Reverse Engineering Malware IDA & Olly Basics 5 parts by otw

Contents

Reverse Engineering Malware: Why You Should Study Reverse Engineering Malware . 3

What is Reverse Engineering Malware? 3

Why Reverse Engineering Malware? 4

Reverse Engineering Malware, Part 1: Getting Started 7

Let's get started! 7

What is Reversing Engineering? 7

Reverse Engineering Applied to Malware 7

Low Level Software 8

Assembly Code 8

Machine Code 9

Compilers 9

The Reversing Process 9

Code Level 9

System level 10

Reversing Tools 10

Legality 11

Reverse Engineering Malware, Part 2: Assembler Language Basics 12

Pieces 12

Registers 12

Flags 14

Instructions 15

Reverse Engineering Malware, Part 3: IDA Pro Introduction 19

Step #1 Download and Install 20

Step #2 Load a PE File 21

Step #3 Start the Disassembly 22

Step 5: Show Imports 26

Step 6: Customize the Analysis 27

Reverse Engineering Malware, Part 4: Windows Internals 30

Virtual Memory 31

Kernel v User Mode 32
Kernel memory Space 32
Paging 32
Objects and Handles 33
Handles 33
Processes 34
Process Initialization 34
Threads 35
Context Switch 35
Win32 API 36
System Calls 36
PE Format 37
Relocation Issues 38
Image Sections 38
Section Alignment 38
DLL's 38
Loading DLL's 39
PE Headers 39

Reverse Engineering Malware, Part 5: OllyDbg Basics 40
Step #1: Starting OllyDbg 41
Step #2: Loading a File into OllyDbg 42
Step #3: Different Views of the Code 43
Breakpoints 49
OllyDbg Frequently Used Shortcuts 50
Complete List of Shortcuts 52

Reverse Engineering Malware: Why You Should Study Reverse Engineering Malware

I am about to embark upon probably the most technically demanding tutorial series, Reverse Engineering Malware. Before I do so, I thought I would take a few moments to explain why you should study and invest your time into reverse engineering. Please take a moment to read the following and then, hopefully, decide whether this discipline is worth your time to advance your career in cyber security.

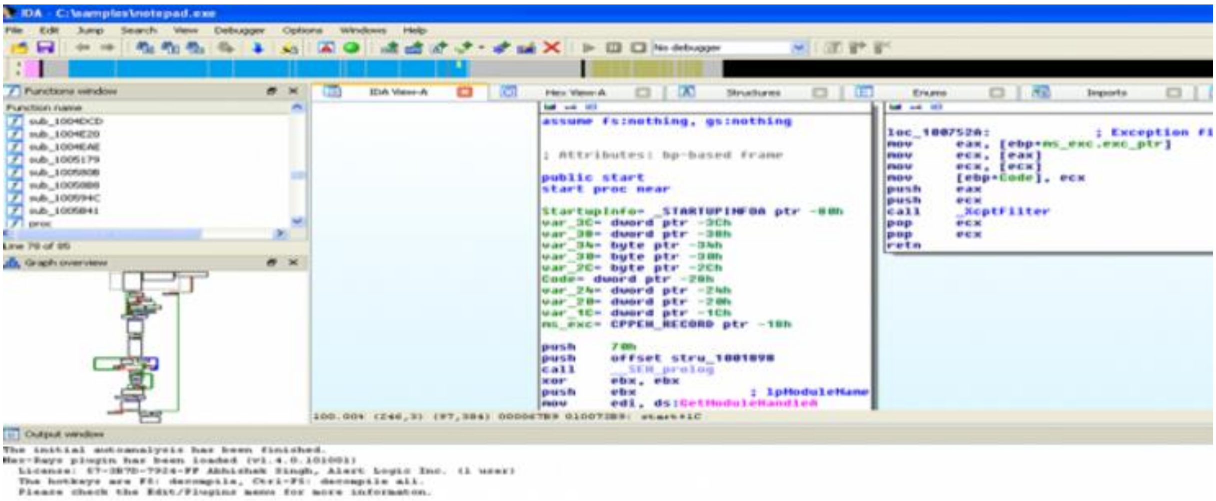
What is Reverse Engineering Malware?

In this series, we will be dissecting **known malware** to understand how it works, its operation and its "signature". According to the Merriam-Webster dictionary, reverse engineering is defined as "disassemble or analyze in detail in order to discover concepts involved in manufacture". That is precisely my intent with this series, **to analyze in detail to discover concepts involved in manufacture"** of malware.

Furthermore, Wikipedia defines reverse engineering as;

the process of discovering the technological principles of a(n)....application through analysis of its structure, function and operation. That involves sometimes taking something apart and analyzing its workings in detail, usually with the intention to ***construct a new device or program that does the same thing*** *without actually copying anything from the original. (my emphasis added)*

We will be using a number of different tools in this analysis including virtual machines, sandboxes, unpackers, disassemblers and debuggers to do so. Wherever possible, I will use free and open source tools.



```
assume fs:nothing, gs:nothing


; Attributes: bp-based frame

public start
start proc near

StartupInfo= _STARTUPINFOA ptr -80h
var_3C= dword ptr -3Ch
var_38= dword ptr -38h
var_34= byte ptr -34h
var_30= byte ptr -30h
var_2C= byte ptr -2Ch
Code= dword ptr -28h
var_24= dword ptr -24h
var_20= dword ptr -20h
var_1C= dword ptr -1Ch
ms_exc= CPPEH_RECORD ptr -18h

push    70h
push    offset stru_1001898
call    __SEH_prolog
xor     ebx, ebx
push    ebx             ; lpModuleName
mov     edi, ds:GetModuleHandleA
```

```
loc_100752A:                    ; Exception fi
mov     eax, [ebp+ms_exc.exc_ptr]
mov     ecx, [eax]
mov     ecx, [ecx]
mov     [ebp+Code], ecx
push    eax
push    ecx
call    _XcptFilter
pop     ecx
pop     ecx
retn
```

```
100.00% (246,3) (97,384) 000067B9 010073B9: start+1C
```

```
The initial autoanalysis has been finished.
Hex-Rays plugin has been loaded (v1.4.0.101001)
  License: 57-3B7D-7924-FF Abhishek Singh, Alert Logic Inc. (1 user)
  The hotkeys are F5: decompile, Ctrl-F5: decompile all.
  Please check the Edit/Plugins menu for more information.
```

Why Reverse Engineering Malware?

#1 To Gain a Deeper and More Thorough Understanding of Applications and Operating Systems

If nothing else, by reverse engineering malware, you will gain a deeper and more thorough understanding of the operating systems and applications. Malware must use and exploit these operating systems and applications for it's own malicious purposes and by dissecting the malware and its operation, you can better understand not only how the malware works, but the functioning of the OS and apps.

#2 Train to Work in Forensic Malware Analysis

Presently, the highest paid and most in-demand sub-discipline in digital forensics is for those capable of dissecting malware and using this information for attribution. When new malware appears, it is most often those that can reverse the malware that are commissioned to attribute its source. This becomes increasingly important in the fields of cyber espionage and cyber warfare between nation states.

By reading and studying this series, Reverse Engineering Malware, you will begin your preparation for this rewarding career.

#3 Build Security Applications

Before can even begin to build security applications to protect systems, you first need to understand how the malware works. Whether working in Intrusion Detection Systems (IDS) development, AV software, firewalls or the latest Artificial Intelligence (AI) based security systems, you must have an understanding how the malware functions and, therefore, how it can be detected and neutralized.

#4 Be Better Prepared as a Forensic Analyst or Incident Response Handler

Reverse Engineering Malware will help incident responders and forensic analysts/investigators to assess quickly the severity of a breach to better plan for recovery. By studying reverse engineering of malware, the forensic investigator can establish the key indicators of a compromise and then plan for containing and recovering from an incident.

