCAB9A00       MOV EDX,TreeDBNo.009AABCC        UNICODE "TreeDBNotes Pro 4.25 (Registered)"
009AABAE  .  E8 55A9A5FF       CALL TreeDBNo.00405508
009AABB3  .  5B                POP EBX                          kernel32.762BED6C
009AABB4  .  5D                POP EBP                          kernel32.762BED6C
009AABB5  .  C3                RETN
009AABB6  >  8BC3              MOV EAX,EBX
009AABB8  .  BA 14AC9A00       MOV EDX,TreeDBNo.009AAC14        UNICODE "TreeDBNotes Pro 4.25 (Unregistered)"
009AABBD  .  E8 46A9A5FF       CALL TreeDBNo.00405508
009AABC2  .  5B                POP EBX                          kernel32.762BED6C
009AABC3  .  5D                POP EBP                          kernel32.762BED6C
009AABC4  L. C3                RETN
009AABC5     00                DB 00
009AABC6     00                DB 00
009AABC7     00                DB 00
009AABC8     42                DB 42                            CHAR 'B'
009AABC9     00                DB 00
009AABCA     00                DB 00
009AABCB     00                DB 00
009AABCC  .  5400 7200 6500 6E UNICODE "TreeDBNo"
009AABDC  .  7400 6500 7300 20 UNICODE "tes Pro "
009AABEC  .  3400 2E00 3200 35 UNICODE "4.25 (Re"
009AABFC  .  6700 6900 7300 74 UNICODE "gistered"
009AAC0C  .  2900 0000         UNICODE ")",0
009AAC10     46                DB 46                            CHAR 'F'
009AAC11     00                DB 00
009AAC12     00                DB 00
009AAC13     00                DB 00
009AAC14  .  5400 7200 6500 6E UNICODE "TreeDBNo"
009AAC24  .  7400 6500 7300 20 UNICODE "tes Pro "
009AAC34  .  3400 2E00 3200 35 UNICODE "4.25 (Un"
009AAC44  .  7200 6500 6700 69 UNICODE "register"
009AAC54  .  6500 6400 2900 00 UNICODE "ed)",0
009AAC5C  r$ 55                PUSH EBP
009AAC5D  .  8BEC              MOV EBP,ESP
009AAC5F  .  33C9              XOR ECX,ECX
```

First notice that we can see where the string is used at address 9AABA9, and we can also see where the string is stored in memory at address 9AABBC. Secondly, notice that both strings are in the same method and a conditional jump is above them. Clicking on that conditional jump at address 9AABA5:

```
009AAB9B   .  53            PUSH EBX
009AAB9C   .  8BDA          MOV EBX,EDX                          TreeDBNo.<ModuleEntryPoint>
009AAB9E   .  80B8 B8150000 00 CMP BYTE PTR DS:[EAX+15B8],0
009AABA5   .v 74 0F         JE SHORT TreeDBNo.009AABB6
009AABA7   .  8BC3          MOV EAX,EBX
009AABA9   .  BA CCAB9A00   MOV EDX,TreeDBNo.009AABCC            UNICODE "TreeDBNotes Pro 4.25 (Registered)"
009AABAE   .  E8 55A9A5FF   CALL TreeDBNo.00405508
009AABB3   .  5B            POP EBX                              kernel32.7747ED6C
009AABB4   .  5D            POP EBP                              kernel32.7747ED6C
