DR. THORSTEN SCHNEIDER

THE BINARY AUDITOR™

BEGINNERS GUIDE

WWW.BINARY-AUDITING.COM

PUBLISHED BY WWW.BINARY-AUDITING.COM

AN UNIVERSITY EXERCISE COMPENDIUM FOR BINARY AUDITING, BINARY SOFTWARE ENGINEERING AND CODE ARTS

First printing, January 2011

Introduction

Many beginners ask how and where to start. In a time where information is spreaded over the internet and information pieces started to get fragmented and to have a loose coupling the aim of The Binary Auditor™is to offer a comprehensive guide. This guide (and the project) is aimed to help beginners in the field of Binary Auditing and Reverse Code Engineering.

So if you are interested to learn this art I am sure that you know what this all is about. Binary Auditing deals with the analysis of binary files - a world without source code, a world build by compilers and messed up by exploitable code, copy protectionists or virus writers. It is a fun and challenging world and The Binary Auditor™is your sports market.

If you are here just because you want to learn how to crack this place might not be the right choice. The Binary Auditor™tries to build up a solid background knowledge for university students and other interested folks. As university lecturer I have spended lot of time solving the "Table of Contents" problem. Where shall we start? And can we offer a full practical course without spending too much time with theory? Is it possible without learning theory first? And if we need theory, which is necessary and how much? Additionally I had the problem with the programming languages. Can you learn Binary Auditing without being a C++ coder? In my opinion: yes. Most of my students joined my Java lecture, a programming language which is quite different from what we will do during The Binary Auditor™. You can even start with all this without having assembly language knowledge? Again: yes, but it will hurt. If you want to enter this fascinating world you have to know that you have to invest lot of your free time.

Understanding binary code is like learning how to ride a bicycle. You see people riding a bike and you are fascinated. So you buy a bike, take your seat.. and you will crash. After several attempts you are next to give up. But you have a friend who recommends a good book for this. You read this book, you take your seat... and you crash. As a result you realize that just watching others (like watching YouTube videos) does not help at all, and reading (books or tutorials) is a nice lecture. But when it comes to a problem slightly different from video or tutorial you will be lost. Finally you realize that you can learn only by doing, so you take your seat again and again... and suddenly things get better, the first curve can be taken even you are not a trial bike or downhill rider yet. But if you train hard enough and you invest your time you will get a perfect biker.

Do not get me wrong. But current young people are YouTube nerds thinking just watching something brings them enlightenment. Forget about videos, DO IT! If you are one of the lazy guys who are from the generation "feed me" then close this document and drop the idea to learn this art.

Students in programming courses can be categorized as effective and ineffective based on their effectiveness in programming. Effective students can write programs and they typically learn programming with moderate effort. Whereas, ineffective students cannot write correct programs and need more personal attention and cognitive support to learn programming. Since the failure rate is high, the ineffective category plays a significant role in the effectiveness of programming courses.

The results of psychological studies in computer programming expertise show that turning a novice into an expert is impossible in a four year undergraduate programme, but competence is possible by practice. Thus lab sessions play a significant role because of the importance of practice in learning programming. But in lab sessions also, some teachers will start giving large programs as assignments to novices rather than starting from small and simple programs. This non-systematic teaching will increase complexity in students' mind, since brain has a natural tendency to learn in an incremental way. In addition to that, novices may not get sufficient individual-feedback during lab session. During the programming task learners receive relatively high levels of feedback on low level issues, such as syntax rules, but tend to receive low levels of feedback on conceptually more difficult issues.

As a cognitive trainer I try to enhance students ability to learn. Typical students have cognitive problems at various levels. Some of them have problems with the very basic cognitive skill called memorization which is essential for knowledge acquisition. Some others have listed cognitive difficulties to comprehend a given problem and a given program, which needs comprehension and analysis skills. Yet another group finds it difficult to apply ('application' cognitive level) the concepts learned to solve problems. Majority finds difficulty to arrive a correct logic, to integrate modules to a working program and difficulty in algorithm design. These are synthesis problems. It is obvious that students also have difficulties in evaluation, which is the highest cognition level. The symptoms for these are their difficulties to justify, defend and describe a program logic. Therefore it is observed that, programming students have serious difficulties in all the levels of Bloom's taxonomy in cognitive domain, independent of the programming language they have used.

Since I am not able to use cognitive training methods during this "internet homework exercise course" I will try to find a special style to train you task driven. This includes special method I have developed such as Cognitive Debugging and Speed Debugging[1]. By the way: The Binary Auditor™is the base for my own university lecture "Ethical Hacking - Binary Auditing and Reverse Code Engineering". It is now your choice: take the blue or the red pill.

[1] You can read a summary of this method at http://www.cognitive-debugging.com.

First steps

So let us start with all this. What do you need to learn Binary Auditing? First you should be able to tolerate lot of frustration. You need some brain and most of your free time. Some ability to think logical is absolutely necessary. By the way, if you have never programmed before, please learn some programming first!
Let us setup our working environment. We need a PC with Windows (later we have a look at Linux and Mac OS X as well). At the moment any Windows above XP is sufficient, so move on with XP, Vista or Windows 7. It is very important that you install a 32 bit system, 64 bit will not work since our debugger is for 32 bit only! I highly recommend to install the operating system within a virtual machine but a standard installed system is fine. We do not deal with evil software pieces killing our working environment. Even when we move to malware analysis (later, later) I can assure that your machine does not get infected!
Next step is to download our weapon. I have included within the training package a binary debugger which is free to use. It is IDA Pro Free version 5.0. The lack of this version is that it can only analyze 32 bit applications. Now you can see why we need a 32 bit system. But trust me, it is good enough for us at the moment. IDA Pro is a perfect tool for our course, it can load and analyze binary files and even display it in a very cool graphical display (you will see this later). Best is, that we can "debug" the application (target) which means that we can step command by command through the target application and we can see what it is really doing. By the way, IDA Pro is not really a debugger but a database. But at the moment let us call it debugger.
Now we need enough material to get trained. The training package includes all we need and I am sure that you have downloaded and extracted the package. Inside the training package you can see various folders:
`exercises/001 - c++ fundamentals`
and
`exercises/002 - assembly language fundamentals`.
is meant for those students who still have to learn C++ and assembly language. The folders contain exercise workbooks and if your C++ is very very rusty you should really do them!