5 Build Your Own Zer0-Day Exploits

The "Holy Grail" of any security researcher, hacker or pentester is to develop a zer0-day exploit. Whether you are a White Hat trying to develop a proof-of-concept (POC) exploit, a Bug Bounty Hunter, or a Black hat looking to exploit the latest new app, you must understand the inner workings of the operating system, the app and probably, the previous malware that has been developed. In this series, we will explore the inner workings of some common operating systems and applications and some malware that has successfully exploited those systems. By learning how these systems have been compromised in the past, you will have a better concept of how to develop your own. In addition, like all software development, it does not make any sense to "reinvent the wheel". All software developers re-use code to save time and money. This applies equally to malware developers (that code re-use can often provide evidence towards attribution). Here, we will study some common and successful malware over the years, many of which have modules that can be re-used.

Without the ability to build your own exploits, your career as a pentester/hacker will be largely limited to running other peoples' code. To reach the highest echelons of the security/pentesting industry you will need to understand previously deployed malware and develop your own.

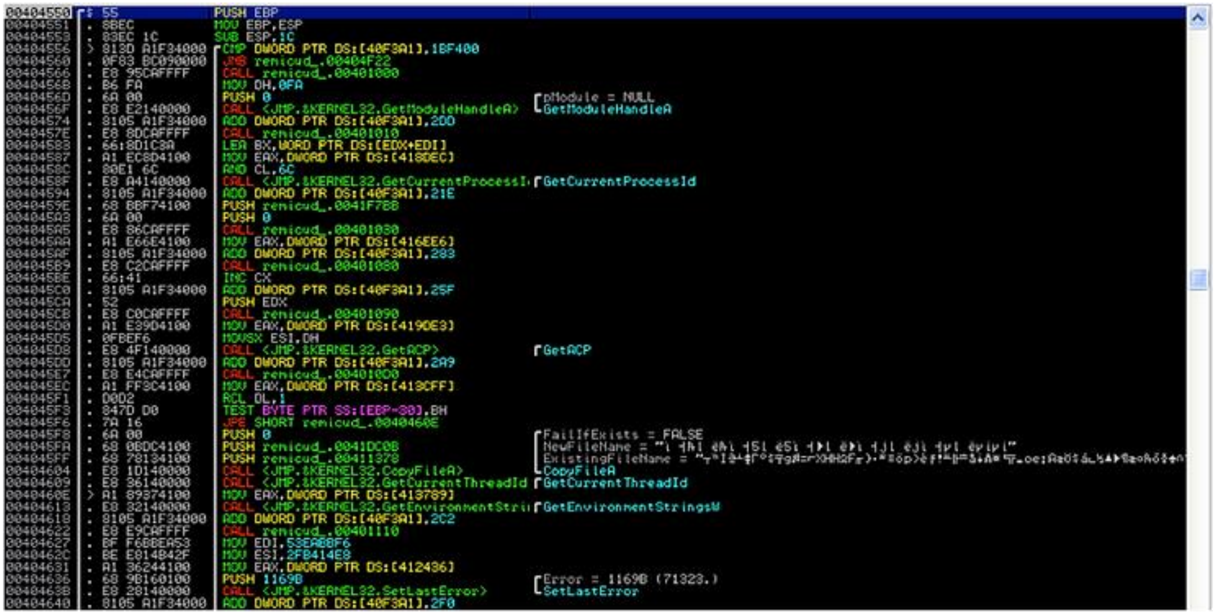
Some of the subjects we will address in this series include;

- **Assembly Language Review**
- **Introduction to Malware Analysis**
- **Reversing with Disassemblers**
- **Reversing with a DeBugger**
- **User Mode Debuggers**
- **Reversing Win32 with IDA Pro**
- **Reversing Stacks and Heaps**
- **Windows Internals**
- **Linux Internals**
- **Reversing Data Structures**
- **Structured Exception Handling**
- **System level Reversing**
- **Reversing Bots**
- **Reversing Infection Vectors**
- **Encoders and Compressors**
- **Auditing Binaries**
- **Binary Diffing**
- **Reversing Encryption**
- **Detecting Debuggers and Disassemblers**

Reverse Engineering Malware, Part 1: Getting Started

Let's get started!

Reverse Engineering malware is a deep and sophisticated subject matter, hence few people actually master it. This is the primary reason why the salaries in this field are SO high. Before we proceed, we need to develop a conceptual framework and elaborate of some strategies and issues relating to reverse engineering malware. So, let' s do that first.



```
00404550 r$ 55              PUSH EBP
00404551  . 8BEC            MOV EBP,ESP
00404553  . 83EC 1C         SUB ESP,1C
00404556  > 813D A1F34000   rCMP DWORD PTR DS:[40F3A1],1BF400
00404560  . 0F83 BC090000   JNB renicud_.00404F22
00404566  . E8 95CAFFFF     CALL renicud_.00401000
0040456B  . B6 FA           MOV DH,0FA
0040456D  . 6A 00           PUSH 0                                      [pModule = NULL
0040456F  . E8 E2140000     CALL <JMP.&KERNEL32.GetModuleHandleA>       [GetModuleHandleA
00404574  . 8105 A1F34000   ADD DWORD PTR DS:[40F3A1],2DD
0040457E  . E8 8DCAFFFF     CALL renicud_.00401010
00404583  . 66:8D1C3A       LEA BX,WORD PTR DS:[EDX+EDI]
00404587  . A1 EC8D4100     MOV EAX,DWORD PTR DS:[418DEC]
0040458C  . 80E1 6C         AND CL,6C
0040458F  . E8 A4140000     CALL <JMP.&KERNEL32.GetCurrentProcessI      rGetCurrentProcessId
00404594  . 8105 A1F34000   ADD DWORD PTR DS:[40F3A1],21E
0040459E  . 68 BBF74100     PUSH renicud_.0041F7BB
004045A3  . 6A 00           PUSH 0
004045A5  . E8 86CAFFFF     CALL renicud_.00401030
004045AA  . A1 E66E4100     MOV EAX,DWORD PTR DS:[416EE6]
004045AF  . 8105 A1F34000   ADD DWORD PTR DS:[40F3A1],283
004045B9  . E8 C2CAFFFF     CALL renicud_.00401080
004045BE  . 66:41           INC CX
004045C0  . 8105 A1F34000   ADD DWORD PTR DS:[40F3A1],25F
004045CA  . 52              PUSH EDX
004045CB  . E8 C0CAFFFF     CALL renicud_.00401090
004045D0  . A1 E39D4100     MOV EAX,DWORD PTR DS:[419DE3]
004045D5  . 0FBEF6          MOVSX ESI,DH
004045D8  . E8 4F140000     CALL <JMP.&KERNEL32.GetACP>                 rGetACP
004045DD  . 8105 A1F34000   ADD DWORD PTR DS:[40F3A1],2A9
004045E7  . E8 E4CAFFFF     CALL renicud_.004010D0
004045EC  . A1 FF3C4100     MOV EAX,DWORD PTR DS:[413CFF]
004045F1  . D0D2            RCL DL,1
004045F3  . 847D D0         TEST BYTE PTR SS:[EBP-30],BH
004045F6  . 7A 16           JPE SHORT renicud_.0040460E
004045F8  . 6A 00           PUSH 0                                      [FailIfExists = FALSE
004045FA  . 68 0BDC4100     PUSH renicud_.0041DC0B                      |NewFileName = [illegible]
004045FF  . 68 78134100     PUSH renicud_.00411378                      |ExistingFileName = [illegible]
00404604  . E8 1D140000     CALL <JMP.&KERNEL32.CopyFileA>              [CopyFileA
00404609  . E8 36140000     CALL <JMP.&KERNEL32.GetCurrentThreadId      rGetCurrentThreadId
0040460E  > A1 89374100     MOV EAX,DWORD PTR DS:[413789]
00404613  . E8 32140000     CALL <JMP.&KERNEL32.GetEnvironmentStri      rGetEnvironmentStringsW
00404618  . 8105 A1F34000   ADD DWORD PTR DS:[40F3A1],2C2
00404622  . E8 E9CAFFFF     CALL renicud_.00401110
00404627  . BF F60BEA53     MOV EDI,53EA0BF6
0040462C  . BE E814842F     MOV ESI,2F8414E8
00404631  . A1 36244100     MOV EAX,DWORD PTR DS:[412436]
00404636  . 68 9B160100     PUSH 1169B                                  [Error = 1169B (71323.)
0040463B  . E8 28140000     CALL <JMP.&KERNEL32.SetLastError>           [SetLastError
00404640  . 8105 A1F34000   ADD DWORD PTR DS:[40F3A1],2F0
```

What is Reversing Engineering?

