VOLUME 1 ISSUE 2

ARTeam eZine

October 2006

Inside This Issue

Editor: Shub-Nigurrath

1. Forewords

Finally time has come to publish the second issue of our little eZine. The first number went well, so unexpectedly well that we decided to do a second issue☺.

This second issue has a fresh look, but what really matters, the content, is there. We have a lot of contributions from team members as well from friends.

We tried to cover most of the little requests we got on our forum, and that were not possible to fit into standalone tutorials.

The content starts with some quite classical methods to patch programs, written by Gabri3l, ThunderPwr and me. Gabri3l will explain a practical method on how to add functionalities to an already existing program (notepad), while ThunderPwr will explain how to more efficiently use what the resources can say. I will instead dig a little into a faster trick to bypass event driven nags.

Then it's time for Buzifier to explain a little about scripting with Olly, and CondZero to explain a trick to handle an ACProtected program, a not so widespread packer but a powerful one nonetheless. Then it's the time of zyzygy which explains something more about Code Obfuscation methods.

Gabri3l again, will argue on a novel method of anti-debugging, based on the NtYeldExecution while anorganix will teach how to code an Oraculum (a serial sniffer) in Delphi.

For three documents deroko will bang our heads: a really interesting anti-debugger trick, some deeper look into PEB and finally some thoughts on recent TheMidas (plus its implementation as xADT plugin).

tHE mUTABLE then will close this issue explaining how to fool an interesting protector, WTM Register Maker, for all skills levels.

I think there are enough goodies for another excellent issue, that will keep you busy for few days..

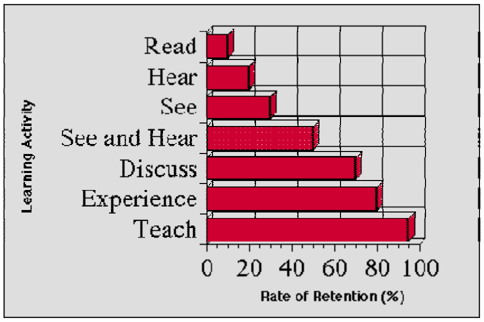
But remember that next issues are also depending on your contributions. Send them to us!

Have phun,
Shub

1. The Cone of Experience, Shub-Nigurrath of ARTeam

I thought to start this issue with a famous sentence usually attributed to William Glasser[1], with several variations:

> *People generally remember:*
>
> *10% of what they read*
>
> *20% of what they hear*
>
> *30% of what they see*
>
> *50% of what they hear and see*
>
> *70% of what they say or write*
>
> *95% of what they teach to others*



Glasser research on learning

I experience myself the truth of this sentence each time I start writing a tutorial; this can perfectly explain in my humble opinion why we do them! Moreover perfectly fits with our motto:

> *I hear and I forget, I see and I remember, I do and I understand*

Since the last issue a lot of interesting things happened to the team. First of all the third birthday of the team: 3 years on the scene and still kicking! Good!

Secondly, we launched the new site: EJ12N completely on his own developed it, starting from the graphic up to the code. He wanted to do the best accessible and nice site of the reversing scene and I think he hit the target.

We spent a lot of evenings/nights talking on what to add and how to solve some issues and so on. The result is under the eyes of everybody!

ARTeam
I hear and I forget, I see and I remember, I do and I understand

- ✓ Mobile version coming soon for all our members who would like to access the site on handheld devices.
- ✓ AA Accessibility: W3C WCAG (Web Content Accessibility Guidelines) and Section 508 (section508.gov). This is essentially equivalent to the AA Accessibility level defined by the W3C.
- ✓ eZine will *probably* be available to read online and/or download plus a couple of other stuffs...
- ✓ Tutorials system will get finished since right now we still have to implement key features such as search and couple other features like RSS. After this we will most likely want to enhance it and ask our members for suggestions.

[1] Usually, because Glasser only reported it from another author, Edgar Dale author of The Cone of Experience (http://schoolof.info/infomancy/?p=230)

- ✓ IRC Live Chat applet in the IRC section. Maybe this way we get more people to actually go in and chat. There will be stats in forum and site for this so users know who's in the channel.
- ✓ Latest X Stats – It will keep you informed of latest forum posts, latest tutorials submitted, latest tools released and well you get it latest everything happening with the site.
- ✓ Tools section. – This is still being thought of but we do have something in mind like hosting most useful RCE tools and such with of course author's permission.
- ✓ The RCE Links section will *not* be a link farm but it'll be more like a guide to guide users to many other RCE resources and sites which might be worth to visit.
- ✓ Beginners Section – We are thinking to launch a whole new section for beginners which will include video tutorials for beginners such as the ones from lena (with permission of course), our own video tutorials, and many other stuff which will be only aimed at beginners. Videos in these sections will *most likely* be available to watch online or download.
- ✓ And a whole lot more you guys will see in the near future.

I forgot something? Ah, yes: new members, new tutorials, .. but I wrote too much and it's time to stop it here! Have a good reading!

Disclaimers

All code included with this tutorial is free to use and modify; we only ask that you mention where you found it. This eZine is also free to distribute in its current unaltered form, with all the included supplements.

All the commercial programs used within the different papers have been used only for the purpose of demonstrating the theories and methods described. No distribution of patched applications has been done under any media or host. The applications used were most of the times already been patched, and cracked versions were available since a lot of time. ARTeam or the authors of the papers cannot be considered responsible for damages to the companies holding rights on those programs. The scope of this eZine as well as any other ARTeam tutorial is of sharing knowledge and teaching how to patch applications, how to bypass protections and generally speaking how to improve the RCE art. We are not releasing any cracked application.

Supplements

This eZine is distributed with Supplements for each paper; the supplements are stored in folders with the same title of the paper. Almost all the papers have supplements, check it.

Verification

ARTeam.esfv can be opened in the ARTeamESFVChecker to verify all files have been released by ARTeam and are unaltered. The ARTeamESFVChecker can be obtained in the release section of the ARTeam site: http://releases.accessroot.com

Table of Contents

1. Adding New Functionality to Old Software, Gabri3l of ARTeam

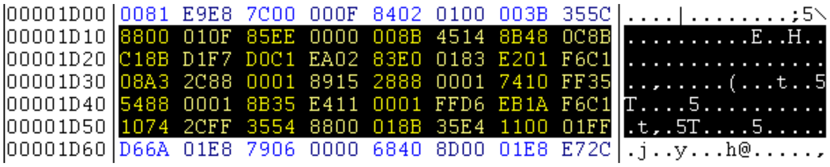
1. Abstract

The goal of this paper is to add a new feature to Notepad that will allow us to, with the click of a menu, add predefined text to the file we are working on. There have been other articles that deal with adding functions to notepad[2,3].In this paper, we are going to examine an old project with new tools. Instead of using a code-cave, as the referenced papers do, we are going to instead create a dynamic link library to perform most of our functionality for us. This will give us an easier and quicker way to write the new functions. It will also allow us to develop future updates to our notepad by simple rewriting and redistributing the DLL.

2. Adding a New Menu Item to Notepad

The first obstacle we face is how to edit the menu of a closed source, compiled executable. To know how to proceed we need to understand how windows software is compiled. Most windows software today contains both executable code and resources. The resources of an executable include icons, dialogs, music and menus. When a windows program is compiled, the executable code of the program is translated to machine code and stored in one section, while the resources the executable uses are stored in another. Because resources are not executable code most compilers store them as plain text representations.

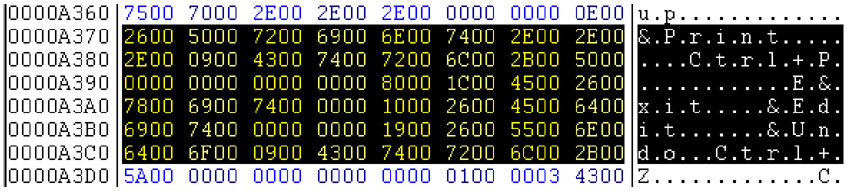
Here is an example of compiled code within Notepad when viewed by a hex editor:



```
00001D00 0081 E9E8 7C00 000F 8402 0100 003B 355C ....|........;5\
00001D10 8800 010F 85EE 0000 008B 4514 8B48 0C8B ..........E..H..
00001D20 C18B D1F7 D0C1 EA02 83E0 0183 E201 F6C1 ................
00001D30 08A3 2C88 0001 8915 2888 0001 7410 FF35 ..,.....(...t..5
00001D40 5488 0001 8B35 E411 0001 FFD6 EB1A F6C1 T....5..........
00001D50 1074 2CFF 3554 8800 018B 35E4 1100 01FF .t,.5T....5.....
00001D60 D66A 01E8 7906 0000 6840 8D00 01E8 E72C .j..y...h@.....,
```

Image 1.1- Compiled Code

Here is an example of the resource section of Notepad when viewed in a hex editor:



```
0000A360 7500 7000 2E00 2E00 2E00 0000 0000 0E00 u.p.............
0000A370 2600 5000 7200 6900 6E00 7400 2E00 2E00 &.P.r.i.n.t.....
0000A380 2E00 0900 4300 7400 7200 6C00 2B00 5000 ....C.t.r.l.+.P.
0000A390 0000 0000 0000 0000 8000 1C00 4500 2600 ............E.&.
0000A3A0 7800 6900 7400 0000 1000 2600 4500 6400 x.i.t.....&.E.d.
0000A3B0 6900 7400 0000 0000 1900 2600 5500 6E00 i.t.......&.U.n.
0000A3C0 6400 6F00 0900 4300 7400 7200 6C00 2B00 d.o...C.t.r.l.+.
0000A3D0 5A00 0000 0000 0000 0000 0100 0003 4300 Z.............C.
```

Image 1.2 - Resource Section

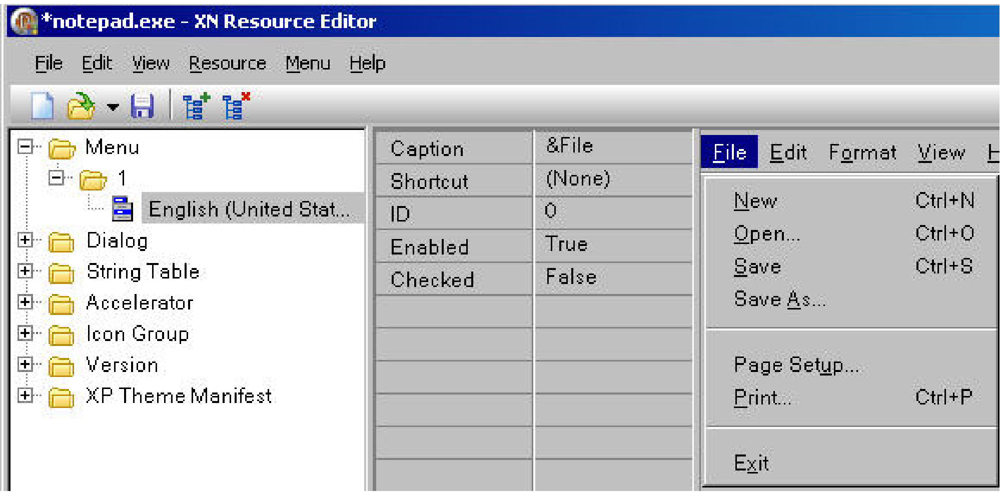
The resources are not packed or altered. This means we have access to all the resources that the program uses in one location. Many programs exist that allow us to edit the resource section of an executable. I am going to use an open source editor called XN Resource Editor[4], but any resource editor will work.

Open Notepad in your resource editor and you will have a better understanding as to how resources are included in an executable.

[2] Razzia's Tutorial for Crippled Programs: www.woodmann.com/fravia/razzcripp.htm

[3] How to Extend Notepad's Functionality by Adding Code to Caves: http://www.woodmann.com/fravia/defiler_notepad.htm

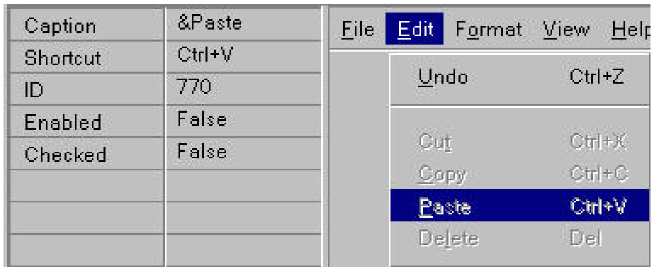
[4] XN Resource Editor: http://www.wilsonc.demon.co.uk/d10resourceeditor.htm



Caption	&File
Shortcut	(None)
ID	0
Enabled	True
Checked	False

Image 1.3 – Notepad within a Resource Editor

The resource editor allows us to view and edit all the resources included within Notepad. Each resource is nicely arranged within a folder system. If we wanted to edit the icon's we could choose the Icon Group folder and browse down to the icon we wished to change. We are going to open the Menu of Notepad. Browsing down we find there is only one menu within notepad, and that is the one we want are going to want to edit. Before we can edit the menu, we need to understand how menus work.
When notepad is idle it is operating in a loop. This loop waits for input from the user. When input is received it generates a message with information on what type of input was received. Depending on the input Notepad responds accordingly. This is called message handling[5]. When notepad is informed that a menu item has been selected it is also informed of the ID number. That way notepad can operate differently depending on what menu item was selected. Each menu item usually has a distinct ID number. For example the ID number for the Paste option in Notepad is 770.



Caption	&Paste
Shortcut	Ctrl+V
ID	770
Enabled	False
Checked	False

Image 1.4 – ID Number for Paste

We have learned two important things here. First, we know that when we create 2 new menu items we can give them distinct ID numbers to help determine how they are handled. Second, we know that somewhere within notepad, there is a routine to check the ID number and redirect to the according function. We will need to find that routine and modify it to redirect to our own functions when it receives the ID number of our new menu items.
A new menu now needs to be added, I will call my menu “QuickText”. Within that menu we will create 2 new menu items called “QuickText1” and “QuickText2. Each menu item will have a unique ID number assigned to them.

First lets add the QuickText menu. Inside XN Resource Editor browse to Notepads menu. Select the View menu, right-click and select Add Item After. This will create a new menu item after View and before Help.

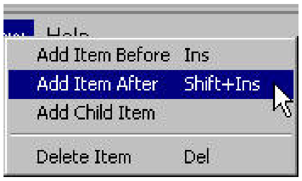
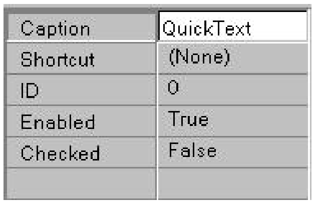


Image 1.5 – Adding a New Menu

[5] Windows Message Handling: http://www.codeproject.com/dialog/messagehandling.asp

Once the new menu is created we need to title it. The title of the menu is created by modifying the Caption.



Caption	QuickText
Shortcut	(None)
ID	0
Enabled	True
Checked	False

Image 1.6 – Modifying a Menu Caption

Now that the menu has been created we can add our menu items. The menu items are called Child Items. We add child items by selecting the QuickText menu and right-clicking. In the new menu that comes up, select Add Child Item.

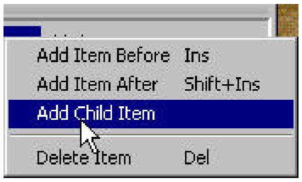
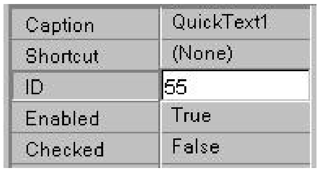


Image 1.7 – Adding a Menu Item

The new menu item will need a caption, and it will also need a unique ID number. ID number needs to be different then that of the other ID numbers already assigned to menu items. I used 55 for QuickText1.



Caption	QuickText1
Shortcut	(None)
ID	55
Enabled	True
Checked	False

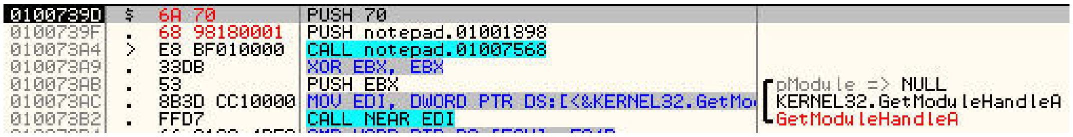
Image 1.8 – Assigning the ID Number

Following the same steps we can add another menu item called QuickText2. I assigned that menu item the ID number 66. Save the modified Notepad.exe as something such as Notepad.QuickText.exe.

We should now have a new Notepad, with a new menu and two new items. You can run notepad and select either QuickText option, but you will notice that nothing happens. This is because the message being passed to Notepad's message handler has an ID value that is not being handled. Our goal now is to locate the message handler so we can find where the ID value is tested, and how we can redirect the function.

3. Locating the Message Handler

The next tool we will use is Ollydbg[6]. We will be using it to analyze the executable and control program execution. This will allow us to find the message handler and pinpoint where the ID value comparison takes place. Open Notepad.exe within Ollydbg. The program will load and we will be at the Entry Point of the executable.



```
0100739D  $  6A 70          PUSH 70
0100739F  .  68 98180001    PUSH notepad.01001898
010073A4  >  E8 BF010000    CALL notepad.01007568
010073A9  .  33DB           XOR EBX, EBX
010073AB  .  53             PUSH EBX                                  [pModule => NULL
010073AC  .  8B3D CC10000   MOV EDI, DWORD PTR DS:[<&KERNEL32.GetMo   KERNEL32.GetModuleHandleA
010073B2  .  FFD7           CALL NEAR EDI                             GetModuleHandleA
```

Image 1.9 – Entry Point of Notepad

[6] Ollydbg: http://www.ollydbg.de

Creating a Message Loop

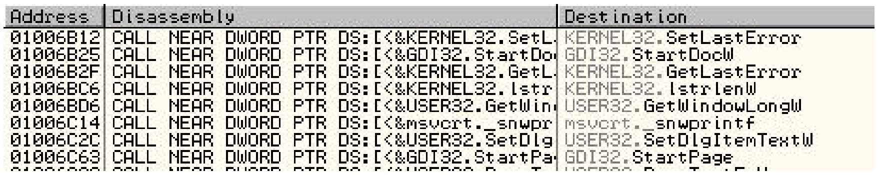
The system automatically creates a message queue for each thread. If the thread creates one or more windows, a message loop must be provided; this message loop retrieves messages from the thread's message queue and dispatches them to the appropriate window procedures.

Because the system directs messages to individual windows in an application, a thread must create at least one window before starting its message loop. Most applications contain a single thread that creates windows. A typical application registers the window class for its main window, creates and shows the main window, and then starts its message loop — all in the **WinMain** function.

You create a message loop by using the **GetMessage** and **DispatchMessage** functions. If your application must obtain character input from the user, include the **TranslateMessage** function in the loop. **TranslateMessage** translates virtual-key messages into character messages. The following example shows the message loop in the **WinMain** function of a simple Windows-based application.

We are going to begin by locating the Message Handler Loop. Take a quick look at how the message handler loop is defined in the MSDN[7]:

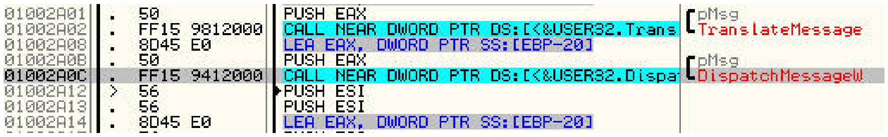
By examining this definition we gain some valuable information. We know that within a standard message loop we are going to find 3 distinct API calls: **GetMessage**, **DispatchMessage**, and **TranslateMessage**. The GetMessage API function retrieves a message from the message queue. TranslateMessage is performed if the message from the queue is a key-press, TranslateMessage then interprets the ASCII character represented by the keyboard key. It then adds that character to the message queue. Finally, DispatchMessage sends the message to the executable's message handler where the program reacts accordingly. We can use these 3 API calls to find where our message loop is located, and subsequently, our message handler.
Within Ollydbg, right-click inside the code frame and select Search For. In the new menu choose All Intermodular Calls. Ollydbg will then search the executable for all the API calls made from within Notepad.exe, a new window will open with the results of that search:



```
Address  Disassembly                               Destination
01006B12 CALL NEAR DWORD PTR DS:[<&KERNEL32.SetL.  KERNEL32.SetLastError
01006B25 CALL NEAR DWORD PTR DS:[<&GDI32.StartDo.  GDI32.StartDocW
01006B2F CALL NEAR DWORD PTR DS:[<&KERNEL32.GetL.  KERNEL32.GetLastError
01006BC6 CALL NEAR DWORD PTR DS:[<&KERNEL32.lstr   KERNEL32.lstrlenW
01006BD6 CALL NEAR DWORD PTR DS:[<&USER32.GetWin.  USER32.GetWindowLongW
01006C14 CALL NEAR DWORD PTR DS:[<&msvcrt._snwpr   msvcrt._snwprintf
01006C2C CALL NEAR DWORD PTR DS:[<&USER32.SetDlg   USER32.SetDlgItemTextW
01006C63 CALL NEAR DWORD PTR DS:[<&GDI32.StartPa.  GDI32.StartPage
```

Image 1.10 – API Search Results

In the Search Results window, press the Destination column header, which will sort the results by the API destination. Scroll down until you find **DispatchMessageW**. That was one of the API functions included in the Message Loop description. Choose the first instance of DispatchMessageW and double-click on it. You should be located here now in the code frame:



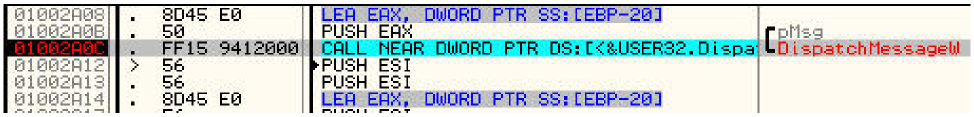
```
01002A01  .  50            PUSH EAX                                     [pMsg
01002A02  .  FF15 9812000  CALL NEAR DWORD PTR DS:[<&USER32.Trans       [TranslateMessage
01002A08  .  8D45 E0       LEA EAX, DWORD PTR SS:[EBP-20]
01002A0B  .  50            PUSH EAX                                     [pMsg
01002A0C  .  FF15 9412000  CALL NEAR DWORD PTR DS:[<&USER32.Dispa       [DispatchMessageW
01002A12  >  56            PUSH ESI
01002A13  .  56            PUSH ESI
01002A14  .  8D45 E0       LEA EAX, DWORD PTR SS:[EBP-20]
```

Image 1.11 – Message Loop

Look above DispatchMessage and you will see TranslateMessage. If you were to run the program and break at CALL NEAR EDI, located below DispatchMessage you would find that it is our call to GetMessage.
We have found the location of our message loop. The next goal is to find the comparison routine of our ID number. Referring back to how the message loop works we know that Dispatch Message sends the current message to Notepad's message handler. If we can stop the program as it is executing DispatchMessageW we can use it to locate Notepad's message handler. Begin by setting a breakpoint on the CALL to DispatchMessageW, do this by selecting the CALL to DispatchMessageW and pressing F2. Now when we run Notepad and it enters the message loop we will stop execution before the message

[7] Using Messages and Message Queues: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/windowing/messagesandmessagequeues/usingmessagesandmessagequeues.asp

handler is called. Make sure you have your breakpoint set and press F9 in Ollydbg to run Notepad. Wait a few seconds and we should stop execution here in our Message Loop:



```
01002A08  .  8D45 E0        LEA EAX, DWORD PTR SS:[EBP-20]
01002A0B  .  50             PUSH EAX                                 [pMsg
01002A0C  .  FF15 9412000   CALL NEAR DWORD PTR DS:[<&USER32.Dispa   LDispatchMessageW
01002A12  >  56            ▸PUSH ESI
01002A13  .  56             PUSH ESI
01002A14  .  8D45 E0        LEA EAX, DWORD PTR SS:[EBP-20]
```

Image 1.12 – Stopping Execution on DispatchMessageW

We are now stopped at our CALL to DispatchMessageW. To find the message handler we are going to use these facts to our advantage:

- DispatchMessageW calls our message handler
- Our message handler is inside Notepad
- As the call is executed we will leave Notepad's memory space and enter User32.dll
- We will have to re-enter Notepad to execute the message handler

What we are going to is step into the CALL DispatchMessageW, which will take us outside of Notepads memory. We are going to then set a breakpoint when the code section of Notepad is accessed[8], this will stop execution when we re-enter Notepads memory space. When execution breaks it is because we have returned to notepads message handler. We will be located right at the beginning of the message handler.

Press F7 to step into DispatchMessageW and place ourselves in User32.dll. Now we need to place our breakpoint on Notepad's code section. In Ollydbg, click on the View menu, in the drop down menu select Memory.

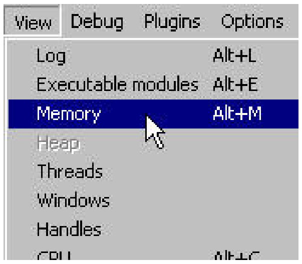
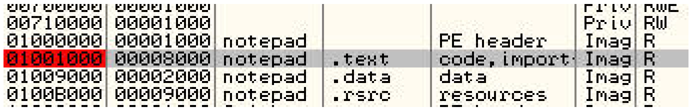


Image 13 – View Memory

A new window will open that displays the memory for this process. In the Owner column of the memory window you will see the name of the executable that resides in that memory space. In the beginning of this paper we discussed that an executable stores different data in different sections. Traditionally executable code is stored in one section, while resources are stored in another. In Ollydbg's memory window the Section column displays the different section names for each of the executables sections. Locate Notepad in the Owner column and we see that Notepad has 3 named sections, and a PE header[9]. Often times the .text section of an executable contains the actual code. You can verify that this is true for Notepad by looking in the Contains column; there you see .text contains code and imports. We could also look at the Contains column and verify that our resources are contained in the named section .rsrc. Back to the project, select the row that has Notepad's .text section, press F2 to set a breakpoint-on-access for that section.



```
00710000 00001000                            Priv RW
01000000 00001000 notepad          PE header Imag R
01001000 00008000 notepad .text    code,import Imag R
01009000 00002000 notepad .data    data      Imag R
0100B000 00009000 notepad .rsrc    resources Imag R
```

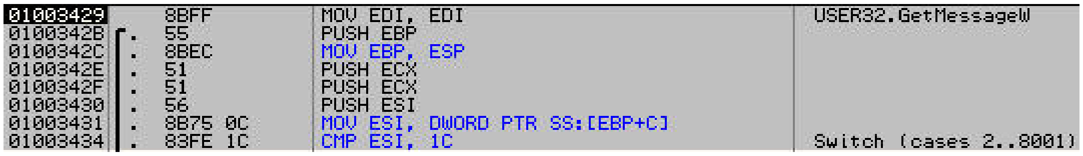
Image 1.14 – Breakpoint on Memory Section Access

Now when we execute the program we will break when DispatchMessage returns to our Message Handler located within the .text section of Notepad. Make sure you have stepped into the DispatchMessageW call, and press F9 to continue execution.

You should trigger your page access breakpoint and find yourself here in Notepad.exe:

[8] A breakpoint on access to memory section can also be referred to as PAGE_GUARD. Creating Guard Pages: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/creating_guard_pages.asp

[9] For more information on PE files and their structure: Microsoft Portable Executable and Common Object File Format Specification: http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx



```
01003429    8BFF        MOV EDI, EDI                         USER32.GetMessageW
0100342B  . 55          PUSH EBP
0100342C  . 8BEC        MOV EBP, ESP
0100342E  . 51          PUSH ECX
0100342F  . 51          PUSH ECX
01003430  . 56          PUSH ESI
01003431  . 8B75 0C     MOV ESI, DWORD PTR SS:[EBP+C]
01003434  . 83FE 1C     CMP ESI, 1C                          Switch (cases 2..8001)
```

Image 1.15 – Notepads Message Handler

We have located the beginning of Notepad's message handler. Within this section of code Notepad compares the type of message it needs to handle and takes appropriate action. Before we can continue we need to know what type of message is sent to Notepad when we choose a menu item. In windows there are over 200 types of messages.[10] A few examples of different types of messages:

WM_CREATE
your window receives this message only once, when it is first created. Use this message to perform tasks that need to be handled in the beginning, such as initializing variables, allocating memory, or creating child windows (buttons and textboxes).

WM_PAINT
This message indicates that it is time for the program to redraw itself. Use the graphical functions to redraw whatever is supposed to be on the window. If you don't draw anything, then the window will just be a boring white (or grey) background, and nobody likes that!

WM_COMMAND
This is a general message that indicates that the user has done something on your window. Either the user has clicked a button, or the user has selected a menu item, or the user has pressed a special "Accelerator" key sequence. The WPARAM and LPARAM fields will contain some descriptions on what happened, so you can find a way to react to this. If you do not process the WM_COMMAND messages, the user will not be able to click any buttons, or select any menu items, and that will be very frustrating indeed.

WM_MOUSEMOVE
This message indicates that the user has moved the mouse. This message is posted to a window when the cursor moves. If the mouse is not captured, the message is posted to the window that contains the cursor. Otherwise, the message is posted to the window that has captured the mouse.

Windows has tried to accommodate every type of message your program would need. But obviously you want to know more than if a button has been pressed or a mouse has moved. You need to know what has been pressed or where the mouse moved to. The message WM_MOUSEMOVE is not enough information to tell you what took place when the mouse was moved. That is why these windows messages also carry parameters. Each message can carry two parameters **wparam** and **lparam**. The parameters of a message help to specify what exactly happened during an event. For example, If you opened an executable and moved your mouse, Windows will send the executable a WM_MOUSEMOVE message. The message will also contain the wparam and lparam parameters. wparam carries the keyflags, which allows you to see if a mouse button was held down during the mouse move. lparam carries the x and y position of the mouse telling you exactly where the mouse is on the screen. Parameters are also used when dealing with buttons and menu items. When a button or menu item is pressed Windows sends a WM_COMMAND to the executable. This WM_COMMAND stores the ID number of the button or menu item in the wparam parameter.

We have a better understanding of messages and message handling, now we can return to locating where notepad compares our menu ID number. We know that when our menu item is selected it sends a WM_COMMAND message. We also know that our ID is stored in the wparam parameter of the message. So first we will try and locate where the program handles the WM_COMMAND message.

You should still be at the beginning of Notepads message handler in Ollydbg. There are different ways for programs to handle the actual messages but the most common way is to use a **switch**.[11] A switch is a programming method that evaluates a variable and performs an action based on the variable's value. The action performed is known as a **case**. Let's say we have a program that takes the number of the month and we want to return the name of the month. Using If-Then-Else statements we could write it as such:

[10] List of Windows Messages: http://wiki.winehq.org/List_Of_Windows_Messages

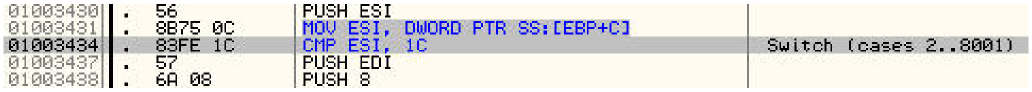
[11] C++ Switch Case Statements: http://www.cprogramming.com/tutorial/lesson5.html

```
IF month = 1
        print “January”
ELSE
        IF month = 2
                print “February”
        ELSE
                IF month = 3
                        print “March”
                ...
```

Or it could be written using a switch statement:

```
Switch(month)
Case 1: print “January”
Case 2: print “February”
Case 3: print “March”
```

The switch statement is a much more efficient and aesthetic solution. Because of this many programs use the switch statement to evaluate and handle messages. Using Ollydbg we can see the Notepads switch statement for handling messages. Scroll down from the beginning of the message handler and you will see:



```
01003430  .  56          PUSH ESI
01003431  .  8B75 0C     MOV ESI, DWORD PTR SS:[EBP+C]
01003434  .  83FE 1C     CMP ESI, 1C                      Switch (cases 2..8001)
01003437  .  57          PUSH EDI
01003438  .  6A 08       PUSH 8
```

Image 1.16 – Switch Statement in Notepad

This is the start of the switch statement that handles all the window messages. Using this switch we can locate where Notepad evaluates WM_COMMAND. Highlight the line that says Switch (cases 2..8001) and right-click. Choose Go-To from the right-click menu and in the new menu choose More Cases:

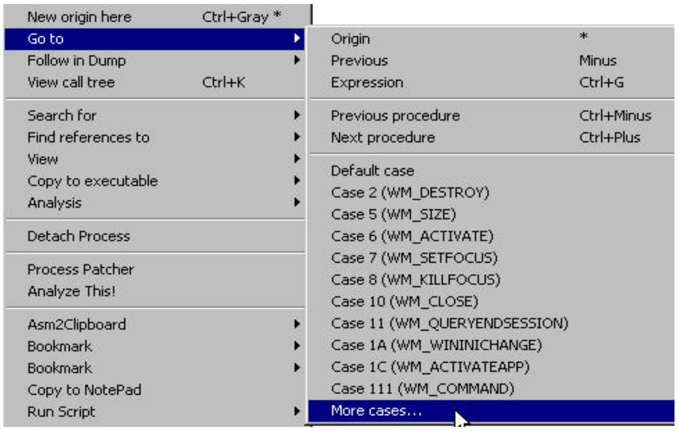
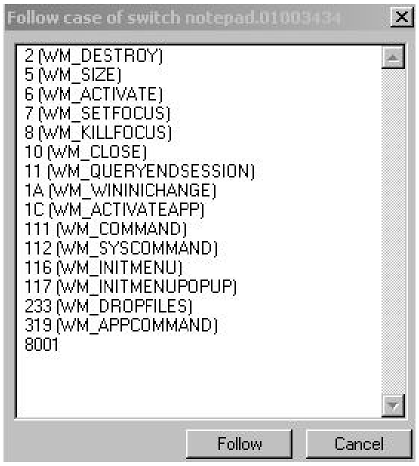


Image 1.17 – Locating Cases within Ollydbg

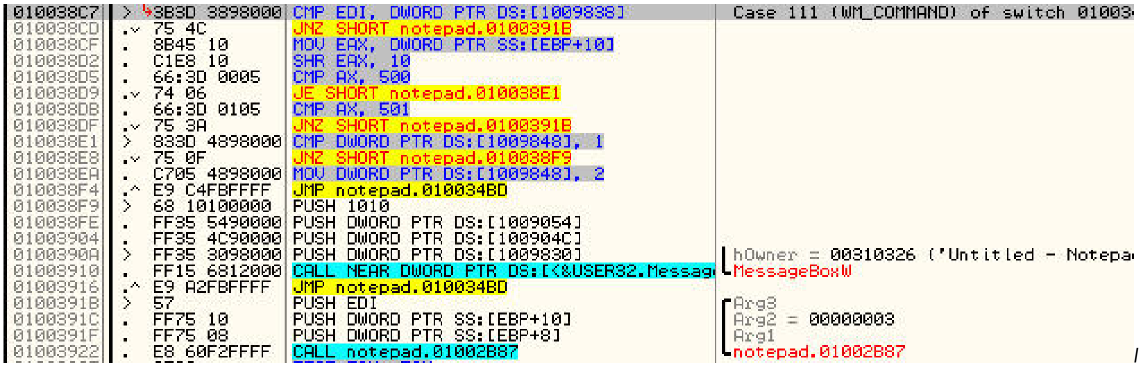
A new window should open within OllyDbg allowing you to see all the different cases that this switch handles:



```
2 (WM_DESTROY)
5 (WM_SIZE)
6 (WM_ACTIVATE)
7 (WM_SETFOCUS)
8 (WM_KILLFOCUS)
10 (WM_CLOSE)
11 (WM_QUERYENDSESSION)
1A (WM_WININICHANGE)
1C (WM_ACTIVATEAPP)
111 (WM_COMMAND)
112 (WM_SYSCOMMAND)
116 (WM_INITMENU)
117 (WM_INITMENUPOPUP)
233 (WM_DROPFILES)
319 (WM_APPCOMMAND)
8001
```

Image 1.18 – Viewing Cases within Ollydbg

We can see that Case 111 is WM_COMMAND. Highlight the line and press Follow to go directly to the beginning of Notepads handler for WM_COMMAND. You will find yourself here in Ollydbg:



```
010038C7  > 3B3D 3898000  CMP EDI, DWORD PTR DS:[1009838]          Case 111 (WM_COMMAND) of switch 01003
010038CD  .v 75 4C        JNZ SHORT notepad.0100391B
010038CF  .  8B45 10      MOV EAX, DWORD PTR SS:[EBP+10]
010038D2  .  C1E8 10      SHR EAX, 10
010038D5  .  66:3D 0005   CMP AX, 500
010038D9  .v 74 06        JE SHORT notepad.010038E1
010038DB  .  66:3D 0105   CMP AX, 501
010038DF  .v 75 3A        JNZ SHORT notepad.0100391B
010038E1  >  833D 4898000 CMP DWORD PTR DS:[1009848], 1
010038E8  .v 75 0F        JNZ SHORT notepad.010038F9
010038EA  .  C705 4898000 MOV DWORD PTR DS:[1009848], 2
010038F4  .^ E9 C4FBFFFF  JMP notepad.010034BD
010038F9  >  68 10100000  PUSH 1010
010038FE  .  FF35 5490000 PUSH DWORD PTR DS:[1009054]
01003904  .  FF35 4C90000 PUSH DWORD PTR DS:[100904C]
0100390A  >  FF35 3098000 PUSH DWORD PTR DS:[1009830]              hOwner = 00310326 ('Untitled - Notepa
01003910  .  FF15 6812000 CALL NEAR DWORD PTR DS:[<&USER32.Messag  MessageBoxW
01003916  .^ E9 A2FBFFFF  JMP notepad.010034BD
0100391B  >  57           PUSH EDI                                 Arg3
0100391C  .  FF75 10      PUSH DWORD PTR SS:[EBP+10]               Arg2 = 00000003
0100391F  .  FF75 08      PUSH DWORD PTR SS:[EBP+8]                Arg1
01003922  .  E8 60F2FFFF  CALL notepad.01002B87                    notepad.01002B87
```

mage 1.19 – WM_COMMAND Handler in Notepad

Examining the code there is no noticeable functions that look as if they are comparing a button or menu ID number. We do, however, see that there are 2 calls within the WM_COMMAND case. We have a call to MessageBoxW and a call to Notepad.01002B87. Now keep in mind your offsets may look different for the second call, but it does not change the call. Using common sense, since the comparison is not taking place among the code we are in currently, and the comparison obviously is not taking place within MessageBoxW, we can assume that within that second call is probably where our ID number comparison is taking place.
Highlight the second CALL and right-click. In the new menu that opens up choose Follow. That will step into the call to Notepad.01002B87.

