R4ndom's Tutorial #15: Using The Call Stack

by R4ndom on Jul.18, 2012, under Beginner, Reverse Engineering, Tutorials

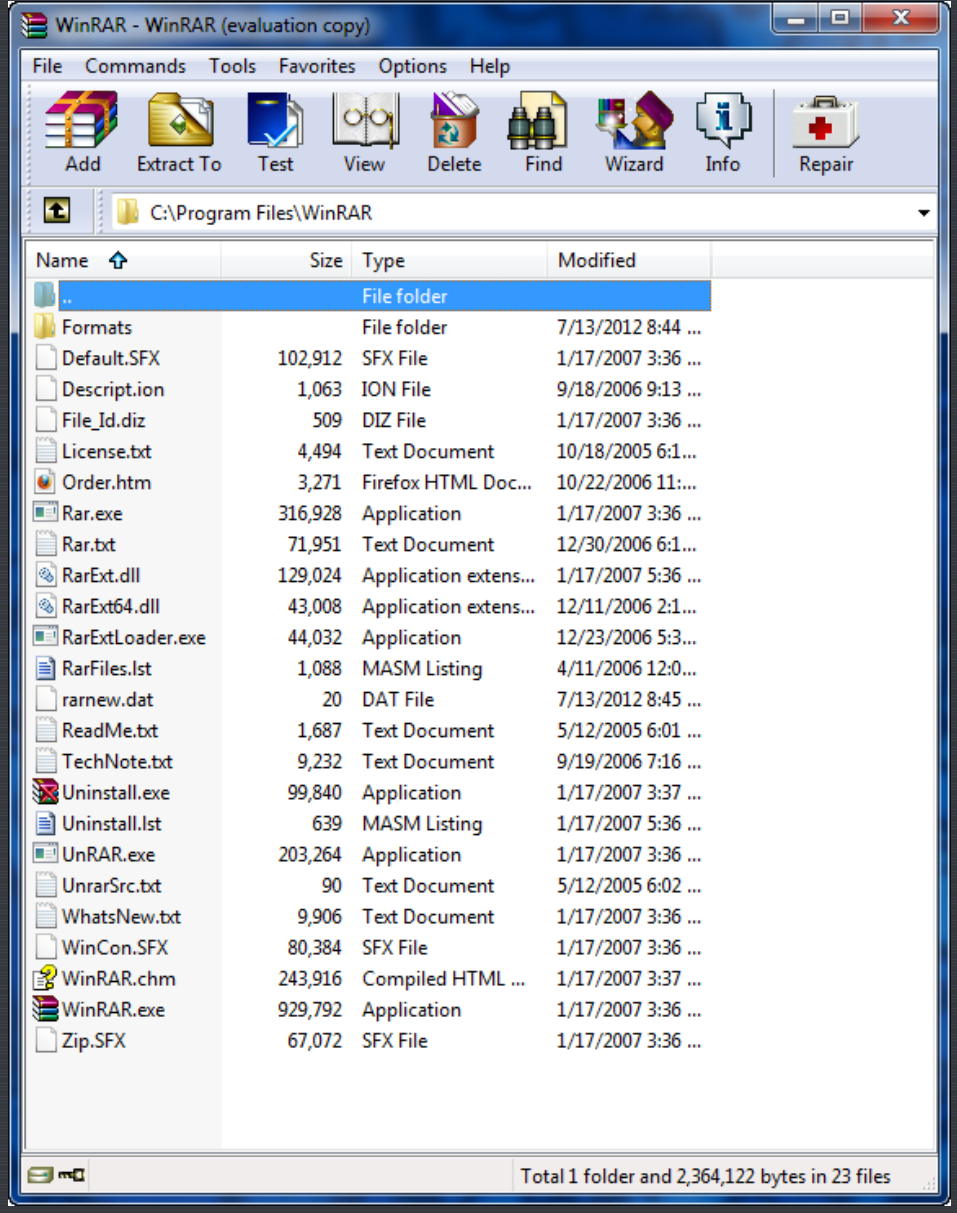
Introduction

In this tutorial we will be removing a nag from a 'real' program. In an attempt to help out the author's, who spend a great deal of time creating these apps, I have attempted to pick an app that will do the least amount of harm. This time, I did a Google search for "Cracked Software" and this program came up with the most hits, including tutorials, serial numbers, keygens, you name it. Because it is so incredibly easy to get a crack for this app, I figured someone would probably not have much trouble getting it anyway. But please, if you do like it, pay for it.

We will also be adding a couple tricks to our arsenal for reverse engineering. One note, if you are running these tutorials under 64-bit windows 7 (like I am), Olly 1.10, even my version, the call stack trick will not work. My suggestion is to do what I do: Run Olly 2.0 just to perform the trick (and get the correct address) then switch back over to my version of Olly for the rest of it. Or just use Olly 2.0- there are a lot of nice features in it and it has been fixed to work properly with 64-bit operating systems.

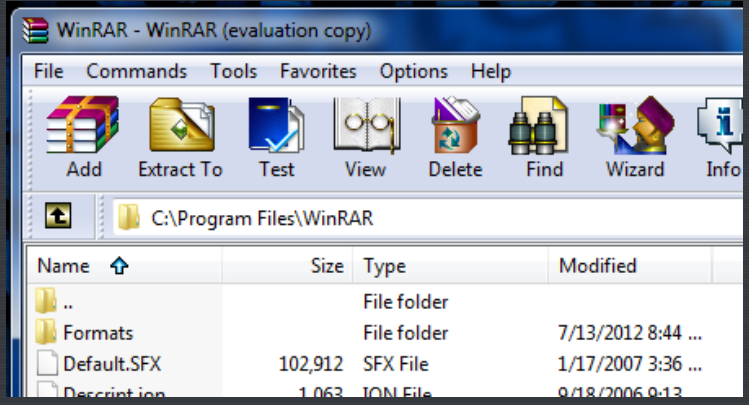
Investigate The App

This program comes with the restriction that after 40 days (probably a biblical thing?) a nag will appear, and trust me, I know from experience, it is very naggy. Unfortunately, since the nag doesn't appear for 40 days (and 40 nights? -sorry) you have two choices; you can install the app and wait 40 days before reading this tutorial or you can simply set your system time to today plus 41 days, do the tutorial, and reset the date back to today. Make sure you do one or the other before doing the tutorial or it won't match



and after a second…

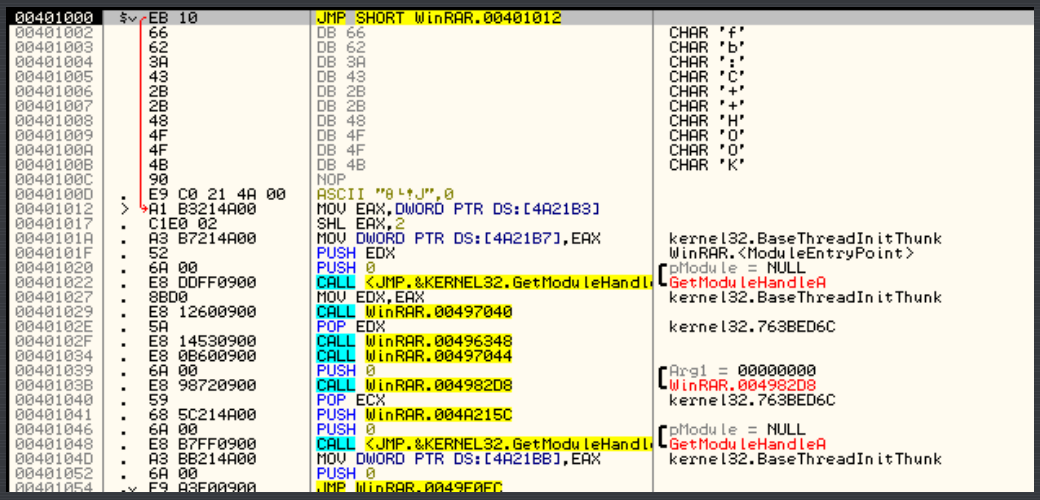
This nag pops up. It pops up a lot while using the app. It is highly annoying. Also, we can see at the top ‘evaluation copy’:



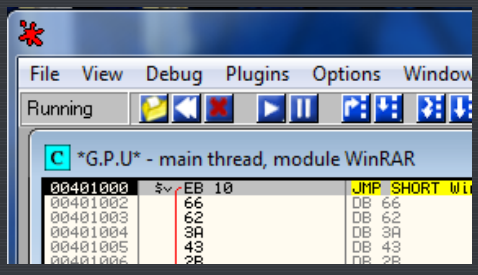
I am going to show you two ways to get to the relevant code.

