R4ndom's Tutorial #14: NAGS (And I don't Mean Your Mother)

by R4ndom on Jul.16, 2012, under Beginner, Reverse Engineering, Tutorials

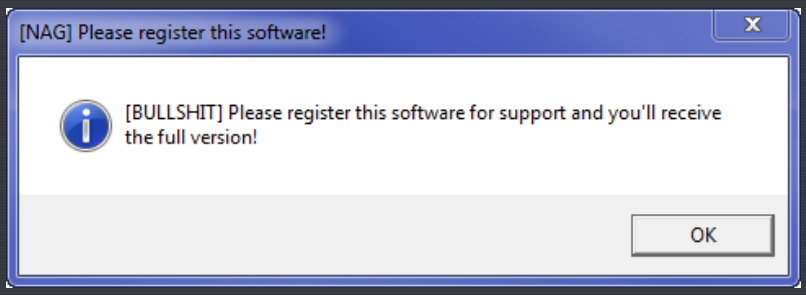
Introduction

Nags, or nag screens, are generally message boxes that pop up to remind you that your trial is ending, you need to register, a reminder about visiting the website… basically anything that's nagging and not necessary (like most bosses). Many Freeware programs come free because they're full of nags (ads, time-trials, re-directs). Commercial software also includes them often, reminding you "you have 18 days left to try this product." etc. Getting rid of nags is a central theme in reverse engineering, and sometimes provides it's own set of challenges. In this tutorial we will be going over two apps that have nags. We will then bypass them so they no longer show, and then patch them so they won't ever come back.

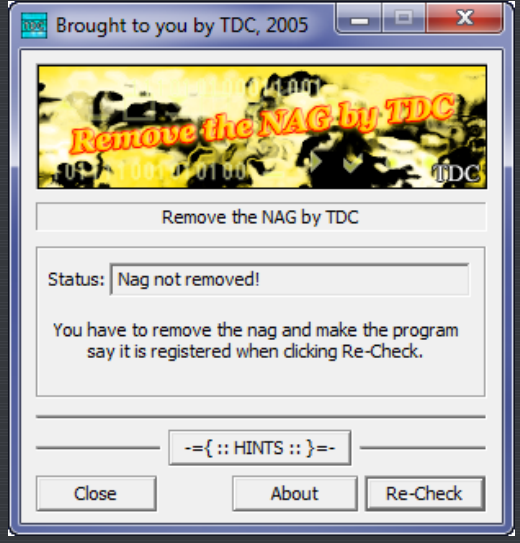
I will also be introducing a new plugin for Olly called IDAFicator. It has many features and settings. you can download the plugin from the tools page. Because there are so many features, I am also including a tutorial by the author of IDAFicator in the download for this tutorial. I highly recommend watching it as there are a lot of very cool features to this plugin.

The First App

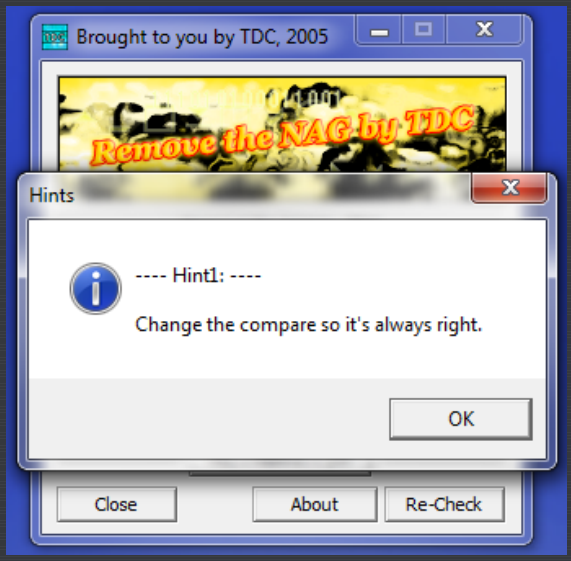
The first binary we will look at is Nag1.exe. Running the program immediately pops up the nag:



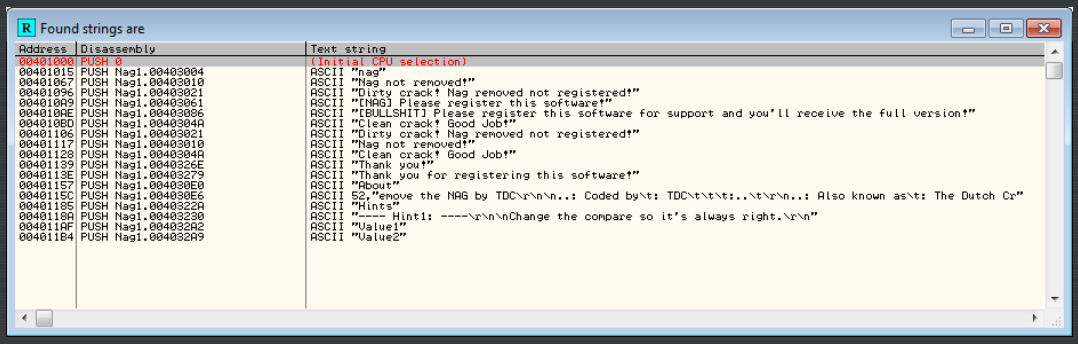
You can obviously tell this was made by a cracker . Anyway, after clicking OK you get the main screen:



Notice it says “Nag not removed!. I, of course, could not help clicking the “Hints” button and was rewarded with some very detailed information:

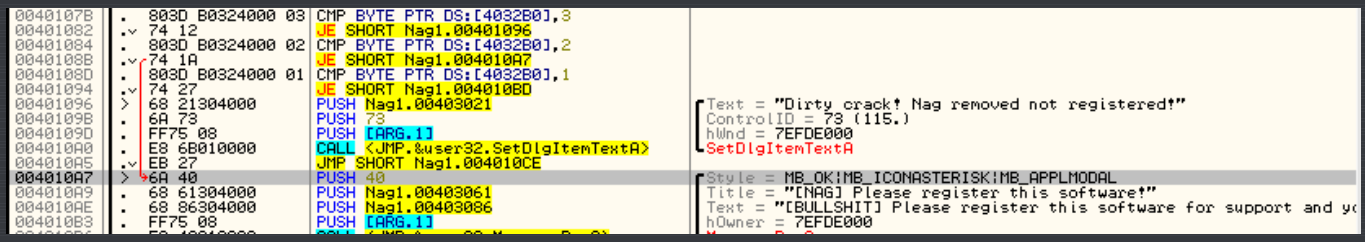


Gee, thanks. Load the app in Olly and let’s try old reliable: search for strings:

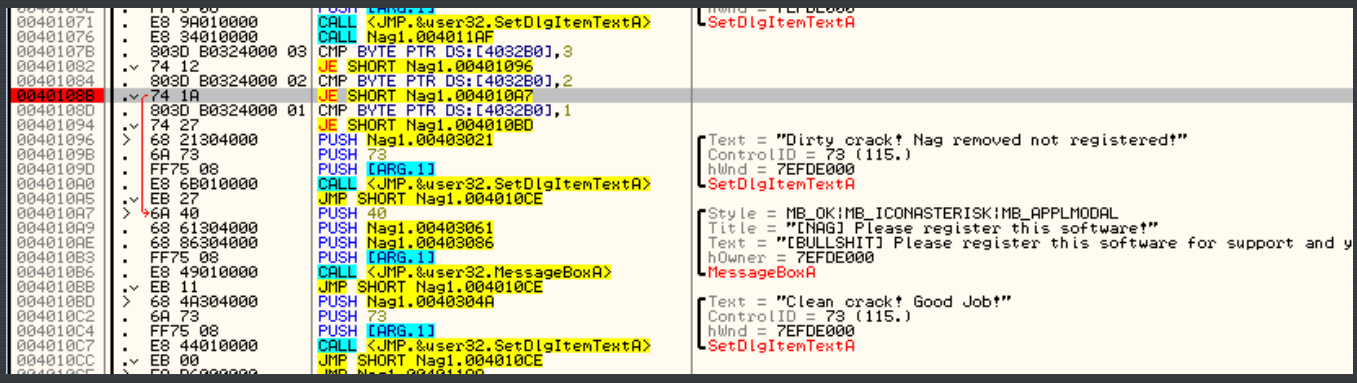


and we’re in luck. You can see the text for the nag screen at address 4010AE. Let’s dbl-click that and jump to where the nag is created:

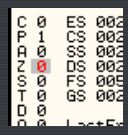
Hmm, an interesting string above it, but let's ignore that for now. Let's click on the first line of the MessageBoxA code at address 4010A7 to see where it's called from:



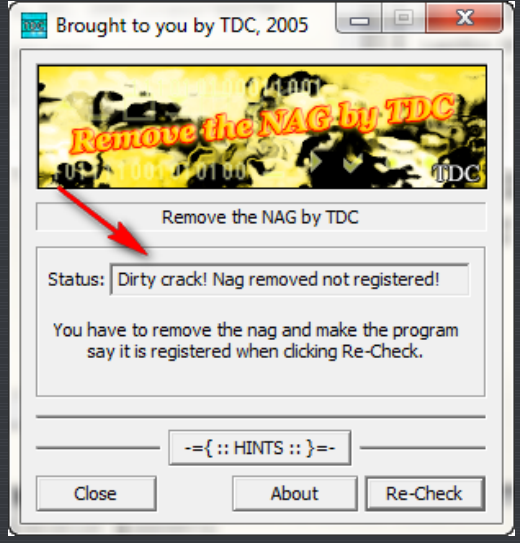
And we can see that it is called from a JE instruction at address 40108B, right after a compare. Well, we certainly recognize this scenario 😃. Let's place a BP on that JE instruction:



And run the app. We break at our BP and see that we are going to jump to the nag screen instructions, so let's make it not jump:

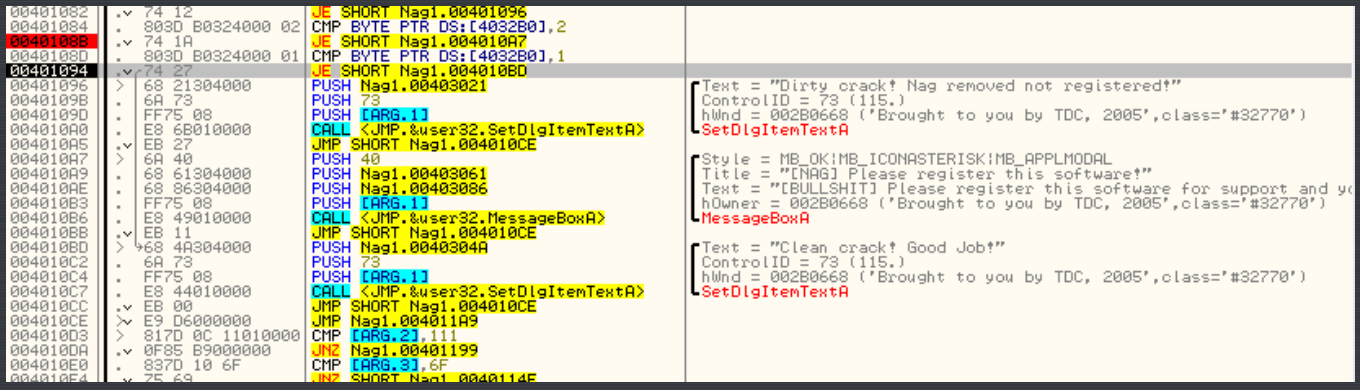


and then run the app:

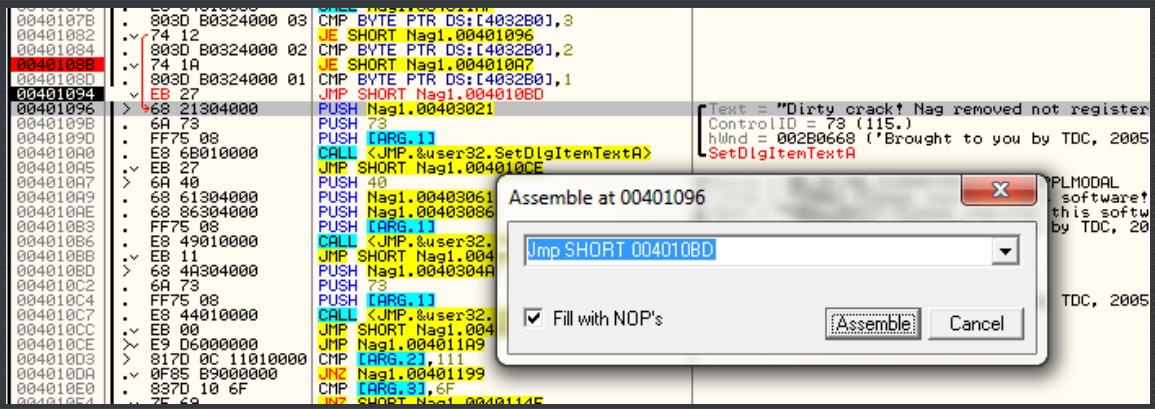


So, that's what the "Dirty Crack" thing was all about- apparently we didn't patch enough. Let's restart the app and Olly will pause at our BP. Zero out the zero flag again:

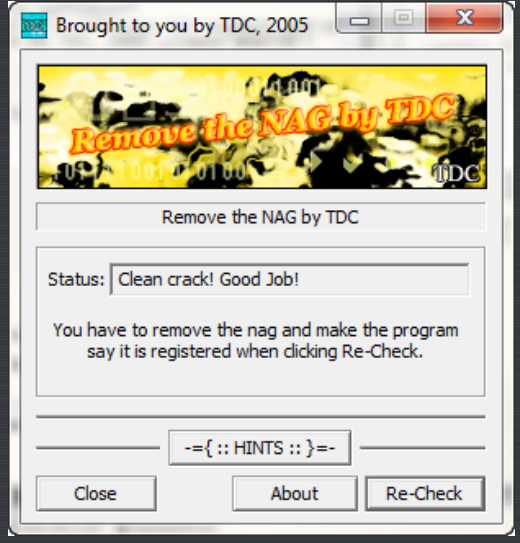
and let's step twice to the next jump. As you could probably guess by now, this jump SHOULD jump to our good boy, but instead falls through to our bad boy:



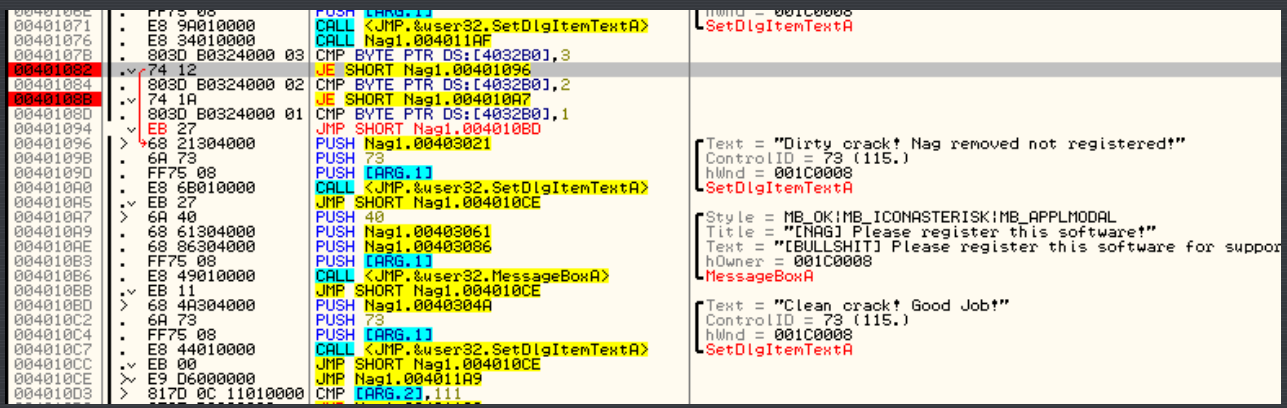
Let's just patch that to always jump:



and when we run the app we see we were right:



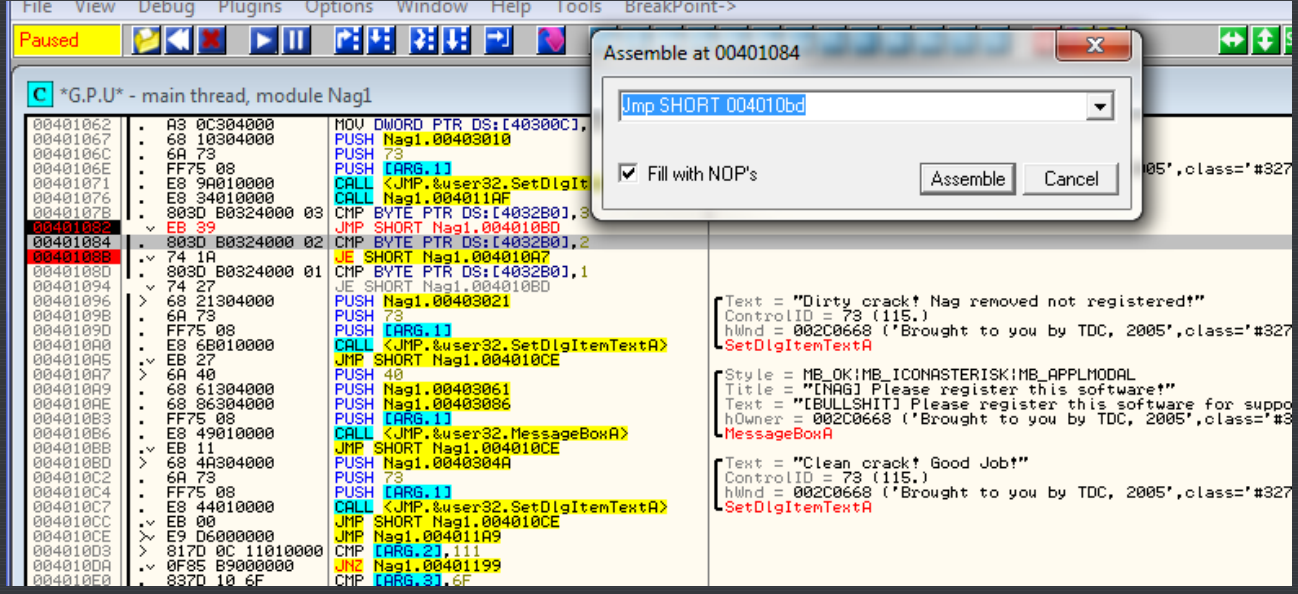
We can obviously patch this program by keeping our current patch and going back to address 40108B (where we originally zeroed out the zero flag) and patch it to never jump. Saving these two patches will work fine. But I also want to show you (as I have mentioned before, and if I haven't I should have) that there are ALWAYS other ways to patch an app, usually many. Restart the app and scroll to our BP:



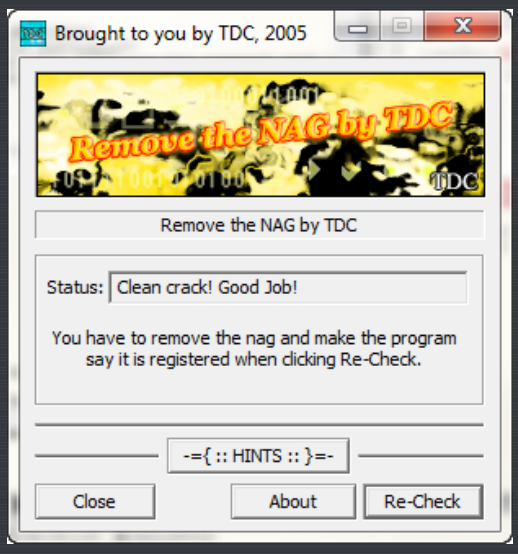
Notice that this collection of instructions is something like this (in a high-level language):

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

We know that since the nag screen is displayed by default, the contents of memory address 4032B0 will always equal 2, as that's the jump that is taken. Well, what if we just bypassed this whole if/then clause and immediately jumped to the good boy? So if we replace the very first jump to just jump to the good boy, we would only need one patch. Try it:

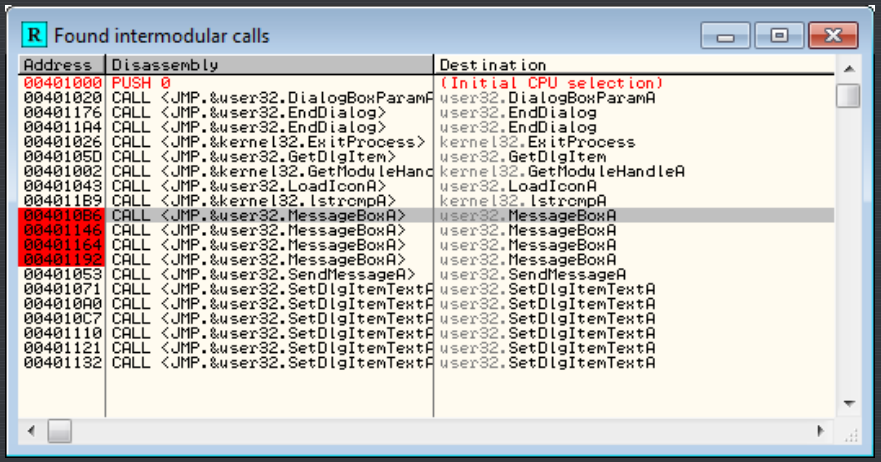


Now run the app:

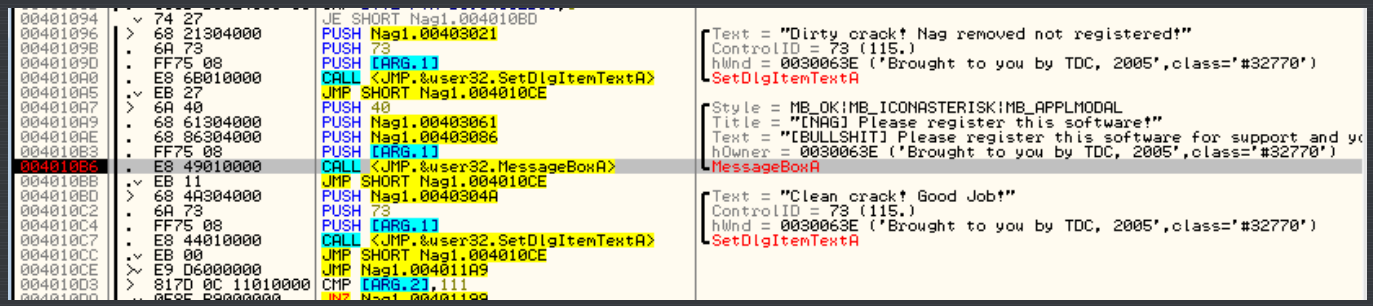


And you can see that it accomplished the same thing. Another, even more elegant solution may be to think, “If the contents of 4032B0 are always equal to 2, and to hit the good boy message it needs to be 1, why not just place a 1 in this memory location and we’ll always hit the good boy?” You should try this. Restart the app, click on the dump window, go to address 4032B0 and binary edit it to be a one. Did it work?

Another thing to keep in mind is there are always other ways to find the code section we are looking for. For example, if we couldn’t use strings in this example, we could do a search for Intermodular calls:



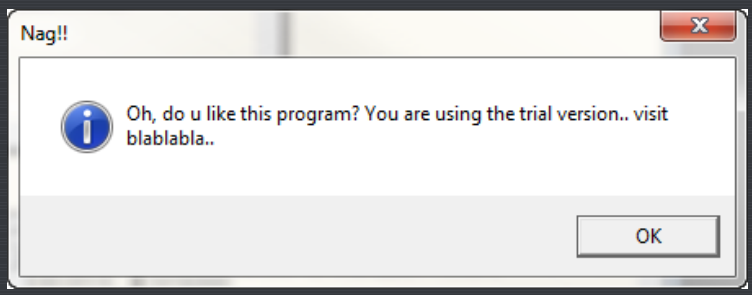
Notice that there are 4 calls to MessageBoxA. Right click one of them and choose “Place a breakpoint on every call to MessageBoxA”. When you run the app, before anything is displayed we stop at the following line of code:



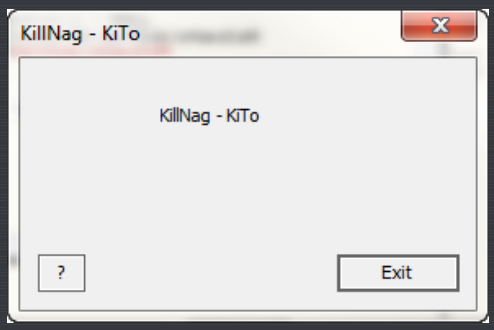
Look familiar? It is the nag messagebox !! So always keep in mind that there are more than one way to accomplish something. Soon, we will also be learning some other techniques that can be used (like windows message handlers) that will give you an even bigger bag of tricks.

The Second App

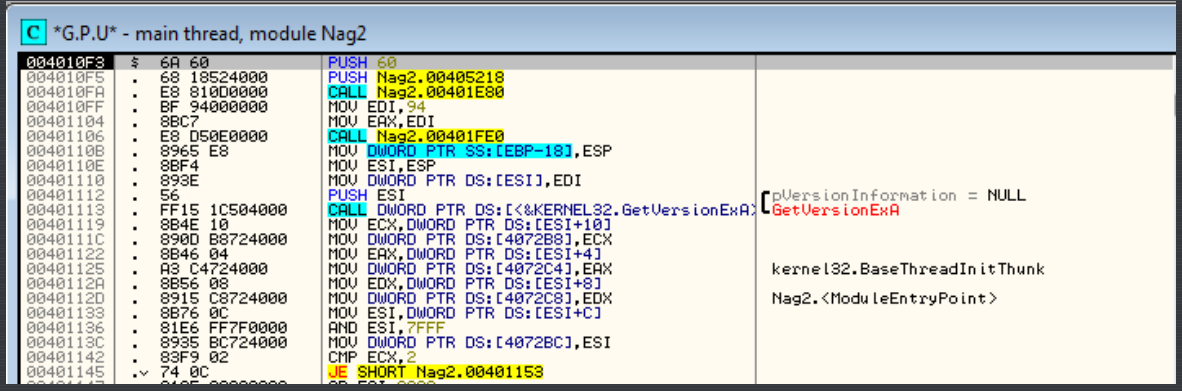
Now let’s take a look at Nag2.exe. It is similar but we will solve it in different ways. When we start the app, we get the expected nag:



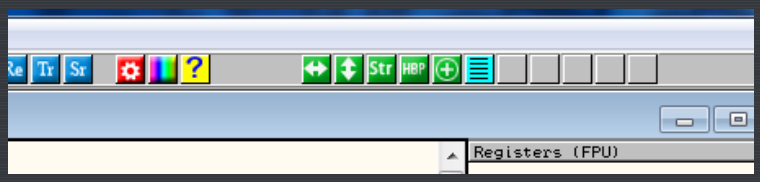
and after clicking OK we get the main screen:



At this point I closed the app and loaded it into Olly:

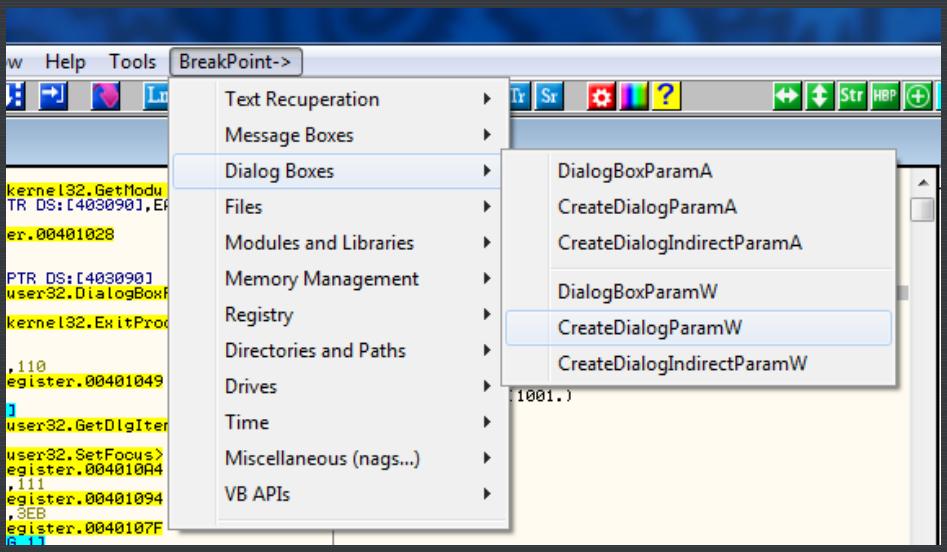


First things first, let's see if there are any strings. One thing I wanted to point out here is the plugin IDAFicator. Among many additions, it provides a group of buttons along the top that making searching for strings a lot easier. When clicking the strings button (Str), it shows both ASCII and Unicode and also brings the cursor up to the top automatically so you don't have to scroll to the top your self. Here's what the buttons look like:



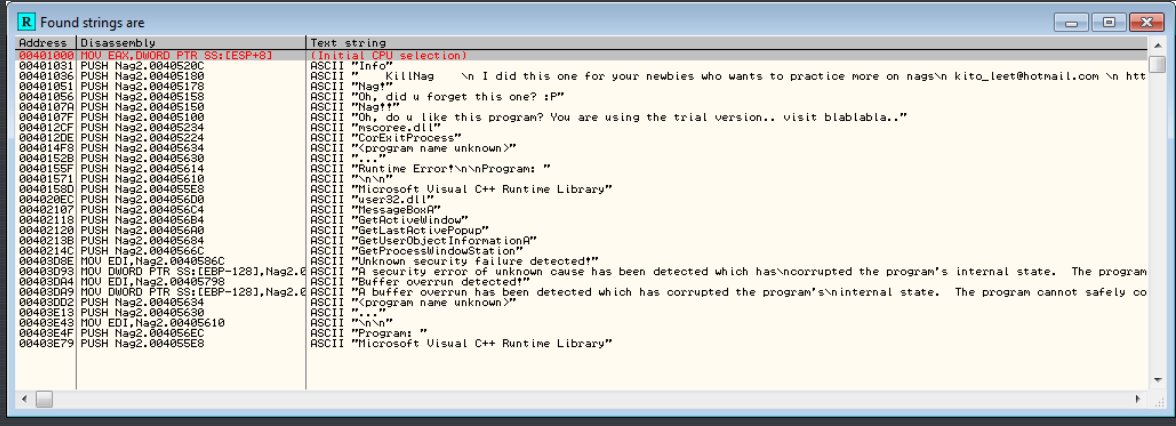
The first button (the left and right arrow) take you forward and back. For example, if you click on a call, then press enter to go to that call, clicking the first icon will take you back to the call instruction. Right-clicking takes you forward. The second button will attempt to find the beginning of the current function, while right-clicking will attempt to find the end. The next is the strings button. Next is the Hardware BreakPoints button. It brings up a nice dialog that shows you all of your hardware breakpoints. Very handy. The target icon opens the folder where your app resides and the list icon brings up a dialog to enter multiple lines of assembly code, used for if you are changing a substantial part of the exe.