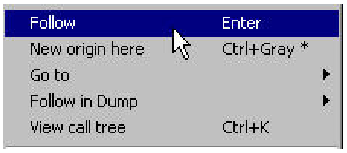
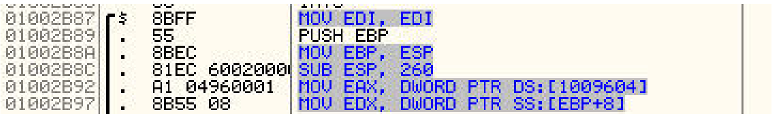


Image 1.20 – Follow a CALL in Ollydbg

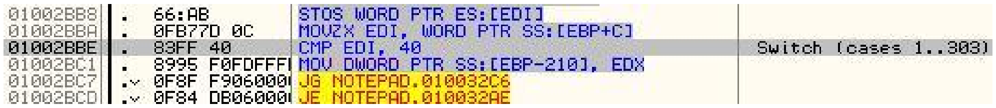
After following the CALL within Ollydbg you will find yourself at the beginning of a new function:



```
01002B87  $  8BFF          MOV EDI, EDI
01002B89  .  55            PUSH EBP
01002B8A  .  8BEC          MOV EBP, ESP
01002B8C  .  81EC 6002000  SUB ESP, 260
01002B92  .  A1 04960001   MOV EAX, DWORD PTR DS:[1009604]
01002B97  .  8B55 08       MOV EDX, DWORD PTR SS:[EBP+8]
```

Image 1.21 – Function Beginning

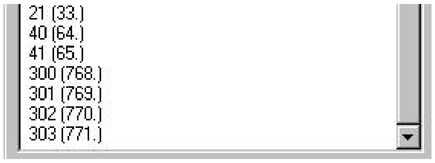
Take a quick look at the code within this function. Scrolling down we see that we are quickly presented with another switch statement. We see **switch (cases 1..303)** in the comments section to the right of one of our code locations. Note, if you do not see the switch comment then press CTRL+A to analyze the code.



```
01002BB8  .   66:AB          STOS WORD PTR ES:[EDI]
01002BBA  .   0FB77D 0C      MOVZX EDI, WORD PTR SS:[EBP+C]
01002BBE  .   83FF 40        CMP EDI, 40                          Switch (cases 1..303)
01002BC1  .   8995 F0FDFFF   MOV DWORD PTR SS:[EBP-210], EDX
01002BC7  .v  0F8F F906000   JG NOTEPAD.010032C6
01002BCD  .v  0F84 DB06000   JE NOTEPAD.010032AE
```

Image 1.22 – Switch

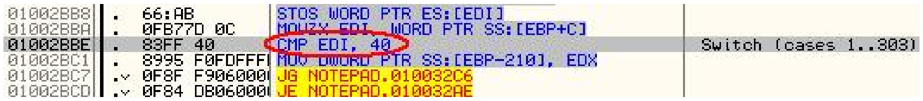
We need to find out more information about exactly what numbers this switch is comparing. As we did before, highlight the first line of the switch and choose Go-To, then select More Cases. We are presented with a new box that contains the list of the cases for this switch. Scroll to the bottom and look at some of the cases. The first number you see is the hexadecimal number; the next number in parentheses is the decimal number equivalent. Do any of the numbers look familiar?



```
21 (33.)
40 (64.)
41 (65.)
300 (768.)
301 (769.)
302 (770.)
303 (771.)
```

Image 1.23 – Switch Cases

Hopefully you recognize the number 770. If you did not recognize the number refer back to [Image 1.4]. 770 is the decimal value for the Paste menu item in Notepad. If we were to look through the resource section again we would find 768 is the ID number of Cut menu item, 769 is the ID number of Copy, etc... This means that we have successfully found Notepads ID number comparison function and we also have located the switch within that function that compares our ID number.
Write down the location of the comparison routine we will need to jump back to this location later in the paper. In my case the comparison function begins at **1002B87** your address value may differ but refer to [Image 1.21] to verify that the code is the same. We will also need to write down what register is being used within our switch. This will let us know where our ID number is being stored. We will use that register when we write our DLL. Scroll back to the beginning of the switch and we find that the register that is being compared in our switch is EDI.



```
01002BB8  .   66:AB          STOS WORD PTR ES:[EDI]
01002BBA  .   0FB77D 0C      MOVZX EDI, WORD PTR SS:[EBP+C]
01002BBE  .   83FF 40        CMP EDI, 40                          Switch (cases 1..303)
01002BC1  .   8995 F0FDFFF   MOV DWORD PTR SS:[EBP-210], EDX
01002BC7  .v  0F8F F906000   JG NOTEPAD.010032C6
01002BCD  .v  0F84 DB06000   JE NOTEPAD.010032AE
```

Image 1.24 – ID Number Comparison

We have now found the location of the comparison function and we know what register is being compared. The next step will be to redirect the function to account for our two new menu items. We could redirect to a code-cave and hard-code all values and variables within Notepad. We are not going to take that approach; instead we are going to develop a DLL. Notepad will be modified to load the DLL and then we will redirect our ID comparison function to another function within the DLL. Before we can continue within Notepad we need to start developing our DLL

4. Developing the QuickText DLL

4.1 Planning

Developing a DLL can be intimidating for someone new to the world of reverse-engineering and coding. In this section we will determine how our DLL needs to function. We will then develop a flowchart to show the desired execution. Finally I will walk you through creating your QuickText DLL. To begin we are going to write a simple flowchart to demonstrate how our DLL is going to work. Even if you do not know how to write a DLL this will help you understand how the code is functioning. The following is our desired execution of Notepad and the QuickText DLL. The Red objects on our flowchart show the modifications to Notepads execution. The dark blue objects illustrate the function of our DLL:

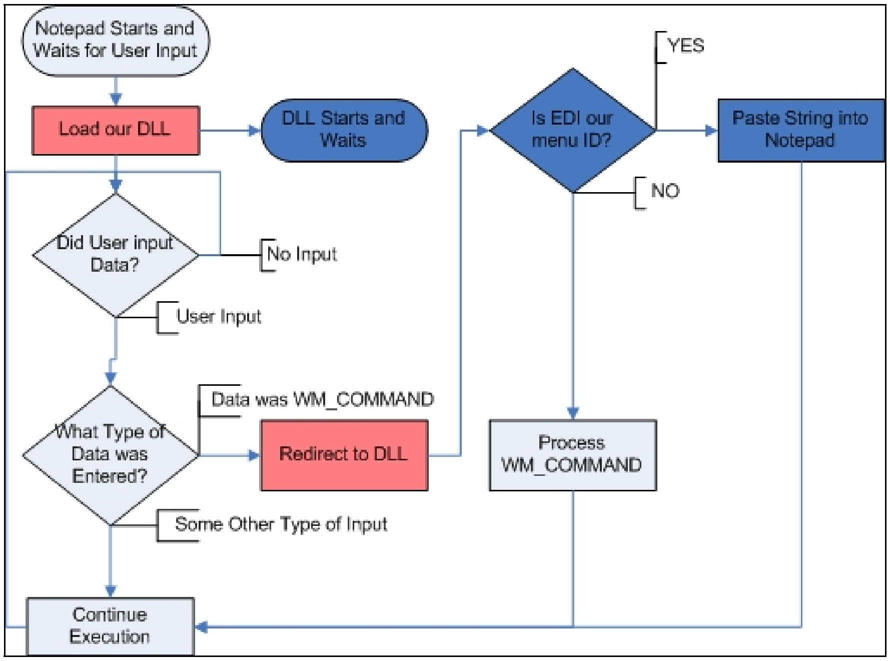


Image 1.25 – Simple Flowchart Representation of Desired Notepad Execution

- First Notepad begins
- Next there is a modification within Notepad that will load our Quicktext DLL
- Execution will continue until Notepad receives a WM_COMMAND message
- At that point Notepad has been modified to call a function within our QuickText DLL
  - ✓ Within that function EDI is being compared to our two Menu Ids
  - ✓ If EDI equals one of our Menu IDs then we will paste the string corresponding to that menu into Notepad
  - ✓ Otherwise we will return to Notepad and allow it to continue.

We cannot modify Notepad yet because we do not have a working DLL to load or redirect to. First we need to focus on how to achieve our desired DLL functionality. Writing a DLL will not be that hard, we can use a simple skeleton to help us develop it. However, within our QuickText DLL we are going to need to create a function that will compare EDI and paste the specific string into Notepad. I will be calling that function QuickPaste. That will be the function we have Notepad call if it receives a WM_COMMAND message. The next flowchart is an illustration of loading the DLL and the execution that needs to take place within the QuickPaste Function:

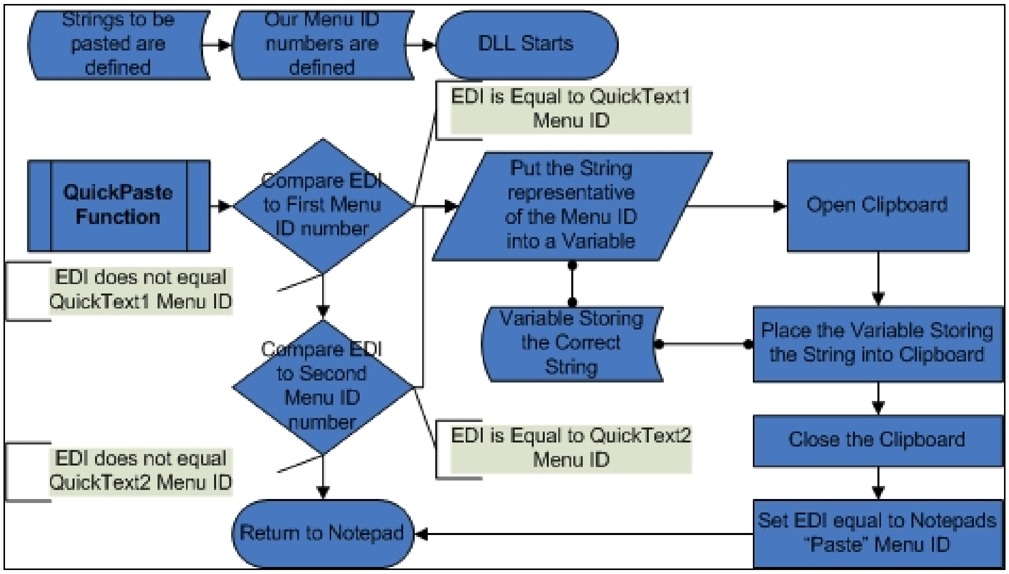

Strings to be pasted are defined
Our Menu ID numbers are defined
DLL Starts
EDI is Equal to QuickText1 Menu ID
QuickPaste Function
Compare EDI to First Menu ID number
Put the String representative of the Menu ID into a Variable
Open Clipboard
EDI does not equal QuickText1 Menu ID
Variable Storing the Correct String
Place the Variable Storing the String into Clipboard
Compare EDI to Second Menu ID number
EDI is Equal to QuickText2 Menu ID
Close the Clipboard
EDI does not equal QuickText2 Menu ID
Return to Notepad
Set EDI equal to Notepads "Paste" Menu ID

Image 1.26 – Flowchart Representation of QuickText DLL

Examining this illustration gives us a better idea of how we need to develop our DLL. The chart at the top shows how the DLL loads into memory. Below that we find the execution flow of the QuickPaste function. If you noticed I decided to use the clipboard to accomplish our goal. If the value of EDI is equal to one of our Menu ID's then we open the clipboard and place the string assigned to that Menu ID into the clipboard. Before returning to Notepad I specifically placed the value of the Paste Menu ID into EDI. When we return Notepad will execute the Paste function placing the string from the clipboard into it's own text field. That cuts down on the amount of code we have to write. Working with the clipboard in windows is not necessarily trivial, there isn't just a single API call you can use in this case. Because the clipboard can store images, text, and files it becomes a little more complicated. Below are the actions we need to take along with their corresponding Windows API functions:

Action	API Function
Allocate memory within our process to store our string	GlobalAlloc
We then need to lock the memory so Windows does not discard it.	GlobalLock
Next we move our string into the allocated memory	N/A
We can now open the clipboard to be used by Notepad	OpenClipboard
Remove anything that was formerly in the clipboard	EmptyClipboard
Set the clipboard data to TEXT and move our allocated memory into the clipboard	SetClipboardData
Now close the clipboard so other processes can use it	CloseClipboard
We then unlock our allocated memory so it can be deleted	GlobalUnlock

We have now spent some time outlining and planning how our DLL will work. We know the flow of execution and the desired functionality. We have established the API functions we will need to use, and are finally ready to begin coding our DLL.

4.2 Coding

If you are completely new to ASM programming I will briefly walk you through your assembler and IDE.[12] There are many different tutorials out there to help you learn assembly, If I were to explain the basics of the language I would be repeating what is already available. I can recommend two great series of tutorials to help you begin learning ASM[13,14]. For this paper you only need to know enough to install the IDE and assembler. If you have read and understood the flow of execution so far you will be able to follow along with the commented ASM code.

We will be using MASM32[15] as our assembler language in this paper. For our IDE you will need both RadASM v2.x and the RadASM Assembly Programming Pack. These are located at the RadASM site[16]. I recommend installing both RadASM and MASM32 to your C: directory. This will help eliminate the need to modify library and include paths in RadASM. If you have issues with setting up RadASM to work with MASM refer to the RadASM help file. It can be downloaded from their website.

The basics of writing a DLL in MASM are simple. Since we are coding our DLL using the RadASM IDE (Interactive Development Environment), it will remove some of the work when coding our assembly programs.

1. Open RadASM and select File->New Project.
2. Choose MASM for the Assembler and check DLL Project. Enter in both the Project Name and what the Project file will be called. I named my project QuickText. Press Next to continue.
3. Do not choose a Template, just press Next.
4. For File Creation only choose ASM and DEF, and choose BAK for the Folder Creation. Press Next to continue.
5. In the next window press Finish to begin your project.

When you press Finish you will be presented with a new project in RadASM. In the right hand pane of the program you will see your two files **QuickText.asm** and **Quicktext.def**. Quicktext.asm will contain the main

[12] IDE is defined as: Interactive Development Environment
[13] Iczelion's Tutorial Series: http://win32assembly.online.fr/tutorials.html
[14] Win32 Assembler Coding for Crackers by Goppit: http://tutorials.accessroot.com
[15] MASM32: http://www.masm32.com/
[16] RadASM IDE: http://www.radasm.com/

executable body of code for our DLL. Quicktext.def will contain definitions of any functions our DLL will export. The large main pane in the center of the program is the coding window where you write the assembly code.

First we need to open our QuickText.asm file in the coding window so we can begin writing the DLL. Double-Click on QuickText.asm and that will open it up in the coding window. We have spent a lot of time developing the function of this DLL and have thoroughly examined how it is going to work. We have also discovered what functions are needed to achieve our desired execution. Because of this I am not going to directly explain the QuickText code. I have commented every line and I recommend reading through it. When you have read through the code, and feel comfortable that you understand how it is functioning, you can past the source below directly into the coding window.

```
;QuickText DLL v1.0 by Gabri3l [ARTeam]
;Supplement to Adding New Functionality to Old Software
;##\
;Adds new functionality to a modified notepad.
;Allows interception of the Message handler for WM_COMMAND.
;Compares menu ID number against new modified numbers and acts accordingly
;##/

;##\ Processor definition and includes

.586
.model flat, stdcall
option casemap:none

include windows.inc
include user32.inc
include kernel32.inc
includelib user32.lib
includelib kernel32.lib

;##/

.data
wQuickText1 db "QuickText1",0Dh,0Ah,0 ;Character String to Paste into Notepad
wQuickText2 db "QuickText2",0Dh,0Ah,0 ;Character String to Paste into Notepad
dBytes dw 100h                        ;Buffer Size for Clipboard Memory

.data?
hMem dd 4 dup(?)                      ;Handle to Allocated Memory
pAlloc dd 4 dup(?)                    ;Pointer to First Byte in Allocated Memory

.code

DLLEntry proc hInstDLL:DWORD, reason:DWORD, unused:DWORD    ;*QUICKTEXT ENTRY FUNCTION*

.if reason == DLL_PROCESS_ATTACH      ;initialization code for when DLL is loaded
        mov eax,TRUE                  ;put TRUE in EAX to continue loading the DLL
.endif
        Ret                           ;Return

DLLEntry Endp                         ;*END OF QUICKTEXT ENTRY FUNCTION*

QuickPaste proc                       ;*QUICKPASTE FUNCTION*

.IF EDI==55
        Mov EDI, OFFSET wQuickText1   ;If Menu ID = 55 MOV offset of first Character
                                      ;String into EDI
.ELSEIF EDI==66
        Mov EDI, OFFSET wQuickText2   ;If Menu ID = 66 MOV offset of second Character
                                      ;String into EDI
.ELSE
        RET                           ;If Menu ID does not equal 55 or 66 Return to
Notepad
.ENDIF

INVOKE GlobalAlloc,GMEM_MOVEABLE, dBytes     ;Allocate dBytes of Memory to load Character
String
mov hMem,EAX                          ;Move the Handle of the Allocated Memory into hMem

Invoke GlobalLock,EAX                 ;Lock the Allocated Memory
```

```
mov pAlloc,EAX                       ;Move Pointer to First Byte of Allocated Memory into
pAlloc

MOV ECX,EDI                          ;Move offset of Character String into ECX
XOR EBX,EBX                          ;Zero Out EBX
Mov BL, BYTE Ptr DS:[ECX]            ;Move First Byte of Character string into BL

.While BL!=NULL                      ;Loop while BL is not a Null Character
        Mov Dword Ptr DS:[EAX],EBX   ;Move Character stored in BL into Allocated Memory
        INC EAX                      ;Increment to next Byte in Memory
        INC ECX                      ;Increment to next Byte in String
        Mov BL, BYTE Ptr DS:[ECX]    ;Move next Character into BL
.ENDW

Mov Dword Ptr DS:[EAX],00            ;Move NULL into Last Byte of Allocated Memory to end
string

INVOKE OpenClipboard,NULL            ;Open Clipboard for this Process

MOV EBX, Dword PTR DS:[hMem]         ;Move the Handle of Allocated memory into EBX

INVOKE EmptyClipboard                ;Empty old Clipboard Contents
INVOKE SetClipboardData,CF_TEXT,EBX  ;Set Clipboard Data equal to Allocated Memory
INVOKE CloseClipboard                ;Close clipboard
INVOKE GlobalUnlock,hMem             ;Unlock the Allocated Memory

MOV EDI, 302h                        ;Move "Paste" Menu ID number into EDI
ADD ESP,2                            ;Balance the Stack
RET                                  ;Return to Notepad

QuickPaste EndP                      ;*END OF QUICKPASTE FUNCTION*

end DLLEntry
```

After we have written the preceding code we need to export our QuickPaste function. This is done so other programs, like Notepad, can use it. We do this by defining the exports in the QuickText.def file. Our definition file needs to include two lines. We need to define the name of our dynamic library and we also need to declare any functions we want to export from the DLL. When a function is exported that means it is made available to any module in the address space that wants to call it. Exporting a function will allow us to find the address of that function by using the GetProcAddress API feature.

Below are the definitions to be included in QuickText.def:

```
LIBRARY   QuickText                                ;The name of our library
EXPORTS   QuickPaste                               ;The name of the exported function
```

After all the code has been entered for both QuickText.asm and QuickText.def we can build our DLL.

1. In RadASM choose Make->Build to compile the DLL
2. Output results will be displayed in a window at the bottom of the program. Your DLL will be located in the ...RadASM/MASM/Projects/QuickText/ folder

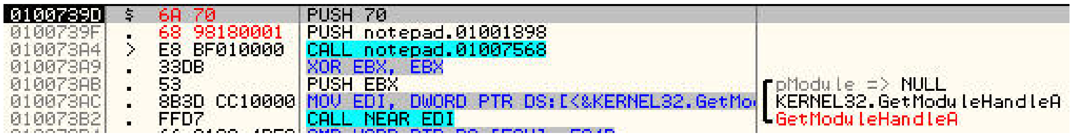
5. Modifying Notepad to Load and Use QuickText.dll

We have finally created our QuickText DLL. Now we need to modify Notepad so it will load our DLL during initialization. We are going to begin by redirecting Notepad's Entry Point to a **Code Cave**[17]. That is done by adding a JUMP at the entry point that jumps to our code cave. This jump can overwrite some commands if needed because we can emulate them within our cave. The next step will be to load QuickText.dll. This can be done by calling **LoadLibrary** and using "QuickText.dll" as the argument. LoadLibrary is a Windows API function that is used to map an executable module, like a DLL, into memory. LoadLibrary will look in the programs directory for "QuickText.dll" and, if the dll is found, it will load it into memory. LoadLibrary will then return the handle of our DLL in EAX. Once our module is loaded we need to find the location of our QuickPaste function. Windows provides another API function that allows us to do this easily. Because we exported our function in the QuickText.def file, it is listed in the exports section of our DLL. We can look it up using the **GetProcAddess** API function. Calling GetProcAddress and passing it the return of LoadLibrary (which was the handle of our DLL), and the

[17] Code Cave: Unused memory space within a programs allocated memory. It can be used to store information and code without changing the size of the original program.

argument "QuickPaste" will return the location of the QuickPaste function in EAX. We can then store that address and use it in the Menu ID comparison. After we have stored the address we need to execute any commands we overwrote with our JUMP and return to regular program execution. Finally we will need to modify the Menu ID comparison so it will call the QuickPaste function using the address we received from GetProcAddress.

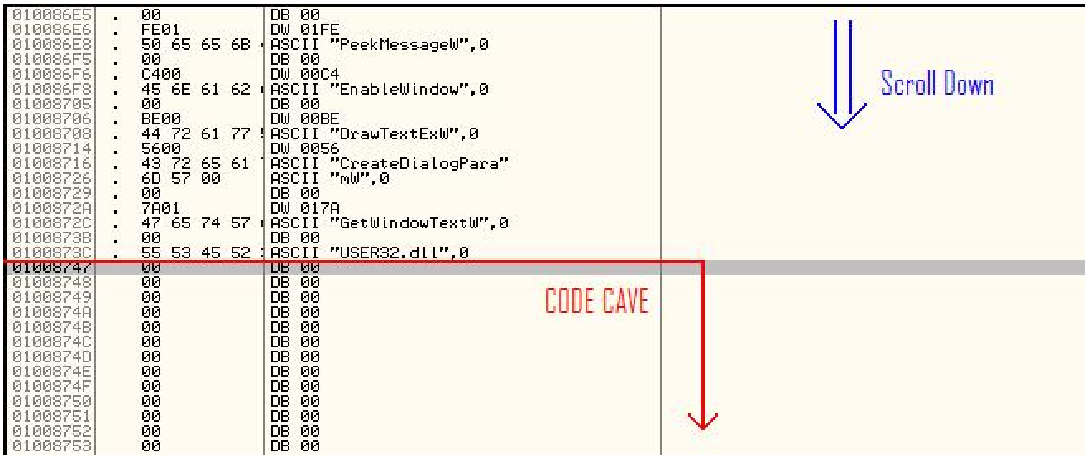
We start by first loading Notepad.exe into Ollydbg. You will find yourself here at Notepad's Entry Point:



```
0100739D  $  6A 70          PUSH 70
0100739F  .  68 98180001    PUSH notepad.01001898
010073A4  >  E8 BF010000    CALL notepad.01007568
010073A9  .  33DB           XOR EBX, EBX
010073AB  .  53             PUSH EBX                                   [pModule => NULL
010073AC  .  8B3D CC10000   MOV EDI, DWORD PTR DS:[<&KERNEL32.GetMo    KERNEL32.GetModuleHandleA
010073B2  .  FFD7           CALL NEAR EDI                              GetModuleHandleA
```

Image 1.27 – Notepad.exe Entry Point

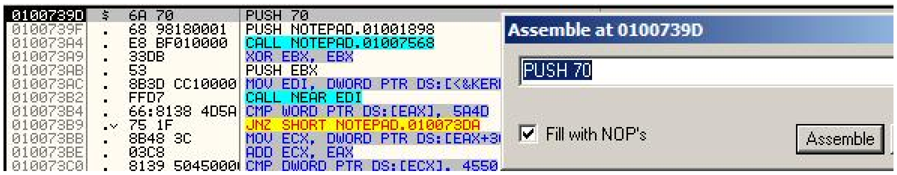
A quick trick to finding a code cave is by scrolling Olly's code window until we get to the last real instruction and just see 0's.



```
010086E5  .  00            DB 00
010086E6  .  FE01          DW 01FE
010086E8  .  50 65 65 6B   ASCII "PeekMessageW",0
010086F5  .  00            DB 00
010086F6  .  C400          DW 00C4
010086F8  .  45 6E 61 62   ASCII "EnableWindow",0
01008705  .  00            DB 00
01008706  .  BE00          DW 00BE
01008708  .  44 72 61 77   ASCII "DrawTextExW",0
01008714  .  5600          DW 0056
01008716  .  43 72 65 61   ASCII "CreateDialogPara"
01008726  .  6D 57 00      ASCII "mW",0
01008729  .  00            DB 00
0100872A  .  7A01          DW 017A
0100872C  .  47 65 74 57   ASCII "GetWindowTextW",0
0100873B  .  00            DB 00
0100873C  .  55 53 45 52   ASCII "USER32.dll",0
01008747     00            DB 00
01008748     00            DB 00
01008749     00            DB 00
0100874A     00            DB 00
0100874B     00            DB 00
0100874C     00            DB 00
0100874D     00            DB 00
0100874E     00            DB 00
0100874F     00            DB 00
01008750     00            DB 00
01008751     00            DB 00
01008752     00            DB 00
01008753     00            DB 00
```

Image 1.28 – Code Cave in Notepad.exe

We see that our code cave starts at 1008747. This section is where we are going to redirect our entry point to. Go back to Notepad's entrypoint where we are going to assemble a JUMP. Highlight the PUSH 70 line and Press SPACEBAR. This will open the assembly window which allows us to modify the programs code.



```
0100739D  $  6A 70          PUSH 70
0100739F  .  68 98180001    PUSH NOTEPAD.01001898
010073A4  .  E8 BF010000    CALL NOTEPAD.01007568
010073A9  .  33DB           XOR EBX, EBX
010073AB  .  53             PUSH EBX
010073AC  .  8B3D CC10000   MOV EDI, DWORD PTR DS:[<&KERI
010073B2  .  FFD7           CALL NEAR EDI
010073B4  .  66:8138 4D5A   CMP WORD PTR DS:[EAX], 5A4D
010073B9  .v 75 1F          JNZ SHORT NOTEPAD.010073DA
010073BB  .  8B48 3C        MOV ECX, DWORD PTR DS:[EAX+3
010073BE  .  03C8           ADD ECX, EAX
010073C0  .  8139 5045000   CMP DWORD PTR DS:[ECX], 4550
```

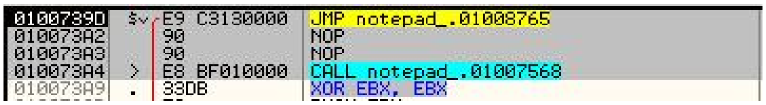
Assemble at 0100739D

PUSH 70

☑ Fill with NOP's

Image 1.29 – Ollydbg Assembly Window

I prefer to leave a little space at the beginning of the code cave just in case I need to make some modifications. So I will choose to jump to 1008765 instead of 1008747. Enter **JUMP 1008765** into the Assembly box. Make sure you check **Fill with NOP's** and press **Assemble**. You will see the JUMP has overwritten PUSH 70 and PUSH NOTEPAD.01001989 with a JUMP NOTEPAD.01008765. We have now redirected our entry point to the code cave. Your code should look like this:



```
0100739D  $v E9 C3130000   JMP notepad_.01008765
010073A2     90            NOP
010073A3     90            NOP
010073A4  >  E8 BF010000   CALL notepad_.01007568
010073A9  .  33DB          XOR EBX, EBX
```

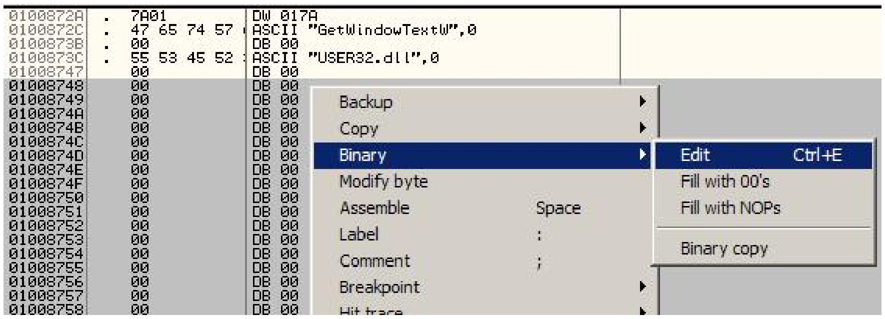
Image 1.30 – Entry Point Redirected

We now need to figure out what code we enter into our code cave. The code I post below is incorrect in it's syntax but will give you the idea of what we need to accomplish in our code cave.

```
PUSH "QuickText.dll"                    ;  PUSH POINTER TO "Quicktext.dll" ONTO THE
```

```
STACK
CALL LoadLibrary                                 ;  CALL LoadLibraryA FUNCTION
PUSH "QuickPaste"                                ;  PUSH POINTER TO "QuickPaste" ONTO THE
STACK
PUSH EAX                                         ;  PUSH THE HANDLE TO QuickText DLL RETURNED
BY LOADLIBRARY
CALL GetProcAddress                              ;  CALL GetProcAddress
MOV  Stored_Address, EAX                         ;  MOVE LOCATION OF QuickPaste FUNCTION INTO
STORED_ADDRESS
PUSH 70                                          ;  EMULATE OVERWRITTEN COMMAND
PUSH 01001898                                    ;  EMULATE OVERWRITTEN COMMAND
JMP  010073A4                                    ;  JUMP BACK TO PROGRAM EXECUTION
```

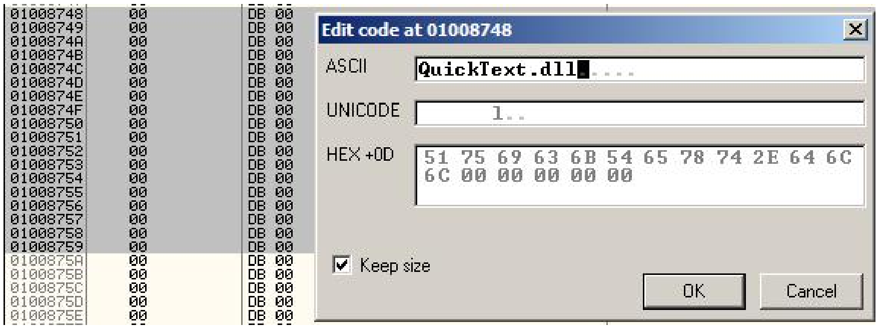
Most of this code we could enter just as it is written. However we cannot use variables and constants in our code, like "QuickText.dll", QuickPaste, and Stored_Address. Instead we have to define the variables ourselves and reference their location in memory. It is simple to do, all we do is write the strings into Notepad. When we need to reference them we just use whatever memory location they were written at. Knowing that, it is time to start adding code to our code cave. We are going to start by creating our two constants "QuickText.dll" and "QuickPaste" in Notepad. Go to the beginning of our code cave at 1008747. Highlight line 1008748 and, while holding the mouse button, down drag down to select about 15 lines below it. Once you have the lines selected, right-click and choose **Binary** and then **Edit**



```
0100872A  .  7A01          DW 017A
0100872C  .  47 65 74 57   ASCII "GetWindowTextW",0
0100873B  .  00            DB 00
0100873C  .  55 53 45 52   ASCII "USER32.dll",0
01008747     00            DB 00
01008748     00            DB 00
01008749     00            DB 00
0100874A     00            DB 00
0100874B     00            DB 00
0100874C     00            DB 00
0100874D     00            DB 00
0100874E     00            DB 00
0100874F     00            DB 00
01008750     00            DB 00
01008751     00            DB 00
01008752     00            DB 00
01008753     00            DB 00
01008754     00            DB 00
01008755     00            DB 00
01008756     00            DB 00
01008757     00            DB 00
01008758     00            DB 00
```

Image 1.31 - Binary Edit

A new window will open that allows you to edit the memory you have selected. We are going to add our first string "QuickText.dll" into our selected memory location. Type QuickText.dll into the ASCII box in the edit menu. When you are finished press Okay.



```
01008748     00            DB 00
01008749     00            DB 00
0100874A     00            DB 00
0100874B     00            DB 00
0100874C     00            DB 00
0100874D     00            DB 00
0100874E     00            DB 00
0100874F     00            DB 00
01008750     00            DB 00
01008751     00            DB 00
01008752     00            DB 00
01008753     00            DB 00
01008754     00            DB 00
01008755     00            DB 00
01008756     00            DB 00
01008757     00            DB 00
01008758     00            DB 00
01008759     00            DB 00
0100875A     00            DB 00
0100875B     00            DB 00
0100875C     00            DB 00
0100875D     00            DB 00
0100875E     00            DB 00
```

Edit code at 01008748

ASCII: QuickText.dll.....

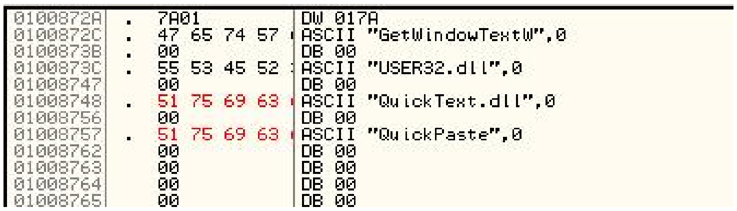
UNICODE: 1..