Let us start!

We will jump directly into the world of Binary Auditing. Let us examine folder `exercises/003 - hll pattern mapping`. If you have not installed IDA Pro you should do now! Now go and start IDA Pro. You will get the following screen:

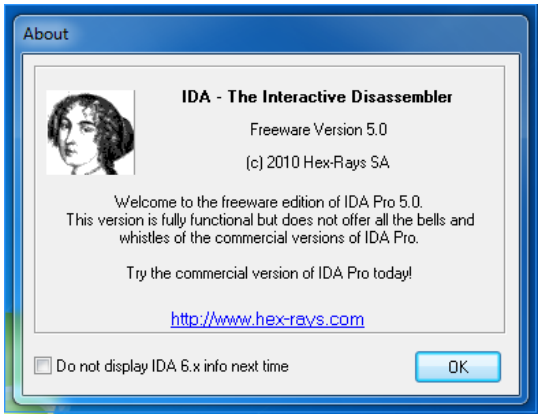

Figure 1: Welcome screen of IDA Pro Free 5.0. Note that we can use the checkbox to disable the IDA 6.x info next time.

The difference between the free IDA Pro and the commercial one is that we do not have any support for 64 bit Windows, Linux, Mac OSX and other platforms. Anyway this is fine for us, we just need to work with 32 bit and do not need other platforms for now.
After clicking the OK button we see the start screen of IDA Pro. Two windows get opened. The front window is to disassemble a new file or to open a recent project. The help box is new with IDA 5.0 Freeware and can be helpful if you get stuck.

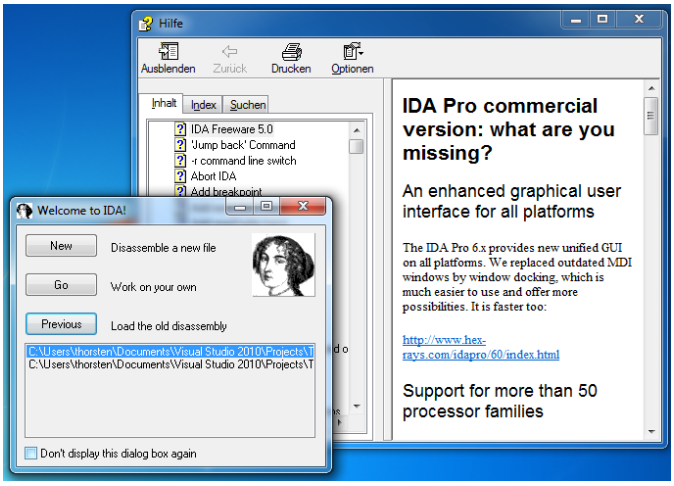

Figure 2: IDA Pro showing help system and asks which file to open. You can see that I had opened 2 files before. For you this might be empty at the moment.

Let us start with the first file we would like to analyze. After clicking the "New" button you will get a dialog asking what kind of file you would like to analyze. Most important for us are **PE Executable** and **PE Dynamic Library**[2]. Just click at **PE Executable**, then click at the **OK** button.

[2] PE Dynamic Library is nothing else than DLL.

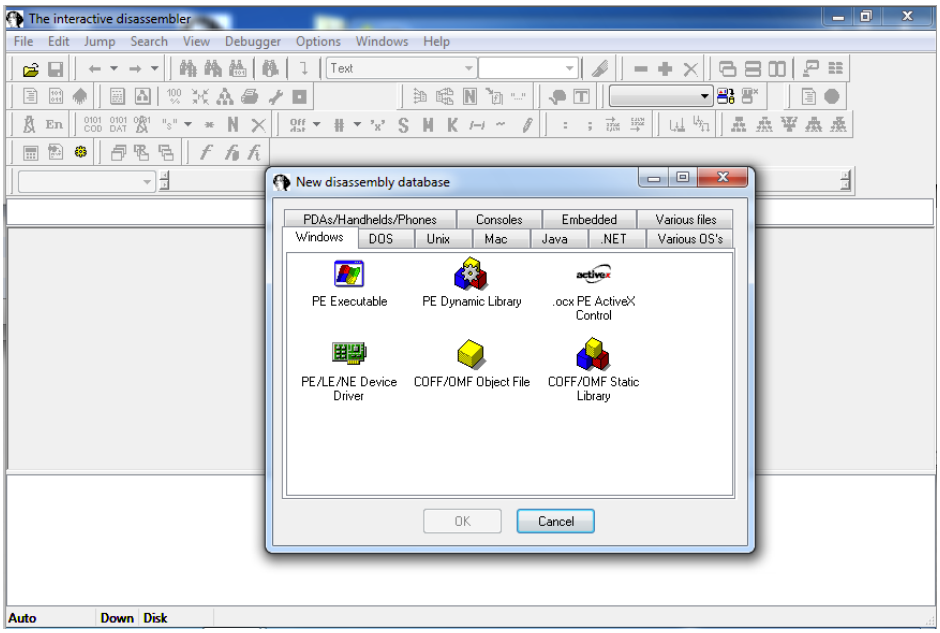

Figure 3: IDA Pro asking what kind of file we would like to open.

All you have to do now is to navigate to the folder where you have extracted the training package. Move to the folder `exercises/003 - hll pattern mapping`. Then move to the folder `Part 1 - Common Code`.
Select the file `a01_identify_variables.exe` and hit the Open button.

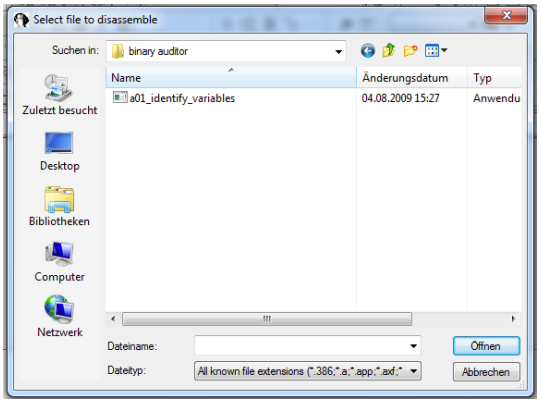

Figure 4: Open a specific file with IDA Pro. Note that my folder is quite empty because I just have copied this exercise file to this folder.

