

# IDA – Interactive Disassembler

## Chapter 1 - Preparations

Part 0 – Introduction.....	Page 3
Part 1 – Installing IDA.....	Page 4
Part 2 – Getting the Plugins and Addons.....	Page 4
Part 3 – Installing the Plugins and Addons.....	Page 5
Part 4 – Changing the Auto Comments.....	Page 7
Part 5 – Config Files.....	Page 7

## Chapter 2 – First Approach

Part 6 – Our First Approach.....	Page 8
Part 7 – The Main Window.....	Page 11
Part 8 – Accessing the Plugins.....	Page 13
Part 9 – The Options Dialog.....	Page 14

## Chapter 3 – The different Windows

Part 10 – Hexview.....	Page 20
Part 11 – Function Window.....	Page 21
Part 12 – Names Window.....	Page 22
Part 13 – Strings References.....	Page 23
Part 14 – Imports.....	Page 24
Part 15 – Exports.....	Page 25
Part 16 – Cross-references.....	Page 26
Part 17 – Function Calls.....	Page 27

## Chapter 4 – Navigating through the Code

Part 18 – Arrows in front of the Code.....	Page 28
Part 19 – Following Jumps.....	Page 29
Part 20 – Using the Forward/Backward Arrows.....	Page 30
Part 21 – Using Cross References.....	Page 30
Part 22 – The Jump Menu.....	Page 31

## **Chapter 5 – Making the Code more readable**

Part 23 – Adding Comments.....	Page 33
Part 24 – Adding Lines.....	Page 34
Part 25 – Renaming Functions, Locations and more.....	Page 35

# Chapter 1 – Preparations

## Part 0 – Introduction

Hi everyone,

This is my first tutorial and the first lesson so please don't be rude. Due to the fact that English is not my native language there may be errors. Feel free to contact me so that I can correct them.

Some people may ask why I have written this tutorial since everyone who is into cracking knows how to deal with IDA and newbies normally use W32DASM, changing later when they are advanced. I am trying a different approach. It's 2003 now. W32DASM has lots of mistakes and is less powerful than IDA. I decided to make this tutorial for newbies as a First Approach to IDA so that their first tool is a powerful and helpful one for learning how to crack programs. IDA offers Auto Comments so the Assembler language isn't as cryptic for newbies.

Of course, it is useful to have an Assembler Book as a reference but some things may become clearer by just viewing the comments that may be advanced. I won't expect any Assembler knowledge in this tutorial and Assembler will be addressed in my second tutorial. I want this tutorial to cover the most used functions in IDA. It will not be complete and won't replace the help file from IDA. Make sure to read the help file if you run into problems.

I will try to explain a lot of things with screenshots but don't expect a graphical step-by-step walkthrough for every case. I set goal of one week to complete this tutorial because in one week I promised my first lesson.

!!

2 Paragraphs were delete due to  
internal group infos and nicks  
from group members

!!

## **Part 1 – Installing IDA**

Installing IDA is very simple because it doesn't really need any installation. Just extract all the files from the release you have to your favorite folder and make sure to extract the subdirectories properly.

e.g. *C:\Program Files\Datarescue\IDA PRO Advanced 4.30\*

In this tutorial I will use IDA PRO Advanced 4.30

## **Part 2 – Getting the Plugins and Addons**

Ok, so far so good. There are some nice Plugins and Addons out in the web. The ones I mention here are very useful and I am sure you will need them often. Some will be useful when you try to crack harder programs.

- LoadINT 4.21 – For changing the AutoComments displayed in IDA
- Flair Tools 4.16 – For creating your own signatures
- SIE Plugin – Adds Windows for Strings, Imports, Exports to IDA
- Ida2Softice – Creates NMS files of your current Database which make Debugging your apps easier
- Ida 4.3 SDK - Gives you the possibility to write your own Plugins

Get them at the following URLs:

<http://mostek.subcultural.com>  
<http://wasm.ru/toollist.php?list=13>

**Notice: In case the URLs are down, don't ask the people mentioned in the Introduction or me to send you one of these files. We don't spread Warez and Files so don't even try. Use [www.google.com](http://www.google.com) or your favorite search engine to find the files.**

## **Part 3 – Installing the Plugins and Addons**

### LoadInt 4.21:

Extract all files to your main IDA folder. Make sure to rename the README file from the LoadInt 4.21 package to something else for further usage.

e.g. *C:\Program Files\Datarescue\IDA PRO Advanced 4.30\*

### Flair Tools 4.16:

Create a sub directory in your IDA folder and extract all files including subdirectories to that folder.

e.g. *C:\Program Files\Datarescue\IDA PRO Advanced 4.30\Flair Tools*

### Ida Pro SDK 4.30:

Create a subdirectory in your IDA folder and extract all files including subdirectories to that folder.

e.g. *C:\Program Files\Datarescue\IDA PRO Advanced 4.30\Ida SDK 4.3*

### SIE Plugin:

Extract the files to a temporary directory and copy the file *plugs.plw* to your IDA plugin directory.

e.g. *C:\Program Files\Datarescue\IDA PRO Advanced 4.30\Plugins*

Make sure to copy the correct file concerning your version of IDA to your Plugin folder. For example, if you use IDA PRO Advanced 4.30 use the *plugs.plw* from the following folder:

e.g. *C:\tempdirectory\4.30\plugs.plw*

### SIE Plugin (continued):

Now you need to edit the file *plugins.cfg* in your IDA Plugins folder.

e.g. *C:\Program Files\Datarescue\IDA PRO Advanced 4.30\Plugins\Plugins.cfg*

You need to add the following lines at the end of the file:

Strings_BugFix	plugs	0	3
Exports	plugs	SHIFT-E	2
Imports	plugs	SHIFT-I	1
Strings	plugs	SHIFT-S	0

Save the file afterwards and delete the temporary folder. See the explanations in the file *Plugins.cfg* for further details.

### Ida2Softice Plugin:

Extract the files to a temporary directory and copy the file *i2s.plw* to your IDA Plugin directory.

e.g. *C:\Program Files\Datarescue\IDA PRO Advanced\4.30\Plugins*

Make sure to copy the correct file concerning your version of IDA to your Plugin folder. For example if you use IDA PRO Advanced 4.30 use the *i2s.plw* from the following folder:

e.g. *C:\tempdirectory\4.30\i2s.plw*

Now you need to edit the file *plugins.cfg* in your IDA Plugins folder.

e.g. *C:\Program Files\Datarescue\IDA PRO Advanced\Plugins.cfg*

You need to add the following lines at the end of the file:

I2S_Setup	i2s	0	3
I2S_Source_Info	i2s	Ctrl-F12	2
I2S_Save_NMS	i2s	Shift-F12	1
I2S_Conversion	i2s	F12	0

Save the file afterwards and delete the temporary folder. See the explanations in the file *Plugins.cfg* for further details.

## **Part 4 – Changing the Autocomments**

If you have installed LoadINT 4.21 there will be a file *PC.CMT* in your IDA main folder. Use your favorite editor to change and advance the comments for the Assembler command to your needs. Also, take a good Assembler book and advance the comments to your liking. However, this is just a hint that might be helpful as you start to use Assembler.

This is what you need to do so that IDA shows your changed Autocomments.

1. Edit the *PC.CMT* with your favorite editor and save it
2. Call the File *COMPILE.BAT* from the IDA main folder

For further details take a look at the Readme file included with LoadINT 4.21. Make sure that IDA is not running when running *COMPILE.BAT* or the file will produce an error even if the syntax of the CMT files is correct. That is because the program tries to write to the file *IDA.INT* and if IDA is running, the file is open and can't write to it.

e.g. *C:\Program Files\Datarescue\IDA PRO Advanced 4.30\PC.CMT*

e.g. *C:\Program Files\Datarescue\IDA PRO Advanced 4.30\Compile.bat*

e.g. *C:\Program Files\Datarescue\IDA PRO Advanced 4.30\README*

## **Part 5 – Config Files**

It is very useful to know the configuration files because IDA doesn't save the options you set. After reading the following chapters you might want to make some changes. I won't give you details about the configuration files. In fact, they are very well commented and it should be an easy task to change them to your needs, for example adding Macros, changing Hotkeys or just changing the display.

Here are the locations and names of the configuration files:

e.g. *C:\Program Files\Datarescue\IDA PRO Advanced\IDA.CFG*

e.g. *C:\Program Files\Datarescue\IDA PRO Advanced\IDAGUI.CFG*

e.g. *C:\Program Files\Datarescue\IDA PRO Advanced\IDATUI.CFG*

e.g. *C:\Program Files\Datarescue\IDA PRO Advanced\SIG\Autoload.cfg*

e.g. *C:\Program Files\Datarescue\IDA PRO Advanced\Plugins\Plugins.cfg*

e.g. *C:\Program Files\Datarescue\IDA PRO Advanced\IDS\IDS NAMES*

There are more but they are for different processor modules that we won't need here.

## Chapter 2 – First Approach

### Part 6 – Our First Approach

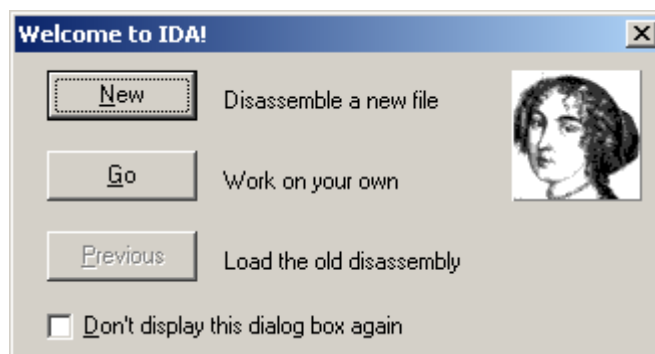
After showing you many things and doing a lot of preparations, it is time to start IDA and take a first look at the program. There are a lot of executable files. Which is the correct one?

IDA2.EXE	This file is used when you run OS/2
IDAX.EXE	This file is used when you run DOS and want to run it with DOS/4GW Extension in Protected Mode
IDAW.EXE	This file is used for normal DOS Mode
IDAG.EXE	This file is used when you use Win95 or above and features a nice Graphical User Interface

In this tutorial I will only handle the GUI version of IDA because it's the most used version.

Start *IDAG.EXE* and press "OK" when the License Dialog is shown.