The First Way

So let's load it in Olly and get started:



Start the app and wait for the nag to appear. Once it appears (and before closing it) click in Olly and click the pause button (next to the play button):



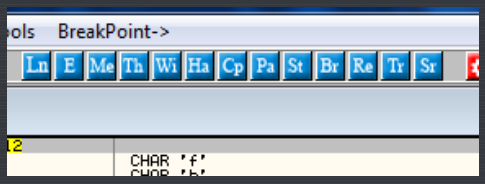
Now, we want to find out where that nag came from, and ultimately, what decided to show it. Of course we could search for strings or intermodular calls, but I guarantee you that these tricks will not be helpful for a lot of apps out there. So let's learn another trick…

The Call Stack

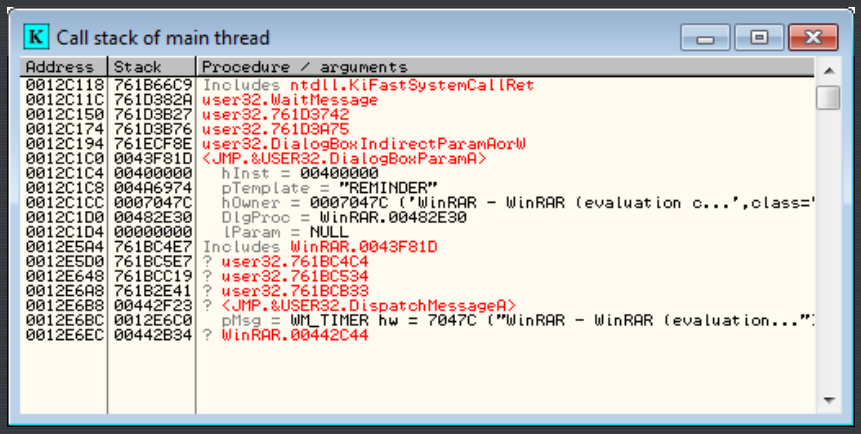
The call stack is Olly's attempt at tracing through the code that got us here and trying to figure out which functions were called. It also attempts to show you arguments that were passed to the function. All of this can be accomplished by using the 'normal' stack in the bottom right, but the call stack is a much nicer view of this data. Keep in mind that Olly is not perfect at this, though, and you can't use everything in this window as gospel (oops, I did it again). There is a lot of guessing going on. Also, many times, this window will be blank. This is usually caused by Olly getting completely lost or when reverse engineering a Visual

Basic program (VB programs don't call functions the same way as real programming languages do).

To view the call stack, click the "St" button if using my version of Olly:

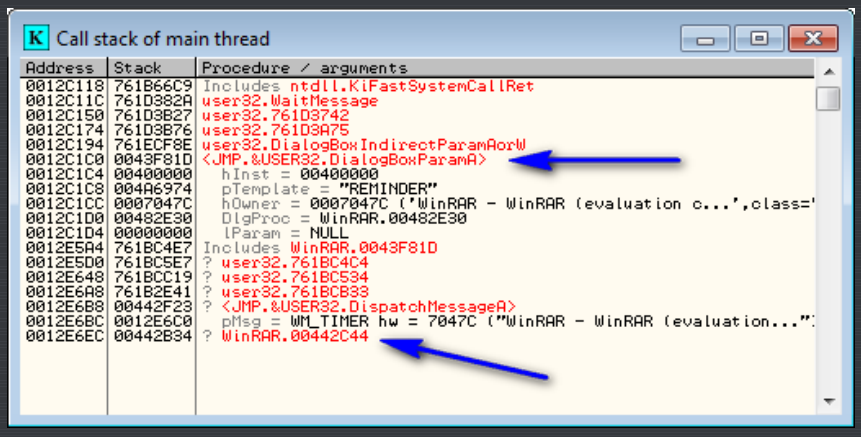


Or, if using the default version of Olly, click the 'K' toolbar button. Apparently, 'Call' starts with a 'K' in the author's native language:

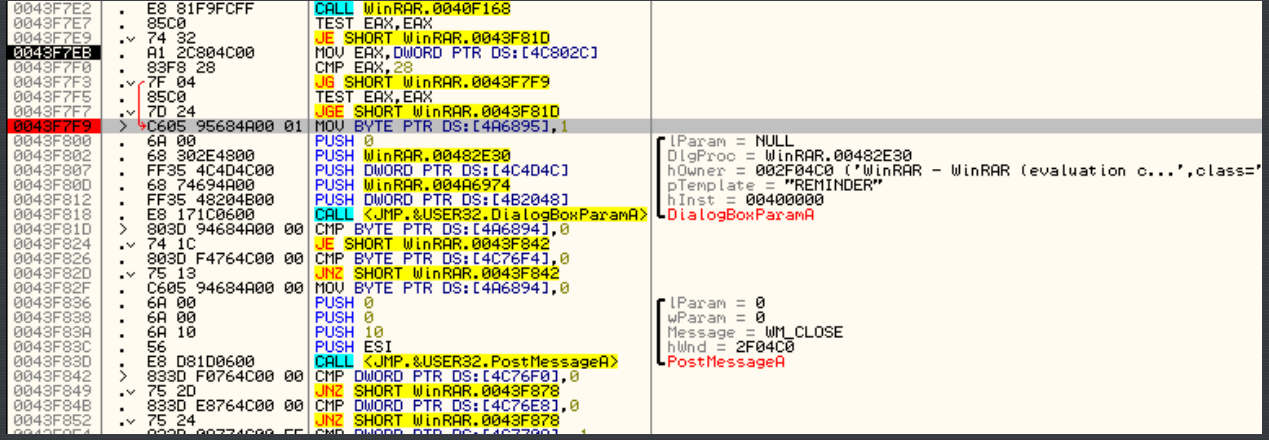


A couple things to point out…The most recent call is at the top, just like the stack. 'Includes' means that this instruction was involved in the call, but Olly doesn't know exactly how. A question mark means Olly is a little unsure about that line, so take it with a grain of salt.