Now IDA Pro opens another window. There are many options we can set when we need to do more complex analysis. At the moment we just accept what we see. Our file is a Portable Executable file (your .exe file) so we just click at OK. Later, when you have played enough with IDA Pro you can open recents projects via the shown menu. Otherwise, IDA Pro saves the analysis database in the same folder where the file is located and your analysis is saved if you like to pause your analysis.

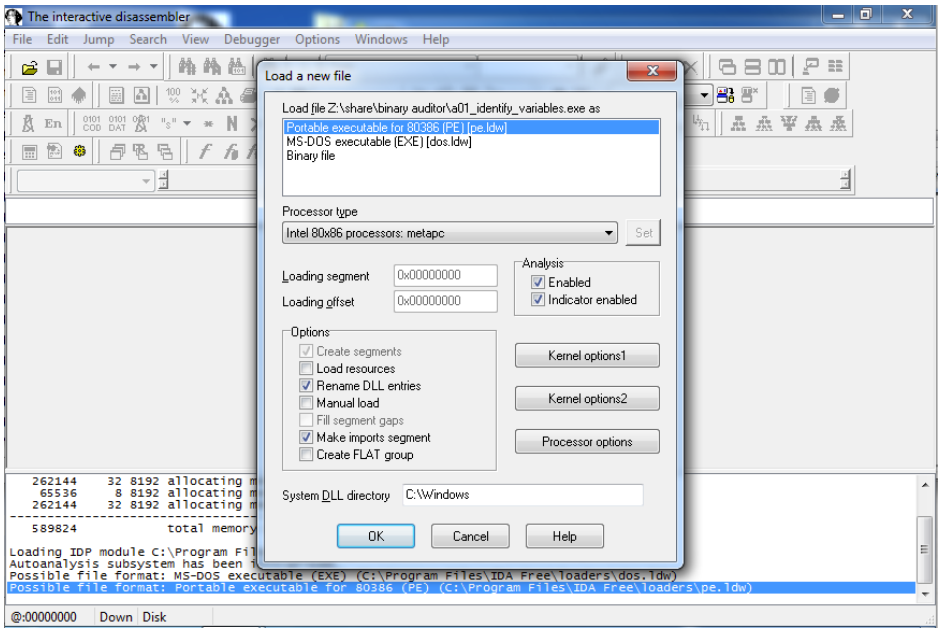

Figure 5: If you are interested to know more about the specific settings I highly recommend "The IDA Pro Book".

Now IDA Pro gives us some important information. It seems that the input file was linked with debug information. You might be surprised but you will find many files with this since many developers forgot about compiler settings. In our case this is not a mistake, it was intended by me to simplify your analysis. Let us load the corresponding PDB file from the server and we click as Yes.

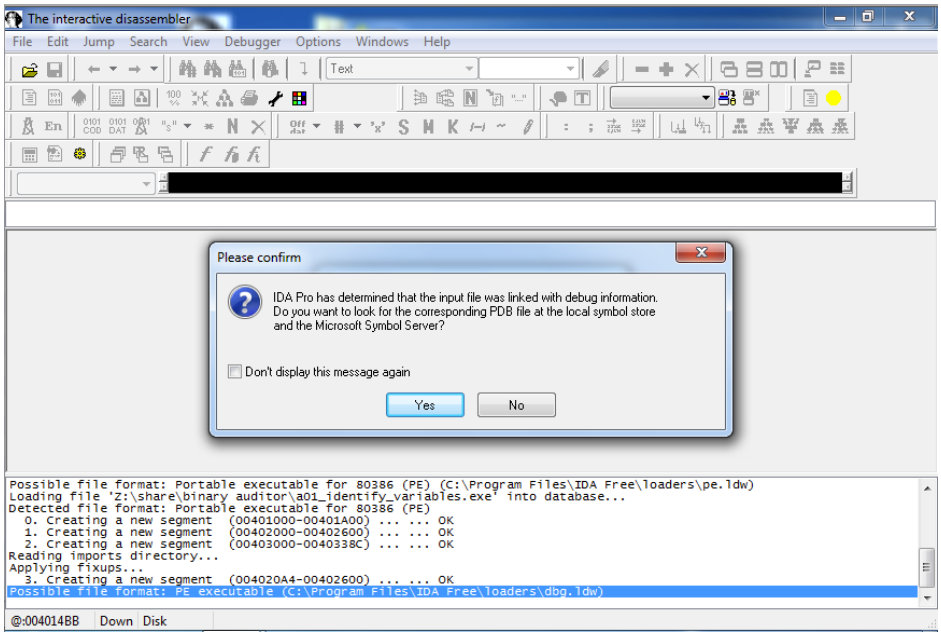

Figure 6: IDA Pro has detected that we can use debug information included.

IDA Pro now opens the file and analyzes it. For this small file the analysis is very fast. After analysis we see that IDA Pro opened many windows. Later you might set your own settings but for me this is too messy. All these icons do disturb me at smaller screen resolutions so let us disable all these nifty buttons below the top menu.

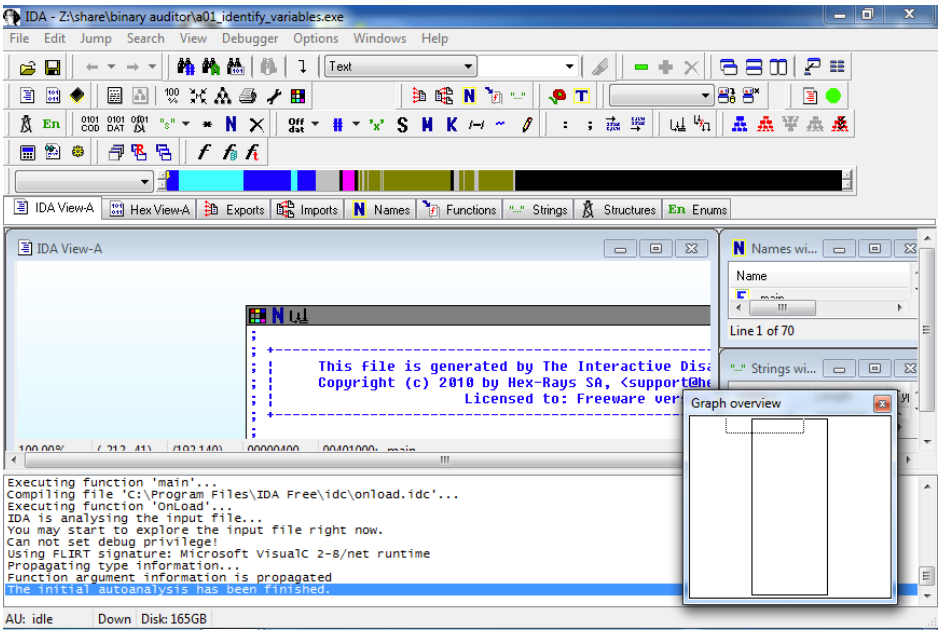

Figure 7: Note that the screenshot has been done with only few pixels in size. Normally I do debugging at a 30" display with 2500 pixel width. But any resolution above 1024 pixel should be fine.

