

The Legend of Random

Task 01

Reverse Engineering

What is Reverse Engineering?

Reverse engineering is the process of taking a compiled binary and attempting to recreate the original way the program works.

What is reverse engineering used for?

Reverse engineering can be applied to many areas of computer science, but here are a couple of generic categories;

1. Making it possible to interface to legacy code (where you do not have the original code source).
2. Breaking copy protection (ie. Impress your friends and save some \$\$).
3. Studying virii and malware.
4. Evaluating software quality and robustness.
5. Adding functionality to existing software.

The first category is reverse engineering code to interface with existing binaries when the source code is not available.

The second category is the biggest. It disables time trials, defeat registration, and basically everything else to get commercial software for free.

Reverse Engineering needs for studying virus and malware code because not a lot of virus coders out there don't send instructions on how they wrote the code, what it is supposed to accomplish, and how it will accomplish this (unless they are really dumb).

In the fourth category, when creating large systems like Operating Systems, reverse engineering is used to make sure that the system does not contain any major vulnerabilities, security flaws, and frankly, to make it as hard as possible to allow crackers to crack the software.

The final category is adding functionality to existing software. As examples, graphics used in web design software and when want to change it.

What knowledge is required?

Basic understanding of how program flow works

Familiarization with Assembly Language

Reverse engineering Tools

Experimentation (playing with different packers/protectors/encryption schemes, learning about programs originally written in different programming languages (even Delphi), deciphering anti-reverse engineering tricks)

What kinds of tools are used?

1. Disassemblers

Disassemblers attempt to take the machine language codes in the binary and display them in a friendlier format. They also extrapolate data such as function calls, passed variables and text strings.

2. Debuggers

Debuggers first analyze the binary, much like a disassembler. Debuggers then allow the reverser to step through the code, running one line at a time and investigating the results. This is invaluable to discover how a program works.

3. Hex editors

Hex editors allow you to view the actual bytes in a binary, and change them. They also provide searching for specific bytes, saving sections of a binary to disk, and much more.

4. PE and resource viewers/editors

Every binary designed to run on a windows machine (and Linux for that matter) has a very specific section of data at the beginning of it that tells the operating system how to set up and initialize the program. It tells the OS how much memory it will require, what support DLLs the program needs to borrow code from, information about dialog boxes and such. This is called the Portable Executable, and all programs designed to run on windows need to have one.

5. System Monitoring tools

When reversing programs, it is sometimes important (and when studying viruses and malware, of the utmost importance) to see what changes an application makes to the system; are there registry keys created or queried? are there .ini files created? are separate processes created, perhaps to thwart reverse engineering of the application? Examples of system monitoring tools are procmon, regshot, and process hacker.

6. Miscellaneous tools and information

There are tools such as scripts, unpackers, packer identifiers etc and also Windows API helps to know exactly what called functions are doing.

7. Beer.

Let's get Start

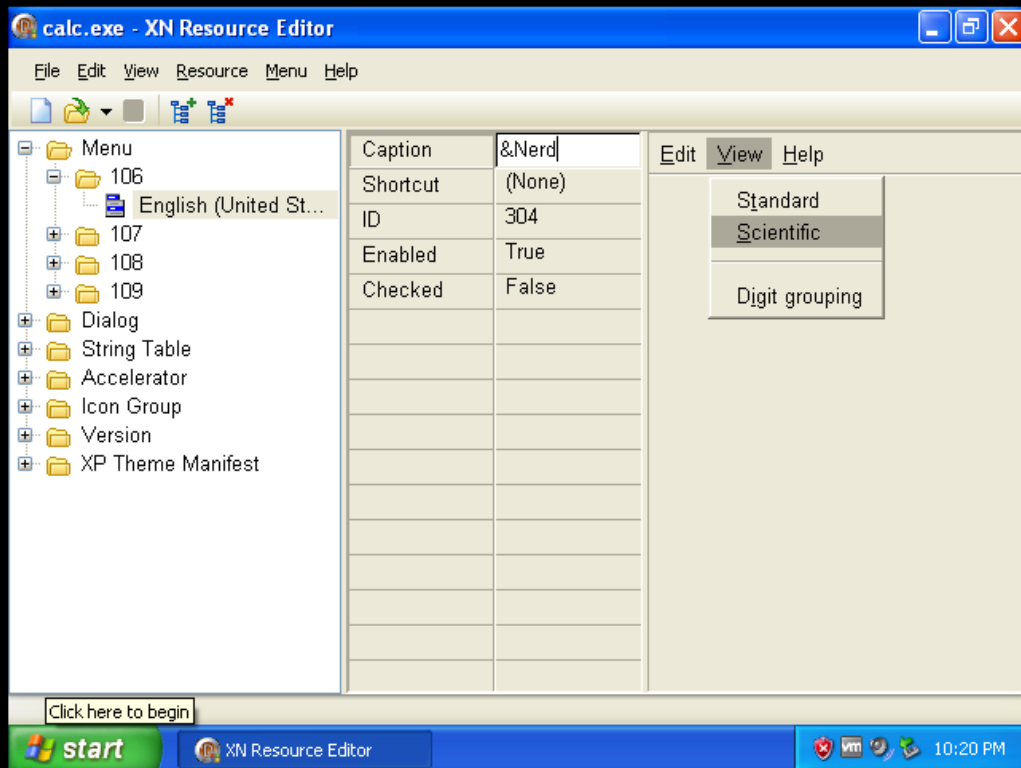
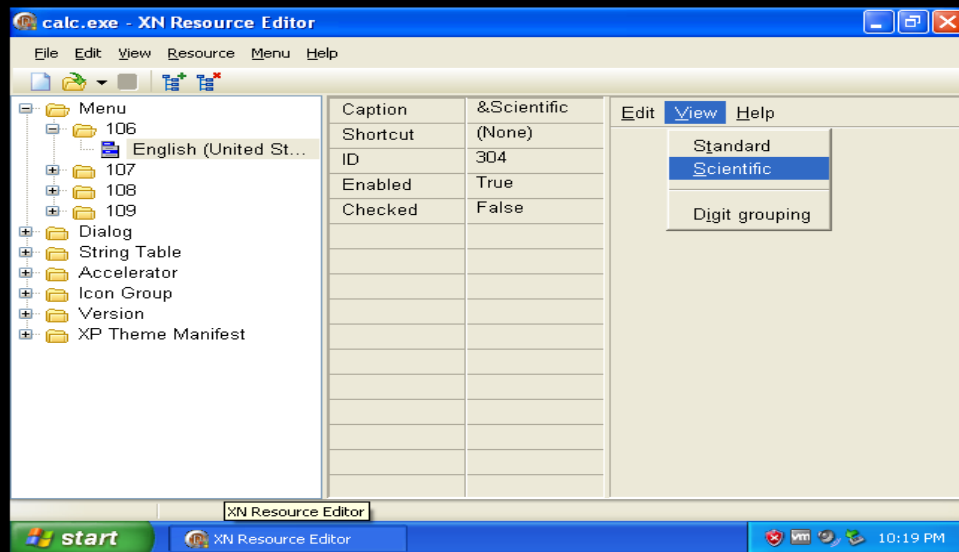
Here I have used a resource viewer/editor called XN Resource Editor. It is freeware. Basically, this program allows to see the resource section in an exe file, as well as modify these resources. In here this file allows to change the menus, icons, graphics, dialogs, you name it, in programs. Let's try one ourselves.

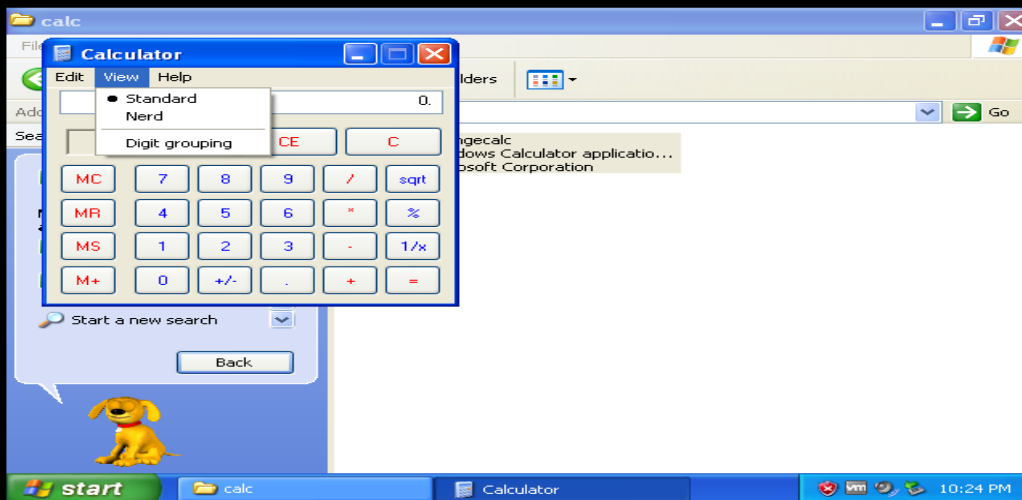
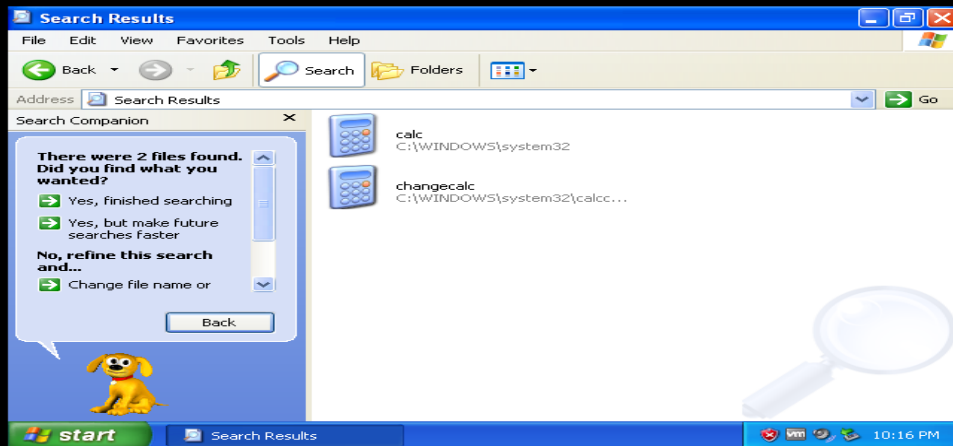
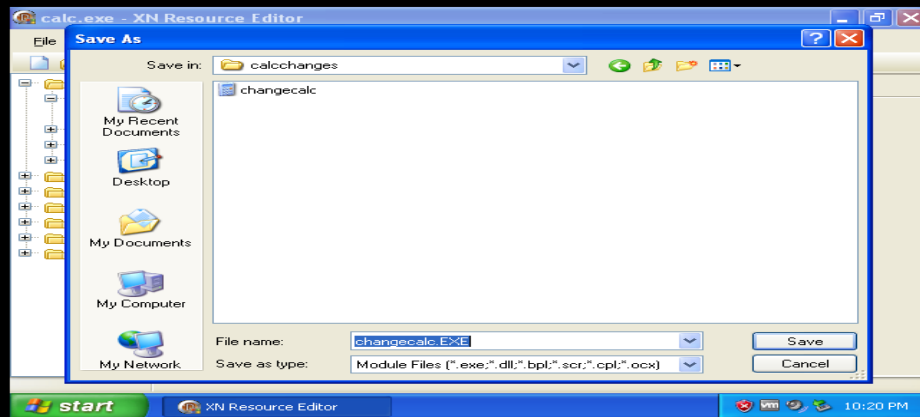
First, run XN. Click on the load icon on top, and click over to Windows\System32\ and load calc.exe. There are bunch of folders available.