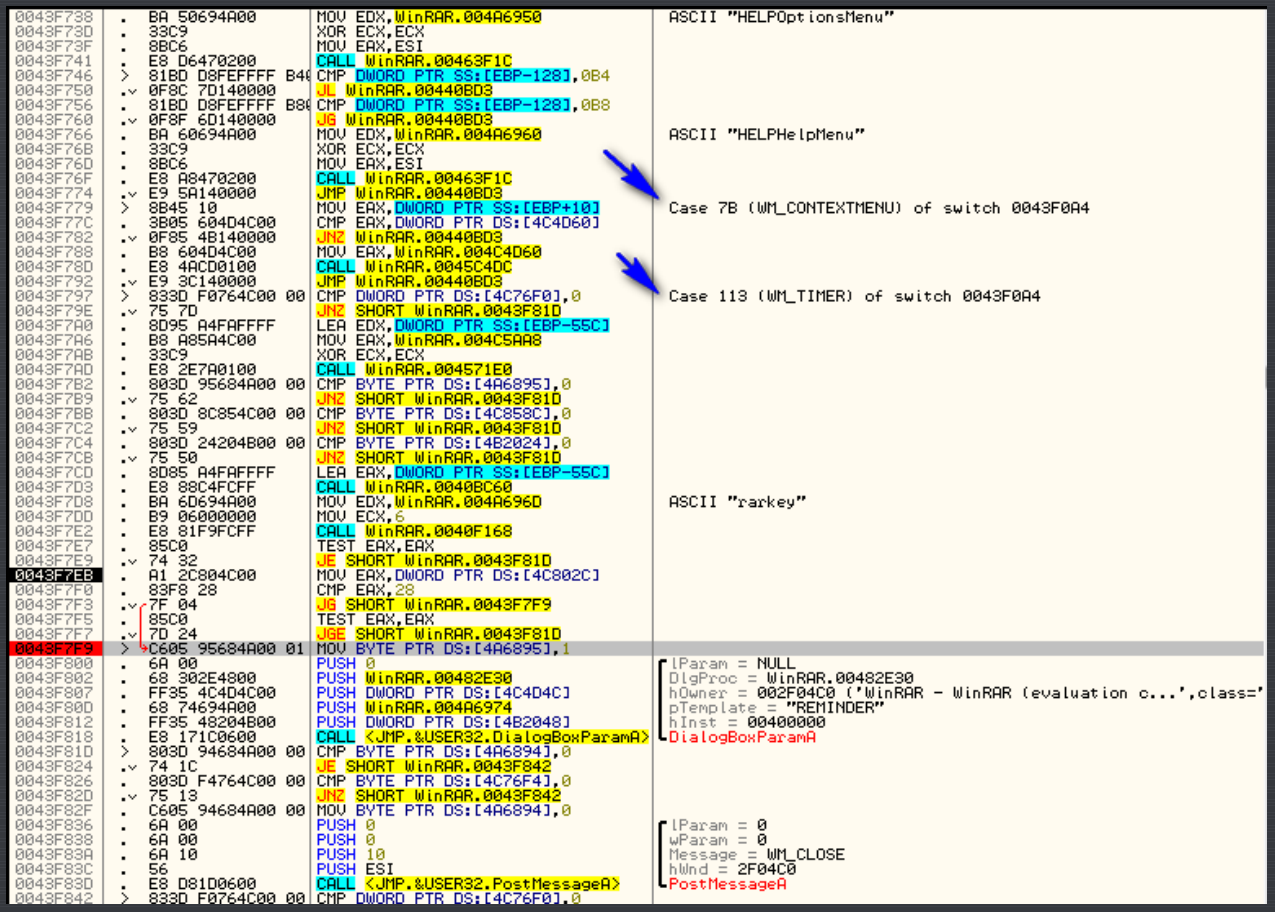
In our specific example, we can see a ntdll function, some user32 functions, a call to DialogBoxParamA with arguments, some more calls to user32, and a call at the bottom from our app, WinRAR. Here is how to think of this: WinRAR, at address 442C44, called DispatchMessageA, in this case with a message to display a dialog box. User32 then called the DialogBoxParamA function to display the dialog box with the text "evaluation copy" in the title, along with some other arguments. User32 then displayed this dialog and is currently waiting for our input, using WaitMessage to do it.



The important things in this window are the call to the dialog box and the call from our app. Generally, when using the call stack, you start at the top and find the first item that you can use to find the code section you are interested in. If that one doesn't work, you go further down the list, checking each function call until we get 'back' in the code far enough to find our compare/jump that determines if this function is called or not. Let's try checking out the call to DialogBoxParamA by double-clicking that line:



and we jump to the call to DialogBoxParamA. I have placed a BP at the beginning of this set of instructions that performs the setup and call to display the dialog. Above it, we can see several conditional jumps which pop out to us. If you scroll up farther, you will notice some comments that say “Case XX (WM_Something) of switch 0043F0A4″ where XX is a hex number:



This is Olly’s way of showing a switch statement. If you scroll up, you will notice that this is a really big switch statement. If you have Windows programming experience, you will recognize the “WM_SOMETHING” statements as Windows messages, and you will recognize this whole section of code as a message procedure for Windows messages. Don’t worry if you don’t know anything about this as a future tutorial will be going over windows message procedures in greater detail. For now, we are only interested in the case that involves our call to the dialog box. Here, we can see the entire case:

```
0043F76D  .  8BC6            MOV EAX,ESI
0043F76F  .  E8 A8470200     CALL WinRAR.00463F1C
0043F774  .v E9 5A140000     JMP WinRAR.00440BD3
0043F779  >  8B45 10         MOV EAX,DWORD PTR SS:[EBP+10]      Case 7B (WM_CONTEXTMENU) of switch 0043F0A4
0043F77C  .  3B05 604D4C00   CMP EAX,DWORD PTR DS:[4C4D60]
0043F782  .v 0F85 4B140000   JNZ WinRAR.00440BD3
0043F788  .  B8 604D4C00     MOV EAX,WinRAR.004C4D60
0043F78D  .  E8 40CD0100     CALL WinRAR.004571D0
0043F792  .v E9 3C140000     JMP WinRAR.00440BD3
0043F797  >  833D F0764C00 00 CMP DWORD PTR DS:[4C76F0],0       Case 113 (WM_TIMER) of switch 0043F0A4
0043F79E  .v 75 7D           JNZ SHORT WinRAR.0043F81D
0043F7A0  .  8D95 A4FAFFFF   LEA EDX,DWORD PTR SS:[EBP-55C]
0043F7A6  .  B8 A85A4C00     MOV EAX,WinRAR.004C5AA8
0043F7AB  .  33C9            XOR ECX,ECX
0043F7AD  .  E8 2E7A0100     CALL WinRAR.004571E0
0043F7B2  .  803D 95684A00 00 CMP BYTE PTR DS:[4A6895],0
0043F7B9  .v 75 62           JNZ SHORT WinRAR.0043F81D
0043F7BB  .  803D 8C854C00 00 CMP BYTE PTR DS:[4C858C],0
0043F7C2  .v 75 59           JNZ SHORT WinRAR.0043F81D
0043F7C4  .  803D 24204B00 00 CMP BYTE PTR DS:[4B2024],0
0043F7CB  .v 75 50           JNZ SHORT WinRAR.0043F81D
0043F7CD  .  8D85 A4FAFFFF   LEA EAX,DWORD PTR SS:[EBP-55C]
0043F7D3  .  E8 88C4FCFF     CALL WinRAR.0040BC60
0043F7D8  .  BA 6D694A00     MOV EDX,WinRAR.004A696D            ASCII "rarkey"
0043F7DD  .  B9 06000000     MOV ECX,6
0043F7E2  .  E8 81F9FCFF     CALL WinRAR.0040F168
0043F7E7  .  85C0            TEST EAX,EAX
0043F7E9  .v 74 32           JE SHORT WinRAR.0043F81D
0043F7EB  .  A1 2C804C00     MOV EAX,DWORD PTR DS:[4C802C]
0043F7F0  .  83F8 28         CMP EAX,28
0043F7F3  .v 7F 04           JG SHORT WinRAR.0043F7F9
0043F7F5  .  85C0            TEST EAX,EAX
0043F7F7  .v 7D 24           JGE SHORT WinRAR.0043F81D
0043F7F9  >  C605 95684A00 01 MOV BYTE PTR DS:[4A6895],1
0043F800  .  6A 00           PUSH 0                             lParam = NULL
0043F802  .  68 302E4800     PUSH WinRAR.00482E30               DlgProc = WinRAR.00482E30
0043F807  .  FF35 4C4D4C00   PUSH DWORD PTR DS:[4C4D4C]         hOwner = 002F04C0 ('WinRAR - WinRAR (evaluation c...',class
0043F80D  .  68 74694A00     PUSH WinRAR.004A6974               pTemplate = "REMINDER"
0043F812  .  FF35 48204B00   PUSH DWORD PTR DS:[4B2048]         hInst = 00400000
0043F818  .  E8 171C0600     CALL <JMP.&USER32.DialogBoxParamA> DialogBoxParamA
0043F81D  >  803D 94684A00 00 CMP BYTE PTR DS:[4A6894],0
0043F824  .v 74 1C           JE SHORT WinRAR.0043F842
0043F826  .  803D F4764C00 00 CMP BYTE PTR DS:[4C76F4],0
0043F82D  .v 75 13           JNZ SHORT WinRAR.0043F842
0043F82F  .  C605 94684A00 00 MOV BYTE PTR DS:[4A6894],0
0043F836  .  6A 00           PUSH 0                             lParam = 0
0043F838  .  6A 00           PUSH 0                             wParam = 0
0043F83A  .  6A 10           PUSH 10                            Message = WM_CLOSE
0043F83C  .  56              PUSH ESI                           hWnd = 2F04C0
0043F83D  .  E8 D81D0600     CALL <JMP.&USER32.PostMessageA>    PostMessageA
```