Do a right click somewhere at this buttons bars. Here you can see that really everything has been enabled. We will change this to get more place at the desktop. For this just disable the menu entry **Main**.

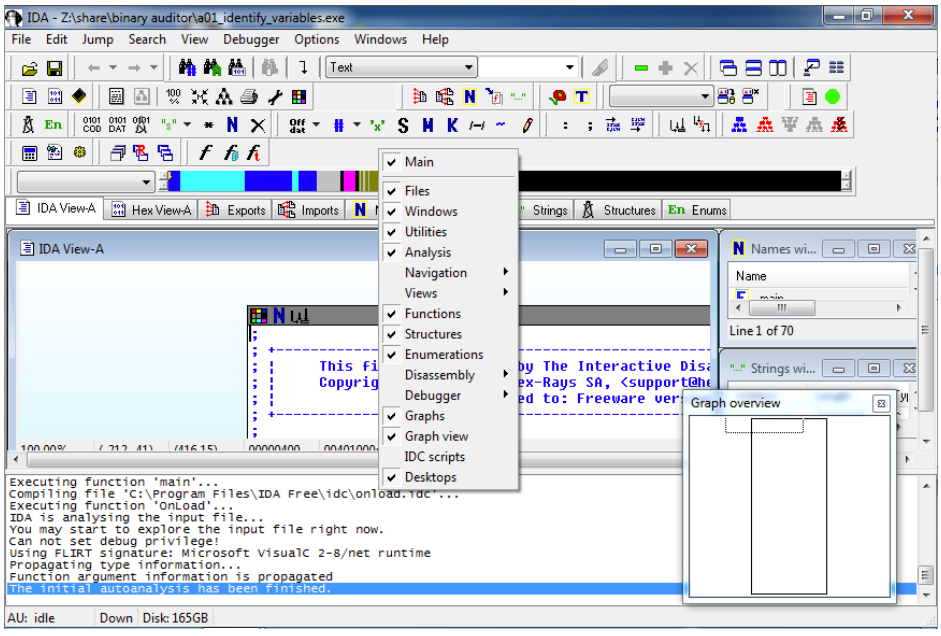

Figure 8: The right click menu shows us what we can view or not.

You will see now the following screen. Note that this is just my preference. Do this as you like but we do not need these buttons at the moment, really!.

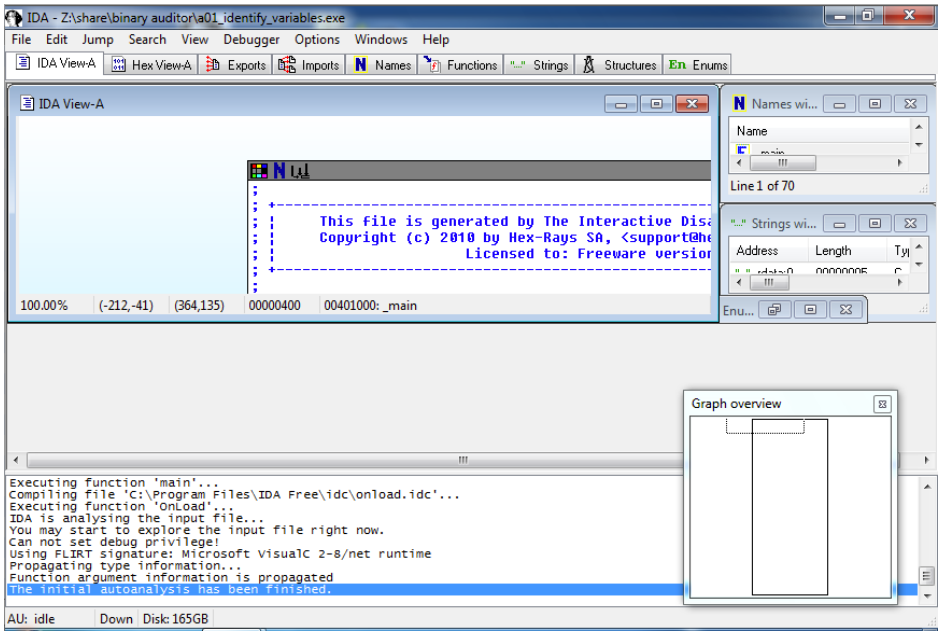

Figure 9: Better without nifty buttons and more place for the important things.

This looks still a little bit messy, so let us resize the graphical window. It has the title **IDA View-A**. After resizing you get the following screen.