In Windows\System32\ there are folders for Bitmaps, Menu, Dialog, String Table, IconGroup etc. To do changes, click on the plus next to Menu. Then there's a folder with a number as a name. This is the ID that windows will use to access this resource in the program. In side this folder there's an icon for "English (United States)". When click on it and see a diagram of menu.



Now, click on the menu option "Scientific". Change caption field to "&Scientific". "&" shows 'Hot-Key' in this case it's "S". If instead wanted the 'e' to be the hot-key, it would look like this "Sci&entific". In the Caption field, replace the &Scientific with "&Nerd". This will now change the menu option to "Nerd" and use the hot-key 'N'. Now, go up to File (in XN Resource) and choose "Save As..." Save new version of calc to "calcchange" and saved in another new folder names as "change calc" and then run it.





Task 02

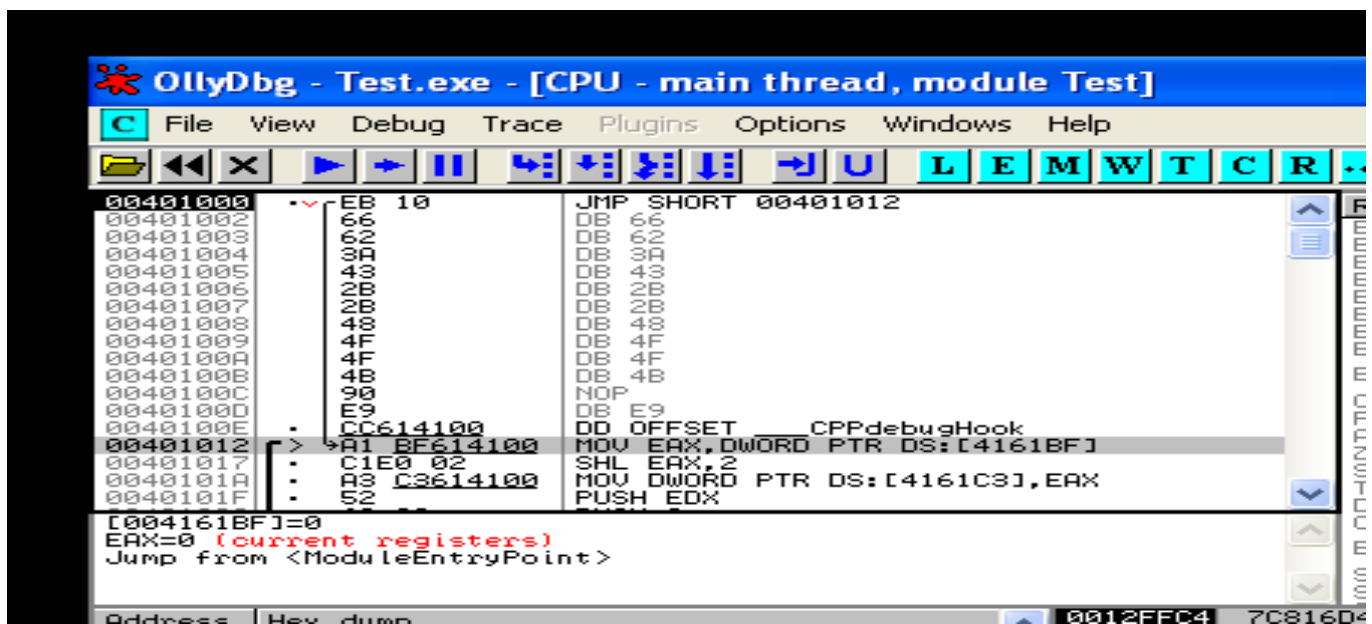
Olly Debugger

What is Olly Debugger?

” Olly is also a “dynamic” debugger, it allows the user to change quite a few things while running the program. This is very important when experimenting with a binary, when trying to figure out how it works. Olly has many, many great features, and this is the main debugger used for reverse engineering.

An Overview

Here is the Olly’s main display, along with some labels.



There are 4 separated main fields;

1. Disassembly
2. Registers
3. Stack
4. Dump

1. Disassembly

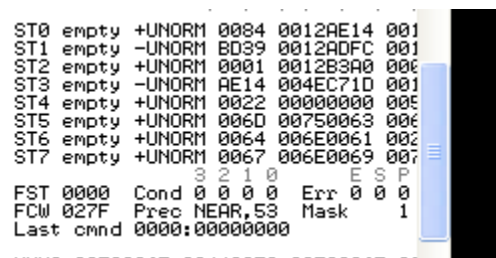
Window contains the main disassembly of the code for the binary. This is where Olly displays information in the binary, including the opcodes and translated assembly language.

2. Registers



Registers are temporary holders for values, much like a variable in any high-level programming language.

3. The Stack



The stack is a section of memory reserved for the binary as a 'temporary' list of data. This data includes pointers to addresses in memory, strings, markers, and most importantly, return addresses for the code to return to when calling a function. In the stack, there's a "First In, Last Out" data structure. that the CPU has placed on the stack for when the current function is done, so that it will know where to return to. In Olly, you can right click on the stack and choose 'modify' to change the contents.