HEX +0D: 51 75 69 63 6B 54 65 78 74 2E 64 6C 6C 00 00 00 00 00

☑ Keep size

Image 1.32 – Entering ASCII String into Memory

Once the string has been written into memory press **CTRL+A** to re-analyze the code. Olly will then recognize the string and you will see it in your code window. We now need to do the same thing for the next string "QuickPaste". I entered the QuickPaste string starting at location 1008757, one BYTE after the QuickText.dll. Follow the steps above to enter the QuickPaste string. Your final modification should look like this:



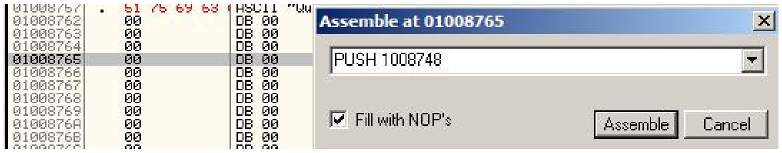
```
0100872A  .  7A01          DW 017A
0100872C  .  47 65 74 57   ASCII "GetWindowTextW",0
0100873B  .  00            DB 00
0100873C  .  55 53 45 52   ASCII "USER32.dll",0
01008747     00            DB 00
01008748  .  51 75 69 63   ASCII "QuickText.dll",0
01008756     00            DB 00
01008757  .  51 75 69 63   ASCII "QuickPaste",0
01008762     00            DB 00
01008763     00            DB 00
01008764     00            DB 00
01008765     00            DB 00
```

Image 1.33 – String Constants

We now have our two string constants written into memory, the only other thing we need to do is find a location for our variable Stored_Address, which will hold the location of the QuickPaste function. Finding a variable location is easy; all we need to do is determine an empty code location that we will write information into. In my case I chose a location farther down the code cave at **1008798**. No preparation is needed to use this location as a variable. We will just use that memory location as we enter the code into our cave. Now that we have all our constants and variables memory locations defined we can use them in our code:

```
PUSH 1008748                                  ;  PUSH POINTER TO "Quicktext.dll" ONTO THE
STACK
CALL LoadLibraryA                             ;  CALL LoadLibraryA FUNCTION
PUSH 1008757                                  ;  PUSH POINTER TO “QuickPaste” ONTO THE
STACK
PUSH EAX                                      ;  PUSH THE HANDLE TO QuickText DLL RETURNED
BY LOADLIBRARY
CALL GetProcAddress                           ;  CALL GetProcAddress
MOV  DWORD PTR DS:[1008798],EAX               ;  MOVE LOCATION OF QuickPaste FUNCTION INTO
STORED_ADDRESS
PUSH 70                                       ;  EMULATE OVERWRITTEN COMMAND
PUSH 01001898                                 ;  EMULATE OVERWRITTEN COMMAND
JMP  010073A4                                 ;  JUMP BACK TO PROGRAM EXECUTION
```

This code can now be entered directly into our code cave. We do this the same way we added the JUMP to our cave. We will assemble each line of our cave with each line of code above. Start at 1008765, select the line and Assemble. In the Assembly Box enter the first line of our code; PUSH 1008748. Press Assemble to write the code:



```
01008757  .  51 75 69 63   ASCII "Qu
01008762     00            DB 00
01008763     00            DB 00
01008764     00            DB 00
01008765     00            DB 00
01008766     00            DB 00
01008767     00            DB 00
01008768     00            DB 00
01008769     00            DB 00
0100876A     00            DB 00
0100876B     00            DB 00
```

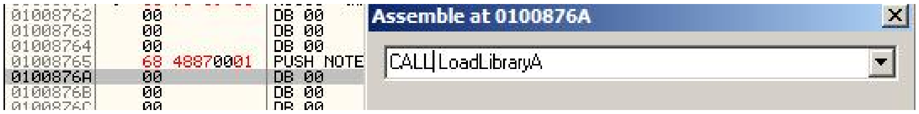
Assemble at 01008765

PUSH 1008748

☑ Fill with NOP's

Image 1.34 – Assembling Code in a Code Cave

Next we will Assemble line 100876A. Usually your Assembly box will remain open, if it is not open then just select the next empty line and press SPACE to assemble. Enter our next line of code into the Assembly box; CALL LoadLibraryA and press Assemble[18].



```
01008762     00            DB 00
01008763     00            DB 00
01008764     00            DB 00
01008765     68 48870001   PUSH NOTE
0100876A     00            DB 00
0100876B     00            DB 00
```

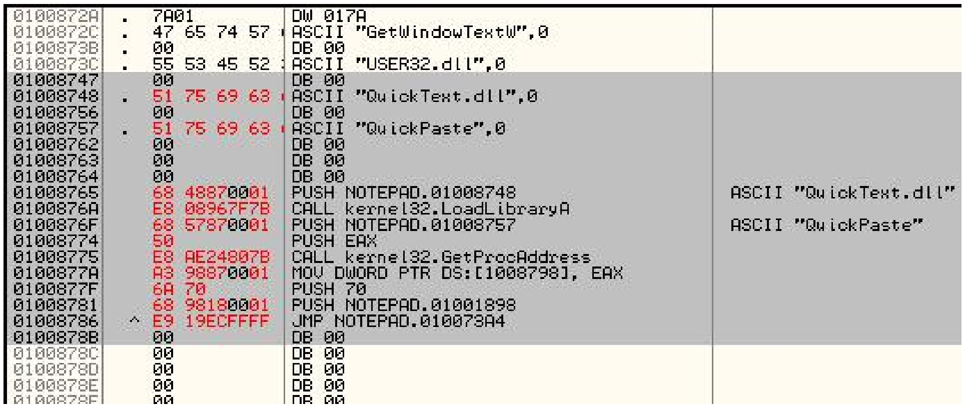
Assemble at 0100876A

CALL LoadLibraryA

Image 1.35 – Assembling Code in a Code Cave (Continued)

Continue to Assemble each line of code until you are finished with the block of code. Your final product should look like this:

[18] A note: doing this way, the LoadLibrary will be directly called; when saved it will be resolved with the address of the PC on which you assembled the code. Generally speaking it is better to do it through the import gate, otherwise will not work on different systems. We left this as a note in order to keep the discussion clear, but you should consider it.



```
0100872A  .  7A01          DW 017A
0100872C  .  47 65 74 57   ASCII "GetWindowTextW",0
0100873B  .  00            DB 00
0100873C  .  55 53 45 52   ASCII "USER32.dll",0
01008747     00            DB 00
01008748  .  51 75 69 63   ASCII "QuickText.dll",0
01008756     00            DB 00
01008757  .  51 75 69 63   ASCII "QuickPaste",0
01008762     00            DB 00
01008763     00            DB 00
01008764     00            DB 00
01008765     68 48870001   PUSH NOTEPAD.01008748                ASCII "QuickText.dll"
0100876A     E8 08967F7B   CALL kernel32.LoadLibraryA
0100876F     68 57870001   PUSH NOTEPAD.01008757                ASCII "QuickPaste"
01008774     50            PUSH EAX
01008775     E8 AE24807B   CALL kernel32.GetProcAddress
0100877A     A3 98870001   MOV DWORD PTR DS:[1008798], EAX
0100877F     6A 70         PUSH 70
01008781     68 98180001   PUSH NOTEPAD.01001898
01008786   ^ E9 19ECFFFF   JMP NOTEPAD.010073A4
0100878B     00            DB 00
0100878C     00            DB 00
0100878D     00            DB 00
0100878E     00            DB 00
0100878F     00            DB 00
```

Image 1.36 – Final Code Cave Code

We now need to save our modifications to a new executable. Right-click within the code window and choose **Copy To Executable** from the right-click menu. Then select **All Modifications.**

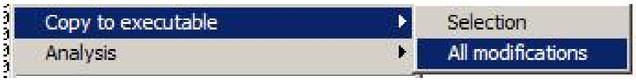


Image 1.37 – Copy To Executable

A new window will open up with a new file which has all the modifications we have made to the Notepad executable. We need to save this file as an new exe. Right-Click in the new window and choose **Save File.** A dialog box will open up asking what you would like to name the new file. You can use whatever name you desire, I named my file Notepad.Modified.exe.

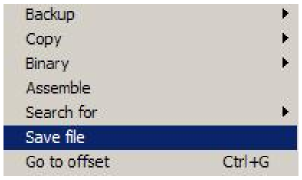
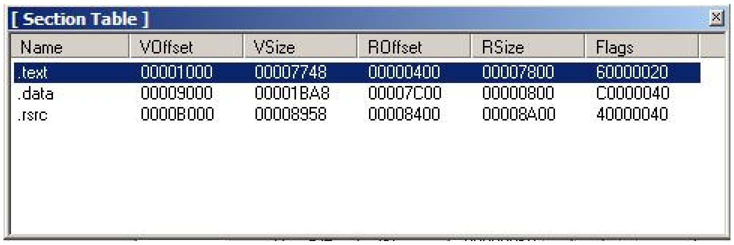


Image 38 – Save new executable to file

You can now close Ollydbg. Our Notepad is modified so it will load our QuickText.dll. It can then find the location of the QuickPaste function and store it in a variable to be used later. There is only one problem. If you attempt to run the modified Notepad you will encounter an error! Why? The reason is this command:

```
;  MOVE LOCATION OF QuickPaste FUNCTION INTO STORED_ADDRESS
MOV  DWORD PTR DS:[1008798],EAX
```

That command is writing information directly into the memory location 1008798. We are getting an error because Notepad has set a flag in it's characteristics saying that the section we located our variable in is not writeable memory. Thankfully this is an easy thing to fix. All we have to do is edit the characteristics of that section so we can write to memory. This can be accomplished with a Portable Executable editor such as LordPE[19]. Once you have downloaded and installed LordPE open it up and choose the **PE Editor** button. Locate and open your modified version of Notepad. You will be presented with the executable editor menu. Our goal is to edit the section characteristics of Notepad so we can write to memory. Select the **Sections** button to view Notepad's sections. You will be presented with a Sections Table window.

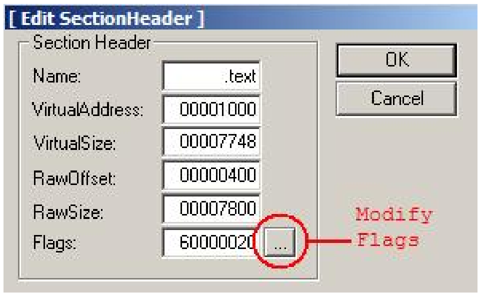


Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	00007748	00000400	00007800	60000020
.data	00009000	00001BA8	00007C00	00000800	C0000040
.rsrc	0000B000	00008958	00008400	00008A00	40000040

Image 1.39 - Sections Table in LordPE

[19] LordPE: http://mitglied.lycos.de/yoda2k/LordPE/info.htm

Right-Click on the .text row, and choose **Edit Section Header** from the Right-Click menu. This will open up a new window showing just the information for the .text section. We need to modify the flags for this section. Choose the **...** button located next to the Flags textbox:



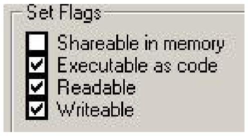
[Edit SectionHeader]

Section Header

Name: .text
VirtualAddress: 00001000
VirtualSize: 00007748
RawOffset: 00000400
RawSize: 00007800
Flags: 60000020

Image 1.40 - Modify Flags in LordPE

A new window will open where we can set individual flags for this section. Set the **Writeable** flag by checking the box next to its name.



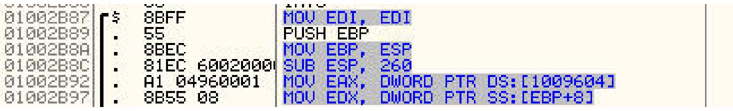
Set Flags

- ☐ Shareable in memory
- ☑ Executable as code
- ☑ Readable
- ☑ Writeable

Image 41 - Setting a Flag in LordPE

When you have set the flag press **OK** until you return to the Section Table. Close the table, and you will be find yourself back at the PE Editor. Press **SAVE** to save the modified flag to our executable. You can now close LordPE and test out Notepad. It runs perfectly now! We can move onto the final step; modifying the Menu ID number comparison.

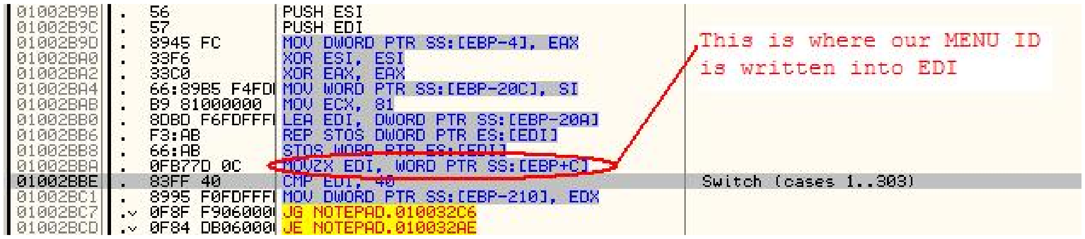
The last thing we need to do is to add a CALL to QuickPaste in the Menu ID comparison routine. We already have all the information we need to accomplish this. We know where the routine is, we are finding the location of QuickPaste and storing it in 1008798. All we need to do now is find a suitable location to place our CALL. Begin by opening your modified Notepad in Ollydbg. Go to the beginning of our Menu ID comparison located at 1002B87



```
01002B87 ┌$  8BFF          MOV EDI, EDI
01002B89 │.  55            PUSH EBP
01002B8A │.  8BEC          MOV EBP, ESP
01002B8C │.  81EC 6002000  SUB ESP, 260
01002B92 │.  A1 04960001   MOV EAX, DWORD PTR DS:[1009604]
01002B97 │.  8B55 08       MOV EDX, DWORD PTR SS:[EBP+8]
```

Image 1.42 – Beginning of Menu ID Comparison Function.

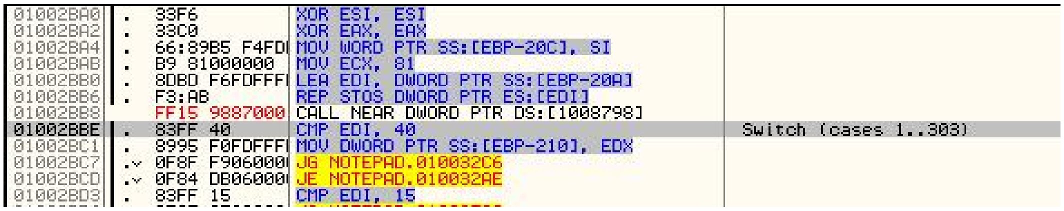
We are going to look for a suitable location to insert our CALL. We know that the Menu ID is going to be stored in EDI. So we want to get as close to the location where the Menu ID is moved into the register as we can. This will cut down on the amount of instructions we may need to emulate, because remember that if we overwrite anything we need to make sure we emulate it so our program continues to function. Scroll down in Ollydbg until we get to here:



```
01002B9B  .  56            PUSH ESI
01002B9C  .  57            PUSH EDI
01002B9D  .  8945 FC       MOV DWORD PTR SS:[EBP-4], EAX
01002BA0  .  33F6          XOR ESI, ESI
01002BA2  .  33C0          XOR EAX, EAX
01002BA4  .  66:89B5 F4FD  MOV WORD PTR SS:[EBP-20C], SI
01002BAB  .  B9 81000000   MOV ECX, 81
01002BB0  .  8DBD F6FDFFF  LEA EDI, DWORD PTR SS:[EBP-20A]
01002BB6  .  F3:AB         REP STOS DWORD PTR ES:[EDI]
01002BB8  .  66:AB         STOS WORD PTR ES:[EDI]
01002BBA  .  0FB77D 0C     MOVZX EDI, WORD PTR SS:[EBP+C]
01002BBE  .  83FF 40       CMP EDI, 40                         Switch (cases 1..303)
01002BC1  .  8995 F0FDFFF  MOV DWORD PTR SS:[EBP-210], EDX
01002BC7  .v 0F8F F906000  JG NOTEPAD.010032C6
01002BCD  .v 0F84 DB06000  JE NOTEPAD.010032AE
```

Image 1.43 – Menu ID Moved Into EDI

We see above that our Menu ID is being moved into EDI directly above a compare. We do not necessarily want to overwrite the compare function as it is critical to the switch. This does not give us much room to work with. We are just going to have to overwrite the command that move the Menu ID and the command directly above it if we are going to fit our CALL into this code. Select **STOS WORD PTR ES:[EDI]** at line 1002BB8 and press SPACE to Assemble the line. We are going to now add our CALL to the QuickPaste function. The code we are going to enter is: **CALL DWORD PTR DS:[1008798]**. That code will be calling the address location stored at 1008798, which is where we stored the QuickPaste function address. Enter the code above in the Assembly box and assemble the new CALL. Your modified code will look like this:



```
01002BA0  .  33F6           XOR ESI, ESI
01002BA2  .  33C0           XOR EAX, EAX
01002BA4  .  66:89B5 F4FDI  MOV WORD PTR SS:[EBP-20C], SI
01002BAB  .  B9 81000000    MOV ECX, 81
01002BB0  .  8DBD F6FDFFFI  LEA EDI, DWORD PTR SS:[EBP-20A]
01002BB6  .  F3:AB          REP STOS DWORD PTR ES:[EDI]
01002BB8     FF15 9887000   CALL NEAR DWORD PTR DS:[1008798]
01002BBE  .  83FF 40        CMP EDI, 40                              Switch (cases 1..303)
01002BC1  .  8995 F0FDFFFI  MOV DWORD PTR SS:[EBP-210], EDX
01002BC7  .v 0F8F F906000I  JG NOTEPAD.010032C6
01002BCD  .v 0F84 DB06000I  JE NOTEPAD.010032AE
01002BD3  .  83FF 15        CMP EDI, 15
```

Image 1.44 – Modified Menu ID Routine

Now that the modification has been made to the Menu ID comparison function we need to save it to an executable. Follow the same steps we took before to copy all our modifications made in Ollydbg to a new executable. You can name this final executable whatever you desire. I named my modified Notepad; Notepad.Final.exe. We can now close out OllyDbg. We are finished modifying Notepad, we added new menus, we modified it so it will load our QuickText dll during initialization, and finally we added a CALL that will redirect the function that compares the Menu ID to our own QuickPaste function. With our modified Notepad complete, we only have one more step to take.

6. QuickText DLL Revisited

We need to make a quick modification to the QuickText DLL. In Notepad we added the CALL to the QuickPaste function by overwriting some commands. We need to emulate those same commands within the QuickPaste function to ensure stability. Open up your QuickText DLL project in RadASM. Navigate down to the beginning of the QuickPaste function and add the two highlighted lines below:

```
...
DLLEntry Endp                         ;*END OF QUICKTEXT ENTRY FUNCTION*

QuickPaste proc                       ;*QUICKPASTE FUNCTION*
STOS WORD PTR ES:[EDI]                ;Emulate replaced Notepad function
MOVZX EDI, WORD PTR SS:[EBP+0Ch]      ;Emulate replaced Notepad function

.IF EDI==55
        Mov EDI, OFFSET wQuickText1   ;If Menu ID = 55 MOV offset of first Character
                                      ;String into EDI
.ELSEIF EDI==66
...
```

Build the DLL the same way we did before:

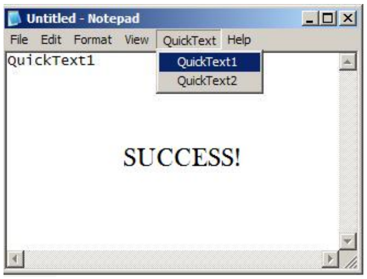
1. In RadASM choose Make->Build to compile the DLL
2. Output results will be displayed in a window at the bottom of the program. Your DLL will be located in the ...RadASM/MASM/Projects/QuickText/ folder

After the new QuickText DLL is built, be sure to copy it into the same folder as our Notepad.Final so it can be loaded by Notepad. Now that we have emulated the replaced functions we are ready to move onto the testing of our new Notepad.

7. Results &Final Remarks

We have completed all the work that needed to be done to on our quest to add a new feature to Notepad. If we applied the steps correctly we will be able to add predefined text by simply choosing an option from our menu. It is finally time to test our newly created Notepad functionality.

1. Locate Notepad.Final.exe
2. Verify that QuickText.dll is in the same folder as Notepad.Final.exe
3. Run the program...



QuickText1

SUCCESS!

Image 1.45 – Modified Notepad Running

The project was completed successfully, but that is not the end of the journey. You have the option to build on this information by adding new menu items and changing the features they provide, creating a custom program all on your own through editing more resources and menu's, or maybe enhancing the portability by making the dll read from a file; allowing others to change the defined text. It is true that this topic has been covered many times. Hopefully the application of new tools and a different approach made the paper feel new and useful. I tried my best to put a lot of new information and strong explanation into this paper.

2. Patching by using resource, ThunderPwr of ARTeam

1. Introduction

The aim of this paper is to show another way of patching one application and keep it registered by using the information gathered from application resource.

Target used for this essay was DLL to Lib, a nice application useful for converting a DLL library into a static library.

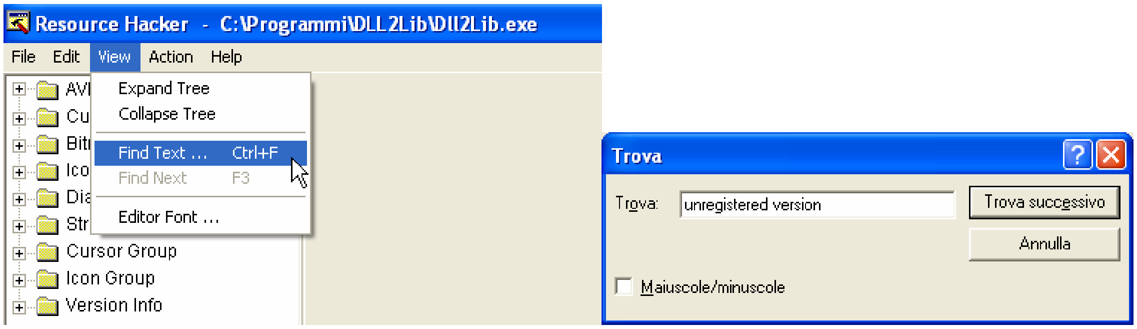
Target Name: DLL to LIB v1.42

Target URL: http://www.binary-soft.com/dll2lib/dll2lib.htm

Since this isn't a full cracking tutorial I'll not cover the unpacking stage, I've done it manually in a very simple way (AsProtect 1.2 / 1.2c-> Alexey Solodovnikov). Take care by using Stripper v2.07f your PC may reboot, as a hint reach OEP and use ImportREC, show invalid and use Trace 1, now you've only one unresolved API, keep it by using the Disassemble/HexView function and is easy to show that is was the GetProcAddress API.

2. Resource searching

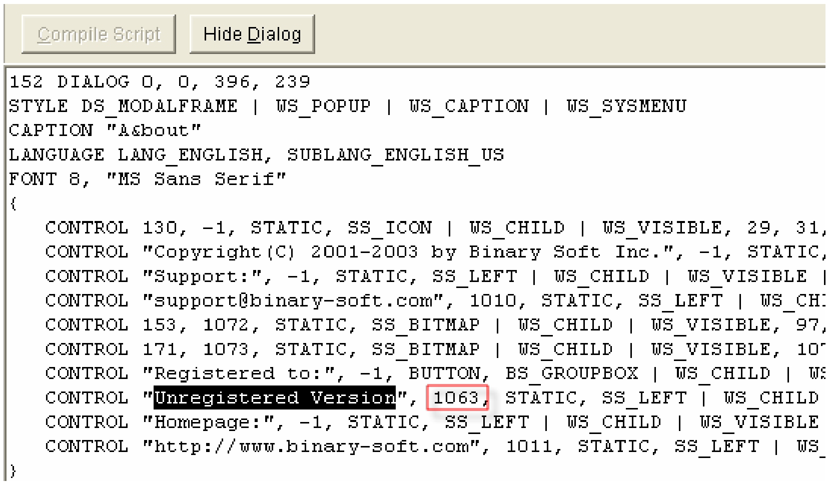
Start ResHacker and perform a text search for the string "Unregistered Version" with ResHacker:



Trova: unregistered version

□ Maiuscole/minuscole

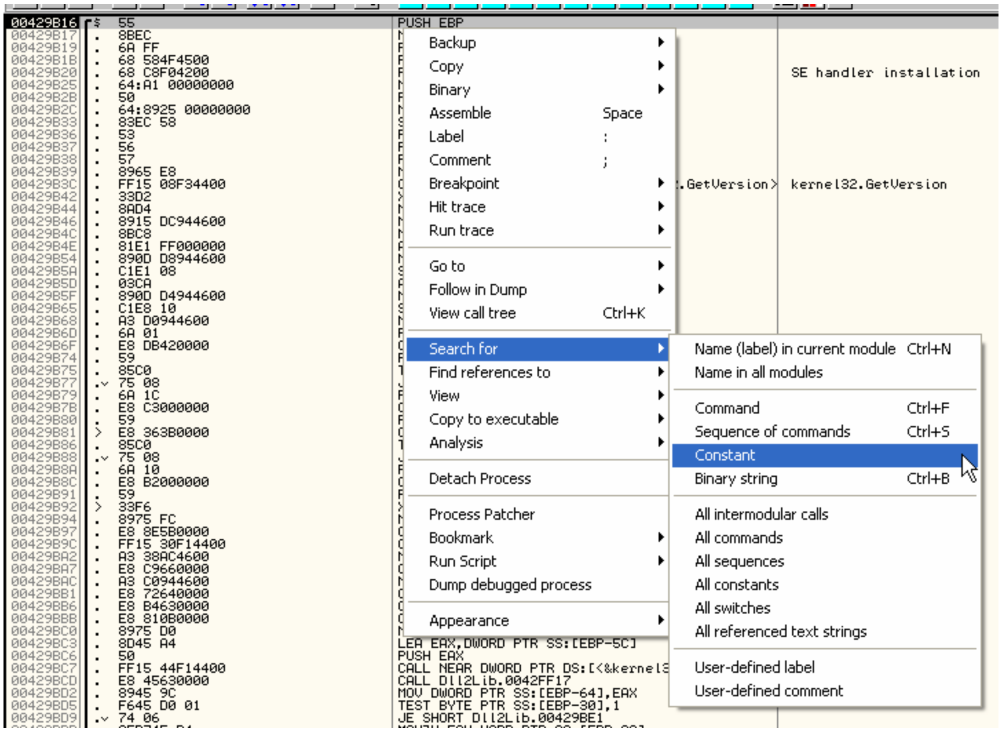
Image 2.1 – ResHacker



```
152 DIALOG 0, 0, 396, 239
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "A&bout"
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
FONT 8, "MS Sans Serif"
{
   CONTROL 130, -1, STATIC, SS_ICON | WS_CHILD | WS_VISIBLE, 29, 31,
   CONTROL "Copyright(C) 2001-2003 by Binary Soft Inc.", -1, STATIC,
   CONTROL "Support:", -1, STATIC, SS_LEFT | WS_CHILD | WS_VISIBLE |
   CONTROL "support@binary-soft.com", 1010, STATIC, SS_LEFT | WS_CHI
   CONTROL 153, 1072, STATIC, SS_BITMAP | WS_CHILD | WS_VISIBLE, 97,
   CONTROL 171, 1073, STATIC, SS_BITMAP | WS_CHILD | WS_VISIBLE, 10
   CONTROL "Registered to:", -1, BUTTON, BS_GROUPBOX | WS_CHILD | WS
   CONTROL "Unregistered Version", 1063, STATIC, SS_LEFT | WS_CHILD
   CONTROL "Homepage:", -1, STATIC, SS_LEFT | WS_CHILD | WS_VISIBLE
   CONTROL "http://www.binary-soft.com", 1011, STATIC, SS_LEFT | WS_
}
```

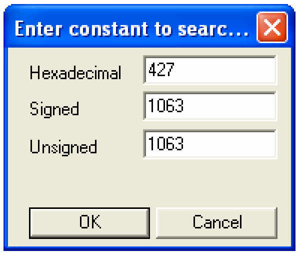
Image 2.2 – Resource inspection with ResHacker

Image 2.2 shows that the needed resource ID is equal to 1063 in decimal, 0x427 hexadecimal. Load the target in OllyDbg and perform a search for all constants equal to 0x427.



```
00429B16 ┌$  55                 PUSH EBP
00429B17 .   8BEC
00429B19 .   6A FF
00429B1B .   68 584F4500
00429B20 .   68 C8F04200                                        SE handler installation
00429B25 .   64:A1 00000000
00429B2B .   50
00429B2C .   64:8925 00000000
00429B33 .   83EC 58
00429B36 .   53
00429B37 .   56
00429B38 .   57
00429B39 .   8965 E8
00429B3C .   FF15 08F34400                         .GetVersion> kernel32.GetVersion
00429B42 .   33D2
00429B44 .   8AD4
00429B46 .   8915 DC944600
00429B4C .   8BC8
00429B4E .   81E1 FF000000
00429B54 .   890D D8944600
00429B5A .   C1E1 08
00429B5D .   03CA
00429B5F .   890D D4944600
00429B65 .   C1E8 10
00429B68 .   A3 D0944600
00429B6D .   6A 01
00429B6F .   E8 DB420000
00429B74 .   59
00429B75 .   85C0
00429B77 .v  75 08
00429B79 .   6A 1C
00429B7B .   E8 C3000000
00429B80 .   59
00429B81 >   E8 363B0000
00429B86 .   85C0
00429B88 .v  75 08
00429B8A .   6A 10
00429B8C .   E8 B2000000
00429B91 .   59
00429B92 >   33F6
00429B94 .   8975 FC
00429B97 .   E8 8E5B0000
00429B9C .   FF15 30F14400
00429BA2 .   A3 38AC4600
00429BA7 .   E8 C9660000
00429BAC .   A3 C0944600
00429BB1 .   E8 72640000
00429BB6 .   E8 B4630000
00429BBB .   E8 810B0000
00429BC0 .   8975 D0
00429BC3 .   8D45 A4              LEA EAX,DWORD PTR SS:[EBP-5C]
00429BC6 .   50                   PUSH EAX
00429BC7 .   FF15 44F14400        CALL NEAR DWORD PTR DS:[<&kernel3
00429BCD .   E8 45630000          CALL Dll2Lib.0042FF17
00429BD2 .   8945 9C              MOV DWORD PTR SS:[EBP-64],EAX
00429BD5 .   F645 D0 01           TEST BYTE PTR SS:[EBP-30],1
00429BD9 .v  74 06                JE SHORT Dll2Lib.00429BE1
```

Image 2.3 – search the resource into OllyDbg



Enter constant to searc…

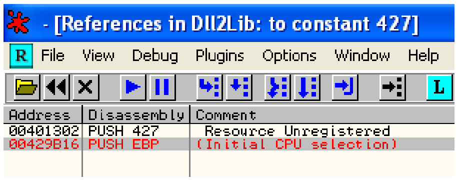
Hexadecimal 427

Signed 1063

Unsigned 1063

Image 2.4 – OllyDbg constant search form

Press Ok:

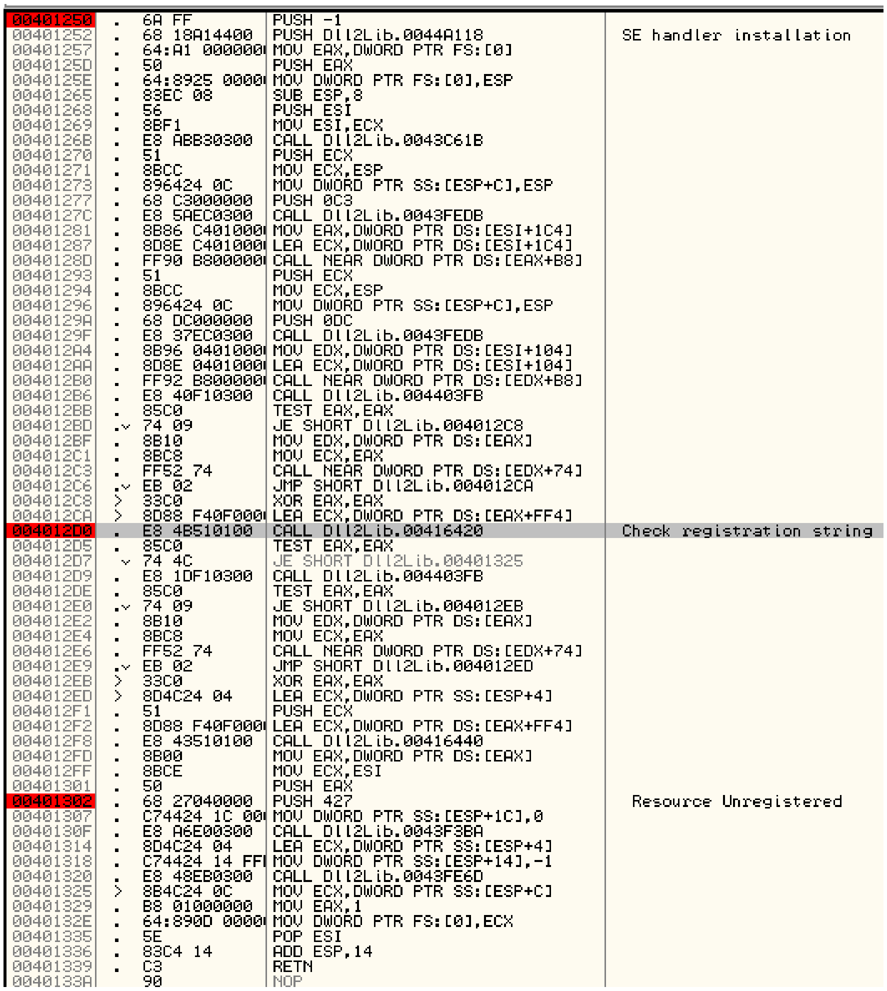


- [References in Dll2Lib: to constant 427]

```
Address  Disassembly Comment
00401302 PUSH 427     Resource Unregistered
00429B16 PUSH EBP    (Initial CPU selection)
```

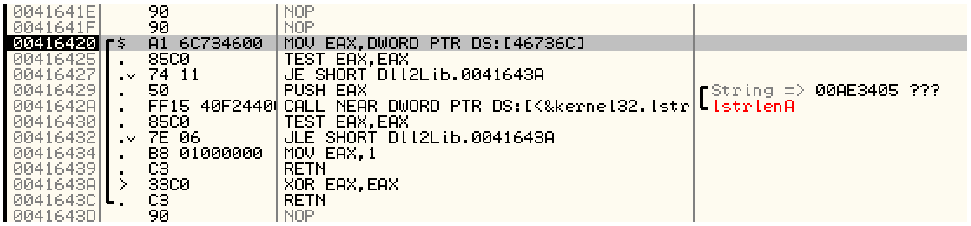
Image 2.5 – search result

Press Ctrl+G and write 0x00401302, we're into this code:



```
00401250  .  6A FF          PUSH -1
00401252  .  68 18A14400    PUSH Dll2Lib.0044A118                  SE handler installation
00401257  .  64:A1 000000   MOV EAX,DWORD PTR FS:[0]
0040125D  .  50             PUSH EAX
0040125E  .  64:8925 0000   MOV DWORD PTR FS:[0],ESP
00401265  .  83EC 08        SUB ESP,8
00401268  .  56             PUSH ESI
00401269  .  8BF1           MOV ESI,ECX
0040126B  .  E8 ABB30300    CALL Dll2Lib.0043C61B
00401270  .  51             PUSH ECX
00401271  .  8BCC           MOV ECX,ESP
00401273  .  896424 0C      MOV DWORD PTR SS:[ESP+C],ESP
00401277  .  68 C3000000    PUSH 0C3
0040127C  .  E8 5AEC0300    CALL Dll2Lib.0043FEDB
00401281  .  8B86 C401000   MOV EAX,DWORD PTR DS:[ESI+1C4]
00401287  .  8D8E C401000   LEA ECX,DWORD PTR DS:[ESI+1C4]
0040128D  .  FF90 B800000   CALL NEAR DWORD PTR DS:[EAX+B8]
00401293  .  51             PUSH ECX
00401294  .  8BCC           MOV ECX,ESP
00401296  .  896424 0C      MOV DWORD PTR SS:[ESP+C],ESP
0040129A  .  68 DC000000    PUSH 0DC
0040129F  .  E8 37EC0300    CALL Dll2Lib.0043FEDB
004012A4  .  8B96 0401000   MOV EDX,DWORD PTR DS:[ESI+104]
004012AA  .  8D8E 0401000   LEA ECX,DWORD PTR DS:[ESI+104]
004012B0  .  FF92 B800000   CALL NEAR DWORD PTR DS:[EDX+B8]
004012B6  .  E8 40F10300    CALL Dll2Lib.004403FB
004012BB  .  85C0           TEST EAX,EAX
004012BD  .v 74 09          JE SHORT Dll2Lib.004012C8
004012BF  .  8B10           MOV EDX,DWORD PTR DS:[EAX]
004012C1  .  8BC8           MOV ECX,EAX
004012C3  .  FF52 74        CALL NEAR DWORD PTR DS:[EDX+74]
004012C6  .v EB 02          JMP SHORT Dll2Lib.004012CA
004012C8  >  33C0           XOR EAX,EAX
004012CA  >  8D88 F40F000   LEA ECX,DWORD PTR DS:[EAX+FF4]
004012D0  .  E8 4B510100    CALL Dll2Lib.00416420                  Check registration string
004012D5  .  85C0           TEST EAX,EAX
004012D7   v 74 4C          JE SHORT Dll2Lib.00401325
004012D9  .  E8 1DF10300    CALL Dll2Lib.004403FB
004012DE  .  85C0           TEST EAX,EAX
004012E0  .v 74 09          JE SHORT Dll2Lib.004012EB
004012E2  .  8B10           MOV EDX,DWORD PTR DS:[EAX]
004012E4  .  8BC8           MOV ECX,EAX
004012E6  .  FF52 74        CALL NEAR DWORD PTR DS:[EDX+74]
004012E9  .v EB 02          JMP SHORT Dll2Lib.004012ED
004012EB  >  33C0           XOR EAX,EAX
004012ED  >  8D4C24 04      LEA ECX,DWORD PTR SS:[ESP+4]
004012F1  .  51             PUSH ECX
004012F2  .  8D88 F40F000   LEA ECX,DWORD PTR DS:[EAX+FF4]
004012F8  .  E8 43510100    CALL Dll2Lib.00416440
004012FD  .  8B00           MOV EAX,DWORD PTR DS:[EAX]
004012FF  .  8BCE           MOV ECX,ESI
00401301  .  50             PUSH EAX
00401302  .  68 27040000    PUSH 427                                Resource Unregistered
00401307  .  C74424 1C 00   MOV DWORD PTR SS:[ESP+1C],0
0040130F  .  E8 A6E00300    CALL Dll2Lib.0043F3BA
00401314  .  8D4C24 04      LEA ECX,DWORD PTR SS:[ESP+4]
00401318  .  C74424 14 FF   MOV DWORD PTR SS:[ESP+14],-1
00401320  .  E8 48EB0300    CALL Dll2Lib.0043FE6D
00401325  >  8B4C24 0C      MOV ECX,DWORD PTR SS:[ESP+C]
00401329  .  B8 01000000    MOV EAX,1
0040132E  .  64:890D 0000   MOV DWORD PTR FS:[0],ECX
00401335  .  5E             POP ESI
00401336  .  83C4 14        ADD ESP,14
00401339  .  C3             RETN
0040133A     90             NOP
```

Now place the software breakpoints (F2) showed above and restart the target by using Ctrl+F2 then press Shift+F9, reach the address 0x004012D0 and enter into the call code (F7):



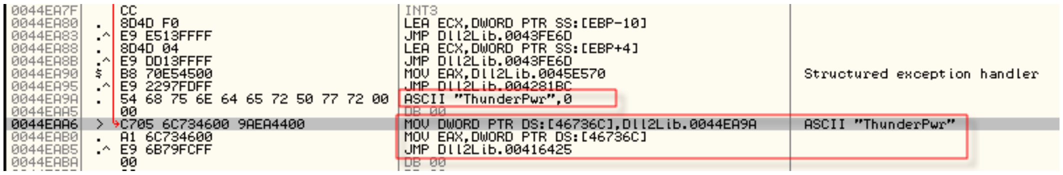
```
0041641E     90            NOP
0041641F     90            NOP
00416420   $ A1 6C734600   MOV EAX,DWORD PTR DS:[46736C]
00416425   . 85C0          TEST EAX,EAX
00416427   .v 74 11        JE SHORT Dll2Lib.0041643A
00416429   . 50            PUSH EAX                                  String => 00AE3405 ???
0041642A   . FF15 40F2440  CALL NEAR DWORD PTR DS:[<&kernel32.lstr   lstrlenA
00416430   . 85C0          TEST EAX,EAX
00416432   .v 7E 06        JLE SHORT Dll2Lib.0041643A
00416434   . B8 01000000   MOV EAX,1
00416439   . C3            RETN
0041643A   > 33C0          XOR EAX,EAX
0041643C   . C3            RETN
0041643D     90            NOP
```

Target was search for the registration name, if no valid one will be found the function return with EAX=0 and then the PUSH 427 will be executed then our registration name will be Unregistered Version, ok now we've got the trick. To defeat the protection system and keep the target registered simply put in [46736C] a pointer to a valid string.