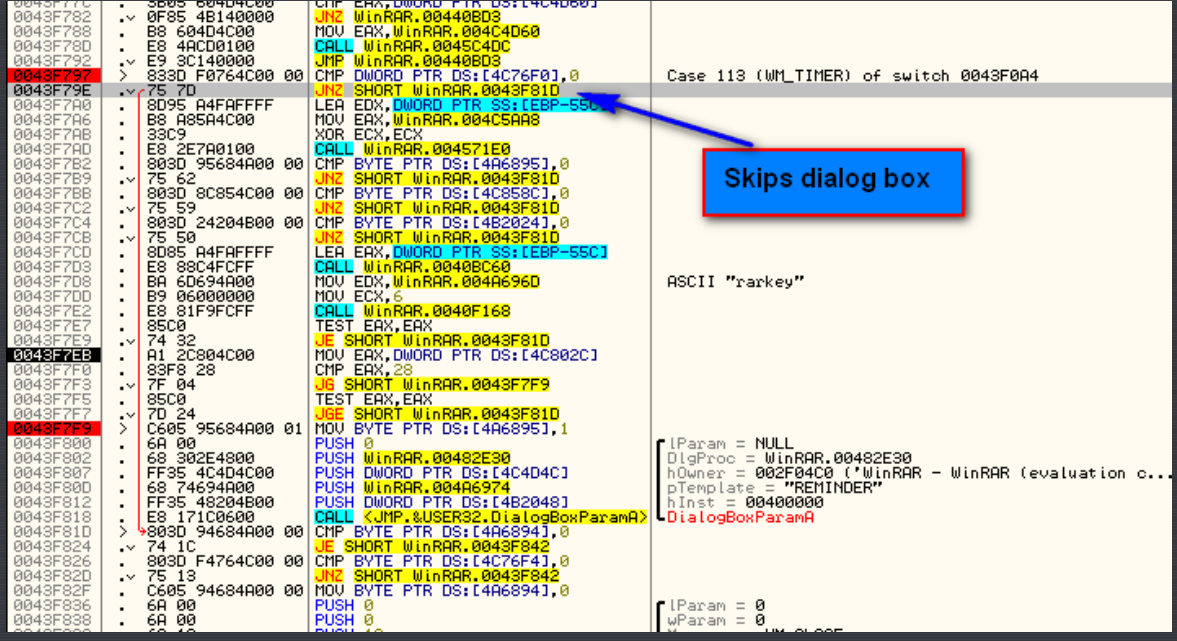
You will notice that it is in the section that handles WM_TIMER messages. This is very telling. Why would our dialog box be in a message handler for when a timer goes off? We will see shortly…

Also notice that after the dialog is called there are several conditional jumps and compares:

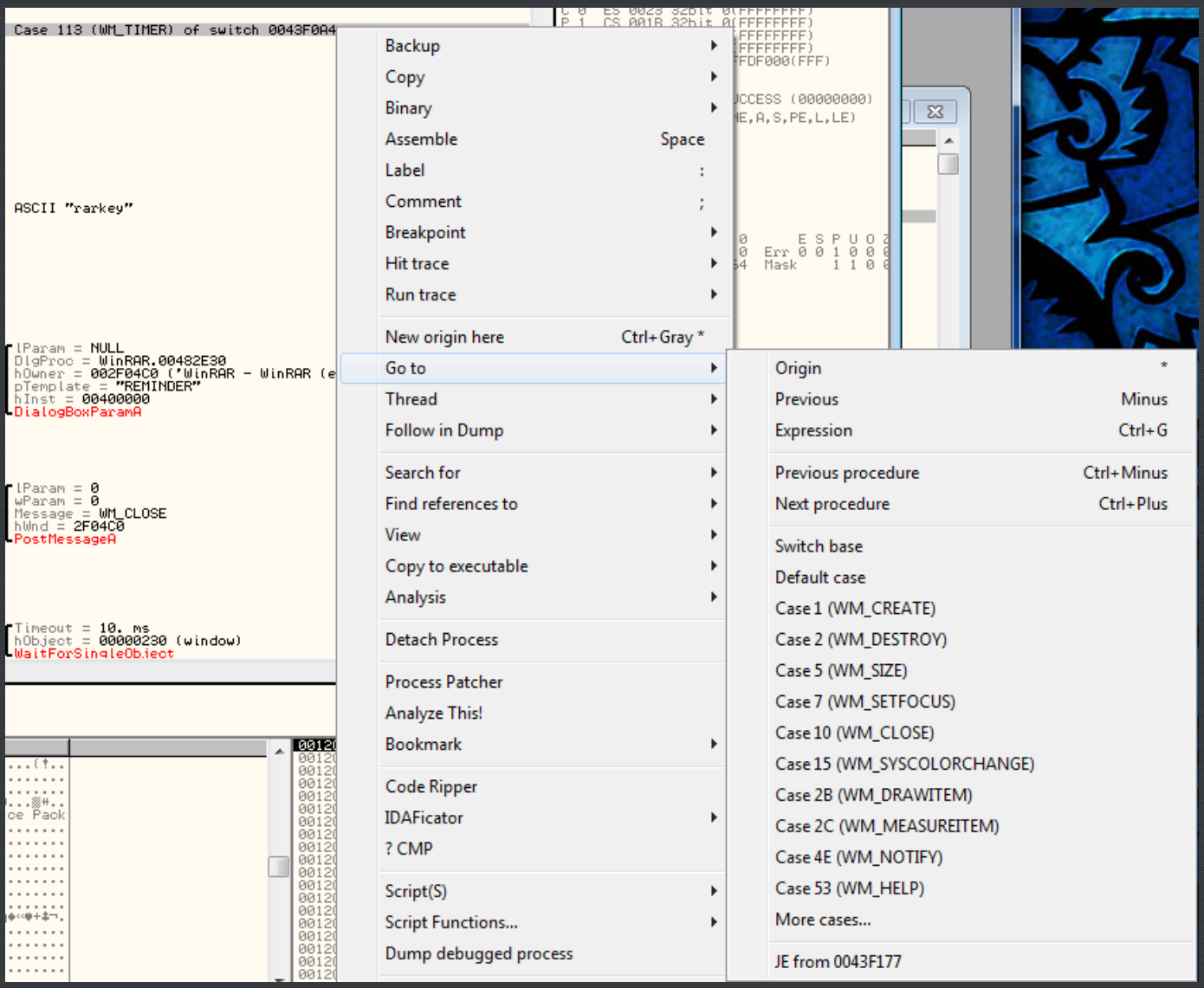
```
0043F7F7  .v 7D 24           JGE SHORT WinRAR.0043F81D
0043F7F9  >  C605 95684A00 01 MOV BYTE PTR DS:[4A6895],1
0043F800  .  6A 00           PUSH 0                             lParam = NULL
0043F802  .  68 302E4800     PUSH WinRAR.00482E30               DlgProc = WinRAR.00482E30
0043F807  .  FF35 4C4D4C00   PUSH DWORD PTR DS:[4C4D4C]         hOwner = 002F04C0 ('WinRAR - WinRAR (evaluat
0043F80D  .  68 74694A00     PUSH WinRAR.004A6974               pTemplate = "REMINDER"
0043F812  .  FF35 48204B00   PUSH DWORD PTR DS:[4B2048]         hInst = 00400000
0043F818  .  E8 171C0600     CALL <JMP.&USER32.DialogBoxParamA> DialogBoxParamA
0043F81D  >  803D 94684A00 00 CMP BYTE PTR DS:[4A6894],0
0043F824  .v 74 1C           JE SHORT WinRAR.0043F842
0043F826  .  803D F4764C00 00 CMP BYTE PTR DS:[4C76F4],0
0043F82D  .v 75 13           JNZ SHORT WinRAR.0043F842
0043F82F  .  C605 94684A00 00 MOV BYTE PTR DS:[4A6894],0
0043F836  .  6A 00           PUSH 0                             lParam = 0
0043F838  .  6A 00           PUSH 0                             wParam = 0
0043F83A  .  6A 10           PUSH 10                            Message = WM_CLOSE
0043F83C  .  56              PUSH ESI                           hWnd = 2F04C0
0043F83D  .  E8 D81D0600     CALL <JMP.&USER32.PostMessageA>    PostMessageA
0043F842  >  833D F0764C00 00 CMP DWORD PTR DS:[4C76F0],0
0043F849  .v 75 2D           JNZ SHORT WinRAR.0043F878
0043F84B  .  833D E8764C00 00 CMP DWORD PTR DS:[4C76E8],0
```

These jump the execution of the app based on what we clicked in the dialog box; if you clicked “Close” it will jump to the code to close the window etc.

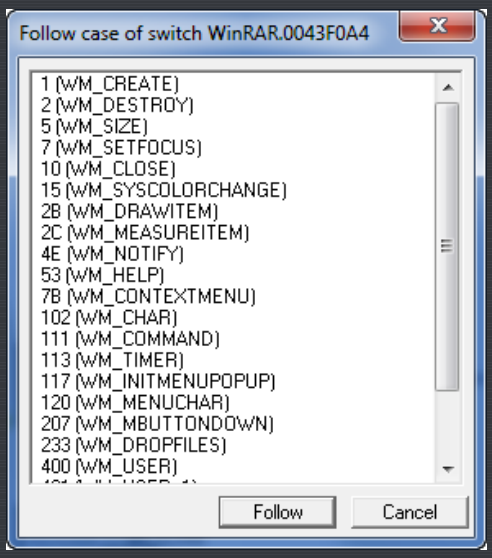
Scrolling back up to the beginning of the switch/case section we see that there is an initial compare and jump:



The jump here jumps over the call to open the nag dialog (along with a lot of other code). Let's see what this initial compare/jump is. Right-click on the line that has the "Case 113 (WM_TIMER)" in it and select "Goto":



You will see in the drop down window that Olly shows us several of the cases that can be handled by this switch statement. Clicking on “More cases…” opens a dialog showing us all of them:

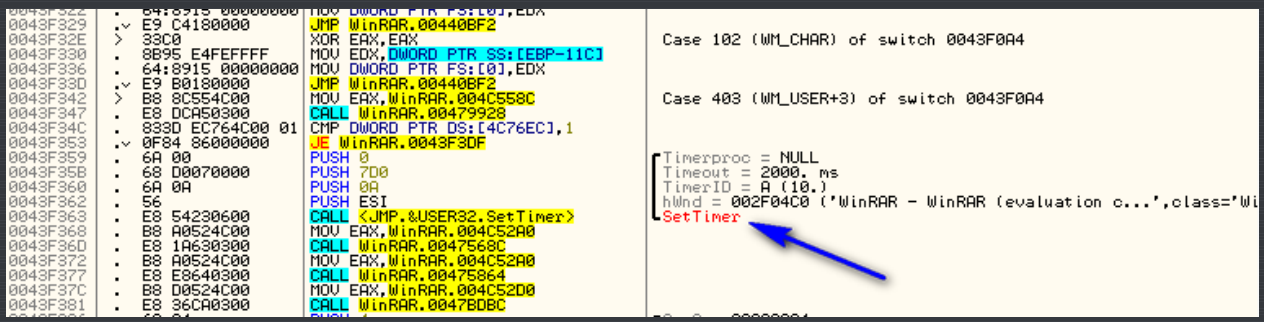


Clicking on a couple of these and selecting “Follow” will take you to the code that handles that case. You will notice that at the beginning of all of them is a compare/jump. What this means is that the assembly language handles this switch/case statement as a huge if/then statement. It would look something like this (in psuedo-code):

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

So, at the beginning of each case, we check if this is the case for the particular message coming thru, and if it’s not we jump to the next compare. If it is, we skip the jump and fall through to the code that handles the message.

Now, because our specific case is a WM_TIMER messages, we can learn (by looking up the message WM_TIMER on Google) that this message is a handler for when the timer goes off. That means that somewhere this timer must have been started. Scrolling up (quite a bit) we see the culprit:

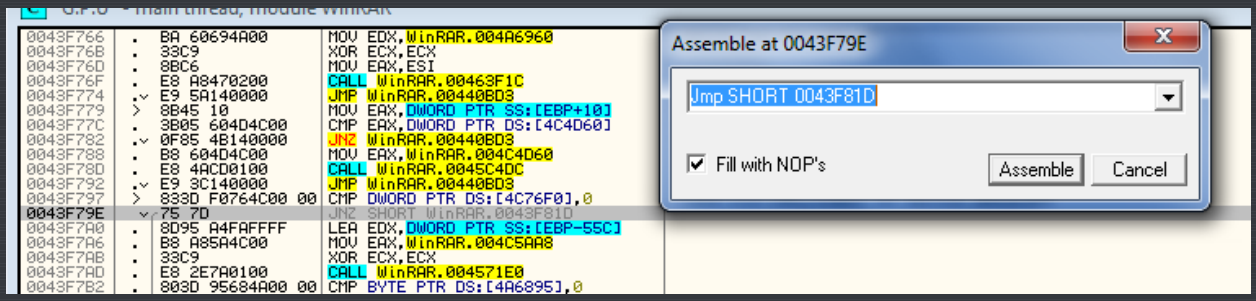


So now we can guess how we can override this nag…

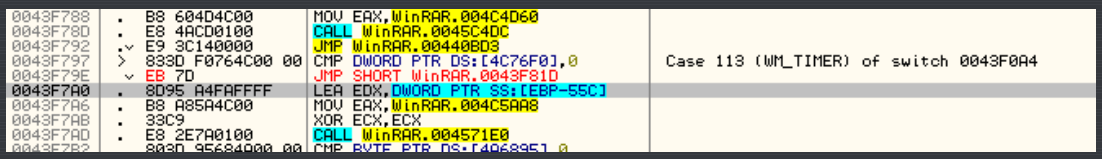
Patching the App

The easiest way is to make this message handler not do anything when the timer goes off. And the easiest way to do that is to make sure we jump over this case every time. So go to the beginning of this case (113