Now you should see a Dialog that gives you three choices:



1. New (Disassemble a new File)
2. Go (Work on your own. This will start IDA without disassembling a file)
3. Previous (Load a previously disassembled file)

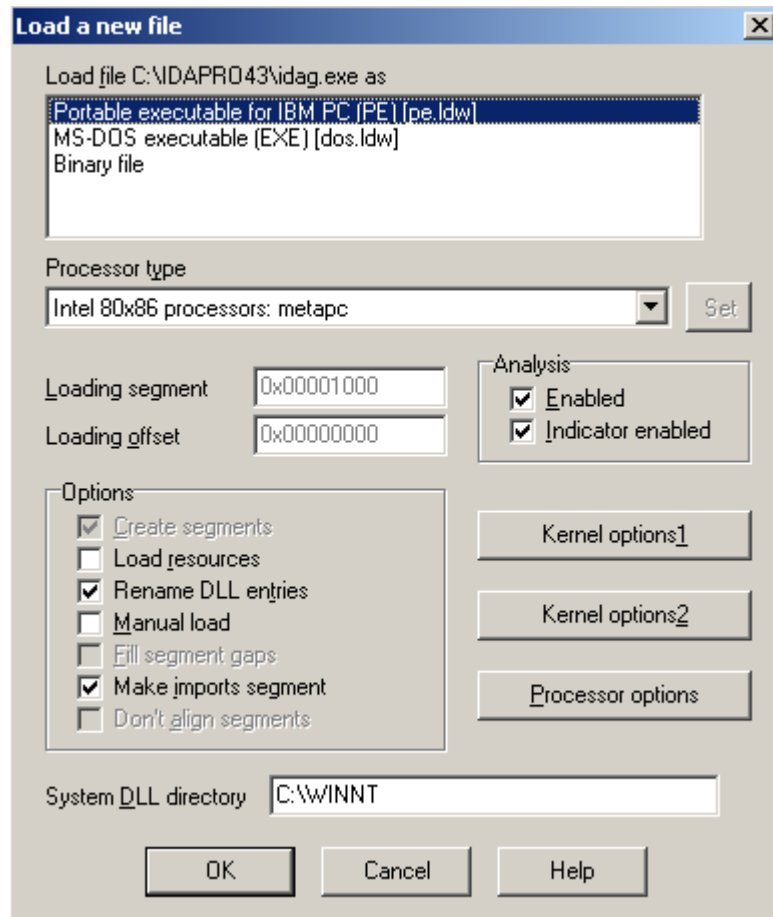
If you select "Don't display this dialog box again" you will automatically start in the 2<sup>nd</sup> mode the next time. In case the dialog box is still shown at the start of the program check the configuration files and make a setting there.



This is your first start now so choose “New”. Next there is a File Dialog where you can select the file you wish to disassemble. I suggest you choose IDAG.EXE in our IDA main folder and press “OK” afterwards.

e.g. C:\Program Files\Datarescue\IDA PRO Advanced 4.30\IDAG.EXE

Now IDA prompts with another Dialog that looks like the following:



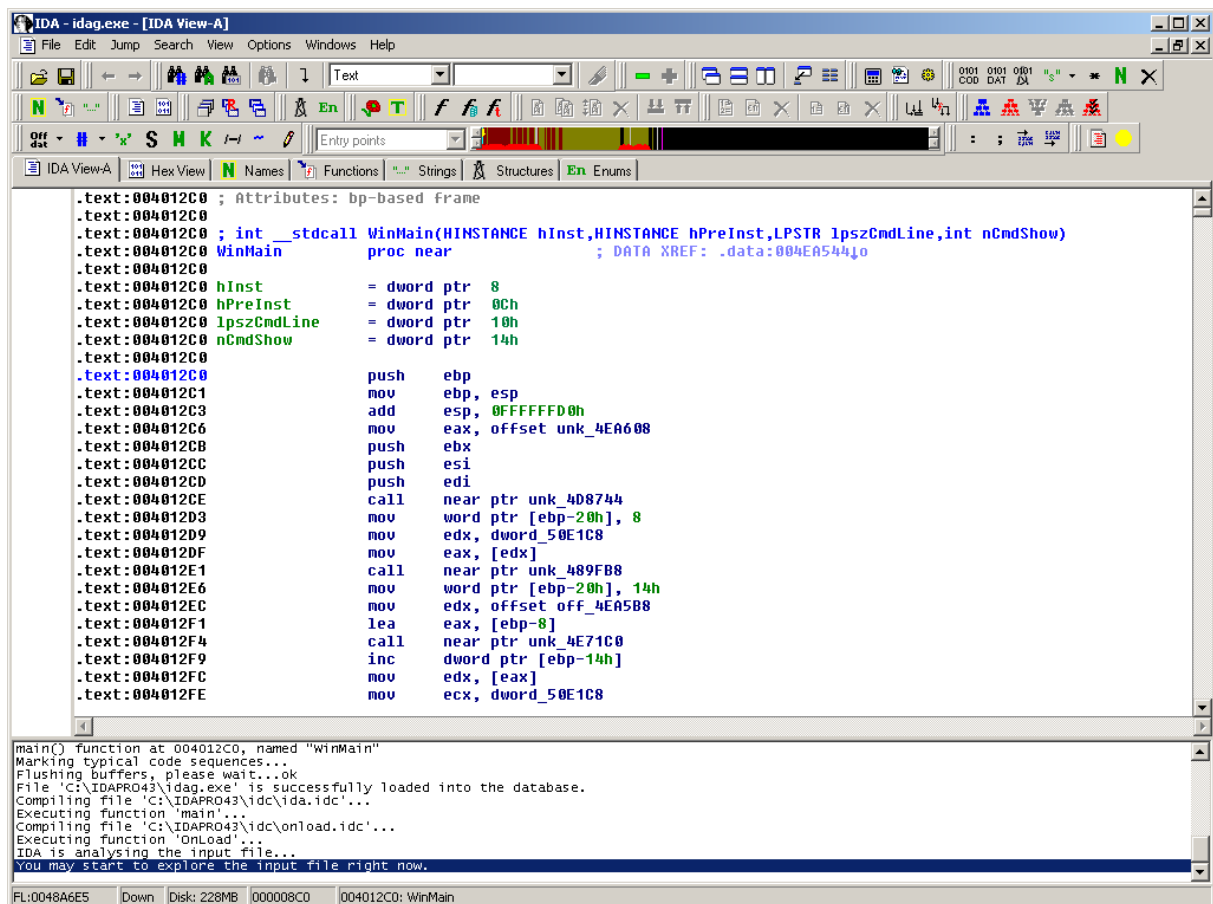
In this dialog we can tell IDA what we know about our file. Most Windows files are so-called PE files. It is a defined format of how the files look. So in 99% of all cases this is the correct choice.

For Processor type we keep the setting :  
Intel 80x86 processors: metapc.

This means IDA tries to use all possible Assembler commands even the Intel specific ones and MMX instruction set to show us our code. There are more I haven't mentioned but this setting is the most useful. If you know exactly for which CPU the program was written, here is where you can change the processor the program was written for. Also, here we keep *metapc* in 99% of all cases.

After knowing how to handle IDA you may take a look at the Kernel Options and Processor Options to do some fine-tuning but for now just keep the standard settings and press "OK".

Now IDA should start working. After displaying some messages and building up the screen, you are able to take a first look at your disassembled code (Deadlisting). The first thing we do now is arranging the Toolbar and moving the Overview Navigation Window to the Toolbars. Next, increase the window size of "IDA View A" to maximum. Now our program should look like the following:



```
.text:004012C0 ; Attributes: bp-based frame
.text:004012C0
.text:004012C0 ; int __stdcall WinMain(HINSTANCE hInst,HINSTANCE hPreInst,LPSTR lpszCmdLine,int nCmdShow)
.text:004012C0 WinMain      proc near          ; DATA XREF: .data:004E0544j0
.text:004012C0
.text:004012C0 hInst       = dword ptr  8
.text:004012C0 hPreInst    = dword ptr  0Ch
.text:004012C0 lpszCmdLine = dword ptr  10h
.text:004012C0 nCmdShow   = dword ptr  14h
.text:004012C0
.text:004012C0      push     ebp
.text:004012C1      mov      ebp, esp
.text:004012C3      add     esp, 0FFFFFFD0h
.text:004012C6      mov     eax, offset unk_4EA608
.text:004012C8      push   ebx
.text:004012CC      push   esi
.text:004012CD      push   edi
.text:004012CE      call   near ptr unk_4D8744
.text:004012D3      mov    word ptr [ebp-20h], 8
.text:004012D9      mov    edx, dword_50E1C8
.text:004012DF      mov    eax, [edx]
.text:004012E1      call   near ptr unk_489FB8
.text:004012E6      mov    word ptr [ebp-20h], 14h
.text:004012EC      mov    edx, offset off_4EA5B8
.text:004012F1      lea   eax, [ebp-8]
.text:004012F4      call   near ptr unk_4E71C0
.text:004012F9      inc   dword ptr [ebp-14h]
.text:004012FC      mov    edx, [eax]
.text:004012FE      mov    ecx, dword_50E1C8

main() function at 004012C0, named "winMain"
Marking typical code sequences...
Flushing buffers, please wait...ok
File 'C:\IDAPRO43\idag.exe' is successfully loaded into the database.
Compiling file 'C:\IDAPRO43\idc\ida.idc'...
Executing function 'main'...
Compiling file 'C:\IDAPRO43\idc\onload.idc'...
Executing function 'onLoad'...
IDA is analysing the input file...
You may start to explore the input file right now.
```

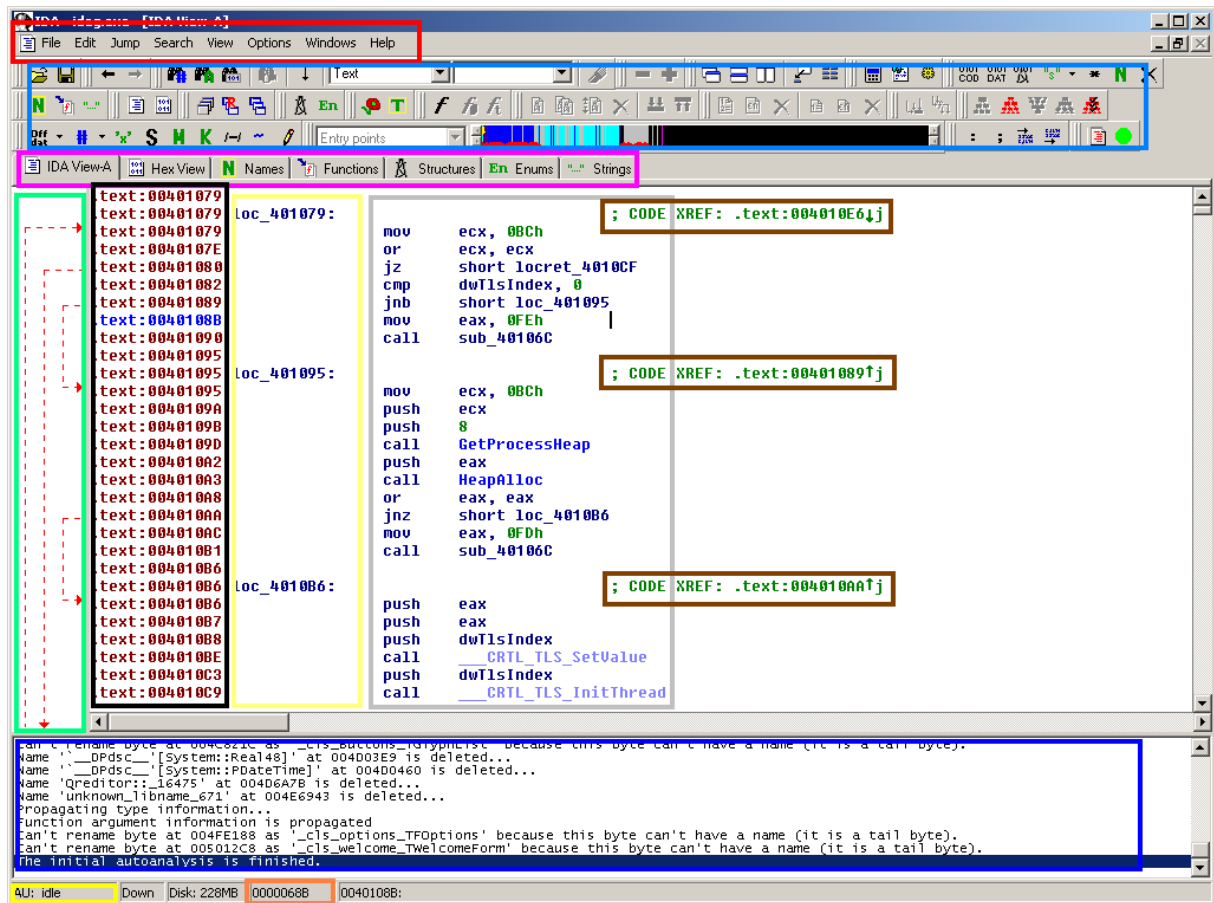
FL:0048A6E5    Down    Disk: 228MB    000008C0    004012C0: WinMain

The yellow small circle (light) at the left of our Toolbar shows us that IDA is thinking and still working on creating our Deadlisting. There are three possible colors:

- Green : Ready
- Yellow: Thinking
- Red: Critical

Depending on your CPU, the disassembling may take some time. When the disassembling is finished, the small circle will turn green and a message saying “The initial autoanalysis is finished” will be displayed in the status window.