You will also notice a new menu item called “Breakpoint->” that opens a drop down of many used API calls so you can set breakpoints on them automatically:

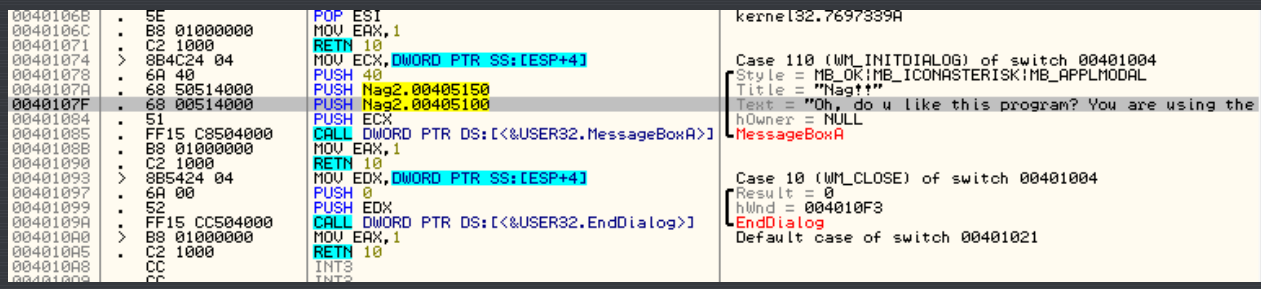


Lastly, there is a context menu item added that allows you to restore hidden bytes, which we will get to in a future tutorial.

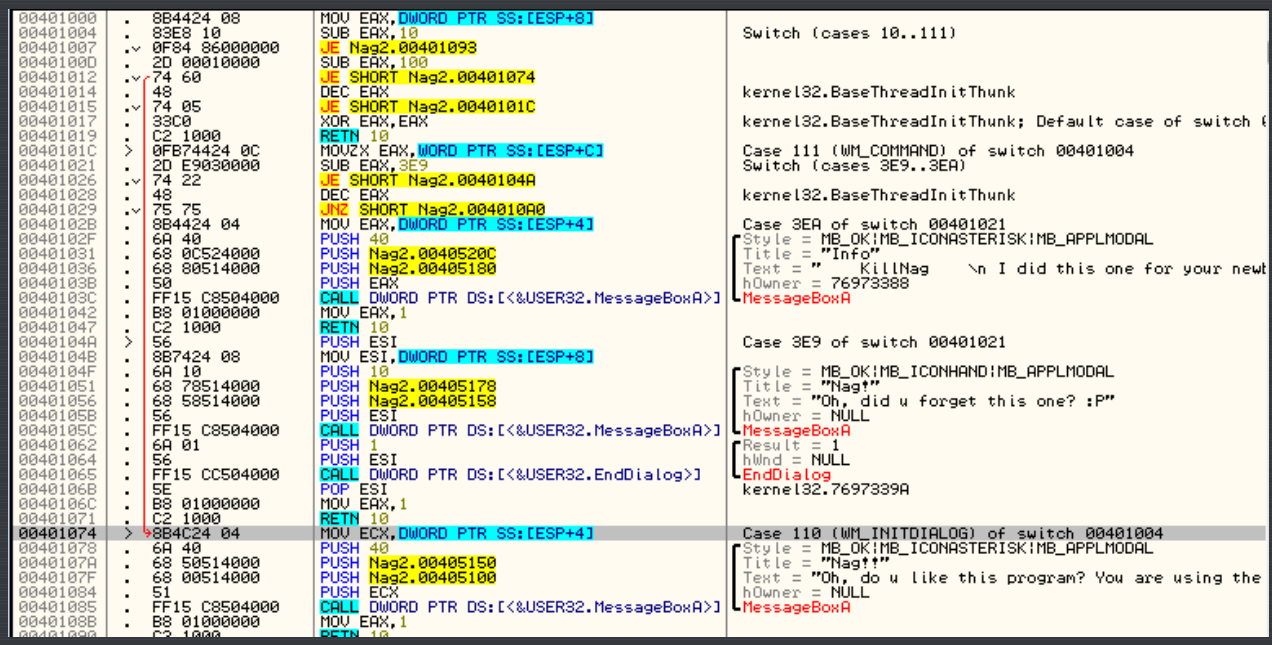
So go ahead and click in the strings button (“Str”) on the new button bar:



and on the seventh line down we see our nag's text, so let's double-click:



To see the nag's method. It is a self contained method (there is a RETN above and below it) so we know it is called from somewhere. Click on the first line of it at address 401074 to see where it's called from:

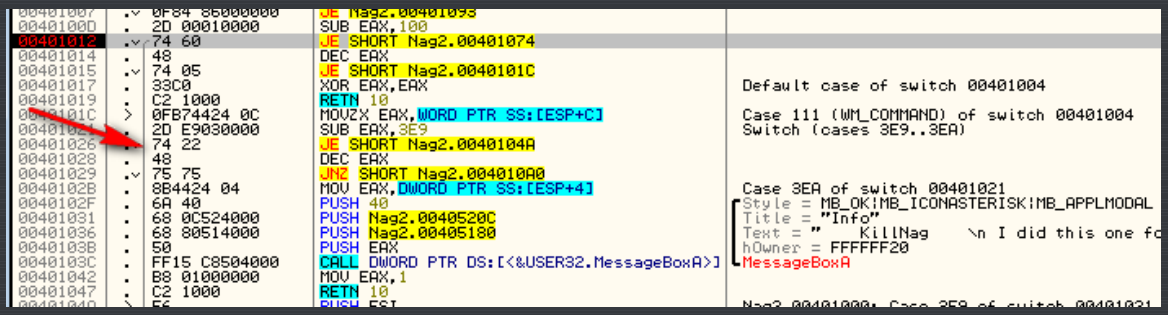


and we can see it's called from 401012, a JE instruction. Let's put a breakpoint on this and run the app:

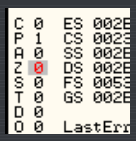
and we break on that JE instruction. Notice that it is not calling our nag screen. The reason for this is we happen to be in the middle of Window's message handler. I will be going into depth on message handlers in another tutorial, but for now just know that all GUI windows programs have a message handler and Windows sends various message through it. Depending on which message comes thru (and whether we

wish to do anything out of the ordinary when a certain action is achieved) we can add our own code to override Window's normal routines. For example, when we click the 'X' to close a window, Windows will send a message through the message handler that says "hey, the user wants to close the window." We can either let the message go through, in which case Windows will handle it and close the window, or we can 'trap' this message and do what we want (maybe pop up a dialog that says "You have not saved, are you sure you want to quit?".

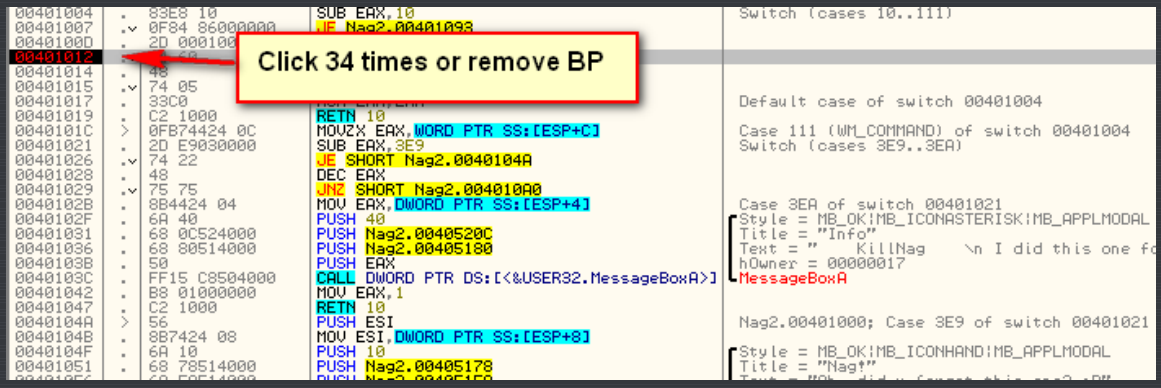
Our breakpoint happens to be right in the middle of this, so the first message that has come through does not match the message that this app expects to override in order to show the nag:



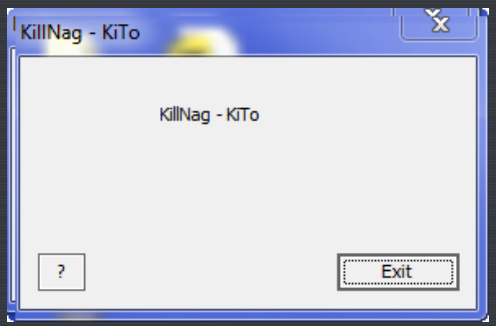
Go ahead and hit F9 to run the app and we will stop at the same BP, but this time, the jump will be taken, showing our nag. Let's tell Olly to not show the nag:



Now, if we leave this breakpoint, 34 more messages will be sent through this message handler. You can either keep the BP in place and click run 34 times (in which case, at some point you will see the window appear, the buttons being drawn etc) or you can remove the BP and just hit run once. In this case, the call is not made to the nag again so removing the BP and running it is fine:



We then have our main screen:

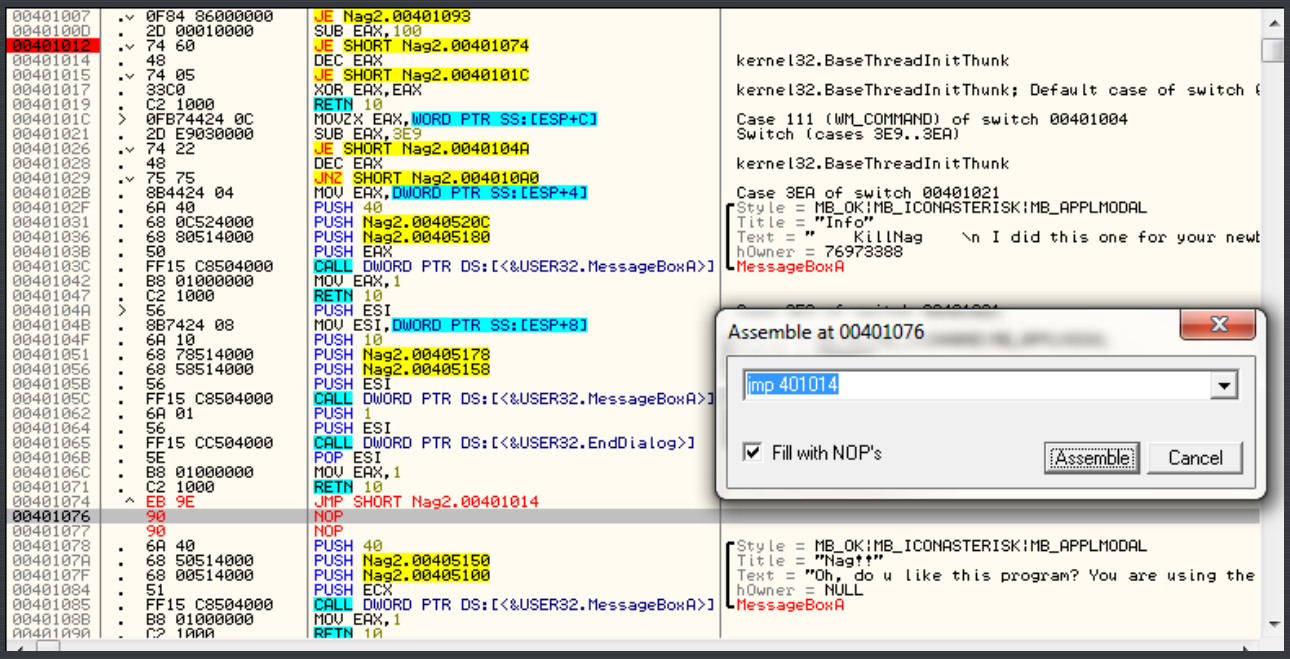


Notice that the initial nag is now gone.

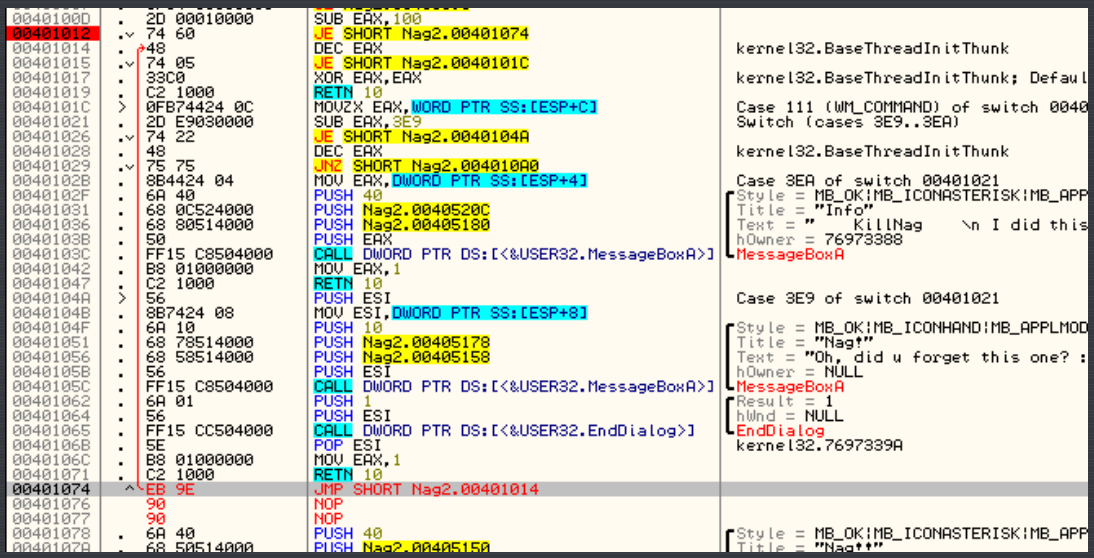
Patching the App

Normally what we would do is patch the JE instruction that jumps to the nag with a NOP so that it never

jumps, but I wanted to show you another way this patch can be accomplished. We know that when the correct message comes through the message handler (in this case the second message) our nag code will be called. Well, what if we allowed the jump to the nag, but changed the nag to just jump right back again?

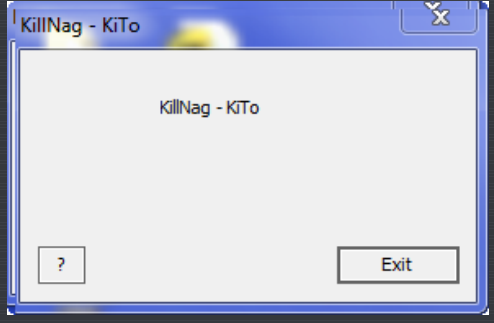


Here, the jump will be made to the nag instructions at 401074, but then we will immediately jump back to the line after the initial jump (401014). Basically, our program will jump, then jump right back to the next line:

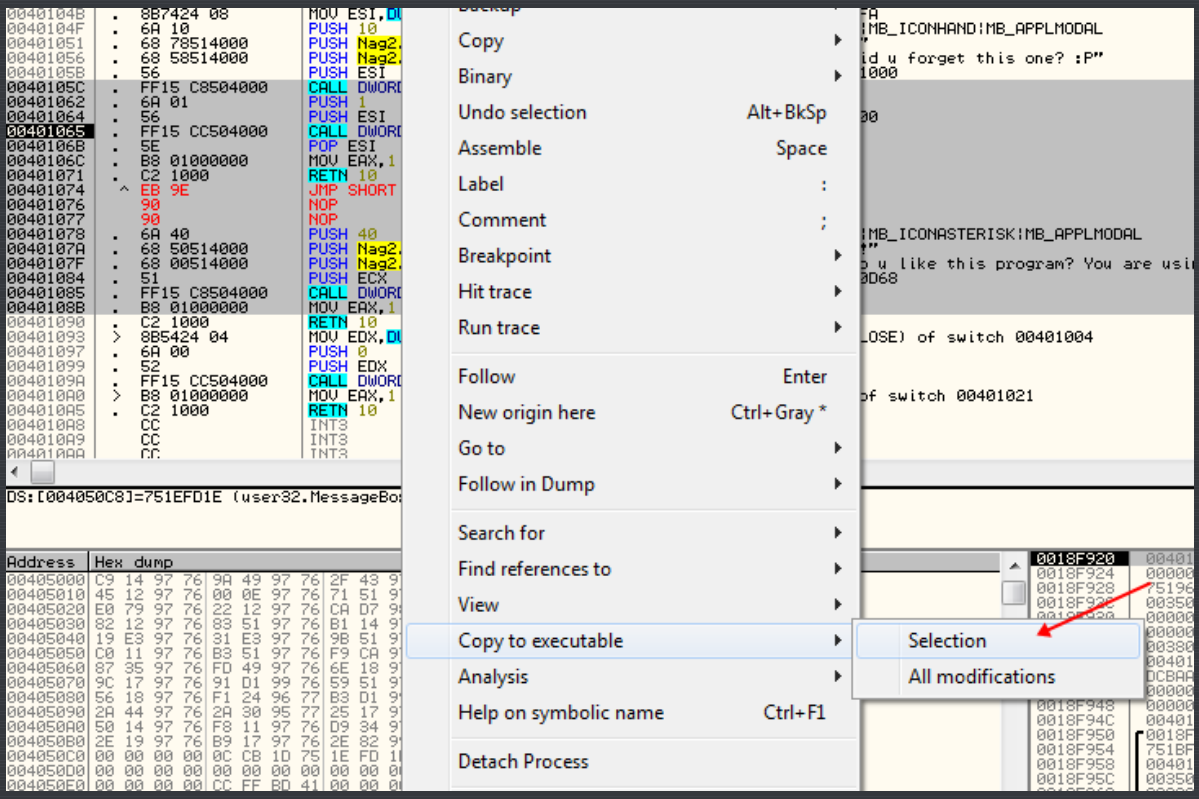


There is really no difference between NOPing the JE instruction at 401012 or adding a jump back at 401074, but I wanted you to start noticing that there are always multiple ways to patch- sometimes NOPing a call is not the best way. Remember, you OWN this binary- you can add whatever code you want, so don't be afraid to modify it, especially when learning.

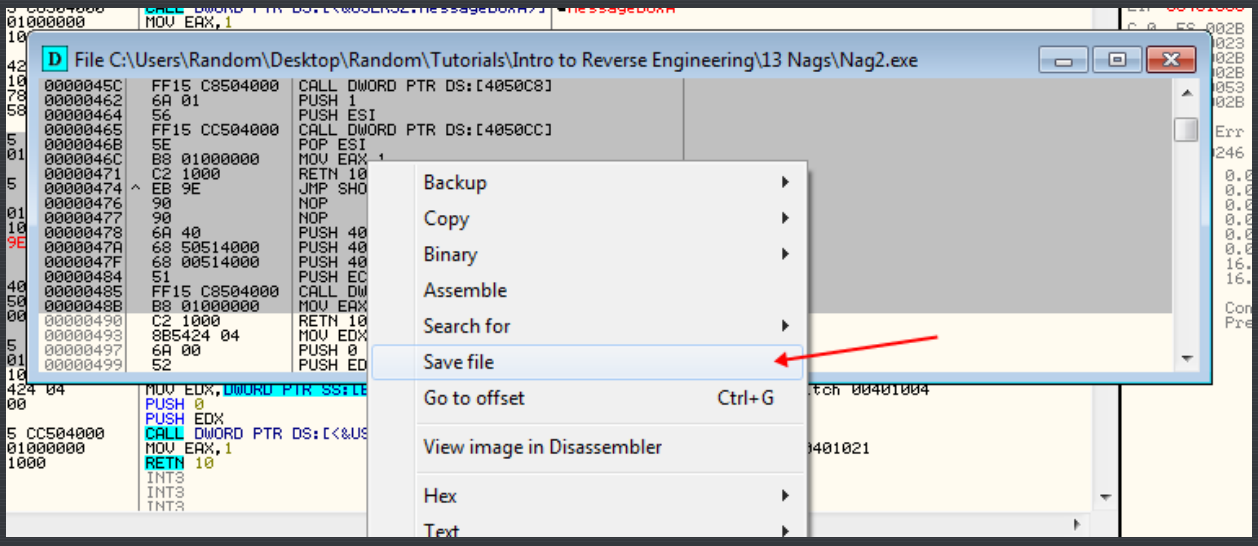
Running the app shows that the nag has been bypassed just the same:



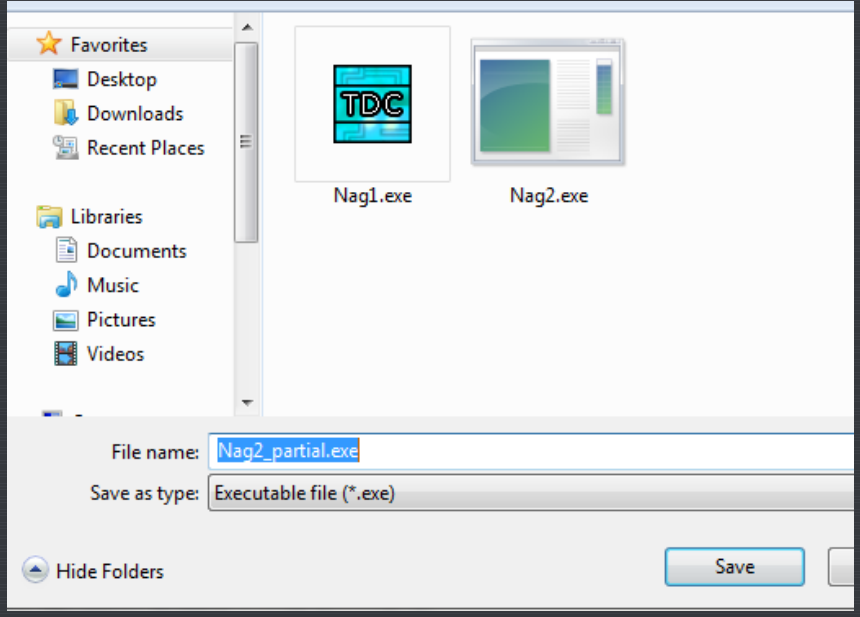
Now let's save the patch. Highlight the changed code (it's OK if you highlight more), and select "Copy to executable" -> "Selection":



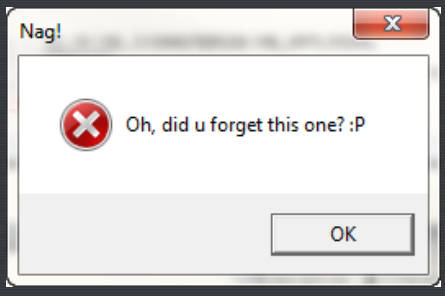
Then right click in this new window and select "Save file":



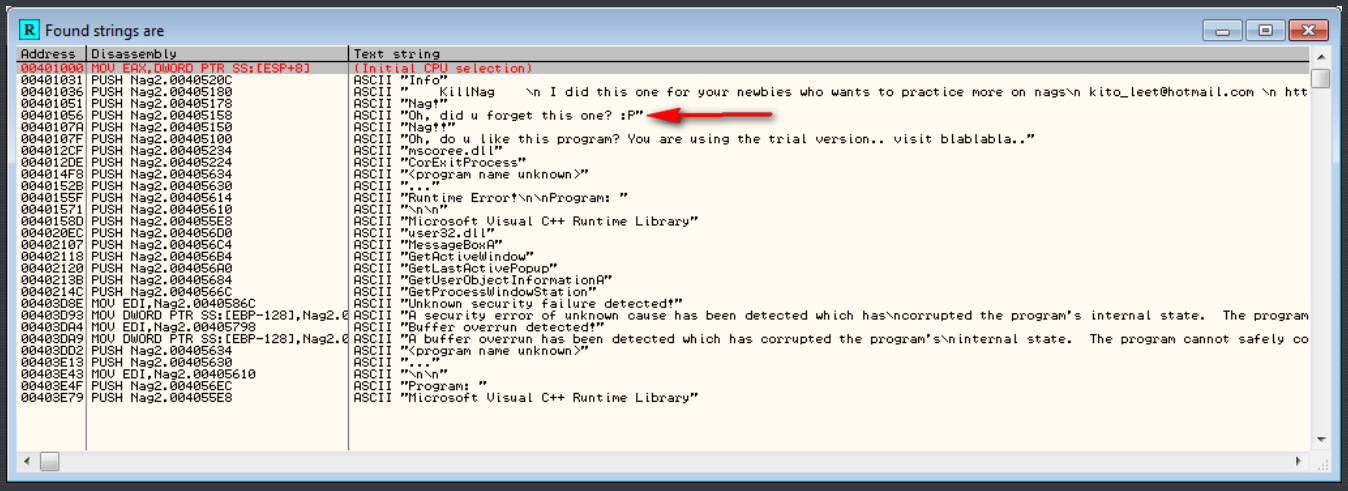
I saved it as a different name, in this case Nag2_partial.exe. You'll see why I called it partial in a minute:



OK. Go ahead and load this new patched program in Olly and let's try it out. We jump right to the main screen, so we know the patch worked. Now click exit and we get:



Uh oh. I guess the author was really determined here. Let's go find this second nag. Go back to strings and we can see that this nag's text is in there as well:

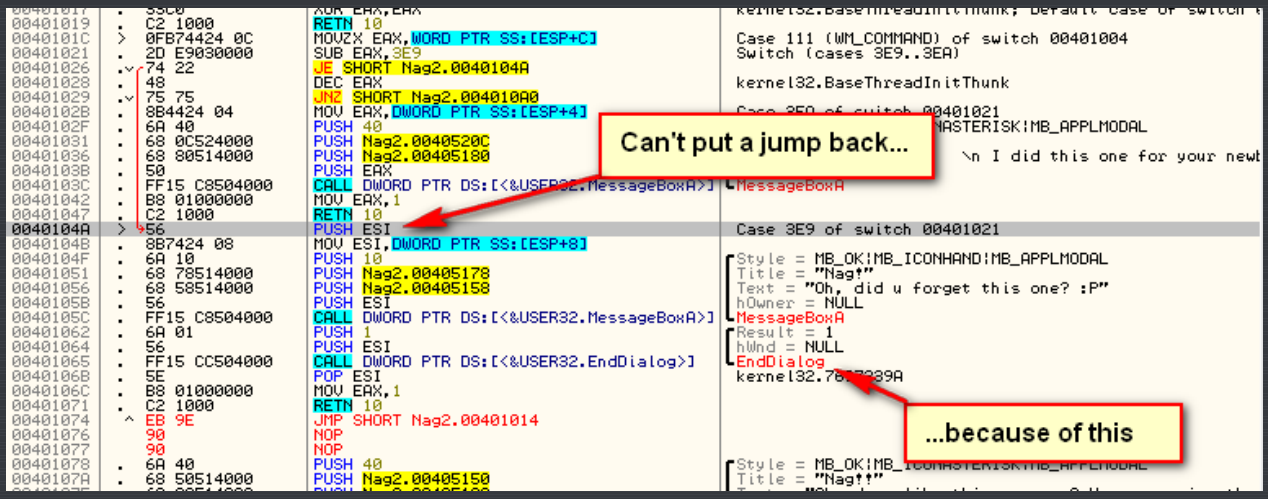


Many, many, many apps do this; they start with a nag, and after closing the app, they add another one. Most of the time, when searching for the first nag's text string, you will just automatically look for any others. Dbl-click on this text:

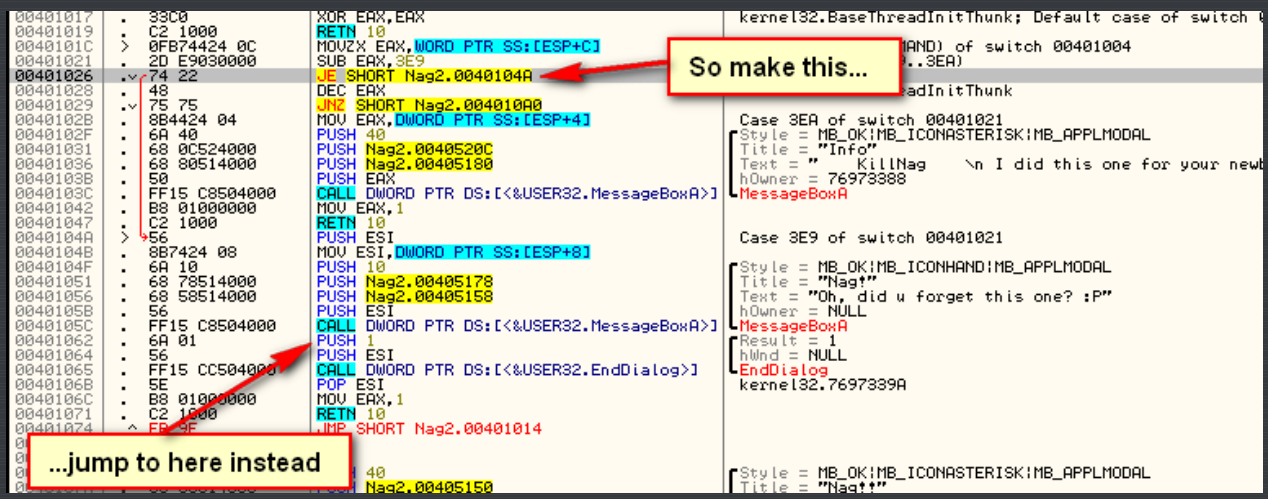
```
0040103C   .  FF15 C8504000   CALL DWORD PTR DS:[<&USER32.MessageBoxA>]   MessageBoxA
00401042   .  B8 01000000     MOV EAX,1
00401047   .  C2 1000         RETN 10
0040104A   >  56              PUSH ESI                                    Case 3E9 of switch 00401021
0040104B   .  8B7424 08       MOV ESI,DWORD PTR SS:[ESP+8]
0040104F   .  6A 10           PUSH 10                                     Style = MB_OK|MB_ICONHAND|MB_APPLMODAL
00401051   .  68 78514000     PUSH Nag2.00405178                          Title = "Nag!"
00401056   .  68 58514000     PUSH Nag2.00405158                          Text = "Oh, did u forget this one? :P"
0040105B   .  56              PUSH ESI                                    hOwner = NULL
0040105C   .  FF15 C8504000   CALL DWORD PTR DS:[<&USER32.MessageBoxA>]   MessageBoxA
00401062   .  6A 01           PUSH 1                                      Result = 1
00401064   .  56              PUSH ESI                                    hWnd = NULL
00401065   .  FF15 CC504000   CALL DWORD PTR DS:[<&USER32.EndDialog>]     EndDialog
0040106B   .  5E              POP ESI                                     kernel32.7697339A
0040106C   .  B8 01000000     MOV EAX,1
00401071   .  C2 1000         RETN 10
00401074   ^  EB 9E           JMP SHORT Nag2.00401014
00401076      90              NOP
00401077      90              NOP
00401078   .  6A 40           PUSH 40                                     Style = MB_OK|MB_ICONASTERISK|MB_APPLMODAL
0040107A   .  68 50514000     PUSH Nag2.00405150                          Title = "Nag!!"
0040107F   .  68 00514000     PUSH Nag2.00405100                          Text = "Oh, do u like this program? You are using the
00401084   .  51              PUSH ECX                                    hOwner = NULL
00401085   .  FF15 C8504000   CALL DWORD PTR DS:[<&USER32.MessageBoxA>]   MessageBoxA
0040108B   .  B8 01000000     MOV EAX,1
00401090   .  C2 1000         RETN 10
```