009AABB5   .  C3            RETN
009AABB6   >  8BC3          MOV EAX,EBX
009AABB8   .  BA 14AC9A00   MOV EDX,TreeDBNo.009AAC14            UNICODE "TreeDBNotes Pro 4.25 (Unregistered)"
009AABBD   .  E8 46A9A5FF   CALL TreeDBNo.00405508
009AABC2   .  5B            POP EBX                              kernel32.7747ED6C
009AABC3   .  5D            POP EBP                              kernel32.7747ED6C
009AABC4   L. C3            RETN
009AABC5      00            DB 00
009AABC6      00            DB 00
009AABC7      00            DB 00
009AABC8      42            DB 42                                CHAR 'B'
```

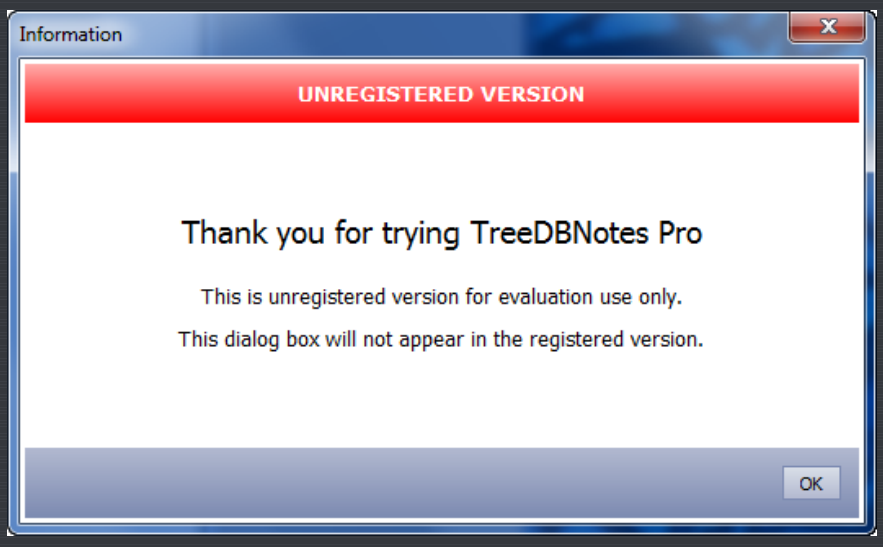
we can see that if the result is equal, we will jump to the “unregistered” version of the string. We obviously don’t want this to happen. Let’s place a BP on this JE instruction and start the app:

```
009AAB94   L. C3            RETN
009AAB95      8D40 00       LEA EAX,DWORD PTR DS:[EAX]
009AAB98   $  55            PUSH EBP
009AAB99   .  8BEC          MOV EBP,ESP
009AAB9B   .  53            PUSH EBX
009AAB9C   .  8BDA          MOV EBX,EDX                          TreeDBNo.<ModuleEntryPoint>
009AAB9E   .  80B8 B8150000 00 CMP BYTE PTR DS:[EAX+15B8],0
009AABA5   .v 74 0F         JE SHORT TreeDBNo.009AABB6
009AABA7   .  8BC3          MOV EAX,EBX
009AABA9   .  BA CCAB9A00   MOV EDX,TreeDBNo.009AABCC            UNICODE "TreeDBNotes Pro 4.25 (Registered)"
009AABAE   .  E8 55A9A5FF   CALL TreeDBNo.00405508
009AABB3   .  5B            POP EBX                              kernel32.7747ED6C
009AABB4   .  5D            POP EBP                              kernel32.7747ED6C
009AABB5   .  C3            RETN
009AABB6   >  8BC3          MOV EAX,EBX
009AABB8   .  BA 14AC9A00   MOV EDX,TreeDBNo.009AAC14            UNICODE "TreeDBNotes Pro 4.25 (Unregistered)"
009AABBD   .  E8 46A9A5FF   CALL TreeDBNo.00405508
009AABC2   .  5B            POP EBX                              kernel32.7747ED6C
009AABC3   .  5D            POP EBP                              kernel32.7747ED6C
009AABC4   L. C3            RETN
009AABC5      00            DB 00
009AABC6      00            DB 00
009AABC7      00            DB 00
```

Olly will break at this line and you will notice that we are going to jump to the bad boy. Let’s change that:

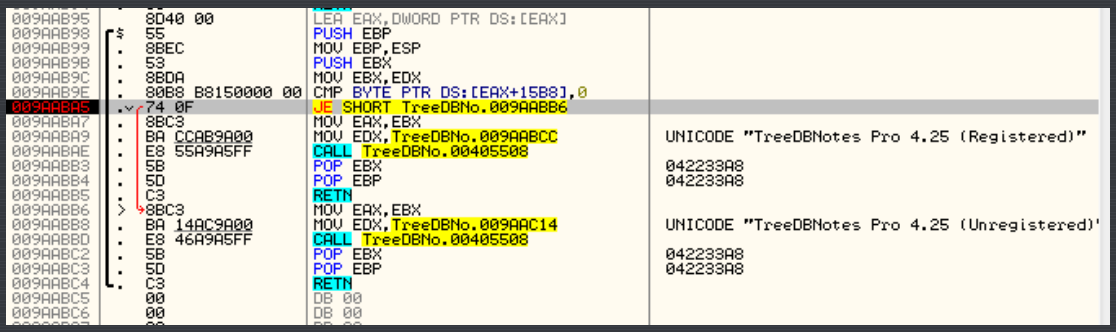
```
C 0  ES 0023 32bi
P 1  CS 001B 32bi
A 0  SS 0023 32bi
Z 0  DS 0023 32bi
S 0  FS 003B 32bi
T 0  GS 0000 NULL
D 0
O 0  LastErr ERRO
```

and run the app. Olly will then break at this same line again, wanting to jump to the bad boy. Let’s change it again by zeroing out the zero register and hitting run. This will happen one more, and clearing out the zero flag, we finally get some feedback:



So that didn’t work. So patching that one check does not make us unregistered, although if you click OK and zero out the flag one more time, you will notice that it does take off the “unregistered” title of the main window:

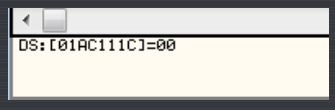
So at least we know we're on the right track. What we are going to have to do is step this up to the next 'level' and investigate a little further. Re-start the app so that we break at our breakpoint and let's investigate a little more:



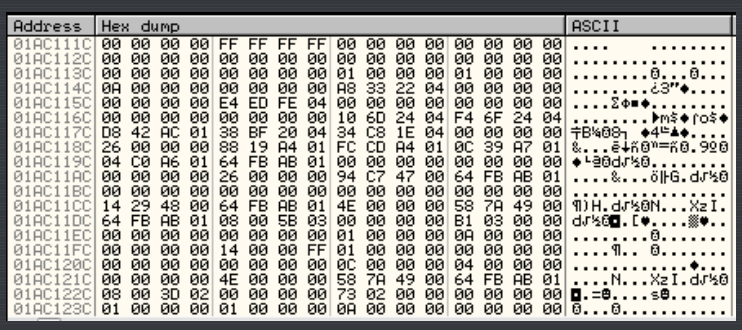
There is no call before the compare, but before the JE instruction there is a compare at address 9AAQB9E:

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

So, based on the outcome of this compare, we are either registered or we're not. EAX+15B8 is just a memory address, in this case a global variable as it starts with DS:. What we hope is that this is the only check that the app is registered or not. If it is not, we will need to go find out where else the app checks for registration status. Clicking on the compare instruction shows us what EAX+15B8 is:



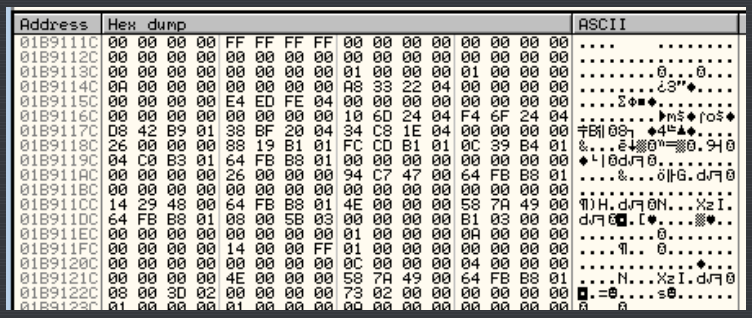
So right click on this address and choose "Follow in dump":



****Your address will almost certainly be different than mine. That is OK. Just follow along and replace your address with mine and it will run fine ****

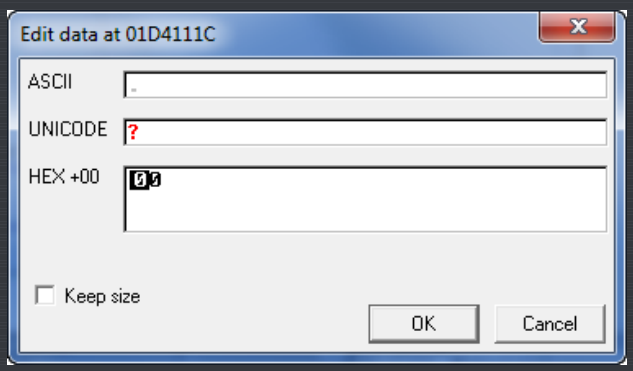
Here we can see the address that is checked for being registered or not; it is the first 00 at address 1AC111C *(on my computer at least)*. That means that if the contents of this memory location were to be anything other than zero, this routine would assume we were registered. This also means that there are probably other routines in the app that check this memory location which is why the main screen shows "Registered" while another part of the app knows we're not. Since we only bypassed this routine's natural flow after checking the memory contents, any other routine that checks it was not bypassed.

First things first, let's set this memory address to non-zero so we know that at least this routine will always work the way we want. Set a breakpoint on the compare line (9AABA5) and delete our other BP. Re-start the app and Olly will break. Right click on the compare line and choose "Follow in dump" -> "Memory location" as Olly reset our dump window when we restarted. One thing you may notice is that the memory address that the compare instruction checks is different this time:

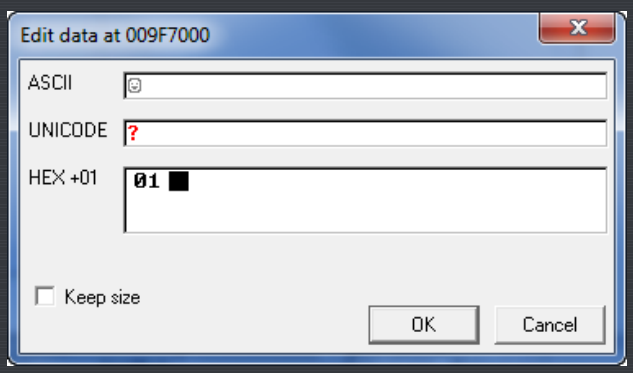


My first one was 1AC111C and it is now 1B9111C. Yours will be different than mine, but just notice that the second time through, the memory address that stores the registered/not-registered flag is different.