## Part 7 – The Main Window



Here is a short explanation of the shown information:

Red Rectangle:

Like in every other program we see the Menu Bar of IDA

Light Blue Rectangle:

Toolbar to reach most of the options of IDA by clicking on the icon

Pink Rectangle:

Different windows like “IDA View A” (our main view), Hexview, Strings, Names, Functions, Imports, Exports, Crossreferences and so on

Green Rectangle:

Arrows show where the jumps in the code block lead and are useful to recognize small loops or to follow a function.

Black Rectangle:

The section name followed by the virtual address: This is the same address you would see in Softice while debugging

Light Yellow Rectangle:

The Code locations can be compared to jump marks. Every location jumped to is marked like that except the functions themselves.

Grey Rectangle:

The Code of our disassembled program.

Brown Rectangles:

Code References: They show from which points of the program the Code locations are accessed. When double clicking on them you reach the code where the location or function is called or jumped to.

Dark Blue Rectangle:

This is the Status window that shows our last actions and tells us what IDA is doing at the moment.

Dark Yellow Rectangle:

A small status display that shows if IDA is working at the moment and at which location it is working.

Orange Rectangle:

The file offset of our current code location: Very useful when you try to patch a program and want to know the translation from virtual address <-> file offset to find the correct location you want to patch.

Blue Text in Black Rectangle:

Our current code line we are working on.

## **Part 8 – Accessing the Plugins**

In Part 3 of this tutorial I explained how to install plugins for IDA. Now I will show you how to reach them in IDA.

There are two ways. You can either use the hotkey you defined when adding the lines to the file *plugins.cfg* or you can access them manually by going through the following menus:

e.g. *Edit/Plugins/String References*

e.g. *Edit/Plugins/Imports*

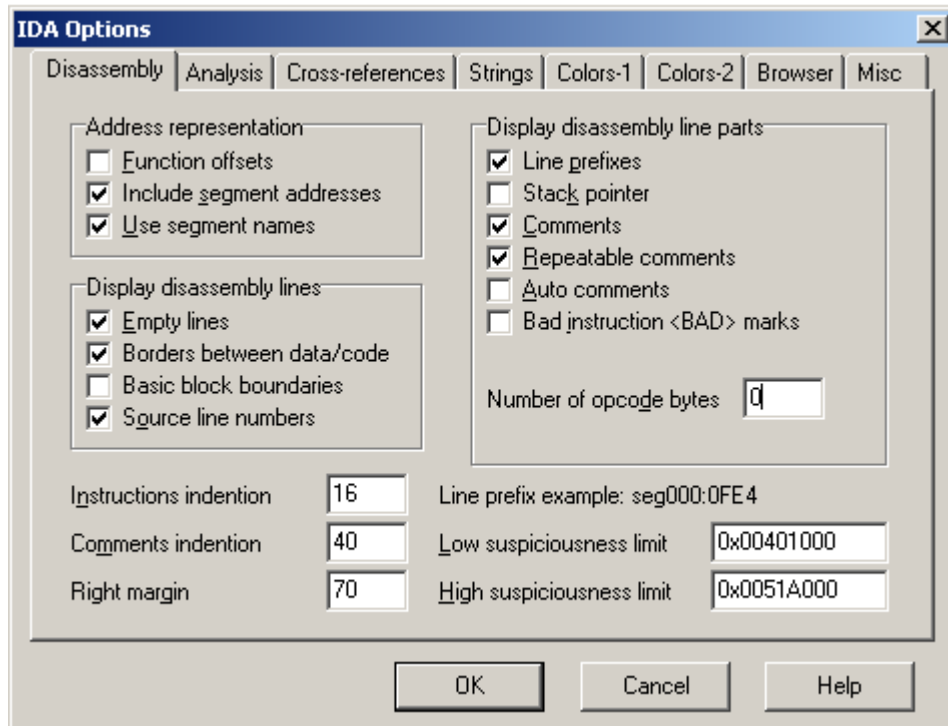
e.g. *Edit/Plugins/Exports*

e.g. *Edit/Plugins/Ida2Softice*

After calling them they will either pop up a new window or a new window is shown at the Pink Rectangle Area.

## **Part 9 – Showing more information and Alignment**

To access the Options, select Options/General from the Menu bar and you should get the following Popup:



Now check the following checkboxes:

Stack pointer, Auto comments, Bad instruction marks, and Basic Block boundaries

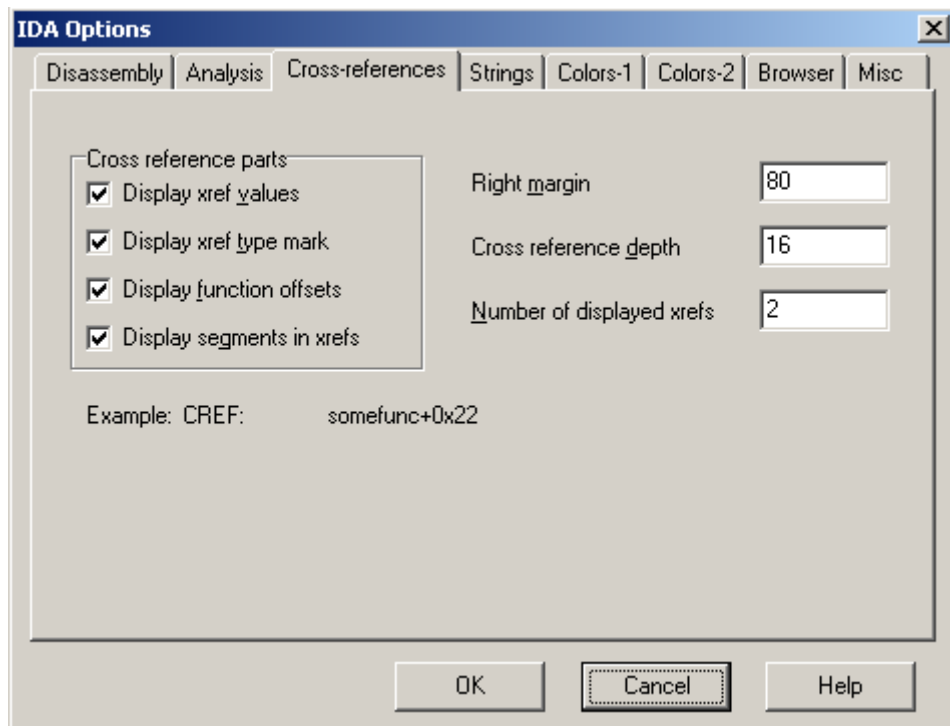
Enter the number “8” at the textfield at “Number of opcode bytes”

Take a look at the 2 textboxes in the bottom left called:

Instruction indentation, Comments indentation

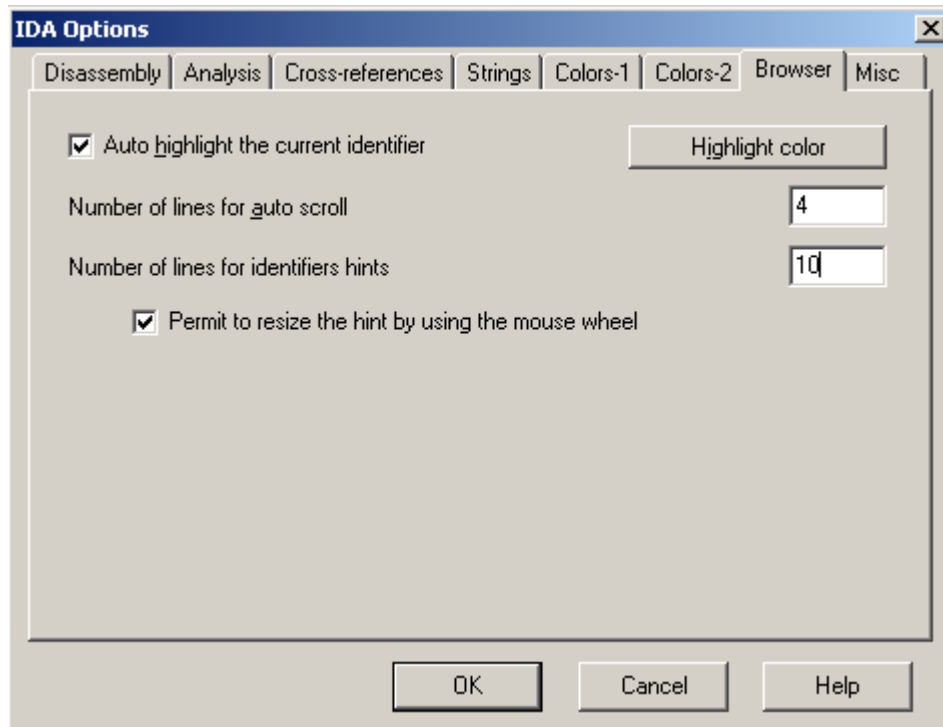
By increasing these numbers you move these parts to the right. By decreasing them you move them to left. Play with these values later to arrange your display and enter these values in your configuration files.

Now switch to the Cross-references window at the top and you should see the following:



The interesting part here is “Number of displayed xrefs”. In Part 7, I gave a short explanation of Cross-references. IDA normally shows only 2 of them but suppose a function is called 5 times in our code and we want to check all five calls to this function. We need to show more than 2. I suggest you set a value around 200 or so in this box and that should normally show you all Cross-references of the code. Also, make sure to change this value in the configuration file.

In the Menu Color-1 and Color-2 you are able to change all colors as you wish. I won't give another explanation for this because it should be very simple. Instead of the colors we will take a small look at the "Browser" menu. It is also very useful because it gives us the opportunity to see code by just moving over a jump or a call without going to this location.