4 The Dump

Address	Hex dump	ASCII
00416000	00 00 3C A4 40 00 00 20 B8 A4 40 00 00 1C A0 4F	█ 8000 2000 1C00 4F00
00416010	41 00 00 20 8C A9 40 00 00 20 B8 A0 40 00 00 00	A 1000 2000 8C00 A900 4000 0000 2000 B800 A000 4000 0000 0000
00416020	04 B0 40 00 00 00 38 B0 40 00 00 05 28 C7 40 00	0400 B000 4000 0000 0000 3800 B000 4000 0000 0000 0500 2800 C700 4000
00416030	00 04 B8 C4 40 00 00 0A F4 E5 40 00 00 0A 20 EF	0000 0400 B800 C400 4000 0000 0A00 F400 E500 4000 0000 0A00 2000 EF00
00416040	20 00 00 00 28 C7 40 00 00 01 44 30 40 00 00 00	2000 0000 0000 0000 2800 C700 4000 0000 0000 0100 4400 3000 4000 0000 0000
00416050	20 F0 40 00 00 03 10 FF 40 00 00 02 3C 04 41 00	2000 F000 4000 0000 0000 0300 1000 FF00 4000 0000 0000 0200 3C00 0400 4100
00416060	00 03 74 06 41 00 00 06 A4 0B 41 00 00 01 74 00	0000 0300 7400 0600 4100 0000 0000 0600 A400 0B00 4100 0000 0000 0100 7400
00416070	41 00 00 01 18 0F 41 00 00 00 9C 10 41 00 00 00	A 0000 0000 0100 1800 0F00 4100 0000 0000 0000 9C00 1000 4100 0000 0000
00416080	B4 10 41 00 00 20 39 36 41 00 00 1F EC 4F 41 00	1000 B400 1000 4100 0000 0000 2000 3900 3600 4100 0000 0000 1F00 EC00 4F00 4100
00416090	00 1E 40 01 40 00 00 1E B8 12 40 00 00 1E CC 26	0000 1E00 4000 0100 4000 0000 0000 1E00 B800 1200 4000 0000 0000 1E00 CC00 2600
004160A0	40 00 00 1E A4 29 40 00 00 1E 04 27 40 00 00 1E	4000 0000 0000 1E00 A400 2900 4000 0000 0000 1E00 0400 2700 4000 0000 0000 1E00
004160B0	E0 5E 40 00 00 1E 24 26 40 00 00 20 E8 A4 40 00	E000 5E00 4000 0000 0000 1E00 2400 2600 4000 0000 0000 2000 E800 A400 4000
004160C0	00 1F C0 00 00 00 1F 67 63 74 20 31 39 32 61	0000 1F00 C000 0000 0000 0000 1F00 6700 6300 7400 2000 3100 3900 3200 6100
004160D0	40 00 00 20 F4 00 40 00 00 01 2F B1 40 00 00 00	4000 0000 0000 2000 F400 0000 4000 0000 0000 0100 2F00 B100 4000 0000 0000
004160E0	50 B0 40 00 00 00 EC C7 40 00 00 03 38 FF 40 00	5000 B000 4000 0000 0000 0000 EC00 C700 4000 0000 0000 0300 3800 FF00 4000
004160F0	00 02 C4 04 41 00 00 03 F4 04 41 00 00 02 2C 0F	0000 0200 C400 0400 4100 0000 0000 0300 F400 0400 4100 0000 0000 0200 2C00 0F00
00416100	41 00 00 1E D8 A0 40 00 00 1E 88 12 40 00 00 1E	A 0000 0000 1E00 D800 A000 4000 0000 0000 1E00 8800 1200 4000 0000 0000 1E00
00416110	9C 26 40 00 00 1E 84 29 40 00 00 1E 04 26 40 00	9C00 2600 4000 0000 0000 1E00 8400 2900 4000 0000 0000 1E00 0400 2600 4000
00416120	00 1E 00 5D 40 00 00 1E 44 26 40 00 00 0C 58 21	0000 1E00 0000 5D00 4000 0000 0000 1E00 4400 2600 4000 0000 0000 0C00 5800 2100
00416130	F1 FF FF 53 42 6F 72 60 61 6E 64 20 43 2B 20 20	F100 FF00 FF00 5300 4200 6F00 7200 6000 6100 6E00 6400 2000 4300 2B00 2000 2000
00416140	20 00 43 6F 70 73 73 65 20 43 6F 72 70 6F 72 61	2000 0000 4300 6F00 7000 7300 7300 6500 2000 4300 6F00 7200 7000 6F00 7200 6100
00416150	20 49 6E 70 72 63 73 65 20 43 6F 72 70 6F 72 61	2000 4900 6E00 7000 7200 6300 7300 6500 2000 4300 6F00 7200 7000 6F00 7200 6100
00416160	74 69 6F 6E 00 00 00 00 00 00 00 41 00 00 00	7400 6900 6F00 6E00 0000 0000 0000 0000 0000 0000 0000 4100 0000 0000
00416170	B0 60 41 00 2C 61 41 00 01 00 00 00 00 00 00 00	B000 6000 4100 0000 2C00 6100 4100 0000 0100 0000 0000 0000 0000 0000 0000
00416180	88 1E 40 00 44 E2 40 00 70 E2 40 00 00 00 00 00	8800 1E00 4000 0000 4400 E200 4000 0000 7000 E200 4000 0000 0000 0000 0000
00416190	8C 7A 41 00 10 8A 41 00 1A 8A 41 00 7C 00 41 00	8C00 7A00 4100 0000 1000 8A00 4100 0000 1A00 8A00 4100 0000 7C00 0000 4100
004161A0	04 01 41 00 64 02 41 00 E8 03 41 00 0C 75 41 00	0400 0100 4100 0000 6400 0200 4100 0000 E800 0300 4100 0000 0C00 7500 4100

Entry point of main module

The dump window is a built-in hex viewer that lets you see the raw binary data, only in memory as opposed to on disk. Usually it shows two views of the same data; hexadecimal and ASCII.

The Toolbar



All of the buttons are accessible from the “Debug” drop down menu.

Re-load

To restart the app and pause it at the entry point. All patches (see later) will be removed, some breakpoints will be disabled, and the app will not have run any code yet, well, most of the time anyway.

Run and Pause do just that.

Step In

Run one line of code and then pause again, calling into a function call if there was one.

Step Over

Does the same thing, but jumps over a call to another function.

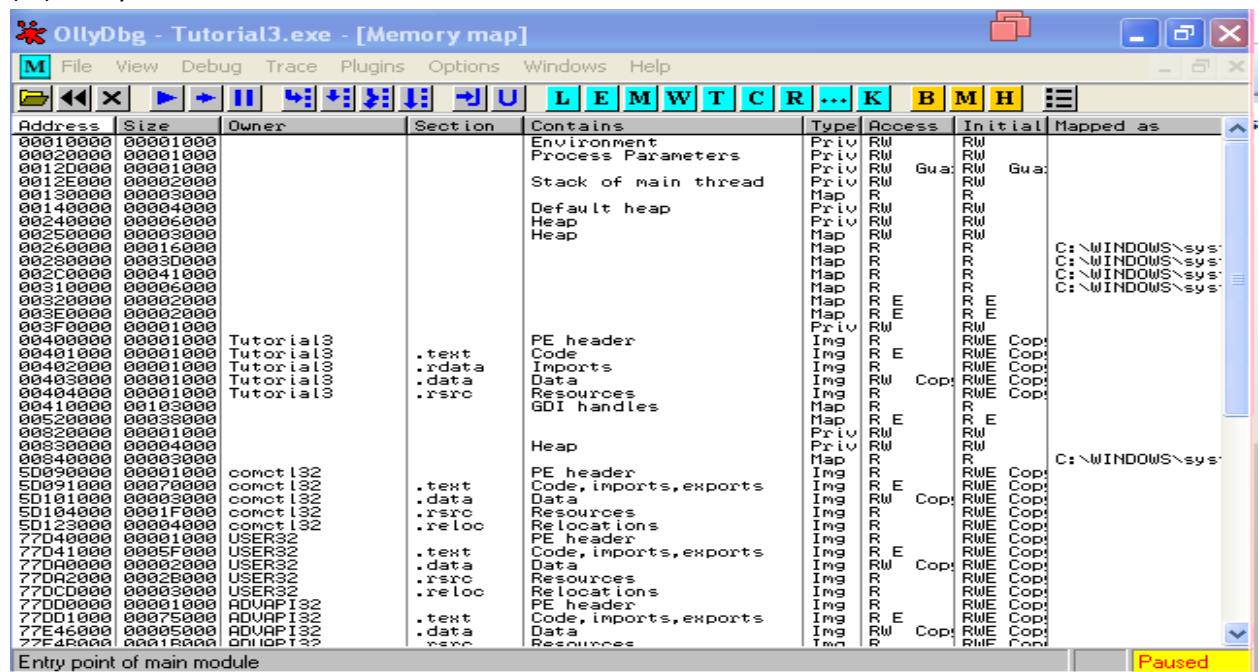
Animate

Just like Step In and Over except it does it especially if it's a polymorphic binary and can watch the code change.



Some of the more common windows right now:

1. (M)emory



The memory window displays all of the memory blocks that the program has allocated. It includes the main sections of the running app. This window also shows the type of block, the access rights, the size and the memory address where the section is loaded.

2. (P)atches

Patches				
Address	Size	State	Old	New
00401239	7.	Active	PUSH EBX	NOP
0040124F	2.	Active	PUSH 0	PUSH 10
00401267	6.	Active	PUSH DWORD PTR DS:[426028]	JMP SHORT showstri.004

This window displays any “patches” you have made, any changes to the original code. Notice that the state is set as “Active”. If you re-load the app, patches will become disabled. In order to re-enable them (or disable them) simply click on the desired patch and hit the spacebar. This toggles the patch on/off.

3. (B)reakpoints

Breakpoints				
Address	Module	Active	Disassembly	Comment
0040124C	showstri	Always	ADD ESP,4	
00401273	showstri	Always	PUSH DWORD PTR DS:[426020]	
0040129C	showstri	Always	PUSH EBP	

Shows where all of the current breakpoints are set.

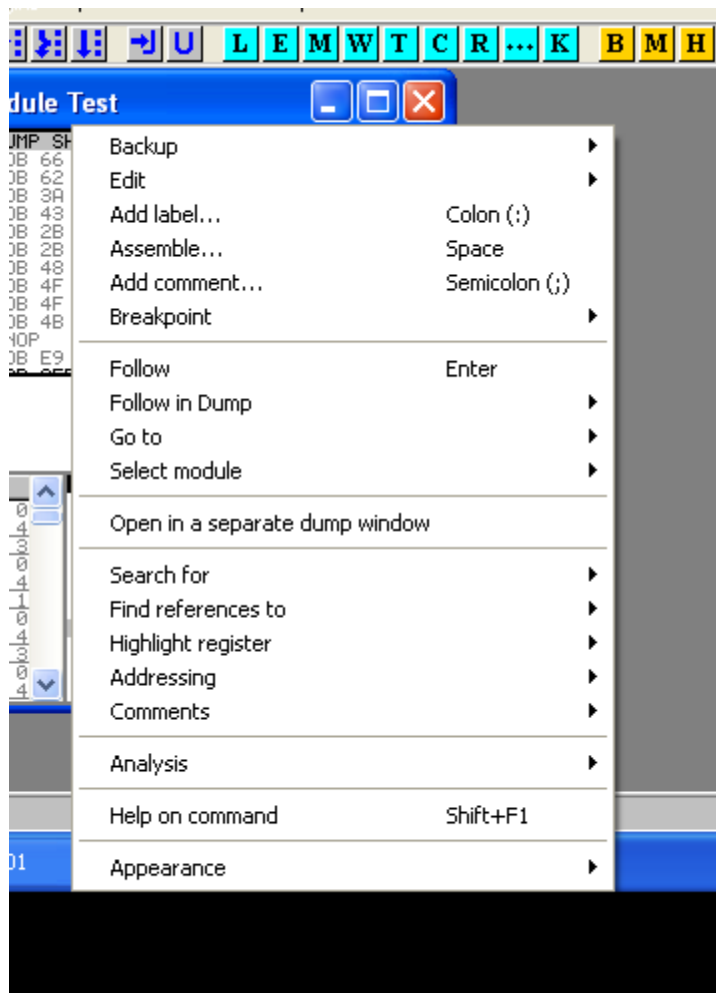
4. (K)all Stack

Breakpoints				
Address	Module	Active	Disassembly	Comment
0040124C	showstri	Always	ADD ESP,4	
00401273	showstri	Always	PUSH DWORD PTR DS:[426020]	
0040129C	showstri	Always	PUSH EBP	

This window is different from the “Stack”. It shows a lot more info about calls being made in the code, the values sent to those functions, and more.