Although definitions vary a bit about what exactly is reverse engineering, **in this series** we will trying to determine what a piece of software (malware) does even when we don't have access to the source code (usually the case). After determining what the software does, then we will attempt to (1) either tweak it to do something slightly different or (2) re-construct it in another piece of software (malware).

Reverse Engineering Applied to Malware

Reverse engineering is used on both termini of malware development and delivery. At the developer terminus, reverse engineering is used to find vulnerabilities in operating systems and applications that the malware can exploit. In addition, the developers can use reverse engineering to find and use a module from someone else's malware. Like all

software developers, malware developers re-use useful code from others' software. No sense in re-inventing the wheel even when doing malware development.
At the other terminus, forensic investigators and incident handlers can use reverse engineering to trace what a piece of malware does and what harm it might bring. Furthermore, reverse engineering can often give the forensic investigator a clue to the origin and attribution of the malware.

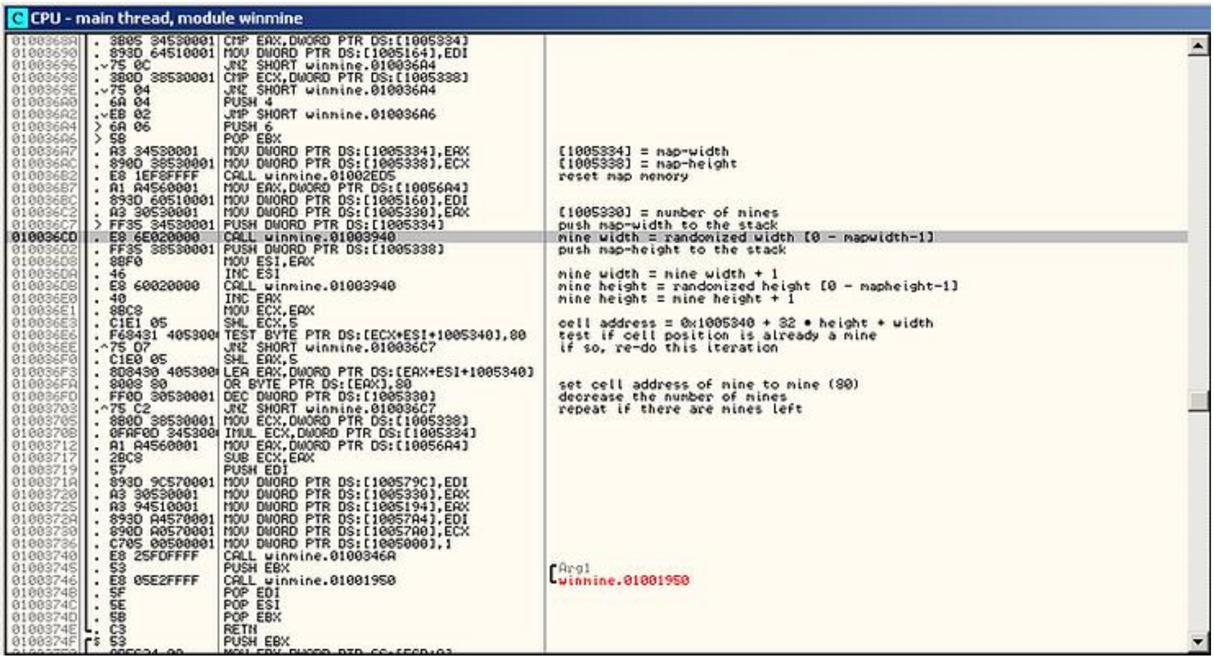
Low Level Software

In reverse engineering software, we often are working in low-level software. The source code is most often not available to us, but the low-level software **alway**s is.

Assembly Code

Assembly is the lowest level in the software chain and although we don't have access to the source code, various tools can reduce the source code to assembly. Each instruction in any higher level language must be visible to the assembly language code. There is no magic here, each instruction must be reduced to one or more assembly instructions. In most cases, we will be working with this simple assembly code when reverse engineering.

Obviously, to be successful at reversing, we must be familiar with assembly language code. Unfortunately, there is not a **single** assembly language, but rather an assembly language for each type of processor (x86, x64, ARM, PPC, etc). To master reversing, we must master the assembly code of our chosen platform. In this series, we will be examining x86, x64 and ARM assembly.



```
0100368A  . 3B05 34530001 CMP EAX,DWORD PTR DS:[1005334]
01003690  . 893D 64510001 MOV DWORD PTR DS:[1005164],EDI
01003696  .v75 0C         JNZ SHORT winmine.010036A4
01003698  . 3B0D 38530001 CMP ECX,DWORD PTR DS:[1005338]
0100369E  .v75 04         JNZ SHORT winmine.010036A4
010036A0  . 6A 04         PUSH 4
010036A2  .vEB 02         JMP SHORT winmine.010036A6
010036A4  > 6A 06         PUSH 6
010036A6  > 5B            POP EBX
010036A7  . A3 34530001   MOV DWORD PTR DS:[1005334],EAX       [1005334] = map-width
010036AC  . 890D 38530001 MOV DWORD PTR DS:[1005338],ECX       [1005338] = map-height
010036B2  . E8 1EF8FFFF   CALL winmine.01002ED5                reset map memory
010036B7  . A1 A4560001   MOV EAX,DWORD PTR DS:[10056A4]
010036BC  . 893D 60510001 MOV DWORD PTR DS:[1005160],EDI
010036C2  . A3 30530001   MOV DWORD PTR DS:[1005330],EAX       [1005330] = number of mines
010036C7  > FF35 34530001 PUSH DWORD PTR DS:[1005334]          push map-width to the stack
010036CD  . E8 6E020000   CALL winmine.01003940                mine width = randomized width [0 - mapwidth-1]
010036D2  . FF35 38530001 PUSH DWORD PTR DS:[1005338]          push map-height to the stack
010036D8  . 8BF0          MOV ESI,EAX
010036DA  . 46            INC ESI                              mine width = mine width + 1
010036DB  . E8 60020000   CALL winmine.01003940                mine height = randomized height [0 - mapheight-1]
010036E0  . 40            INC EAX                              mine height = mine height + 1
010036E1  . 8BC8          MOV ECX,EAX
010036E3  . C1E1 05       SHL ECX,5                            cell address = 0x1005340 + 32 * height + width
010036E6  . F68431 405300 TEST BYTE PTR DS:[ECX+ESI+1005340],80 test if cell position is already a mine
010036EE  .^75 D7         JNZ SHORT winmine.010036C7           if so, re-do this iteration
010036F0  . C1E0 05       SHL EAX,5
010036F3  . 8D8430 405300 LEA EAX,DWORD PTR DS:[EAX+ESI+1005340]
010036FA  . 8008 80       OR BYTE PTR DS:[EAX],80              set cell address of mine to mine (80)
010036FD  . FF0D 30530001 DEC DWORD PTR DS:[1005330]           decrease the number of mines
01003703  .^75 C2         JNZ SHORT winmine.010036C7           repeat if there are mines left
01003705  . 8B0D 38530001 MOV ECX,DWORD PTR DS:[1005338]
0100370B  . 0FAF0D 345300 IMUL ECX,DWORD PTR DS:[1005334]
01003712  . A1 A4560001   MOV EAX,DWORD PTR DS:[10056A4]
01003717  . 2BC8          SUB ECX,EAX
01003719  . 57            PUSH EDI
0100371A  . 893D 9C570001 MOV DWORD PTR DS:[100579C],EDI
01003720  . A3 30530001   MOV DWORD PTR DS:[1005330],EAX
01003725  . A3 94510001   MOV DWORD PTR DS:[1005194],EAX
0100372A  . 893D A4570001 MOV DWORD PTR DS:[10057A4],EDI
01003730  . 890D A0570001 MOV DWORD PTR DS:[10057A0],ECX
01003736  . C705 00500001 MOV DWORD PTR DS:[1005000],1
01003740  . E8 25FDFFFF   CALL winmine.0100346A
01003745  . 53            PUSH EBX                             Arg1
01003746  . E8 05E2FFFF   CALL winmine.01001950                winmine.01001950
0100374B  . 5F            POP EDI
0100374C  . 5E            POP ESI
0100374D  . 5B            POP EBX
0100374E  . C3            RETN
0100374F  $ 53            PUSH EBX
```

Machine Code

Machine code or binary code is the code read by the CPU. Machine code and assembly are two different representations of the same thing. Machine code is simply a sequence of bits that contain instructions for the CPU.
Assembly language is simply textual representation of machine code that makes them more easily human readable (but not much more). Each assembly language command is represented by a number called the opcode, short for operation code.