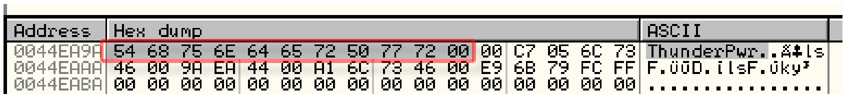
3. Patching

Restart the target and put a memory breakpoint on access on 0x0046736C, this is to make sure if the same code will be called during startup to check if the target was registered or not, press Shift+F9.

OllyDbg on start and before show the main window will break into the same routine, now scroll down to the end of the file to look for some free space and write the patch below:

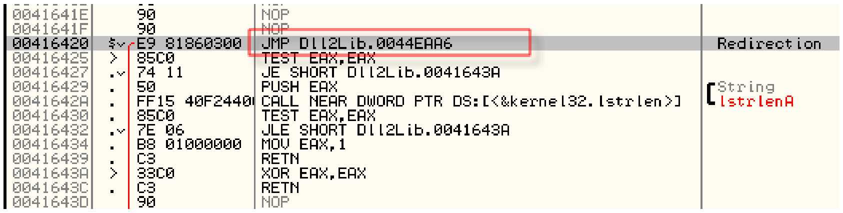


```
0044EA7F     CC                               INT3
0044EA80   . 8D4D F0                          LEA ECX,DWORD PTR SS:[EBP-10]
0044EA83   .^ E9 E513FFFF                     JMP Dll2Lib.0043FE6D
0044EA88   . 8D4D 04                          LEA ECX,DWORD PTR SS:[EBP+4]
0044EA8B   .^ E9 DD13FFFF                     JMP Dll2Lib.0043FE6D
0044EA90   $ B8 70E54500                      MOV EAX,Dll2Lib.0045E570             Structured exception handler
0044EA95   .^ E9 2297FDFF                     JMP Dll2Lib.004281BC
0044EA9A   . 54 68 75 6E 64 65 72 50 77 72 00 ASCII "ThunderPwr",0
0044EAA5     00                               DB 00
0044EAA6   > C705 6C734600 9AEA4400           MOV DWORD PTR DS:[46736C],Dll2Lib.0044EA9A   ASCII "ThunderPwr"
0044EAB0   . A1 6C734600                      MOV EAX,DWORD PTR DS:[46736C]
0044EAB5   .^ E9 6B79FCFF                     JMP Dll2Lib.00416425
0044EABA     00                               DB 00
```



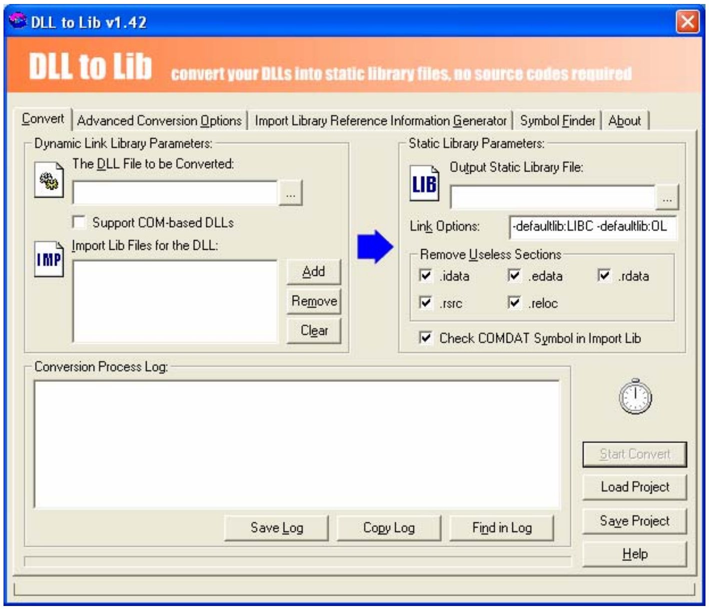
```
Address   Hex dump                                          ASCII
0044EA9A  54 68 75 6E 64 65 72 50 77 72 00 00 C7 05 6C 73  ThunderPwr..Ç♣ls
0044EAAA  46 00 9A EA 44 00 A1 6C 73 46 00 E9 6B 79 FC FF  F.ÜÛD.¡lsF.ûky³
0044EABA  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Also apply the redirection to our cave:



```
0041641E     90            NOP
0041641F     90            NOP
00416420   $v E9 81860300  JMP Dll2Lib.0044EAA6                      Redirection
00416425   > 85C0          TEST EAX,EAX
00416427   .v 74 11        JE SHORT Dll2Lib.0041643A
00416429   . 50            PUSH EAX                                  String
0041642A   . FF15 40F2440  CALL NEAR DWORD PTR DS:[<&kernel32.lstrlen>]   lstrlenA
00416430   . 85C0          TEST EAX,EAX
00416432   .v 7E 06        JLE SHORT Dll2Lib.0041643A
00416434   . B8 01000000   MOV EAX,1
00416439   . C3            RETN
0041643A   > 33C0          XOR EAX,EAX
0041643C   . C3            RETN
0041643D     90            NOP
```

Now run the target and look at the window caption:


DLL to Lib v1.42
DLL to Lib convert your DLLs into static library files, no source codes required
Convert
Advanced Conversion Options
Import Library Reference Information Generator
Symbol Finder
About
Dynamic Link Library Parameters:
The DLL File to be Converted:
...
IMP
Add
Remove
Clear
Static Library Parameters:
Output Static Library File:
LIB
...
Remove Useless Sections
Conversion Process Log:
Start Convert
Load Project
Save Project
Help
Save Log
Copy Log
Find in Log

☐ Support COM-based DLLs

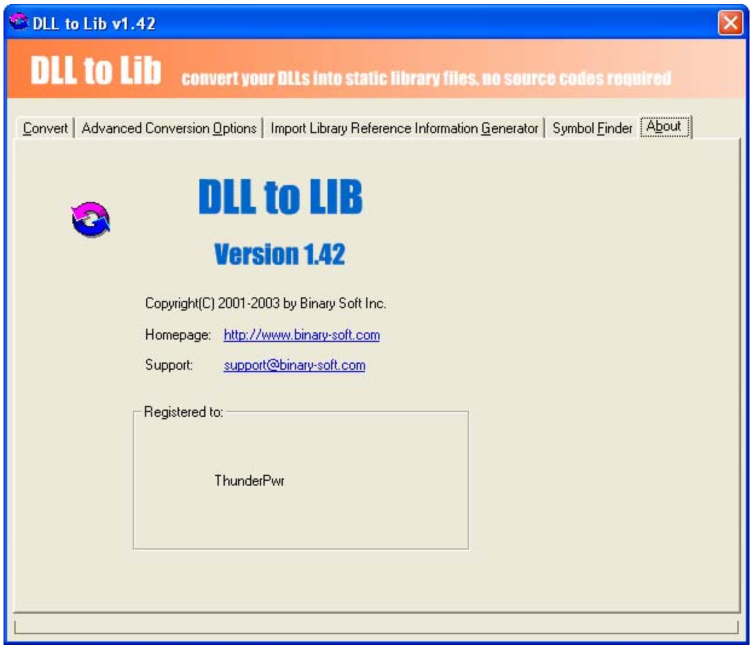
Import Lib Files for the DLL:

Link Options: -defaultlib:LIBC -defaultlib:OL

☑ .idata ☑ .edata ☑ .rdata
☑ .rsrc ☑ .reloc

☑ Check COMDAT Symbol in Import Lib

Also look into the About menu:


DLL to Lib v1.42
DLL to Lib convert your DLLs into static library files, no source codes required
Convert
Advanced Conversion Options
Import Library Reference Information Generator
Symbol Finder
About
DLL to LIB
Version 1.42

Copyright(C) 2001-2003 by Binary Soft Inc.

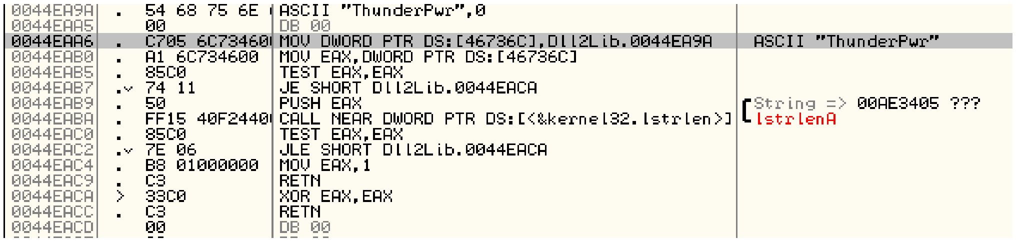
Homepage: http://www.binary-soft.com

Support: support@binary-soft.com

Registered to:

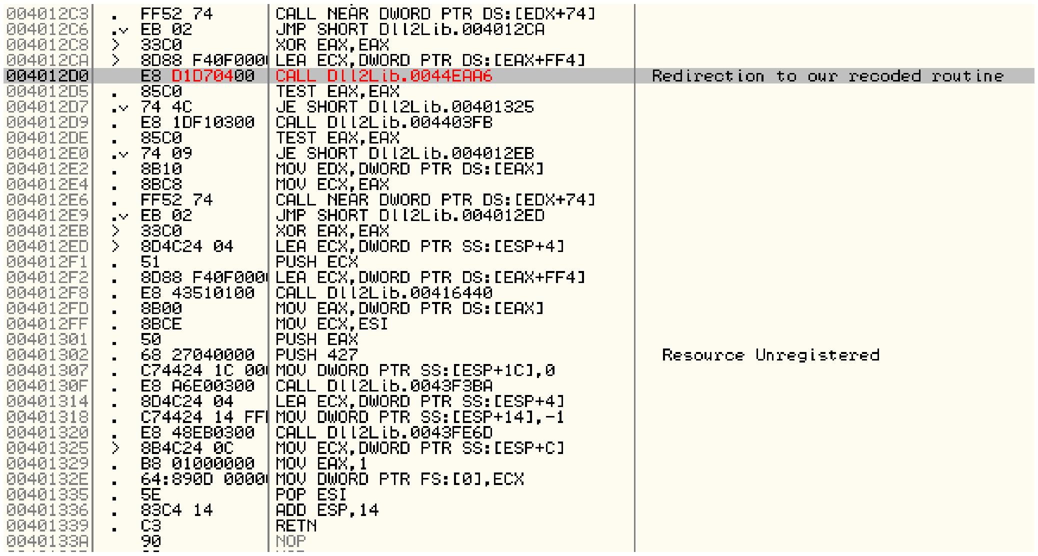
ThunderPwr

Ok but this isn't working, if you try to convert a dll target will crash, reason is about the reusing of this code for another task: we need to patch all the code from the original call.
To do it apply this patch instead the previous one:



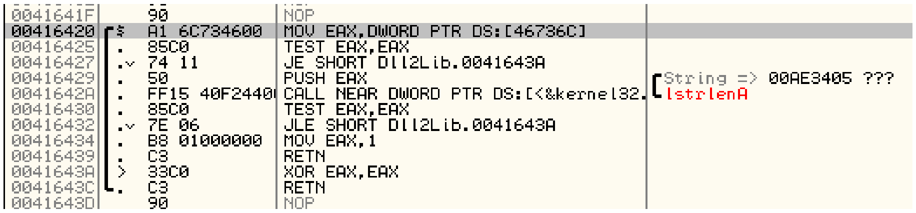
```
0044EA9A  .  54 68 75 6E  ASCII "ThunderPwr",0
0044EAA5     00           DB 00
0044EAA6  .  C705 6C73460 MOV DWORD PTR DS:[46736C],Dll2Lib.0044EA9A     ASCII "ThunderPwr"
0044EAB0  .  A1 6C734600  MOV EAX,DWORD PTR DS:[46736C]
0044EAB5  .  85C0         TEST EAX,EAX
0044EAB7  .v 74 11        JE SHORT Dll2Lib.0044EACA
0044EAB9  .  50           PUSH EAX                                      [String => 00AE3405 ???
0044EABA  .  FF15 40F2440 CALL NEAR DWORD PTR DS:[<&kernel32.lstrlen>]  lstrlenA
0044EAC0  .  85C0         TEST EAX,EAX
0044EAC2  .v 7E 06        JLE SHORT Dll2Lib.0044EACA
0044EAC4  .  B8 01000000  MOV EAX,1
0044EAC9  .  C3           RETN
0044EACA  >  33C0         XOR EAX,EAX
0044EACC  .  C3           RETN
0044EACD     00           DB 00
```

Above code is the original one but this time I have forced the registration string.
Now change also the redirection from the registration checking from 0x004012D0 then:



```
004012C3  .   FF52 74        CALL NEAR DWORD PTR DS:[EDX+74]
004012C6  .v  EB 02          JMP SHORT Dll2Lib.004012CA
004012C8  >   33C0           XOR EAX,EAX
004012CA  >   8D88 F40F000   LEA ECX,DWORD PTR DS:[EAX+FF4]
004012D0      E8 D1D70400    CALL Dll2Lib.0044EAA6              Redirection to our recoded routine
004012D5  .   85C0           TEST EAX,EAX
004012D7  .v  74 4C          JE SHORT Dll2Lib.00401325
004012D9  .   E8 1DF10300    CALL Dll2Lib.004403FB
004012DE  .   85C0           TEST EAX,EAX
004012E0  .v  74 09          JE SHORT Dll2Lib.004012EB
004012E2  .   8B10           MOV EDX,DWORD PTR DS:[EAX]
004012E4  .   8BC8           MOV ECX,EAX
004012E6  .   FF52 74        CALL NEAR DWORD PTR DS:[EDX+74]
004012E9  .v  EB 02          JMP SHORT Dll2Lib.004012ED
004012EB  >   33C0           XOR EAX,EAX
004012ED  >   8D4C24 04      LEA ECX,DWORD PTR SS:[ESP+4]
004012F1  .   51             PUSH ECX
004012F2  .   8D88 F40F000   LEA ECX,DWORD PTR DS:[EAX+FF4]
004012F8  .   E8 43510100    CALL Dll2Lib.00416440
004012FD  .   8B00           MOV EAX,DWORD PTR DS:[EAX]
004012FF  .   8BCE           MOV ECX,ESI
00401301  .   50             PUSH EAX
00401302  .   68 27040000    PUSH 427                            Resource Unregistered
00401307  .   C74424 1C 00   MOV DWORD PTR SS:[ESP+1C],0
0040130F  .   E8 A6E00300    CALL Dll2Lib.0043F3BA
00401314  .   8D4C24 04      LEA ECX,DWORD PTR SS:[ESP+4]
00401318  .   C74424 14 FF   MOV DWORD PTR SS:[ESP+14],-1
00401320  .   E8 48EB0300    CALL Dll2Lib.0043FE6D
00401325  >   8B4C24 0C      MOV ECX,DWORD PTR SS:[ESP+C]
00401329  .   B8 01000000    MOV EAX,1
0040132E  .   64:890D 0000   MOV DWORD PTR FS:[0],ECX
00401335  .   5E             POP ESI
00401336  .   83C4 14        ADD ESP,14
00401339  .   C3             RETN
0040133A      90             NOP
```

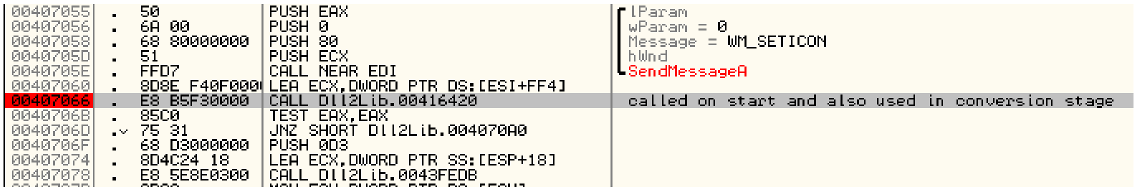
and also restore the original code:



```
0041641F      90             NOP
00416420  $   A1 6C734600    MOV EAX,DWORD PTR DS:[46736C]
00416425  .   85C0           TEST EAX,EAX
00416427  .v  74 11          JE SHORT Dll2Lib.0041643A
00416429  .   50             PUSH EAX                            String => 00AE3405 ???
0041642A  .   FF15 40F2440   CALL NEAR DWORD PTR DS:[<&kernel32. lstrlenA
00416430  .   85C0           TEST EAX,EAX
00416432  .v  7E 06          JLE SHORT Dll2Lib.0041643A
00416434  .   B8 01000000    MOV EAX,1
00416439  .   C3             RETN
0041643A  >   33C0           XOR EAX,EAX
0041643C  .   C3             RETN
0041643D      90             NOP
```

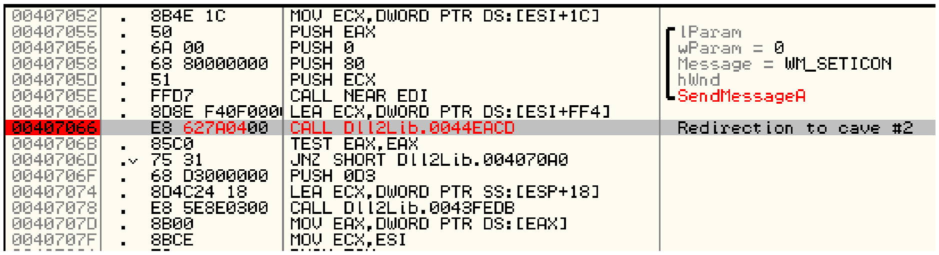
This patch forces the About box and the target to be registered but we have also to patch the original call for the caption window then simply put a breakpoint on 0x00416420 and run the target when OllyDbg break simply reach the RETN instruction and press F8 to go into the caller code.

Now apply the patch below:



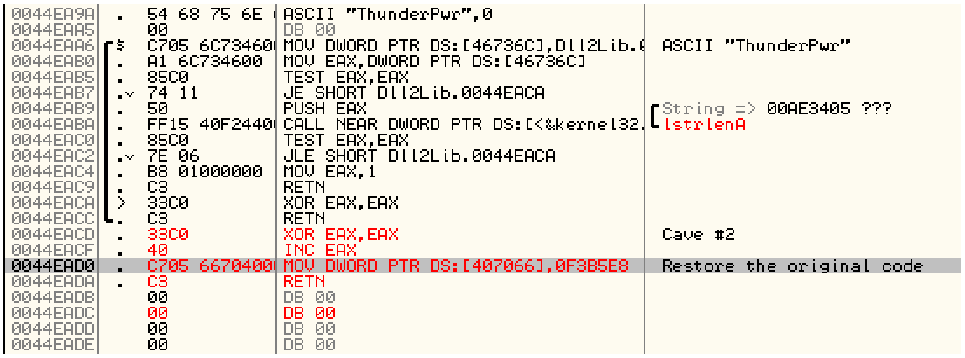
```
00407055  .   50             PUSH EAX                            lParam
00407056  .   6A 00          PUSH 0                              wParam = 0
00407058  .   68 80000000    PUSH 80                             Message = WM_SETICON
0040705D  .   51             PUSH ECX                            hWnd
0040705E  .   FFD7           CALL NEAR EDI                       SendMessageA
00407060  .   8D8E F40F000   LEA ECX,DWORD PTR DS:[ESI+FF4]
00407066  .   E8 B5F30000    CALL Dll2Lib.00416420               called on start and also used in conversion stage
0040706B  .   85C0           TEST EAX,EAX
0040706D  .v  75 31          JNZ SHORT Dll2Lib.004070A0
0040706F  .   68 D3000000    PUSH 0D3
00407074  .   8D4C24 18      LEA ECX,DWORD PTR SS:[ESP+18]
00407078  .   E8 5E8E0300    CALL Dll2Lib.0043FEDB
```

our routine is called on start and also into the converting process then we can't simply patch the redirection to our cave because for other task it crash the target, to solve in a simple way the trouble we can patch the call to another cave, execute the patched code and then patch the original redirection to go into the old call, let me explain better, first patch the call with a redirection to our new cave:



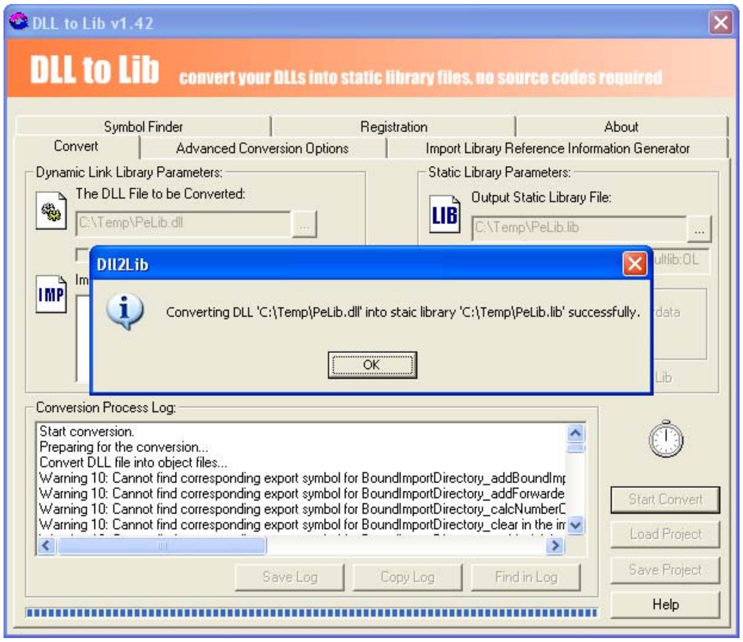
```
00407052  .   8B4E 1C        MOV ECX,DWORD PTR DS:[ESI+1C]
00407055  .   50             PUSH EAX                            lParam
00407056  .   6A 00          PUSH 0                              wParam = 0
00407058  .   68 80000000    PUSH 80                             Message = WM_SETICON
0040705D  .   51             PUSH ECX                            hWnd
0040705E  .   FFD7           CALL NEAR EDI                       SendMessageA
00407060  .   8D8E F40F000   LEA ECX,DWORD PTR DS:[ESI+FF4]
00407066      E8 627A0400    CALL Dll2Lib.0044EACD               Redirection to cave #2
0040706B  .   85C0           TEST EAX,EAX
0040706D  .v  75 31          JNZ SHORT Dll2Lib.004070A0
0040706F  .   68 D3000000    PUSH 0D3
00407074  .   8D4C24 18      LEA ECX,DWORD PTR SS:[ESP+18]
00407078  .   E8 5E8E0300    CALL Dll2Lib.0043FEDB
0040707D  .   8B00           MOV EAX,DWORD PTR DS:[EAX]
0040707F  .   8BCE           MOV ECX,ESI
```

then write into the cave the following code:



```
0044EA9A    .   54 68 75 6E ( ASCII "ThunderPwr",0
0044EAA5        00            DB 00
0044EAA6   r$   C705 6C73460( MOV DWORD PTR DS:[46736C],Dll2Lib.(  ASCII "ThunderPwr"
0044EAB0   |.   A1 6C734600   MOV EAX,DWORD PTR DS:[46736C]
0044EAB5   |.   85C0          TEST EAX,EAX
0044EAB7   |.v  74 11         JE SHORT Dll2Lib.0044EACA
0044EAB9   |.   50            PUSH EAX                            rString => 00AE3405 ???
0044EABA   |.   FF15 40F2440( CALL NEAR DWORD PTR DS:[<&kernel32.  LlstrlenA
0044EAC0   |.   85C0          TEST EAX,EAX
0044EAC2   |.v  7E 06         JLE SHORT Dll2Lib.0044EACA
0044EAC4   |.   B8 01000000   MOV EAX,1
0044EAC9   |.   C3            RETN
0044EACA   |>   33C0          XOR EAX,EAX
0044EACC   L.   C3            RETN
0044EACD    .   33C0          XOR EAX,EAX                          Cave #2
0044EACF    .   40            INC EAX
0044EAD0    .   C705 6670400( MOV DWORD PTR DS:[407066],0F3B5E8    Restore the original code
0044EADA    .   C3            RETN
0044EADB        00            DB 00
0044EADC        00            DB 00
0044EADD        00            DB 00
0044EADE        00            DB 00
```

In this way the first time we keep on execution with the patched code and for all the remainder time is executed the original code, more simple but effective!


DLL to Lib v1.42
DLL to Lib
convert your DLLs into static library files, no source codes required
Symbol Finder
Registration
About
Convert
Advanced Conversion Options
Import Library Reference Information Generator
Dynamic Link Library Parameters:
The DLL File to be Converted:
C:\Temp\PeLib.dll
Static Library Parameters:
Output Static Library File:
LIB
C:\Temp\PeLib.lib
IMP
Dll2Lib
OK
Start Convert
Load Project
Save Project
Help
Save Log
Copy Log
Find in Log

Converting DLL 'C:\Temp\PeLib.dll' into staic library 'C:\Temp\PeLib.lib' successfully.

Conversion Process Log:

```
Start conversion.
Preparing for the conversion...
Convert DLL file into object files...
Warning 10: Cannot find corresponding export symbol for BoundImportDirectory_addBoundImp
Warning 10: Cannot find corresponding export symbol for BoundImportDirectory_addForwarde
Warning 10: Cannot find corresponding export symbol for BoundImportDirectory_calcNumberC
Warning 10: Cannot find corresponding export symbol for BoundImportDirectory_clear in the im
```

Well now we can said all done!

4. Final Remarks

This paper shown how to use resources to bypass protections, also I pointed some details about code patching.

3. Patching Event Driven Nags, Shub-Nigurrath of ARTeam

1. Abstract

This is a little tutorial about a method, well known indeed, of patching event driven nags.
The target is Back2Life for TC where TC stands for Total Commander. This version of Back2Life is a file system plugin for TC that unerases files found on the hard disks and drives generally, more or less like the full program Back2Life, still of the same company.

Target Name: B2L4TC 2.33
URL: http://www.grandutils.com/Back2Life4TC/

An event driven nag is a dialog not created with a simple MessageBox, but a complex dialog with its own message pump and different events, handling the nag itself. Example of these nags are nags with a running timeout inside, or animated nags, or just nags that are meant to be a little more difficult to be removed. The fact is that message pumps are on the one hand very simple to program and on the other hand, not so simple to follow and are always a tedious task.. Most of the times moreover, like this one, the messages also initializes parts of the programs checked later as an anti-tampering countermeasure.

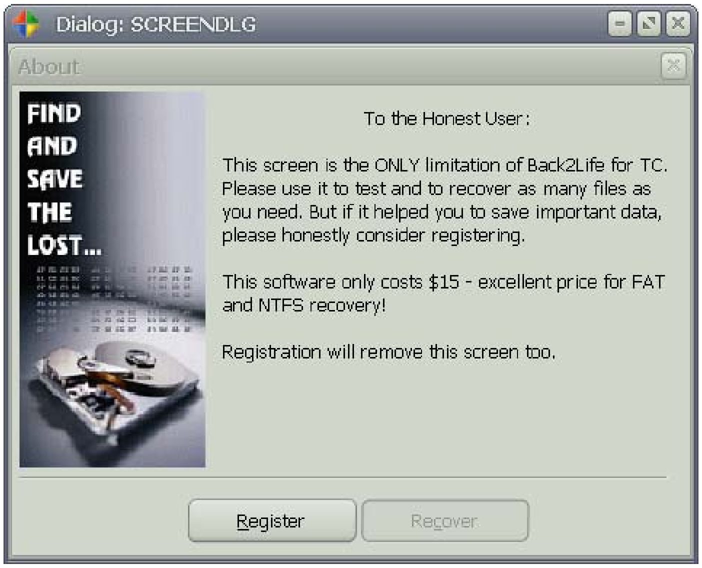
2. Analyzing the target

The protection is quite simple as the author states, just to discourage illegal use of the program: a nag with a countdown counter of 15 seconds after which a "Recover" button is activated. The program, a dll renamed with extension .wlx as requested by TC, is packed with UPX and has no CRC checks that can complicate things. It is programmed in Delphi

There is one specific issue anyway; the dialog initializes different program variables and internal function pointers into different places. Its creation involves several variables and function pointers required to correctly run the application after the nag has been shown. Just skipping the nag won't make the program running. It's a clever nag..

Further analyzing the nag code we can see that the 14 seconds countdown is done setting 14 times a 1000 msec (1 sec) timer with a call to SetTimer.

Starting from the most distant point, open a resource editor and see which the resource responsible of the nag is. You will easily recognize that it is the SCREENDLG dialog, which contains the text and a hidden field which will contain the countdown message of the trial.



To the Honest User:

This screen is the ONLY limitation of Back2Life for TC. Please use it to test and to recover as many files as you need. But if it helped you to save important data, please honestly consider registering.

This software only costs $15 - excellent price for FAT and NTFS recovery!

Registration will remove this screen too.

Image 3.1 SCREENDLG resource

As I already said, just skipping the nag like anyone might have thought won't work: the program in this case has three different behaviors, depending where you skipped the dialog:

1. Wont' recover the deleted files
2. Recovers the deleted files but doesn't place into the recovered file any data (the file will be "empty").
3. Hangs because there are some function pointers not correctly initialized.

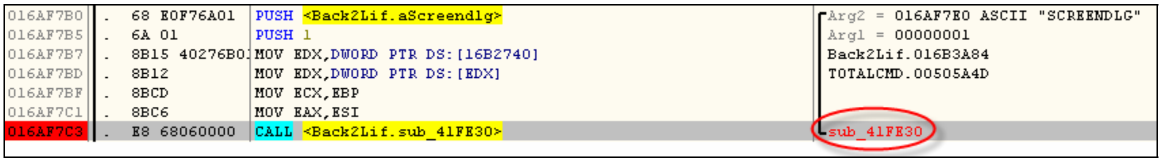
Definitely a cleaver dialog!

3. Approach and analysis

I thought of a different approach: rather than going into the program digging where it fixes all the required function callbacks and variables I will force it to work exactly as it works when you wait the timeout and press "Proceed" button; I am going to automate the dialog. The approach is general enough to be applied to any other dialog you want to "automate" and allows you to not loose your time searching where the program fill the check variables.

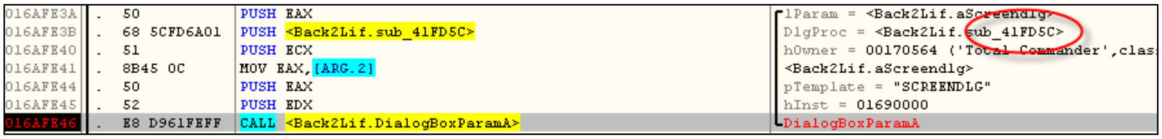
As I said the SCREENDLG is the resource involved with this dialog. Disassembling with the help of IDA the plug-in allows us to export from IDA the .MAP file and import it into OllyDbg, so as the Delphi specific things will be clear even from OllyDbg[20].

The dialog is used here:



```
016AF7B0  .  68 E0F76A01    PUSH <Back2Lif.aScreendlg>            ┌Arg2 = 016AF7E0 ASCII "SCREENDLG"
016AF7B5  .  6A 01          PUSH 1                                │Arg1 = 00000001
016AF7B7  .  8B15 40276B01  MOV EDX,DWORD PTR DS:[16B2740]        │Back2Lif.016B3A84
016AF7BD  .  8B12           MOV EDX,DWORD PTR DS:[EDX]            │TOTALCMD.00505A4D
016AF7BF  .  8BCD           MOV ECX,EBP                           │
016AF7C1  .  8BC6           MOV EAX,ESI                           │
016AF7C3  .  E8 68060000    CALL <Back2Lif.sub_41FE30>            └sub_41FE30
```

and the **sub_41FE30** contains this body:

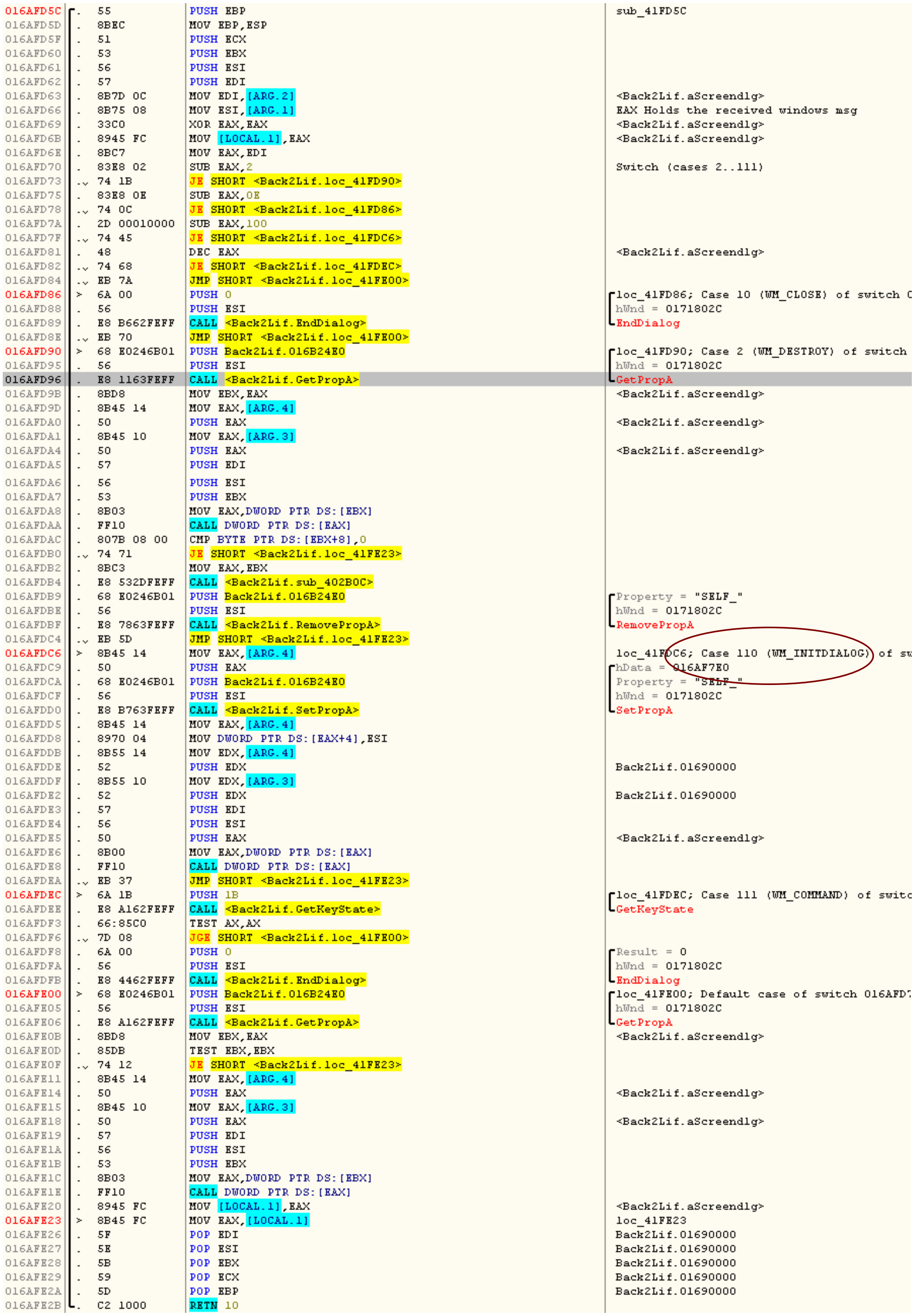


```
016AFE3A  .  50             PUSH EAX                              ┌lParam = <Back2Lif.aScreendlg>
016AFE3B  .  68 5CFD6A01    PUSH <Back2Lif.sub_41FD5C>            │DlgProc = <Back2Lif.sub_41FD5C>
016AFE40  .  51             PUSH ECX                              │hOwner = 00170564 ('Total Commander',clas
016AFE41  .  8B45 0C        MOV EAX,[ARG.2]                       │<Back2Lif.aScreendlg>
016AFE44  .  50             PUSH EAX                              │pTemplate = "SCREENDLG"
016AFE45  .  52             PUSH EDX                              │hInst = 01690000
016AFE46  .  E8 D961FEFF    CALL <Back2Lif.DialogBoxParamA>       └DialogBoxParamA
```

Where, clearly, a call to DialogBoxParamA is the call that creates the dialogbox. The important thing is then, not to skip this call, but to follow where the messages sent to the dialogbox, are handled.
First of all place a Breakpoint to the call of DialogBoxParamA and you will immediately see (to see it you must know how to use the plugin inside TC, and debug the whole TC with OllyDbg) that the DlgProc passed is **Back2Lif.sub_41FD5C**.

The DlgProc is the following one:

[20] I did this to allow reading more efficiently Delphi code into OllyDbg, as explained into several tutorials of ours