Useful settings here are:

Auto highlight the current identifier : Yes

Number of lines for auto scroll : 4

Number of lines for identifiers hints : 30-40

Permit to resize the hint by using the mouse wheel : yes

That was lots of settings. Now let's see the effects. Our Window should now look something like the picture on the following page:



```

.text:0040FDE1 050 8B 55 B8      mov edx, [ebp+var_40]
.text:0040FDE4 050 52             push edx
.text:0040FDE5 060 E8 7F 86 0C 00 call ]_strchr_0 ; Call Procedure
.text:0040FDE5      ;
.text:0040FDE6 060 83 C4 08      add esp, 8 ; Add
.text:0040FDED 058 8B F8      mov edi, eax
.text:0040FDEF 058 85 FF      test edi, edi ; Logical Compare
.text:0040FDF1 058 74 05      jz short loc_40FDF8 ; Jump if Zero (ZF=1)
.text:0040FDF1      ;
.text:0040FDF1      ;
.text:0040FDF1      ;
.text:0040FDF3 058 47             inc edi ; Increment by 1
.text:0040FDF4 058 C6 47 FF 00   mov byte ptr [edi-1], 0
.text:0040FDF8      ;
.text:0040FDF8      ; CODE XREF: sub_40FD10+E1F
.text:0040FDF8 058 85 FF      test edi, edi ; Logical Compare
.text:0040FDF8      ;
.text:0040FDF8 058 75 08      jnz short loc_40FE04 ; Jump if Not Zero (ZF=0)
.text:0040FDF8      ;
.text:0040FDF8      ;
.text:0040FDF8      ;
.text:0040FDFC 058 8D 43 01      lea eax, [ebx+1] ; Load Effective Address
.text:0040FDFF 058 E8 6C E3 FF FF call sub_40E170 ; Call Procedure
.text:0040FDFF      ;
.text:0040FE04      ;
.text:0040FE04      ; CODE XREF: sub_40FD10+E1F
.text:0040FE04 058 8A 17      mov dl, [edi]
.text:0040FE06 058 3A 55 0C      cmp dl, [ebp+arg_4] ; Compare Two Operands
.text:0040FE09 058 75 5B      jnz short loc_40FE66 ; Jump if Not Zero (ZF=0)
.text:0040FE09      ;
.text:0040FE09      ;
.text:0040FE09      ;
.text:0040FE0B 058 66 C7 45 E4 2C 00 mov [ebp+var_1C], 2Ch
.text:0040FE11 058 8B 45 0C      mov eax, [ebp+var_44]
.text:0040FE14 058 E8 67 E3 FF FF call sub_40E180 ; Call Procedure
.text:0040FE14      ;

```

Red Rectangle:

Here we see the Stack Pointer. Every time we put (“push”) something on the stack the number is increased by 4 and every time we get something from the stack (“pop”) the number is decreased by 4. This is sometime helpful to see which pushes belong to a certain call.

Orange Rectangles:

Part 4 explained how to manually change these Auto comments. Here you see the use of it. Each line is automatically commented with these predefined comments. It is very helpful to see what these commands in the Codeblock actually do. And as a newbie you will see that Assembler isn't as cryptic as it seems at first.

## Green Rectangle:

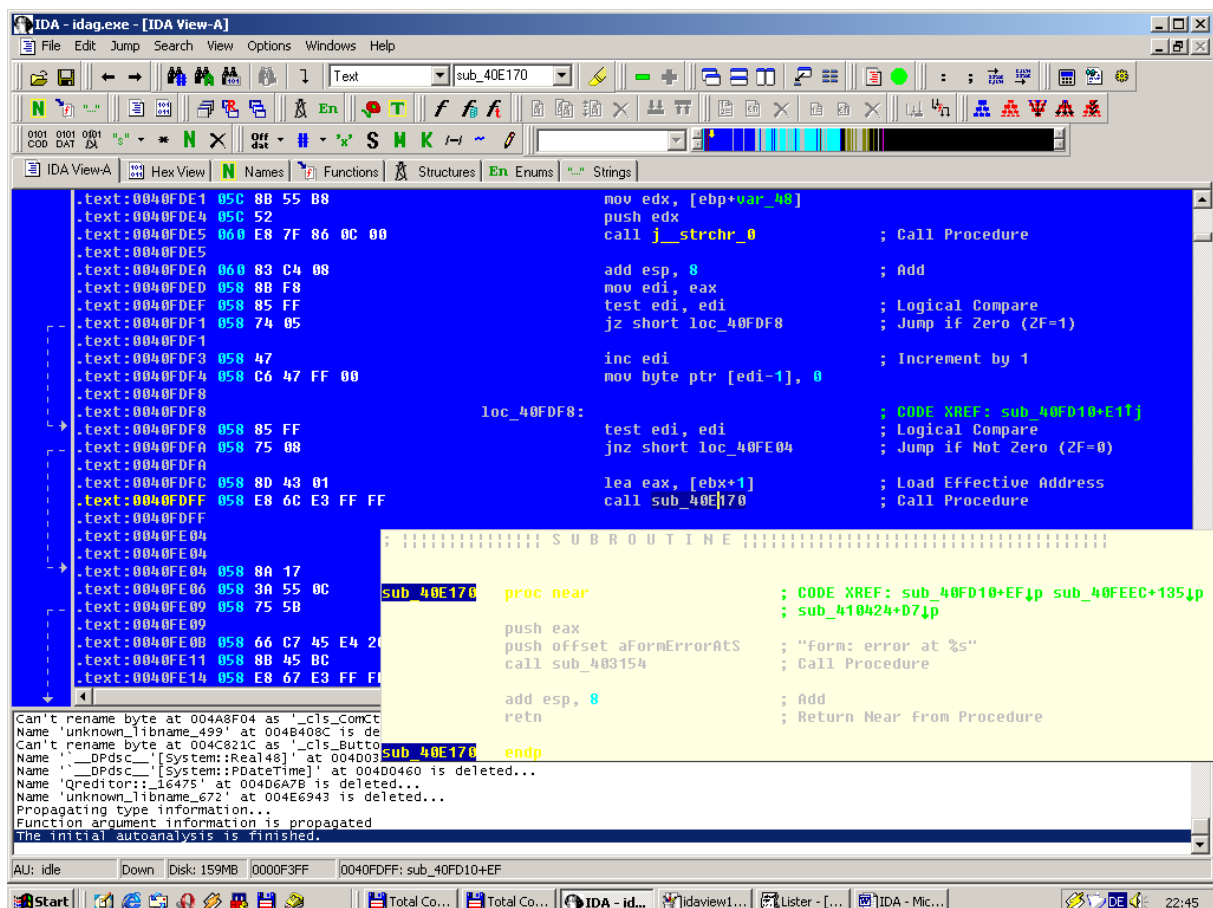
Here we see our Opcodes. It is nothing more than the Hexvalues for the command shown at the left. But why have I switched this on now? Sometimes when you need to patch a program you will see exactly this numbers in your Hexeditor. Maybe you read some tutorials about changing a 74h to 75h without knowing what this means. Here is a short explanation with an example. Take these two lines :

```
.text:0040FDF1  058 74 05  jz short loc_40FDF8  ; Jump if Zero (ZF=1)
.text:0040FDFA  058 75 08  jnz short loc_40FE04  ; Jump if Not Zero (ZF=0)
```

Ignore the 058 as representation of the Stack-pointer here. Each Instruction here is represented by 2 byte and the 74 stands for JZ (Jump if Zero) and the 75 stands for JNZ (Jump if Not Zero). So changing 74h to 75h at the virtual address 0040FDF1 would give you the following code:

```
.text:0040FDF1  058 75 05  jnz short loc_40FDF8  ; Jump if Not Zero(ZF=0)
```

## The Browser Function:



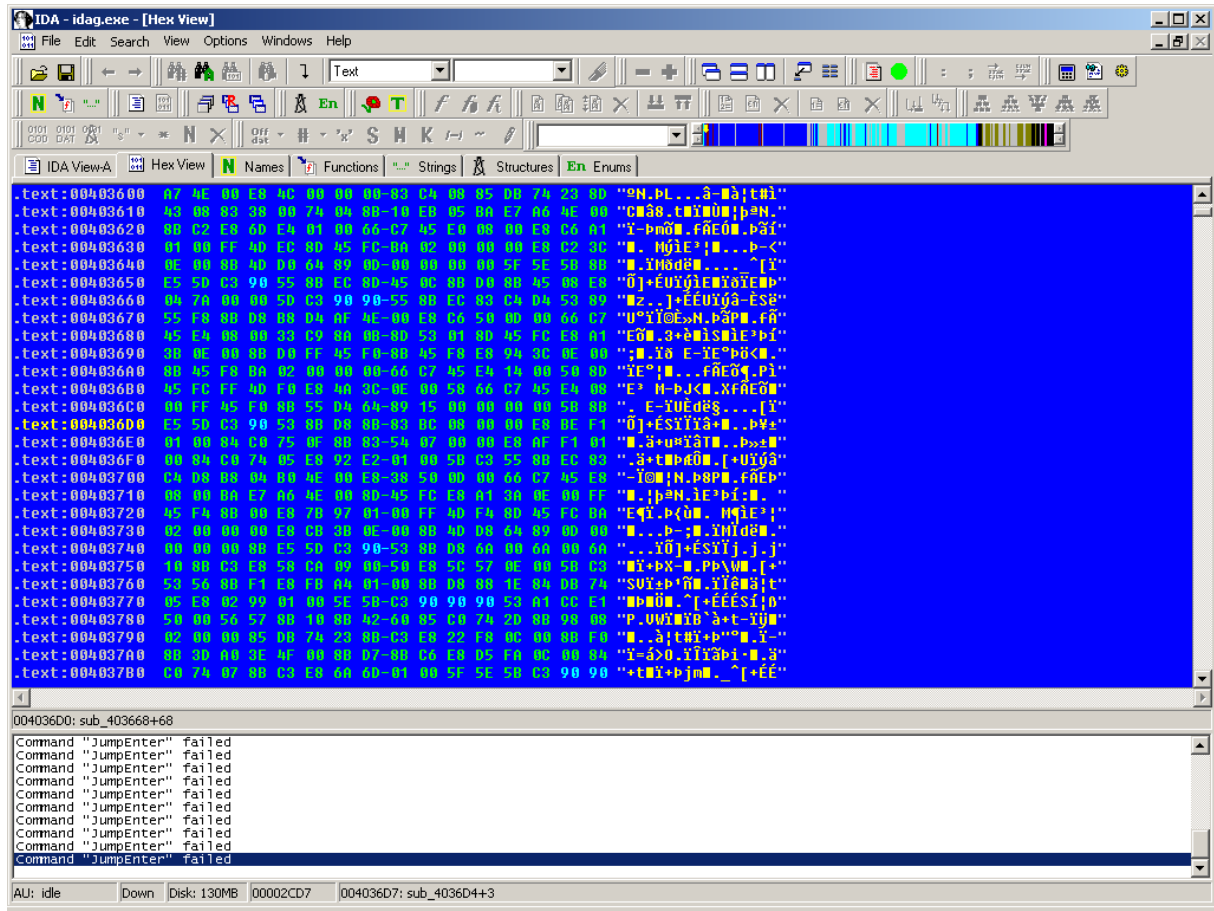
By just clicking once on a call or jump you can see a preview of the code you would reach by double-clicking on it. Sometimes a preview is enough information to decide if you want to take a further look or just stay at the place you are now.

### Basic Block Boundaries:

Maybe you don't recognize this setting at once, but there are spaces between the code lines that make it easier to read. Normally all code lines are displayed directly after each other.

