R4ndom's Tutorial #17: Working With Delphi Binaries

by R4ndom on Aug.08, 2012, under Beginner, Reverse Engineering, Tutorials

In this tutorial we will discuss working with binaries written in Delphi. Delphi binaries are quite different then binaries written in other languages. You can generally tell a Delphi program by the numerous calls (far more than a typical program) as well as some other techniques we will be discussing.

Included in the tutorial download are the two crackme's, the Delphi Decompiler (DeDe), and ExeInfoPE, available on the tutorials page.

You will also need Resource Hacker (and OllyDBG) available on the tools page.

Delphi

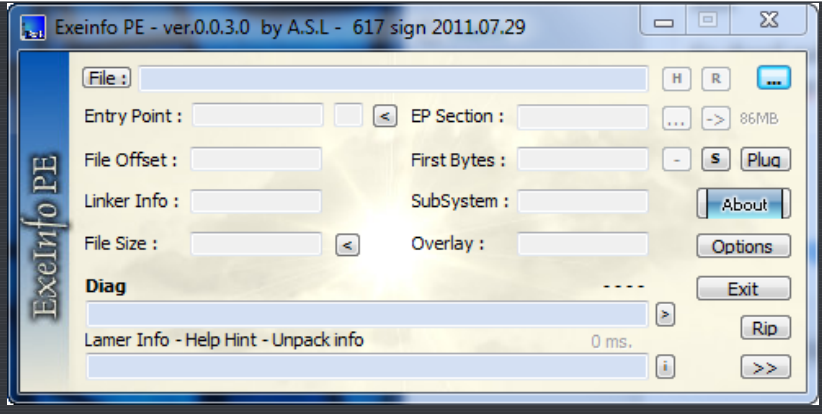
Most program construction in Delphi is by using forms, which are basically just windows or dialog boxes. You design them using a graphic utility to 'paint' the form, meaning to add buttons, edit boxes etc wherever you would like them. Really, the only thing you need to do after that is let the Delphi compiler know which actions you would like to handle and what you would like to do should that action happen. For example, when a button is clicked, you may want to open a file dialog box. In this case you would let the Delphi compiler know that the button should be associated with your code that you provide, and this code simply opens the file dialog.

These forms, along with everything associated with them (strings, sizes, colors) are stored in resources, in theory like a normal C++ application, but implemented far differently. One interesting thing is that Delphi associates all of these resources by name, meaning that the name you called the specific resource is the name that will be hard-coded into the executable, and the name that the executable will use to 'look up' resources. This is both good and bad. Good in that you can easily find these names that are associated with resources. Bad in that they are all stored in one area and are not logically int3erwoven into the code, so finding the code that goes along with a resource (say, a button click) is a lot tougher.

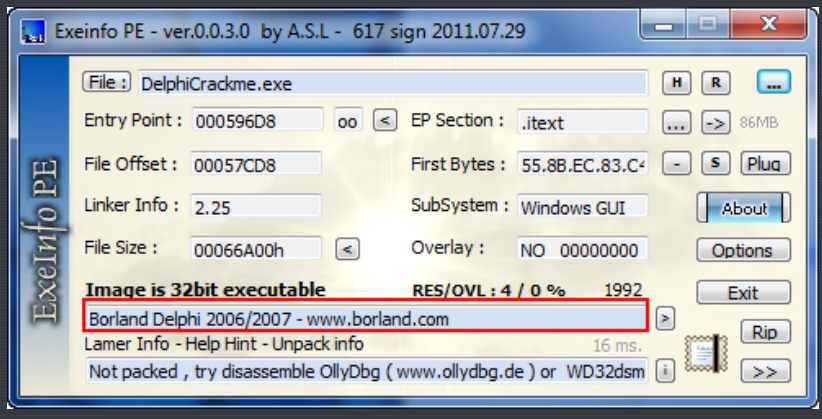
Much of the programming is done for you, other than, say, C++. There is a tremendous amount done 'behind the scenes' in a Delphi program. This is one reason the code looks different than what you may expect.

The first target

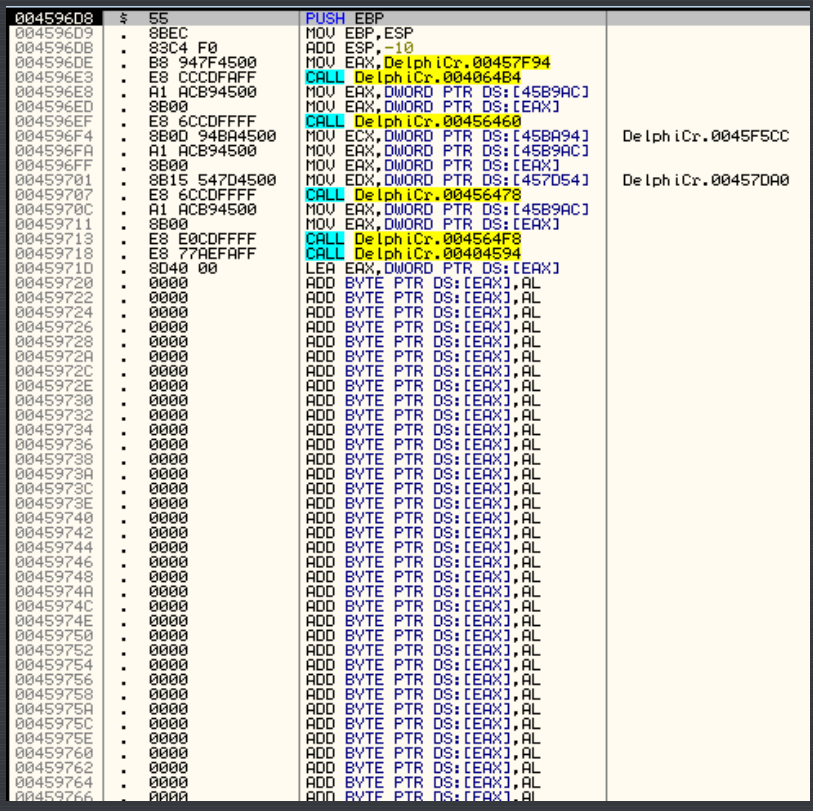
One of the first questions you may have is, "How do I know that I'm dealing with a Delphi program?" After a lot more experience, they stand out like a sore thumb, but until then, we are going to use a tool that will help. Go ahead and run ExeInfoPE. This program is generally used to discover what packer has been used on a packed binary (and we will use it a great deal when we get to packing). But, lucky for us, if the program is not packed it also tells us what language the program was written in. When you first run ExeInfoPE you will see the startup screen:



Go ahead and load our first target, DelphiCrackme.exe into ExeInfoPE and you will see the various fields populated (you can just drag the crackme icon and drop it into the ExeInfoPE window):



Here we can see that ExeInfoPE has found that this binary was compiled in Delphi. Under that you cannot also see that it is not packed. Also, just as a quick aside, as soon as you load the binary into Olly you will see that you're dealing with something different:

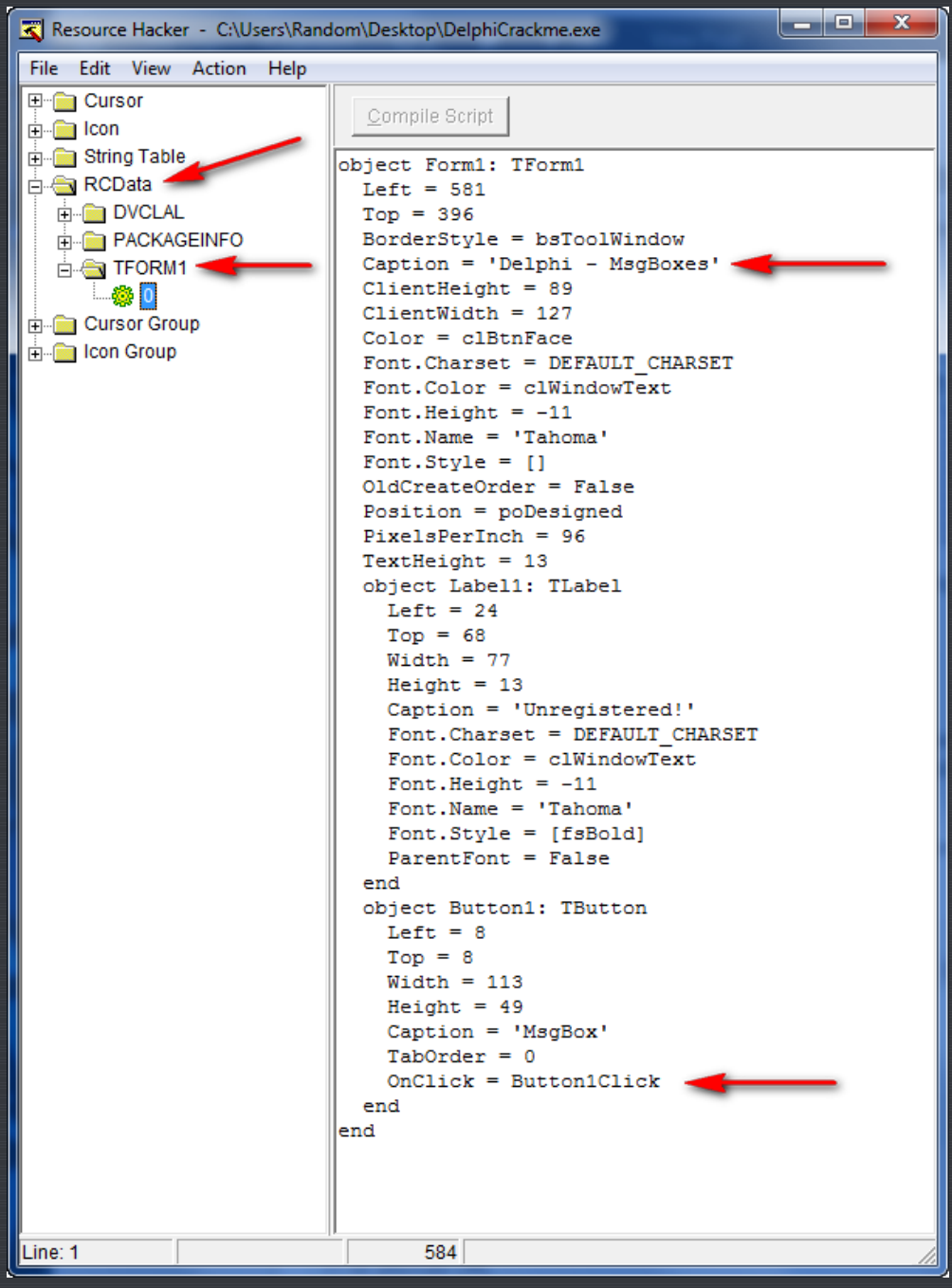


You can tell it doesn't look like the typical apps we've been dealing with.

Looking at Delphi resources

One of the most important differences, at least to a reverse engineer, of a Delphi program is the resources. Loading up the DelphiCrackme into Resource Hacker, you should notice a new folder that is not typically not there called RCDATA. Opening this folder shows us the resource sections associated with this binary:

**** If you do not have Resource Hacker you can download it from the tools page ****



Generally, the most important sub-folder (resource sections) are the TFORM sections. These are the windows/dialog boxes in the Delphi program. In this particular crackme, we can see that there is one form, TFORM1. Clicking on the little flower inside TFORM1 opens the main data area for this section in Resource Hacker (as you can see above). This data tells you everything about the form; the size, the colors, the placement on the screen, the title (caption), any fields or buttons it has in it- everything.

Usually, the first place I look is the 'Caption' as this tells you what will be in the title bar in the window. In this case it's "Delphi – MsgBoxes". The importance of this field is, in an app that has many forms called TFORM1, TFORM2, TFORM3… it is difficult to know which form is associated with which window. Looking

at the captions can help distinguish this. For example, the caption may say “Register” letting us know it’s the registration screen, or “About” for the about screen.