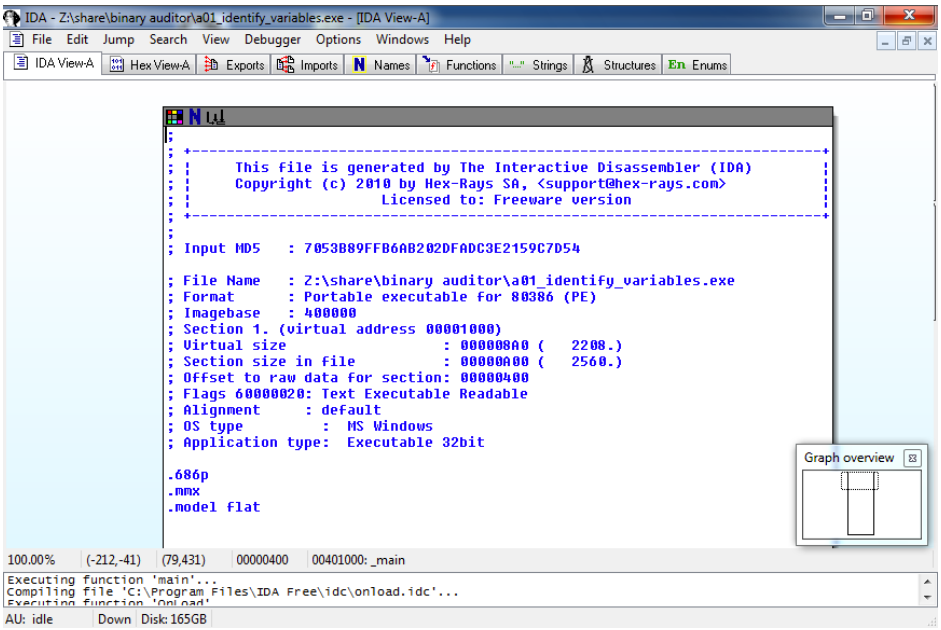

Figure 10: With the graphical view it makes much more fun to analyze and to debug applications.

The little small window at bottom right is the graph overview. You will find it handy when it comes to more complex code. At the moment it looks quite empty but this is OK. Now in full screen you can see the IDA Pro analysis of our example exercise. Using your mouse you can pan the window to any direction you like. I will leave now the screen shots and copy the code directly to this tutorial.

First exercise

The exercise is to identify variables. Let us have a look at the begin of the main function of the application. You will see the following code:

```
1  ; int __cdecl main(int argc,const char **argv,const char *envp)
2  _main proc near
3
4  var_38= qword ptr -38h
5  var_2C= dword ptr -2Ch
6  var_25= byte ptr -25h
7  var_24= dword ptr -24h
8  var_20= word ptr -20h
9  var_1C= dword ptr -1Ch
10 var_18= word ptr -18h
11 var_14= word ptr -14h
12 var_F= byte ptr -0Fh
13 var_E= byte ptr -0Eh
14 var_D= byte ptr -0Dh
15 var_C= dword ptr -0Ch
16 var_8= qword ptr -8
17 argc= dword ptr  8
18 argv= dword ptr  0Ch
19 envp= dword ptr  10h
```

IDA has done a good job for us. A line 1 we can see the function signature, IDA Pro detected the main function and tells us what parameters are expected[3]. From line 4 to 19 we see that there have been defined many many variables. We do not know their real names and understanding what they mean will be our job. But what we can see is what kind of variables we have. Some are qword, some are dword and some do have the type byte[4].

[3] Note that this application does not need any parameters

[4] **Exercise:** It is now your job to find out what qword, dword and byte is!

Let us check what the real code gives:

```
1  push    ebp
2  mov     ebp, esp
3  sub     esp, 3Ch
4  mov     [ebp+var_D], 0
5  mov     [ebp+var_D], 0FFh
6  mov     [ebp+var_25], 80h
7  mov     [ebp+var_25], 7Fh
8  xor     eax, eax
9  mov     [ebp+var_18], ax
10 mov     ecx, 0FFFFh
11 mov     [ebp+var_18], cx
12 mov     edx, 0FFFF8000h
13 mov     [ebp+var_14], dx
14 mov     eax, 7FFFh
15 mov     [ebp+var_14], ax
16 mov     [ebp+var_1C], 0
17 mov     [ebp+var_1C], 0FFFFFFFFh
18 mov     [ebp+var_1C], 80000000h
19 mov     [ebp+var_1C], 7FFFFFFFh
20 mov     [ebp+var_24], 0
21 mov     [ebp+var_24], 0FFFFFFFFh
22 mov     [ebp+var_2C], 80000000h
23 mov     [ebp+var_2C], 7FFFFFFFh
24 mov     [ebp+var_F], 1
25 mov     [ebp+var_E], 0
26 fld     ds:flt_4020F0
27 fstp    [ebp+var_C]
28 fld     ds:dbl_4020E8
29 fstp    [ebp+var_8]
30 fld     ds:dbl_4020E8
31 fstp    [ebp+var_38]
32 mov     ecx, 41h
33 mov     [ebp+var_20], cx
34 xor     eax, eax
35 mov     esp, ebp
36 pop     ebp
37 retn
```

Have a look at lines 1 to 3. Strange things happen here. A register EBP gets placed on the stack, then we copy some registers at line 2 and subtract the magic value 3*Ch* from ESP.
Just three lines but this will be your first true and big exercise! To understand what is happening here we have to check the Intel manuals. The document we need is "Intel 64 and IA-32 Architectures Software Developers Manual Volume 1 Basic Architecture". Move to chapter 6 and read it carefully! This is a very important step, do not override it. The first 3 lines are responsible to setup the stack frame and you **have** to understand the stack!

Now have a look at the lines 34 to 36. Those lines are again for the stack. Lines 1 to 3 build up the stack, lines 34 to 36 clean the stack. If you have read the Intel manual carefully you should be now able to explain line 37 - the *retn* command. If you can not explain this command go and check the manuals again!
Lines 4 to 33 seem to contain our valid code. This example is an easy one and you see just a sequence of commands. Later you will meet more complex code with branches inside and you will reach the point where you can not understand the code without debugging it. But first go and try to understand the code[5]. To make things easier for you I will provide the source code of this example. Can you figure out which lines of the C++ code respond to the lines in IDA Pro? Remember that it is very important that you are able to identify variables within disassembly, else you will fail even when analyzing easy targets.

[5] **Exercise:** have a look at lines 26 to 31. These commands are doing something "different". Check the Intel manuals for these commands and answer the question what they are doing!