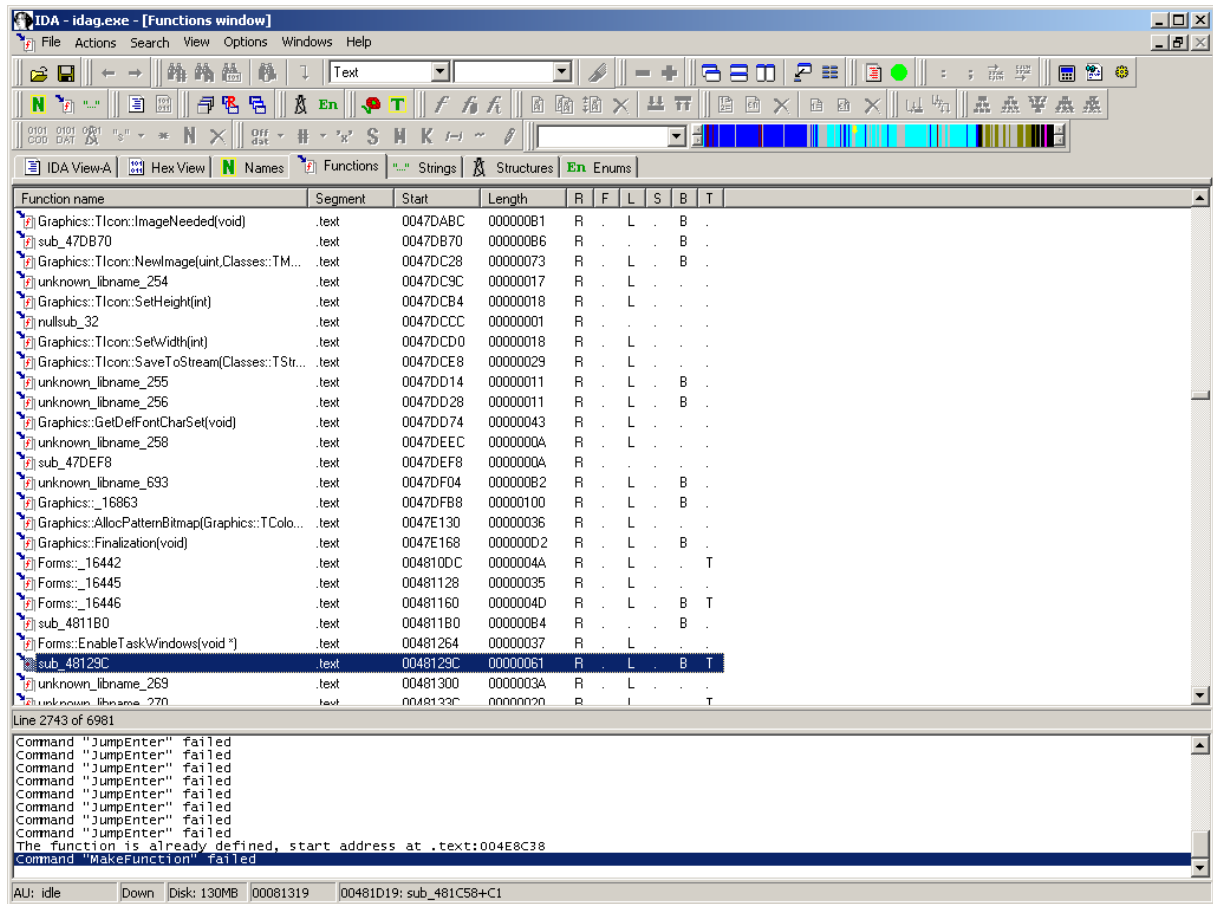
## Chapter 3

### Part 10 –Hex View



This Window doesn't need much explanation. By clicking on a Hexvalue you will automatically go to the code location in the Main View although it displays the message "Command 'JumpEnter' failed" in the Status window.

## Part 11 – Functions

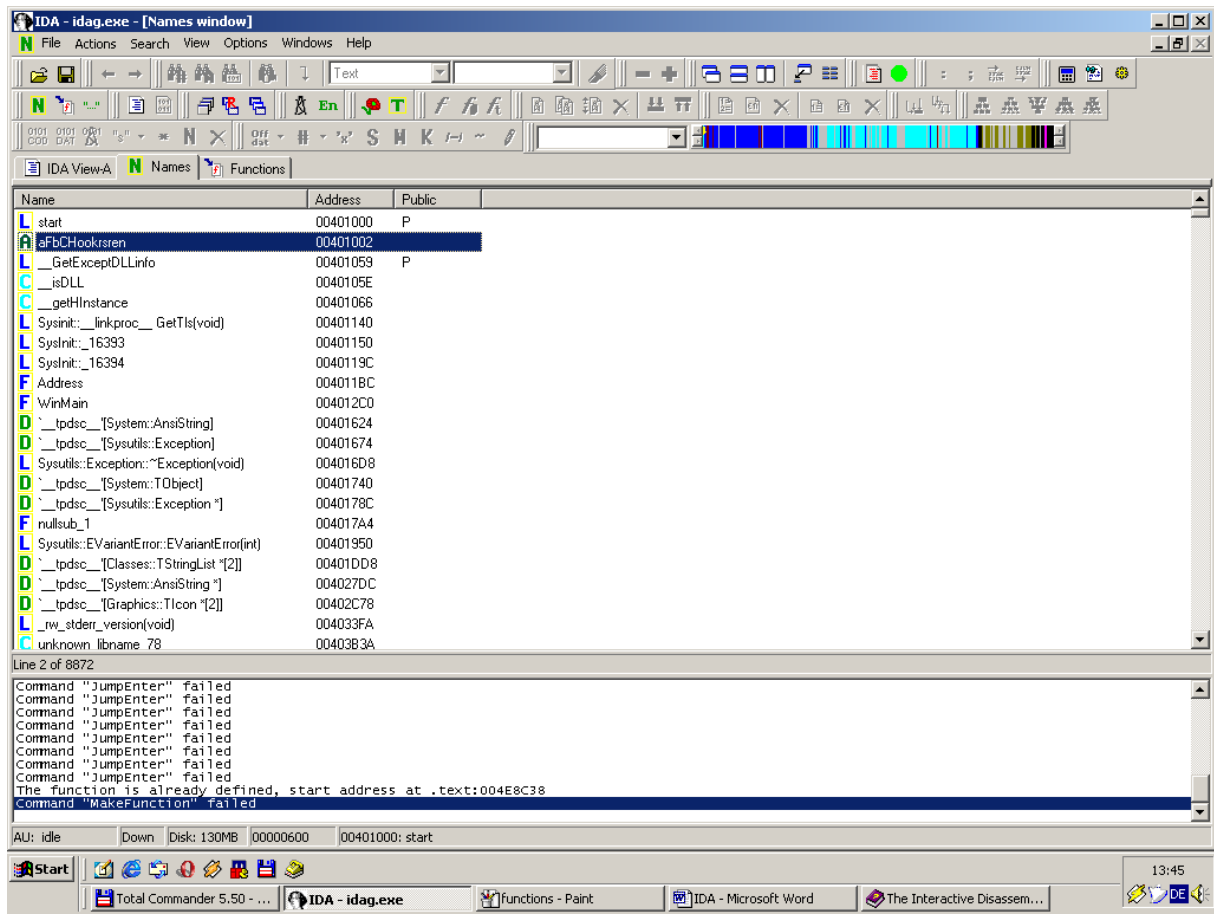


Here we see all functions recognised by IDA. By double-clicking or pressing “Enter” on one of them you will immediately reach this code in the Main View. Another nice feature here is the ability to search. Just enter the string you are searching for and, if it is found, you will reach this name. Also, take a look at the “Action” Menu while working in this window.

Here is a small explanation of the Letters behind the function name:

- R - function returns to the caller
- F - far function
- L - library function
- S - static function
- B - BP based frame. IDA will automatically convert all frame pointer [BP+xxx] operands to stack variables.
- T - function has type information

## Part 12 – Names

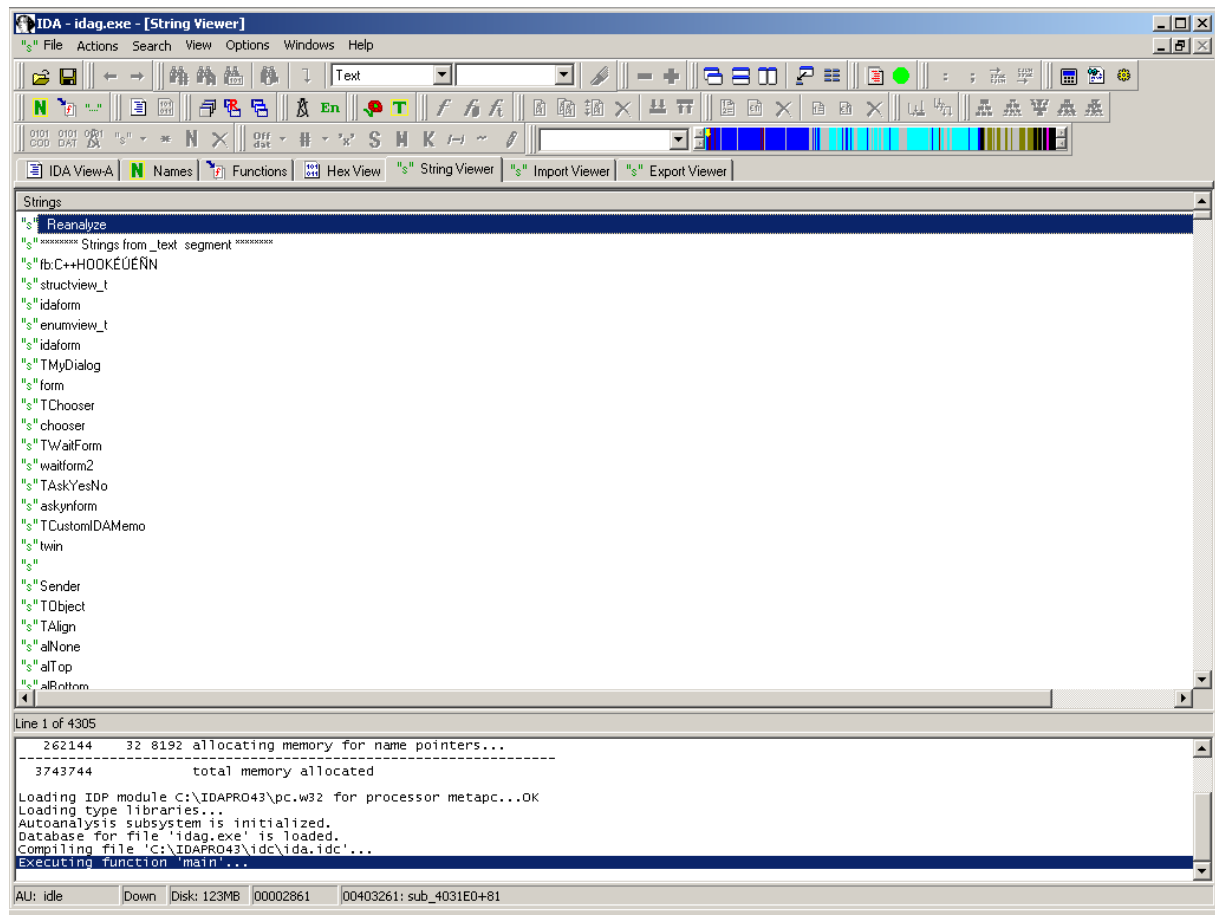


Here IDA displays all found names from your file. In this window you are also able to search for names by just entering your string. Double-clicking or “Enter” will bring you to the location of the name. The small icons in front of the names have the following meaning:

- L (dark blue) - library function
- F (dark blue) - regular function
- C (light blue) - instruction
- A (dark green) - ascii string
- D (light green) - data
- I (purple) - imported name

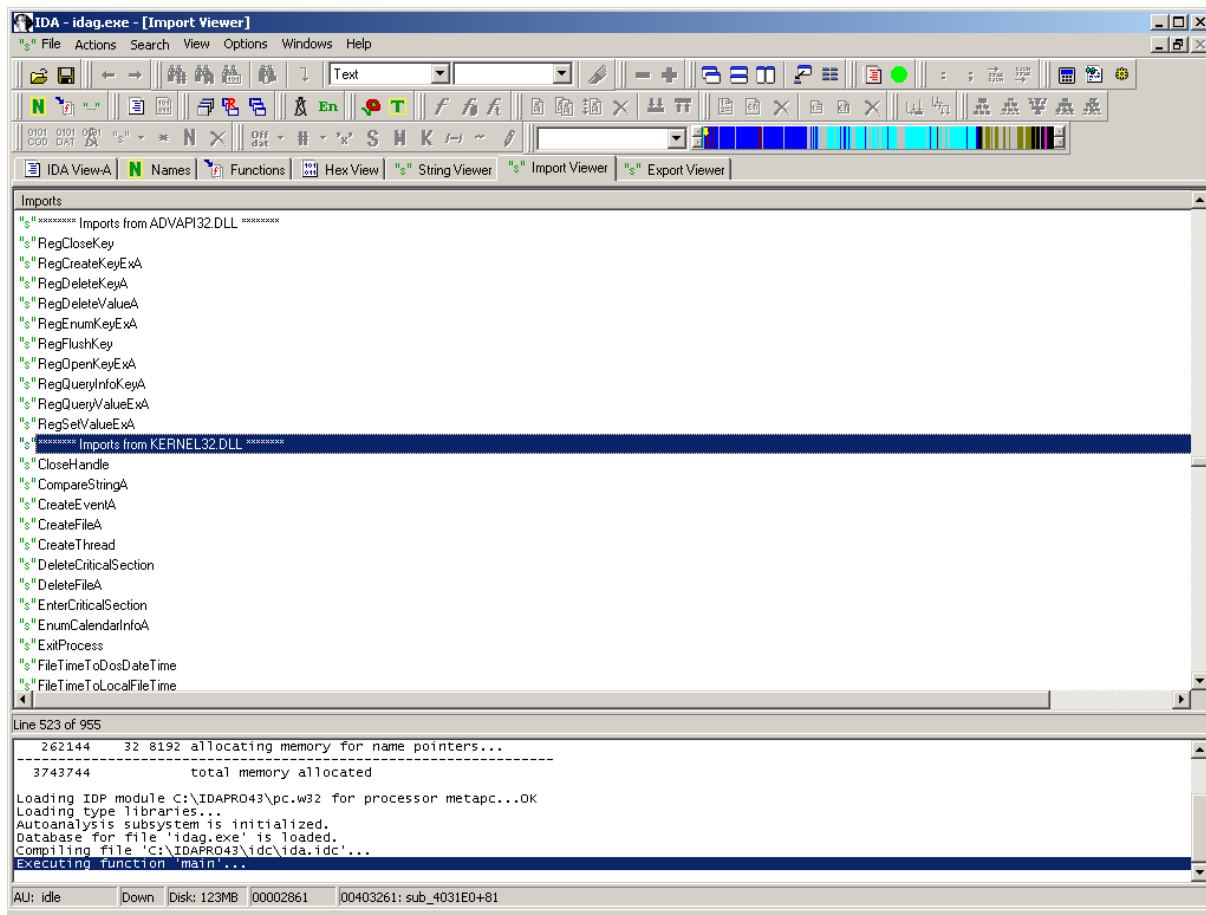
You may also take a look at the “Action” menu.

## Part 13 – String Viewer



This window shows us all found Strings. If you are looking for Strings like “Demo”, “Shareware”, “Trial”, “Invalid registration key” and so on, this window will be your source. You are also able to search for strings by just entering your search string. Double-Clicking on your found String will bring you to the Code Location where the String is used.

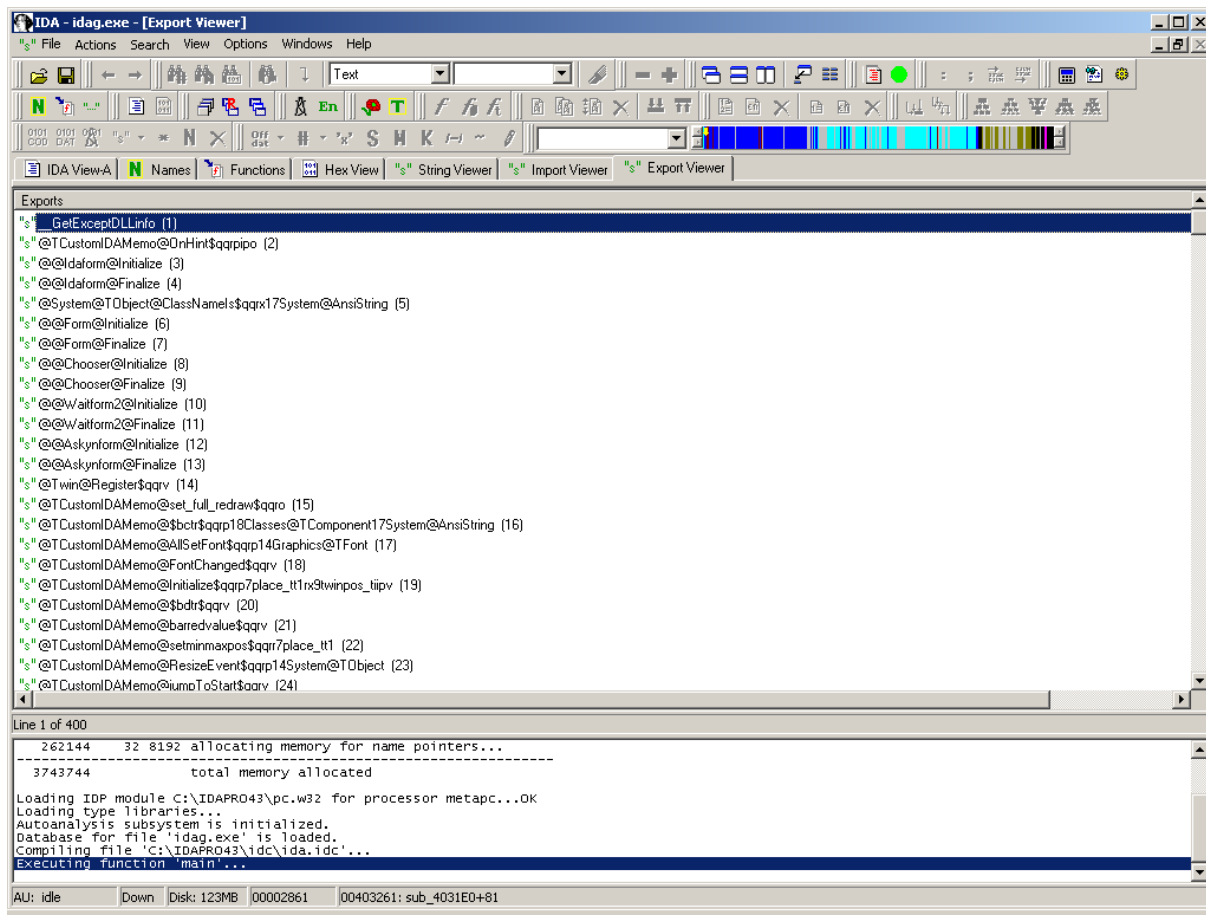
## Part 14 – Import Viewer



This window is very important because here you can see all functions that our program uses from different DLL's (Dynamic Link Library). As you might know, we don't need to rewrite a function that displays, for example, a Window every time. It's the same as if you would rewrite a search or sort algorithm every time you need it. There you write it once and use it later when you need it by just calling your function. Here it's the same. Many function are finished and we just need to use them. The import window tells us which DLL's are used and which functions are called, for example, functions reading and writing to the Registry. This is not rarely used for storing Serial numbers or Registration keys. Reading and writing a file is often used for Key-File routines. Here is where you can collect your ideas to attack the protections or find useful breakpoints for usage in Softice later. As in the other windows, you are also able to search for strings by just entering them.



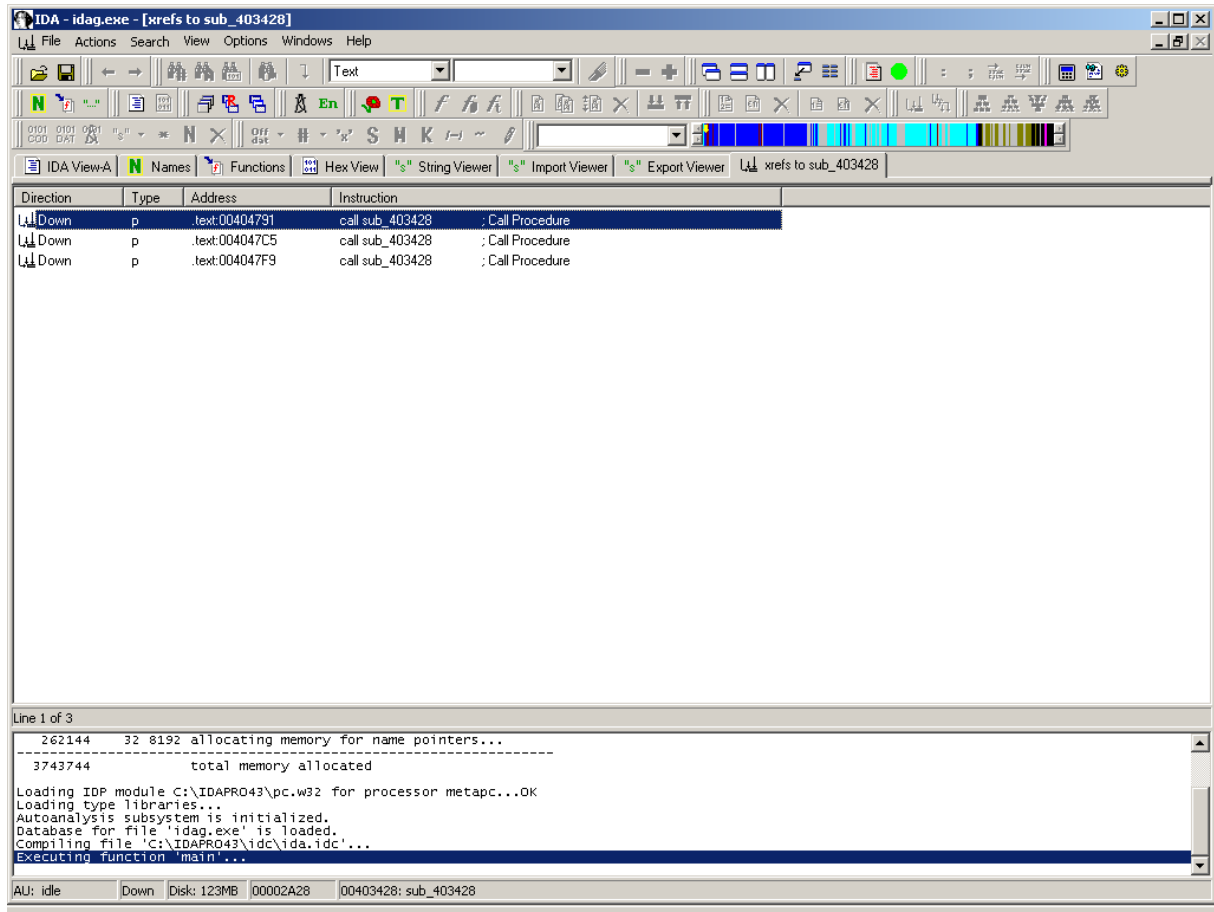
## Part 15 – Export Viewer



This window is very useful when reversing DLL's because it displays all functions that may be called and used by different programs.