Compilers

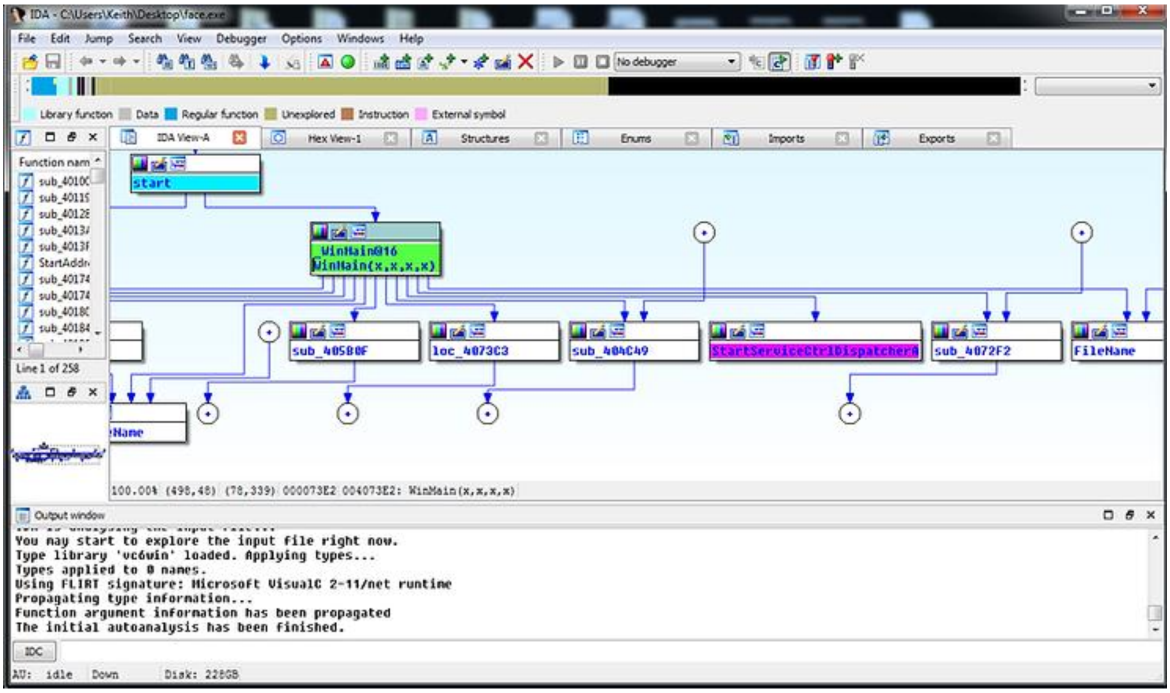
Compilers convert source code into machine code. One of the biggest challenges in the reversing process is that compilers tend to optimize the code to make it more efficient and perform better. Therefore, the same code compiled by two different compilers will actually generate slightly different machine code making our job of reversing more difficult.

The Reversing Process

The reversing process can usually be broken down into at least two types; (1) code level and (2) system level.

Code Level

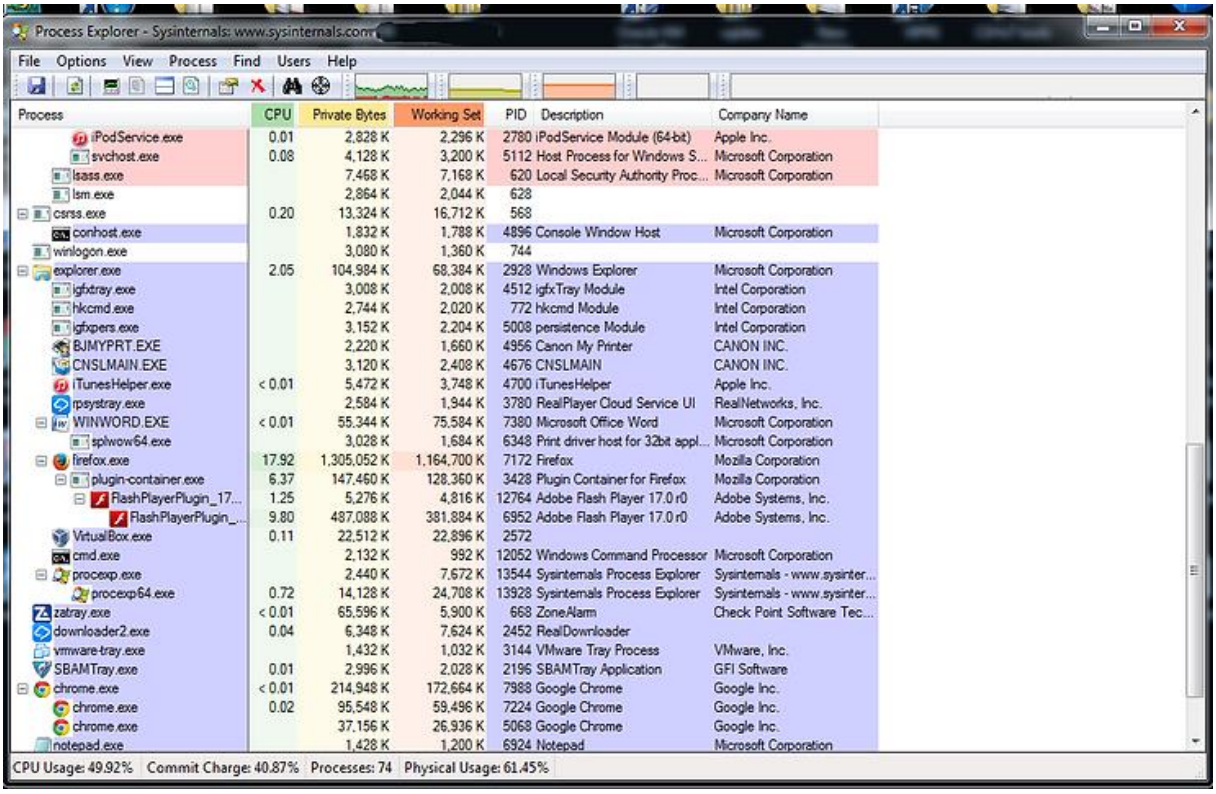
When we do code level reversing, we are attempting to extract the software's code concepts and algorithms from the machine code. This requires a solid understanding of such things as how the CPU works, how the operating system works and the process of software development. We will be using such tools as IDA Pro, SoftIce, Ollydbg, Ghidra and some others in this process.



```
You may start to explore the input file right now.
Type library 'vc6win' loaded. Applying types...
Types applied to 0 names.
Using FLIRT signature: Microsoft VisualC 2-11/net runtime
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.
```

System level

System level reversing involves running tools to obtain information about the software, inspect the program, inspect the executables, and track the program's input and output. Most of this information will come from the operating system. We will be using such tools as SysInternals Suite, Tripwire, lsof, Wireshark, and others.