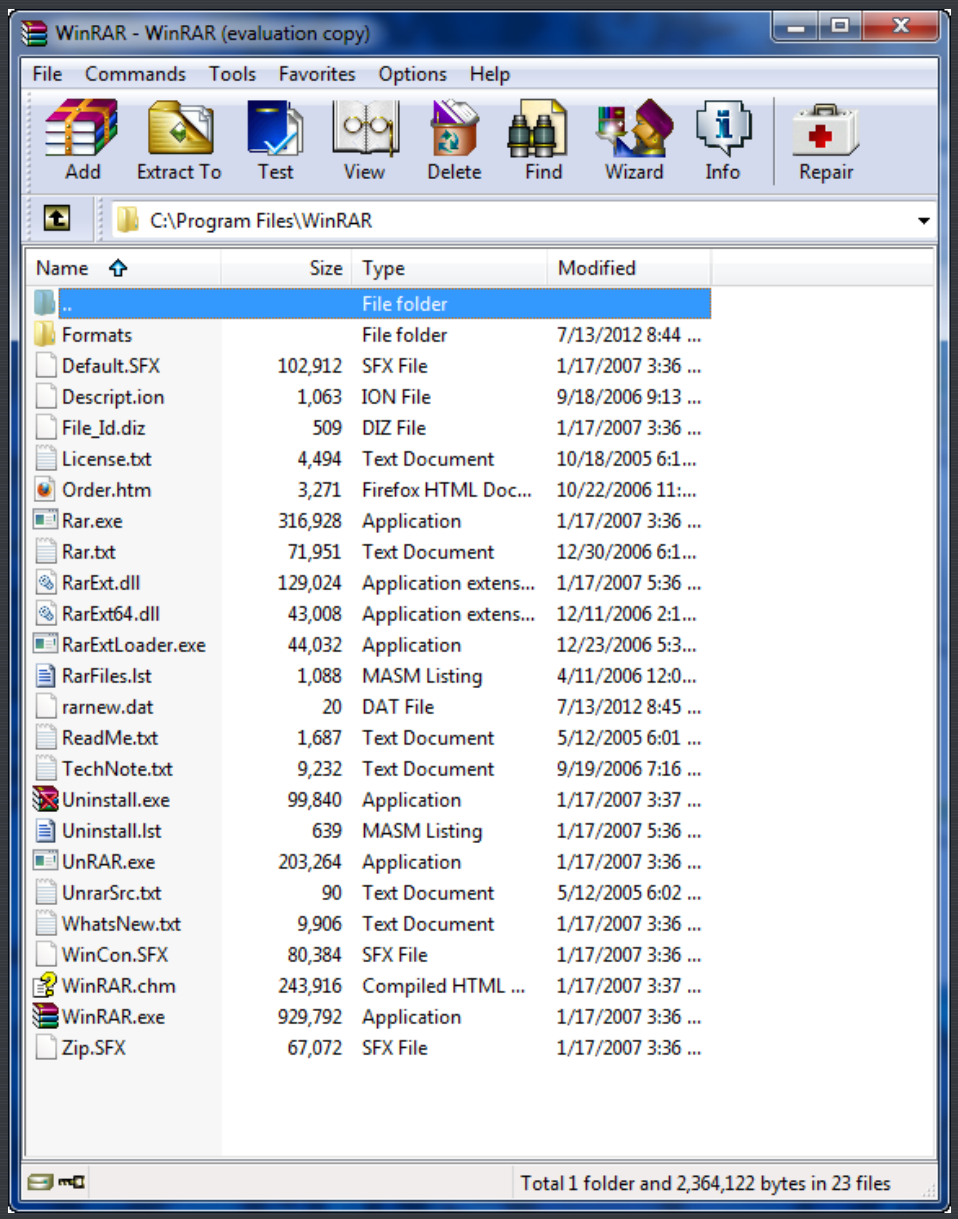
– WM_TIMER) where it checks if this is the correct case and change it to always jump:



and here' s what the patch looks like:



Now, whenever this message procedure gets a message that the timer has gone off, it will simply ignore it 😀 . Run the app and you will notice after a while that the nag does not show up anymore:

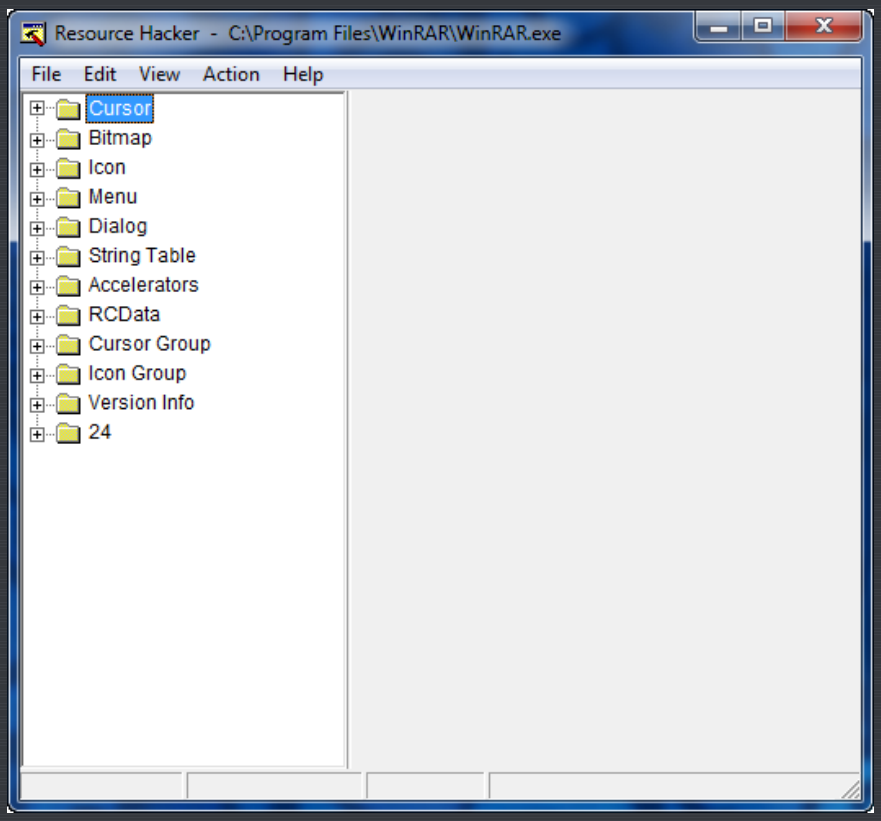


The program still says “evaluation” in the title, but we will come back to this in the tutorial on windows messages (as it’s a little more complicated to change this- go ahead and try. It’s the best way to learn!) But for now, even though it says “evaluation” it works fine and will never expire. Well, that’s not exactly true; it will expire but it won’t do anything about it . Make sure you save the patch (as we did in several previous tutorials) to keep the changes.