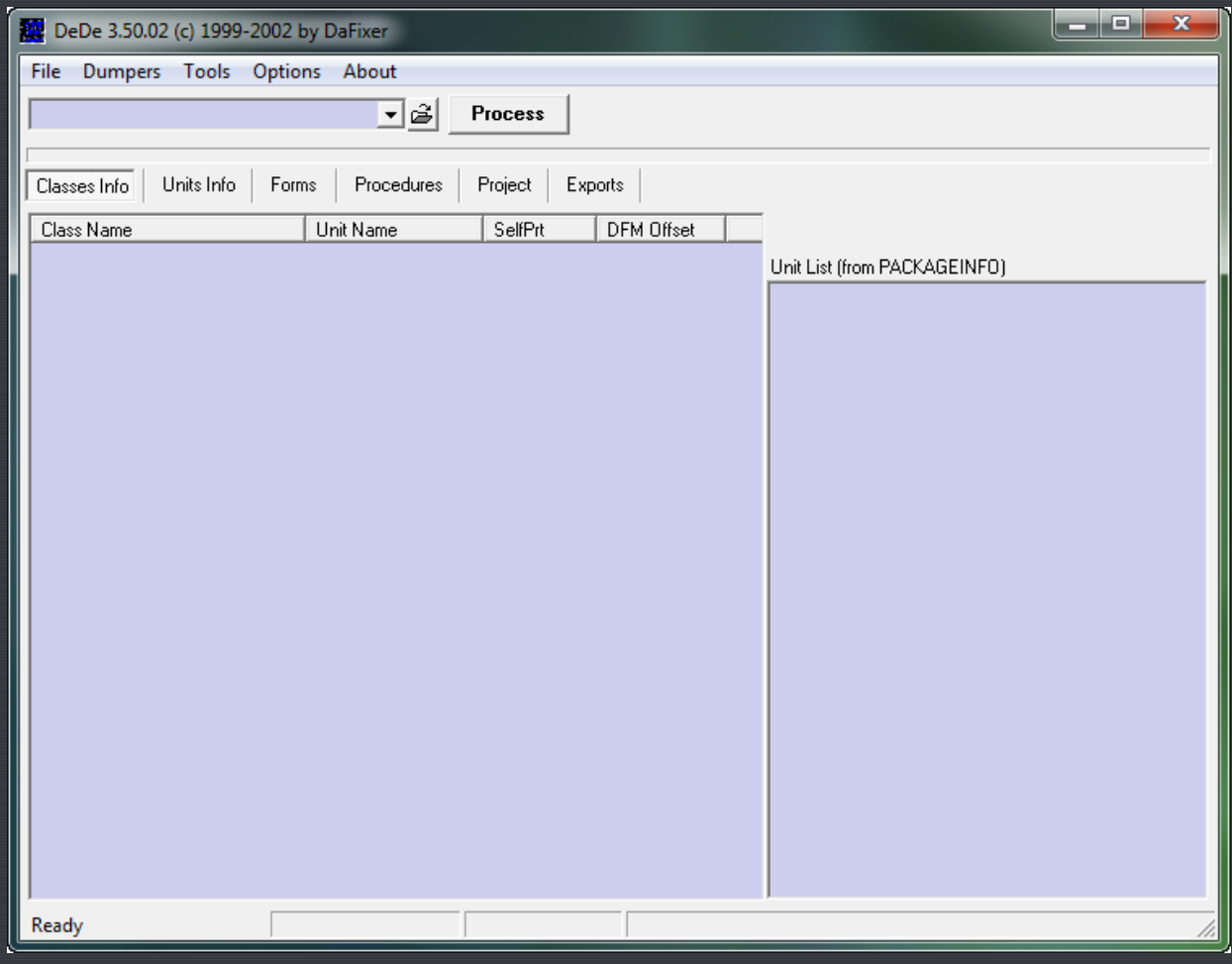
Finally, of importance to us is the button objects at the bottom. The reason these are important is because we generally want to trap the program after a button has been hit, say, after hitting OK on the registration screen after entering our username and serial. The important thing about the buttons is the button name for the method when the button is clicked. In this case it’s “Button1Click”. As I said earlier, Delphi programs connect everything with ASCII names, so when the app wishes to run the code associated with clicking this button, it will look up the name “Button1Clicked” to find the method.

From viewing this file in a resource viewer, we have gathered that there is one form (window) with one button. The caption of he window is “Delphi – MsgBoxes” and the callback function that handles the click of the button is called “Button1Click”.

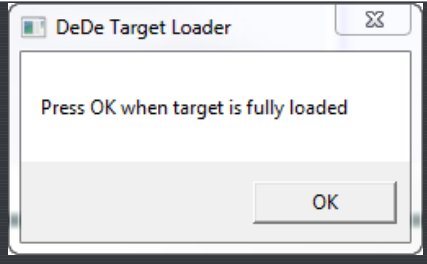
Now let’s move on to using one of the most important tools in dealing with Delphi programs…

Using DeDe

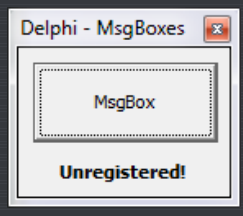
Delphi Decompiler loads a Delphi program and breaks it down for you, showing all the forms data we’ve seen, but also where all the methods are called, the address of all the methods, and the method names. It also shows a complete decompilation of the binary if we wish, along with capabilities to modify it. Let’s go ahead and run DeDe. After a really kick-ass splash screen (if you’re 9) we see the main window:



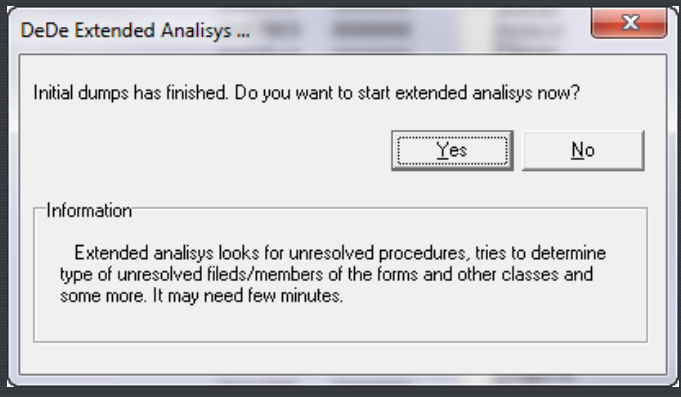
First, we need to load our program in so DeDe can decompile it. You can either select the open folder icon or just drag our DelphiCrackme into the DeDe window and choose ‘Yes’ to allow DeDe to begin processing the binary. At this point, DeDe pops up a message box asking if the target has loaded:



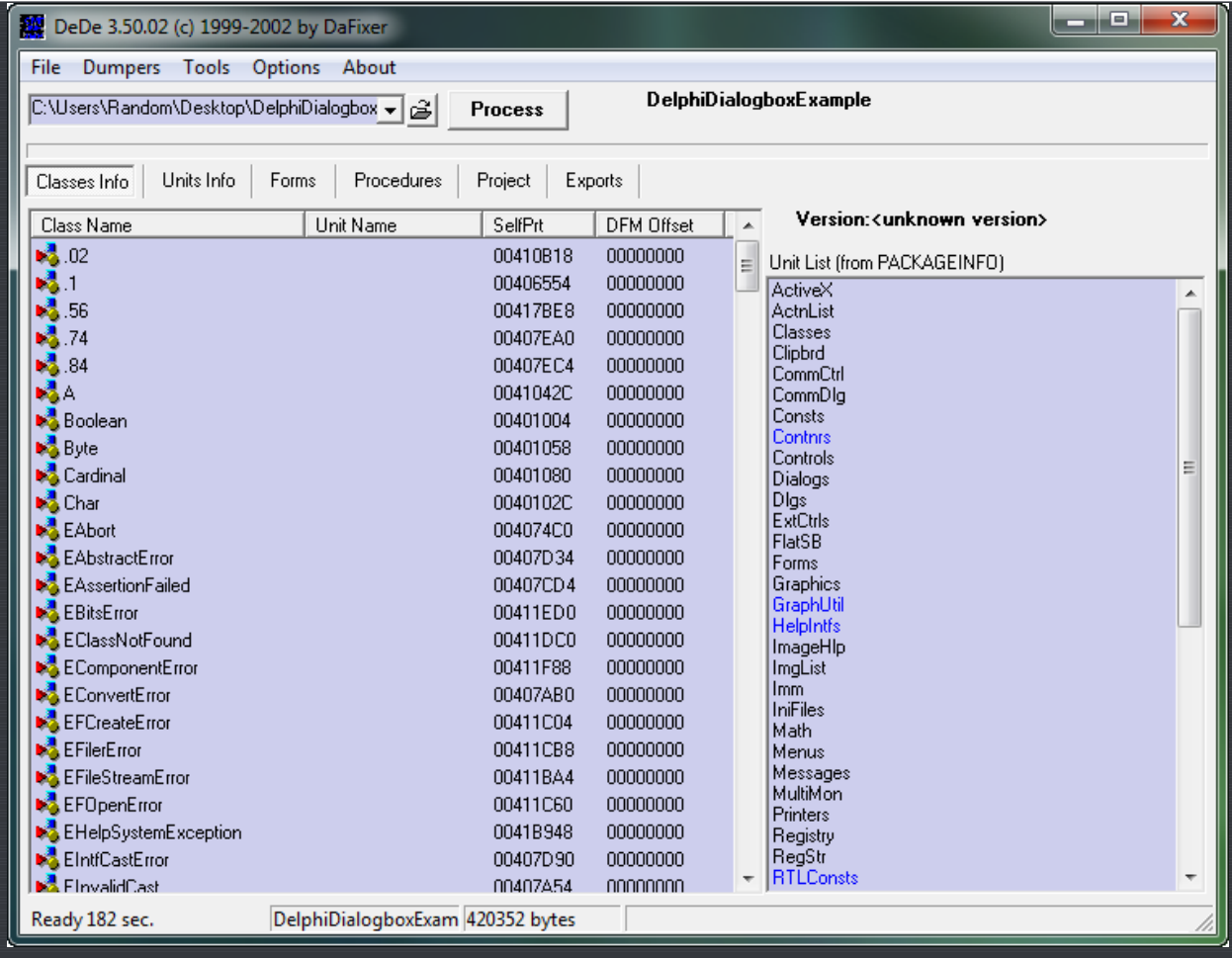
at the same time, it runs the target as we can see the target's main window appear:



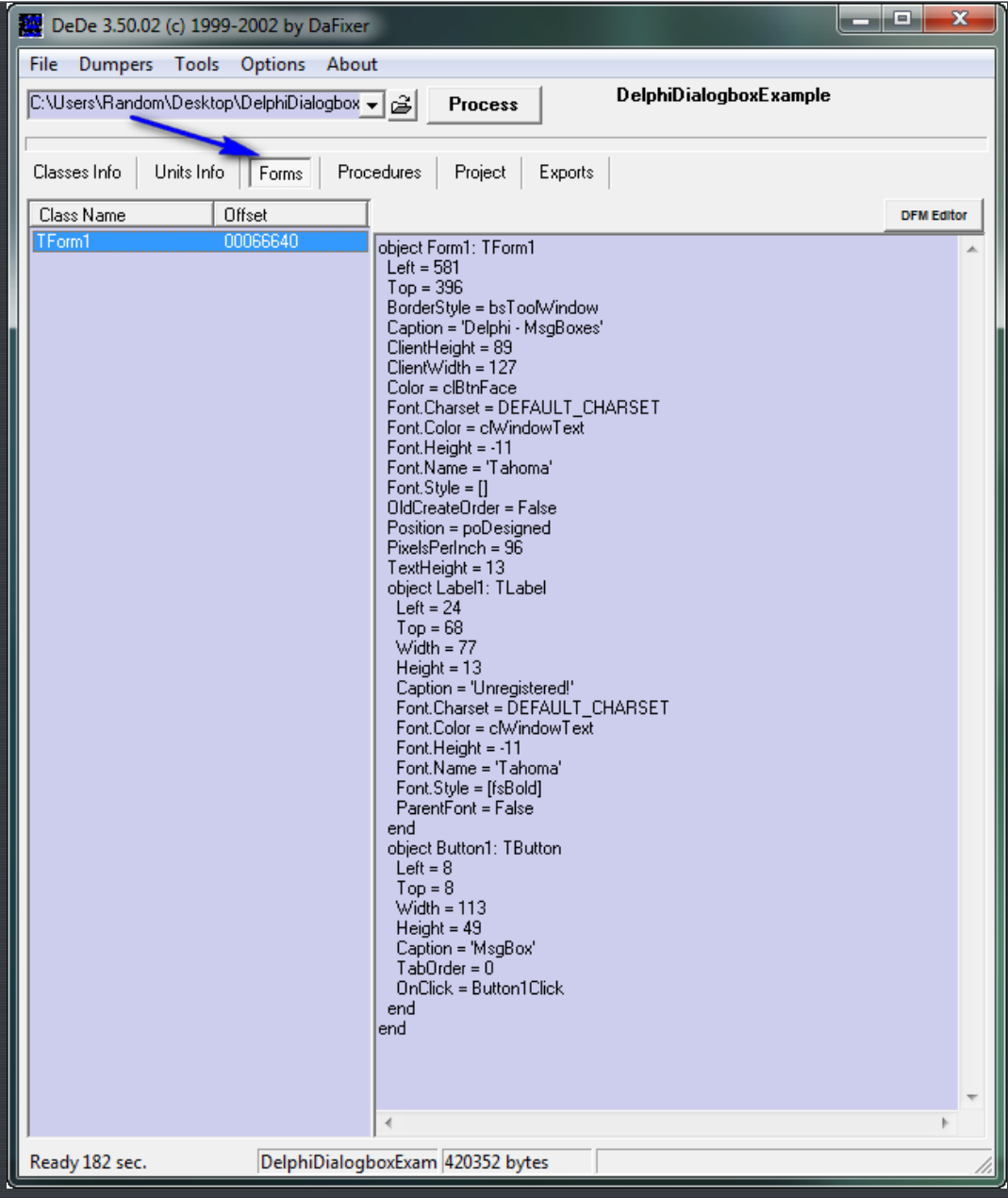
Sometimes, if there is a nag or splash that pops up at he beginning of running the target, you will want to proceed to the main part of the program before telling DeDe to process the app. In this case, the target is already fully loaded, so you can click the OK button and allow DeDe to proceed. deDe will then close the target and ask if you want to perform more robust processing on the app:



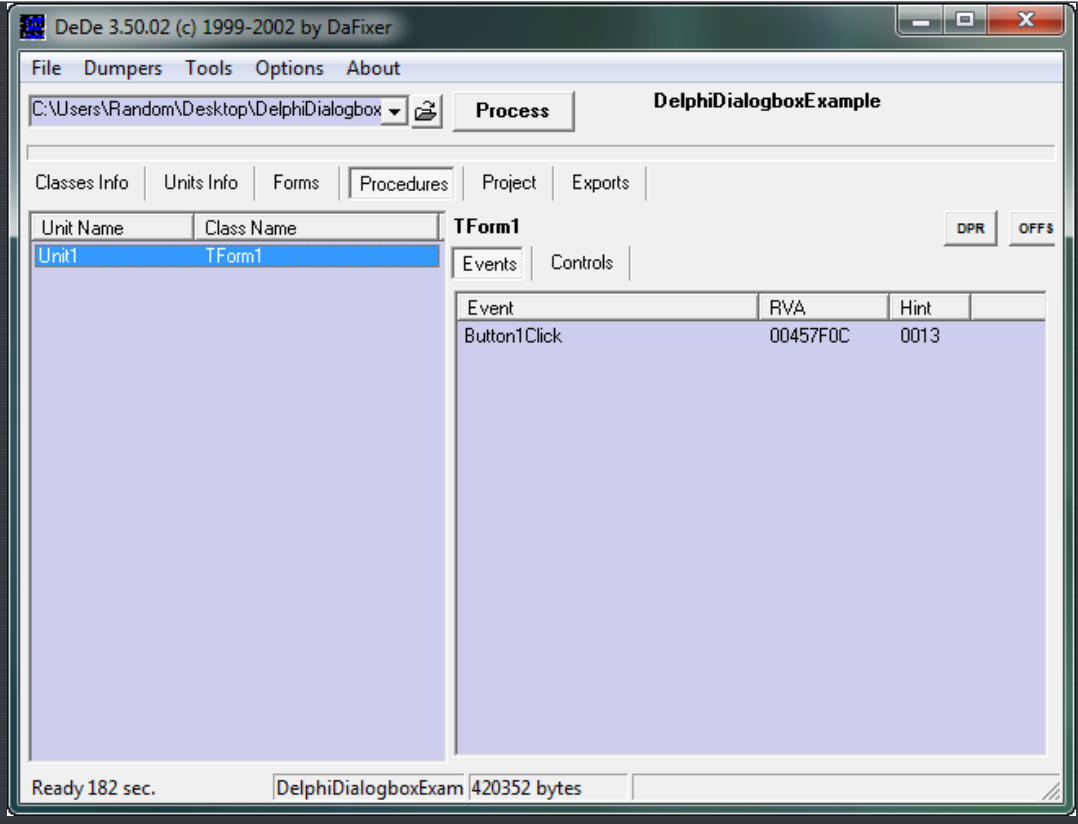
I usually choose no here, as clicking yes has never given me any more info that I need. DeDe will then finish processing the target and the main window will be populated:



DeDe defaults to showing us the class info, as we can see by the “Classes Info” tab being depressed. You can scroll through the list if you like, but what we want is the “Forms” tab:



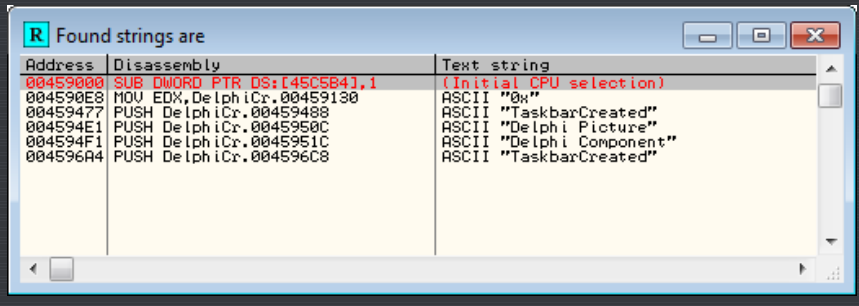
Here, we can see the info we saw in Resource Hacker; the attributes for the form. The reason I show you this is so that in the future you can bypass the Resource Hacker step and just look at it in DeDe. Now click on the “Procedures” tab. This is the most important tab in DeDe:



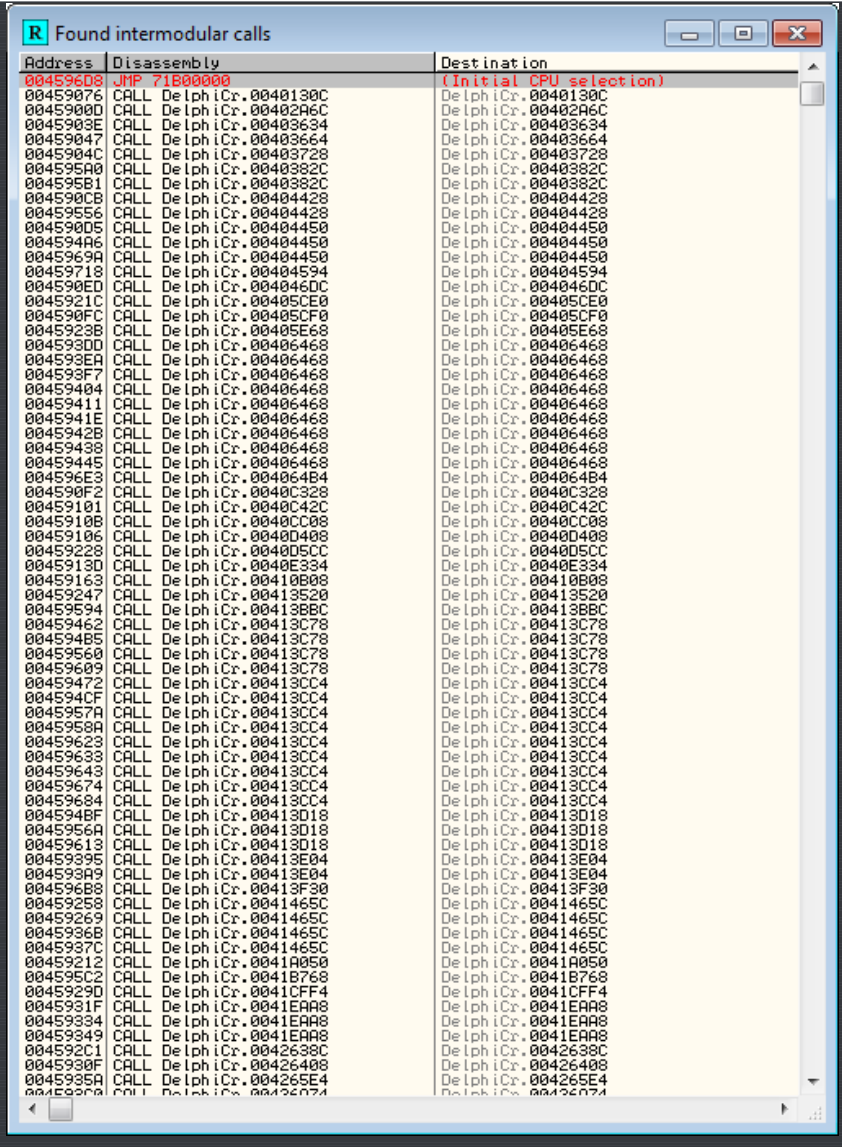
Here, DeDe shows us the method callback names and addresses for the TForm1 form. Since this is a very simple program, there is only one button, and therefore, only one callback. BUT, the nice thing is we now know the address of this callback- 457F0C. Remember that address! Now let's load the crackme into Olly and see what we can do…

Finding the patch

If you do a search for strings you will see that you are our of luck:



Searching for intermodular calls also is a dead giveaway for a Delphi program:



You will notice that Delphi makes A LOT of calls.

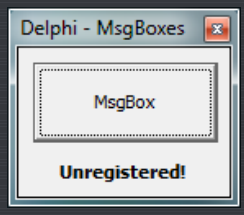
Normally, at this point, we would try to run the app and pause it on the bad boy message, but you will find that this won't work in this case because when you return, you will be about 15 layers down in calls, and finding the actual code that handles the processing of the message box is almost impossible.

BUT, if we recall from DeDe, we know the address that is called when the button is clicked. It's 457F0C. So let's go check that address out in Olly:

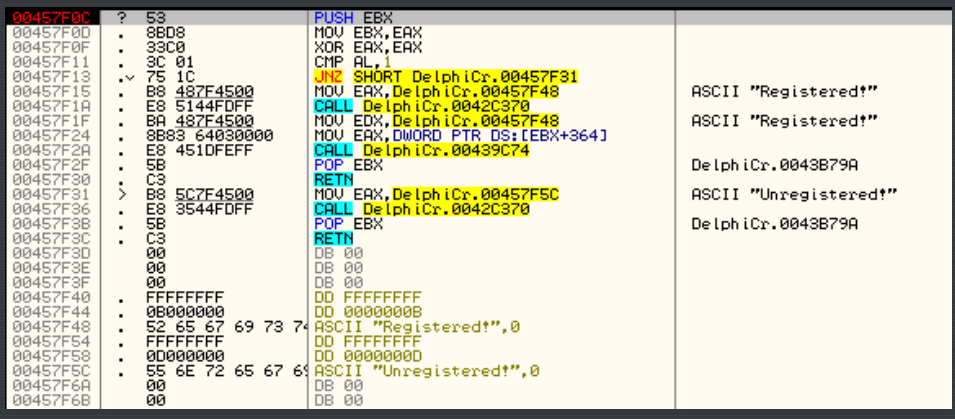
```
00457F0C   ?  53              PUSH EBX
00457F0D   .  8BD8            MOV EBX,EAX                 kernel32.BaseThreadInitThunk
00457F0F   .  33C0            XOR EAX,EAX                 kernel32.BaseThreadInitThunk
00457F11   .  3C 01           CMP AL,1
00457F13   .v 75 1C           JNZ SHORT DelphiCr.00457F31
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                     kernel32.75B2339A
00457F30   .  C3              RETN
00457F31   >  B8 5C7F4500     MOV EAX,DelphiCr.00457F5C   ASCII "Unregistered!"
00457F36   .  E8 3544FDFF     CALL DelphiCr.0042C370
00457F3B   .  5B              POP EBX                     kernel32.75B2339A
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 74 ASCII "Registered!",0
00457F54   .  FFFFFFFF        DD FFFFFFFF
00457F58   .  0D000000        DD 0000000D
00457F5C   .  55 6E 72 65 67 69 ASCII "Unregistered!",0
00457F6A      00              DB 00
00457F6B      00              DB 00
00457F6C   .  55              PUSH EBP
00457F6D   .  8BEC            MOV EBP,ESP
00457F6F   .  33C0            XOR EAX,EAX                 kernel32.BaseThreadInitThunk
```