The Context Menu

Right-clicking anywhere in the disassembly section brings it up:



Binary

Allows editing of the binary data on a byte-by-byte level. This is where you may change a “Unregistered” string buried in a binary to “Registered”.

Breakpoint

Allows you to set a breakpoint.

Search For

A rather large sub-menu, and it’s where you search the binary for data such as strings, function calls etc.

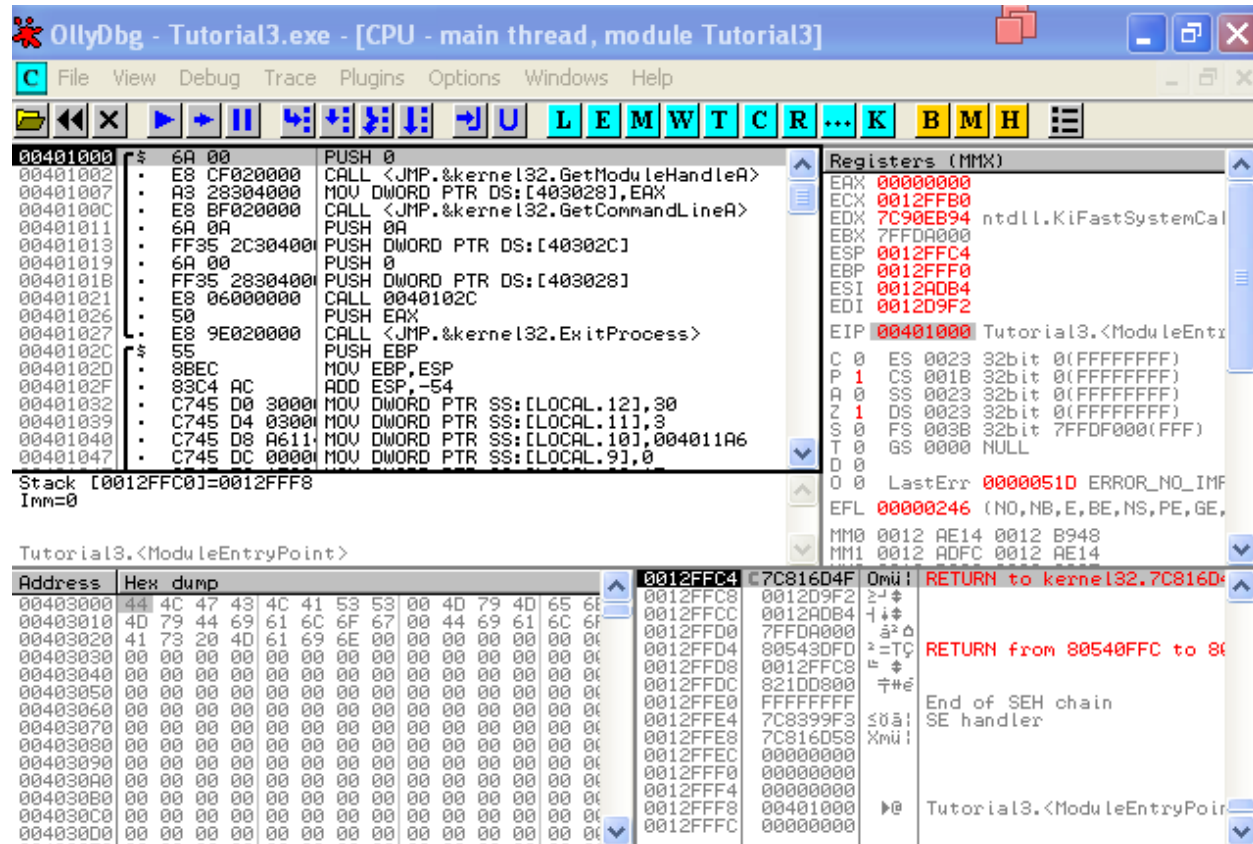
Analysis

Forces Olly to re-analyze the section of code you are currently viewing. Sometimes Olly gets confused as to whether you are viewing code or data.

Task 03

Loading the app

As the first step load "FirstProgram.exe", file to the Olly and it completed analysis and stop at the programs Entry Point (EP):



Here in the first column the EP is at address 401000.This is a pretty standard starting point for an executable.

OllyDbg - Tutorial3.exe - [Memory map]								
File View Debug Trace Plugins Options Windows Help								
Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00010000	00001000			Environment	Priv	RW	RW	
00020000	00001000			Process Parameters	Priv	RW	RW	
00120000	00001000				Priv	RW	Gua	Gua
0012E000	00002000			Stack of main thread	Priv	RW	RW	
00130000	00003000				Map	R	R	
00140000	00004000			Default heap	Priv	RW	RW	
00240000	00006000			Heap	Priv	RW	RW	
00250000	00003000			Heap	Map	RW	RW	
00260000	00016000				Map	R	R	C:\WINDOWS\sys:
00280000	00030000				Map	R	R	C:\WINDOWS\sys:
002C0000	00041000				Map	R	R	C:\WINDOWS\sys:
00310000	00006000				Map	R	R	C:\WINDOWS\sys:
00320000	00002000				Map	R E	R E	
003E0000	00002000				Map	R E	R E	
003F0000	00001000				Priv	RW	RW	
00400000	00001000	Tutorial3		PE header	Img	R	RWE Cop	
00401000	00001000	Tutorial3	.text	Code	Img	R E	RWE Cop	
00402000	00001000	Tutorial3	.rdata	Imports	Img	R	RWE Cop	
00403000	00001000	Tutorial3	.data	Data	Img	RW	RWE Cop	
00404000	00001000	Tutorial3	.rsrc	Resources	Img	R	RWE Cop	
00410000	00103000			GDI handles	Map	R	R	
00520000	00038000				Map	R E	R E	
00820000	00001000			Heap	Priv	RW	RW	
00830000	00004000				Priv	RW	RW	
00840000	00003000				Map	R	R	C:\WINDOWS\sys:
50090000	00001000	comctl32		PE header	Img	R	RWE Cop	
50091000	00070000	comctl32	.text	Code, imports, exports	Img	R E	RWE Cop	
50101000	00003000	comctl32	.data	Data	Img	RW	RWE Cop	
50104000	0001F000	comctl32	.rsrc	Resources	Img	R	RWE Cop	
50123000	00004000	comctl32	.reloc	Relocations	Img	R	RWE Cop	
77040000	00001000	USER32		PE header	Img	R	RWE Cop	
77041000	0005F000	USER32	.text	Code, imports, exports	Img	R E	RWE Cop	
770A0000	00002000	USER32	.data	Data	Img	RW	RWE Cop	
770A2000	0002B000	USER32	.rsrc	Resources	Img	R	RWE Cop	
770DC000	00003000	USER32	.reloc	Relocations	Img	R	RWE Cop	
77DD0000	00001000	ADVAPI32		PE header	Img	R	RWE Cop	
77DD1000	00075000	ADVAPI32	.text	Code, imports, exports	Img	R E	RWE Cop	
77E46000	00005000	ADVAPI32	.data	Data	Img	RW	RWE Cop	
77F4B000	0001B000	ADVAPI32	.rsrc	Resources	Img	R	RWE Cop	
Entry point of main module								Paused

In the address column, at location 401000, the row contains,

“FirstPro” (First Program) → 1000

“.text, → “SFX, code”

This is the 1000h bytes long “code” for the program and it starts at address 401000 in memory.

“.rdata” → contains data and imports at address 402000,

“.data” → contains nothing at address 403000 (actually contains global variables and random data but Olly doesn’t what kind of data it stored)

“.rsrc” → contains resources (eg: dialog boxes, images, text etc.)