Process	CPU	Private Bytes	Working Set	PID	Description	Company Name
iPodService.exe	0.01	2,828 K	2,296 K	2780	iPodService Module (64-bit)	Apple Inc.
svchost.exe	0.08	4,128 K	3,200 K	5112	Host Process for Windows S...	Microsoft Corporation
lsass.exe		7,468 K	7,168 K	620	Local Security Authority Proc...	Microsoft Corporation
lsm.exe		2,864 K	2,044 K	628		
csrss.exe	0.20	13,324 K	16,712 K	568		
conhost.exe		1,832 K	1,788 K	4896	Console Window Host	Microsoft Corporation
winlogon.exe		3,080 K	1,360 K	744		
explorer.exe	2.05	104,984 K	68,384 K	2928	Windows Explorer	Microsoft Corporation
igfxtray.exe		3,008 K	2,008 K	4512	igfxTray Module	Intel Corporation
hkcmd.exe		2,744 K	2,020 K	772	hkcmd Module	Intel Corporation
igfxpers.exe		3,152 K	2,204 K	5008	persistence Module	Intel Corporation
BJMYPRT.EXE		2,220 K	1,660 K	4956	Canon My Printer	CANON INC.
CNSLMAIN.EXE		3,120 K	2,408 K	4676	CNSLMAIN	CANON INC.
iTunesHelper.exe	< 0.01	5,472 K	3,748 K	4700	iTunesHelper	Apple Inc.
rpsystray.exe		2,584 K	1,944 K	3780	RealPlayer Cloud Service UI	RealNetworks, Inc.
WINWORD.EXE	< 0.01	55,344 K	75,584 K	7380	Microsoft Office Word	Microsoft Corporation
splwow64.exe		3,028 K	1,684 K	6348	Print driver host for 32bit appl...	Microsoft Corporation
firefox.exe	17.92	1,305,052 K	1,164,700 K	7172	Firefox	Mozilla Corporation
plugin-container.exe	6.37	147,460 K	128,360 K	3428	Plugin Container for Firefox	Mozilla Corporation
FlashPlayerPlugin_17...	1.25	5,276 K	4,816 K	12764	Adobe Flash Player 17.0 r0	Adobe Systems, Inc.
FlashPlayerPlugin_...	9.80	487,088 K	381,884 K	6952	Adobe Flash Player 17.0 r0	Adobe Systems, Inc.
VirtualBox.exe	0.11	22,512 K	22,896 K	2572		
cmd.exe		2,132 K	992 K	12052	Windows Command Processor	Microsoft Corporation
procexp.exe		2,440 K	7,672 K	13544	Sysinternals Process Explorer	Sysinternals - www.sysinter...
procexp64.exe	0.72	14,128 K	24,708 K	13928	Sysinternals Process Explorer	Sysinternals - www.sysinter...
zatray.exe	< 0.01	65,596 K	5,900 K	668	ZoneAlarm	Check Point Software Tec...
downloader2.exe	0.04	6,348 K	7,624 K	2452	RealDownloader	
vmware-tray.exe		1,432 K	1,032 K	3144	VMware Tray Process	VMware, Inc.
SBAMTray.exe	0.01	2,996 K	2,028 K	2196	SBAMTray Application	GFI Software
chrome.exe	< 0.01	214,948 K	172,664 K	7988	Google Chrome	Google Inc.
chrome.exe	0.02	95,548 K	59,496 K	7224	Google Chrome	Google Inc.
chrome.exe		37,156 K	26,936 K	5068	Google Chrome	Google Inc.
notepad.exe		1,428 K	1,200 K	6924	Notepad	Microsoft Corporation

CPU Usage: 49.92% Commit Charge: 40.87% Processes: 74 Physical Usage: 61.45%

Reversing Tools

Reverse Engineering tools can be broken down to several categories. These include;

(1) **System-level Tools**

These tools sniff, monitor and explore the software we are examining. In most cases, they use the operating system to gather info on the malware.

(2) **Disassemblers**

Disassemblers take the software and generate the assembly code for the program. In this way, we can examine the inner workings of the malware without seeing the source code.

(3) **Debuggers**

A debugger enables us to observe a program while it is running. It enables us to set breakpoints and trace through the code.

(4) **Decompilers**

A decompiler attempts to take an executable and re-create the source code in a high-level language. Although imperfect due to the fact that compilers vary and omit steps for efficiency, this can still be a productive process in the reversing discipline.

Legality

The legality of reverse engineering has always been controversial. The question of legality revolves around the issue of the social and economic impact of reverse engineering. For instance, if you were to reverse engineer Microsoft's Excel and then re-sell it, that would very likely be deemed illegal. If you are reverse engineering malware to decipher its capabilities and origins, that will likely be deemed legal.

Copyright law and the Digital Millenium Copyright Act (DMCA) are key pieces of legislation pertinent to reverse engineering. Some have claimed that creating an intermediate copy of a software program during the reverse engineering process is in itself a violation of the Copyright law. Fortunately, the courts have disagreed.

On the other hand, the DMCA protects copyright protected systems from being copied. In almost every case, circumvention of DMCA protections involves reverse engineering. We will look at a few of those ways in this course of study.

Copyright protections usually involve Digital Rights Management technology and circumvention of these systems is **ALWAYS** illegal even for personal use. It is illegal even to develop or make available such means to circumvent DRM.

There is an exception, however. You may reverse and circumvent copyright protection on software for the purpose of evaluating or improving the security of a computer system. It is this exception that our work falls within.

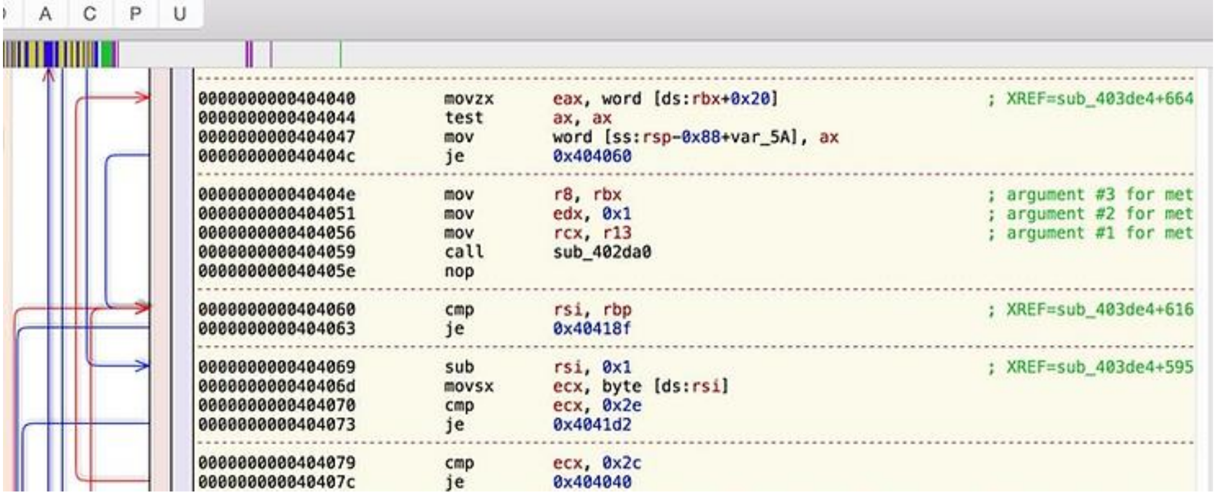
Conclusion

I hope that this introduction has given you a framework for understanding the reverse engineering malware process and has whet your appetite for what is to come. Keep coming back as I step your through the exciting process of reverse engineering malware!

Reverse Engineering Malware, Part 2: Assembler Language Basics

Most of the work we will be doing in reverse engineering will be with assembler language. This simple and sometimes tedious language can reveal a plethora of information on the source code. When we can't see or recover the source code of the malware or other software, we can use tools such as dis-assemblers and debuggers to recover the underlying assembler of the software. From there, of course, we can then decipher what the software was attempting to do.