Ahhhhh. That looks much better 🙂 Let's place a BP at the beginning of this (457F0C) and run the app:

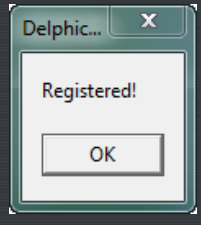
*** Olly may give you a warning that you are setting a BP in a data section. Just ignore it. ***



Notice it says we are unregistered. Also, notice the name in the title. “Delphi – MsgBoxes”. And there is one button. All this should look familiar. Go ahead and press the button and Olly will break at our BP:



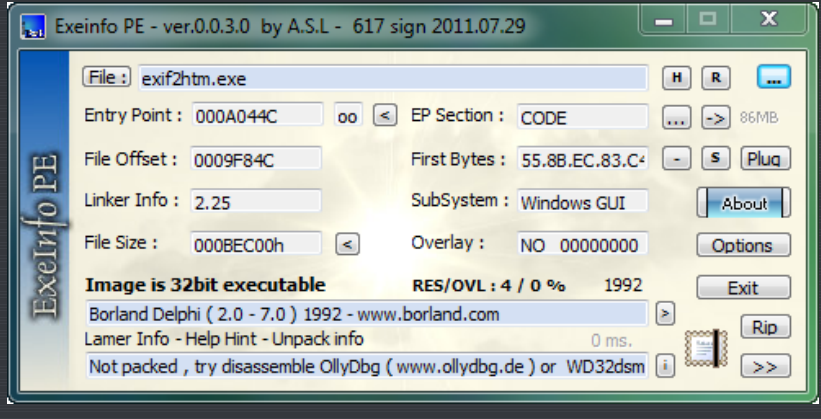
Now it is just a matter of stepping the code and leading Olly the right way to display the good boy instead of the bad boy. I don’t need to tell you how to do this. (If I do, please go back and re-read the tutorials in this series.)



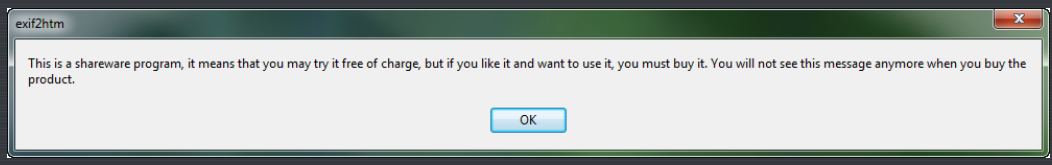
The second target

In my quest to find targets that will not hurt anyone, the second app we will be looking at is freeware, though it does have a nag that is displayed until you register the app. Registration is free. It was the least downloaded app in the “tools” category on Cnet, with 4 downloads in the last year. It is a program called Exif2htm. I have no idea what an exif file is, but apparently you can convert them into html files using this program.

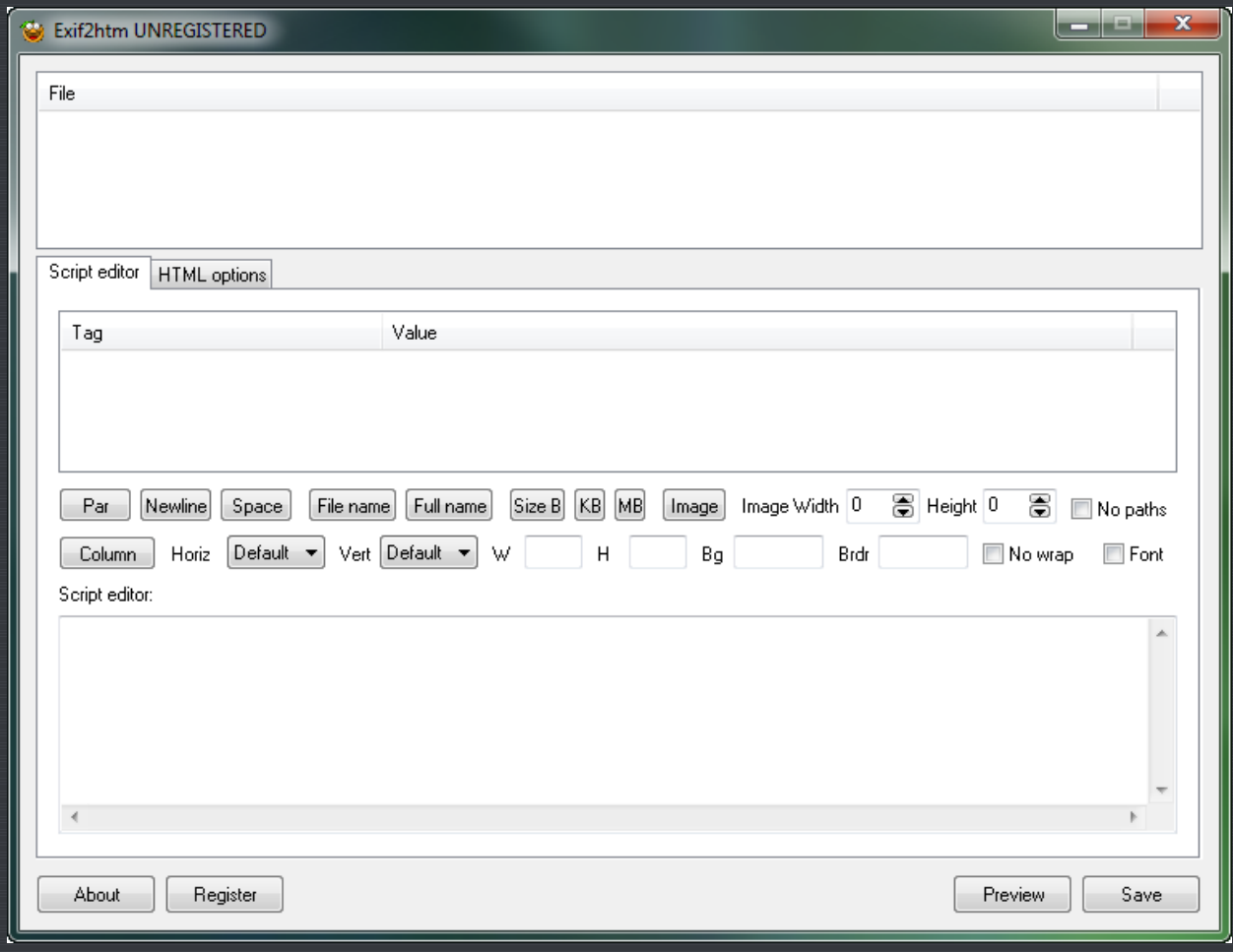
When we load the app into ExeInfoPE we see that it is in fact a Delphi program and not packed:



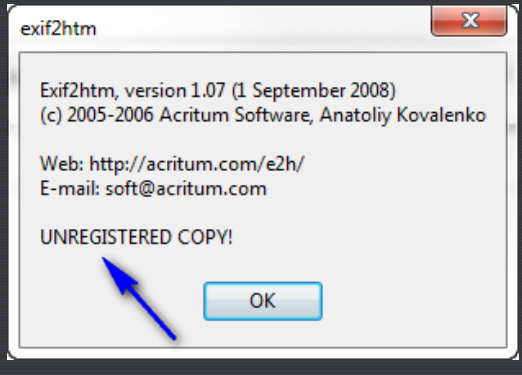
Running the app, we can see the nag popup:



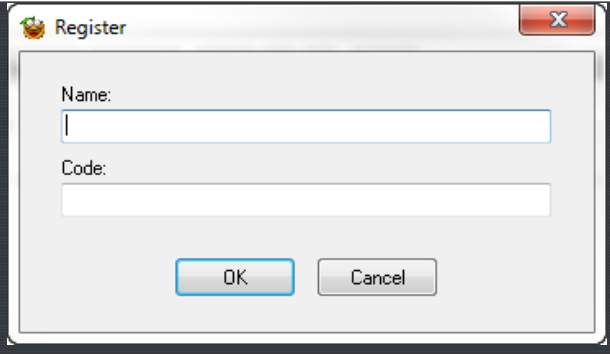
Clicking OK we can see what took a graphic designer a long time to create such a colorful UI:



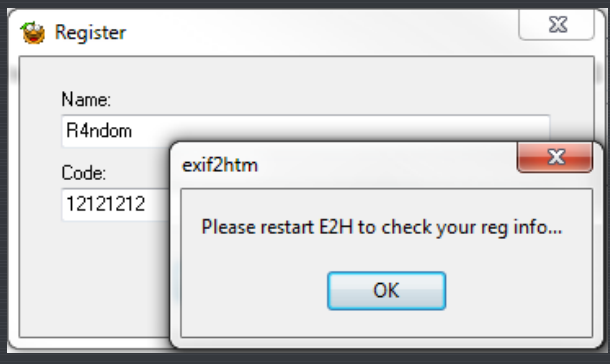
Anyway, we can see it is UNREGISTERED at the top. Clicking “About” gives us the about screen:



and clicking “Register” on the bottom gives us the registration screen:

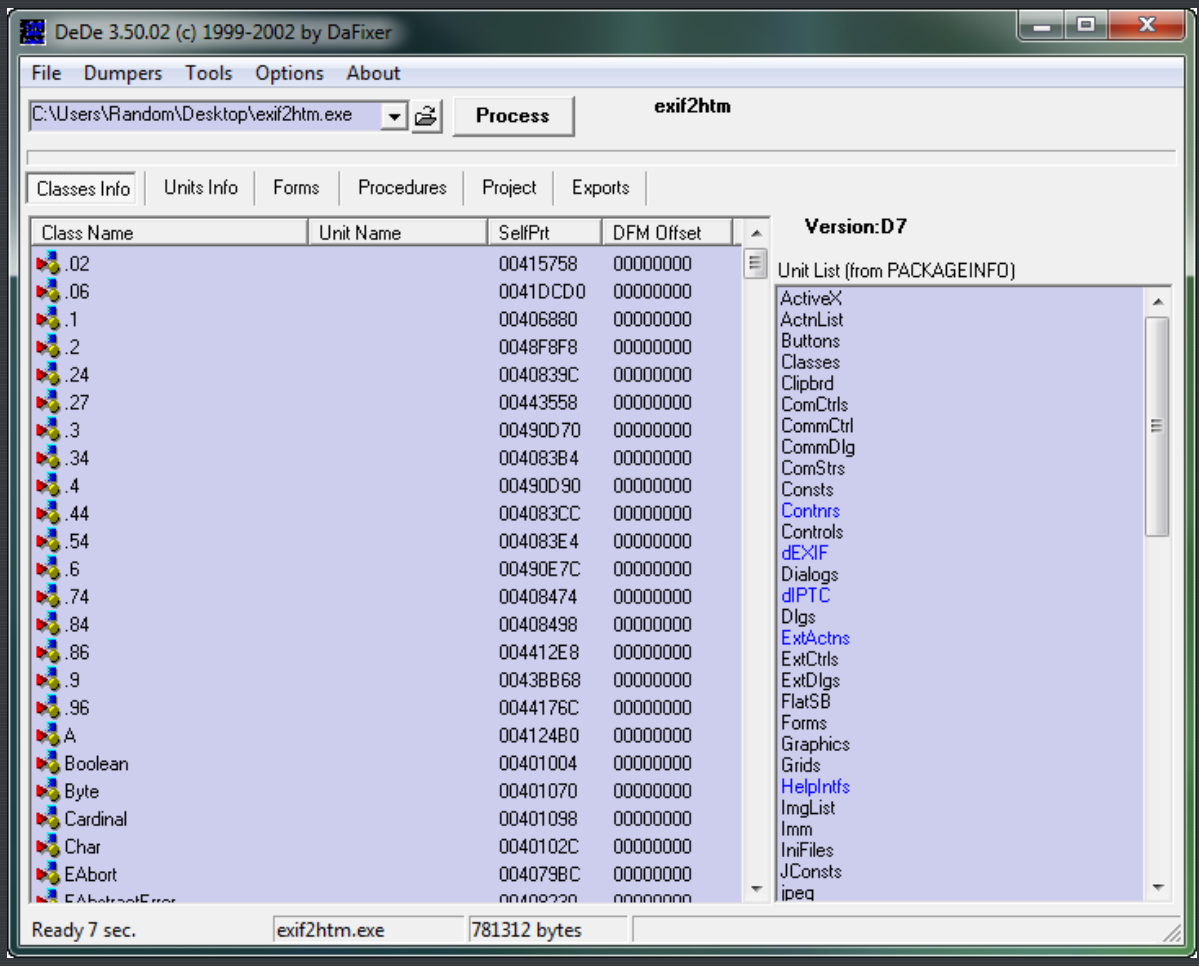


Entering false data gives us an unfortunate message:

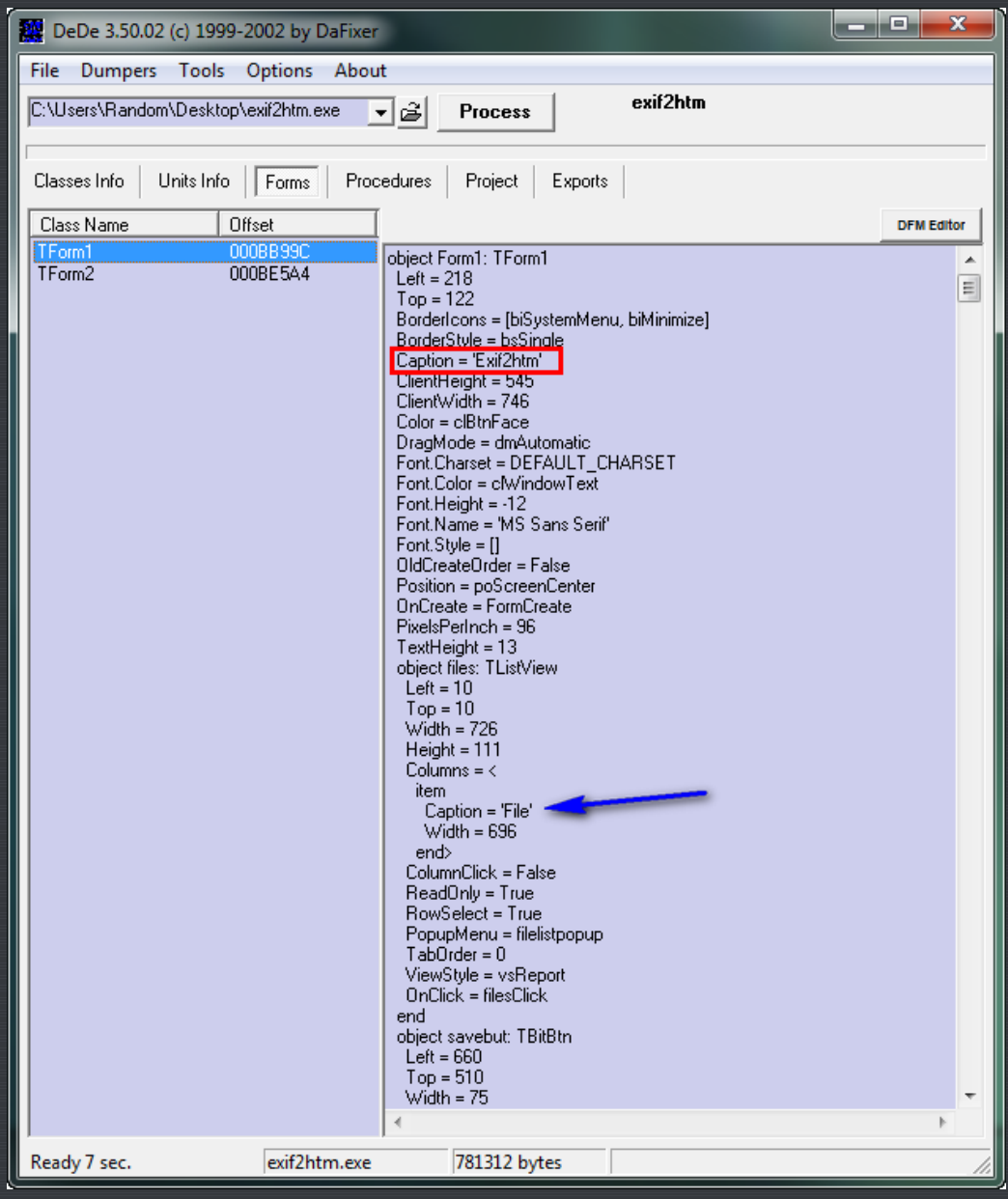


Using DeDe

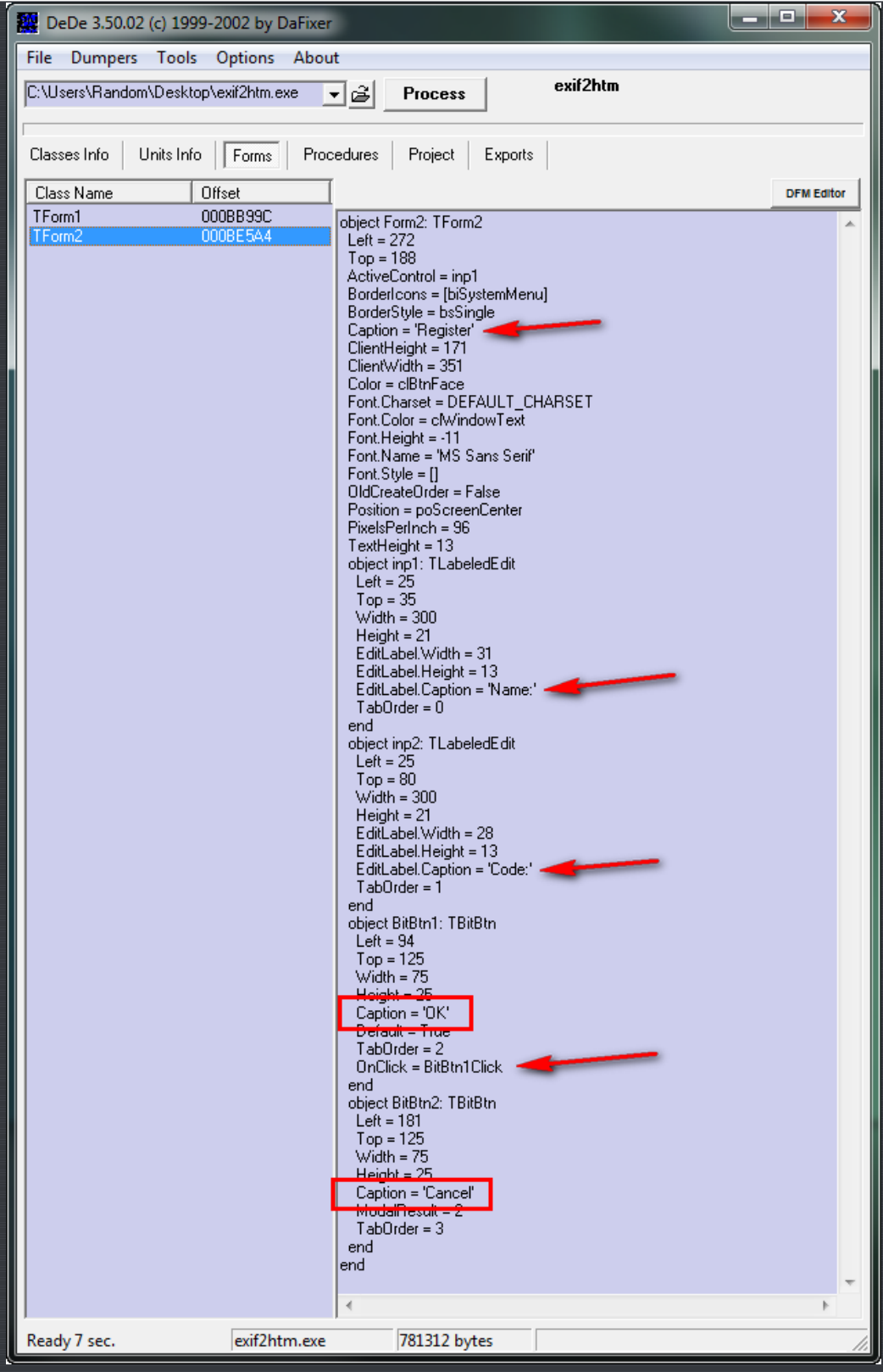
Let's load the app into DeDe and see what we can see. Make sure you click past the nag before hitting OK for DeDe to proceed:



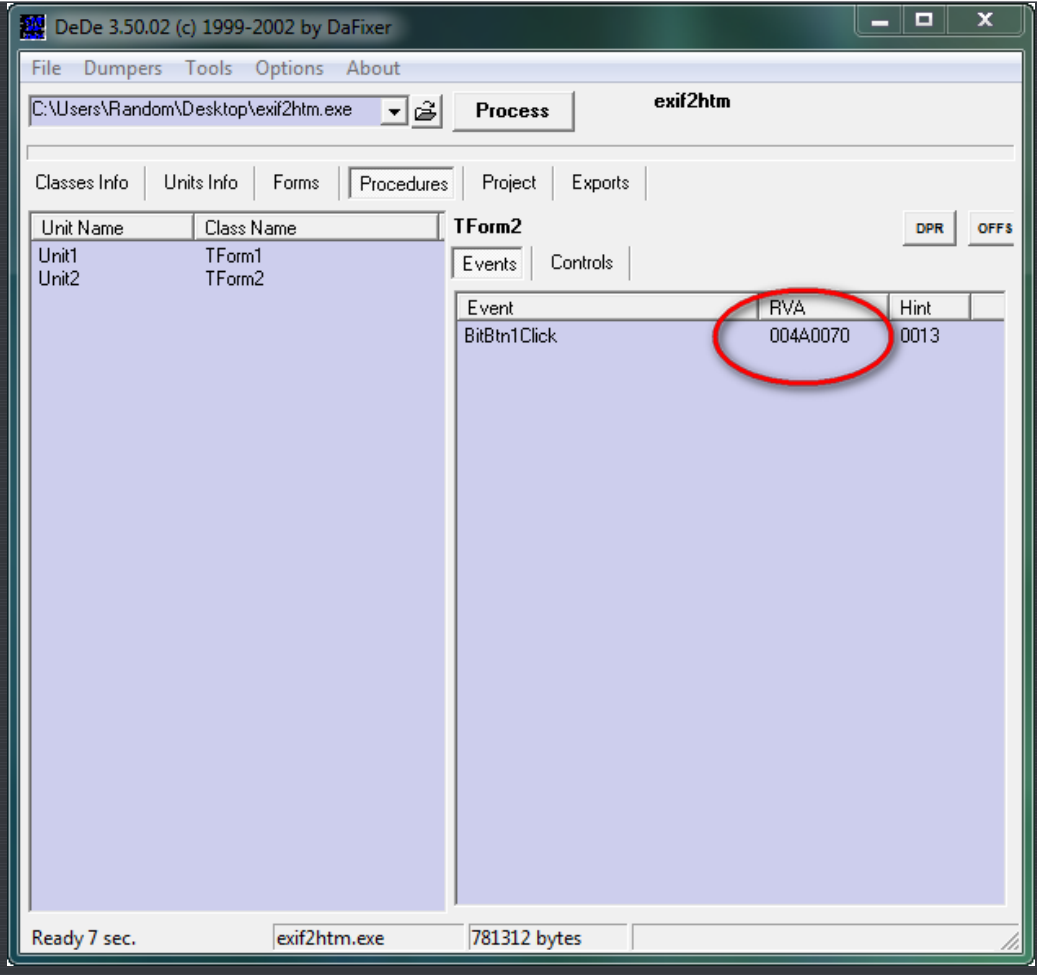
Clicking on Forms gives us the forms window. Clicking on TForm1, we can see that this is probably the main window:



We can also see a caption for Files, and if we look down farther, there are more for “save”, “print” and “editor”. Clicking on TForm2 is far more interesting, though:



So the caption is ‘Register’, we can see the labels for the edit boxes are ‘Name’ and ‘Code’, and we can see two buttons at the bottom, ‘OK’, and ‘Cancel’. Let’s click on the “Procedures” tab and see what we’ve got:



Clicking on Unit2 – TForm2 we can see that there is one method, “BitBtn1Click”, which we saw in the forms window corresponds with the “OK” button on the bottom of the registration window. We can also see that DeDe shows the RVA of the address for this method. Something tells me that is going to be very helpful here. Let’s write down that address and load up the app in Olly.