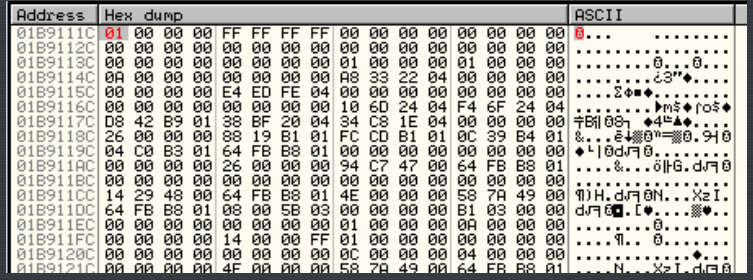
Click on the "00" in the dump (at 1B9111C in my dump), right click and choose "Binary" -> "Edit":



Let's enter 01:

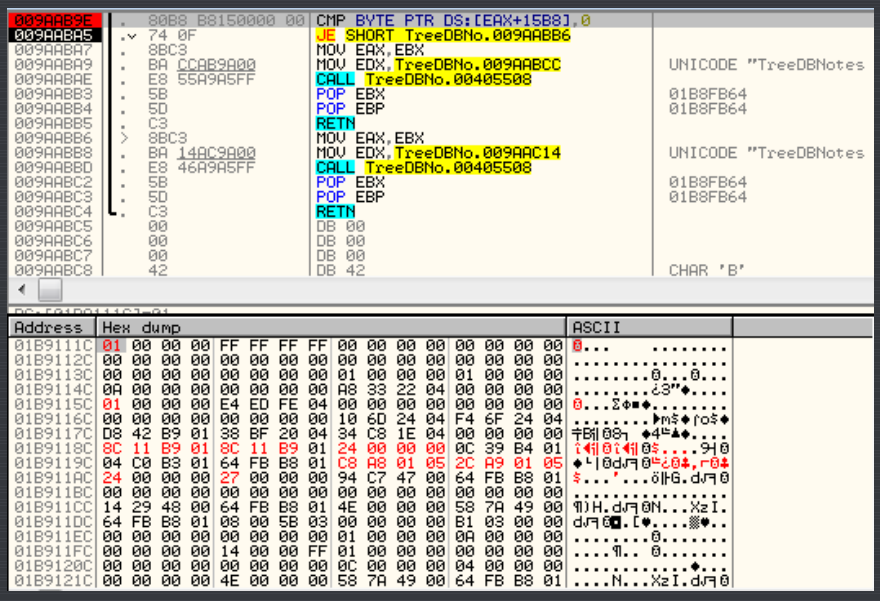


and notice it has been updated in the dump:

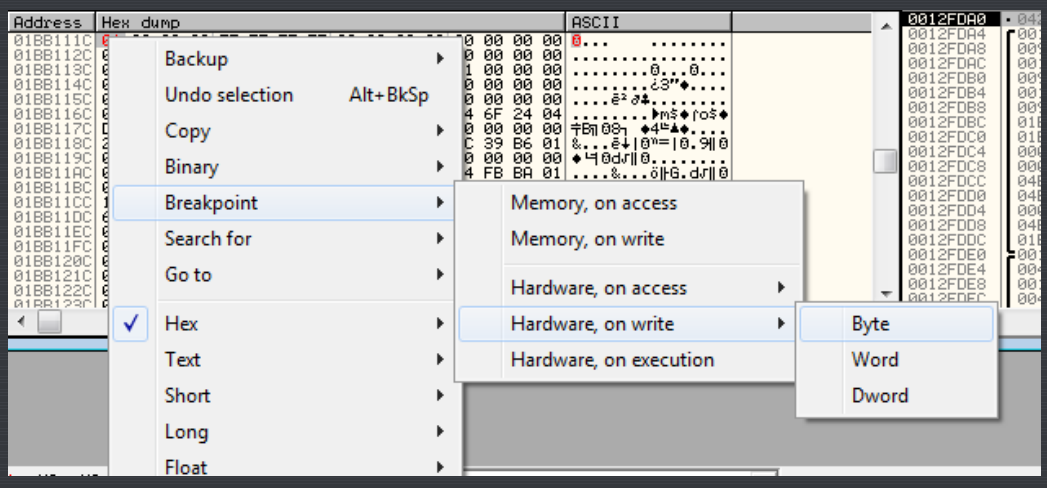


No go ahead and run it till we break again. You will notice that the memory contents have changed back to zeroes and that we are now going to jump to the bad boy again. This means that somewhere in the app, a secondary check was done that reset our registered flag back to zero. What we need to do is find where this is being set and make sure it doesn't happen. To do this, we want to set a hardware breakpoint on this memory location to tell Olly to stop whenever the app writes to this location. We want to chose 'write' because somewhere a zero is being written to this address.