Listing 1: A01 - Variables

```
1  int main(int argc, char* argv[])
2  {
3     unsigned char myChar; // 1 byte
4     myChar = 0;
5     myChar = 255;
6
7     signed char mySignedChar; // 2 bytes
8     mySignedChar = -128;
9     mySignedChar = 127;
10
11    unsigned short int myShort;
12    myShort = 0; myShort = 65535;
13
14    signed short int mySignedShort;
15    mySignedShort = -32768;
16    mySignedShort = 32767;
17
18    unsigned int myInt;  // 4 bytes
19    myInt = 0;
20    myInt = 4294967295 ;
21
22    signed int mySignedInt; // 4 bytes
23    myInt = -2147483648;
24    myInt = 2147483647;
25
26    unsigned long int myLong; // 4 bytes
27    myLong=0;
28    myLong=4294967295;
29
30    signed long int mySignedLong; // 4 bytes
31    mySignedLong=-2147483648;
32    mySignedLong=2147483647;
```

Listing 2: Continued A01 - Variables

```
33
34      bool myTrue; myTrue = true; // 1 byte
35      bool myFalse; myFalse = false; // 1 byte
36
37      float myFloat; myFloat = 5.3431243774; // 4 bytes
38
39      double myDouble; myDouble = 5.3431243774; // 8 bytes
40
41      long double myLongDouble;
42      myLongDouble = 5.3431243774; // 8 bytes
43
44      wchar_t myWChar; myWChar = 'A'; //2 or 4 bytes
45
46      return 0;
47  }
```

Let us move one more step beyond this dead code analysis. Maybe it would be easier to step through the running code line by line. Mark line 1 with your mouse, right click. In the context menu you will see a menu entry named "Add Breakpoint F2". Click it. Line 1 get now colorized with red which means that when you debug the application it will stop immediately at this point.
Now have a look at the top menu. There you will find "Debugger". Click it. Then click at "Start the process".

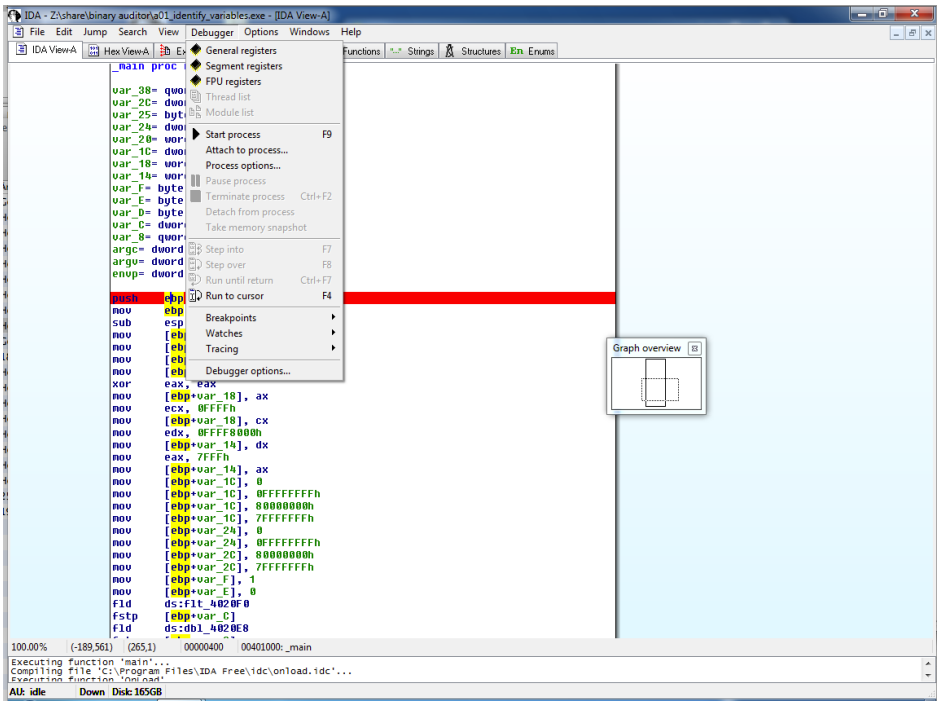

Figure 11: Debugger menu. Important is the F9 key to start the process.

The following warning is for you to prevent running a malware target without intention. We do not have any problems with this application so there is no problem for us to click at Yes.

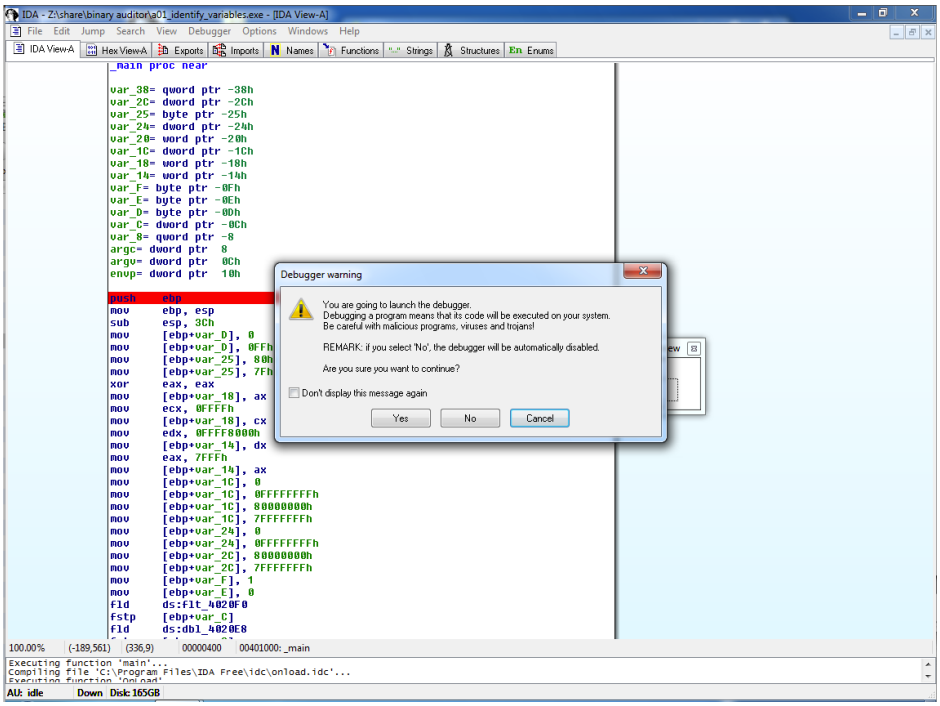

Figure 12: IDA Pro warning. You can disable this "nag" if you like and click the checkbox. Doing this just disables the warning for the current project and is not a global setting.

You will notice that the complete screen is doing now weird things. Many new windows appear, the old ones disappear. The big black window is our application which is correct since we have to analyze a console application.