Finding the patch

This does not look like a very friendly binary in Olly:

```
004A044C  $- E9 AFFB6571      JMP 71B00000
004A0451     F0               DB F0
004A0452     B8               DB B8
004A0453  .  0C024A00         DD exif2htm.004A020C
004A0457     E8               DB E8
004A0458     E8               DB E8
004A0459     62               DB 62                          CHAR 'b'
004A045A     F6               DB F6
004A045B  .- FFA1 8C964A00    JMP DWORD PTR DS:[ECX+4A968C]  exif2htm.004AABEC
004A0461     8B               DB 8B
004A0462     00               DB 00
004A0463     E8               DB E8
004A0464     2C               DB 2C                          CHAR ','
004A0465     72               DB 72                          CHAR 'r'
004A0466     FD               DB FD
004A0467     FF               DB FF
004A0468     8B               DB 8B
004A0469     0D               DB 0D
004A046A  .  9C974A00         DD exif2htm.004A979C
004A046E     A1               DB A1
004A046F  .  8C964A00         DD exif2htm.004A968C
004A0473     8B               DB 8B
004A0474     00               DB 00
004A0475     8B               DB 8B
004A0476     15               DB 15
004A0477  .  28944900         DD exif2htm.00499428
004A047B     E8               DB E8
004A047C     2C               DB 2C                          CHAR ','
004A047D     72               DB 72                          CHAR 'r'
004A047E     FD               DB FD
004A047F     FF               DB FF
004A0480     8B               DB 8B
004A0481     0D               DB 0D
004A0482  .  D8974A00         DD exif2htm.004A97D8
004A0486     A1               DB A1
004A0487  .  8C964A00         DD exif2htm.004A968C
004A048B     8B               DB 8B
004A048C     00               DB 00
004A048D     8B               DB 8B
004A048E     15               DB 15
004A048F  .  B0FE4900         DD exif2htm.0049FEB0
004A0493     E8               DB E8
004A0494  .  14 72            ADC AL,72
004A0496  .  FD               STD
004A0497  .- FFA1 8C964A00    JMP DWORD PTR DS:[ECX+4A968C]  exif2htm.004AABEC
004A049D     8B               DB 8B
004A049E     00               DB 00
004A049F     E8               DB E8
004A04A0     88               DB 88
```

Let's go to our saved address and see what we have (I placed a BP on it to remember it):

```
004A006D     00               DB 00
004A006E     8BC0             MOV EAX,EAX                    kernel32.BaseThreadInitThunk
004A0070  ┌. 55               PUSH EBP
004A0071  |. 8BEC             MOV EBP,ESP
004A0073  |. 33C9             XOR ECX,ECX
004A0075  |. 51               PUSH ECX
004A0076  |. 51               PUSH ECX
004A0077  |. 51               PUSH ECX
004A0078  |. 51               PUSH ECX
004A0079  |. 51               PUSH ECX
004A007A  |. 53               PUSH EBX
004A007B  |. 56               PUSH ESI
004A007C  |. 8BD8             MOV EBX,EAX                    kernel32.BaseThreadInitThunk
004A007E  |. 33C0             XOR EAX,EAX                    kernel32.BaseThreadInitThunk
004A0080  |. 55               PUSH EBP
004A0081  |. 68 53014A00      PUSH exif2htm.004A0153
004A0086  |. 64:FF30          PUSH DWORD PTR FS:[EAX]
004A0089  |. 64:8920          MOV DWORD PTR FS:[EAX],ESP
004A008C  |. 8D55 FC          LEA EDX,[LOCAL.1]
004A008F  |. 8B83 F8020000    MOV EAX,DWORD PTR DS:[EBX+2F8]
004A0095  |. E8 7A79FBFF      CALL exif2htm.00457A14
004A009A  |. 837D FC 00       CMP [LOCAL.1],0
004A009E  |v 0F84 8C000000    JE exif2htm.004A0130
004A00A4  |. 8D55 F8          LEA EDX,[LOCAL.2]
004A00A7  |. 8B83 FC020000    MOV EAX,DWORD PTR DS:[EBX+2FC]
004A00AD  |. E8 6279FBFF      CALL exif2htm.00457A14
004A00B2  |. 837D F8 00       CMP [LOCAL.2],0
004A00B6  |v 74 78            JE SHORT exif2htm.004A0130
004A00B8  |. B2 01            MOV DL,1
004A00BA  |. A1 CC794100      MOV EAX,DWORD PTR DS:[4179CC]
004A00BF  |. E8 2037F6FF      CALL exif2htm.004037E4
004A00C4  |. 8BF0             MOV ESI,EAX                    kernel32.BaseThreadInitThunk
004A00C6  |. 8D55 F4          LEA EDX,[LOCAL.3]
004A00C9  |. 8B83 F8020000    MOV EAX,DWORD PTR DS:[EBX+2F8]
004A00CF  |. E8 4079FBFF      CALL exif2htm.00457A14
004A00D4  |. 8B55 F4          MOV EDX,[LOCAL.3]
004A00D7  |. 8BC6             MOV EAX,ESI
004A00D9  |. 8B08             MOV ECX,DWORD PTR DS:[EAX]
004A00DB  |. FF51 38          CALL DWORD PTR DS:[ECX+38]
004A00DE  |. 8D55 F0          LEA EDX,[LOCAL.4]
004A00E1  |. 8B83 FC020000    MOV EAX,DWORD PTR DS:[EBX+2FC]
004A00E7  |. E8 2879FBFF      CALL exif2htm.00457A14
004A00EC  |. 8B55 F0          MOV EDX,[LOCAL.4]
004A00EF  |. 8BC6             MOV EAX,ESI
004A00F1  |. 8B08             MOV ECX,DWORD PTR DS:[EAX]
004A00F3  |. FF51 38          CALL DWORD PTR DS:[ECX+38]
004A00F6  |. 8B15 F0954A00    MOV EDX,DWORD PTR DS:[4A95F0]  exif2htm.004AAC58
004A00FC  |. 8B12             MOV EDX,DWORD PTR DS:[EDX]
004A00FE  |. 8D45 EC          LEA EAX,[LOCAL.5]
004A0101  |. B9 68014A00      MOV ECX,exif2htm.004A0168      ASCII "reginfo.dat"
004A0106  |. E8 A547F6FF      CALL exif2htm.004048B0
004A010B  |. 8B55 EC          MOV EDX,[LOCAL.5]
004A010E  |. 8BC6             MOV EAX,ESI
004A0110  |. 8B08             MOV ECX,DWORD PTR DS:[EAX]
004A0112  |. FF51 74          CALL DWORD PTR DS:[ECX+74]
004A0115  |. 8BC6             MOV EAX,ESI
004A0117  |. E8 F836F6FF      CALL exif2htm.00403814
004A011C  |. B8 7C014A00      MOV EAX,exif2htm.004A017C      ASCII "Please restart E2H to check your reg info..."
004A0121  |. E8 BA1FF9FF      CALL exif2htm.004A20E0
004A0126  |. A1 30AD4A00      MOV EAX,DWORD PTR DS:[4AAD30]
```

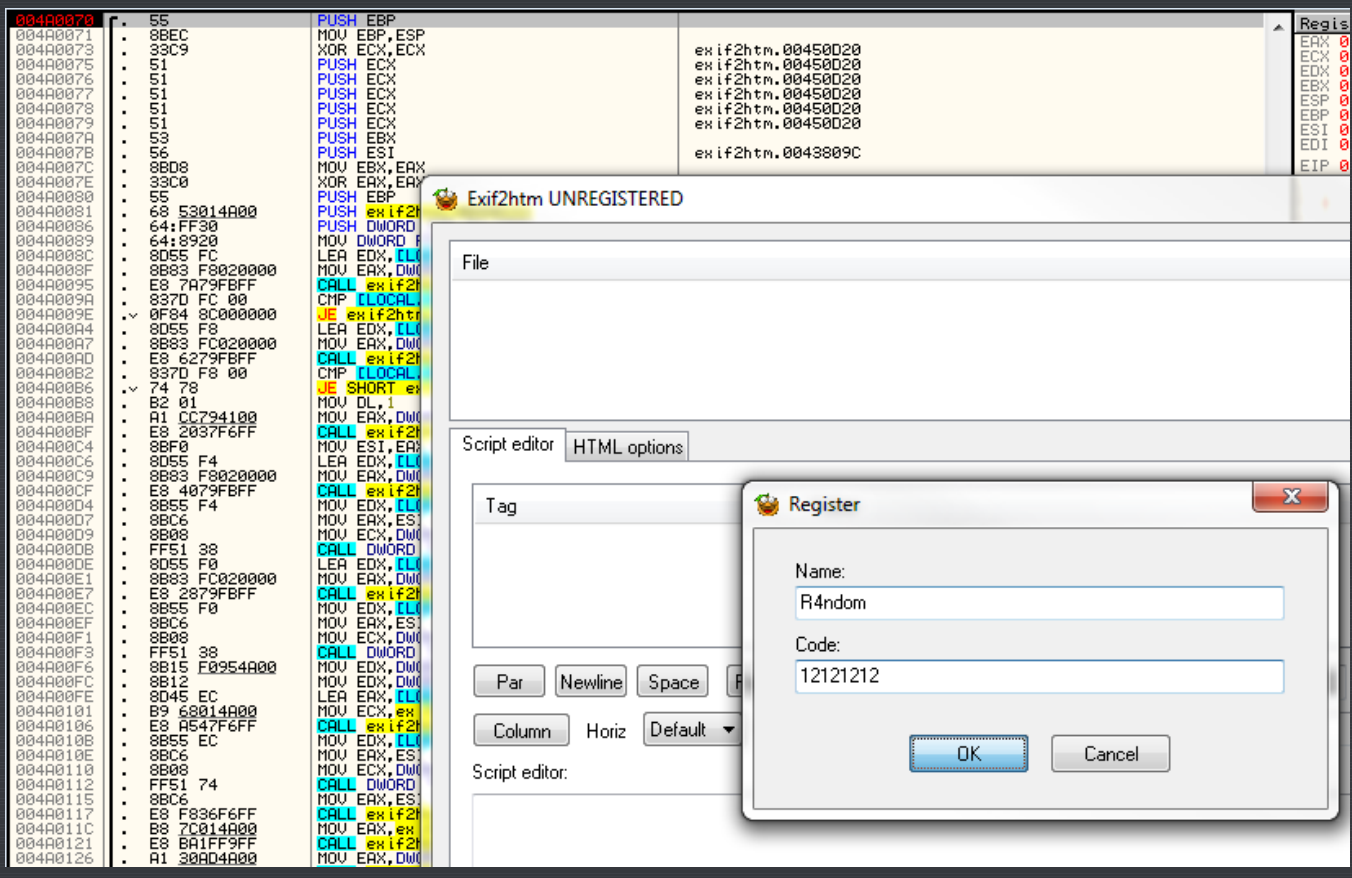
Toward the end we can see the message that comes up telling us to restart he app to see if our registration code worked. This is a technique used quite often, and it does provide a small challenge to us. The problem is we don't know if we typed in the right code until we restart, and the area that checks if it was right could be anywhere in the program. We also can't force our code to be right as we don't know where the instructions are that check if it's correct or not! But using a little common sense, you will see that it's no that much harder to get past a protection like this.