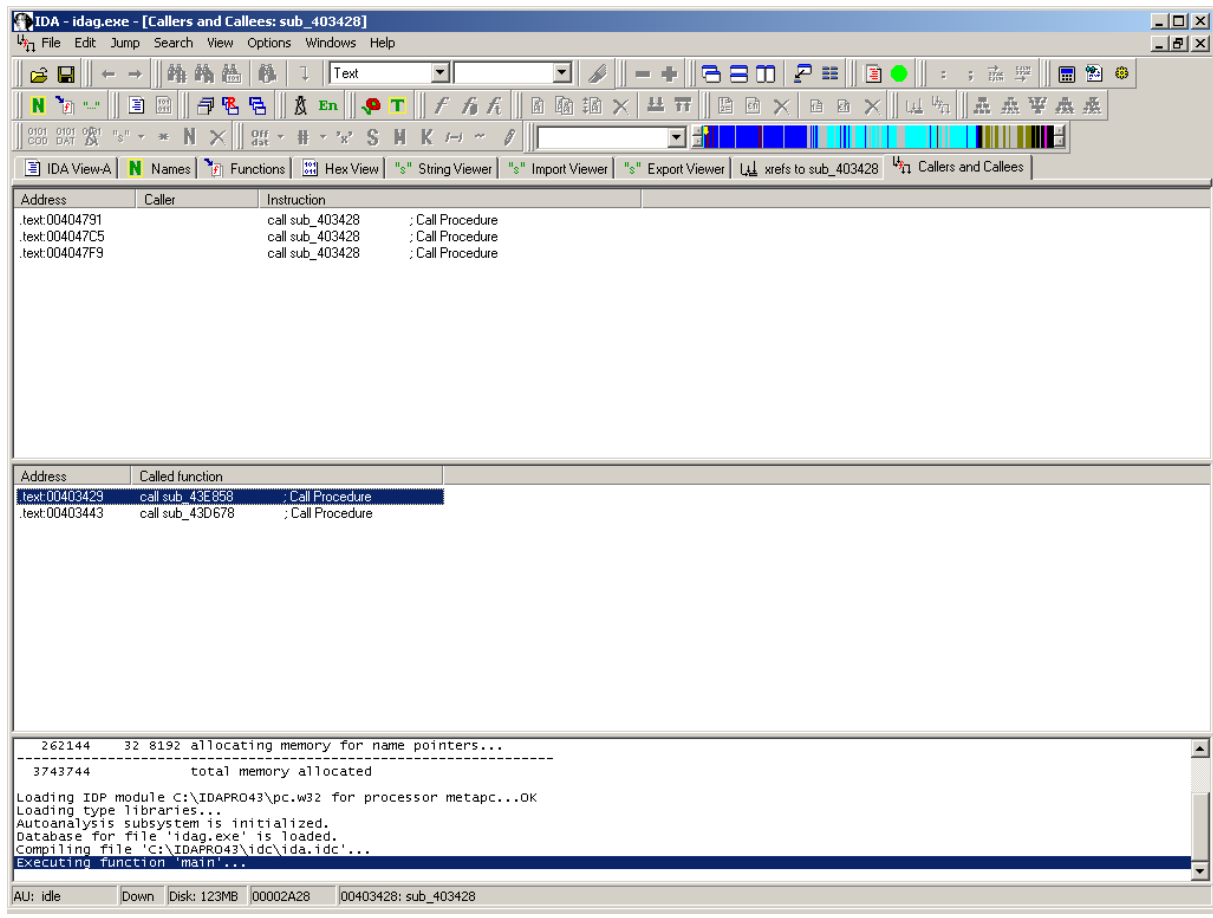
For example, we have a self-written protection scheme from a software company and they provide a DLL managing all Registration functions. You may want to take a look at which functions the DLL provides to get valuable ideas for further proceeding. Again, you are able to search by just entering your strings.

## Part 16 – Cross-references



This window shows all Cross-references of the function in the window we created. That means all Code locations where our function is called. To create this window, place your cursor on the header of the function, select View/Open Subview/Cross-reference and a new window with all Cross-References for this function is created. The name of the window should be “xrefs to ‘function name’”. Normally you see all Cross-references but sometimes, when there are more than 3 or 4 references (maybe 20 or more), creating a Cross-reference window might be useful. Double-Clicking on one of them in the list will bring you to the Code locations.

## Part 17 – Function Calls



This window shows you the Cross-references in the upper part of the window and additionally it also displays all functions called by functions. That is very nice to get a general overview of the function and how many functions you might need to check further.

For example, let's say a function is our Serial Check routine and the first call inside is for converting our Input String to Hex. The second call then checks if our serial is correct. The Cross-references tell you how often and where the Serial Check is performed. To open this window place your cursor on the first line of the function and select:

View/Open Subview/Function Calls

## Chapter 4 – Navigating through the Code

### **Part 18 – The Arrows in front of the Code**

These arrows represent the execution flow, namely the branch and jump instructions. The arrow color can be:

#### Red:

That means the arrow source and destination don't belong to the same function. Usually the branches are within functions and the red color will conspicuously represent branches from or to different functions.

#### Black:

Black is the currently selected arrow. The selection is made by moving to the beginning or the end of the arrow using the Up or Down keys or by left-clicking on the arrow start or the arrow end. The selection is not changed by pressing the PageUp, PageDown, Home, End keys or by using the scrollbar. This allows you to trace the selected arrow far away.

#### Grey:

All other arrows

The arrow thickness can be:

#### Thick:

A backward arrow: Backward arrows usually represent loops. Thick arrows represent the loops in a clear and notable manner.

#### Thin:

Forward arrows.

Finally, the arrows can be solid or dotted. The dotted arrows represent conditional branches where the solid arrows represent unconditional branches.

## Part 19 – Following the Jumps and Calls

I think the best way here is to give a small example. Let's say we are at the following Code Location:

```
.text:0040326F    414 0F 84 9D 00 00 00    jz loc_403312
```

Double-clicking on “loc\_403312” will lead us to the following Code Location at the line 00403312.

```
.text:00403312                                loc_403312:
.text:00403312
.text:00403312    424 8B C3                                mov eax, ebx
.text:00403314    424 81 C4 08 04 00 00    add esp, 408h
.text:0040331A    01C 5F                                pop edi
.text:0040331B    018 5E                                pop esi
.text:0040331C    014 5B                                pop ebx
.text:0040331D    010 C3                                retn
```

Double-clicking on the location at the jump will lead us to the point that the jump would go.

In the same we can use this for following Calls. Check out the following example.

```
.text:00403244    414 8B D7                                mov edx, edi
.text:00403246    414 8B C6                                mov eax, esi
.text:00403248    414 E8 2B B6 03 00    call sub_43E878
.text:0040324D    414 33 C9                                xor ecx, ecx
```

Let's say we are at line 403248 and double-click at “sub\_43E878”. We will land at the following Code location at line 43E878.

```
.text:0043E878                                sub_43E878  proc.text:0043E878
.text:0043E878
.text:0043E878    000 53                                push ebx
.text:0043E879    004 8B D8                                mov ebx, eax
.text:0043E87B    004 56                                push esi
```

Notice: I left out the Comments and Code References for readability.

## **Part 20 – Using the Forward/Backward Arrows**

The third and fourth icons in the Toolbar are usually two arrows, one pointing to the left (Backward Arrow) and the other one to the right (Forward Arrow). These two arrows can be used to move forward and backward in the Code. Take the two examples from Part 19. If we followed the Jump and now want to go back, press the “Backward Arrow”. If you want go to return to the location the Jump led to, press the “Forward Arrow”. Your last moves are stored and may help you navigating through the Code.

## **Part 21 – Using Cross-references**

I think the best way here is to give an example. Take a look at the following Code:

```
.text:004559A9      loc_4559A9:          ; CODE XREF: sub_4557A8+A8 j
.text:004559A9                        ; sub_4557A8+11C j
.text:004559A9                        ; sub_4557A8+1B7 j
.text:00455949
.text:004559A9      0CC 5F              pop edi
.text:004559AA      0C8 5E              pop esi
.text:004559AB      0C4 5B              pop ebx
```

Fine, now let's imagine this location is our badguy-location and we need to check all jumps that lead to this Code location. When we do a double-click on “sub\_455748+A8j” we will reach the first one. By double-clicking on “sub\_4557A8+11Cj” we reach the second and so on. For this example all Code locations would be:

```
.text:00455850      0CC E9 54 01 00 00  jmp loc_4559A9      ; Jump
.text:004558C4      0CC E9 E0 00 00 00  jmp loc_4559A9      ; Jump
.text:0045595F      0CC EB 48           jmp short loc_4559A9 ; Jump
```

Ok, now lets take an example for Code-references leading to a function. Take a look at the following Code:

```
.text:0045565C sub_45565C proc near ; CODE XREF: sub_455464+B1 p
.text:0045565C ; sub_455CAC+44 p
.text:0045565C ; sub_457784+110 p
.text:0045565C
some code
.text:004557A5 sub_45565C endp
```

Lets say this is our Serial-Check function and now we need to check at which locations it is called. By double-clicking on “sub\_455464+B1p” we will reach the first call. By clicking on “sub\_455CAC+44p” the second call and so on. For this example all our Code location would be:

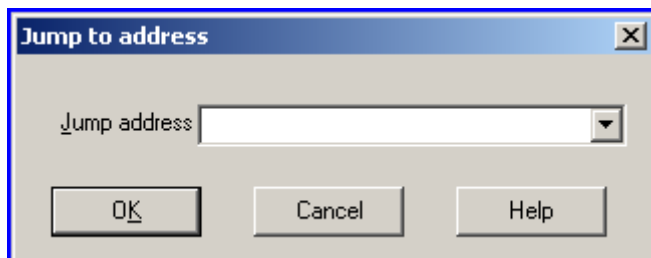
```
.text:00455515 0B0 E8 42 01 00 00 call sub_45565C ; Call Procedure
.text:00455CF0 048 E8 67 F9 FF FF call sub_45565C ; Call Procedure
.text:00457894 064 E8 C3 DD FF FF call sub_45565C ; Call Procedure
```

In fact, it's the same as in Part 19. We just follow things the other way around.

## **Part 22 – The Jump Menu**

The menu explains itself, but I will give two small examples that are useful. Also, take note of the Hotkeys of the other menu entries. They may become very handy.