In this tutorial, I will simply be listing the most basic and fundamental assembler instructions. I suspect most of you will simply use it a a reference as we progress though this study, so make certain to bookmark this page so that you can easily come back to it.



```
0000000000404040         movzx      eax, word [ds:rbx+0x20]                   ; XREF=sub_403de4+664
0000000000404044         test       ax, ax
0000000000404047         mov        word [ss:rsp-0x88+var_5A], ax
000000000040404c         je         0x404060

000000000040404e         mov        r8, rbx                                   ; argument #3 for met
0000000000404051         mov        edx, 0x1                                  ; argument #2 for met
0000000000404056         mov        rcx, r13                                  ; argument #1 for met
0000000000404059         call       sub_402da0
000000000040405e         nop

0000000000404060         cmp        rsi, rbp                                  ; XREF=sub_403de4+616
0000000000404063         je         0x40418f

0000000000404069         sub        rsi, 0x1                                  ; XREF=sub_403de4+595
000000000040406d         movsx      ecx, byte [ds:rsi]
0000000000404070         cmp        ecx, 0x2e
0000000000404073         je         0x4041d2

0000000000404079         cmp        ecx, 0x2c
000000000040407c         je         0x404040
```

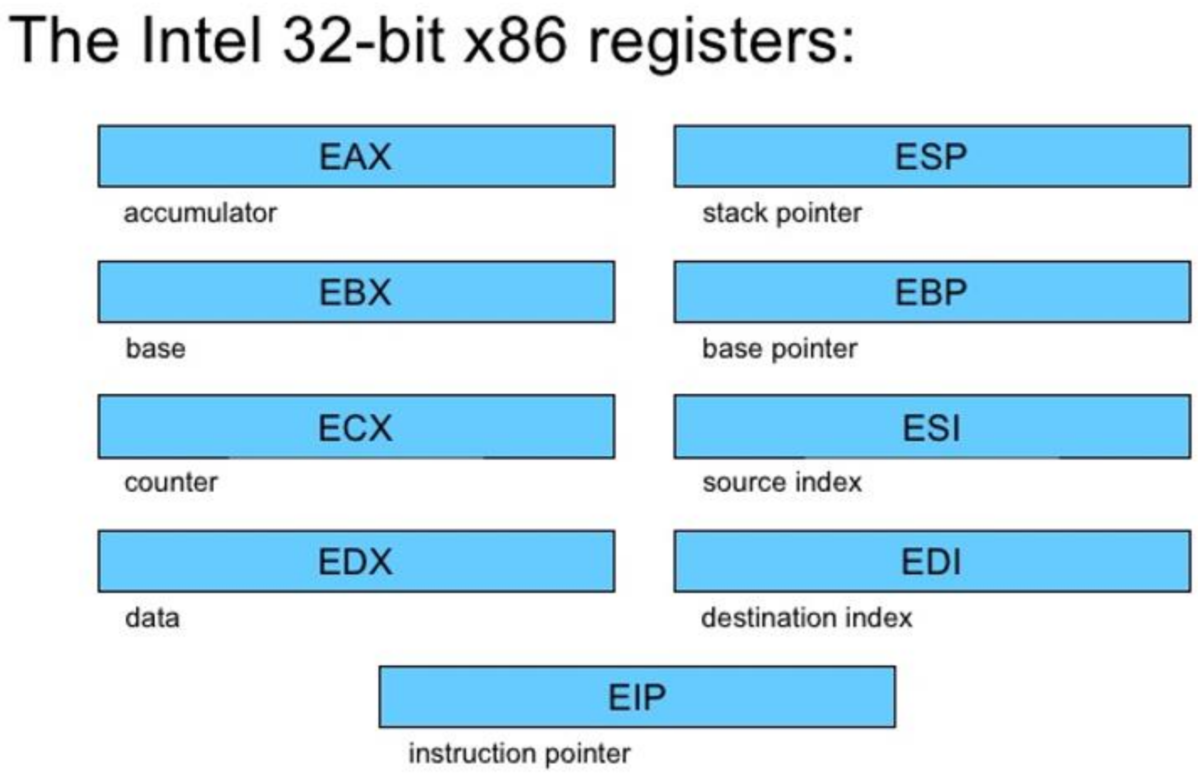
Pieces

Let's begin some every basic concepts. Hopefully, this all review for you, but if not, you need to understand these basic concepts before proceeding in this course of study.

Bit - This is the smallest piece of data. It can be a 0 or 1 or Off or ON.
Byte - a byte is 8 bits. It has a range of equivalent decimal values of 0 to 255
Word - a word is two bytes together or 16 bits
Double Word - a double word is tow words or 32 bits
Kilobyte - a kilobyte is 1024 (32 * 32) bytes
Megabyte - a megabyte is is 1,048,578 bytes (1024 x 1024).

Registers

Registers are places in computer memory where data is stored. When working in the assembler, we are usually using these registers to move and manipulate information, so you should be familiar with them.



These registers are;

EAX - Extended Accumulator Register

EBX - Extended Base Register

ECX - Extended Counter Register

EDX - Extended Data Register

ESI - Extended Source Index

EDI - Extended Destination Index

EBP - Extended Base Pointer

ESP - Extended Stack Pointer

EIP - Extended Instruction Pointer

Flags

Flags are a single bit that indicates status of a register. The flag register on modern 32 bit CPU's is 32 bits long. There are 32 flags. In our studies here, we will only need three of them; (1) the Z flag, the O flag and the C flag.

A flag can only be SET or NOT SET

Z-Flag

The Z-flag (zero flag) is the most useful flag for cracking. It is used in about 90% of all cases. It can be set or cleared by several opcodes when the last instruction that was performed has 0 as a result

O-Flag

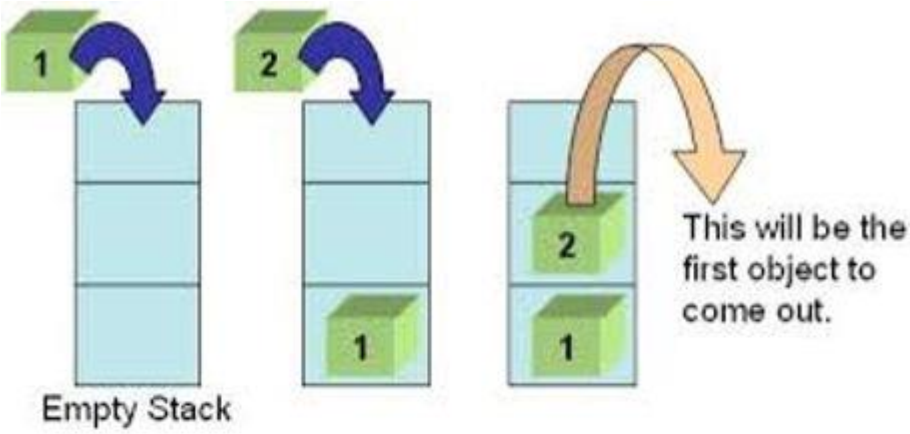
The O-flag (overflow flag) is used in about 4% of all cracking attempts. It is set when the last operation changed the highest bit of the register that gets the result of an operation.

C-Flag

The C-Flag (carry Flag) is used in about 1% of all cracking attempts. It is set, if you add a value to a register, so that it gets bigger than FFFFFFFF or is you subtract a value so that the register value is less than zero.

Stack

The stack is a part of memory where you can store different things for later use. Like a stack of books on a desk where the last on top (last in or LI) is the first to leave (LIFO).



The command PUSH saves the contents of a register on the stack. The command POP grabs the last saved contents of a register from the stack and then places it into a specific register.

Instructions

Assembler language has a small number of fundamental commands. These include;

ADD - The ADD instruction adds a value to a register or memory address.

> **Syntax:**
> ADD destination, source

AND - the AND instruction uses a logical and on two values

> **Syntax:**
> AND destination, source

CALL - the CALL instruction pushes the Relative Virtual Address (RVA) of the instruction that follows to the stack and calls a subprogram or sub-procedure

> **Syntax:**
> CALL something

CDQ - Convert DWORD to QWORD (**C**onvert **D** to **Q**)

> **Syntax:**
> CDQ

CMP - Compare

> the CMP instruction compares two things and can set the C/O/Z flags if the result of the compare fits
>
> **Syntax:**
> CMP destination, source

DEC - Decrement

> the decrement command is used to decrease a value
> decreases a value (value= value -1)
>
> **Syntax:**
> DEC something

DIV - Division

> the DIV command is used to divide EAX through a divisor. The dividend is always EAX, the result is stored in EAX and the modulus is stored in EDX.
>
> **Syntax:**
> DIV divisor

IDIV - Integer division. Signed division and may set C/O/Z flags

Syntax:
IDIV divisor

IMUL - integer multiplication

Syntax:
IMUL value
IMUL dest, value, value
IMUL dest, value

INC - increment, opposite of DEC instruction (value = value +1)

Syntax:
INC register

INT - the INT command generates a call to an interrupt handler

JUMPS - there are a variety of jumps, but the most common and important jumps are;

JE - jump if equal
JG - jump if greater
JGE - jump if greater or equal
JL - jump if lesser
JLE - jump if less or equal
JMP - jump always
JNE - jump if not equal
JNZ - jump if not zero
JZ - jump if zero

LEA - Load Effective Address

Syntax:
LEA destination, source

MOV - move copies the value from the source to the destination

Syntax:
MOV destination, source

MUL - multiply is the same as IMUL but it multiplies unsigned

Syntax:
MUL value

NOP - no operation does nothing

Syntax:
NOP

OR - logical inclusive OR

Syntax:
OR destination, source

POP - the POP instruction loads the value of the byte/word/dword pointer (ESP) and puts it into the destination.

