# THE BINARY AUDITOR<sup>™</sup>

BEGINNERS GUIDE

WWW.BINARY-AUDITING.COM

Copyright © 2011 Dr. Thorsten Schneider

PUBLISHED BY WWW.BINARY-AUDITING.COM

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

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in thepreparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. I cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark orservice mark. Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the CD or programs accompanying it.

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<sup>™</sup> 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<sup>TM</sup> 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<sup>™</sup>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<sup>™</sup>. 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 knowl-edge 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<sup>1</sup>. By the way: The Binary Auditor<sup>™</sup> 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.

<sup>1</sup> You can read a summary of this method at http://www.cognitivedebugging.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:

IDA - The Interactive Disassembler	
Freeware Version 5.0	
(c) 2010 Hex-Rays SA	
Welcome to the freeware edition of IDA Pro 5.0. This version is fully functional but does not offer all the bells and whistles of the commercial versions of IDA Pro.	
Try the commercial version of IDA Pro today!	
http://www.hex-rays.com	
Do not display IDA 6.x info next time OK	

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**<sup>2</sup>. Just click at **PE Executable**, then click at the **OK** button.



<sup>2</sup> 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.

Select file to d	isassemble	-	<b>×</b>
Suchen in:	🕌 binary auditor 👻	G 🤌 📂 🖽 🗸	
ea	Name	Änderungsdatum	Тур
Zuletzt besucht	a01_identify_variables	04.08.2009 15:27	Anwendu
Desktop			
Bibliotheken			
Computer			
	< III		•
Netzwerk	Dateiname:		Öffnen
	Dateityp: All known file extensions (*.386;*.a;*	app;*.axf;* 💌	Abbrechen



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.

The interactive disassembler	
File Edit Jump Search View Debugger Options Windows	Help
Load file Z:\share\binay at A En 000 000 000 000 000 000 000 000 Botable executable for 80 MS-DOS executable (EXE Binay file	udion/a01_jdentily_variables.exe as 388 [PE] [pe.ldw] [dos.ldw]
Intel 80x86 processors: me	tapc v Set
Options Create segments Load resources Rename DLL entries Manual load Fill segment gaps V Make imports segme Create FLAT group	nt Processor options
262144 32 8192 allocating m 65536 8 8192 allocating m 262144 32 8192 allocating m	Windows
589824 total memory OK Loading IDP module C:\Program Fil Autoanalysis subsystem has been Possible file format: MS-DOS executable (EXE) (C:\Progr	Cancel Help am Files\IDA Free\loaders\dos.ldw)
Possible file format: Portable executable for 80386 (PE	) (C:\Program Files\IDA Free\loaders\pe.ldw)
@:00000000 Down Disk	

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.



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

## 10 THE BINARY AUDITOR<sup>TM</sup> BEGINNERS GUIDE

🚺 IDA - Z:\share\binary auditor\a01_identify_variables.exe 📃 🔲 🗶
File Edit Jump Search View Debugger Options Windows Help
📔 🖽 🔶 💹 🛅 🦞 💥 🏔 🥔 🥓 🖽 👘 🖄 👘 🔪 🖓 \cdots 🛛 🥐 🗂 💽 🛶 🖓 🕾 👘 👔 🔶
ੈ En   888 881 981 "8" - + N ×   922 - # - ¥ S H K /→ ~ Ø   : ; क़ थ   uu \n   ♣ ♣ ¥ क़ ♣
圖 智 魯   伊 唱 唱   チ 方 方
The View A The View A the Function B Income In Structure A Structure En Fourier
E de teatre and teatre
Name
; Line 1 of 70
This file is generated by The Interactive Disa
; Copyright (c) 2010 by Hex-Rays SR, <supportant Licensed to: Freeware ver Graph overview</supportant 
100.00% ( 212 41) (102.140) 00000400 00401000 main
Executing function 'main' [compiling file (:]
Executing function 'onLoad' IDA is analysing the input file
You may start to explore the input file right now.
Using FLIRT signature: Microsoft VisualC 2-8/net runtime
Eurotion argument information is propagated
AU: idle Down Disk: 165GB

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.



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

#### 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:

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

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<sup>3</sup>. 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<sup>4</sup>.