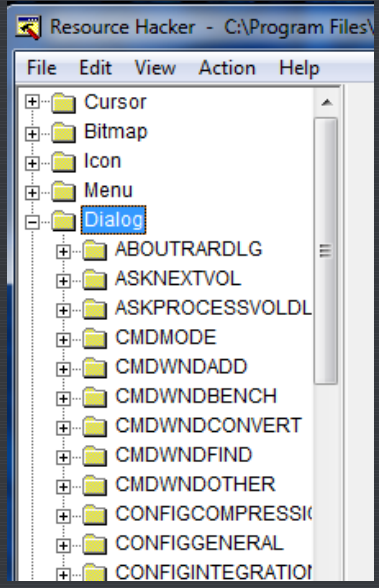
The Second Way

Now, I am going to show you another method we can use (instead of the call stack) to find our dialog box section. Download Resource Hacker if you don’t have it already. You can get it on the tools page. Resource Hacker allows you to view and manipulate the resources inside a PE file. We will get more into resources when we discuss the make-up of the PE file, but for now just know that any resource that the app uses (buttons, dialog boxes, bitmaps, icons, text strings) are stored in a separate section of the file, separate from the code that is. Really, the best way to see what I mean is to open Resource Hacker, load a couple of program in it, and just look around.

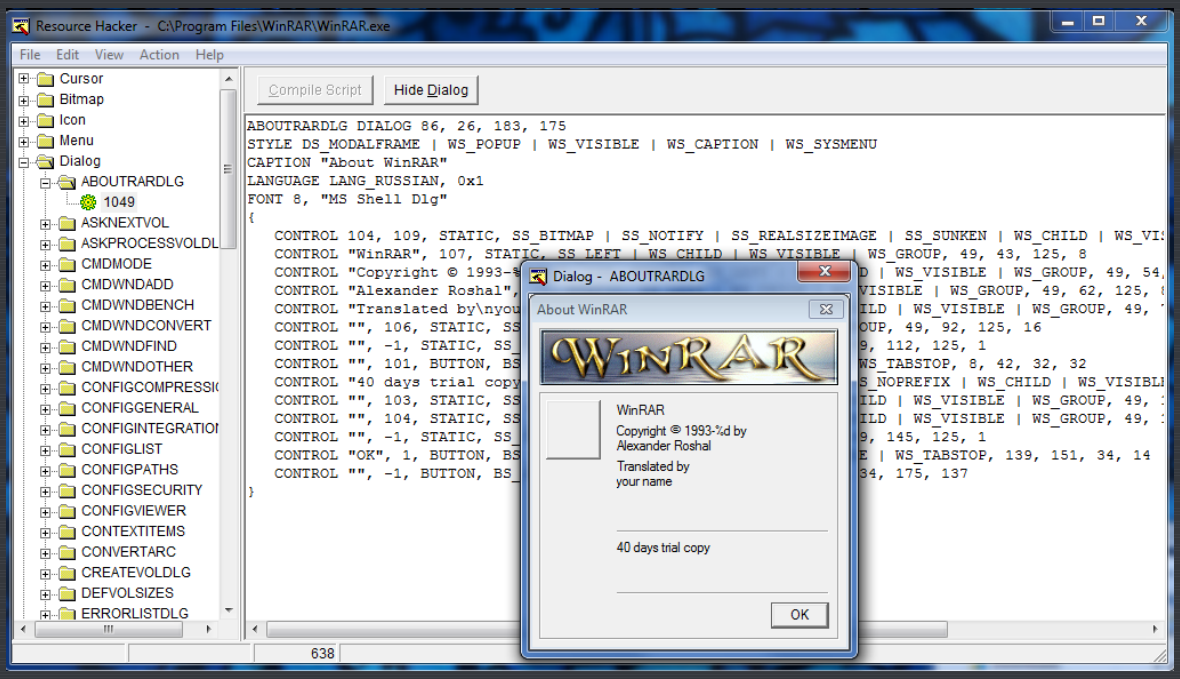
So let’s do just that…Open Resource Hacker and load in our app:



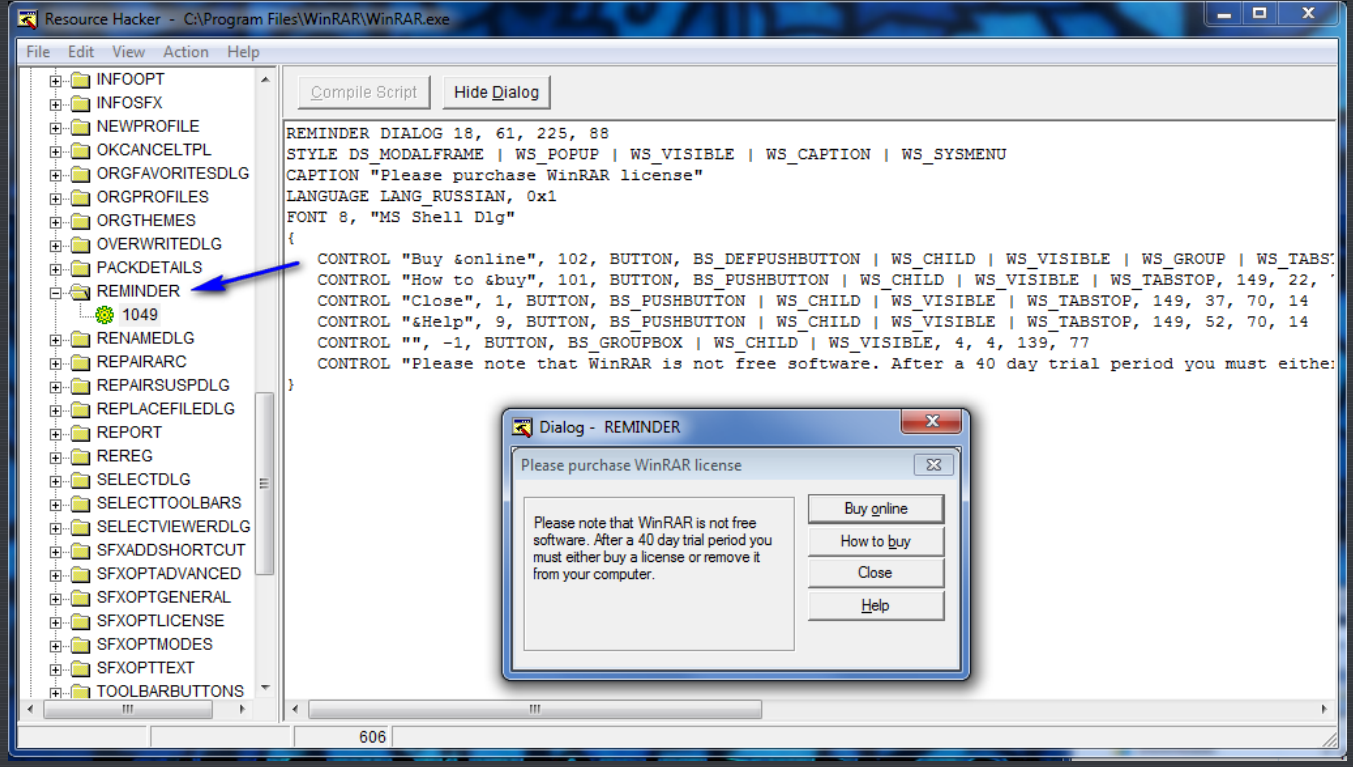
The tree on the left shows you the various resources that are in this app. You can see it contains bitmaps, icons, menus, and most importantly, dialog boxes:



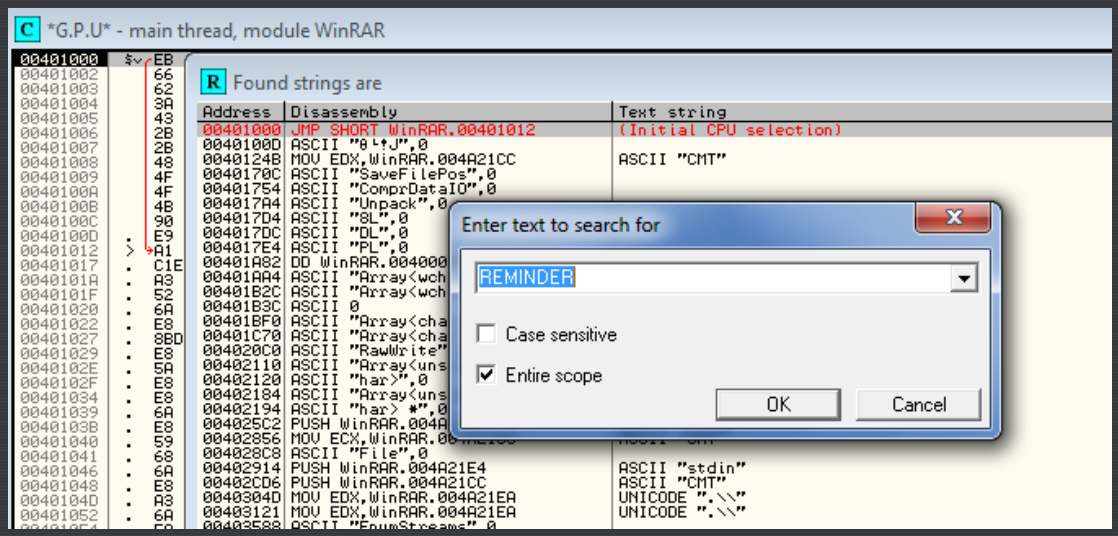
There are a lot of dialog boxes in this app. Go ahead and click on the first one, ABOUTRARDIALOG:



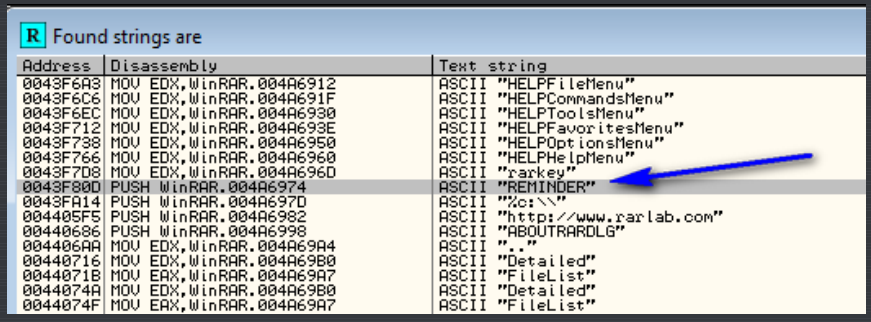
Resource Hacker shows us the pertinent data to this dialog, including the caption (what appears in the title of the window), buttons associated with this dialog, and it's various settings. It also opens a window showing us exactly what the dialog will look like, in this case the About dialog. After clicking through a bunch of them, you will come across the one we want:



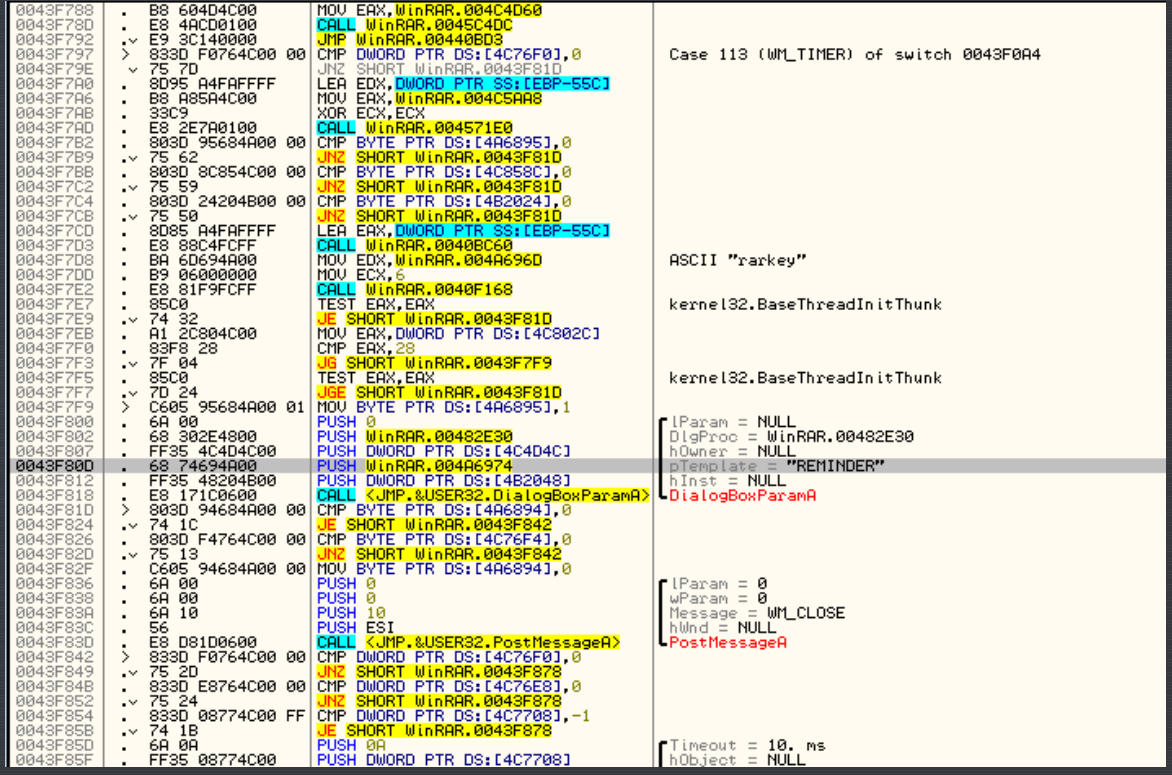
Look familiar? Notice that the name of this dialog is 'REMINDER'. Sometimes Windows uses names to reference a dialog box, and other times an ID. In this case it uses the name 'REMINDER. Now we know all we need to- load up the app in Olly and go to "search for strings". Let's search for 'REMINDER':



and we see it in the list:



Double-clicking on it brings us to the same section we came to by using the call stack:



In fact, you will see that one of the arguments passed to DialogBoxParamA is the name, ‘REMINDER’. If the resource was identified by an id instead of a name, we could find it by right-clicking the disassembly window and selecting “Search for”->”Command”. Then enter into the box ‘PUSH xx” where xx is the ID (in hex) of the resource. This will take you to the call to the dialog box as well.

One More Thing

If you look around at the numerous tutorials on cracking this binary, you will see that one of the methods is to simply delete the dialog box in Resource Hacker. This does work in this case, and you will no longer see the nag, though this won’t work in every case- it all depends on how the program handles a missing resource. Given the simplicity of this technique, it’s always worth trying.

-Till next time

R4ndom

ps. Don’t forget to change your clock back 😀