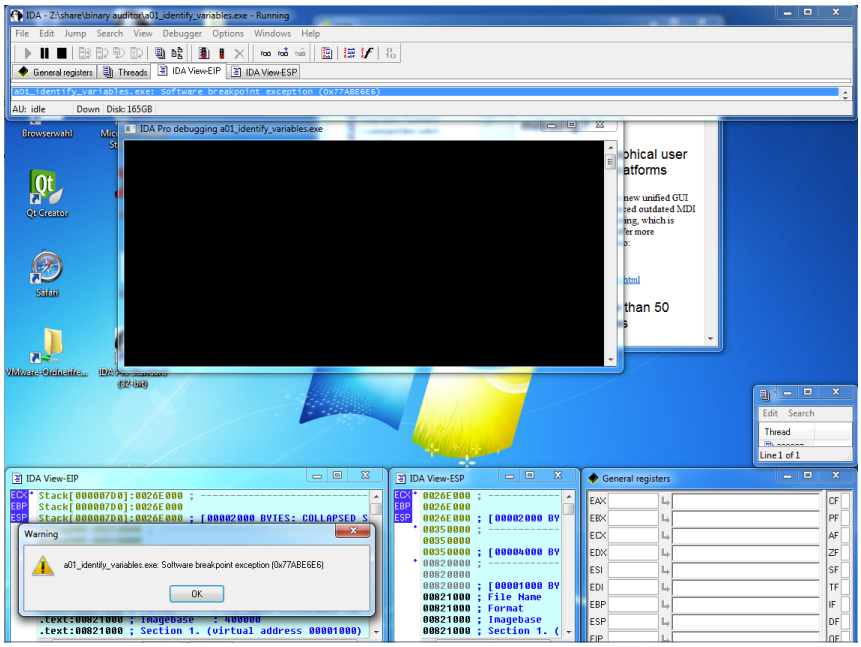

Figure 13: The IDA Pro debug view. Many windows but you will need them for sure.

Note: In my case I can see a warning at bottom left. I just click it away. The debugger stops at some strange line (see the window "IDA View-EIP" at left bottom). You can see that we have stopped somewhere inside ntdll.dll which is definitely not our target. Just go to the menu "Debugger" and click "Continue Debugging". Anyway, after continuing the debugging process I get nagged with a warning:

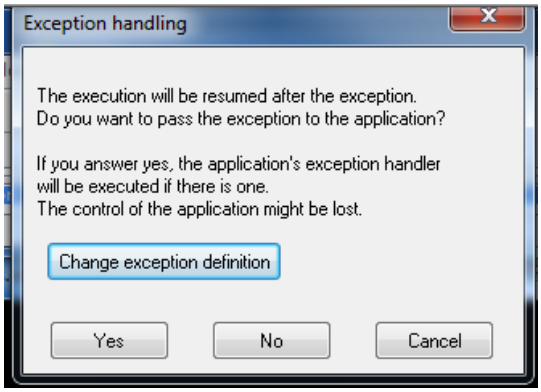

Figure 14: IDA Pro bugs us with an exception. There are some options to fix this, one will be to change the exception definition.

Just click at Yes and you can continue the debugging process.

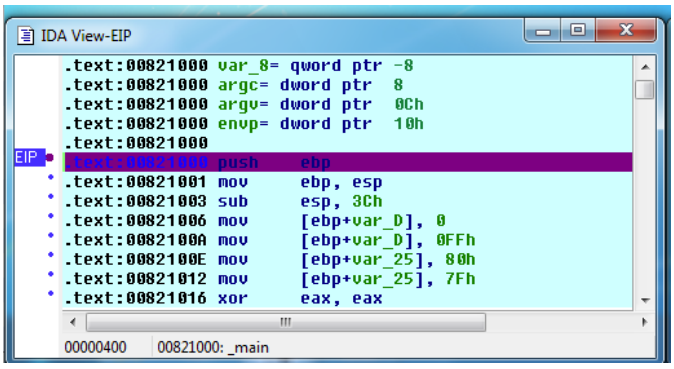

Figure 15: IDA Pro showing the correct debugging position.

"IDA View-EIP" is now showing the correct position of our breakpoint and marked it with purple. This means that the executable has been stopped at this position. Note that the purple line mean, that this line has not been execute yet!

Go and resize this window to full screen.

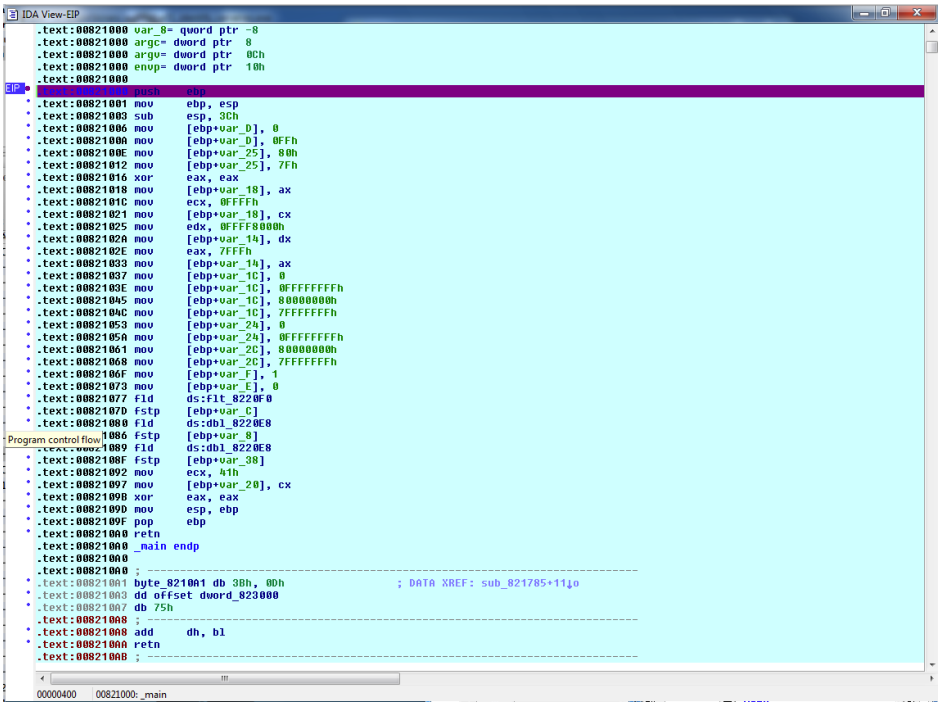

Figure 16: IDA Pro shows our disassembly this time in the debugger window.