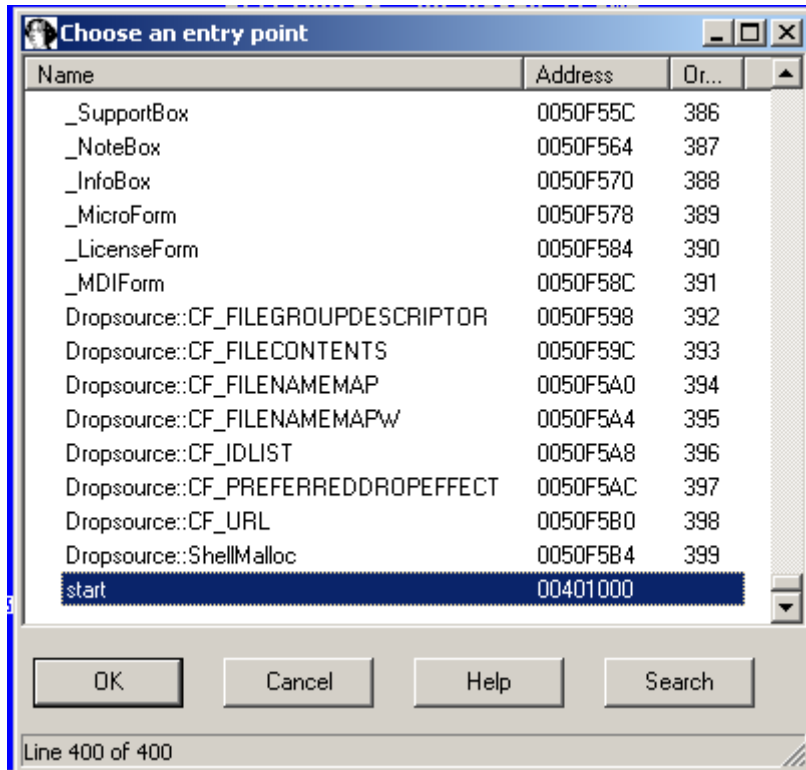
Jump to Address (Hotkey: 'g'):



This is very simple to understand. Just enter the address you want to go to and press “OK”. You also have a history of the last addresses you jumped to.

e.g. *0040BFB2*

Jump to entry point (Hotkey: 'CTRL-E'):



Just select the one you wish from the list and press "OK". The entrypoint "start" is the beginning of the program. That might be very useful if you are looking for Nag-Screens displayed at the program start.

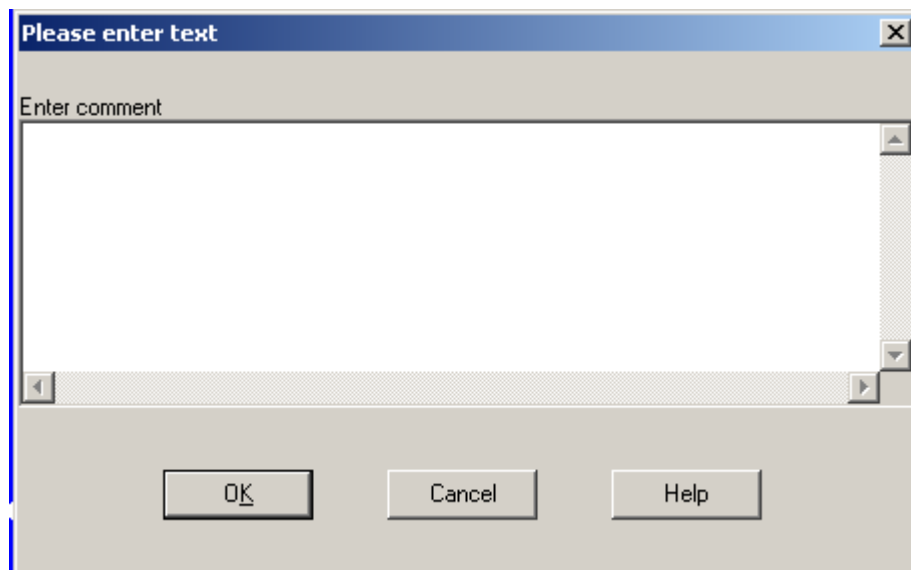


## Chapter 5 – Making the Code more readable

### Part 23 – Adding Comments

It's very useful to comment the code you already worked through and the parts you understood. Otherwise you could step over this code section again and again and lose a lot of time that could be better spent elsewhere. Comments can be added for every Code line. There are 3 possible ways to enter a comment for a code line.

1. Using the Toolbar, there is an icon displaying an “:”
2. Placing your cursor at the end of the code line, pressing the right mouse key and selecting “Enter Comment” at the upcoming menu.
3. Using the Hotkey : “:”



Here you can enter one or several lines for commenting the Code line. Pressing “OK” will add your comment to the Main Window.

You are also able to add “repeatable comments”. They can be accessed by the following 3 ways.

1. Using the Toolbar there is an icon displaying an “;”
2. Placing your cursor at the end of the code line, pressing the right mouse key and selecting “Enter repeatable Comment” at the upcoming menu.
3. Using the Hotkey : “;”

I suggest you try it out yourself. But they are only useful in some cases.

## **Part 24 – Adding Lines**

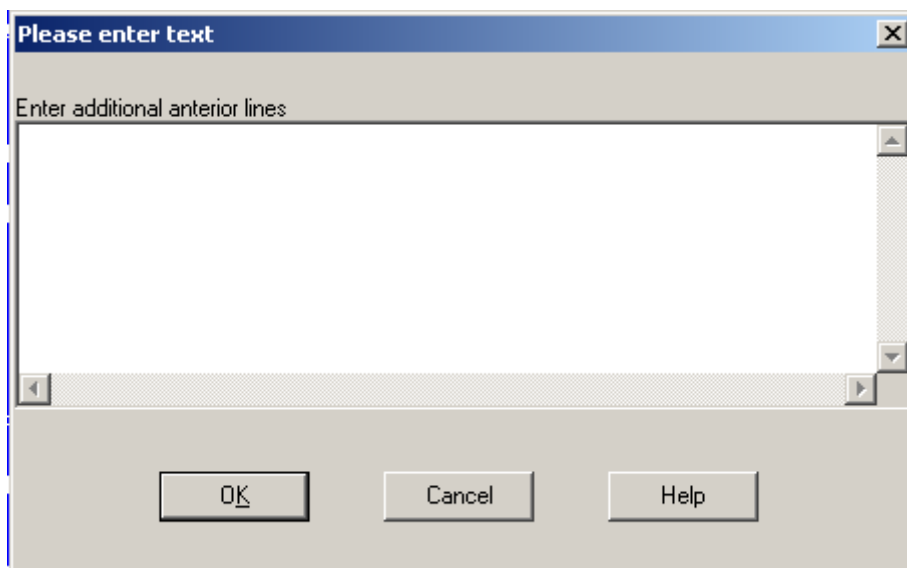
You are able to add Lines with comments between the Code. Here you have 2 possibilities:

1. Creating a Line before your current Code line (Enter additional anterior lines)
2. Creating a Line after your current Coder line (Enter additional posterior lines)

These options can be reached in the following ways:

1. Using the Hotkeys: “INS” or “Shift+INS”
2. Using the 2 Toolbar Icons after the “:” and “;” Icons

The Window for entering your comment lines would look like the following. Depending on which of the two you selected, it will say either “anterior” or “posterior”.

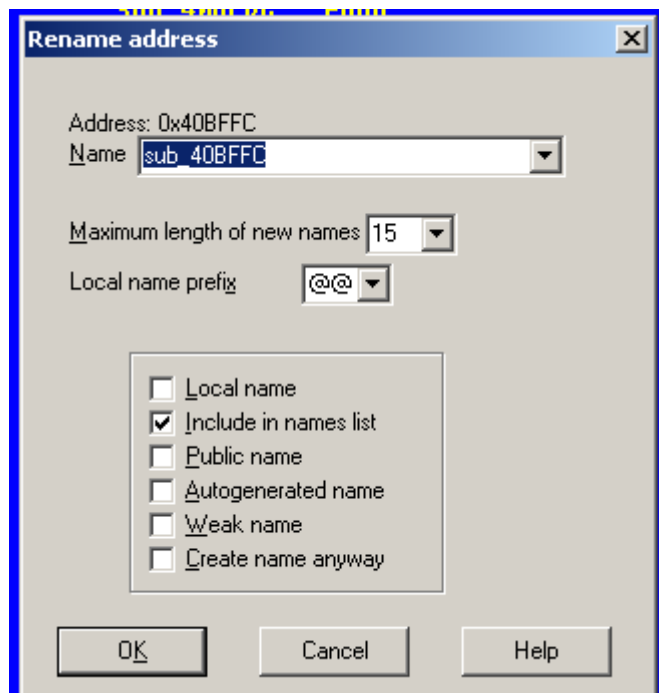


## **Part 25 – Renaming Functions, Locations**

At first you might think Renaming is useless. But it's very helpful once you find out which functions does what or that this location is representing a loop or this variable is used for storing your Serial-Number by just giving some example. It also helps you in reversing your target more effectively.

### Renaming Functions:

Move your cursor on the header of your function. Then right-click with your mouse and select "Rename" in the upcoming menu. A window like the following should pop up:



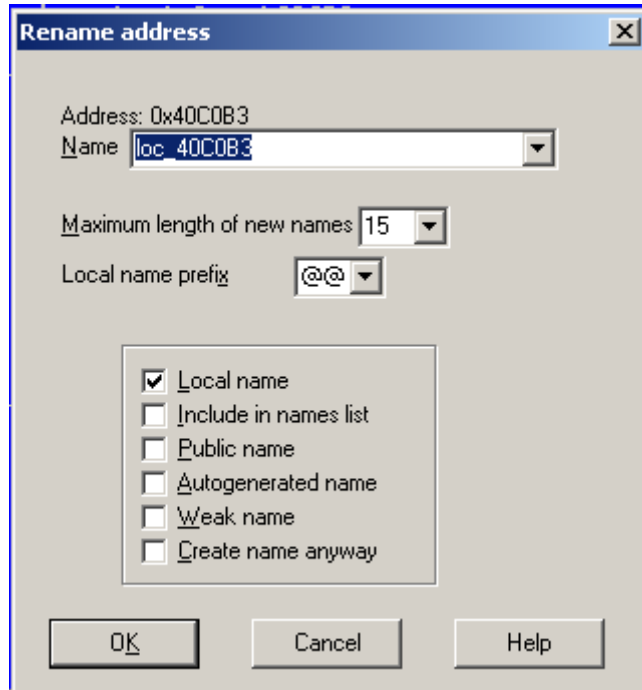
After entering a new name press "OK" and the function is renamed. Now, every representation of the old name is replaced with your new name.

e.g. call sub\_40BFFC to call SerialCheck.

Notice: When you build an NMS file with the Ida2Softice Plugin and you renamed a function this function is displayed in Softice later with your new name.

## Renaming Locations:

Place your cursor on the name of the location. Press the right mouse key and choose "Rename". There will be a window like the following:



Enter a new name for this location and press "OK". Every representation of the old name is replaced with your new name.

e.g. jnz loc\_40C0B3 to jnz myname

You can 'Rename' a lot of things in IDA, not just the two examples I mentioned above, so make sure to try them out. The renaming option is very helpful in making your code more readable.