and here we see the method for this nag. Clicking on the first line of it we can see that the second nag is being called right after the first nag was called, but it uses a different message to trigger it (probably a window destroy message). So when this message comes through, signaling that the user has selected "Exit", the second nag will be called.

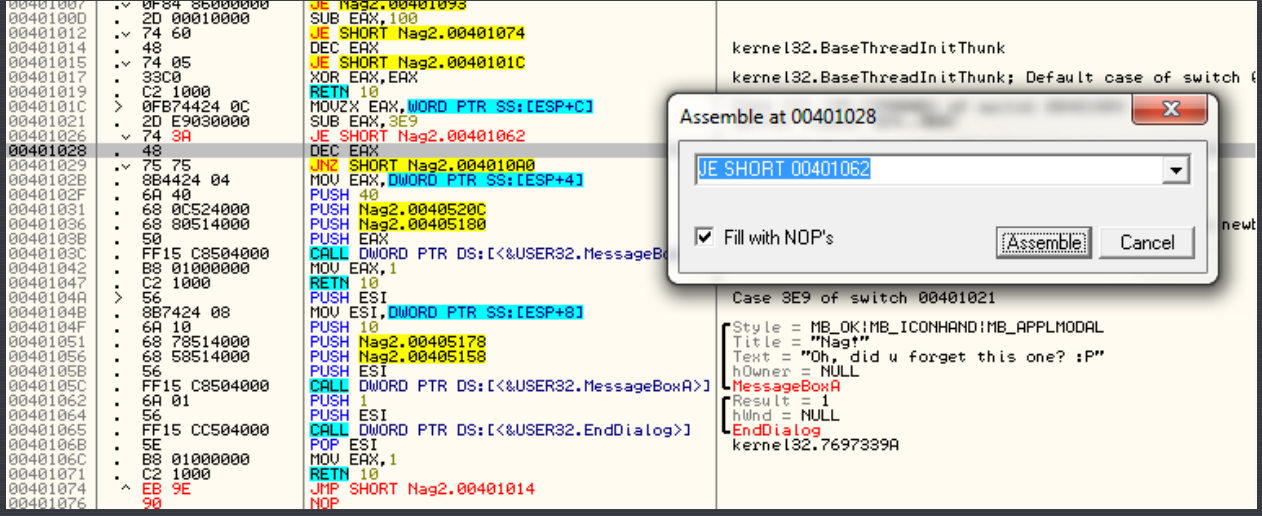
Your first thought may be "why don't we just put another jump in this one to jump right back like we did in the last one. Well, looking closely at this method, we can see that it calls the second nag, but then it immediately calls EndDialog. So jumping right back will not work as our dialog will never close:



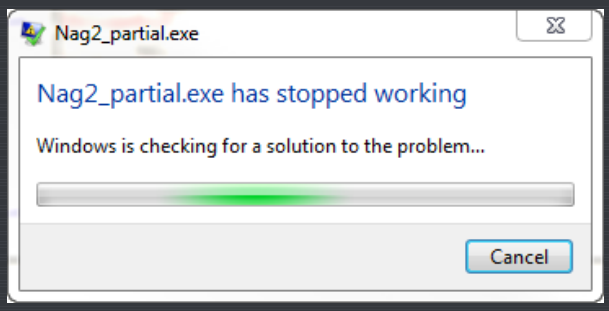
So you next thought might be, "let's just change the JE instruction at 401026 to jump to the EndDialog, jumping right over the nag MessageBoxA instruction." This is a good thought, so let's try it:



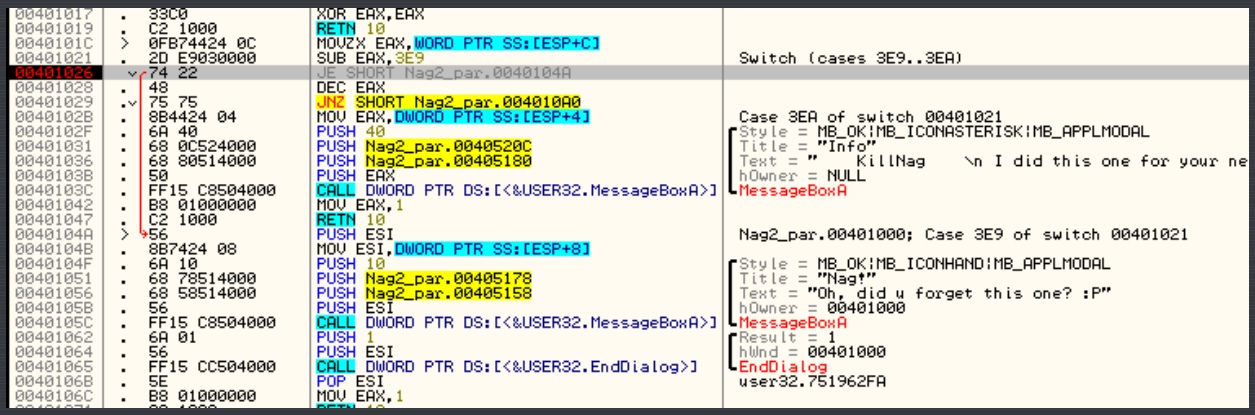
Change the JE instrution at 401026 to jump to 401062 instead, jumping to the first line of EndDialog:



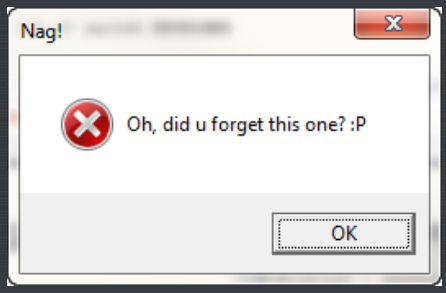
and run the app:



Well, that doesn't look too promising. So we obviously did something wrong. Here's what we're going to do; let's run the app without our patch, stepping through it, and see what it does, then run it with the patch and see how they are different. Re-start the app and click "Exit" and we will break at our patch (which is gone now that we re-started the app):



Step a couple lines and when you step over the call to MessageBoxA you will see the nag:



Now step two more times until we are on our call to EndDialog:

```
00401047  .  C2 1000        RETN 10
0040104A  >  56             PUSH ESI                                 Case 3E9 of switch 00401021
0040104B  .  8B7424 08      MOV ESI,DWORD PTR SS:[ESP+8]             Nag2_par.00401000
0040104F  .  6A 10          PUSH 10                                  Style = MB_OK|MB_ICONHAND|MB_APPLMODAL
00401051  .  68 78514000    PUSH Nag2_par.00405178                   Title = "Nag!"
00401056  .  68 58514000    PUSH Nag2_par.00405158                   Text = "Oh, did u forget this one? :P"
0040105B  .  56             PUSH ESI                                 hOwner = 00111166 ('KillNag - KiTo',class='#32770')
0040105C  .  FF15 C8504000  CALL DWORD PTR DS:[<&USER32.MessageBoxA>] MessageBoxA
00401062  .  6A 01          PUSH 1                                   Result = 1
00401064  .  56             PUSH ESI                                 hWnd = 00111166 ('KillNag - KiTo',class='#32770')
00401065  .  FF15 CC504000  CALL DWORD PTR DS:[<&USER32.EndDialog>] EndDialog
0040106B  .  5E             POP ESI
0040106C  .  B8 01000000    MOV EAX,1
00401071  .  C2 1000        RETN 10
00401074  >  EB 9E          JMP SHORT Nag2_par.00401014              Case 110 of switch 00401004
00401076     90             NOP
```

and let's look at the stack. We can see that there are four items on the stack; a handle to our window, the result of the end dialog, a pointer to the first line of our code (401000), and a return address to user32.

```
0018F9C0  00111166  hWnd = 00111166 ('KillNag - KiTo',class='#32770')
0018F9C4  00000001  Result = 1
0018F9C8  00401000  Nag2_par.00401000
0018F9CC  751962FA  RETURN to user32.751962FA
0018F9D0  00111166
0018F9D4  00000111
0018F9D8  000003E9
0018F9DC  000B1124
0018F9E0  00401000  Nag2_par.00401000
0018F9E4  DCBAABCD
0018F9E8  00000001
0018F9EC  00000000
0018F9F0  00401000  Nag2_par.00401000
0018F9F4  0018F9CC
0018F9F8  751BF943  RETURN to user32.751BF943 from user32.751962D7
0018F9FC  00401000  Nag2_par.00401000
0018FA00  00111166
0018FA04  00000111
0018FA08  000003E9
0018FA0C  000B1124
0018FA10  0E132BDD
0018FA14  00000000
0018FA18  00000111
```