```
016AFD5C ┌.  55            PUSH EBP                                 sub_41FD5C
016AFD5D │.  8BEC          MOV EBP,ESP
016AFD5F │.  51            PUSH ECX
016AFD60 │.  53            PUSH EBX
016AFD61 │.  56            PUSH ESI
016AFD62 │.  57            PUSH EDI
016AFD63 │.  8B7D 0C       MOV EDI,[ARG.2]                          <Back2Lif.aScreendlg>
016AFD66 │.  8B75 08       MOV ESI,[ARG.1]                          EAX Holds the received windows msg
016AFD69 │.  33C0          XOR EAX,EAX                              <Back2Lif.aScreendlg>
016AFD6B │.  8945 FC       MOV [LOCAL.1],EAX                        <Back2Lif.aScreendlg>
016AFD6E │.  8BC7          MOV EAX,EDI
016AFD70 │.  83E8 02       SUB EAX,2                                Switch (cases 2..111)
016AFD73 │.v 74 1B         JE SHORT <Back2Lif.loc_41FD90>
016AFD75 │.  83E8 0E       SUB EAX,0E
016AFD78 │.v 74 0C         JE SHORT <Back2Lif.loc_41FD86>
016AFD7A │.  2D 00010000   SUB EAX,100
016AFD7F │.v 74 45         JE SHORT <Back2Lif.loc_41FDC6>
016AFD81 │.  48            DEC EAX                                  <Back2Lif.aScreendlg>
016AFD82 │.v 74 68         JE SHORT <Back2Lif.loc_41FDEC>
016AFD84 │.v EB 7A         JMP SHORT <Back2Lif.loc_41FE00>
016AFD86 │>  6A 00         PUSH 0                                  ┌loc_41FD86; Case 10 (WM_CLOSE) of switch 0
016AFD88 │.  56            PUSH ESI                                │hWnd = 0171802C
016AFD89 │.  E8 B662FEFF   CALL <Back2Lif.EndDialog>               └EndDialog
016AFD8E │.v EB 70         JMP SHORT <Back2Lif.loc_41FE00>
016AFD90 │>  68 E0246B01   PUSH Back2Lif.016B24E0                  ┌loc_41FD90; Case 2 (WM_DESTROY) of switch
016AFD95 │.  56            PUSH ESI                                │hWnd = 0171802C
016AFD96 │.  E8 1163FEFF   CALL <Back2Lif.GetPropA>                └GetPropA
016AFD9B │.  8BD8          MOV EBX,EAX                              <Back2Lif.aScreendlg>
016AFD9D │.  8B45 14       MOV EAX,[ARG.4]
016AFDA0 │.  50            PUSH EAX                                 <Back2Lif.aScreendlg>
016AFDA1 │.  8B45 10       MOV EAX,[ARG.3]
016AFDA4 │.  50            PUSH EAX                                 <Back2Lif.aScreendlg>
016AFDA5 │.  57            PUSH EDI
016AFDA6 │.  56            PUSH ESI
016AFDA7 │.  53            PUSH EBX
016AFDA8 │.  8B03          MOV EAX,DWORD PTR DS:[EBX]
016AFDAA │.  FF10          CALL DWORD PTR DS:[EAX]
016AFDAC │.  807B 08 00    CMP BYTE PTR DS:[EBX+8],0
016AFDB0 │.v 74 71         JE SHORT <Back2Lif.loc_41FE23>
016AFDB2 │.  8BC3          MOV EAX,EBX
016AFDB4 │.  E8 532DFEFF   CALL <Back2Lif.sub_402B0C>
016AFDB9 │.  68 E0246B01   PUSH Back2Lif.016B24E0                  ┌Property = "SELF_"
016AFDBE │.  56            PUSH ESI                                │hWnd = 0171802C
016AFDBF │.  E8 7863FEFF   CALL <Back2Lif.RemovePropA>             └RemovePropA
016AFDC4 │.v EB 5D         JMP SHORT <Back2Lif.loc_41FE23>
016AFDC6 │>  8B45 14       MOV EAX,[ARG.4]                          loc_41FDC6; Case 110 (WM_INITDIALOG) of sw
016AFDC9 │.  50            PUSH EAX                                ┌hData = 016AF7E0
016AFDCA │.  68 E0246B01   PUSH Back2Lif.016B24E0                  │Property = "SELF_"
016AFDCF │.  56            PUSH ESI                                │hWnd = 0171802C
016AFDD0 │.  E8 B763FEFF   CALL <Back2Lif.SetPropA>                └SetPropA
016AFDD5 │.  8B45 14       MOV EAX,[ARG.4]
016AFDD8 │.  8970 04       MOV DWORD PTR DS:[EAX+4],ESI
016AFDDB │.  8B55 14       MOV EDX,[ARG.4]
016AFDDE │.  52            PUSH EDX                                 Back2Lif.01690000
016AFDDF │.  8B55 10       MOV EDX,[ARG.3]
016AFDE2 │.  52            PUSH EDX                                 Back2Lif.01690000
016AFDE3 │.  57            PUSH EDI
016AFDE4 │.  56            PUSH ESI
016AFDE5 │.  50            PUSH EAX                                 <Back2Lif.aScreendlg>
016AFDE6 │.  8B00          MOV EAX,DWORD PTR DS:[EAX]
016AFDE8 │.  FF10          CALL DWORD PTR DS:[EAX]
016AFDEA │.v EB 37         JMP SHORT <Back2Lif.loc_41FE23>
016AFDEC │>  6A 1B         PUSH 1B                                 ┌loc_41FDEC; Case 111 (WM_COMMAND) of switc
016AFDEE │.  E8 A162FEFF   CALL <Back2Lif.GetKeyState>             └GetKeyState
016AFDF3 │.  66:85C0       TEST AX,AX
016AFDF6 │.v 7D 08         JGE SHORT <Back2Lif.loc_41FE00>
016AFDF8 │.  6A 00         PUSH 0                                  ┌Result = 0
016AFDFA │.  56            PUSH ESI                                │hWnd = 0171802C
016AFDFB │.  E8 4462FEFF   CALL <Back2Lif.EndDialog>               └EndDialog
016AFE00 │>  68 E0246B01   PUSH Back2Lif.016B24E0                  ┌loc_41FE00; Default case of switch 016AFD7
016AFE05 │.  56            PUSH ESI                                │hWnd = 0171802C
016AFE06 │.  E8 A162FEFF   CALL <Back2Lif.GetPropA>                └GetPropA
016AFE0B │.  8BD8          MOV EBX,EAX                              <Back2Lif.aScreendlg>
016AFE0D │.  85DB          TEST EBX,EBX
016AFE0F │.v 74 12         JE SHORT <Back2Lif.loc_41FE23>
016AFE11 │.  8B45 14       MOV EAX,[ARG.4]
016AFE14 │.  50            PUSH EAX                                 <Back2Lif.aScreendlg>
016AFE15 │.  8B45 10       MOV EAX,[ARG.3]
016AFE18 │.  50            PUSH EAX                                 <Back2Lif.aScreendlg>
016AFE19 │.  57            PUSH EDI
016AFE1A │.  56            PUSH ESI
016AFE1B │.  53            PUSH EBX
016AFE1C │.  8B03          MOV EAX,DWORD PTR DS:[EBX]
016AFE1E │.  FF10          CALL DWORD PTR DS:[EAX]
016AFE20 │.  8945 FC       MOV [LOCAL.1],EAX                        <Back2Lif.aScreendlg>
016AFE23 │>  8B45 FC       MOV EAX,[LOCAL.1]                        loc_41FE23
016AFE26 │.  5F            POP EDI                                  Back2Lif.01690000
016AFE27 │.  5E            POP ESI                                  Back2Lif.01690000
016AFE28 │.  5B            POP EBX                                  Back2Lif.01690000
016AFE29 │.  59            POP ECX                                  Back2Lif.01690000
016AFE2A │.  5D            POP EBP                                  Back2Lif.01690000
016AFE2B └.  C2 1000       RETN 10
```

The structure is the classical switch.case, where different cases handle different messages. What is important for us is the WM_INIT_DIALOG message handler:

```
;  loc_41FDC6; Case 110 (WM_INITDIALOG) of switch 013BFD70
```

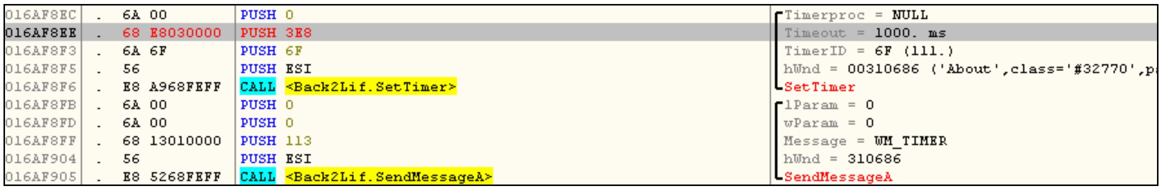
and especially the call [EAX] that jumps out of the DlgProc.

```
016AFDE8  |.  FF10     CALL DWORD PTR DS:[EAX]       ;  <Back2Lif.sub_41F7EC>
```

I then placed a BP to this location and saw where the program is jumping, which is the value of EAX. This is the real message handler of the dialogbox. The previous code is just code stubs placed by the compiler. What the programmer wrote starts at the call we just identified.

The program calls the call at 016AF7EC which contains another switch-case to handle windows messages.

Remember that we are following the WM_INITDIALOG, we will concentrate on the corresponding case:

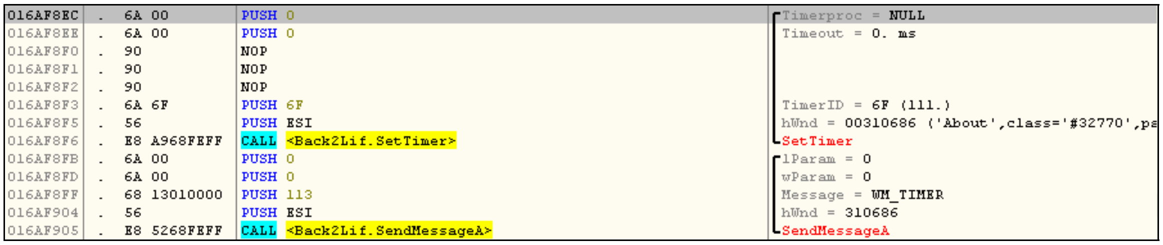


```
016AF8EC  .  6A 00        PUSH 0                              ┌Timerproc = NULL
016AF8EE  .  68 E8030000  PUSH 3E8                            │Timeout = 1000. ms
016AF8F3  .  6A 6F        PUSH 6F                             │TimerID = 6F (111.)
016AF8F5  .  56           PUSH ESI                            │hWnd = 00310686 ('About',class='#32770',p
016AF8F6  .  E8 A968FEFF  CALL <Back2Lif.SetTimer>            └SetTimer
016AF8FB  .  6A 00        PUSH 0                              ┌lParam = 0
016AF8FD  .  6A 00        PUSH 0                              │wParam = 0
016AF8FF  .  68 13010000  PUSH 113                            │Message = WM_TIMER
016AF904  .  56           PUSH ESI                            │hWnd = 310686
016AF905  .  E8 5268FEFF  CALL <Back2Lif.SendMessageA>        └SendMessageA
```

As you can see here there's a call to SetTimer with a 100 ms seconds timeout, this is one of the 14 timers we discussed before (for a total of 14 secondss). Just after SetTimer there's a call to SendMessage which sends a WM_TIMER message in order to let the application immediately handle the timer.

The patch number 1, consists in fixing this SetTimer call and setting the timer's timeout to 0 msec, like the following:

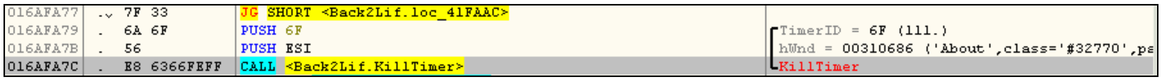
Patch 1:



```
016AF8EC  .  6A 00        PUSH 0                              ┌Timerproc = NULL
016AF8EE  .  6A 00        PUSH 0                              │Timeout = 0. ms
016AF8F0  .  90           NOP                                 │
016AF8F1  .  90           NOP                                 │
016AF8F2  .  90           NOP                                 │
016AF8F3  .  6A 6F        PUSH 6F                             │TimerID = 6F (111.)
016AF8F5  .  56           PUSH ESI                            │hWnd = 00310686 ('About',class='#32770',ps
016AF8F6  .  E8 A968FEFF  CALL <Back2Lif.SetTimer>            └SetTimer
016AF8FB  .  6A 00        PUSH 0                              ┌lParam = 0
016AF8FD  .  6A 00        PUSH 0                              │wParam = 0
016AF8FF  .  68 13010000  PUSH 113                            │Message = WM_TIMER
016AF904  .  56           PUSH ESI                            │hWnd = 310686
016AF905  .  E8 5268FEFF  CALL <Back2Lif.SendMessageA>        └SendMessageA
```

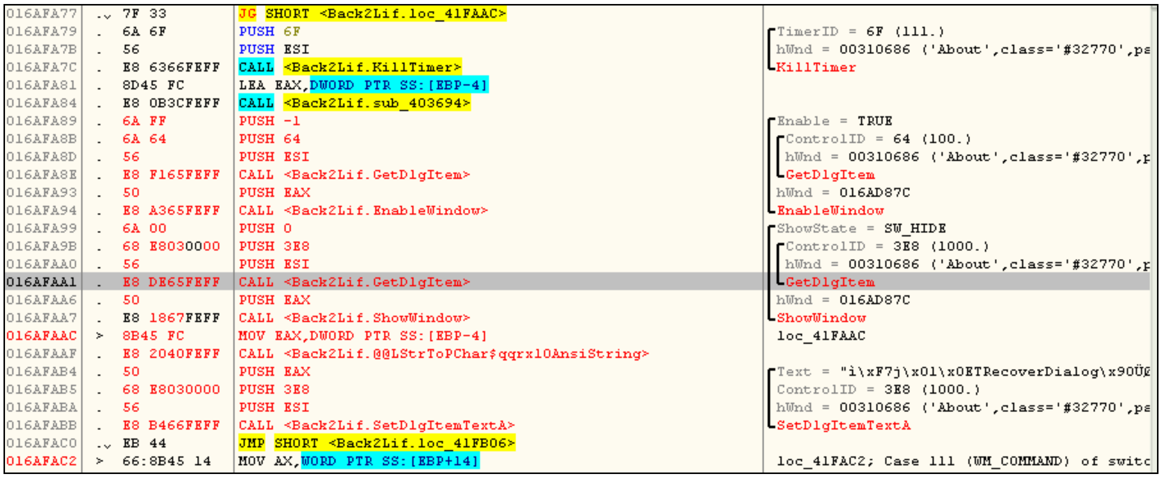
Doing this way the program executes normally its timers, but it does them instantly!

We can now place a breakpoint into the case of WM_TIMER where the program does an interesting call to KillTimer, this function is called when the delay imposed by shareware nag is over.



```
016AFA77  .v 7F 33        JG SHORT <Back2Lif.loc_41FAAC>
016AFA79  .  6A 6F        PUSH 6F                             ┌TimerID = 6F (111.)
016AFA7B  .  56           PUSH ESI                            │hWnd = 00310686 ('About',class='#32770',ps
016AFA7C  .  E8 6366FEFF  CALL <Back2Lif.KillTimer>           └KillTimer
```

Just after the call to KillTimer the program does a series of graphic GUI system calls to set fonts, look & feel.



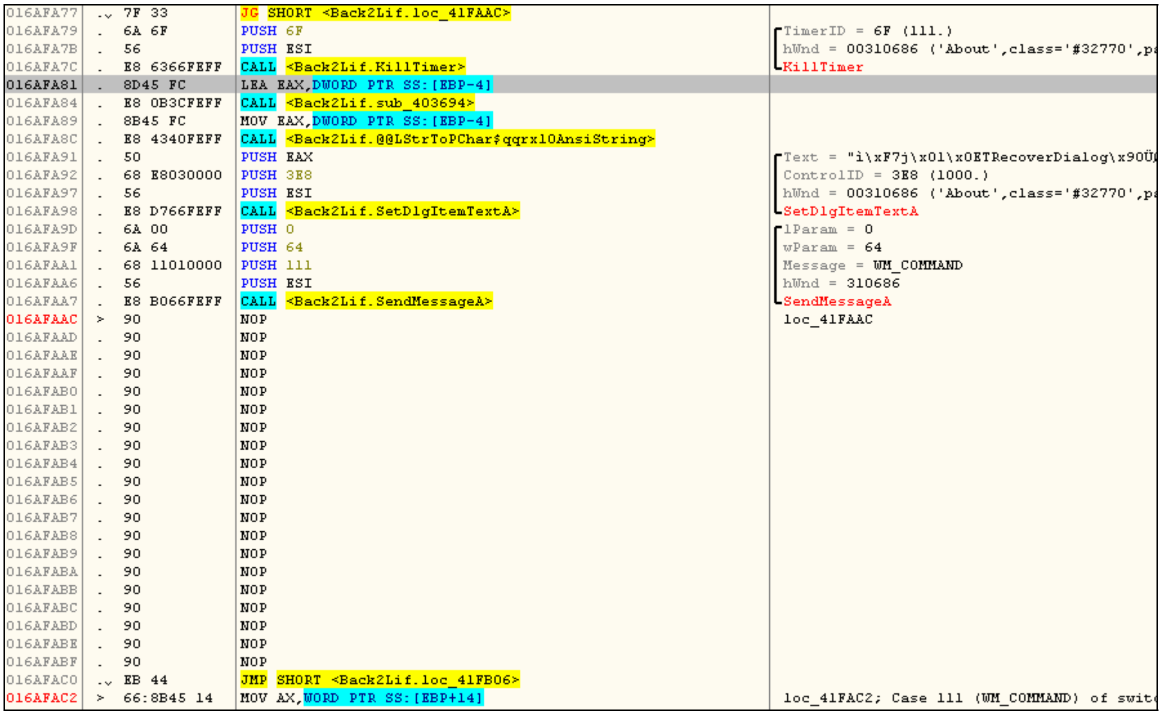
```
016AFA77  .v 7F 33        JG SHORT <Back2Lif.loc_41FAAC>
016AFA79  .  6A 6F        PUSH 6F                                      TimerID = 6F (111.)
016AFA7B  .  56           PUSH ESI                                     hWnd = 00310686 ('About',class='#32770',ps
016AFA7C  .  E8 6366FEFF  CALL <Back2Lif.KillTimer>                    KillTimer
016AFA81  .  8D45 FC      LEA EAX,DWORD PTR SS:[EBP-4]
016AFA84  .  E8 0B3CFEFF  CALL <Back2Lif.sub_403694>
016AFA89  .  6A FF        PUSH -1                                      Enable = TRUE
016AFA8B  .  6A 64        PUSH 64                                      ControlID = 64 (100.)
016AFA8D  .  56           PUSH ESI                                     hWnd = 00310686 ('About',class='#32770',p
016AFA8E  .  E8 F165FEFF  CALL <Back2Lif.GetDlgItem>                   GetDlgItem
016AFA93  .  50           PUSH EAX                                     hWnd = 016AD87C
016AFA94  .  E8 A365FEFF  CALL <Back2Lif.EnableWindow>                 EnableWindow
016AFA99  .  6A 00        PUSH 0                                       ShowState = SW_HIDE
016AFA9B  .  68 E8030000  PUSH 3E8                                     ControlID = 3E8 (1000.)
016AFAA0  .  56           PUSH ESI                                     hWnd = 00310686 ('About',class='#32770',p
016AFAA1  .  E8 DE65FEFF  CALL <Back2Lif.GetDlgItem>                   GetDlgItem
016AFAA6  .  50           PUSH EAX                                     hWnd = 016AD87C
016AFAA7  .  E8 1867FEFF  CALL <Back2Lif.ShowWindow>                   ShowWindow
016AFAAC  >  8B45 FC      MOV EAX,DWORD PTR SS:[EBP-4]                 loc_41FAAC
016AFAAF  .  E8 2040FEFF  CALL <Back2Lif.@@LStrToPChar$qqrx10AnsiString>
016AFAB4  .  50           PUSH EAX                                     Text = "ì\xF7j\x01\x0ETRecoverDialog\x90Ü
016AFAB5  .  68 E8030000  PUSH 3E8                                     ControlID = 3E8 (1000.)
016AFABA  .  56           PUSH ESI                                     hWnd = 00310686 ('About',class='#32770',ps
016AFABB  .  E8 B466FEFF  CALL <Back2Lif.SetDlgItemTextA>              SetDlgItemTextA
016AFAC0  .v EB 44        JMP SHORT <Back2Lif.loc_41FB06>
016AFAC2  >  66:8B45 14   MOV AX,WORD PTR SS:[EBP+14]                  loc_41FAC2; Case 111 (WM_COMMAND) of switc
```

Given that these functions are useless for us (because we want to skip the dialog and if doesn't look nice is the same), we can use this space as a code cave where to code our patch number 2.

Particularly what we will change is the code between 016AFA89 and 016AFABB. We moved to the top the final actions of the original code that were between 016AFAAC and 016AFABB and the modified the code is added just after (from 016AFA9D to 016AFABF). The new code simply calls SendMessageA posting a WM_COMMAND with wParam=64.

The new code becomes:

Patch 2:

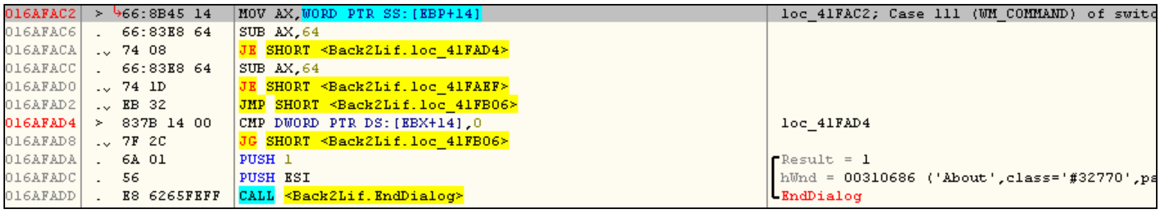


```
016AFA77  .v 7F 33        JG SHORT <Back2Lif.loc_41FAAC>
016AFA79  .  6A 6F        PUSH 6F                                      TimerID = 6F (111.)
016AFA7B  .  56           PUSH ESI                                     hWnd = 00310686 ('About',class='#32770',p
016AFA7C  .  E8 6366FEFF  CALL <Back2Lif.KillTimer>                    KillTimer
016AFA81  .  8D45 FC      LEA EAX,DWORD PTR SS:[EBP-4]
016AFA84  .  E8 0B3CFEFF  CALL <Back2Lif.sub_403694>
016AFA89  .  8B45 FC      MOV EAX,DWORD PTR SS:[EBP-4]
016AFA8C  .  E8 4340FEFF  CALL <Back2Lif.@@LStrToPChar$qqrx10AnsiString>
016AFA91  .  50           PUSH EAX                                     Text = "ì\xF7j\x01\x0ETRecoverDialog\x90Ü
016AFA92  .  68 E8030000  PUSH 3E8                                     ControlID = 3E8 (1000.)
016AFA97  .  56           PUSH ESI                                     hWnd = 00310686 ('About',class='#32770',p
016AFA98  .  E8 D766FEFF  CALL <Back2Lif.SetDlgItemTextA>              SetDlgItemTextA
016AFA9D  .  6A 00        PUSH 0                                       lParam = 0
016AFA9F  .  6A 64        PUSH 64                                      wParam = 64
016AFAA1  .  68 11010000  PUSH 111                                     Message = WM_COMMAND
016AFAA6  .  56           PUSH ESI                                     hWnd = 310686
016AFAA7  .  E8 B066FEFF  CALL <Back2Lif.SendMessageA>                 SendMessageA
016AFAAC  >  90           NOP                                          loc_41FAAC
016AFAAD  .  90           NOP
016AFAAE  .  90           NOP
016AFAAF  .  90           NOP
016AFAB0  .  90           NOP
016AFAB1  .  90           NOP
016AFAB2  .  90           NOP
016AFAB3  .  90           NOP
016AFAB4  .  90           NOP
016AFAB5  .  90           NOP
016AFAB6  .  90           NOP
016AFAB7  .  90           NOP
016AFAB8  .  90           NOP
016AFAB9  .  90           NOP
016AFABA  .  90           NOP
016AFABB  .  90           NOP
016AFABC  .  90           NOP
016AFABD  .  90           NOP
016AFABE  .  90           NOP
016AFABF  .  90           NOP
016AFAC0  .v EB 44        JMP SHORT <Back2Lif.loc_41FB06>
016AFAC2  >  66:8B45 14   MOV AX,WORD PTR SS:[EBP+14]                  loc_41FAC2; Case 111 (WM_COMMAND) of swit
```

Essentially I moved a piece of the original code on the top of the code piece to modify and I then added a call like SendMessage(ESI, WM_COMMAND, 64, 0)

ESI is by definition for a DlgProc the window's handle. Sending a WM_COMMAND message makes the system to call another time the DlgProc (nested call) and jumps to the WM_COMMAND case, with a wParam (stored in EAX) equal to 64.

The following is the piece of code which is executed:



```
016AFAC2  > ┌66:8B45 14   MOV AX,WORD PTR SS:[EBP+14]         loc_41FAC2; Case 111 (WM_COMMAND) of switc
016AFAC6  .  66:83E8 64   SUB AX,64
016AFACA  .v 74 08        JE SHORT <Back2Lif.loc_41FAD4>
016AFACC  .  66:83E8 64   SUB AX,64
016AFAD0  .v 74 1D        JE SHORT <Back2Lif.loc_41FAEF>
016AFAD2  .v EB 32        JMP SHORT <Back2Lif.loc_41FB06>
016AFAD4  >  837B 14 00   CMP DWORD PTR DS:[EBX+14],0         loc_41FAD4
016AFAD8  .v 7F 2C        JG SHORT <Back2Lif.loc_41FB06>
016AFADA  .  6A 01        PUSH 1                              ┌Result = 1
016AFADC  .  56           PUSH ESI                            │hWnd = 00310686 ('About',class='#32770',pa
016AFADD  .  E8 6265FEFF  CALL <Back2Lif.EndDialog>           └EndDialog
```

As you can see AX==64 and [EBX+14] (which is one of those values required to following working of the program) is already set to 0 by previous calls nested into the nag. The result is that the EndDialog is called.

The result is that the dialog appears for a moment and immediately disappears, just like if we waited for the countdown and then pressed "Restore" button.

4. Final Remarks

Doing things this way prevents you from wasting your time analyzing where the nag sets the program things and so on. You just skip the dialog doing like the user would have done, wait (actually less than usual because of patch #1), press the button and exit from the nag.

The result of which, is that patching in the way described stops the program from hanging here:

```
016AE177  |.  8B40 0C       MOV EAX,DWORD PTR DS:[EAX+C]
016AE17A  |.  E8 498BFEFF   CALL <Back2Lif.sub_406CC8>   ; if [EAX+1C]==0 doesn't work!
```

This call in any other case gives problems because EAX is not correctly initialized.

4. Writing OllyDbg Scripts, Buzifer of Team RESURRECTiON

Scripts have become a powerful way to automate tasks that sometimes can require a lot of time and work. Most scripts are written to do some unpacking task and/or find the OEP of protected code. Usually packers/protectors use a logical way to do their actions. If you know how and write a script, you basically have a generic way to defeat it. Almost everything you can do in OllyDbg you can do with a script. The reason behind this paper is for people who never have written scripts and want an explanation of how it works and why.

1. Things Needed.

To use scripts in OllyDbg a plugin is required. Ollyscript has been updated to ODbgScript (by SHaG & Epsylon). The scripting language is very similar to assembly. It has about 97 commands which can be combined and manipulated to do almost everything.

A handy tool for writing scripts is OSEditor, the command list was for the old Ollyscript so I updated it. You can find this tool and ODbgscript in the supplements package.

2. Variables and a first example

Let's take an example: You want to retrieve the codesize of a program and save it for later use. First thing is to declare a variable and give it a good name.

```
var codebase
```

Or maybe this one

```
var cbase
```

Example of bad naming:

```
var mycoolvariable
```

var declares a variable; this is needed to store things from a return value, think of variables as boxes. You use them for storing items. You can name the variable to almost everything except using reserved words; they are the commands in the scripting language. A good habit is to name things so they are self explained and comment your code. This improves the readability of the script and makes it much easier to make changes and track down errors. To write comments use // in the beginning.

To save things the function $RESULT is used. It returns values for other functions. This way we can have the result saved for later use, Instead of being limited to just save one. It works this way:

```
Declare variable.
Do some action and store result in $RESULT
Transfer $RESULT to a variable.
```

An example

```
//This is an example of retrieving the codebase and display it
//in a msgbox.

var cbase                   //declares the variable cbase
GMI eip, CODEBASE           // Now $RESULT is the address to the codebase
mov cbase, $RESULT          //moves the result to our variable
msg cbase                   //Msgbox with the value
ret                         //End script
```

3. Execute commands

These are some basic run commands.

RUN	Execute F9 in Ollydbg
STO	Execute F8 in OllyDbg.
STI	Execute F7 in OllyDbg.
RTR	Executes "Run to return" in OllyDbg

4. Conditional jumps.

A script can repeat something until a statement is true or false. First we compare with the CMP command.

CMP destination, source

Example:

```
cmp y, x  (variables)
cmp eip, 401000
```

Jumps that can be used:

JNE	Jump not equal
JMP	Jump
JE	Jump equal
JBE	Jump If Below or Equal
JB	Jump below
JA	Jump above

Example of a script using conditional jumps:

```
var counter            //declares the variable counter

start:
//Putting a ':' after the text transform the text into a name of a label. This is useful
so we can make jumps to specific parts of the code.

cmp counter,10         //Compares the variable counter with 10
```

```
ja finish              //jump if above 10

sto                    //executes F8 in OllyDbg

inc counter            //Increase our counter by 1, otherwise it would
                       //be an endless loop

jmp start              //If the counter is lower than 10 we jump back

finish:
msg counter
```

5. Writing a script to unpack UPX

Searching for a specific sequence of bytes is very useful. The bytes of a Asm command is displayed to the left in Ollydbg. You can also use binary edit to see them. This method can be applied to a couple of other packers like Aspack.

```
sti                         //Executes F7 (step into)
findop eip, #60#            //Searches code starting at addr
                            //for an instruction, (Find command PUSHAD)
                            //Wildcards can be used

bphws $RESULT,"x"           //Set hardware breakpoint. Available modes are
                            //"r" - read, "w" - write or "x" - execute.

run                         //Executes F9
sti
ret                         //Exit script
```

Another example: searching for kernel32.LoadLibraryA

```
find eip, #FF9674840600#
bp $RESULT
esto           //Shift F9
ret
```

6. Writing scripts to set breakpoints.

All breakpoints in Ollydbg are supported in scripts.

BC	Clear unconditional breakpoint at addr.
BP	Set unconditional breakpoint at addr.
BPC	Clear unconditional breakpoint at addr.
BPCND	Set breakpoint on address addr with condition cond.
BPL	Sets logging breakpoint at address addr that logs expression expr
BPMC	Clear memory breakpoint
BPHWC	Delete hardware breakpoint at a specified address
BPHWS	Set hardware breakpoint. Mode can be "r" - read, "w" - write or "x" - execute.
BPRM	Set memory breakpoint on read. Size is size of memory in bytes.
BPWM	Set memory breakpoint on write. Size is size of memory in bytes.
BC	Clear unconditional breakpoint at addr.

Example of putting breakpoint on MessageBoxA

```
start:
gpa "MessageBoxA", "user32.dll"
// Gets the address of the specified procedure in the specified library.
cmp $RESULT,0
je notfound
bp $RESULT
msg "Breakpoint on MessageBoxA"
ret
```

```
notfound:
msg "No breakpoint on MessageBoxA"
ret
```

Try to improve the following script to include the following:

```
GetDlgItemTextA
GetWindowTextA
lstrcmpA
GetPrivateProfileStringA
GetPrivateProfileIntA
RegQueryValueExA
WritePrivateProfileStringA
WritePrivateProfileIntA
```

This one can be useful to get the correct breaks when serial fishing. Included in the supplements there's a list of the commonly used API calls for different tasks..

7. Using flags

!CF	Carry
!PF	Parity
!AF	Auxiliary carry
!ZF	Zero flag
!SF	Sign
!TF	Trap
!IF	Interrupt
!DF	Direction
!OF	Overflow

Using flags, the following script will execute F7 commands in OllyDbg until Zeroflag is 0.