To see exactly which functions program calls,

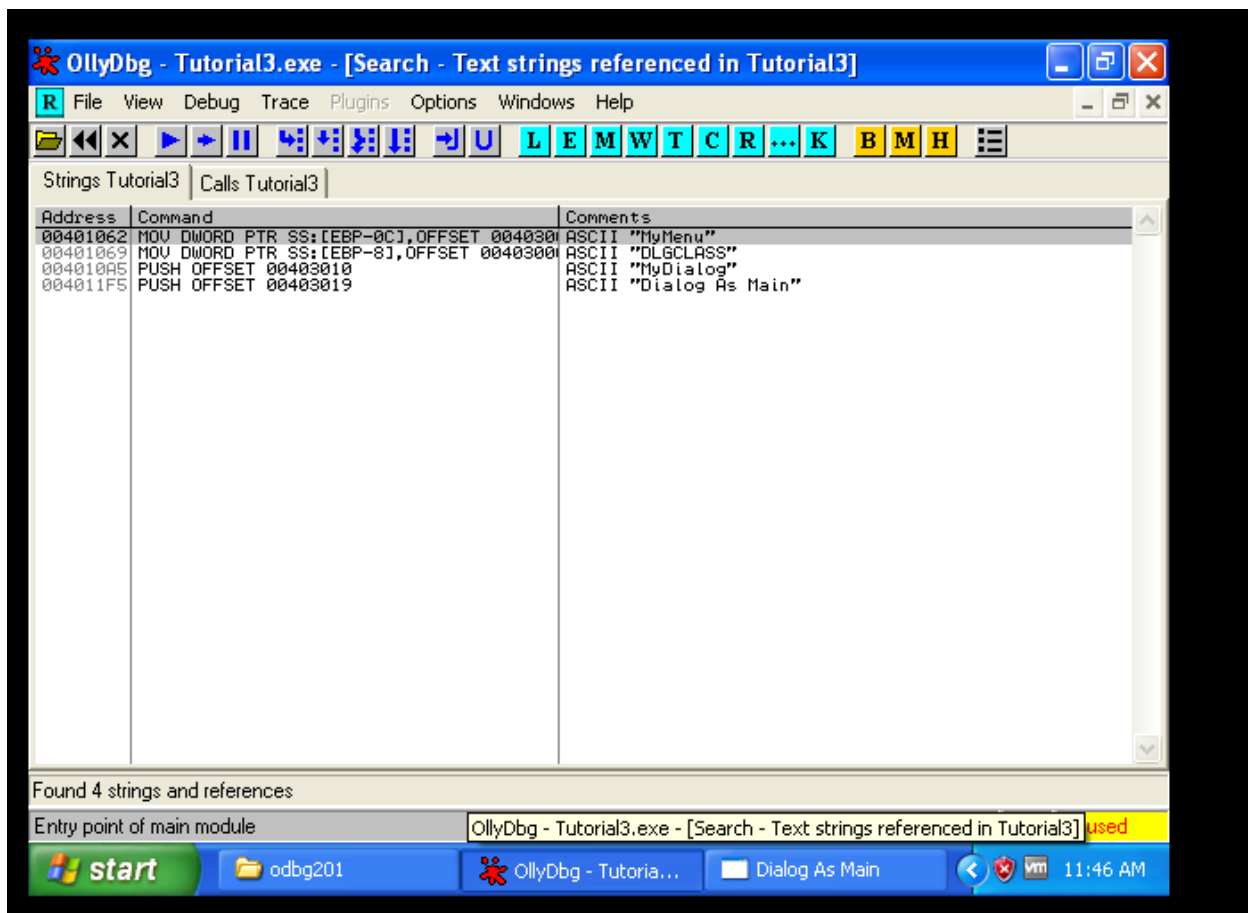
Right-click in Olly’s disassembly window and select “Search For” -> “All Intermodular Calls.

This shows something like the following:

R File View Debug Trace Plugins Options Windows Help				
Calls Tutorial3				
Address	Command	Dest	Dest name	Comments
00401002	CALL <JMP.&kernel32.GetModuleHandleA>	7C80B529	kernel32.GetModuleHandleA	ModuleNam
0040100C	CALL <JMP.&kernel32.GetCommandLineA>	7C812C8D	kernel32.GetCommandLineA	
00401027	CALL <JMP.&kernel32.ExitProcess>	7C81CAA2	kernel32.ExitProcess	
00401077	CALL <JMP.&user32.LoadIconA>	77D521AE	USER32.LoadIconA	hInst = N
00401089	CALL <JMP.&user32.LoadCursorA>	77D4E8FA	USER32.LoadCursorA	hInst = N
00401095	CALL <JMP.&user32.RegisterClassExA>	77D54315	USER32.RegisterClassExA	
0040109A	CALL <JMP.&comctl32.InitCommonControls>	5D0B15DD	comctl32.InitCommonControls	
004010B0	CALL <JMP.&user32.CreateDialogParamA>	77D65EA0	USER32.CreateDialogParamA	TemplateM
004010C9	CALL <JMP.&user32.SendDlgItemMessageA>	77D6152F	USER32.SendDlgItemMessageA	ItemID =
004010E2	CALL <JMP.&user32.SendDlgItemMessageA>	77D6152F	USER32.SendDlgItemMessageA	ItemID =
004010F1	CALL <JMP.&comctl32.ImageList_Create>	5D098B5B	comctl32.ImageList_Create	Arg1 = 10
0040110C	CALL <JMP.&user32.LoadImageA>	77D6F40C	USER32.LoadImageA	Name = 70
00401118	CALL <JMP.&comctl32.ImageList_Add>	5D0D29E3	comctl32.ImageList_Add	Arg3 = 0
0040111F	CALL <JMP.&gdi32.DeleteObject>	77F16A9B	gdi32.DeleteObject	
00401136	CALL <JMP.&user32.SendDlgItemMessageA>	77D6152F	USER32.SendDlgItemMessageA	ItemID =
00401143	CALL <JMP.&user32.GetDlgItem>	77D552A4	USER32.GetDlgItem	ItemID =
00401149	CALL <JMP.&user32.SetFocus>	77D4E5DC	USER32.SetFocus	
00401153	CALL <JMP.&user32.ShowWindow>	77D4D4DE	USER32.ShowWindow	Show = SW
00401158	CALL <JMP.&user32.UpdateWindow>	77D4C064	USER32.UpdateWindow	
0040116A	CALL <JMP.&user32.GetMessageA>	77D6EA45	USER32.GetMessageA	hWnd = NU
0040117A	CALL <JMP.&user32.IsDialogMessageA>	77D65C98	USER32.IsDialogMessage	
00401187	CALL <JMP.&user32.TranslateMessage>	77D48BCE	USER32.TranslateMessage	
00401190	CALL <JMP.&user32.DispatchMessageA>	77D48CB0	USER32.DispatchMessageA	
0040119A	CALL <JMP.&comctl32.ImageList_Destroy>	5D098D2E	comctl32.ImageList_Destroy	
004011B1	CALL <JMP.&user32.PostQuitMessage>	77D6EDEB	USER32.PostQuitMessage	ExitCode
004011EE	CALL <JMP.&user32.GetDlgItemTextA>	77D9AC06	USER32.GetDlgItemTextA	ItemID =
00401202	CALL <JMP.&user32.MessageBoxA>	77D8050B	USER32.MessageBoxA	Caption =
00401219	CALL <JMP.&user32.SetDlgItemTextA>	77D660D5	USER32.SetDlgItemTextA	ControlID
00401229	CALL <JMP.&user32.DestroyWindow>	77D4E666	USER32.DestroyWindow	
0040123C	CALL <JMP.&user32.DefWindowProcA>	77D4DF6B	USER32.DefWindowProcA	

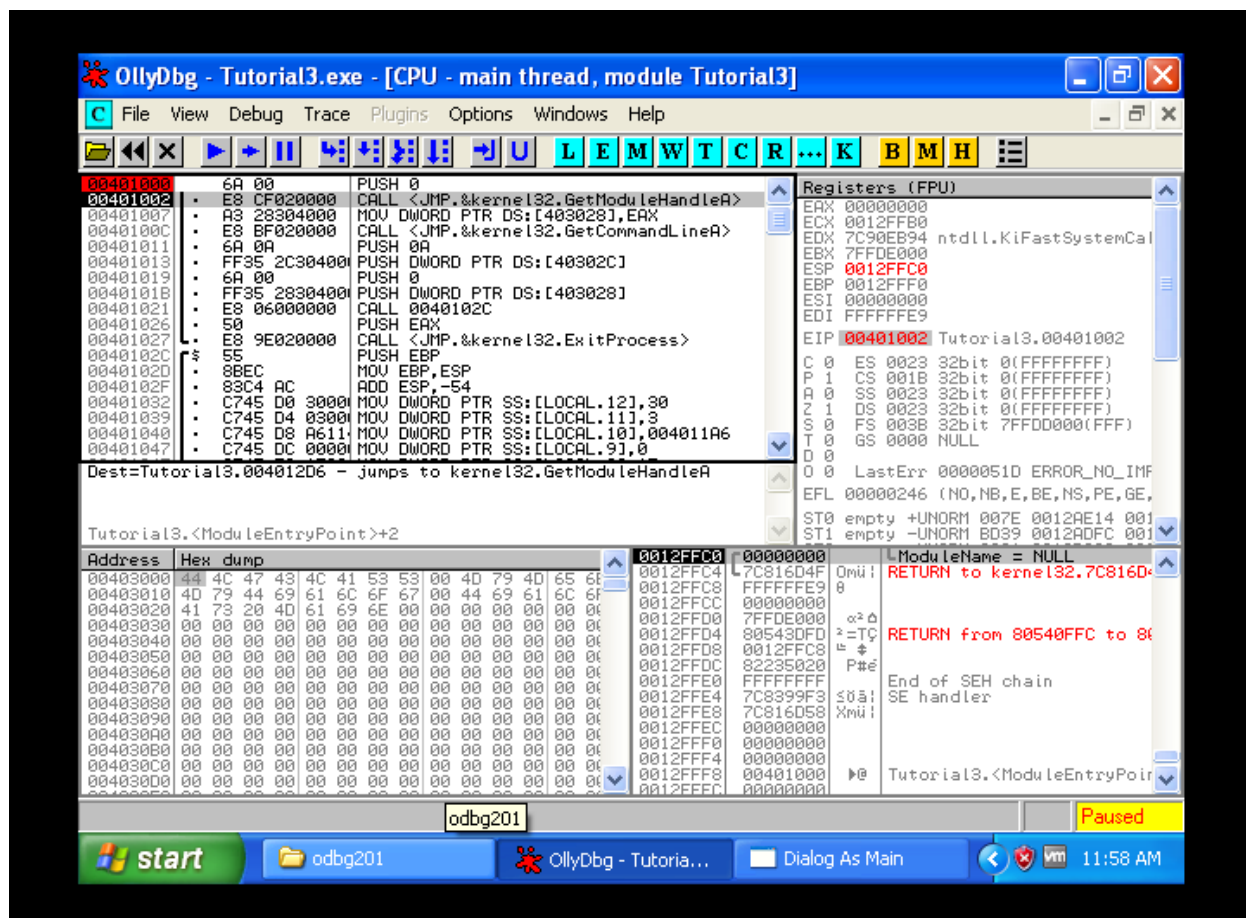
To search for all strings in the app,

Right-click the disassembly window and choose "SearchFor" -> "All Referenced Text Strings"



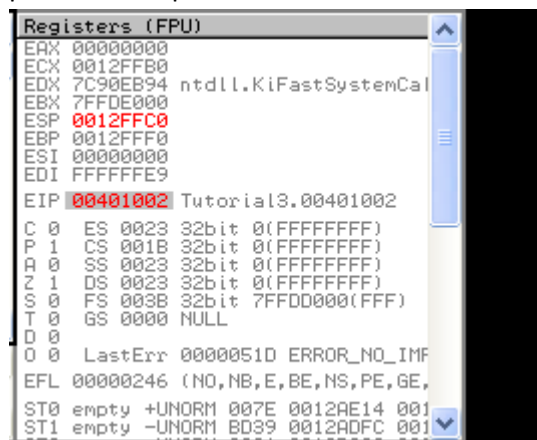
This window shows all text strings it could find in our app.