<sup>3</sup> Note that this application does not need any parameters

<sup>4</sup> **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], o
5	mov	[ebp+var_D], oFFh
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, oFFFFh
11	mov	[ebp+var_18], cx
12	mov	edx, oFFFF8000h
13	mov	$[ebp+var_14]$ , dx
14	mov	eax, 7FFFh
15	mov	[ebp+var_14], ax
16	mov	[ebp+var_1C], o
17	mov	[ebp+var_1C], oFFFFFFFh
18	mov	[ebp+var_1C], 8000000h
19	mov	[ebp+var_1C], 7FFFFFFh
20	mov	[ebp+var_24], o
21	mov	[ebp+var_24], oFFFFFFFh
22	mov	[ebp+var_2C], 8000000h
23	mov	[ebp+var_2C], 7FFFFFFh
24	mov	[ebp+var_F], 1
25	mov	[ebp+var_E], o
26	fld	ds:flt_4020F0
27	fstp	[ebp+var_C]
28	fld	<b>ds</b> : dbl_4020E8
29	fstp	[ebp+var_8]
30	fld	<b>ds</b> : 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 3Ch 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<sup>5</sup>. 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.

Listing 1: Ao1 - Variables

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

<sup>5</sup> **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!

```
16 THE BINARY AUDITOR<sup>TM</sup>
BEGINNERS GUIDE
```