The hint comes from the fact that, after you enter your code, the app must store it somewhere (or at least

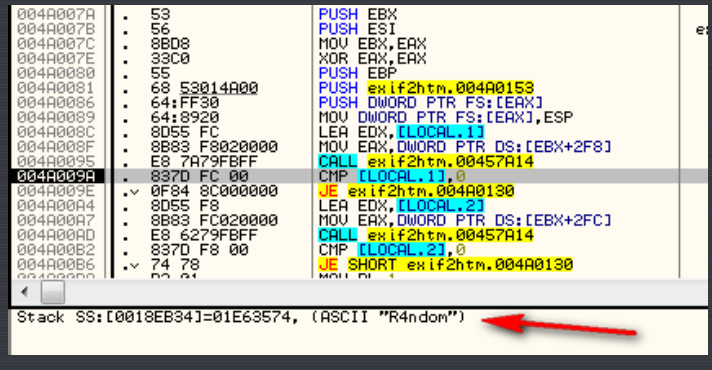
store SOMETHING somewhere) so that when it restarts it can look and see if we stored the correct code in or not. It could either store our entered name and code, or it could store a flag that we are registered, or any of a number of other things. The point is, though, it must store something.

There are only a couple places a program can store data from run to run of the app. Almost always it's the registry or an ini file. So what we have to do is find out where this data is stored so that when we run the app again we can find where this data is processed and checked for if we're registered or not.

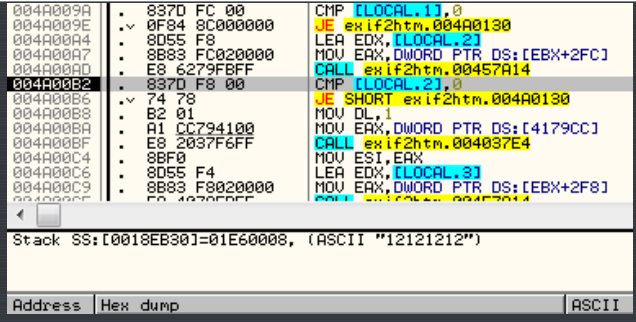
We have our BP set, so let's run the app in Olly. Click on Register, enter some fake data, and click OK. Olly should break at our BP:



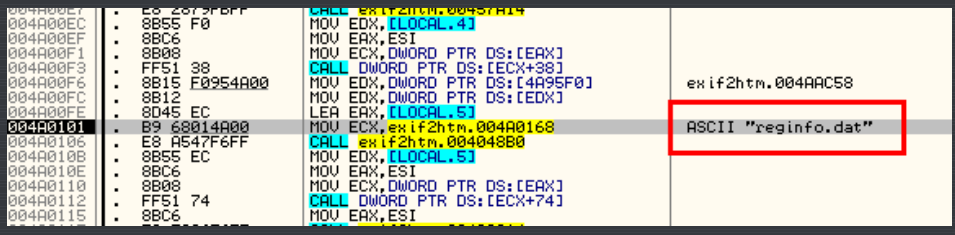
Now let's look at the code where Olly broke. At first is a bunch of pushes that set up a bunch of variables on the stack. We then push some varibles onto the stack and make a call at 4A0095. If you step over the code (stopping at address 4A009A, you will see something interesting in the info window:



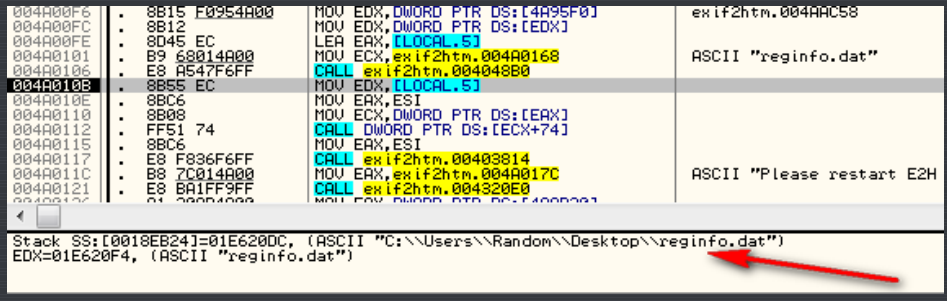
It appears it is doing something with our username. 99.99999999999999999% of the time this will be a check to make sure we actually entered something into the edit text field in the registration window (usually returning a length). The fact that EAX equals 6 on return helps support this hypothesis, though we don't know for sure. There is a check if EAX equals zero right after this and a jump. I'm sure you can guess what that's for. Slowly stepping over the next couple instructions, we then see our code pop up in the window as well:



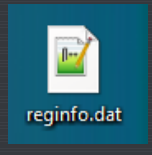
What do you want to bet that this is doing the same thing on our code? EAX again is compared with zero (it is 8 this time, which just happens to be the length of my code 😀) and jumps if it is zero. Next we pass a couple more calls, each of which loads our name and code as arguments again. You can step over this code until we get to the big red flag at address 4A0101:



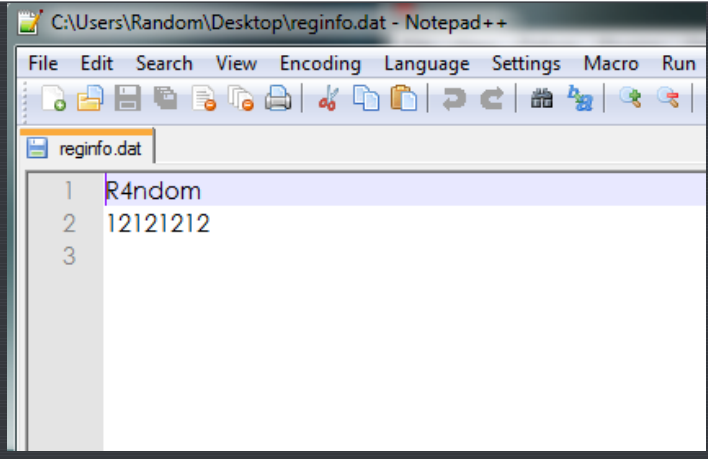
This looks very important. It is a data file that looks like it is going to create. Stepping over the next two lines brings us to address 4A010B:



What appears to be a file path is displayed in the info window. It points to the location I am currently running the app from (my desktop). In this file path is the name of the dat file it looks like it's going to create. Stepping a couple more times until you step over the call at 4A0112, you may notice a little something pop up on your desktop:



Hmmm. That wasn't there before. My file has a Notepad++ icon because that's what I've associated .dat files with- yours may look different. Let's open this file and have a look (you can open it in any text editor):



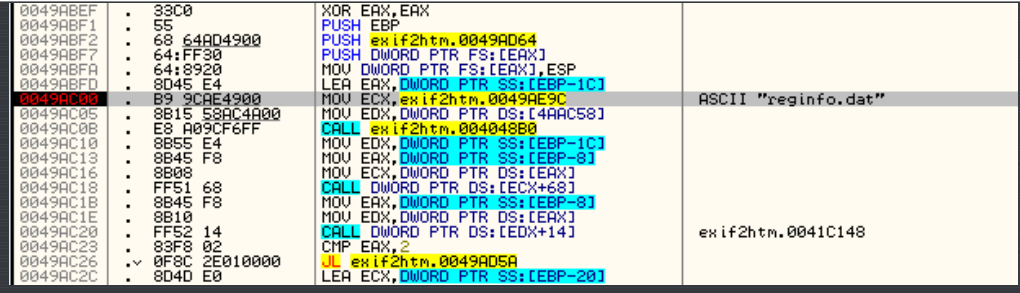
Well, I think we found out where our info will be stored 🙂 We know that the app creates a file called reginfo.dat in the same folder as the app is stored, and in this file is saved our entered name and code. Now that we know how the program is going to check for if we are registered or not, we can use this to find the code. Go in to Olly and do a search for strings and search for “reginfo.dat”. Mine came up with two instances, the second of which was the area we were just looking at where the reginfo file was created. The first looks very interesting though:

```
0049A8F4 ASCII "..",0
0049A900 ASCII ".",0
0049AB37 MOV EDX,exif2htm.0049AE2C        ASCII "Unregistered User"
0049AB77 MOV EDX,exif2htm.0049AE48        ASCII "dd.mm.yyyy"
0049AB86 MOV EDX,exif2htm.0049AE5C        ASCII "hh:nn:ss"
0049AB95 MOV EDX,exif2htm.0049AE70        ASCII "1.07"
0049ABA4 MOV EDX,exif2htm.0049AE80        ASCII "1 September 2008"
0049AC00 MOV ECX,exif2htm.0049AE9C        ASCII "reginfo.dat"
0049AD7F MOV EAX,exif2htm.0049AEB0        ASCII "This is a shareware program, it means that you may try it fre
0049ADA9 MOV EDX,exif2htm.0049AF7C        ASCII " UNREGISTERED"
0049AE2C ASCII "Unregistered Use"
0049AE3C ASCII "r",0
0049AE48 ASCII "dd.mm.yyyy",0
0049AE5C ASCII "hh:nn:ss",0
0049AE70 ASCII "1.07",0
```

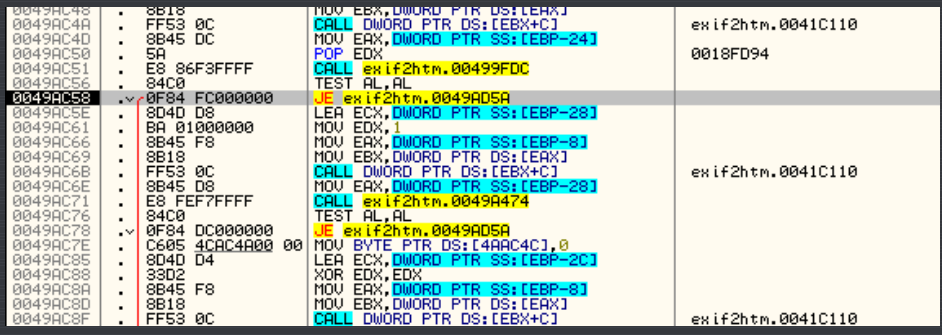
I placed a BP on the reference to reginfo and double clicked to see what the code looks like:

```
0049ABB3  .  E8 947EF6FF      CALL exif2htm.00402A4C
0049ABB8  .  8B45 E8          MOV EAX,DWORD PTR SS:[EBP-18]
0049ABBB  .  8D55 EC          LEA EDX,DWORD PTR SS:[EBP-14]
0049ABBE  .  E8 69EAF6FF      CALL exif2htm.0040962C
0049ABC3  .  8B55 EC          MOV EDX,DWORD PTR SS:[EBP-14]
0049ABC6  .  B8 58AC4A00      MOV EAX,exif2htm.004AAC58
0049ABCB  .  E8 289AF6FF      CALL exif2htm.004045F8
0049ABD0  .  8D45 F4          LEA EAX,DWORD PTR SS:[EBP-C]
0049ABD3  .  E8 CC99F6FF      CALL exif2htm.004045A4
0049ABD8  .  8D45 F0          LEA EAX,DWORD PTR SS:[EBP-10]
0049ABDB  .  E8 C499F6FF      CALL exif2htm.004045A4
0049ABE0  .  B2 01            MOV DL,1
0049ABE2  .  A1 CC794100      MOV EAX,DWORD PTR DS:[4179CC]
0049ABE7  .  E8 F88BF6FF      CALL exif2htm.004037E4
0049ABEC  .  8945 F8          MOV DWORD PTR SS:[EBP-8],EAX
0049ABEF  .  33C0             XOR EAX,EAX
0049ABF1  .  55               PUSH EBP
0049ABF2  .  68 64AD4900      PUSH exif2htm.0049AD64
0049ABF7  .  64:FF30          PUSH DWORD PTR FS:[EAX]
0049ABFA  .  64:8920          MOV DWORD PTR FS:[EAX],ESP
0049ABFD  .  8D45 E4          LEA EAX,DWORD PTR SS:[EBP-1C]
0049AC00  .  B9 9CAE4900      MOV ECX,exif2htm.0049AE9C        ASCII "reginfo.dat"
0049AC05  .  8B15 58AC4A00    MOV EDX,DWORD PTR DS:[4AAC58]
0049AC0B  .  E8 A09CF6FF      CALL exif2htm.004048B0
0049AC10  .  8B55 E4          MOV EDX,DWORD PTR SS:[EBP-1C]    exif2htm.0043809C
0049AC13  .  8B45 F8          MOV EAX,DWORD PTR SS:[EBP-8]
0049AC16  .  8B08             MOV ECX,DWORD PTR DS:[EAX]
0049AC18  .  FF51 68          CALL DWORD PTR DS:[ECX+68]
0049AC1B  .  8B45 F8          MOV EAX,DWORD PTR SS:[EBP-8]
0049AC1E  .  8B10             MOV EDX,DWORD PTR DS:[EAX]
0049AC20  .  FF52 14          CALL DWORD PTR DS:[EDX+14]
0049AC23  .  83F8 02          CMP EAX,2
0049AC26  .v 0F8C 2E010000    JL exif2htm.0049AD5A
0049AC2C  .  8D4D E0          LEA ECX,DWORD PTR SS:[EBP-20]
0049AC2F  .  BA 01000000      MOV EDX,1
0049AC34  .  8B45 F8          MOV EAX,DWORD PTR SS:[EBP-8]
0049AC37  .  8B18             MOV EBX,DWORD PTR DS:[EAX]
```