```
var counter

start:
cmp !ZF,0
je end
inc counter
sti
jmp start

end:

msg "Zeroflag is 0"
```

8. Final Remarks

I hope you have got a basic understanding of how scripts work and continue to practice to code. As soon as you understand the basics the real fun begins.

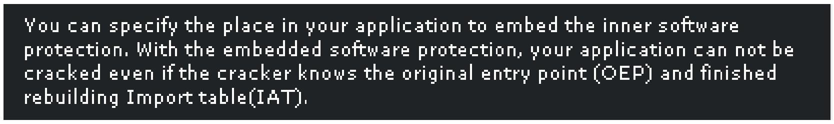
5. Utilizing Code Injection on an ACprotected application, condzero of ARTeam

1. Introduction

Today's target will deal with 1Click DVD Copy v5.0.2.9. protected by ACProtect or as it's now known Ultraprotect.

What is ACProtect?
ACProtect is a software protection application that allows developers to protect software against cracking with special anti-crack techniques under all Windows platforms.

I particularly got a kick out of reading this bullet from the ACProtect website: http://www.ultraprotect.com/



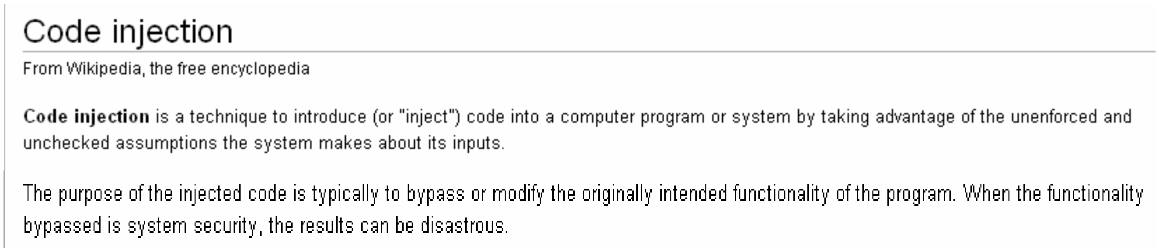
You can specify the place in your application to embed the inner software protection. With the embedded software protection, your application can not be cracked even if the cracker knows the original entry point (OEP) and finished rebuilding Import table(IAT).

Many Software Vendors / Developers make the mistake, when choosing Software Protection, of not protecting ALL their assets. Protecting the main application's module is sometimes not enough. There are many instances, where an application makes use of what I'll call "Helper" dll(s), not to be confused with System dll's. In many cases, these helper dll's are not protected, or if they are, not protected to the same degree as the main module.

We will utilize a very powerful and often overlooked technique in overcoming the limitations of this software, namely code injection. We can utilize a certain helper dll (vso_hwe.dll) to inject code into the process.

2. So what is Code Injection?

The following excerpts on Code injection were taken from the following link (where you can read more): http://en.wikipedia.org/wiki/Code_injection



Code injection

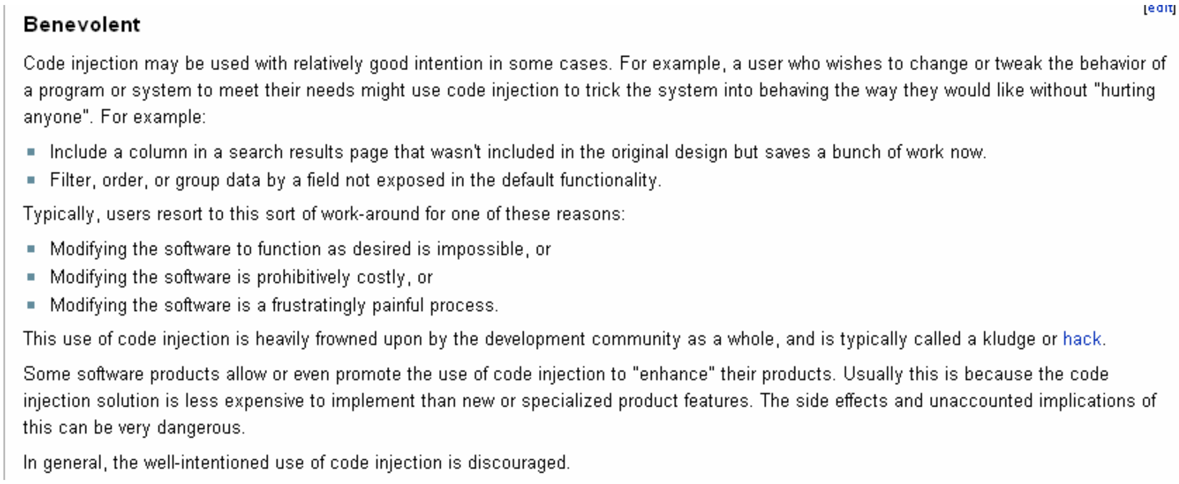
From Wikipedia, the free encyclopedia

Code injection is a technique to introduce (or "inject") code into a computer program or system by taking advantage of the unenforced and unchecked assumptions the system makes about its inputs.

The purpose of the injected code is typically to bypass or modify the originally intended functionality of the program. When the functionality bypassed is system security, the results can be disastrous.

Uses of Code injection:

Malevolent: We will not discuss this here.



[edit]

Benevolent

Code injection may be used with relatively good intention in some cases. For example, a user who wishes to change or tweak the behavior of a program or system to meet their needs might use code injection to trick the system into behaving the way they would like without "hurting anyone". For example:

- Include a column in a search results page that wasn't included in the original design but saves a bunch of work now.
- Filter, order, or group data by a field not exposed in the default functionality.

Typically, users resort to this sort of work-around for one of these reasons:

- Modifying the software to function as desired is impossible, or
- Modifying the software is prohibitively costly, or
- Modifying the software is a frustratingly painful process.

This use of code injection is heavily frowned upon by the development community as a whole, and is typically called a kludge or hack.

Some software products allow or even promote the use of code injection to "enhance" their products. Usually this is because the code injection solution is less expensive to implement than new or specialized product features. The side effects and unaccounted implications of this can be very dangerous.

In general, the well-intentioned use of code injection is discouraged.

Ahhh… this is perfect for our solution!

Some things to keep in mind. Timing is everything when we are modifying a process's memory using an indirect approach such as code injection. If using an external dll to introduce code injection, we can do one of three things.

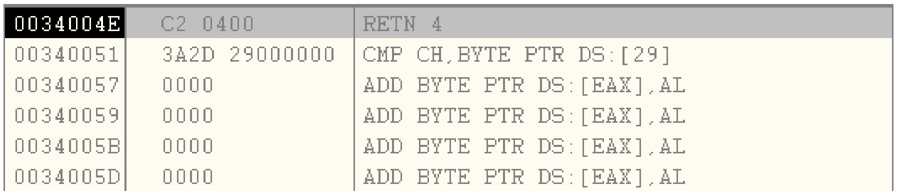
1. Does the dll load when our target is executed? If yes, then we can make the decision to choose a particular function within the dll to hook for our injected code. Create a code cave and patch the beginning of the function to jump to the code cave, perform some processing, maybe insure that we only execute our changes once via a conditional switch, and jump back to the function we hooked.
2. When the dll loads, is the target's code unpacked and exposed to external modifications? If yes (as we will do in this example) create a code cave with our code and change the original OEP to point to our new code cave and simply jump to the original OEP after we're done.
3. Some combination of the above.

3. Solution

Using a debugger, (OllyDbg in this case), uncheck the following options in the Exception tab of the debugger's options:

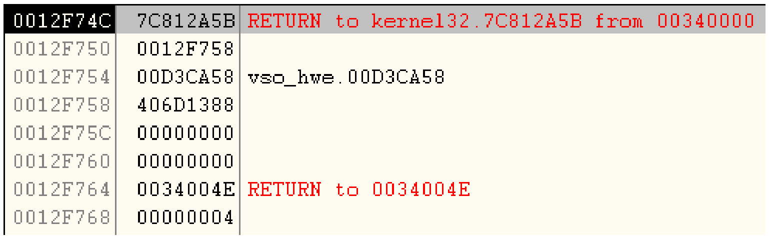
1. Memory Access Violation
2. Ignore also following exceptions or ranges.

Run (F9) the target. It will break on a handled exception and Olly's CPU main thread code section will look similar to the following:



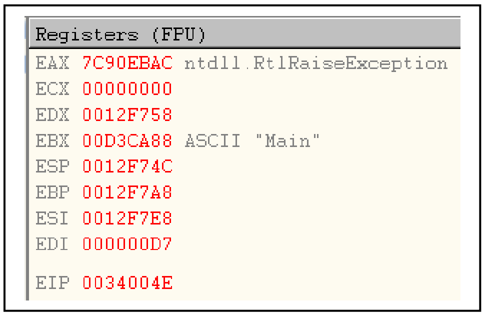
```
0034004E  C2 0400         RETN 4
00340051  3A2D 29000000   CMP CH,BYTE PTR DS:[29]
00340057  0000            ADD BYTE PTR DS:[EAX],AL
00340059  0000            ADD BYTE PTR DS:[EAX],AL
0034005B  0000            ADD BYTE PTR DS:[EAX],AL
0034005D  0000            ADD BYTE PTR DS:[EAX],AL
```

Note: the actual address above may be different on your machine. Take a look at the Stack window and you see something similar to the following:



```
0012F74C  7C812A5B  RETURN to kernel32.7C812A5B from 00340000
0012F750  0012F758
0012F754  00D3CA58  vso_hwe.00D3CA58
0012F758  406D1388
0012F75C  00000000
0012F760  00000000
0012F764  0034004E  RETURN to 0034004E
0012F768  00000004
```

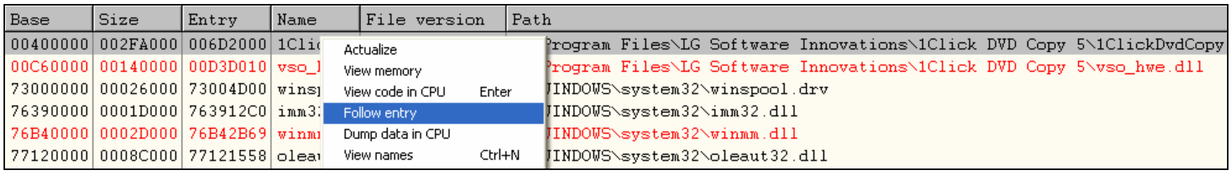
This is our first indication that our chosen "Helper" dll (vso_hwe.dll) has been loaded into the process. Also if you look at the Registers window we have another indication. Look at the EBX register below:



```
Registers (FPU)
EAX 7C90EBAC ntdll.RtlRaiseException
ECX 00000000
EDX 0012F758
EBX 00D3CA88 ASCII "Main"
ESP 0012F74C
EBP 0012F7A8
ESI 0012F7E8
EDI 000000D7

EIP 0034004E
```

Note: The address maybe different on your machine. With this in mind, we can Restart the target, but this time we will check Break on new module (DLL) in Olly's Events tab for debugger options.

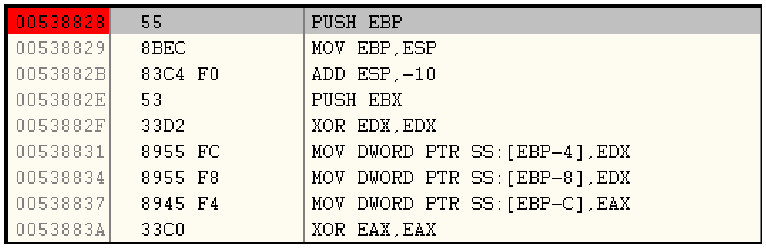
Follow the DLL load events to our chosen dll as seen below and choose follow entry on the main module:



```
Base     Size     Entry    Name     File version   Path
00400000 002FA000 006D2000 1Clic                   rogram Files\LG Software Innovations\1Click DVD Copy 5\1ClickDvdCopy
00C60000 00140000 00D3D010 vso_I                   rogram Files\LG Software Innovations\1Click DVD Copy 5\vso_hwe.dll
73000000 00026000 73004D00 winsj                   INDOWS\system32\winspool.drv
76390000 0001D000 763912C0 imm3:                   INDOWS\system32\imm32.dll
76B40000 0002D000 76B42B69 winmi                   INDOWS\system32\winmm.dll
77120000 0008C000 77121558 oleai                   INDOWS\system32\oleaut32.dll
```

At this point, most of the code section that we are interested in is unpacked. Keep in mind that Ultraprotect, similar to AsProtect, will not necessarily expose everything, so once again, Timing is critical.

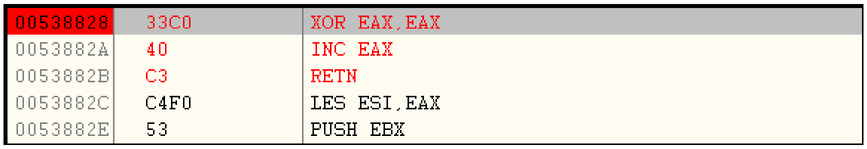
I will quickly jump to the code section of interest. This section (function) performs what I'll call the Boolean "Is Application Registered / Activated" routine. Set a BP as shown below:



```
00538828  55         PUSH EBP
00538829  8BEC       MOV EBP,ESP
0053882B  83C4 F0    ADD ESP,-10
0053882E  53         PUSH EBX
0053882F  33D2       XOR EDX,EDX
00538831  8955 FC    MOV DWORD PTR SS:[EBP-4],EDX
00538834  8955 F8    MOV DWORD PTR SS:[EBP-8],EDX
00538837  8945 F4    MOV DWORD PTR SS:[EBP-C],EAX
0053883A  33C0       XOR EAX,EAX
```

Uncheck the Break on new module in the events tab and Run (F9) the target. You will encounter several exceptions. Simply hit Shift+F9 to get past them until we get to our BP. If you follow the code in the routine, you can see it's doing some serial code / activation code checking. If we follow the return from this procedure, we notice a TEST AL,AL condition. If we did nothing, then AL == 00. The application will break on this procedure many times during the course of executing depending on options / features chosen. If we ran the application normally, we would either get a limitations nag screen and the option to Continue or a registration screen which would allow you to go no further.

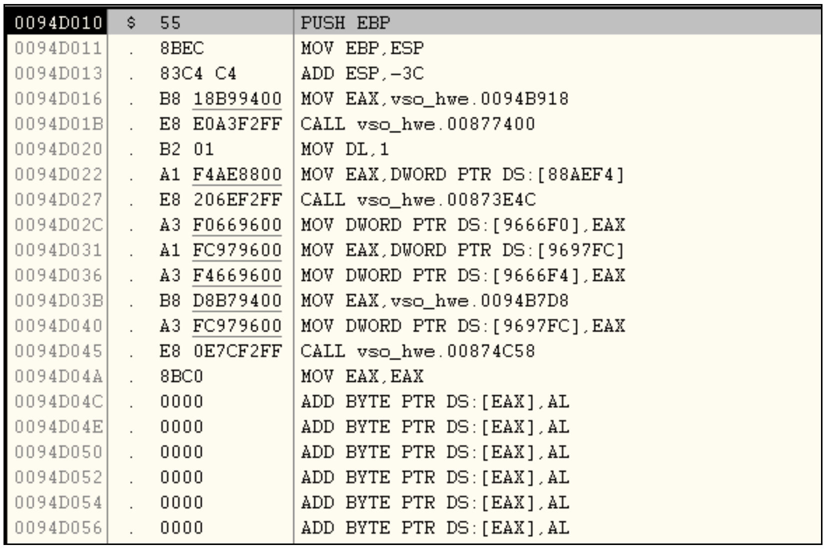
Restart the target again to our BP above. Make the following changes as shown below:



```
00538828  33C0   XOR EAX,EAX
0053882A  40     INC EAX
0053882B  C3     RETN
0053882C  C4F0   LES ESI,EAX
0053882E  53     PUSH EBX
```

Run (F9) the target. There should be no registration screen this time. Now it's time to implement our code injection.

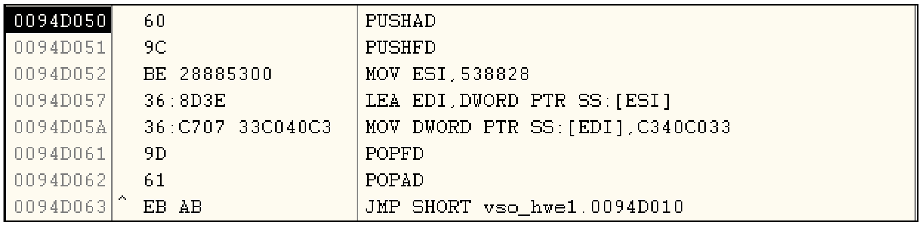
Open vso_hwe.dll in OllyDbg utilizing its ***Request to Load Dll*** feature. You will see something similar to the following:



```
0094D010  $  55            PUSH EBP
0094D011  .  8BEC          MOV EBP,ESP
0094D013  .  83C4 C4       ADD ESP,-3C
0094D016  .  B8 18B99400   MOV EAX,vso_hwe.0094B918
0094D01B  .  E8 E0A3F2FF   CALL vso_hwe.00877400
0094D020  .  B2 01         MOV DL,1
0094D022  .  A1 F4AE8800   MOV EAX,DWORD PTR DS:[88AEF4]
0094D027  .  E8 206EF2FF   CALL vso_hwe.00873E4C
0094D02C  .  A3 F0669600   MOV DWORD PTR DS:[9666F0],EAX
0094D031  .  A1 FC979600   MOV EAX,DWORD PTR DS:[9697FC]
0094D036  .  A3 F4669600   MOV DWORD PTR DS:[9666F4],EAX
0094D03B  .  B8 D8B79400   MOV EAX,vso_hwe.0094B7D8
0094D040  .  A3 FC979600   MOV DWORD PTR DS:[9697FC],EAX
0094D045  .  E8 0E7CF2FF   CALL vso_hwe.00874C58
0094D04A  .  8BC0          MOV EAX,EAX
0094D04C  .  0000          ADD BYTE PTR DS:[EAX],AL
0094D04E  .  0000          ADD BYTE PTR DS:[EAX],AL
0094D050  .  0000          ADD BYTE PTR DS:[EAX],AL
0094D052  .  0000          ADD BYTE PTR DS:[EAX],AL
0094D054  .  0000          ADD BYTE PTR DS:[EAX],AL
0094D056  .  0000          ADD BYTE PTR DS:[EAX],AL
```

Notice there's a whole bunch of binary zeroes beginning at address 0094D04C above. Note the address may be different on your machine. Note our OEP at address 0094D010. We could change this line and jump to our code cave or change the OEP using a PE editor which is what we will do in this Tutorial.

Make the following changes as shown below:

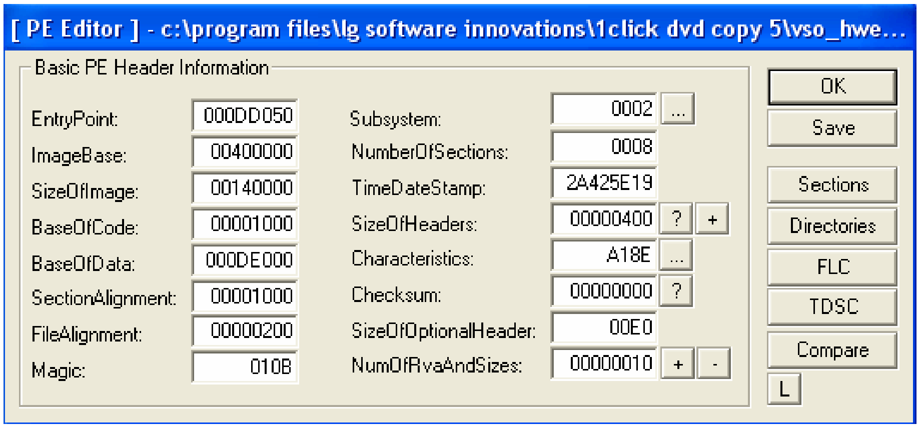


```
0094D050     60                  PUSHAD
0094D051     9C                  PUSHFD
0094D052     BE 28885300         MOV ESI,538828
0094D057     36:8D3E             LEA EDI,DWORD PTR SS:[ESI]
0094D05A     36:C707 33C040C3    MOV DWORD PTR SS:[EDI],C340C033
0094D061     9D                  POPFD
0094D062     61                  POPAD
0094D063  ^  EB AB               JMP SHORT vso_hwe1.0094D010
```

I chose to start at address 0094D050. Note we save our addresses and flags and restore them after our changes.
Since the address of our intended target should always load at the same address (00400000) in the process, we can move the hardcoded value of our target's destination address as shown. We then load this address in order to make our changes. The bytes (1 DWORD) are moved in reverse order. We then jump back to our original OEP. Highlight all your changes and be sure to copy them to the executable.

We are not finished yet.

Using a PE editor, we need to change the OEP (Original Entry Point) of this dll as shown below and save it.



[PE Editor] - c:\program files\lg software innovations\1click dvd copy 5\vso_hwe...

Basic PE Header Information

EntryPoint:	000DD050	Subsystem:	0002
ImageBase:	00400000	NumberOfSections:	0008
SizeOfImage:	00140000	TimeDateStamp:	2A425E19
BaseOfCode:	00001000	SizeOfHeaders:	00000400
BaseOfData:	000DE000	Characteristics:	A18E
SectionAlignment:	00001000	Checksum:	00000000
FileAlignment:	00000200	SizeOfOptionalHeader:	00E0
Magic:	010B	NumOfRvaAndSizes:	00000010

Now when we Run the target again, our code will be injected when this dll loads and initializes.

4. Final Remarks

Code Injection is not very complicated and can greatly simplify the task of modifying the original intended functionality of the program vs. MUP'ing and fixing, writing a loader, etc…
You may wish to insure that a chosen region of binary zeroes in the dll is not overwritten by the process.
Note: This application calls home with some possibly sensitive information (i.e. UserID=, MacID=, etc.) so you may wish to uncheck the option to Enable Update Notification and do this on your own. I also noted an application exception upon application exit if I turned off the Enable Update Notification feature and had my internet connection disabled. Weird. Hope you enjoyed this paper.

6. Code Obfuscation, zyzygy

1. Abstract

This is just a preliminary stage of investigation so the approach might be crude but it works and may lead to better ideas.

Let's begin.

What I understand from Code Obfuscation is that, the code is hidden in the pool of junk data. This trick is extensively used nowadays in packers.

2. Approach

I propose an approach based on these two tools:

Editor: RADASM
Assembler: MASM32

I will use the following code throughout the article. It does nothing but display 2 message boxes.

```
.386
.model flat, stdcall      ;32 bit memory model
option casemap :none      ;case sensitive


include windows.inc
include kernel32.inc
include user32.inc
include shell32.inc

includelib kernel32.lib
includelib user32.lib
includelib shell32.lib


.data
caption db "Fine",0
text db "Hi!",0
text1 db "Bye!",0

.code
start:

    assume fs:nothing     ;setting up an SEH in case things go awry
    push _seh
    push fs:[0]
    mov fs:[0],esp

    invoke MessageBox,NULL,addr text,addr caption,MB_OK

    call @call
@call:pop eax         ;delta offset
    add eax,0Eh       ;add eax with the no. of bytes that will land at the actual code.
    jmp eax             ;jump to the actual code to be executed.

    dd 00E95564h        ;garbage value, random
    dd 0E9830048h
    cmp eax,1           ;actual code
     jne next
    jmp exit
next:
    invoke MessageBox,NULL,addr text1,addr caption,MB_OK
exit:
    invoke ExitProcess,0

_seh:
    pop fs:[0]
    mov ebx,[esp+4]
    mov esp,ebx
    jmp next
end start
```

The code has been commented to guide you the execution flow. Now the garbage codes are:

```
dd 00E95564h
dd 0E9830048h
```

They mask the actual code. Now that we have the source with us, assemble it and load it in a debugger (I used Ollydbg).

```
00401026   . E8 00000000    CALL testt.0040102B
0040102B   $ 58             POP EAX
0040102C   . 83C0 0E        ADD EAX,0E
0040102F   . FFE0           JMP EAX
00401031     64             DB 64                           ;  CHAR 'd'
00401032     55             DB 55                           ;  CHAR 'U'
00401033     E9             DB E9
00401034   . 0048 00        ADD BYTE PTR DS:[EAX],CL
00401037   . 83E9 83        SUB ECX,-7D
0040103A   . F8             CLC
0040103B   . 0175 02        ADD DWORD PTR SS:[EBP+2],ESI
0040103E   . EB 13          JMP SHORT testt.00401053
```

Our main code lies at 00401039. Upon tracing we find that CMP EAX, 1 is executed. If such garbage codes were present all over the actual code then the result would be a much mangled code. For this we shall make use of macros.

Our main garbage code will be the macro and then we shall call it wherever we desire to.

Macro:

```
        garbage macro
                db 0e8h,00,00,00,00    ;call to get delta offset
                pop eax                ;delta offset
                add eax,0Ch            ;add the necessary bytes to jump to the actual code
                jmp eax                ;jump to the actual code
                dw 8965h               ;garbage values
                dd 70e9a654h           ;garbage values
        endm
```

This macro does the job. One point I would like to state is that while calling the APIs, ensure that you use *call* keyword rather than the *invoke* keyword. This will give you more area to add your garbage code.

So here is our new code with the macro:

```
garbage macro

        db 0e8h,00,00,00,00
        pop eax
        add eax,0Ch
        jmp eax
        dw 8965h
        dd 70e9a654h
```

```
endm

.data
caption db "Fine",0
text db "Hi!",0
text1 db "Bye!",0
data db 00h

.code
start:

        assume fs:nothing
        push _seh
        push fs:[0]
        mov fs:[0],esp


        invoke MessageBox,NULL,addr text,addr caption,MB_OK

        garbage

        cmp eax,1
        garbage
        je next
next:
        garbage
        push 0
        push offset caption
        garbage
        push offset text1
        push 0
        garbage
        call MessageBox
        garbage
        push 0
        garbage
        call ExitProcess
_seh:
         pop fs:[0]
         garbage
         mov ebx,[esp+4]
         mov esp,ebx
         garbage
         jmp next
end start
```

As you can see, I have freely used the macro and upon assembling and loading it into a debugger, this is what you see (only a part):

```
0040104A   . 70 74 00       ASCII "pt",0
0040104D   > E8 00000000    CALL test.00401052
00401052   $ 58             POP EAX
00401053   . 83C0 0C        ADD EAX,0C
00401056   . FFE0           JMP EAX
00401058     65             DB 65                           ; CHAR 'e'
00401059     89             DB 89
0040105A     54             DB 54                           ; CHAR 'T'
0040105B     A6             DB A6
0040105C     E9             DB E9
0040105D   . 70 6A 00       ASCII "pj",0
00401060   > 68 00304000    PUSH test.00403000              ; ASCII "Fine"
00401065   . E8 00000000    CALL test.0040106A
0040106A   $ 58             POP EAX
0040106B   . 83C0 0C        ADD EAX,0C
0040106E   . FFE0           JMP EAX
00401070   . 65:8954A6 E9   MOV DWORD PTR GS:[ESI-17],EDX
00401075   . 70 68          JO SHORT test.004010DF
00401077   . 0930           OR DWORD PTR DS:[EAX],ESI
00401079   . 40             INC EAX
0040107A   . 006A 00        ADD BYTE PTR DS:[EDX],CH
0040107D   . E8 00000000    CALL test.00401082
00401082   $ 58             POP EAX
00401083   . 83C0 0C        ADD EAX,0C
00401086   . FFE0           JMP EAX
```

```
00401088     65             DB 65                           ; CHAR 'e'
00401089     89             DB 89
0040108A     54             DB 54                           ; CHAR 'T'
0040108B     A6             DB A6
0040108C     E9             DB E9
0040108D     70             DB 70                           ; CHAR 'p'
0040108E   . E8 63000000    CALL <JMP.&user32.MessageBoxA>  ;\MessageBoxA
00401093   . E8 00000000    CALL test.00401098
00401098   $ 58             POP EAX
00401099   . 83C0 0C        ADD EAX,0C
```

Very mangled code. Only when you trace can the actual code be seen clearly.

The more you know about opcode construction the better you will be in coding more complex algorithms. The stack is a very useful here. You can store the jump locations there for instance.

Consider that the code detects a debugger and with respect to the return value of the function you can decide to decrypt it accordingly. One of the easiest ways to confuse is use registers to jump. This is because if we hardcode the jump says:

```
        0044F42E  |. 74 06          JE SHORT 0044F436
```

The assembler will assemble it such that the debugger will display the address 0044F436. But if you change something like this:

```
mov  eax, 0044F436
jmp eax
```

The assembler may or may not assemble the 0044F436 location as there is no hardcoded jump. Before concluding this article I will present another piece of code, a macro within a macro.

```
garbage1 macro
        dw 025FFh
        add ebx,26h
        db 53h,0C3h
        dw 025ffh
endm

garbage macro
        db 0e8h,00,00,00,00
        pop eax
        cmp data,1     ;a check will determine whether to take jmp eax/push ebx & ret
        db 074h,10h
        mov ebx,eax
        add eax,13h
        db 0ebh,0Ch
        garbage1
        add eax,26h
        jmp eax
        dw 8965h
        dd 0560cee8h
        db 00h
endm
```

I have coded another macro and am calling it from the first one. If you have a look at the import table, FF25 in OllyDbg (or any debugger) is commonly used as a *JMP DWORD PTR.* So I used that as garbage code.

What we are doing here is loading the correct address to jump in ebx, pushing it on the stack and returning. A very common trick. Assemble and load it in your debugger for further analysis.

3. Final Remarks

This is just the basics of the code obfuscation. There are a lot of ideas and concepts that can be used to obfuscate the code.
I hope this helped you to conceive ideas on this and similar topics.

7. Testing for OllyDbg Using NtYieldExecution, Gabri3l of ARTeam

1. Introduction

I stumbled across this interesting OllyDbg detection method when I was trying to debug an error in a small program I was writing. However, when debugging the program in OllyDbg I found that I could not replicate the same results. This led me to dig deeper...

2. Analysis

The API function that was giving me trouble was NtYieldExecution. It was officially introduced in NT 4.0, but according to the metasploit SYSCALL page[21], it may have been undocumented and available in SP 3. Either way the functions purpose is the same. NtYieldExecution will pass execution to another running thread, giving up it's scheduled CPU time.

There is no documented return for NtYieldExecution, but it does return a result stored in EAX. It was this result that changed depending on whether or not the program was inside of OllyDbg. So I knew that there was a return value but I didn't know what it meant. For a better idea of how NtYieldExecution functions we can first look at it in OllyDbg:

```
7C90EA47 >   B8 16010000     MOV EAX, 116               ;Move NtYieldExecution Syscall
                                                        ;Number into EAX
7C90EA4C     BA 0003FE7F     MOV EDX, 7FFE0300
7C90EA51     FF12            CALL NEAR DWORD PTR DS:[EDX] ;ntdll.KiFastSystemCall
7C90EA53     C3              RETN
```

If you were to step into KiFastSystemCall we would see that it executes a SYSENTER. This enters the kernel system call that was specified in EAX. EAX in this case was 116 which is the value for NtYieldExecution. Because the function is located in the kernel we can't dig any deeper with Olly, so we cannot find the return value.

We could use SoftICE or another Ring0 debugger to examine NtYieldExecution at the Kernel level. But rather than go through all that trouble we will check a site that has already done the hard work for us: http://www.winehq.com

"Wine is an Open Source implementation of the Windows API on top of X and Unix. Think of Wine as a compatibility layer for running Windows programs. Wine does not require Microsoft Windows, as it is a completely free alternative implementation of the Windows API consisting of 100% non-Microsoft code."

While the code that Wine uses is not "Microsoft's" it still functions the same. And, even better, it is all open source and documented. Lets take a look at their implementation of NtYieldExecution: http://source.winehq.org/source/dlls/ntdll/sync.c?v=wine20050211#L767

```
/* NtYieldExecution (NTDLL.@)766  */

NTSTATUS WINAPI NtYieldExecution(void)
{
#ifdef HAVE_SCHED_YIELD
    sched_yield();
    return STATUS_SUCCESS;
#else
    return STATUS_NO_YIELD_PERFORMED;
#endif
```

The code above is very easy to understand we can see that NtYieldExecution actually does return a value based on whether it can yield its CPU cycles to another program. It returns either `STATUS_NO_YIELD_PERFORMED` if it was unable to yield its CPU cycles or `STATUS_SUCCESS` if it successfully passed its cycles to another thread.

[21] http://www.metasploit.com/users/opcode/syscalls.html

I would like to cover quickly that NtYieldExecution may have been implemented slightly differently on NT machines. On XP NtYieldExecution can return 2 values. In my tests on NT it seems to only return the STATUS_SUCCESS value, rendering this test unusable on NT machines. We also have this implementation of NtYieldExection found by Shub-Nigurrath[22]:
http://www.koders.com/c/fidD6698E8EFC18C0EB0D5D46FE74EAEE9E47347ED5.aspx?s=NtYieldExecution

```
NTSTATUS STDCALL
NtYieldExecution(VOID)
{
  PsDispatchThread(THREAD_STATE_RUNNABLE);
  return(STATUS_SUCCESS);
}
```

So it looks like this test will only work correctly on XP machines where the implementation of NtYieldExecution returns 2 values. We can use the Wine source to discover the values for both of the STATUS_NO_YIELD_PERFORMED and STATUS_SUCCESS return values:
http://source.winehq.org/source/include/ntstatus.h?v=wine20050211#L114

```
114     #define STATUS_NO_YIELD_PERFORMED                   0x40000024
30      #define STATUS_SUCCESS                              0x00000000
```

So here we discover the two results we receive from NtYieldExecution. EAX = 0 when we successfully yield CPU cycles, and EAX = 40000024 when no cycles were yielded.

With this knowledge gained we can actually develop a test for Olly using this API function.

3. Coding

We will be developing our test as a DLL plugin for the eXtensible Anti-Debug Tester written by Shub-Nigurrath. The eXtensible Anti-Debug Tester (xADT), is located at http://releases.accessroot.com

Our DLL will consist of 3 parts; each part performs a test that returns a different result based on what computer it is running on and whether or not it is in a debugger
Why 3 parts? I had developed this test and I thought that I was able to get a consistent result inside a debugger and a different result outside of a debugger. I then tried it on other XP machines and received different results all together! So I had to rewrite it again with multiple parts. Each test in each part is performed multiple times to help eliminate false positives. By implementing 3 parts and multiple tests we can successfully determine whether or not the program is operating inside Olly.

The following is my development of the plugin. I am going to skip the trial and error, and just give you the facts and the results.

Basically I encountered 2 types of computers. The results of each part are based on the type of computer you are running and the test performed. I do not know the physical or technical differences between types 1 and type 2 computers. These are just results based on observations and tests. Keep in mind that false positives CAN occur from time to time; this means that the EAX will contain 0 even if it is outside Olly. It happens from time to time when there is a high load on the system. We will take as many precautions necessary to eliminate false positive, but they may still occur. If this approach was to be used in an external protection or program then I would recommend increasing the number of times each part is repeated giving you a better chance to dodge system load. You can also repeat the whole test multiple times throughout program execution, and then see if there is an occurrence of a debugger not being detected. Luckily all the false positives seem to go in one direction; I have never had an occurrence where the program said it was NOT debugged when it was so. So if a debugger was not detected at least once across multiple runs then the program is most likely not being debugged. There is still a possibility that some "false" positives are not so false. I have noticed that some of them can occur when you have OllyDbg open and idling at a breakpoint. While the xADT program is not being explicitly debugged there is still the possibility that it is detecting an open debugger.

Anyway, we are going to ignore any false positives for this explanation. I called NtYieldExecution 2 ways. The first way is by trying to create a process but passing the CreateProcess call an invalid executable file name. The second way is by calling CreateProcess and passing it the path to a valid executable. Here I will try and cover the different reactions I received from type 1 and type 2 computers when calling

[22] through the search engine www.koders.com

NtYieldExecution in those 2 different ways. I will also cover how they results can be interpreted and used to determine if it is inside Olly.

Lets say that when EAX!=0 then that means NO
And when EAX=0 that means YES

1. Type 1 computers will always return NO when you call NtYieldProcess outside of a debugger
2. Type 1 computers will ONLY return YES when you are inside Olly and open a real process and then call NtYieldProcess
3. Type 1 computer will return NO if you are inside a debugger and call NtYieldProcess without creating a process
4. Type 2 computers will always return YES when the program is operating inside OLLY no matter how NtYieldProcess is called
5. Type 2 computers will return YES outside Olly if a real process is opened and then NtYieldProcess is called
6. Type 2 computers will ONLY return NO when NtYieldProcess is called without creating a process
7. HOWEVER! Type 2 computers will also return YES outside of Olly if a series of processes was recently created, like happens in PART #2
8. We can use that fact to determine if the program is running on Type 1 computer and inside Olly or running on Type 2 computers outside Olly

A little cheat table using the above information:

PART 1	PART 2	PART 3	RESULT
YES	YES	YES	INSIDE OLLY – TYPE 2 COMPUTER
NO	YES	YES	OUTSIDE OLLY – TYPE 2 COMPUTER
NO	YES	NO	INSIDE OLLY – TYPE 1 COMPUTER
NO	NO	NO	OUTSIDE OLLY – TYPE 1 COMPUTER

Now we can begin to develop our Debug test. First thing we are going to do is create a small program that's only purpose it to close itself. I will call it **Justclose**. This will be the process our main test program creates before calling NtYieldProcess.

The following is the code for our small justclose executable. In this program I am going to set its CPU priority to maximum to have it try and request CPU cycles. Then I will just have it exit, no need to keep it open.

Justclose.asm:

```
.386
.model flat,stdcall
option casemap:none

include  windows.inc
include  kernel32.inc
includelib kernel32.lib

.data?
hInstance HINSTANCE ?