Try hitting F9 (or choose "Run" from the "Debug" menu option). After a second, program will pop up a dialog box (it may open behind Olly, so minimize Olly's window to make sure.)



To do single-stepping,

Reload the application (reload icon, ctrl-F2, or debug->restart) at the start of the application paused. Now press F8. You will notice that the current line selector has gone down one line.



This is because the instruction that we performed, PUSH 0" 'pushed' a zero onto the stack. This shows up on the stack as "pModule = NULL". NULL is another name for zero. You also may have noticed that in the registers window, the ESP and EIP registers have turned red.

```

00401000 6A 00 PUSH 0
00401002 E8 CF020000 CALL <JMP.&kernel32.GetModuleHandleA>
00401007 A3 28304000 MOV DWORD PTR DS:[403028],EAX
0040100C E8 BF020000 CALL <JMP.&kernel32.GetCommandLineA>
00401011 6A 0A PUSH 0A
00401013 FF35 2C304000 PUSH DWORD PTR DS:[40302C]
00401019 6A 00 PUSH 0
0040101B FF35 28304000 PUSH DWORD PTR DS:[403028]
00401021 E8 06000000 CALL 0040102C
00401026 50 PUSH EAX
00401027 E8 9E020000 CALL <JMP.&kernel32.ExitProcess>
0040102C 55 PUSH EBP
0040102D 8BEC MOV EBP,ESP
0040102F 83C4 AC ADD ESP,-54
00401032 C745 D0 3000 MOV DWORD PTR SS:[LOCAL.12],30
00401039 C745 D4 0300 MOV DWORD PTR SS:[LOCAL.11],3
00401040 C745 D8 A611 MOV DWORD PTR SS:[LOCAL.10],004011A6
00401047 C745 DC 0000 MOV DWORD PTR SS:[LOCAL.9],0

```

KERNEL32.GetModuleHandleA returned EAX = 00400000 ('Tutorial3')
EAX=00400000 (Tutorial3.<STRUCT IMAGE_DOS_HEADER>)

A register turns red means that the last instruction run changed that register.

In this case, the **ESP register** (which points to the address of the top of the stack) was incremented by one since we pushed a new value onto the stack.

The **EIP register**, which points to the current instruction that is being run, has also increased by two. This is because we are no longer on address 401000, but 401002. Running that last instruction was two bytes long and now paused on the next instruction. This instruction is at 401002, the current value of EIP.

The screenshot shows a debugger window with three main panes. The top pane displays assembly code with instructions like `JMP DWORD PTR DS:[&kernel32.GetModuleHandleA]`. The middle pane shows the state of CPU registers, with `EIP` (Instruction Pointer) highlighted in red at address `00401206`. The bottom pane shows a hex dump of memory, with the address `0012FFBC` highlighted. The status bar at the bottom indicates the program is 'Paused'.

Now press F8 once more for “Step-Over”. The current line indicator will move down one, the EIP register will stay red and increase by 5 (as the instruction that ran was 5 bytes long) and the stack was brought back to what it originally showed.

To see the other option, re-start the program (ctrl-F2), press F8 to step over the first instruction, but this time hit F7 on the call instruction. Then the entire window looks different.

File View Debug Trace Plugins Options Windows Help

00401000 6A 00 PUSH 0
 00401002 E8 CF020000 CALL <JMP.&kernel32.GetModuleHandleA>
 00401007 A3 28304000 MOV DWORD PTR DS:[403028],EAX
 0040100C E8 BF020000 CALL <JMP.&kernel32.GetCommandLineA>
 00401011 6A 0A PUSH 0A
 00401013 FF35 2C304000 PUSH DWORD PTR DS:[40302C]
 00401019 6A 00 PUSH 0
 0040101B FF35 28304000 PUSH DWORD PTR DS:[403028]
 00401021 E8 06000000 CALL 0040102C
 00401026 50 PUSH EAX
 00401027 E8 9E020000 CALL <JMP.&kernel32.ExitProcess>
 0040102C 55 PUSH EBP
 0040102D 8BEC MOV EBP,ESP
 0040102F 83C4 AC ADD ESP,-54
 00401032 C745 D0 3000 MOV DWORD PTR SS:[LOCAL.12],30
 00401039 C745 D4 0300 MOV DWORD PTR SS:[LOCAL.11],3
 00401040 C745 D8 A611 MOV DWORD PTR SS:[LOCAL.10],004011A6
 00401047 C745 DC 0000 MOV DWORD PTR SS:[LOCAL.9],0

Registers (FPU)
 EAX 00141EE0 ASCII ""C:\Documen
 ECX 0012FFB0
 EDI 7C90EB94 ntdll.KiFastSystem
 EBX 7FFD7000
 ESP 0012FFC4
 EBP 0012FFF0
 ESI 00000000
 EDI FFFFFFFE9
 EIP 00401011 Tutorial3.00401011
 C 0 ES 0023 32bit 0(FFFFFFFF)
 P 1 CS 001B 32bit 0(FFFFFFFF)
 A 0 SS 0023 32bit 0(FFFFFFFF)
 Z 1 DS 0023 32bit 0(FFFFFFFF)
 S 0 FS 003B 32bit 7FFDF000(FFF
 T 0 GS 0000 NULL
 D 0
 O 0 LastErr 0000051D ERROR_NO_
 EFL 00000246 (NO,NB,E,BE,NS,PE,

KERNEL32.GetCommandLineA returned EAX = ""C:\Documents and Settins
 Stack [0012FFC0]=00401011 (Tutorial3.<ModuleEntryPoint>+11)
 Imm=0000000A (decimal 10.)

This is because F7 is “Step-In”, means the call jumped to a new area of memory (EIP = 4012d6).

Now the program paused at the beginning of the program, hit F8 (Step-Over) 4 times and landed on this statement.

00401000 E8 CF020000 CALL <JMP.&kernel32.GetModuleHandleA>
 00401007 A3 28304000 MOV DWORD PTR DS:[403028],EAX
 0040100C E8 BF020000 CALL <JMP.&kernel32.GetCommandLineA>
 00401011 6A 0A PUSH 0A
 00401013 FF35 2C304000 PUSH DWORD PTR DS:[40302C]
 00401019 6A 00 PUSH 0
 0040101B FF35 28304000 PUSH DWORD PTR DS:[403028]
 00401021 E8 06000000 CALL 0040102C
 00401026 50 PUSH EAX
 00401027 E8 9E020000 CALL <JMP.&kernel32.ExitProcess>
 0040102C 55 PUSH EBP
 0040102D 8BEC MOV EBP,ESP
 0040102F 83C4 AC ADD ESP,-54
 00401032 C745 D0 3000 MOV DWORD PTR SS:[LOCAL.12],30
 00401039 C745 D4 0300 MOV DWORD PTR SS:[LOCAL.11],3
 00401040 C745 D8 A611 MOV DWORD PTR SS:[LOCAL.10],004011A6
 00401047 C745 DC 0000 MOV DWORD PTR SS:[LOCAL.9],0
 0040104E C745 E0 1E00 MOV DWORD PTR SS:[ARG.1],1E
 00401055 FF75 08 PUSH DWORD PTR SS:[ARG.1]
 00401058 8F45 E4 POP DWORD PTR SS:[LOCAL.7]
 0040105B C745 F0 1000 MOV DWORD PTR SS:[LOCAL.4],10