Syntax:
POP destination

PUSH - the PUSH instruction stores a value on the stack and decreases it by the size of the operand that was pushed, so that the ESP points to the value that was PUSHed.

Syntax:
PUSH operand

REP - repeat following string instruction. Common uses are REPE(repeat if equal), REPZ (repeat if zero), REPNE (repeat if nonequal), and REPNZ (repeat if non-zero)

Syntax:
REP ins
Where ins is a string operation

RET – return

Syntax:
RET digit

SUB - subtraction. Is the opposite of ADD command. Subtracts the value of the source from the value of destination and stores the result in destination

Syntax:
SUB destination, source

TEST - it performs a logical AND but does not store the value

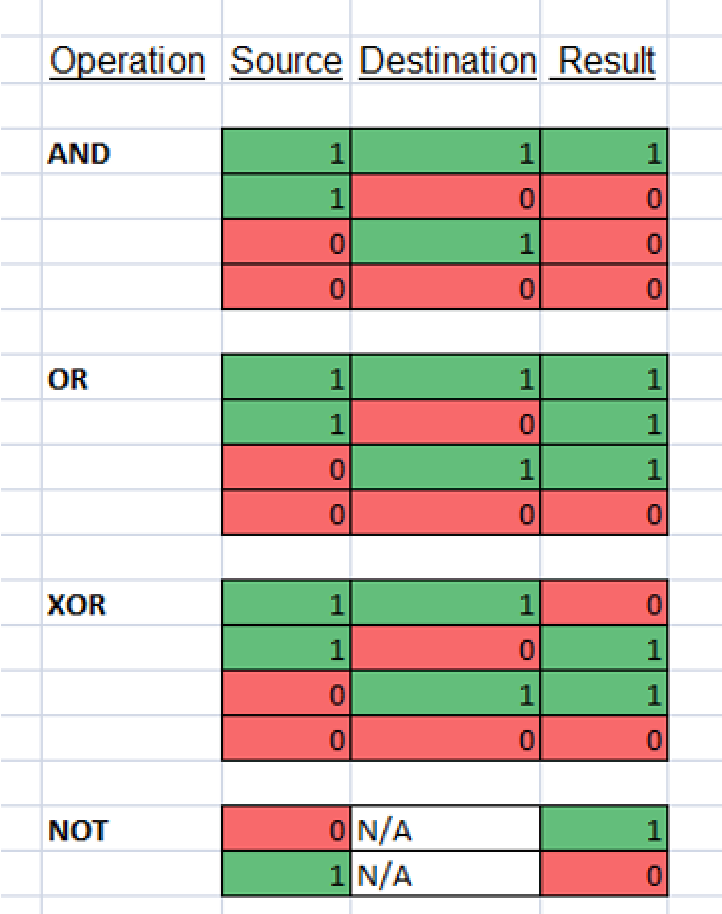
Syntax:
TEST operand1 , operand2

XOR - the XOR instruction connects two values using logical exclusive OR

Syntax:
XOR destination, source

Logical Operations

The table below summarizes the logical operations displaying the results of AND, OR, NOT and XOR when the source or destination is a 1 or 0.



Operation	Source	Destination	Result
AND	1	1	1
	1	0	0
	0	1	0
	0	0	0
OR	1	1	1
	1	0	1
	0	1	1
	0	0	0
XOR	1	1	0
	1	0	1
	0	1	1
	0	0	0
NOT	0	N/A	1
	1	N/A	0

Reverse Engineering Malware, Part 3: IDA Pro Introduction

There are many tools available for reverse engineering, but one disassembler stands alone. Nearly everyone in this industry uses **IDA Pro** to some extent. IDA Pro is a disassembler capable of taking binary programs where we don't have the source code and creating maps and multiple modes of understanding the binaries. It takes source code and represents it as assembler code, so that we can better understand how the original code works. IDA Pro also has a a debugger, but we will focus primarily on its disassembly capabilities in this course.

IDA (Interactive Disassembly) Pro was first developed by Ilfak Guilfanov and sold now by his Leige, Belgium based firm, Hex-Rays. IDA Pro comes in a Windows version (which we will be using here) as well as Linux and MacOS versions.

Let's get started with IDA!

Step #1 Download and Install

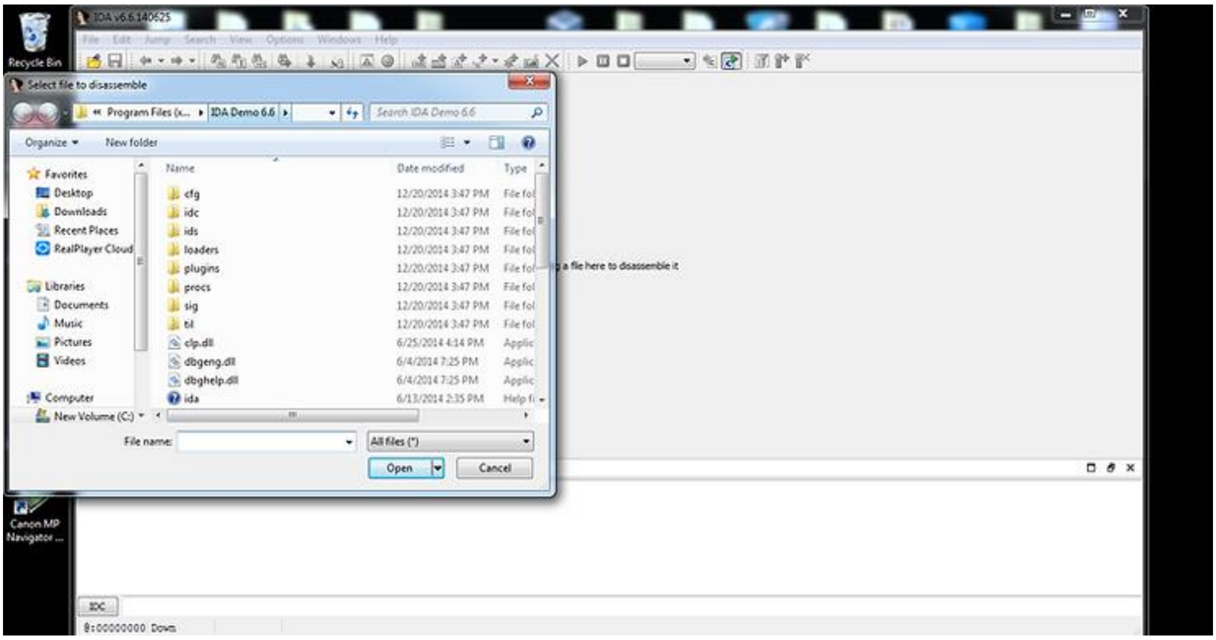
IDA Pro is commercial software, but you can download either the free version or the demo/evaluation version for this course. These versions have some limitations such as;

(1) they will only work on x86 and ARM platforms
(2) they will only work on PE/ELF/Macho-0 formats
(3) you can not save your results and it may time out
(4) a few other limitations.

After downloading IDA Pro, accepting the license agreement, installing Python 2.7, and installing Microsoft Visual C++, IDA pro will install to your system. It should now be in your programs at the Start button in Windows. Locate it and click on the icon. When you do so, IDA will start up with a screen like below. Click on "New".

Step #2 Load a PE File

Since we are working with the demo version, we can only use Portable Executable (PE) files. We can now drag and drop a file into the working center window or click on **File -> Open**.