Re-start the app and run it until we break. Right click the compare and choose "Follow in dump" again as Olly has reset the dump window. Binary edit the first memory location to 01. Notice it's now at a different memory address:

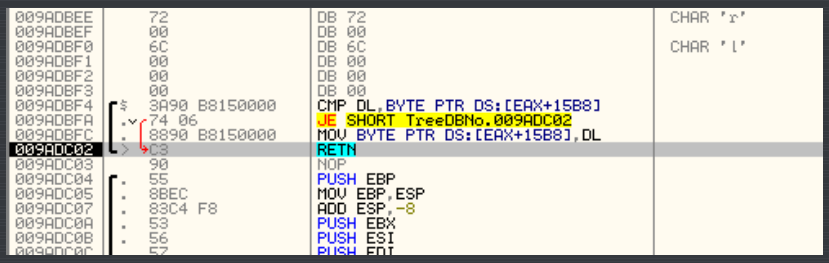


Then right-click on the first value in the dump that we edited and choose"Breakpoint" -> Hardware, on write" -> "byte":

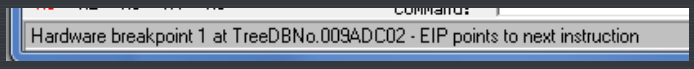


When reverse engineering an app, I generally stay with hardware breakpoints as they are harder for the app to detect. I selected
"byte" as it's only the one byte we want to track.

Now run the app. Olly will break at our normal breakpoint again, and you can see that the 01 value we entered is still there, so so far so good. Run it again and Olly will break in a new section:

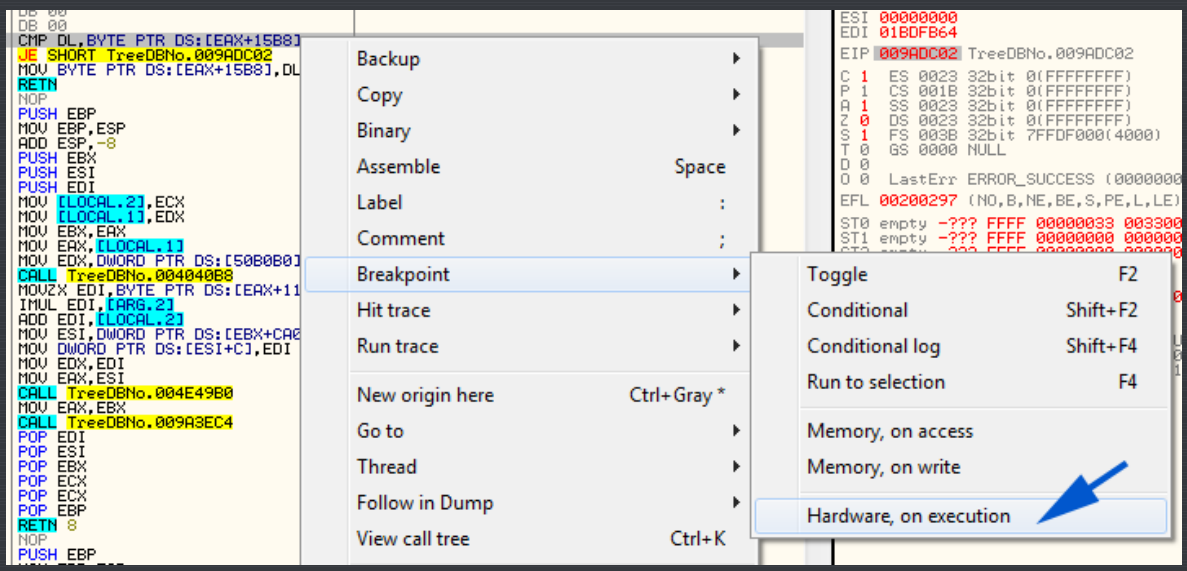


If you look in the bottom left corner of the OllyDBG window, you will see that we broke on our hardware breakpoint:



Patching the App

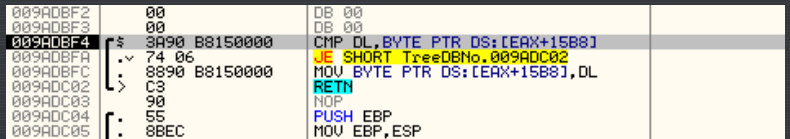
Now, let's study this code. the first instruction compared DL with the memory contents of our edited address. If they are equal, we jump to 9ADC02, which simply returns. If they are not the same, we store the contents of DL into our memory location. We already know that DL equals zero because we saw the memory location change from our 01 back to 00. So this is basically another registration check, and if it fails if puts a zero in the registered/not-registered flag. If it doesn't fail, it leaves it alone. Now let's remove our hardware breakpoint "Debug" -> "Hardware breakpoints" and delete it, and let's place another hardware breakpoint at address 9ADBF4 so that we can break before this routine has run:



Now you may wonder why I didn't just put a regular breakpoint on this. It is because I tried that first! But Olly would not break on it. There are several reasons that could cause this; this code changes polymorphically, so our BP is lost, there is a check in the app for a software breakpoint and the app removes it, the breakpoint is in a section that Olly will not track automatically… It happens. If it does, we need to set a hardware breakpoint on it instead. There are no guarantees that a HW breakpoint will work, as the app may specifically check for these as well, but it is a more robust way of placing a breakpoint, so it usually works.