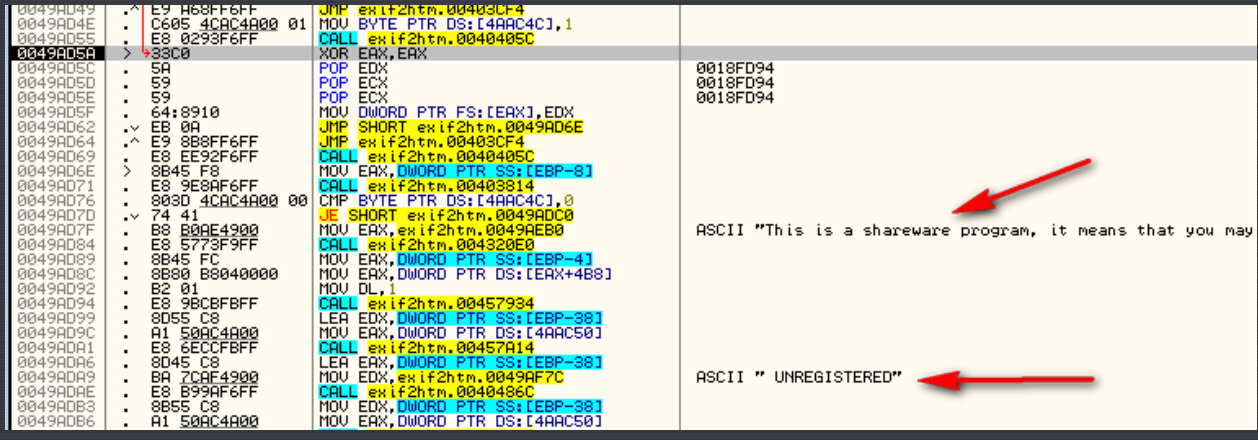
Scrolling up, we can see that there are no conditional jumps, though below our BP we see several. Go ahead and close the app (clicking run in Olly and clicking OK in the registry window and closing the app through the app- not through Olly. We want to make sure all of the code that stores the registered info get’s done). Now, right when we re-start the app we break at our new BP:



Now lets single step to see what's going on…At the first conditional jump at address 49AC26 we do not jump. This could be OK or it could be a potential place we want to patch, we don't know yet, so let's keep stepping. The next set of instructions loads our username and code from the data file and performs some calls with it. This is looking much more likely. When we get to the next conditional jump, we see that we are indeed going to jump:



Stepping in we can see where we jump to:



This is not looking good. Continuing to step, we will eventually run the "Shareware" code, so we know we have gone too far. Let's re-start the app and see what happens if we don't make that last jump.

*** You may wonder why I didn't try patching the conditional jump at address 49AC26. The answer is, I did, and we still got he bad boy message ***

Step until you get to the second conditional jump at address 49AC58:

```
0049AC43  .  33D2            XOR EDX,EDX
0049AC45  .  8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]
0049AC48  .  8B18            MOV EBX,DWORD PTR DS:[EAX]
0049AC4A  .  FF53 0C         CALL DWORD PTR DS:[EBX+C]          exif2htm.0041C110
0049AC4D  .  8B45 DC         MOV EAX,DWORD PTR SS:[EBP-24]
0049AC50  .  5A              POP EDX                            0018FD94
0049AC51  .  E8 86F3FFFF     CALL exif2htm.00499FDC
0049AC56  .  84C0            TEST AL,AL
0049AC58  .v 0F84 FC000000   JE exif2htm.0049AD5A
0049AC5E  .  8D4D D8         LEA ECX,DWORD PTR SS:[EBP-28]
0049AC61  .  BA 01000000     MOV EDX,1
0049AC66  .  8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]
0049AC69  .  8B18            MOV EBX,DWORD PTR DS:[EAX]
0049AC6B  .  FF53 0C         CALL DWORD PTR DS:[EBX+C]          exif2htm.0041C110
0049AC6E  .  8B45 D8         MOV EAX,DWORD PTR SS:[EBP-28]
0049AC71  .  E8 FEF7FFFF     CALL exif2htm.0049A474
0049AC76  .  84C0            TEST AL,AL
0049AC78  .v 0F84 DC000000   JE exif2htm.0049AD5A
0049AC7E  .  C605 4CAC4A00 00 MOV BYTE PTR DS:[4AAC4C],0
0049AC85  .  8D4D D4         LEA ECX,DWORD PTR SS:[EBP-2C]
0049AC88  .  33D2            XOR EDX,EDX
0049AC8A  .  8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]
0049AC8D  .  8B18            MOV EBX,DWORD PTR DS:[EAX]
0049AC8F  .  FF53 0C         CALL DWORD PTR DS:[EBX+C]          exif2htm.0041C110
0049AC92  .  8B55 D4         MOV EDX,DWORD PTR SS:[EBP-2C]
0049AC95  .  B8 3CAC4A00     MOV EAX,exif2htm.004AAC3C
0049AC9A  .  E8 5999F6FF     CALL exif2htm.004045F8
0049AC9F  .  8D4D D0         LEA ECX,DWORD PTR SS:[EBP-30]
0049ACA2  .  BA 01000000     MOV EDX,1
0049ACA7  .  8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]
```

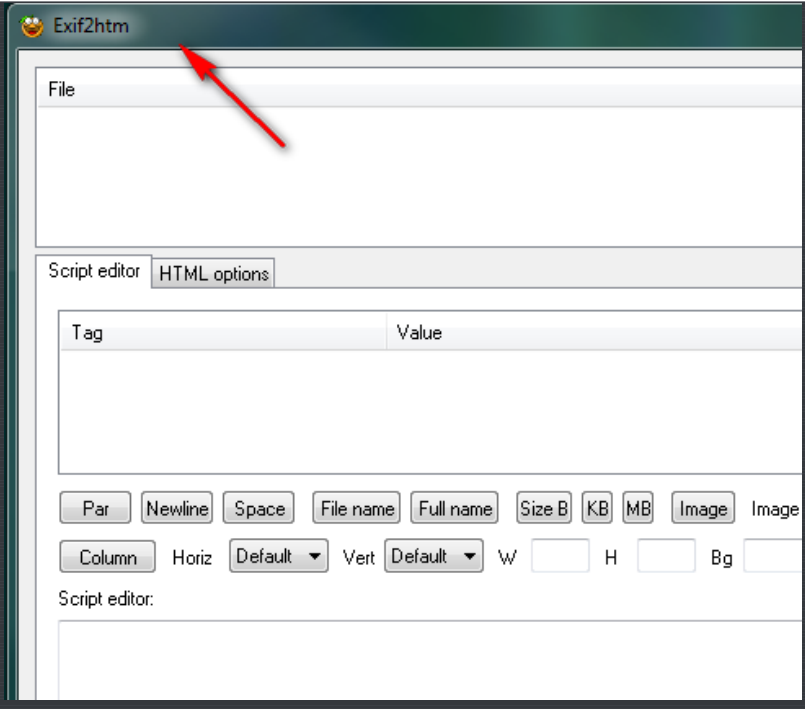
Now, let's tell Olly not to jump by changing he zero flag and keep stepping through code. We will eventually come to the last conditional jump at address 49AC78:

```
0049AC61  .  BA 01000000     MOV EDX,1
0049AC66  .  8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]
0049AC69  .  8B18            MOV EBX,DWORD PTR DS:[EAX]
0049AC6B  .  FF53 0C         CALL DWORD PTR DS:[EBX+C]          exif2htm.0041C110
0049AC6E  .  8B45 D8         MOV EAX,DWORD PTR SS:[EBP-28]
0049AC71  .  E8 FEF7FFFF     CALL exif2htm.0049A474
0049AC76  .  84C0            TEST AL,AL
0049AC78  .v 0F84 DC000000   JE exif2htm.0049AD5A
0049AC7E  .  C605 4CAC4A00 00 MOV BYTE PTR DS:[4AAC4C],0
0049AC85  .  8D4D D4         LEA ECX,DWORD PTR SS:[EBP-2C]
0049AC88  .  33D2            XOR EDX,EDX
0049AC8A  .  8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]
0049AC8D  .  8B18            MOV EBX,DWORD PTR DS:[EAX]
0049AC8F  .  FF53 0C         CALL DWORD PTR DS:[EBX+C]          exif2htm.0041C110
0049AC92  .  8B55 D4         MOV EDX,DWORD PTR SS:[EBP-2C]
0049AC95  .  B8 3CAC4A00     MOV EAX,exif2htm.004AAC3C
0049AC9A  .  E8 5999F6FF     CALL exif2htm.004045F8
0049AC9F  .  8D4D D0         LEA ECX,DWORD PTR SS:[EBP-30]
0049ACA2  .  BA 01000000     MOV EDX,1
0049ACA7  .  8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]
0049ACAA  .  8B18            MOV EBX,DWORD PTR DS:[EAX]
0049ACAC  .  FF53 0C         CALL DWORD PTR DS:[EBX+C]          exif2htm.0041C110
```

Looking down where this jump jumps to we can see that it is the same destination as the previous jump. If you want, you can step through it, though you will see that it is the same outcome, setting our app as shareware. This tells us that this is a second check on our name/code pair. Let's keep Olly from jumping by setting the zero flag again and keep going:

```
0049AC6B  .  FF53 0C         CALL DWORD PTR DS:[EBX+C]          exif2htm.0041C110
0049AC6E  .  8B45 D8         MOV EAX,DWORD PTR SS:[EBP-28]
0049AC71  .  E8 FEF7FFFF     CALL exif2htm.0049A474
0049AC76  .  84C0            TEST AL,AL
0049AC78  .v 0F84 DC000000   JE exif2htm.0049AD5A
0049AC7E  .  C605 4CAC4A00 00 MOV BYTE PTR DS:[4AAC4C],0
0049AC85  .  8D4D D4         LEA ECX,DWORD PTR SS:[EBP-2C]
0049AC88  .  33D2            XOR EDX,EDX
0049AC8A  .  8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]
0049AC8D  .  8B18            MOV EBX,DWORD PTR DS:[EAX]         exif2htm.00417A18
0049AC8F  .  FF53 0C         CALL DWORD PTR DS:[EBX+C]          exif2htm.0041C110
0049AC92  .  8B55 D4         MOV EDX,DWORD PTR SS:[EBP-2C]
0049AC95  .  B8 3CAC4A00     MOV EAX,exif2htm.004AAC3C
0049AC9A  .  E8 5999F6FF     CALL exif2htm.004045F8
0049AC9F  .  8D4D D0         LEA ECX,DWORD PTR SS:[EBP-30]
0049ACA2  .  BA 01000000     MOV EDX,1
0049ACA7  .  8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]
0049ACAA  .  8B18            MOV EBX,DWORD PTR DS:[EAX]         exif2htm.00417A18
```

Now if you keep stepping, you will notice that nothing noticeable happens, so go ahead and run the app. You will notice that our nag does not show and that the main window pops up. Also you will notice that there is no UNREGISTERED text anymore:



We have now forced the app to use whatever name and code have been entered! We have cracked the app

-Till next time

R4ndom