This looks similar to the dead code analysis we have done before but this time we can step line by line through the running code. Now we will do some cool magic which you will really love when you start to analyze more complex targets. Just mark some line in the code and press this long bar at your keyboard (hint: some call it "space"). Suddenly the layout looks different. You can see the graph overview mini view again and the disassembly is placed in a window.

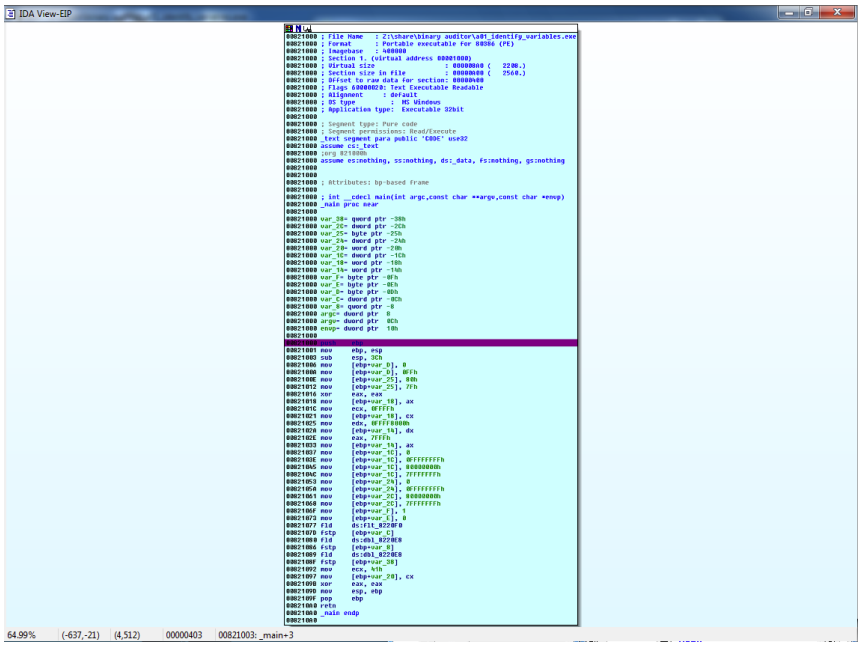

Figure 17: IDA Pro shows our disassembly this time as graph view.

Note that this does not look impressive yet since we do not have anything sophisticated to show, but latest at our exercises with loops and branches you will understand why the graph layout really rocks.

Resize the window now to that size you like and that you can see the menus again. At left you can see our code window. Right top shows "IDA View-ESP". This window is very important for you at this moment and of course later as well. It shows your stack with all necessary information. Do you remember when we talked about setting up the stack and cleaning it? This is now your chance to watch how the stack will be setup when an executable starts! Do the following analysis again and again until you have understood 150% how this works! The window "General registers" at bottom right shows you which registers have which values stored. Check them when we debug through the code.
Note: do this again and again. Watch the stack window and the registers window. Do it slowly and step through it line by line - slow!

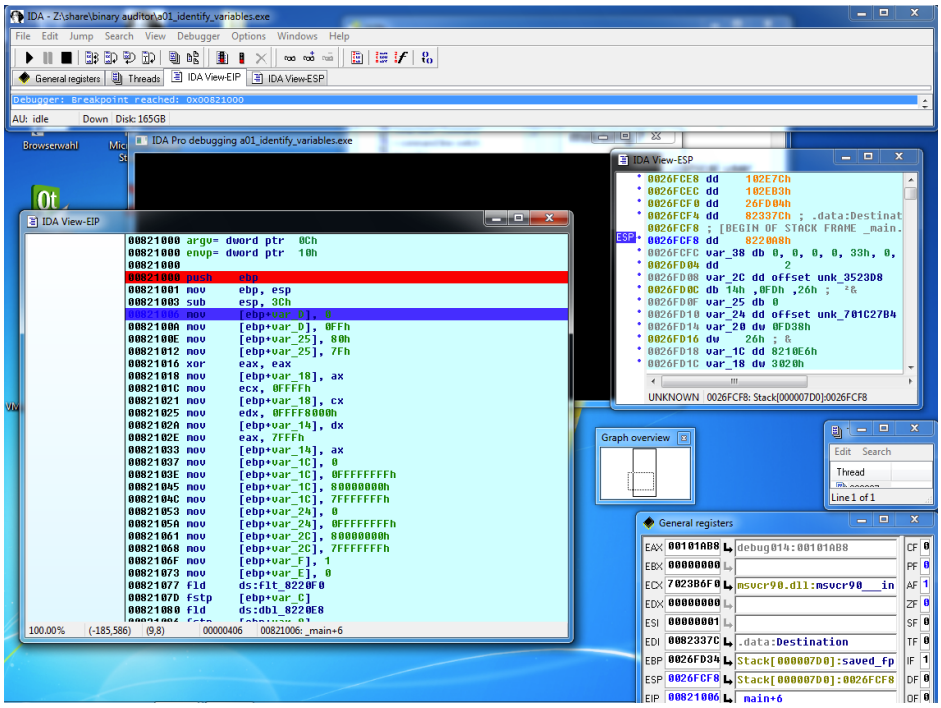

Figure 18: IDA Pro shows our disassembly, the stack window and the general registers.

Let us finally debug this application! Click the menu point "Debugger" at top. You will see 2 menu entries: "Step into F7" and "Step over F8". Step into means that when we later analyze the call of functions we are able to step inside these. Step over means that we execute a function but do not want to look inside these functions and therefore stop over them. Click "Step over" 2 or 3 times and you will see that the line which will be executed next gets colorized in blue. Now watch how the registers change, watch the stack how it changes.

Conclusion

You now know how to start IDA Pro, how to analyze dead code and how to start and use the debugger. Anyway there are many more features inside IDA Pro and it is not the job of this tutorial to explain all of them. If you are in need of a good book about IDA Pro I highly recommend "The IDA Pro Book". This book explains well how to use IDA Pro in various contexts. We will focus on practice, on how to analyze targets and how to deal with challenging problems. IDA Pro is now your bike, "The IDA Pro Book" is your bike manual but I will show you how to ride the bike and how to get a trial bike rider!