**** We will be going over anti-debug tricks more in future tutorials****

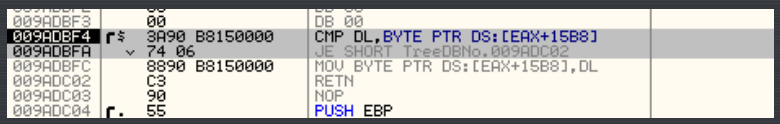
Now restart the app and we will again break at our new hardware breakpoint:



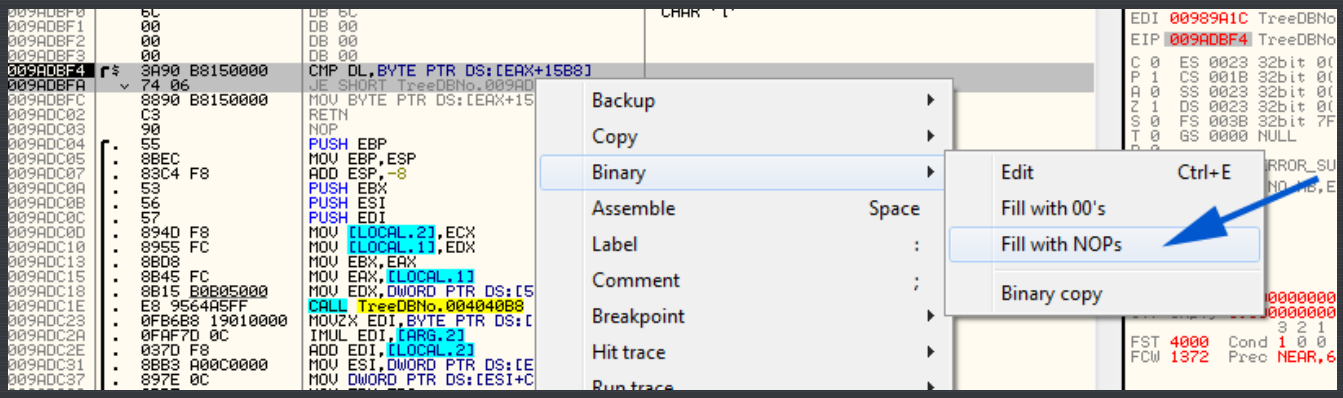
OK, now let's think for a minute. This routine is called before our original break. This routine checks if we are registered or not and puts a zero in the memory address pointed to by [EAX+15B8] if it is not, and a 01 (or any non-zero) if it is. Then our old routine is called, the one that either prints "Registered" or "Unregistered" on the title of the window based on if this memory location contains a 0 or 1. So if we make sure a 1 is put into that memory location every time this routine is run, then any other routines will check that memory location and see that it is a 1 and think that we're registered.

What would happen if we just change this routine to always put a 01 into the proper memory location? Let' try it.