Now restart the app, and when we get to our patch, activate it (patches window, hit space bar with it selected):

```
00401026  v  74 3A          JE SHORT Nag2_par.00401062
00401028  .  48             DEC EAX
00401029  .v 75 75          JNZ SHORT Nag2_par.004010A0
0040102B  .  8B4424 04      MOV EAX,DWORD PTR SS:[ESP+4]             Case 3EA of switch 00401021
0040102F  .  6A 40          PUSH 40                                  Style = MB_OK|MB_ICONASTERISK|MB_APPLMODAL
00401031  .  68 0C524000    PUSH Nag2_par.0040520C                   Title = "Info"
00401036  .  68 80514000    PUSH Nag2_par.00405180                   Text = "     KillNag    \n I did this one for your new
0040103B  .  50             PUSH EAX                                 hOwner = NULL
0040103C  .  FF15 C8504000  CALL DWORD PTR DS:[<&USER32.MessageBoxA>] MessageBoxA
00401042  .  B8 01000000    MOV EAX,1
00401047  .  C2 1000        RETN 10                                  Nag2_par.00401000; Case 3E9 of switch 00401021
0040104A  >  56             PUSH ESI
0040104B  .  8B7424 08      MOV ESI,DWORD PTR SS:[ESP+8]             Style = MB_OK|MB_ICONHAND|MB_APPLMODAL
0040104F  .  6A 10          PUSH 10                                  Title = "Nag!"
00401051  .  68 78514000    PUSH Nag2_par.00405178                   Text = "Oh, did u forget this one? :P"
00401056  .  68 58514000    PUSH Nag2_par.00405158                   hOwner = 00401000
0040105B  .  56             PUSH ESI                                 MessageBoxA
0040105C  .  FF15 C8504000  CALL DWORD PTR DS:[<&USER32.MessageBoxA>] Result = 1
00401062  .  6A 01          PUSH 1                                   hWnd = 00401000
00401064  .  56             PUSH ESI                                 EndDialog
00401065  .  FF15 CC504000  CALL DWORD PTR DS:[<&USER32.EndDialog>]  user32.751962FA
0040106B  .  5E             POP ESI
0040106C  .  B8 01000000    MOV EAX,1
00401071  .  C2 1000        RETN 10                                  Case 110 of switch 00401004
00401074  >  EB 9E          JMP SHORT Nag2_par.00401014
00401076     90             NOP
```

Patches

Address	Size	State	Old	New
00401026	2.	Active	JE SHORT Nag2_par.0040104A	JE SHORT Nag2_par.00401062

ke this program? You are using the

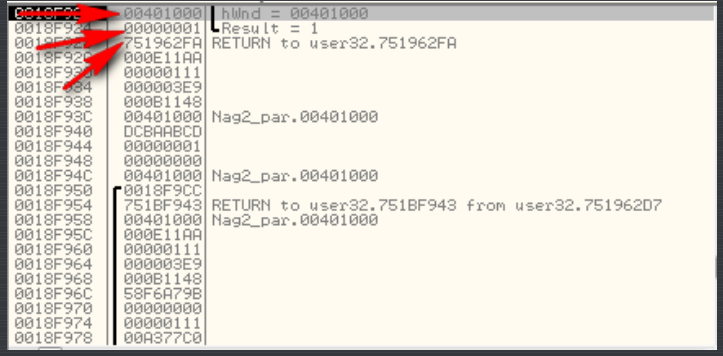
0401004

tch 00401021

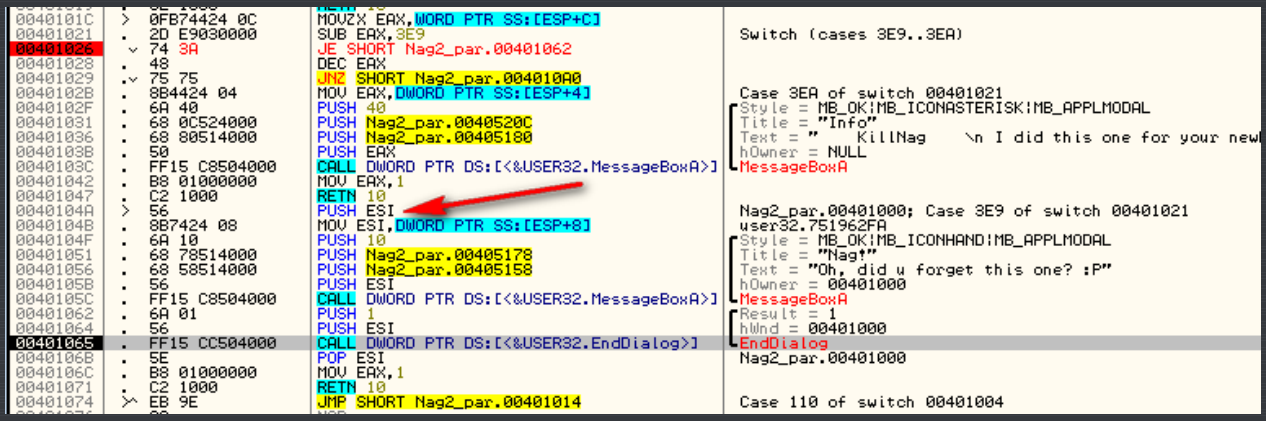
Now we will jump over the message box nag call. Step until you get to the EndDialog call:

```
0040105C  .  FF15 C8504000  CALL DWORD PTR DS:[<&USER32.MessageBoxA>] MessageBoxA
00401062  .  6A 01          PUSH 1                                   Result = 1
00401064  .  56             PUSH ESI                                 hWnd = 00401000
00401065  .  FF15 CC504000  CALL DWORD PTR DS:[<&USER32.EndDialog>]  EndDialog
0040106B  .  5E             POP ESI                                  Nag2_par.00401000
0040106C  .  B8 01000000    MOV EAX,1
00401071  .  C2 1000        RETN 10
00401074  >  EB 9E          JMP SHORT Nag2_par.00401014              Case 110 of switch 00401004
```

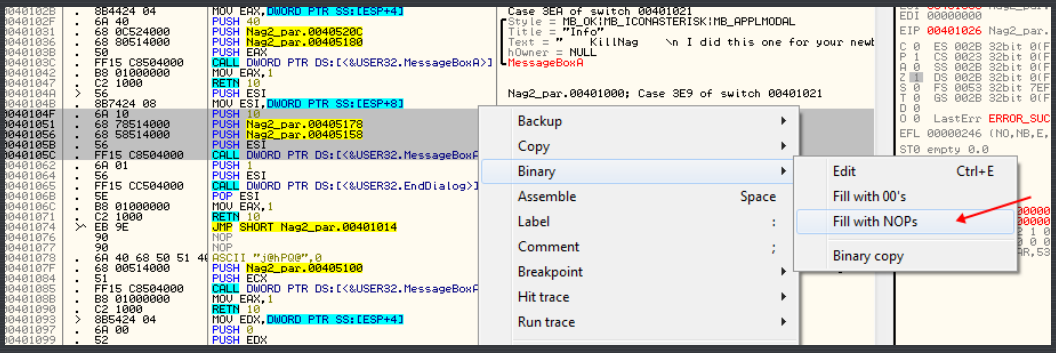
and let's look at our stack. We have the handle to the window, the result code, and a return to user32. We are missing the pointer to the first line of our code at 401000!!!



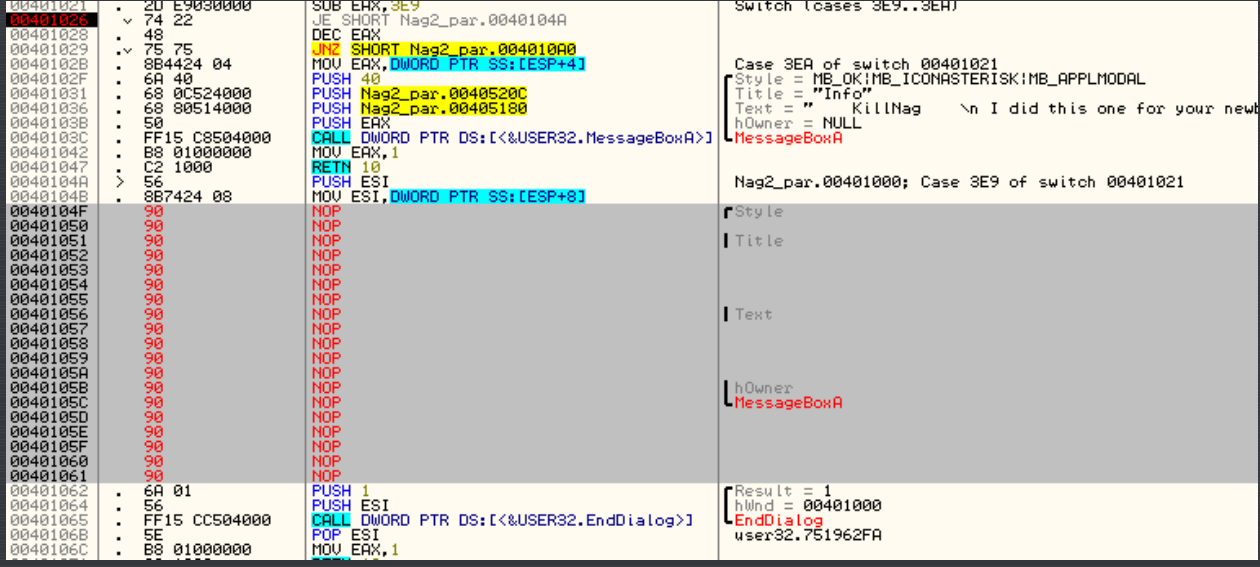
If you scroll up and take a look at the call to the second nag, you will notice that before the message box is created, ESI is pushed onto the stack. This is a pointer to our code. It just so happens that this program pushes it before calling message box, though it could have been done after. So we are missing an important push that the app needs in order to run EndDialog properly. The problem is we have some initialization code we want, then a call to a nag we don't want, then a call to EndDialog that we do want:



Well, let's get rid of the code we don't want. Highlight the MessageBoxA instructions (from 40104F to 40105C) and right click. Select "Binary" -> "fill with NOPs":



And bam! no more call to our nag:



Now when you run the app, you will notice that the app closes normally. You can now save this patch and there will be no nags left 😀 .

-Till next time

R4ndom