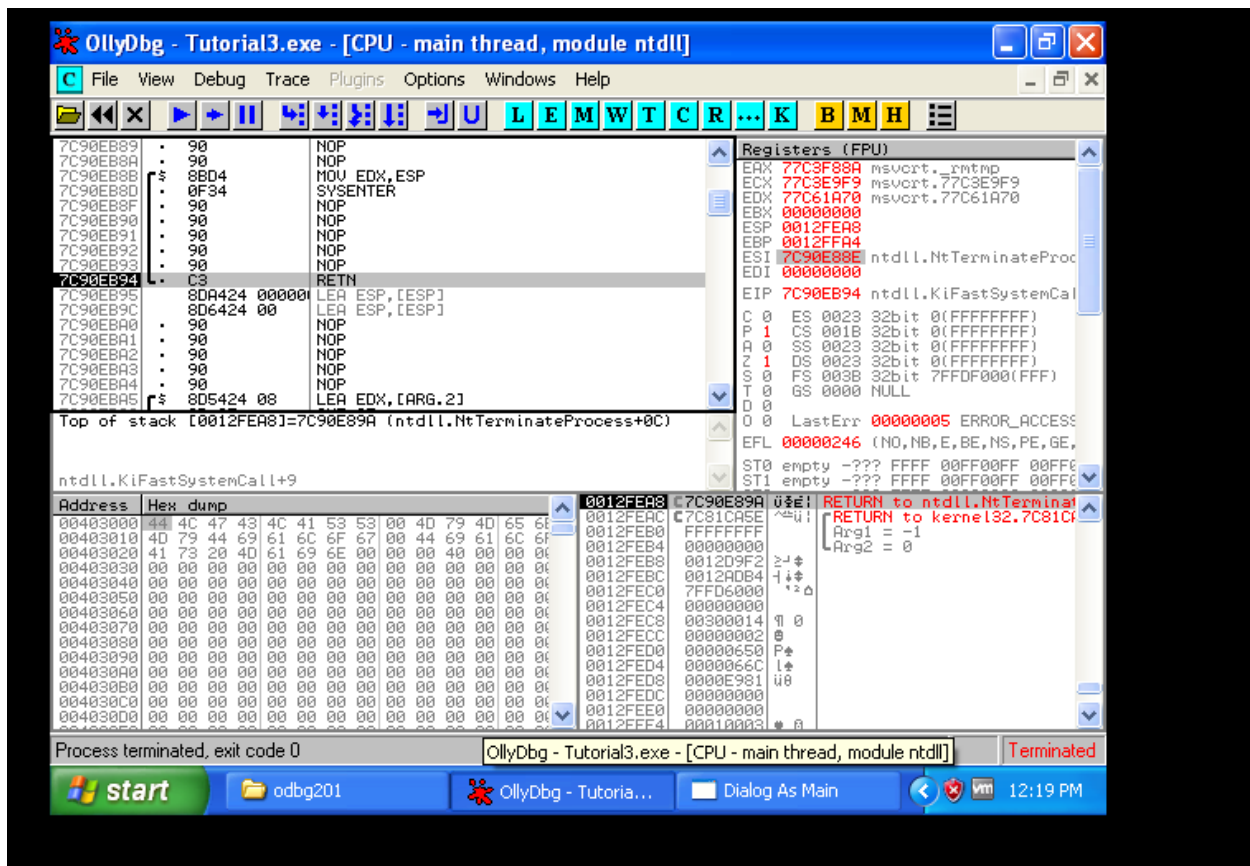
JUMP TO kernel32.getmodulehandlea
 KERNEL32.GetCommandLineA
 Arg3 = 0A
 Arg2 = 0
 Arg1 = Tutorial3.<STRUCT IMAGE_DOS_HEADER>
 Tutorial3.0040102C
 KERNEL32.ExitProcess
 Tutorial3.0040102C(guessed Arg1,Arg2,Arg3,Arg4)
 ExitCode = 0
 Entry point

There are 4 PUSH statements in a row. This time, watch the stack window as you hit F8 4 times and watch the stack grow, it actually grows down.

In this case it is because these 4 numbers are being passed as parameters to a function.

Now press F8 one more time and noticed that it will say “Running” in the active bar in Olly and dialog box will show up. This is because we stepped-over the call that actually has most of the program in it.

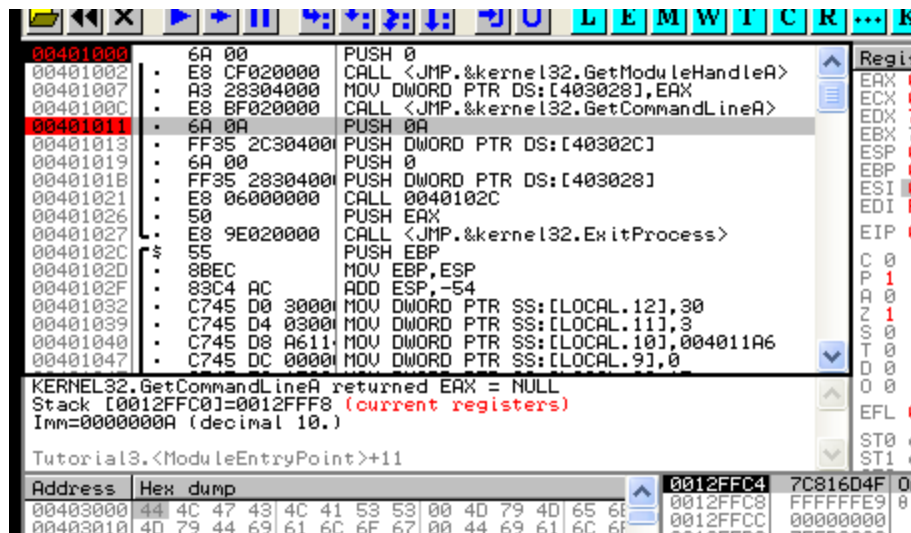
To fix that, Click over the program and hit the ‘close’ button to end the app. Olly will immediately pause on the next line after the call.



You will also notice that our program has disappeared. This is the Windows API that stops an application, Olly has paused program after it has closed the window. But before it has actually been terminated! If now press F9, the program will terminate, the active bar in Olly will say "Terminated" and no longer debugging anything.

Breakpoints

Re-load the app (ctrl-F2) and double-click on the line at address 401011 in the second column. Address 401011 will now turn red. This means set a breakpoint on address 401011. Breakpoints force Olly to pause execution when it reaches it. There are different types of breakpoints in order to stop execution on different events.



Software Breakpoints

A software breakpoint replaces the byte at your breakpoint address with a 0xCC opcode, which is an interrupt 3. This is a special interrupt that tells the operating system that a debugger wishes to pause here and to give control over to the debugger before executing the instruction.

To set a software breakpoint,

Double-click on the opcode column, or highlight the row which you want the breakpoint on, right-click on it, and choose Breakpoints->Toggle (or hit F2).

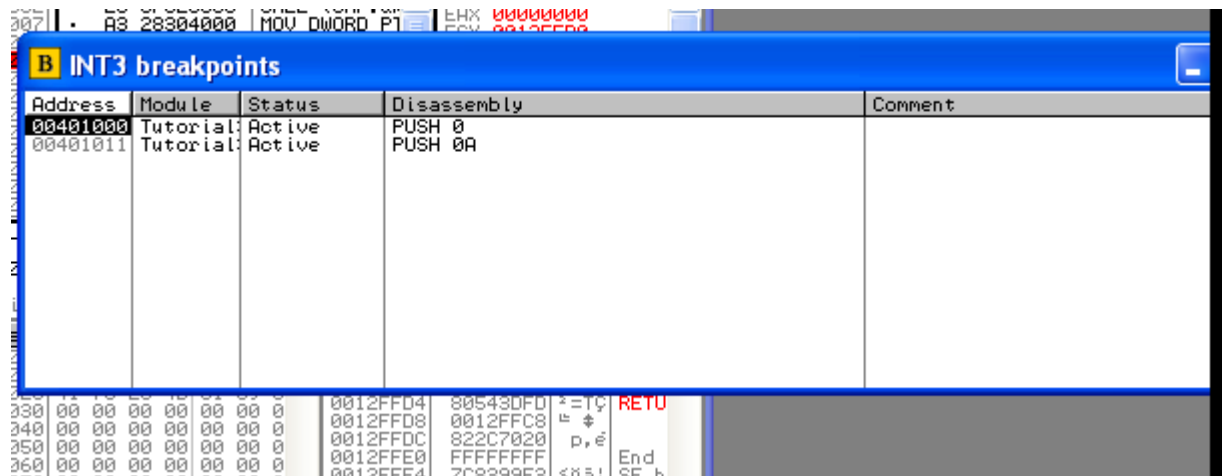
To remove the breakpoint,

Double-click on the same line or right-click and select Breakpoint->Remove Software Breakpoint (or hit F2 again).

To see an entry in the breakpoint window that shows currently set BP,

Click on the "Br" toolbar icon or select View->Breakpoints

As a practical I've a BP (breakpoint) set at address 401011 and program is paused at the first instruction, hit F9 (run) now the program will run but will pause at the line with our BP on it.



If once highlight a breakpoint and click the space bar, the breakpoint will toggle between enabled and disabled. You can also highlight a breakpoint row and hit “DEL” key which will remove the breakpoint.

Lastly, restart the program, go into the breakpoints window, highlight the breakpoint we set at address 401011, and hit the space bar. The “Active” column will change to “Disabled”. Now run the program (F9). You will notice that Olly did not stop at our BP because it was disabled.