Listing 2: Continued Ao1 - Variables

```
33
      bool myTrue; myTrue = true; // 1 byte
34
      bool myFalse; myFalse = false; // 1 byte
35
36
       float myFloat; myFloat = 5.3431243774; // 4 bytes
37
38
      double myDouble; myDouble = 5.3431243774; // 8 bytes
39
40
      long double myLongDouble;
41
      myLongDouble = 5.3431243774; // 8 bytes
42
43
      wchar_t myWChar; myWChar = 'A'; //2 or 4 bytes
44
45
46
       return o;
   }
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. 18 THE BINARY AUDITOR<sup>TM</sup> BEGINNERS GUIDE

**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:

Exception handling	
The execution will be resumed after the exception. Do you want to pass the exception to the application? If you answer yes, the application's exception handler	
will be executed if there is one. The control of the application might be lost. Change exception definition	
Yes No Cancel	

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.

ID.	A View-EIP	
	.text:00821000 var_8= qword ptr -8 .text:00821000 argc= dword ptr 8 .text:00821000 argv= dword ptr 0Ch .text:00821000 envp= dword ptr 10h .text:00821000	
EIP •	text:00021001 mov ebp, esp text:00021002 cub ecp 20b	
	text:00021006 nov [ebp+uar_D], 0 .text:0002100A mov [ebp+uar_D], 0	
	.text:0082100E mov [ebp+var_25], 80h .text:00821012 mov [ebp+var_25], 7Fh .text:00821016 xor eax, eax	Ŧ
	<	Þ
	00000400 00821000: _main	

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.

. 🗐 ID	A View-EIP		
	.text:00821000	var 8= gword ptr -8	
	.text:00821000	arge= dword ptr 8	-
	text-88821888	argue dword ptr ACb	
	text:00021000	enung dword per 10h	
	toxt:88921888		
EIP .	. LCAL. 00021000	auch aba	
· · · ·	tout + 00021000	publi eup	
	.Lext.00021001	wover eup, esp	
	.Lext:00821003	sub esp, ach	
1 1	.text:00821006	nov [eop+var_u], e	
1 1	.text:0082100A	nov [ebp+var_b], ØFFN	
1 1	.text:0082100E	mov [ebp+var_25], 80h	
1 1	.text:00821012	nov [ebp+var_25], 7Fh	
1 .	.text:00821016	xor eax, eax	
	.text:00821018	mov [ebp+var_18], ax	
· ·	.text:0082101C	movecx, OFFFFh	
· ·	.text:00821021	mov [ebp+var_18], cx	
· ·	.text:00821025	mov edx, OFFFF8000h	
1 •	.text:0082102A	mov [ebp+var 14], dx	
÷ •	.text:0082102E	nov eax, 7FFFh	
· ·	.text:00821033	mov [ebp+var 14], ax	
•	.text:00821037	mov řebp+var 101. G	
•	.text:0082103E	mov [ebp+var 10], ØFFFFFFh	
· ·	.text:00821045	mov [ebp+var 10], 8888888886	
	text:00021045	mou [abp+tiar 10] ZEEFEEEb	
•	text:00021053	mou [abp-tar_26] 6	
1.	text:00021050	mou [obp-vul_24], 0	
+ .	.text.00021054		
	.text.00021001		
	.Lext:00021000		
1.	.LEX1:0082100F		
Η.	.Lext:00821073		
1 1	.text:008210//	+10 US:+1C 8220F0	
1 1	.text:008210/D	+stp [ebp+var_U]	
1	.text:00821080	FIG ds:db1_8228E8	
Progr	am control flow 1986	Fstp [ebp+var_8]	
-	1089	F1d ds:db1_8220E8	
	.text:0082108F	fstp [ebp+var_38]	
	.text:00821092	mov ecx, 41h	
1.	.text:00821097	mov [ebp+var_20], cx	
1.	.text:0082109B	xor eax, eax	
· ·	.text:0082109D	mov esp, ebp	
· ·	.text:0082109F	pop ebp	
+ •	.text:008210A0	iretn i i i i i i i i i i i i i i i i i i i	
1	.text:008210A0	nain endp	
1	.text:008210A0		
	.text:008210A0	· · · · · · · · · · · · · · · · · · ·	
t •	.text:008210A1	bute 8210A1 db 3Bh, 0Dh ; DATA XREF: sub 821785+1110	
•	.text:008210A3	dd offset dword 823000	
1 ·	.text:00821007	db 75b	
1	text:88821888	· · · · · · · · · · · · · · · · · · ·	
t •	text:00821008	add db bl	
+ •	text - 88821866	retn	
1	toyt - 88821800	· · · · · · · · · · · · · · · · · · ·	
1		3	
1	4	III.	,
2	00000400 0000100	No	
	00000400 0082100	Ju; _main	

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.

E] IDA View-EIP	
	III N GA
	08821000; File Name : Z:\share\binary auditor\a01_identify_warlables.exe
	Wegging : Ingebar : A0000
	D0021000 ; Section 1. (virtual address 00001000)
	00221000; Ujrtual size : 00000000 ( 2200.)
	BR21000 ; Offset to raw data for section: 00000408
	08021000 ; Flags 60000020: Text Executable Readable
	Describe ; Alignment : default
	DH821000 ; Application type: Executable 32bit
	0821000
	section ; segment permissions: Read/Execute
	00021000_text segment para public 'CODE' use32
	UR21000 ASSUME CS_CEXT
	UN821000 assume esinothing, ssinothing, dsi_data, fsimothing, gsinothing
	0821000
	08021000 ; Attributes: bp-based frame
	08821000
	www.rowe; intocorci main(int argc,const char ++arge,const char +enep) 00021000 nain proc mear
	0821000
	east reserved at a state of the
	one too var_25- but ptr -25h
	08821000 var_23= dieord ptr -23h
	BB21000 var 12 vend otr - 10
	00021000 var_18- word ptr -18h
	SN21005 Var_14- word ptr -14m SN21005 var_5- but str -07b
	0821000 var E= byte ptr -0Eh
	0021000 var D- byte ptr -00h
	electrono da condica ptr - sca electrono var 8 e quord ptr - 8
	08221000 args= duord ptr 8
	election myse during by the second seco
	04821000
	00221001 nov ebp. esp
	082210(3 sub esp. 3Ch
	weezinke nov (exprove_), o 08021000 nov (exprove_), o 08021000 nov (exprove_), off
	0082100E nov [ebp+var_25], 80h
	BBS21012 NOV [objevar_25], 7Fh BBS21016 Yorn Bay Bay
	00021018 nov [ebp+var_18], ax
	0002101C nov ccx, 0FFFh
	8821025 nov edx, 4FFFF8000
	08221020 nov [ebp+var_14], dx
	exertise nov eax, rent
	08221037 nov [cbp+var_10], 0
	weeling nov [copyer_10], with the second
	8882104C nov [ebpvar_10], 7FFFFFFh
	DERIVITIES ROU [CDP+GAT_20], 0 EMERICAN ROU [CDP+GAT_20], 0 EMER
	68621051 nov [etp+var_20], 80000000
	00821058 nov [cbp+var_2C], 7FFFFFFh 00821058 nov [cbp+var_2C], 1
	88821873 nov [etp=var_6], 0
	00821077 F10 05:F11.8220F0 0092107 F10 00000000000000000000000000000000
	68221090 f1d ds:c01_0220E8
	88221086 Fstp [ebp+oar.8] 69221086 [std. dc-ebb.922059
	6882168F fstp [cbp+oar_38]
	BBS21092 ROU CEX, ATh BBS21092 ROU EDMANN 201 FY
	sectors and following the sector sect
	UB21098 XOP eax, eax
	BAR21070 NOV esk, esk BAR21070 NOV esk, ebp
	944721194 xor e2x, e3x 948721950 xor e5p, e5p 94872195 pop e5p 94872196 pop e5p
	100221000 park 6.24, 6.24 100221000 park 6.24, 6.24 100221000 park 6.24 10022100 park 6.24 100221000 park 6.24 10022100000000000000000000000000000000
	90021000 2010 2010 2010 2010 2012001 2010 2010

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!