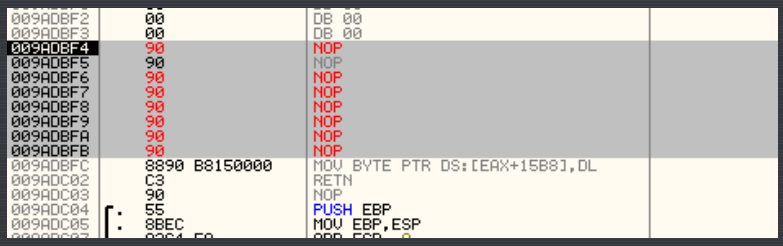
Now the next question is what's the easiest way to do that. Well, we have the memory location already being populated with something (DL) at address 9ADBFC, so we could just change the DL top a one. The problem with this is that changing the DL to a one will add a byte to the length of this instruction, and this will overwrite our RETN statement.What about if we replace the compare and jump instructions and instead just load 01 into DL. That way, on the last line, DL will be moved into our memory location! So here's what we do- highlight the two compare and jump instructions:



Then right-click and choose "Binary" -> "Fill with NOPs":

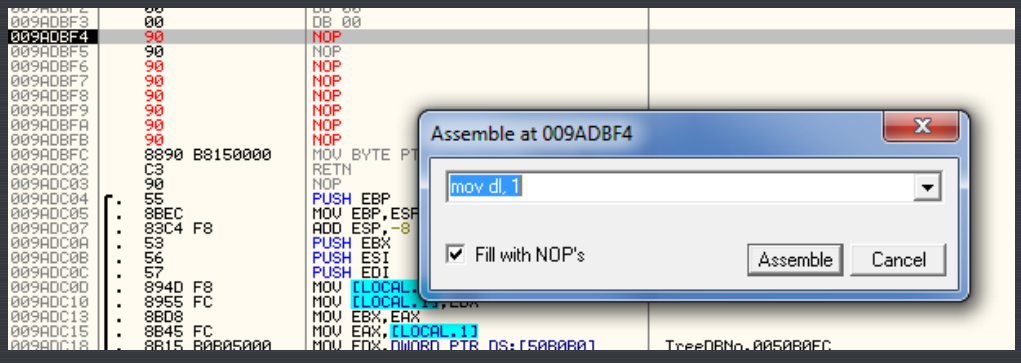


Which gives us this:



This step isn't required, but it makes it a lot easier to see what you're doing.

Now click on the first NOP at address 9ADBF4 and hit the space bar. This will bring up the assemble window. Then enter MOV DL, 1:



Click Assemble then Cancel. That gives us this:

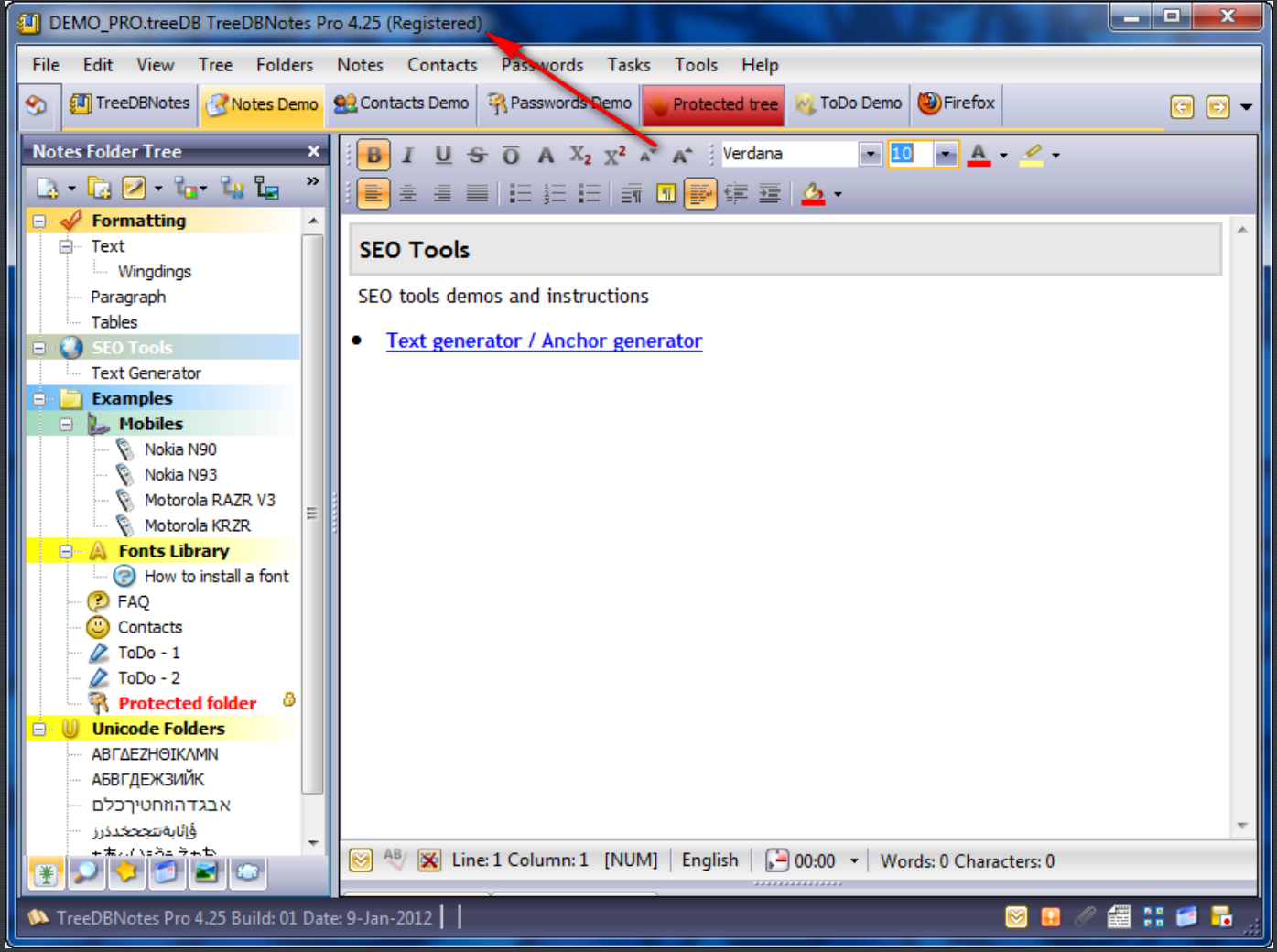
```
009ADBF2     00              DB 00
009ADBF3     00              DB 00
009ADBF4     B2 01           MOV DL,1
009ADBF6     90              NOP
009ADBF7     90              NOP
009ADBF8     90              NOP
009ADBF9     90              NOP
009ADBFA     90              NOP
009ADBFB     90              NOP
009ADBFC     8890 B8150000   MOV BYTE PTR DS:[EAX+15B8],DL
009ADC02     C3              RETN
009ADC03     90              NOP
009ADC04  /: 55              PUSH EBP
009ADC05  |: 8BEC            MOV EBP,ESP
```

Now, whenever this routine is called, a one will be put into the memory flag instead of a zero. Since we are still paused on the first line of this routine, you can single step to see DL being loaded with 1, and then the 1 being put into the memory address (you may need to go to the proper address in your dump as Olly has probably reset it again). Now run the app and Olly will break at our original breakpoint:

```
009AAB95     8D40 00         LEA EAX,DWORD PTR DS:[EAX]
009AAB98  /$ 55              PUSH EBP
009AAB99  |. 8BEC            MOV EBP,ESP