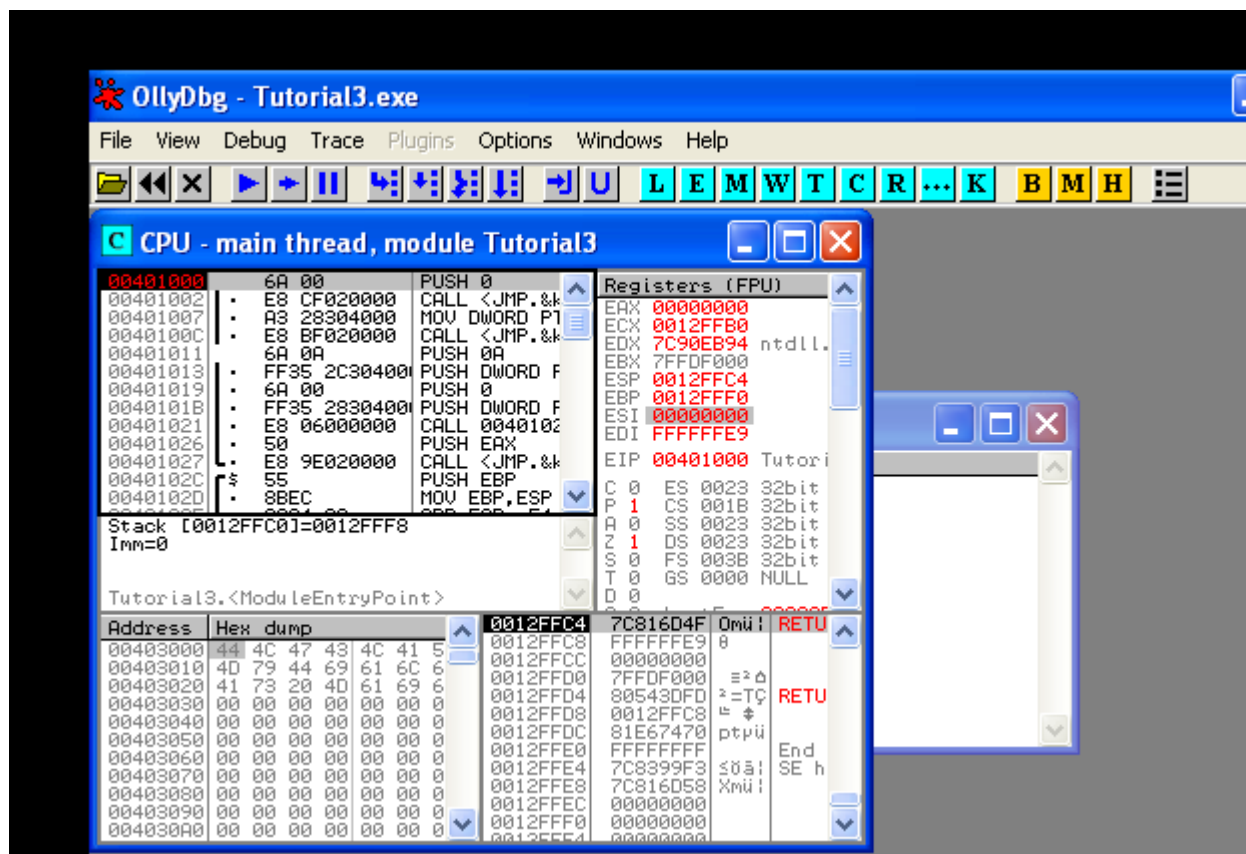
Hardware Breakpoints

A hardware breakpoint uses the CPU’s debug registers. There are 8 of these registers built into the CPU, R0-R7. Even though there are 8 built into the chip, we can only use four of them to breaking on reading, writing or executing a memory section.

The difference between hardware and software breakpoints is hardware BP’s don’t change the process’s memory, so they can be more reliable, especially in programs that are packed or protected.

To set a hardware breakpoint,

Right-clicking on the desired line, selecting Breakpoint, and then choosing Hardware, on Execution.



To see what memory BP's have set,

Select "Debug" and "Hardware Breakpoints"

Memory map								
Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
77D40000	00001000	USER32		PE header	Img	R	RWE Copy	
77D41000	0000F000	USER32	.text	Code, imports, exports	Img	R E	RWE Copy	
77DA0000	00002000	USER32	.data	Data	Img	RW Copy	RWE Copy	
77DA2000	0000B000	USER32	.rsrc	Resources	Img	R	RWE Copy	
77DC0000	00003000	USER32	.reloc	Relocations	Img	R	RWE Copy	
77DD0000	00001000	ADVAPI32		PE header	Img	R	RWE Copy	
77DD1000	000075000	ADVAPI32	.text	Code, imports, exports	Img	R E	RWE Copy	
77E46000	00005000	ADVAPI32	.data	Data	Img	RW Copy	RWE Copy	
77E4B000	0001B000	ADVAPI32	.rsrc	Resources	Img	R	RWE Copy	
77E66000	00005000	ADVAPI32	.reloc	Relocations	Img	R	RWE Copy	
77E70000	00001000	RPCRT4		PE header	Img	R	RWE Copy	
77E71000	00009000	RPCRT4	.text, .ord	Code, imports, exports	Img	R E	RWE Copy	
77EFA000	00001000	RPCRT4	.data	Data	Img	RW	RWE Copy	
77EFB000	00001000	RPCRT4	.rsrc	Resources	Img	R	RWE Copy	
77EFC000	00005000	RPCRT4	.reloc	Relocations	Img	R	RWE Copy	

Memory Breakpoints

Sometimes you may find a string or a constant in the program's memory, but you don't know where in the program it is accessed. Using a memory breakpoint tells Olly that you want to pause whenever ANY instruction in the program reads or writes to that memory address (or groups of addresses.)

To select a memory breakpoint:

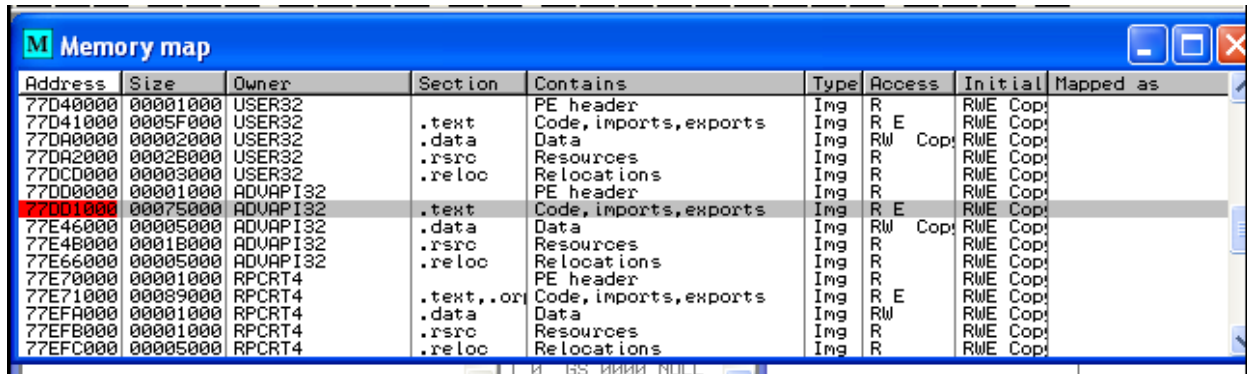
Right-click on the desired line and select Breakpoint->Memory, On Access or Memory, On Write

To set a BP on an address in the memory dump,

Highlight one or more bytes in the dump window, right click on them and select Breakpoint->Memory, On Access or Memory, On Write

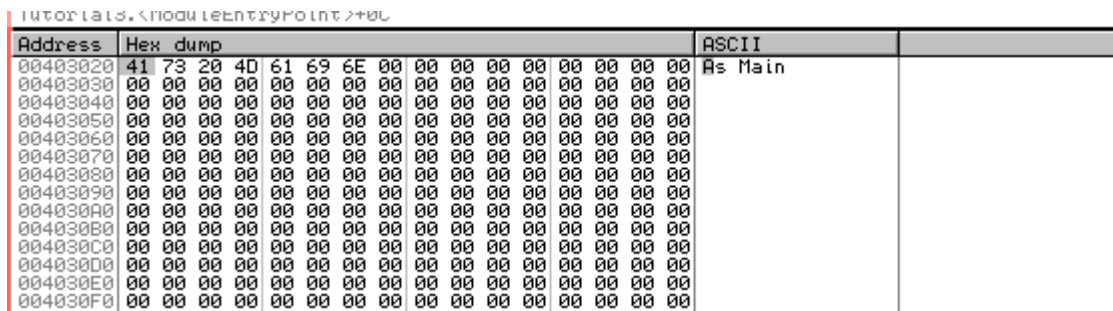
To set a BP for an entire section of memory,

Open Memory window ("Me" icon or View->Memory), right click the section of memory you desire, and right-click and choose "Set Break On Access for either Access or Write.



Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
77D40000	00001000	USER32		PE header	Img	R	RWE Cop	
77D41000	00005F00	USER32	.text	Code, imports, exports	Img	R E	RWE Cop	
77DA0000	00002000	USER32	.data	Data	Img	RW Cop	RWE Cop	
77DA2000	00002B00	USER32	.rsrc	Resources	Img	R	RWE Cop	
77DCD000	00003000	USER32	.reloc	Relocations	Img	R	RWE Cop	
77DD0000	00001000	ADVAPI32		PE header	Img	R	RWE Cop	
77E01000	00007500	ADVAPI32	.text	Code, imports, exports	Img	R E	RWE Cop	
77E46000	00005000	ADVAPI32	.data	Data	Img	RW Cop	RWE Cop	
77E4B000	00001B00	ADVAPI32	.rsrc	Resources	Img	R	RWE Cop	
77E66000	00005000	ADVAPI32	.reloc	Relocations	Img	R	RWE Cop	
77E70000	00001000	RPCRT4		PE header	Img	R	RWE Cop	
77E71000	00009000	RPCRT4	.text, .ord	Code, imports, exports	Img	R E	RWE Cop	
77EFA000	00001000	RPCRT4	.data	Data	Img	RW	RWE Cop	
77EFB000	00001000	RPCRT4	.rsrc	Resources	Img	R	RWE Cop	
77EFC000	00005000	RPCRT4	.reloc	Relocations	Img	R	RWE Cop	

There are three ways to set a memory break point.



Address	Hex dump	ASCII
00403020	41 73 20 40 61 69 6E 00 00 00 00 00 00 00 00 00	As Main
00403030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00403040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00403050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00403060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00403070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00403080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00403090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004030A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004030B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004030C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004030D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004030E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
004030F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

Using the Dump Pane

Right click on reference and select "Follow in Dump" and the dump pane will show you the address section. You can also right-click anywhere in the dump pane and select "Go To" to enter an address to view.

Now, press F8 eight times and it shows the current instruction at address 401021 that says CALL FirstPro.40102c. This is a CALL instruction, so eventually come back to 401021.

Now the screen looked.

Here I typed “waruna ” over top of the “Dialog As Main” string.

04. Now click OK and run the app. Switch over to our program, type something in No tags and select “Options” -> “Get Text”. Now look at the dialog box.