.code
start:
INVOKE GetCurrentThread
INVOKE SetThreadPriority,eax,THREAD_PRIORITY_TIME_CRITICAL
INVOKE GetCurrentProcess
INVOKE SetPriorityClass,eax,REALTIME_PRIORITY_CLASS
INVOKE ExitProcess,EAX
end start
```

Our dll can now use justclose.exe by creating the process and then calling NtYieldExecution. We can then monitor the results of the function to determine if the program is inside or outside of Olly. The following is the steps I will take in my DLL and how I interpret the results of each part.

Deroko has written a plugin for xADT in ASM, and the source can be found in the plugin_examples directory of the program. I have based my dll on the skeleton of his code. The first thing we are going to do is retrieve the full path to our module. We can then modify that path to point to the location of our previously created executable "justclose.exe". This gives us a full path to justclose.exe and will allow us to put it in our plugin directory. Now we can begin on the testing portion of the DLL. The test I perform consists of 3 parts. I will briefly outline what the part does, and why each part is necessary.

Part #1 of the DLL performs a CreateProcess using an invalid string for the process name

- In every test outside Olly the return in EAX will be 0x40000024
- If 0 is returned in EAX then the program is operating inside Olly
- However, on some computers this test will return 0x40000024 in EAX even while in Olly that is why we perform part 2

Part #2 opens a new process called justclose.exe which does nothing except close immediately.

3. However, by just opening a program, execution is now yielded differently
4. On some computers if the return of EAX is not 0 then it is outside Olly and we can assume that it is not being debugged
5. However on other computers this test will always return 0 in EAX, inside or outside of Olly, if that occurs we need to go to Part #3

Part #3 performs a CreateProcess using an invalid string for the process name again

3. However this time the results are counter intuitive.
4. When the program is operating outside Olly, specific computers that returned 0x40000024 in EAX with PART #1 and 0 when opening a real process in Part #2 will now continue to return 0 for this part
5. However for other computers that the program is open in Olly that returned a value in EAX with PART #1 and 0 when opening a real process in Part #2 will now return 0x40000024, the same as they did in Part #1
6. So if you consistently receive EAX=0 for part 3 you are operating outside Olly

You can view the final source for the xADT extension NtYieldExecution.asm in the Supplements folder of the ARTeam ezine.

4. Final Remarks

So we have another Olly detection method, however this one is easily defeated by just constantly returning 0x40000024 when NtYieldExecution is called. It could be more complicated if the protected program included something along the lines of this:
http://www.winehq.com/hypermail/wine-devel/2005/08/att-0050/01-foo.c

There was another idea that deroko and I discussed however I did not implement it. It is possible to just call NtYieldExecution by its interrupt. This would prevent a Ring-3 program from hooking the function. The call could be implemented through either INT 2E or SYSENTER. Some sample code is provided below:

```
INVOKE CreateProcess,addrsInvalid,NULL,NULL,NULL,TRUE,00000008h,NULL,NULL,addr
startInfo,addr processinfo    ;Begin a new process using an invalid name for the
                              ;process name

xor edx,edx                   ;No Arguments Needed
MOV EAX,116h                  ;Move "NtYieldExecution" Syscall number into EAX
int 2eh                       ;Yield Execution to running process using Interrupt

ret                           ;Return from DebugTest1
DebugTest1 endp
```

Just some things to think about... I hope you found this interesting and enjoyed the read.

8. Coding a Serial Sniffer (Oraculum), anorganix of ARTeam

1. Opening words

First of all, I want to say that if you are an experienced reverser, then this article will look like child's play, but for people that are new to reversing it will (probably) be useful. I remember how hard it was for me at the beginning to understand things that were considered "everyone knows why this is done like that". This is why I want to try to explain things in an easy manner, for everyone to understand. Read ahead...

2. What is a Serial Sniffer and when to use it?

I know that the term used for this kind of programs is "Oraculum"[23], but in this article I prefer to call it a Serial Sniffer. The purpose of this paper is to cover a situation where you can't understand the algorithm of a target or can't code a keygen for it. Our target here is a simple CrackMe that I wrote to be able to show you in practice what needs to be done to defeat this situation. Please remember that the registration algorithm of the CrackMe is very simple and has no protection; our purpose here is to write a sniffer, not a keygen.

3. Things needed to get started

The tools:

Required Tools

- » OllyDbg
- » Borland Delphi (or any other programming language for writing the sniffer)

...and of course, do not forget our target CrackMe.

4. Inside the target

Let's fire up OllyDbg and find the serial check. If you didn't use PEiD before, now you'll know that the app was written in Delphi:

Target loaded in Olly

```
0045073C    PUSH EBP
0045073D    MOV EBP,ESP
0045073F    ADD ESP,-10
00450742    MOV EAX,CrackMe.0045055C
00450747    CALL CrackMe.00405BC8
```

Let's run it and see what we get if we enter a dummy name/serial combination… we get a "invalid code entered!" message. Open up the "Referenced text strings" window and search for the nasty message. Place a breakpoint on the registration call, like below and press the "Check" button again:

Getting closer

```
004503EF    CALL CrackMe.0040421C
004503F4    JNZ SHORT CrackMe.00450408
004503F6    MOV EDX,CrackMe.00450450        ;ASCII "Code accepted!"
004503FB    MOV EAX,DWORD PTR DS:[EBX+2FC]
00450401    CALL CrackMe.0042F44C
00450406    JMP SHORT CrackMe.00450418
00450408    MOV EDX,CrackMe.00450468        ;ASCII "Invalid code entered!"
```

[23] See also the several tutorials of Shub-Nigurrath on the subject, into ARTeam tutorials pages

You will break at 004503EF. Have a look at the EAX and EDX registers… yup, EAX holds the good serial and EDX holds the dummy serial we entered. It's needless to say that if EAX and EDX were equal, the program would give the "Code accepted!" message. In this case, this is all the information we need to code a sniffer – we know that at address 004503EF (from now on, named magic address) EAX holds the good serial. So let's proceed to the coding part.

5. Coding a Serial Sniffer with Delphi

To develop the sniffer, I will use Delphi 7 Enterprise. You can use whatever language you like, as long as you can follow my steps. Before the actual coding part, let's think for a minute what we need to do:

- start the program in suspended mode
- read the original bytes we are going to patch at the magic address
- write some bytes at the magic address, to make program enter an infinite-loop
- let the program run
- monitor if the program arrived at the infinite-loop (at magic address)
- if previous step is done, suspend the program and sniff the serial from EAX
- restore the original bytes (clear the infinite-loop) and resume the program

The code is not commented 100%, hopefully you will understand:

Sniffer code (Delphi)

```
{...}

const
  // this is the code we will write go make
  // the program go into an infinite-loop
  LOOP: array [0..1] of Byte = ($EB,$FE);

{...}

function SniffSerial(PI: PROCESS_INFORMATION; Ctx: _Context): string;
var
  X: Cardinal;
  Buff: PChar;
begin
  // allocate some memory
  GetMem(Buff,50);

  // suspend the program and get the context
  SuspendThread(PI.hThread);
  GetThreadContext(PI.hThread,Ctx);

  // read the value that [EAX] holds (the good serial)
  ReadProcessMemory(PI.hProcess,Pointer(Ctx.Eax),Buff,50,X);

  // set the result and free the buffer
  Result:=Trim(Buff);
  FreeMem(Buff);
end;

procedure TfrmMain.btnSniffClick(Sender: TObject);
var
  PI: PROCESS_INFORMATION;
  SI: STARTUPINFO;
  Context: _CONTEXT;
  Buffer: PChar;
  ORIG: array [0..1] of Byte;
  S: string;
  W: DWORD;
begin
  // disable button (avoid starting target multiple times)
  btnSniff.Enabled:=False;

  // allocate some memory and initialize vars
  GetMem(Buffer,255);
  FillChar(PI,SizeOf(TProcessInformation),#0);
  FillChar(SI,SizeOf(TStartupInfo),#0);
  SI.cb:=SizeOf(SI);
  // create the process (suspended)
  if not CreateProcess('CrackMe.exe',nil,nil,nil,False,
                       CREATE_SUSPENDED,nil,nil,SI,PI) then
  begin
       // enable button
    btnSniff.Enabled:=True;

    // set new log
    lblLog.Caption:='Failed to load process!';
    Exit;
  end;

  // read original bytes
  ReadProcessMemory(PI.hProcess,Pointer($004503EF),@ORIG,2,W);

  // write inifnite-loop
  WriteProcessMemory(PI.hProcess,Pointer($004503EF),@LOOP,2,W);

  // resume the program
  ResumeThread(PI.hThread);
  Context.ContextFlags:=$00010000+15+$10;

  // set new log
  lblLog.Caption:='Process patched!'+#13+
                  'Now enter a name and press the "Check" button...';

  while GetThreadContext(PI.hThread,Context) do
  begin
    // did we reach the infinite-loop?
    if Context.Eip=$004503EF then
    begin
      // get the serial and put in into „S”
      S:=SniffSerial(PI,Context);

      // restore original bytes and resume target
      WriteProcessMemory(PI.hProcess,Pointer($004503EF),@ORIG,2,W);
      ResumeThread(PI.hThread);

      // copy serial to clipboard and set new log
      Clipboard.AsText:=S;
      lblLog.Caption:='Your serial has been copied to clipboard!';
    end;

    // wait 10 miliseconds
    Sleep(10);
    Application.ProcessMessages;

    // if user wants to close the sniffer before exiting the target, close the target
too
    if WantToClose then
    begin
      TerminateThread(PI.hThread,0);
      Close;
    end;
  end;

  // free memory allocated @ beginning
  FreeMem(Buffer);

    // enable button
  btnSniff.Enabled:=True;
end;
```

Compile the source attached with this article if you can't manage it. Hopefully you will have a nice and running Serial Sniffer.

6. Final Remarks

Well, this is the end of this story; I hope all the things said here will be useful in broadening your knowledge. I suggest as usual using this material for learning purposes only, and not for cracking programs. Thank you for reading this article!

9. Ring 3 debugger detection via INVALID_HANDLE exception, deroko of ARTeam

1. Introduction

Well I discovered this during my little journey with ExeCryptor, I also made a huge mistake, which showed as not been as huge as I was expecting it to be.
Story goes… I was playing with ExeCryptor and I patched CreateThread with retn just to avoid new thread creation, of course, TLS callback was also patched after 1st one gets executed, just to simulate existence of one thread. Of course, ExeCryptor ran perfectly without a problem, then I moved to OllyDbg and applied same trick. But after passing exceptions to debuggy, suddenly ExeCryptor process exited, (I also used hook in ntoskrnl.exe!NtOpenProcess and NtReadVirtualMemory to deny read of olly process memory), but without retn in CreateThread it worked without a problem. Also my nonintrusive oepfinder worked without a problem. So, the only thing that was logical is that somehow EAX has a random value after hook CreateThread is called.
Looking at conditions in my case I had this situation: nonintrusive tracer hooks CreateThread but it works without a problem, and Olly also hooks CreateThread but it fails. So I used bpx in CreateRemoteThread (internally called by CreateThread) as 2nd layer API to avoid BPX detection in ExeCryptor. When I break at CreateRemoteThread I assembled simple patch to return from it. Then run app and soon Exception invalid handle occurred. I passed exception with shift+f9 and b00m, thread exception handler was called and resulted in process termination (e.g. it wasn't supposed to occur at that point).

2. Coding

Then I wrote simple program to see if this is new anti-debug trick:

```
        push    0deadc0deh
        callW   CloseHandle
```

Yep, it seems like exception is generated when this is ran trough ring3 debugger (olly, debug loader etc…) but under normal conditions this would never occur. So I was sure I have discovered a new anti-debug trick, ultimate (you will see why) anti-debug trick. After giving hint to one person, that person showed his understanding for this anti-debug trick, mostly lame understanding. I'm glad that protection developers have same understanding of windows system as that person and they didn't see real potential of this trick.

I'll now show you how stupid software developers are and those whom were using my hints to promote themselves on every single forum.

There is Native API known as NtRaiseException which is exported in ntdll.dll, its prototype is very simple:

```
NTSYSAPI NTSTATUS NTAPI NtRaiseException(
        IN PEXCEPTION_RECORD ExceptionRecord,
        IN PCONTEXT ThreadContext,
        IN BOOLEAN HandleException );
```

With this native API you may raise exception of any kind and force thread exception handler to be executed. But here comes a catch, in normal condition CloseHandle will never raise EXCEPTION_INVALID_HANDLE, never!!! But NtRaiseException can force thread exception handler to be called when EXCEPTION_INVALID_HANDLE is generated.

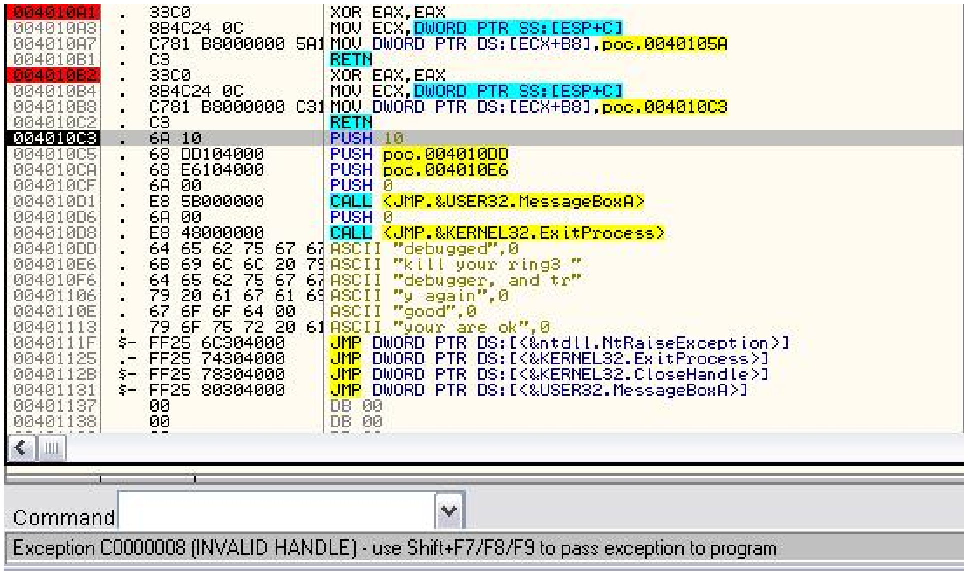
If you are debugging with ring3 and EXCEPTION_INVALID_HANDLE occurs due to dummy argument passed to CloseHandle you should continue execution with DBG_CONTINUE flag passed to ContinueDebugEvent, or in other words, press F9 in Olly. But when NtRaiseException is used to raise EXCEPTION_INVALID_HANDLE you should use DBG_EXCEPTION_NOT_HANDLED and force thread exception handler to be called.

In other words one time you have to use DBG_EXCEPTION_NOT_HANDLED in other case you should use DBG_CONTINUE (shift+f9 and f9). So if you ignore exception and pass it with DBG_EXCEPTION_NOT_HANDLED, the one raised by NtRaiseException will work okay, but if you pass exception to debugy when it is generated by CloseHandle you will call thread exception handler when it is not supposed to occur, in such way SEH which might be used to decrypt part of code will actually decrypt wrong part of code, or simply redirect you to ExitProcess.

To prove my theory and show how protection developers are stupid I will show you a little proof of concept code, I even heard that some of them used this trick after it was leaked but only with CloseHandle(dummy_handle), so lame, so lame, I would never protect my software with such protection, never:

```
start:         push    offset sehhandle1
               push    dword ptr fs:[0]
               mov     dword ptr fs:[0], esp

               mov     ctx.context_ContextFlags, 10007h
               mov     ctx.context_esp, esp
               mov     ctx.context_eip, offset __debugged
               mov     ctx.context_segCs, cs
               mov     ctx.context_segDs, ds
               mov     ctx.context_segFs, fs
               mov     ctx.context_segEs, es
               mov     ctx.context_segSs, ss

               push    1
               push    offset ctx
               push    offset exception
               callW   NtRaiseException

__safe0:       pop     dword ptr fs:[0]
               add     esp, 4

               push    offset sehhandle2
               push    dword ptr fs:[0]
               mov     dword ptr fs:[0], esp

               push    0deadc0deh
               callW   CloseHandle
               pop     dword ptr fs:[0]
               add     esp, 4

               push    40h
               push    offset stitle
               push    offset sabout
               push    0
               callW   MessageBoxA

               push    0
               callW   ExitProcess


sehhandle1:    xor     eax, eax
               mov     ecx, [esp+0ch]
               mov     [ecx.context_eip], offset __safe0
               retn

sehhandle2:    xor     eax, eax
               mov     ecx, [esp+0ch]
               mov     [ecx.context_eip], offset __debugged
               retn


__debugged:    push    10h
               push    offset dabout
               push    offset dtitle
               push    0
               callW   MessageBoxA
               push    0
               callW   ExitProcess

dabout         db      "debugged", 0
dtitle         db      "kill your ring3 debugger, and try again",0
stitle         db      "good", 0
sabout         db      "your are ok", 0
```

Oki first we have to fill context struct with needed data, we store there ESP, EIP, segment registers and exception flags. Note that EIP is set to point to __debugged label, that is because we are telling process to go there (simulating that exception occurred at that EIP), also we call NtRaiseException passsing to it pointer of exception code (0c0000008h), pointer to context structure and 1 (bool HandleException, if set to 1 call thread exception handler), so running this code in OllyDbg with unchecked exceptions will stop here:



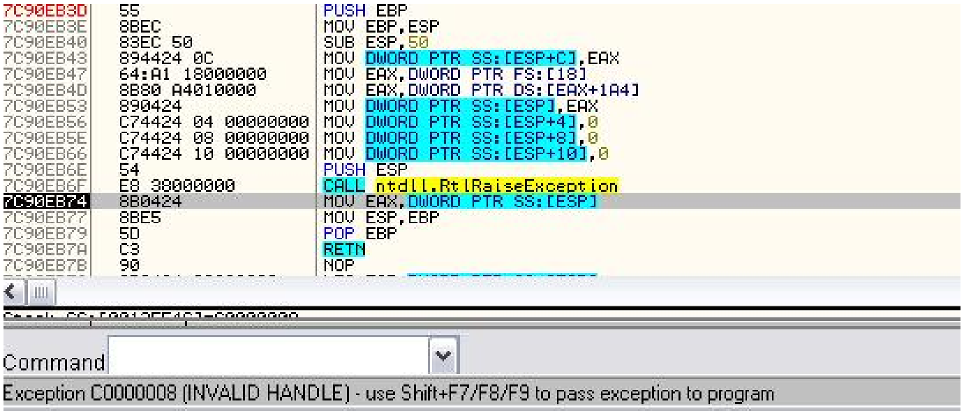
```
004010A1  .  33C0              XOR EAX,EAX
004010A3  .  8B4C24 0C         MOV ECX,DWORD PTR SS:[ESP+C]
004010A7  .  C781 B8000000 5A1 MOV DWORD PTR DS:[ECX+B8],poc.0040105A
004010B1  .  C3                RETN
004010B2  .  33C0              XOR EAX,EAX
004010B4  .  8B4C24 0C         MOV ECX,DWORD PTR SS:[ESP+C]
004010B8  .  C781 B8000000 C31 MOV DWORD PTR DS:[ECX+B8],poc.004010C3
004010C2  .  C3                RETN
004010C3  .  6A 10             PUSH 10
004010C5  .  68 DD104000       PUSH poc.004010DD
004010CA  .  68 E6104000       PUSH poc.004010E6
004010CF  .  6A 00             PUSH 0
004010D1  .  E8 5B000000       CALL <JMP.&USER32.MessageBoxA>
004010D6  .  6A 00             PUSH 0
004010D8  .  E8 48000000       CALL <JMP.&KERNEL32.ExitProcess>
004010DD  .  64 65 62 75 67 67 ASCII "debugged",0
004010E6  .  6B 69 6C 6C 20 79 ASCII "kill your ring3 "
004010F6  .  64 65 62 75 67 67 ASCII "debugger, and tr"
00401106  .  79 20 61 67 61 69 ASCII "y again",0
0040110E  .  67 6F 6F 64 00    ASCII "good",0
00401113  .  79 6F 75 72 20 61 ASCII "your are ok",0
0040111F  $- FF25 6C304000     JMP DWORD PTR DS:[<&ntdll.NtRaiseException>]
00401125  .- FF25 74304000     JMP DWORD PTR DS:[<&KERNEL32.ExitProcess>]
0040112B  $- FF25 78304000     JMP DWORD PTR DS:[<&KERNEL32.CloseHandle>]
00401131  $- FF25 80304000     JMP DWORD PTR DS:[<&USER32.MessageBoxA>]
00401137     00                DB 00
00401138     00                DB 00
```

Command

Exception C0000008 (INVALID HANDLE) - use Shift+F7/F8/F9 to pass exception to program

You see, EIP is set to __debugged label, if we press F9 (DBG_CONTINUE) we will end up in bad boy message and ExitProcess, but if we press shift+f9 to call thread exception handler, what will occur in normal condition when NtRaiseException is called,
we will call sehhandle1 which is at : 4010A1h, that part of code will redirect execution to __safe0 label in above source. And we have simulated normal conditions.

But soon we have call to CloseHandle with dummy handle to generate this exception again:



```
7C90EB3D   55                 PUSH EBP
7C90EB3E   8BEC               MOV EBP,ESP
7C90EB40   83EC 50            SUB ESP,50
7C90EB43   894424 0C          MOV DWORD PTR SS:[ESP+C],EAX
7C90EB47   64:A1 18000000     MOV EAX,DWORD PTR FS:[18]
7C90EB4D   8B80 A4010000      MOV EAX,DWORD PTR DS:[EAX+1A4]
7C90EB53   890424             MOV DWORD PTR SS:[ESP],EAX
7C90EB56   C74424 04 00000000 MOV DWORD PTR SS:[ESP+4],0
7C90EB5E   C74424 08 00000000 MOV DWORD PTR SS:[ESP+8],0
7C90EB66   C74424 10 00000000 MOV DWORD PTR SS:[ESP+10],0
7C90EB6E   54                 PUSH ESP
7C90EB6F   E8 38000000        CALL ntdll.RtlRaiseException
7C90EB74   8B0424             MOV EAX,DWORD PTR SS:[ESP]
7C90EB77   8BE5               MOV ESP,EBP
7C90EB79   5D                 POP EBP
7C90EB7A   C3                 RETN
7C90EB7B   90                 NOP
```

Command

Exception C0000008 (INVALID HANDLE) - use Shift+F7/F8/F9 to pass exception to program

Oki, CloseHandle generated new exception and NOW we should only press F9 (DBG_CONTINUE) and avoid calling of installed handler which is at : 4010B2h and will redirect EIP to __debugged label.

So to sum this up when the exception is raised via NtRaiseException we have to pass the exception to debugger with DBG_EXCEPTION_NOT_HANDLED, but when such exception is generated using CloseHandle we have to pass exception with DBG_CONTINUE. So let me think. You may NOT set OllyDbg or your debug loader to pass this exception only with DBG_EXCEPTION_NOT_HANDLED or DBG_CONTINUE, if you do that; you will be caught using this trick. I guess that protection developers and person whom I gave hint didn't figure this yet, that shows only their knowledge to exploit some bugs in Windows system. Maybe this isn't a bug; maybe this exception is generated purposely for software developers to check when they have changed something in their source code during debugging. But luckily MS didn't think that this can be used to detect ring3 debuggers, and here you go.

3. Final Remarks

Future protection system should use this exception combined with NtRaiseException in more than 10 places of their protection system, just to avoid a simple passing of this exception to debug.

Also this exception can be generated directly using sysenter and INT 2eh to communicate with ntoskrnl.exe because CloseHandle (NtCloseHandle) and NtRaiseException are both called via sysenter and INT 2eh on Win2k systems. Only problem is to determine on which system are you running, but you can simply open ntdll.dll and read both function from it into some buffer and call them directly. Or use this macro for XP to call directly native APIs (as I did in prc-ko.xp virus – proof of concept):

```
@sysenter          macro  syscall, parameters
                   local  __@@1, __@@2
                   push   eax
                   jmp    __@@2
__@@1:
                   mov    eax, syscall
                   mov    edx, esp
                   dw     340Fh                      ;sysenter 0F34h
__@@2:
                   call   __@@1
                   add    esp, (parameters*4) + 4    ;+ 1 dummy EIP
endm
```

Well that's all..

10.PEB Dll Hooking, a novel method to hook dlls, deroko of ARTeam

1. Introduction

This will be a very short article, because I'm only showing the idea and how to do it. There is no need for me to write 20 pages to show you simple trick called PEB Dll hooking.

2. Method

Before you ask why is this a good method to attack protectors you first have to know how they work! Every normal protector will hide APIs that it will use during unpacking of our target. It is just how they work; trying to make a static analysis of a protector is a little bit harder. To be able to communicate with the kernel, protector has to call some APIs. Most of the times those are the APIs exported by kernel32.dll, and some protectors are also using exports of ndll.dll to detect debuggers or to fool them. They don't import APIs, most of the times they only import one or several APIs just to make PE file win2k compatible (win2k won't run exe w/o at least one import), instead of using import table, they will use GetProcAddress or custom implementation of GetProcAddress to find APIs (some using CRC some by names). To get base of kernel32.dll protectors will use several tricks that are common for locating k32 base:

- GetModuleHandleA/W
- LoadLibraryA/W
- PEB scanning
- K32 address on stack at entry point
- walking trough SEH chain where last record is pointing to k32.dll
- use MZ loop on 1 imported API from k32.dll

Imagine now scenario where each call to GetModuleHandleA("kernel32.dll") will return address of your .dll, when GetProcAddress or custom implementation of GetProcAddress is used it will actually scan your .dll and locate APIs in it and you may do what ever you want with APIs. To avoid hooking of GetModuleHandle and LoadLibrary we can go deeper and mess with PEB and actually hijack .dll via PEB hooking.

Lets take a look what is important for us:

```
kd> dt nt!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x01c EnvironmentPointer : Ptr32 Void
   +0x020 ClientId         : _CLIENT_ID
   +0x028 ActiveRpcHandle  : Ptr32 Void
   +0x02c ThreadLocalStoragePointer : Ptr32 Void
   +0x030 ProcessEnvironmentBlock : Ptr32 _PEB
   +0x034 LastErrorValue   : Uint4B
   +0x038 CountOfOwnedCriticalSections : Uint4B
   +0x03c CsrClientThread  : Ptr32 Void
```

At offset +30h of TEB (Thread Environment Block) is located PEB (Process Environment Block), which will describe state of process in memory. There is plenty of nice information in PEB, but we are interested in PEB_LDR_DATA here:

```
kd> dt nt!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
   +0x003 SpareBool        : UChar
   +0x004 Mutant           : Ptr32 Void
   +0x008 ImageBaseAddress : Ptr32 Void
   +0x00c Ldr              : Ptr32 _PEB_LDR_DATA
   +0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
```

PEB_LDR_DATA is a simple structure that will describe the state of each loaded module for this process (.dlls, main proggy itself), also this structure is being accessed via GetModuleHandle, and ntdll!LdrLoadDll (internally called by LoadLibrary) it looks like this:

```
kd> dt nt!_PEB_LDR_DATA
   +0x000 Length           : Uint4B
   +0x004 Initialized      : UChar
   +0x008 SsHandle         : Ptr32 Void
   +0x00c InLoadOrderModuleList : _LIST_ENTRY
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY
   +0x024 EntryInProgress  : Ptr32 Void
```

These lists are actually used to locate lists of LDR_MODULE or LDR_DATA_TABLE_ENTRY structures which will describe the state of each loaded module:

```
kd> dt nt!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY
   +0x008 InMemoryOrderLinks : _LIST_ENTRY
   +0x010 InInitializationOrderLinks : _LIST_ENTRY
   +0x018 DllBase          : Ptr32 Void
   +0x01c EntryPoint       : Ptr32 Void
   +0x020 SizeOfImage      : Uint4B
   +0x024 FullDllName      : _UNICODE_STRING
   +0x02c BaseDllName      : _UNICODE_STRING
   +0x034 Flags            : Uint4B
   +0x038 LoadCount        : Uint2B
   +0x03a TlsIndex         : Uint2B
   +0x03c HashLinks        : _LIST_ENTRY
   +0x03c SectionPointer   : Ptr32 Void
   +0x040 CheckSum         : Uint4B
   +0x044 TimeDateStamp    : Uint4B
   +0x044 LoadedImports    : Ptr32 Void
   +0x048 EntryPointActivationContext : Ptr32 Void
   +0x04c PatchInformation : Ptr32 Void
kd>
```

Because GetModuleHandle or LoadLibrary will use these lists (as well as some other win apis resposnbile for modules enumeration) we may fake DllBase, EntryPoint and SizeOfImage and GetModuleHandle will return base of our hooking .dll. You may already see the weak side of this approach, if we are faking .dll in PEB, we also have to export same APIs as hooked .dll, if we don't do so GetProcAddress will fail on our hooking .dll. To make my work easier instead of typing manually all exports of certain .dll I created simple proggy called dllcreator.c. It will make .asm/.def/.inc skeleton for my hooking .dll and all that is left to do is to add .dll entry point which will perform PEB hooking and to choose which APIs I'll hook.

Hooking .dll is very simple when you know how to walk through PEB_LDR_DATA, I will show you my hook from DllEntry point (weird how people chose to really name entry point in .dll DllEntry, it doesn't matter as long as linker knows where is entry point, I use start ☺)

```
public C start
start           proc
                arg     imagebase
                arg     reason
                arg     reserved
```

1st we have to check reason, if it is PROCESS_ATTACH we perform hooking, otherwise, we simple exit from dll entry point callback:

```
        pusha
        cmp     reason, 1
        jne     __e_dllinit
```

Next step is to locate InLoadOrderModuleList (if you chose other, you will have to calucalte negative offset to struct start and then you may access modules using your predefined struct, or if you like to make code less readable use indexing and don't bother with calculating negative offsets ☺)

```
        mov     eax, dword ptr fs:[30h]
        mov     eax, [eax+0ch]
        mov     esi, [eax+0ch]
```

Now we have to get base of kernel32.dll so we can walk trough LDR_MODULE and hook our kernel32.dll (note that I'm using LoadLibraryA, because this is my skeleton, and if I'm hooking some other .dll that is not loaded I have to use LoadLibraryA before I can hook it):

```
        call    LoadLibraryA, offset szkernel32
        mov     old_dll_base, eax
        xchg    eax, ebx
```

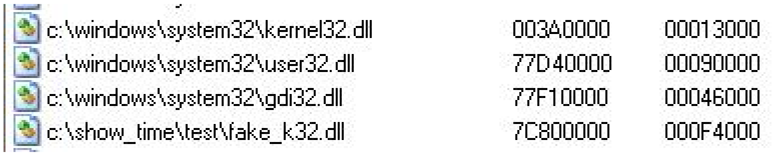
Now we simply walk trough LDR_MODULE and we search for our hooking dll, and .dll that we wanna hook:

```
__find_dll:     cmp     [esi.lm_baseaddress], ebx
                je      __esiedi
                lodsd
                xchg    eax, esi
                jmp     __find_dll

__esiedi:       cmp     ebx, imagebase
                je      __hook
                mov     edi, esi
                mov     ebx, imagebase
                jmp     __find_dll
```

At this point edi is pointing to LDR_MODULE of target .dll and esi is pointing to LDR_MODULE of our hooking .dll, all we have to do is exchange data between these 2 structs so our hooking .dll becomes our target .dll and vice verse.

```
__hook:         mov     eax, ebx
                xchg    eax, [edi.lm_baseaddress]
                mov     [esi.lm_baseaddress], eax

                add     ebx, [ebx+3ch]
                mov     eax, [ebx.pe_addressofentrypoint]
                add     eax, imagebase
                xchg    eax, [edi.lm_entrypoint]
                mov     [esi.lm_entrypoint], eax

                mov     eax, [ebx.pe_sizeofimage]
                xchg    eax, [edi.lm_sizeofimage]
                mov     [esi.lm_sizeofimage], eax

__e_dllinit:    popa
                mov     eax, 1
                leave
                retn    0ch
                endp
```

Voila, kernel32 is hooked via PEB, here is snippet from LordPE:



```
c:\windows\system32\kernel32.dll       003A0000    00013000
c:\windows\system32\user32.dll         77D40000    00090000
c:\windows\system32\gdi32.dll          77F10000    00046000
c:\show_time\test\fake_k32.dll         7C800000    000F4000
```

Do you see any difference? Yes you do… original kernel32.dll is now named as fake_k32.dll, and fake_k32.dll is now kernel32.dll ☺

Oki, this attack is good, works great, but we face one big problem here. Our loaded fake k32 won't be used to fill import table of our target ☹ It will be used later on when GetModuleHandleA or LoadLibraryA is used, which is bad because there are some protectors that will use imports to locate base of kernel32.dll or other used .dll. Before we come to the solution to this problem we will have to know what is really going on when a new process is created. It is important, very very important to understand.

I will only briefly describe what is going on, when we call CreateProcessA/W, internally there will be called NtCreateProcess to map file in memory and also to map ntdll.dll. After this gets done, we are back to ring3 and then new thread is being created. During Thread creation windows will use APC to call ntdll!LdrInitializeThunk which is responsible for walking trough import descriptor and will load all needed libraries and fill IAT. New Thread creation is here:

```
.text:7C819A3C                 call    _BaseInitializeContext@20
...
.text:7C819A9C                 push    eax
.text:7C819A9D                 call    ds:__imp__NtCreateThread@32
```

And in _BaseInitializeContext@20 it will set EIP to point to:

```
.text:7C8105AF                 cmp     [ebp+arg_14], 1
.text:7C8105B3                 mov     [eax+CONTEXT.Eax], ecx
.text:7C8105B9                 mov     ecx, [ebp+arg_8]
.text:7C8105BC                 mov     [eax+CONTEXT.Ebx], ecx
.text:7C8105C2                 push    20h
.text:7C8105C4                 pop     ecx
.text:7C8105C5                 mov     [eax+CONTEXT.SegEs], ecx
.text:7C8105CB                 mov     [eax+CONTEXT.SegDs], ecx
.text:7C8105D1                 mov     [eax+CONTEXT.SegSs], ecx
.text:7C8105D7                 mov     ecx, [ebp+arg_10]
.text:7C8105DA                 mov     [eax+CONTEXT.SegFs], 38h
.text:7C8105E4                 mov     [eax+CONTEXT.SegCs], 18h
.text:7C8105EE                 mov     [eax+CONTEXT.EFlags], 3000h
.text:7C8105F8                 mov     [eax+CONTEXT.Esp], ecx
.text:7C8105FE                 jnz     loc_7C814D67
.text:7C810604                        mov     [eax+CONTEXT.Eip], offset
_BaseThreadStartThunk@8
.text:7C81060E
.text:7C81060E loc_7C81060E:
.text:7C81060E                 add     ecx, 0FFFFFFFCh
.text:7C810611                 mov     [eax+CONTEXT.ContextFlags], 10007h
.text:7C810617                 mov     [eax+CONTEXT.Esp], ecx
.text:7C81061D                 pop     ebp
.text:7C81061E                 retn    14h
.text:7C81061E _BaseInitializeContext@20 endp
```

BaseThreadStartThunk will call the entry point of our new thread, but TLS callbacks are executed from LdrInitilizeThunk called by APC during Thread creation.

```
.text:7C80B4D4                 push    10h
.text:7C80B4D6                 push    offset dword_7C80B518
.text:7C80B4DB                 call    __SEH_prolog
.text:7C80B4E0                 and     dword ptr [ebp-4], 0
.text:7C80B4E4                 mov     eax, large fs:18h
.text:7C80B4EA                 mov     [ebp-20h], eax
```

```
.text:7C80B4ED                 cmp     dword ptr [eax+10h], 1E00h
.text:7C80B4F4                 jnz     short loc_7C80B505
.text:7C80B4F6                 cmp     _BaseRunningInServerProcess, 0
.text:7C80B4FD                 jnz     short loc_7C80B505
.text:7C80B4FF                 call    ds:__imp__CsrNewThread@0
.text:7C80B505                 push    dword ptr [ebp+0Ch]
.text:7C80B508                 call    dword ptr [ebp+8]  <-- call entrypoint
.text:7C80B50B                 push    eax
.text:7C80B50C                 call    _ExitThread@4
```

Now to force NT loader to load fake_k32.dll instead of real kernel32.dll we will use a little magic here in LdrInitializeProcess (called by LdrInitializeThunk which is executed using APC):

```
.text:7C9222F4                 mov     word ptr [ebp+var_100], 18h
.text:7C9222FD                 mov     word ptr [ebp+var_100+2], 1Ah
.text:7C922306                 mov     [ebp+var_FC], offset aKernel32_dll
.text:7C922310                 call    _LdrpLoadDll@24
…
.text:7C922538 aKernel32_dll:
.text:7C922538                 unicode 0, <kernel32.dll>,0
.text:7C922552                 align 4
```

When the thread is suspended APC is not ran yet, at this point we may mess with LdrInitilaizeThunk as much as we want, APC will be executed once new thread is resumed. At this point we may hardcode the value of kernel32.dll in our loader and use it to overwrite Unicode string "kernel32.dll" with our fake_k32.dll and force LdrInitilizeThunk to fill imports of kernel32.dll with our exports from fake_k32.dll. I personally couldn't find any good way to scan for this value because there are several occurrences of Unicode string "kernel32.dll" in ntdll.dll and I don't know if order of "good" strings is changed in older or newer versions of ntdll.dll. So you will have to find this value by yourself and hardcode it in a loader.

Here is how it looks like in one upx packed executable when loader –b is used (hooking also in LdrInitilizeThunk):

```
001B:00413F77  CALL      [ESI+000140A8]
001B:00413F7D  OR        EAX,EAX
001B:00413F7F  JZ       _00413F88
001B:00413F81  MOV       [EBX],EAX
001B:00413F83  ADD       EBX,04
001B:00413F86  JMP      _00413F69
001B:00413F88  CALL      [ESI+000140AC]
001B:00413F8E  POPAD
...
001B:003A1CD2  JMP       [KERNEL32!GetProcAddress] <-- hooked import
001B:003A1CD8  RET
...
KERNEL32!GetProcAddress
001B:7C80AC28  MOV       EDI,EDI
001B:7C80AC2A  PUSH      EBP
001B:7C80AC2B  MOV       EBP,ESP
001B:7C80AC2D  PUSH      ECX
001B:7C80AC2E  PUSH      ECX
001B:7C80AC2F  PUSH      EBX
001B:7C80AC30  PUSH      EDI
```

Bingo, imports are hooked with my fake_k32.dll and now I can log action of protector w/o a problem.

Well that's it, no more to talk about PEB dll hooking, I hope you got the idea? If not, check the sources, they will help you ☺

If you want to get maximum stealth, erase hooked .dll from list entries, it is not hard, just requires walking trough all 3 list entries and unlinking the ones that are pointing to original .dll, in such way you will have only kernel32 or other .dll loaded while fake_xxx.dll will be gone from modules list.

11.TheMida: no more Ring0?, deroko of ARTeam

1. Introduction

Well old news, TheMida isn't using ring0 anymore to hook IDT or SDT so we may use SoftICE to play with the new TheMida. Oki, we take some target protected by new TheMida (APIMonitor, SilhouetteFX, ExactSpent …) and we start it without SoftICE and it works, we check IDT in WARK[24] and everything is normal, we check also SDT, yep everything is normal there too.

2. Debugging a target

So we start our SoftICE and run application. Amazing, debugger detected. Heh, funny, TheMida is running without a problem in OllyDbg but it can't run while SoftICE is active. So what is going on?

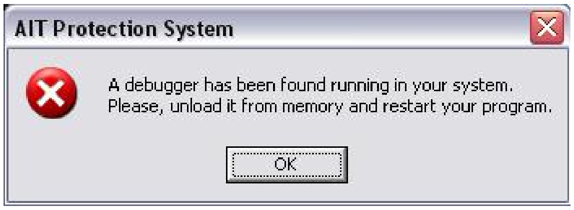
Well themida developers have abandoned offensive ring0 driver and they are using now nice, SoftICE friendly driver, but still SoftICE is detected. There are a few tricks to detect SoftICE from ring3:

- UnhandledExceptionFilter
- INT 1h
- INT 41h
- INT 3h
- CreateFile
- NtQuerySystemInformation

I saw INT 1h used and also NtQuerySystemInformation while I was debugging TheMida, INT 1h is used to simple avoid single stepping. I patched INT 1h/INT 41h to DPL of 0 to avoid SoftICE detection and also hooked NtCreateFile to avoid SoftICE detection via CreateFile.

NtQuerySystemInformation is used in TheMida to get base/range of ntoskrnl.exe, hal.dll and win32k.sys drivers (main windows OS components ☺), didn't see for what it is using them but it was obvious that it isn't using ring3 to detect SoftICE. So the only solution is that TheMida is using its own drivers to detect the presence of SoftICE. There are a few tricks to detect the presence of SoftICE using a driver.
SafeCast, for example, uses distance between INT 1/INT 3 and DR7 to detect the presence of SoftICE, but the story with TheMida is much more interesting!

Oki, first we have to see how message looks like (not very descriptive):

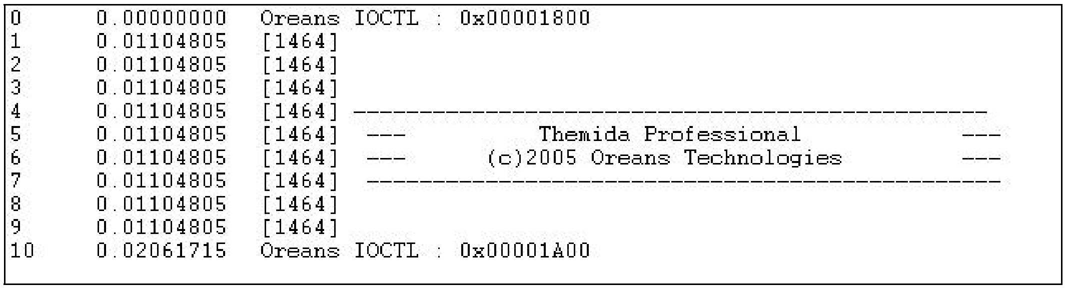


AIT Protection System

A debugger has been found running in your system.
Please, unload it from memory and restart your program.

If you break in MessageBoxA you will only see that a retn is taking the program to ExitProcess: hunting the debugger detection will not be an easy task!

Now let's see what IOCTL codes is TheMida using before it detects our debugger:



```
0       0.00000000   Oreans IOCTL : 0x00001800
1       0.01104805   [1464]
2       0.01104805   [1464]
3       0.01104805   [1464]
4       0.01104805   [1464] ------------------------------------------------
5       0.01104805   [1464]  ---             Themida Professional             ---
6       0.01104805   [1464]  ---       (c)2005 Oreans Technologies            ---
7       0.01104805   [1464]  ------------------------------------------------
8       0.01104805   [1464]
9       0.01104805   [1464]
10      0.02061715   Oreans IOCTL : 0x00001A00
```

[24] WARK, http://www.zero-g.it/RE/exetools/Wark13.rar

And so? Lets break into the driver when IOCTL 0x1A00 is used, if you try to break at DeviceIoControl it simply won't work, the reason for this is again simple, if you have read my tutorial about TheMida with oreans.sys[25] you could see that TheMida rebases some dlls and makes breaking in APIs a little bit harder. So we are going to break at ntoskrnl!NtDeviceIoControlFile and see what is really going on when IOCTL code 0x1A00 is used:

Here we go, 2nd break at IopXxxControlFile (internaly called by NtDeviceIoControlFile):

```
0008:80578BF5  CALL      _IopXxxControlFile
0008:80578BFA  POP       EBP
```

And on the stack:

```
0010:F2C15D08 000000EC  00000000  00000000  00000000
0010:F2C15D18 0013FF2C  00001A00  00AA5172  00000010
```

Now you see IOCTL = 0x1A00 so we enter into driver:

```
0008:F86F72A0  PUSH      EBP
0008:F86F72A1  MOV       EBP,ESP
0008:F86F72A3  ADD       ESP,-04
0008:F86F72A6  PUSH      ESI
0008:F86F72A7  PUSH      EDI
0008:F86F72A8  PUSH      EBX
0008:F86F72A9  MOV       EDI,[EBP+0C] <--- PIRP
0008:F86F72AC  XOR       EAX,EAX
```

And we continue our quest till we find where it is playing with IOCTL = 0x1A00h:

```
0008:F86F74C0  CMP       DWORD PTR [ESI+0C],00001800
0008:F86F74C7  JZ       _F86F74D3
0008:F86F74C9  CMP       DWORD PTR [ESI+0C],00
0008:F86F74CD  JNZ      _F86FAB4E
```

Nope, continue:

```
0008:F86FAB4E  CMP       DWORD PTR [ESI+0C],00001801
0008:F86FAB55  JNZ      _F86FAE80
```

Nope, continue:

```
0008:F86FAE80  CMP       DWORD PTR [ESI+0C],00001802
0008:F86FAE87  JNZ      _F86FAED1
```

Nope, continue:

```
0008:F86FAED1  CMP       DWORD PTR [ESI+0C],00001D00
0008:F86FAED8  JNZ      _F86FB938
```

C'mon…

```
0008:F86FB938  CMP       DWORD PTR [ESI+0C],00001A00
0008:F86FB93F  JNZ      _F86FC07A
0008:F86FB945  MOV       ESI,[EDI+0C]  <--- IRP.SystemBuffer
```

Oh, finaly, as you may see ESI point to irp.irp_systembuffer and then, *BOOM*:

```
0008:F86FBE66  CALL      [ESI]
```

It is redirecting execution to code pointed by SystemBuffer, well look where it goes:

```
0008:00AD12E1  PUSH      EBP
0008:00AD12E2  CALL     _00AD12E7
0008:00AD12E7  POP       EBP
0008:00AD12E8  SUB       EBP,06823EE8
0008:00AD12EE  JMP      _00AD1304
```

[25] Deroko, TheMida Defeating Ring0, http://tutorials.accessroot.com

```
0008:00AD12F3  MOV       EDI,37CE5484
0008:00AD12F8  JGE      _00AD1309
0008:00AD12FA  INC       EBX
```

Ohoho, 0x1A00 is only gateway for ring3 code to become ring0 code ☺ Luckily this is used only 2 times in TheMida so lets trace this code because it seems like this is going to perform some debugger checks (what else would be the reason for new TheMida to use Ring0). After a little bit of tracing we find very interesting stuff here:

```
0008:00AD150E  CMP       BYTE PTR [ECX],68
0008:00AD1511  JNZ      _00AD162D
```

Well ECX is pointing to address of INT 41h and then TheMida checks for push instruction. When SoftICE is not loaded int 41h will point to HalpDispatchInterrupt, but when SoftICE is loaded we will have this hook code:

```
0008:F3645662  PUSH      _HalpDispatchInterrupt
0008:F3645667  JMP      _F358BACB
0008:F364566C  SUB       EAX,8003F400
0008:F3645671  PUSH      F4868B5C
0008:F3645676  JMP      _F35FE601
```

And HalpDispatchInterupt looks like:

```
hal!HalpDispatchInterrupt:
806e79cc 54               push    esp
806e79cd 55               push    ebp
806e79ce 53               push    ebx
806e79cf 56               push    esi
806e79d0 57               push    edi
806e79d1 83ec54           sub     esp,0x54
806e79d4 8bec             mov     ebp,esp
806e79d6 89442444         mov     [esp+0x44],eax
```

TheMida is scanning for SoftICE hook in INT 41h if there is push (68h) debugger detected, otherwise, everything is just fine. So let's go and write simple hook to make our SoftICE invisible for this scan, shall we?

Use ExAllocatePool and allocate small piece of memory because we only need 7 bytes to make our patch (or we can find some unused place in ntoskrnl.exe and assemble our patch there), here is patch anyway:

```
:idt 41
0041  IntG32   0008:81C1E040  DPL=0  P

:u 81c1e040
0008:81C1E040  NOP
0008:81C1E041  PUSH    F3645662
0008:81C1E046  RET
0008:81C1E047  ADD     [EAX],AL

:u f3645662
0008:F3645662  PUSH    _HalpDispatchInterrupt
0008:F3645667  JMP     _F358BACB
```

Run TheMida protected application (the one with oreans32.sys) and it will start w/o a problem.

3. Final Remarks

If you don't want to write your own hooking "engine", you can use the loader supplied with this document, but IMHO, people that are using SoftICE already know driver programming so… I feel lame for providing a loader with this document..

Well I wish to thank to ARTeam, ma mates ☺, Snow Panther for cool DS 3.2 patches, 29a for the best eZine (apart from this one!) and, of course, you for reading this small contribution.

Note that into the supplements folder relative to this paper you can also find a plugin for xADT which implements this anti-debugger trick (int_hooks.dll), follow xADT distribution documentation to install it..

12. WTM Register Maker v2.0 case study, tHE mUTABLE

1. Abstract

In this journey we are going to analyze *WTM Register Maker v2.0: http://www.webtoolmaster.com*, which manages your serials for your shareware, protects your exe files against cracking with crypto technology. And had a lot of nice features like: serial is needed for extract protected exe file, small loader, fast, there is no way to sniffing right serial / only brutal force, protect your software against cracking/hacking.

So, the objective of this work is trying to demystify and annihilate how *WTM Register Maker* works. To do this, we will step through many levels of protection elimination starting with unpacking, cracking, inline patching, aesthetical modifications, where another tools like HzorInline and aPE failed to accomplish their task (in their automated configurations).

In this work I managed to think of the most optimized solution (nothing new) especially when it comes to inline patching (you'll see later why) to defeat the nag screen from the loaders. Having said that, another approach will be explained also just for completeness by trying to explain its advantages and disadvantages. *Why this and why not that*.

The methods used to perform this task, that is, analytical, numerical, and experimental.

2. The Anatomy of Destruction

How WTM Register maker works, it does depends on the loader static linking! No metamorphism at all (the same implementation for every time) only the General program serial key differs. Browse the folder where you installed it and you will notice that there are two files (load.dat and load2.dat) which responsible for adding the layer of protection to the protected file (the only limitation in this shareware version is the nag screen added to the loaders, when it's defeated the program is full). These two files are real executable so don't got confused with .dat extension, try to rename it to load.exe and load2.exe and they'll run normally as any other executable file, and by the way they are both packed with PECompact too.

2.1. Why and When load.dat or load2.dat

If you noticed both are the same sizes *(92.0 KB)*, but if you do a hex comparison using WinHex v13.0 SR-12 a total of *9,049* differences found (of course after unpacking). So, definitely they are not the same.

You can check this by yourself following renaming trial method inside Olly through a process of protecting a demo program. The conclusion:

Under the File tab there is a check box option [***Encrypt only the first 100kb of your protected file***], the activity of this option and the size of the file to be protected determine whether to link *load.dat* or load2.dat.

- If this option is **C**hecked and the file to be protected is less than 100kb RTool Engine will stick to load2.dat for registration scheme loader, otherwise (**U**nchecked) *load.dat* will be used.

- If this option is **U**nchecked and the file to be protected is greater than 100kb RTool Engine will stick to load.dat loader, otherwise (**C**hecked) load2.dat will be used. Check Image 12.1.

- For further investigation check this area (Delphi compiler. Map file applied from DeDe)

```
004C47F6  |> \8D55 E0            LEA EDX,[LOCAL.8]
004C47F9  |.  33C0               XOR EAX,EAX
004C47FB >|.  E8 D0E0F3FF        CALL RToolD.004028D0
                                 ; system.ParamStr(Integer):String
004C4800  |.  8B45 E0            MOV EAX,[LOCAL.8]
004C4803  |.  8D55 E4            LEA EDX,[LOCAL.7]
```

```
004C4806 >|.  E8 413FF4FF         CALL RToolD.0040874C
                                  ; sysutils.ExtractFilePath(AnsiString):AnsiString
004C480B  |.  8D45 E4             LEA EAX,[LOCAL.7]
004C480E  |.  BA 344B4C00         MOV EDX,RToolD.004C4B34    ; ASCII "load.dat"
004C4813 >|.  E8 44F5F3FF         CALL RToolD.00403D5C       ; system.@LStrCat
004C4818  |.  8B45 E4             MOV EAX,[LOCAL.7]
004C481B >|.  E8 843EF4FF         CALL RToolD.004086A4
                                  ; sysutils.FileExists(AnsiString):Boolean
004C4820  |.  84C0                TEST AL,AL
004C4822  |.  75 0F               JNZ SHORT RToolD.004C4833
004C4824  |.  B8 484B4C00         MOV EAX,RToolD.004C4B48    ;  ASCII "Loader not found."
004C4829 >|.  E8 76F1F9FF         CALL RToolD.004639A4
                                  ; dialogs.ShowMessage(AnsiString)
004C482E  |.  E9 3B020000         JMP RToolD.004C4A6E
004C4833  |>  8D55 DC             LEA EDX,[LOCAL.9]
004C4836  |.  8B83 38030000       MOV EAX,DWORD PTR DS:[EBX+338]
004C483C >|.  E8 0720FDFF         CALL RToolD.00496848
                            ; mask.TCustomMaskEdit.GetText(TCustomMaskEdit):AnsiString
004C4841  |.  8B45 DC             MOV EAX,[LOCAL.9]
004C4844 >|.  E8 CF3CF4FF         CALL RToolD.00408518
                                  ; sysutils.StrToInt(AnsiString):Integer;
                                  ; Hex the General Serial Number
004C4849  |.  8BF0                MOV ESI,EAX
004C484B  |.  8D55 D8             LEA EDX,[LOCAL.10]
004C484E  |.  8B83 44030000       MOV EAX,DWORD PTR DS:[EBX+344]
004C4854 >|.  E8 EF1FFDFF         CALL RToolD.00496848
                            ; mask.TCustomMaskEdit.GetText(TCustomMaskEdit):AnsiString
004C4859  |.  8D45 D8             LEA EAX,[LOCAL.10]
004C485C  |.  BA 644B4C00         MOV EDX,RToolD.004C4B64      ; ASCII ".ver"
004C4861 >|.  E8 F6F4F3FF         CALL RToolD.00403D5C         ; system.@LStrCat;
004C4866  |.  8B45 D8             MOV EAX,[LOCAL.10]           ; .ver version of the
                                                               ; original file
004C4869  |.  50                  PUSH EAX                     ; .ver created
004C486A  |.  8D55 D4             LEA EDX,[LOCAL.11]
004C486D  |.  8B83 44030000       MOV EAX,DWORD PTR DS:[EBX+344]

.       .      .
.       .      .

004C4A6E  |> \33C0                XOR EAX,EAX
004C4A70  |.  5A                  POP EDX
004C4A71  |.  59                  POP ECX
004C4A72  |.  59                  POP ECX
004C4A73  |.  64:8910             MOV DWORD PTR FS:[EAX],EDX
004C4A76  |.  68 9D4A4C00         PUSH RToolD.004C4A9D
004C4A7B  |>  8D45 94             LEA EAX,[LOCAL.27]
004C4A7E  |.  BA 0E000000         MOV EDX,0E
004C4A83 >|.  E8 70F0F3FF         CALL RToolD.00403AF8  ; system.@LStrArrayClr
004C4A88  |.  8D45 D4             LEA EAX,[LOCAL.11]
004C4A8B  |.  BA 0B000000         MOV EDX,0B
004C4A90 >|.  E8 63F0F3FF         CALL RToolD.00403AF8  ; system.@LStrArrayClr
004C4A95  \.  C3                  RETN
```

Note: I'll assign **C**hecked and **U**nchecked as a representation for whether this option: "Encrypt only the first 100 kb of your protected file" is active or not. And PF (Protected File).

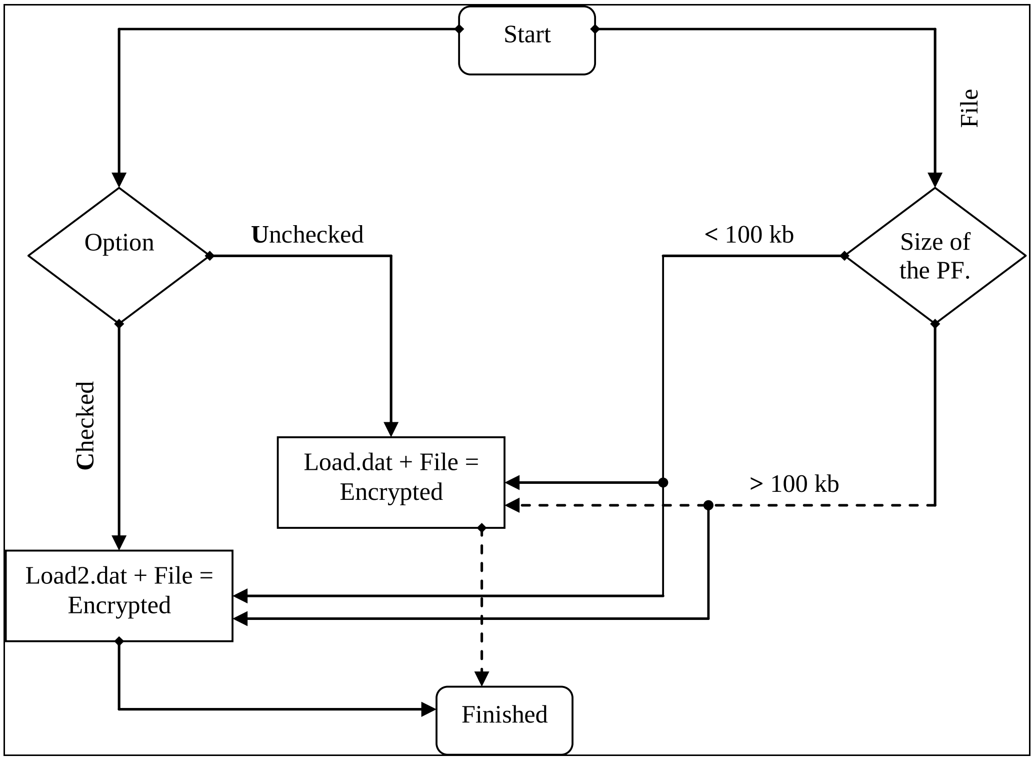


Image 12.1 - load.dat and load2.dat mechanism

3. Unpacking

Load it (*RTool.exe*) in *RDG Packer Detector v0.6.4 BETA* and voila ***PECompact v2.5x – v2.7x.*** In fact nothing new in this case study than adopting the ESP method for unpacking.

3.1. *Unpacking.ESP Method*

Our next step is to load it in Olly and then pressing F8 (Step Over) twice so that the ESP stack pointer is changed to (`ESP 0012FFC4`). In the Register Window: Right-click on the ESP register → Follow in Dump. (ESP register contains the address to the top of the stack).

In the Dump Window: Highlight the first four bytes (`C8 32 4F 00`) and Right-click → Breakpoint → Hardware, on access → Dword. Now OllyDbg will stop when the first four bytes are accessed.
Now Press F9 (Run) Four times wait for the program to be unpacked and we will be break at a JMP. The code looks like the following:

```
004F338A  - FFE0             JMP EAX               ; RTool.004C5D94
004F338C    94               XCHG EAX,ESP
004F338D    5D               POP EBP               ; kernel32.77E814C7
004F338E    4C               DEC ESP
004F338F    00C0             ADD AL,AL
004F3391    334F 00          XOR ECX,DWORD PTR DS:[EDI]
004F3394    D033             SAL BYTE PTR DS:[EBX],1
004F3396    4F               DEC EDI
```

Now press Step into F7 (Enter the JMP EAX address `004F338A`). This will bring us to the OEP (`RTool.004C5D94`). The code looks like the following:

```
004C5D94    55               PUSH EBP
004C5D95    8BEC             MOV EBP,ESP
004C5D97    83C4 F4          ADD ESP,-0C
004C5D9A    B8 AC5B4C00      MOV EAX,RTool.004C5BAC
004C5D9F    E8 C007F4FF      CALL RTool.00406564
004C5DA4    A1 907E4C00      MOV EAX,DWORD PTR DS:[4C7E90]
```

```
004C5DA9     8B00                 MOV EAX,DWORD PTR DS:[EAX]
004C5DAB     E8 7C40F8FF          CALL RTool.00449E2C
004C5DB0     8B0D 947F4C00        MOV ECX,DWORD PTR DS:[4C7F94]    ; RTool.004CA9FC
004C5DB6     A1 907E4C00          MOV EAX,DWORD PTR DS:[4C7E90]
004C5DBB     8B00                 MOV EAX,DWORD PTR DS:[EAX]
004C5DBD     8B15 E43A4C00        MOV EDX,DWORD PTR DS:[4C3AE4]    ; RTool.004C3B30
004C5DC3     E8 7C40F8FF          CALL RTool.00449E44
004C5DC8     A1 907E4C00          MOV EAX,DWORD PTR DS:[4C7E90]
004C5DCD     8B00                 MOV EAX,DWORD PTR DS:[EAX]
004C5DCF     E8 F040F8FF          CALL RTool.00449EC4
004C5DD4     E8 C3DBF3FF          CALL RTool.0040399C
004C5DD9     8D40 00              LEA EAX,DWORD PTR DS:[EAX]
```

Now dump it using OllyDump Plug-in with Rebuild Import check box checked.

Fine, our program is out of prison now.

And do the same steps as above for *load.dat* and load2.dat

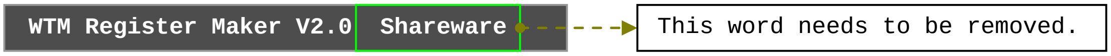
Note: Don't worry for the extension .dat these are real executable files.

4. Cracking (RTool.exe + load.dat + load2.dat) = Full (0x1)

There are many methods to defeat Shareware text string from RTool.exe, Nag screen from load.dat & load2.dat.. But in our case we'll adopt the minimum modification so that the inline patching technique will just work fine. You may ask why not just apply the patches on the unpacked loaders and the nag screen is gone, No the protection loader won't work with the unpacked one even if you enter the correct serial1 and serial2. (try it and you'll see what I mean).

4.1. Aesthetical modification

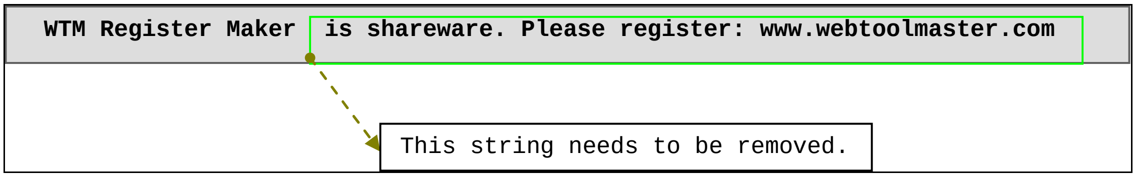
The first target is RTool.exe caption "Shareware" and the About tab Shareware…



WTM Register Maker V2.0 Shareware

This word needs to be removed.

Image 12.2 - RTool caption



WTM Register Maker is shareware. Please register: www.webtoolmaster.com

This string needs to be removed.

Image 12. 3 – RTool About tab string

You wonder why all of this, it's only a string I can search for it using any hex-editor and then delete it. But if you want to apply the patches using inline patching it won't work because *there are only a few bytes available to be used in the inline patching method*. And that's why we seek the minor modification to get our patched version works perfectly without adding any section to the executable file.

Inside Olly Search for all referenced text string then right click → Search for text → and write "Shareware" in the text box with Entire scope option checked and press ok. That's it our first hit → double click and you are in:

```
004DFC3B    .  43 61 70 74 69 6F 6E     ASCII "Caption" : Crystal Clear
004DFC42       06                       DB 06
004DFC43    .  21                       DB 21 ;  CHAR '!'
004DFC44    .  57 54 4D 20 52 65 67 69>ASCII "WTM Register Mak"
004DFC54    .  65 72 20 56 32 2E 30 20>ASCII "er V2.0 Sharewar"
004DFC64    .  65                       ASCII "e"
```

By changing only one byte "Shareware" word will be gone and our mission is accomplished. So, right click on the `004DFC54    .  65 72 20 56 32 2E 30 20>ASCII "er V2.0 Sharewar"` → Follow in Dump.

```
004DFC54  65 72 20 56 32 2E 30 20 53 68 61 72 65 77 61 72  er V2.0 Sharewar
004DFC64  65 0C 43 6C 69 65 6E 74 48 65 69 67 68 74 03 5A  e.ClientHeight Z
```

The first letter of the "*Shareware*" word starts at address `004DFC5C`. Click on `53` (in dump window) and press CTRL+E to edit data at this address and write `00`. Now save the executable and yes the "*Shareware*" is completely removed by changing only one byte.

And now apply the same approach for the Image 12.3.

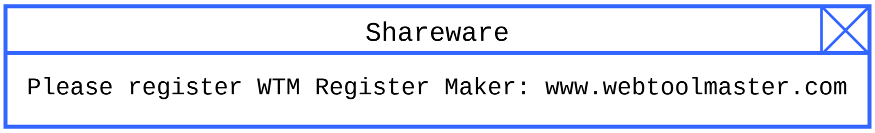
```
004E24E9   .  47 57 54 4D 20 52       ASCII "GWTM R"
004E24EF   .  65 67 69 73 74 65 72 20>ASCII "egister Maker is"
004E24FF   .  20 73 68 61 72 65 77 61>ASCII " shareware. Plea"
004E250F   .  73 65 20 72 65 67 69 73>ASCII "se register: www"
004E251F   .  2E 77 65 62 74 6F 6F 6C>ASCII ".webtoolmaster.c"
004E252F   .  6F 6D 00                ASCII "om",0
```

```
004E24DF  6C 07 43 61 70 74 69 6F 6E 06 47 57 54 4D 20 52  l Caption GWTM R
004E24EF  65 67 69 73 74 65 72 20 4D 61 6B 65 72 20 69 73  egister Maker is
004E24FF  20 73 68 61 72 65 77 61 72 65 2E 20 50 6C 65 61   shareware. Plea
004E250F  73 65 20 72 65 67 69 73 74 65 72 3A 20 77 77 77  se register: www
004E251F  2E 77 65 62 74 6F 6F 6C 6D 61 73 74 65 72 2E 63  .webtoolmaster.c
004E252F  6F 6D 00 00 0F 54 62 73 53 6B 69 6E 53 74 64 4C  om.. TbsSkinStdL
```

The first letter of the "*`is shareware. Please register: www.webtoolmaster.com`*" string starts at address `004E24FD`. Click on `69` (in dump window) and press CTRL+E to edit data at this address and write `00`. Now save the executable and yes the "*`is shareware. Please register: www.webtoolmaster.com`*" is completely removed by changing only one byte.

4.2. Nag screen load.dat

Change the extension to exe and this loader works fine but a nag screen always appear to register as in the following figure.



Shareware

Please register WTM Register Maker: www.webtoolmaster.com

Image 12.4 – load(2).dat nag screen

It's an easy task to remove this nag screen load the unpacked version in OllyDbg and do a search for all referenced text string or set a BP on MessageBoxA API. And we land here:

```
0042F19E  |.  6A 00              PUSH 0                ; /Style = MB_OK|MB_APPLMODAL
0042F1A0  |.  68 5CF24200        PUSH load.0042F25C    ; |Title = "Shareware"
0042F1A5  |.  68 68F24200        PUSH load.0042F268    ; |Text = "Please register WTM
Register Maker: www.webtoolmaster.com"
0042F1AA  |.  6A 00              PUSH 0                ; |hOwner = NULL
0042F1AC  |.  E8 8767FDFF        CALL <JMP.&user32.MessageBoxA>    ; \MessageBoxA
```

There is no conditional jump to bypass this nag screen, another approach would be to nop the call to the message box and the nag screen no longer appear at the start. (***Remember what I mentioned before you cannot use the unpacked loaders to protect any software, it won't work anymore so you have to restrict your modification to minimum. Taking into consideration the available bytes for inline patching later on***).

Having said that nopping the call to the message box won't serve our task. So, another approach is to change HWND hWnd, // handle of owner window value (Identifies the owner window of the message box to be created. If this parameter is NULL, the message box has no owner window.).

In this case hOwner parameter is NULL, the trick is to change its value to nonzero and the nag screen is defeated, because in this case the message box has an owner which does not exist). I used 0xFF:

```
0042F1AA   |.  6A FF               PUSH -1                  ; |hOwner = FFFFFFFF
```

4.3. Nag screen load2.dat

Apply same technique as before.

```
0042F1B6   |.  6A 00               PUSH 0                 ; /Style = MB_OK|MB_APPLMODAL
0042F1B8   |.  68 5CF24200         PUSH load2.0042F274    ; |Title = "Shareware"
0042F1BD   |.  68 68F24200         PUSH load2.0042F280    ; |Text = "Please register WTM 
Register Maker: www.webtoolmaster.com"
0042F1C2   |.  6A 00               PUSH 0                 ; |hOwner = NULL
0042F1C4   |.  E8 8767FDFF         CALL <JMP.&user32.MessageBoxA>     ; \MessageBoxA
```

After Modification:

```
0042F1C2   |.  6A FF               PUSH -1                  ; |hOwner = FFFFFFFF
```

5. INILINE Patching

5.1. RTool.exe Case

Load the original (Packed) RTool.exe in Olly and follow the same step as explained in the Unpacking section till here (but don't step into this jmp):

```
004F338A   - FFE0                JMP EAX                  ; RTool.004C5D94: OEP

004F338C     94                  XCHG EAX,ESP
004F338D     5D                  POP EBP
004F338E     4C                  DEC ESP
004F338F     00C0                ADD AL,AL
004F3391     334F 00             XOR ECX,DWORD PTR DS:[EDI]
004F3394     D033                SAL BYTE PTR DS:[EBX],1
004F3396     4F                  DEC EDI
004F3397     00A8 334F00AC       ADD BYTE PTR DS:[EAX+AC004F33],CH
004F339D     334F 00             XOR ECX,DWORD PTR DS:[EDI]
004F33A0     0000                ADD BYTE PTR DS:[EAX],AL
004F33A2     0000                ADD BYTE PTR DS:[EAX],AL
004F33A4     0000                ADD BYTE PTR DS:[EAX],AL
004F33A6     0000                ADD BYTE PTR DS:[EAX],AL
004F33A8     0000                ADD BYTE PTR DS:[EAX],AL
004F33AA     0000                ADD BYTE PTR DS:[EAX],AL
004F33AC     0000                ADD BYTE PTR DS:[EAX],AL
004F33AE     0000                ADD BYTE PTR DS:[EAX],AL
004F33B0     0000                ADD BYTE PTR DS:[EAX],AL
004F33B2     0000                ADD BYTE PTR DS:[EAX],AL
004F33B4     0000                ADD BYTE PTR DS:[EAX],AL
004F33B6     0000                ADD BYTE PTR DS:[EAX],AL
004F33B8     0000                ADD BYTE PTR DS:[EAX],AL
004F33BA     0000                ADD BYTE PTR DS:[EAX],AL
004F33BC     0000                ADD BYTE PTR DS:[EAX],AL
004F33BE     0000                ADD BYTE PTR DS:[EAX],AL
```

We need to find a padded area of zero's to inject our patched byte. I know there are a plenty down this address but they are not accessible to be used in our inline patching (try and you'll got this error "Unable to locate data in executable file"), more than that, even after this 004F338A the area that follow this address is very critical to fit our changes (I tried before and it doesn't work). This phenomenon refer to the fact that the Virtual Size is larger than the Raw Size (as in figure 5) so you can't save your changes because it doesn't exist in the executable it's only Virtual, you may overcome this problem by adding a new section which is a tedious task to do for three programs, or you can make the Virtual Size equal the Raw Size by adding the missing bytes using any hex editor.
In our case, we won't adopt any of these methods, we'll stick to the normal method with some careful and optimization. And it's done.

```
No  | Name      | VSize     | VOffset   | RSize     | ROffset   | Charact.  |
01  | CODE      | 000F0000  | 00001000  | 0004F800  | 00000400  | E0000020  |
02  | .rsrc     | 00003000  | 000F1000  | 00002400  | 0004FC00  | E0000020  |
```

Figure 5 – Virtual and Raw Size

5.1.1 Owned it

So we'll locate address 004F33AC as a starting point to inject the modified bytes; it's better to save the registers and flags contents before you inject your own code and after that restore them so that not to interfere with the flow of the original executable registers and flags contents.

⇒ Redirection to our cave

Another constraint on our patching mode is that we need to redirect the jump from OEP to our cave using a short jump so that not to overwrite many instructions. And here it is finalized.

```
004F338A   /EB 20               JMP SHORT RTool.004F33AC ; JMP to our cave
004F338C   |94                  XCHG EAX,ESP
004F338D   |5D                  POP EBP
004F338E   |4C                  DEC ESP
004F338F   |00C0                ADD AL,AL
004F3391   |334F 00             XOR ECX,DWORD PTR DS:[EDI]
004F3394   |D033                SAL BYTE PTR DS:[EBX],1
004F3396   |4F                  DEC EDI
004F3397   |00A8 334F00AC       ADD BYTE PTR DS:[EAX+AC004F33],CH
004F339D   |334F 00             XOR ECX,DWORD PTR DS:[EDI]
004F33A0   |0000                ADD BYTE PTR DS:[EAX],AL
004F33A2   |0000                ADD BYTE PTR DS:[EAX],AL
004F33A4   |0000                ADD BYTE PTR DS:[EAX],AL
004F33A6   |0000                ADD BYTE PTR DS:[EAX],AL
004F33A8   |0000                ADD BYTE PTR DS:[EAX],AL
004F33AA   |0000                ADD BYTE PTR DS:[EAX],AL
004F33AC   \60                  PUSHAD       ; Save the contents of the registers
004F33AD    C605 5CFC4D00 00    MOV BYTE PTR DS:[4DFC5C],0 ; our modified byte
004F33B4    C605 FC244E00 00    MOV BYTE PTR DS:[4E24FC],0 ; our modified byte
004F33BB    61                  POPAD        ; Restore registers contents
004F33BC    FFE0                JMP EAX ; JMP to OEP EAX = 004C5D94
```

If you have enough space (another program scenario) had better to do it like this:

```
xxxxxxxx   \00                  PUSHAD       ; Save the contents of the registers by
                                               pushing them on the stack (32 bit)

xxxxxxxx   \00                  PUSHFD       ; Save the contents of the EFLAGS by
                                               pushing them onto the stack (32 bit)

xxxxxxxx    0000 00000000 00    MOV x PTR DS:[xxxxxxxx],0 ; our modified byte
xxxxxxxx    0000 00000000 00    MOV x PTR DS:[xxxxxxxx],0 ; our modified byte

xxxxxxxx    00                  POPFD        ; Restore EFLAGS contents by popping the
                                               top of the stack into 32-bit EFLAGS
                                               register

xxxxxxxx    00                  POPAD        ; Restore registers contents by popping
                                               the top of the stack into 32-bit
                                               registers

xxxxxxxx    0000                JMP xxxxxxxx ; JMP to OEP xxxxxxxx
```

5.2. load.dat Case

```
0043F9D2   . /EB 13             JMP SHORT load.0043F9E7
0043F9D4     |F8F34200          DD load.0042F3F8    ;  ASCII "U‹ìƒÄô¸ óB"
0043F9D8     |08FA4300          DD load.0043FA08
0043F9DC     |10FA4300          DD load.0043FA10
0043F9E0     |F0F94300          DD load.0043F9F0
0043F9E4     |F4                DB F4
0043F9E5   . |F9                STC
```

```
0043F9E6    . |43               INC EBX
0043F9E7    > \C605 ABF14200 FF MOV BYTE PTR DS:[42F1AB],0FF
0043F9EE    .  FFE0             JMP EAX ; JMP to OEP EAX = 0042F3F8
```

Note: 0FF “0” must be written before values that begins with a letter.

5.3. load2.dat Case

```
0043F9D2    . /EB 13            JMP SHORT load2.0043F9E7
0043F9D4    . |10F4             ADC AH,DH
0043F9D6    . |42               INC EDX
0043F9D7    . |0008             ADD BYTE PTR DS:[EAX],CL
0043F9D9    . |FA               CLI
0043F9DA    . |43               INC EBX
0043F9DB    . |0010             ADD BYTE PTR DS:[EAX],DL
0043F9DD    . |FA               CLI
0043F9DE    . |43               INC EBX
0043F9DF    . |00F0             ADD AL,DH
0043F9E1    . |F9               STC
0043F9E2    . |43               INC EBX
0043F9E3    . |00F4             ADD AH,DH
0043F9E5    . |F9               STC
0043F9E6    . |43               INC EBX
0043F9E7    > \C605 C3F14200 FF MOV BYTE PTR DS:[42F1C3],0FF
0043F9EE    .  FFE0             JMP EAX ; JMP to OEP EAX = 0042F410
```

6. Final Remarks

The most important thing to look at in this paper is know how to find your path around the axe of interest so that to understand the interrelations between each object and to defeat the obstacles till you got satisfied.

Finding multi-solution with a degree of optimization for the same problem gives you a better solid understanding in code infrastructure analysis.

Always try to give yourself a space and time when you start doing something new for the first time like I’ve done in this short journey.

7. References

[1] “RCE Emphasizing on Breaking Software Protection”, tHE mUTABLE , http://tutorials.accessroot.com
[2] “Working with IMPORT TABLES part 3” , Ricardo Narvaja, English version, Translated by Innocent

13. ARTeam eZine #3 Call for Papers

ARTeam members are asking for your article submissions on subjects related to Reverse-Engineering.

We wanted to provide the community with somewhere to distribute interesting, sometimes random, reversing information. Not everyone likes to write tutorials, and not everyone feels that the information they have is enough to constitute a publication of any sort. I'm sure all of us have hit upon something interesting while coding/reversing and have wanted to share it but didn't know exactly how. Or if you have cracked some interesting protection but didn't feel like writing a whole step by step tutorial, you can share the basic steps and theory here. If you have an idea for an article, or just something fascinating you want to share, let us know.

Examples of articles are a new way to detect a debugger, or a new way to defeat debugger detection, or how to defeat an interesting crackme.

The eZine is more about sharing knowledge, as opposed to teaching. So the articles can be more generic in nature. You don't have to walk a user through step by step. Instead you can share information from simple theory all the way to "sources included"

What we are looking for in an article submission:

1. Clear thought out article. We are asking you to take pride in what you submit.
2. It doesn't have to be very long. A few paragraphs is fine, but it needs to make sense.
3. Any format is fine, but to save our time possibly send them in WinWord Office or text format.
4. If you include pictures please center them in the article. If possible please add a number and label below each image.
5. If you use references please add them as footnotes where used.
6. If you include code snippets inside a document other than .txt please use a monospace font to allow for better formatting and possibly use a syntax colorizer
7. Anonymous articles are fine. But you must have written it. No plagiarism!
8. Any other questions you may have feel free to ask

We are accepting articles from anyone wanting to contribute. That means you.

We want to make the eZine more of a community project than a team release. If your article is not used, it's not because we don't like it. It may just need some work. We will work with you to help develop your article if it needs it.

Questions or Comments please visit http://forum.accessroot.com