009AAB9B  |. 53              PUSH EBX
009AAB9C  |. 8BDA            MOV EBX,EDX
009AAB9E  |. 80B8 B8150000 00 CMP BYTE PTR DS:[EAX+15B8],0
009AABA5  |. 74 0F           JE SHORT TreeDBNo.009AABB6
009AABA7  |. 8BC3            MOV EAX,EBX
009AABA9  |. BA CCAB9A00     MOV EDX,TreeDBNo.009AABCC          UNICODE "TreeDBNotes Pro 4.25 (Registered)"
009AABAE  |. E8 55A9A5FF     CALL TreeDBNo.00405508
009AABB3  |. 5B              POP EBX                            042233A8
009AABB4  |. 5D              POP EBP                            042233A8
009AABB5  |. C3              RETN
009AABB6  |> 8BC3            MOV EAX,EBX
009AABB8  |. BA 14AC9A00     MOV EDX,TreeDBNo.009AAC14          UNICODE "TreeDBNotes Pro 4.25 (Unregistered)"
009AABBD  |. E8 46A9A5FF     CALL TreeDBNo.00405508
009AABC2  |. 5B              POP EBX                            042233A8
009AABC3  |. 5D              POP EBP                            042233A8
009AABC4  |. C3              RETN
009AABC5     00              DB 00
```

and we cann see that we are going to fall through to the correct string. Go ahead and keep running and we will break in our modified registration check routine, and it will put a 01 into our address again as we planned. This will go back and forth a couple times until finally:

We are now registered!!!! Go ahead and run the program (open a demo file) and Olly will break several more times in our registration routine, but each time it will go the right way. Soon you will get to the main screen:



and you will see that we are still registered. Clicking on the about screen shows:

Congratulations. You have patched your first crack 😃

Don't forget to save it back to disk. Open the Hardware breakpoints window ("Debug" -> "Hardware breakpoints") and click the Follow button on our BP. That will take us to our patch. Highlight everything we changed, right-click and select "Copy to executable". The right-click in the new window and select "Save to disk". Save it as the original file name. Now quit Olly and run the app and experience it is all it registered glory!!!!!

-Till next time

R4ndom