Name	Date modified	Type
cfg	12/20/2014 3:47 PM	File fol
idc	12/20/2014 3:47 PM	File fol
ids	12/20/2014 3:47 PM	File fol
loaders	12/20/2014 3:47 PM	File fol
plugins	12/20/2014 3:47 PM	File fol
procs	12/20/2014 3:47 PM	File fol
sig	12/20/2014 3:47 PM	File fol
til	12/20/2014 3:47 PM	File fol
clp.dll	6/25/2014 4:14 PM	Applic
dbgeng.dll	6/4/2014 7:25 PM	Applic
dbghelp.dll	6/4/2014 7:25 PM	Applic
ida	6/13/2014 2:35 PM	Help fi

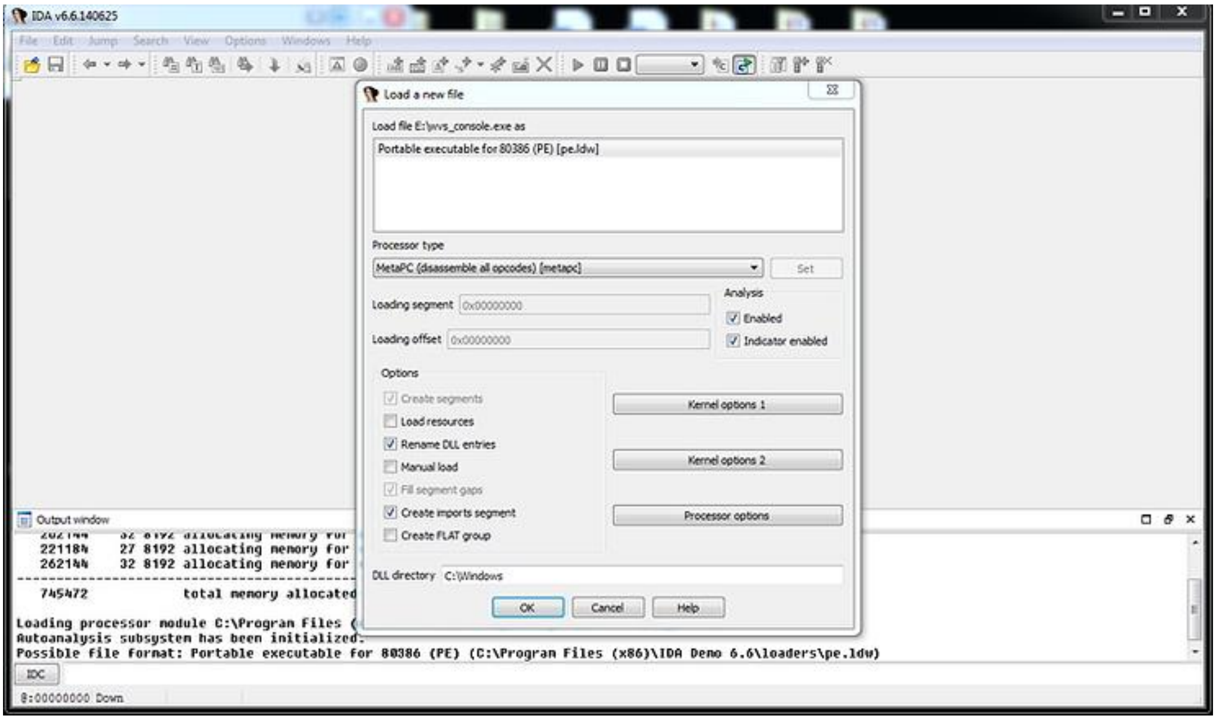
File name:

All files (*)

Open Cancel

```
8:00000000 Down
```

After selecting a file to disassemble and analyze, the window below will pop up. As you can see, IDA was able to automatically determine the type of file (portable executable) and processor type (x86). Click on "OK."



Load file E:\wvs_console.exe as

Portable executable for 80386 (PE) [pe.ldw]

Processor type: MetaPC (disassemble all opcodes) [metapc] ▾ Set

Loading segment: 0x00000000

Loading offset: 0x00000000

Analysis
☑ Enabled
☑ Indicator enabled

Options
☑ Create segments
☐ Load resources
☑ Rename DLL entries
☐ Manual load
☑ Fill segment gaps
☑ Create imports segment
☐ Create FLAT group

Kernel options 1

Kernel options 2

Processor options

DLL directory: C:\Windows

OK Cancel Help

```
   262144    32 8192 allocating memory for
   221184    27 8192 allocating memory for
   262144    32 8192 allocating memory for
------------------------------------------
   745472           total memory allocated

Loading processor module C:\Program Files (
Autoanalysis subsystem has been initialized.
Possible file format: Portable executable for 80386 (PE) (C:\Program Files (x86)\IDA Demo 6.6\loaders\pe.ldw)
```

```
8:00000000 Down
```

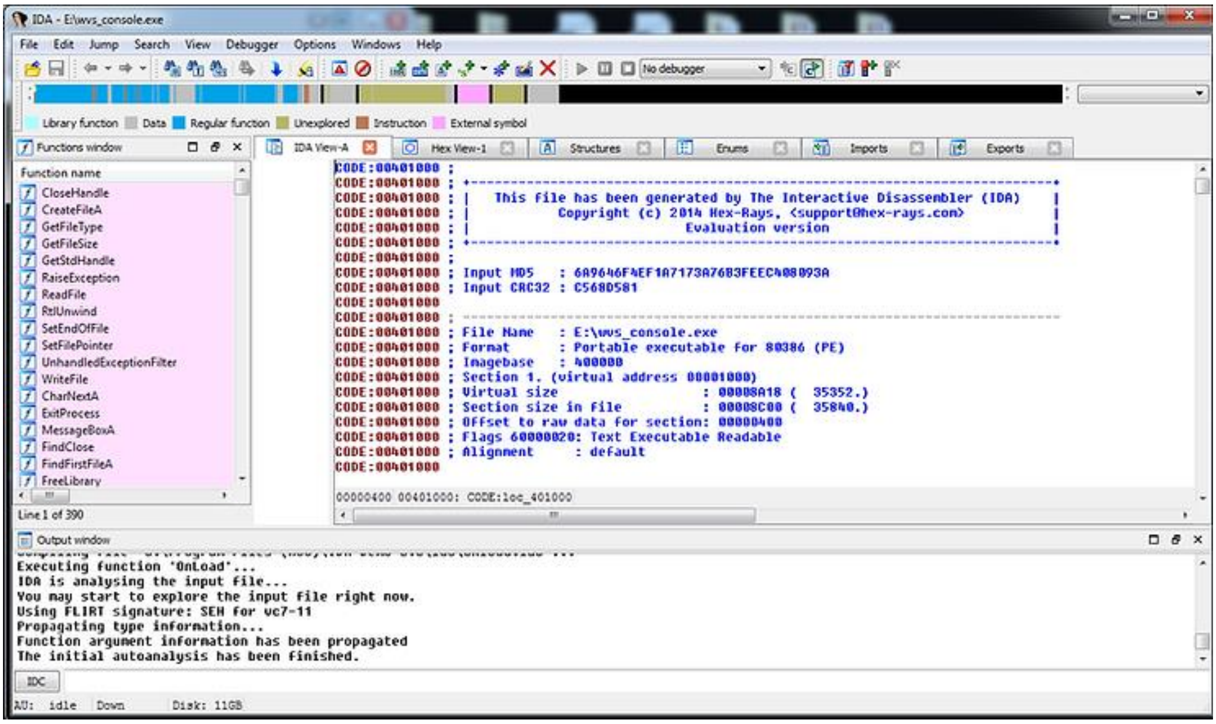
When IDA begins its disassembly and analysis, it analyzes the entire file and places the information into a database. This database has four files:

1. .id0 – contains contents of B-tree-style database
2. .id1 – contains flags that describe each program byte
3. .nam – contains index information related to named program locations
4. .til – contains information about local type definitions

Whenever you go to close IDA, it will ask you whether you want to save these database files. If you do, these four files will be archived into a single IDB file. When people refer to the IDA database, this is what they are referring to. These files will be saved and available to you at any time. You will see these files saved in the same directory as the file you are analyzing.

Step #3 Start the Disassembly

In this lab, I will be using small .exe file that is part of the Acunetix Web Vulnerability scanner. Its not malware, but it makes a good beginner demo. You can use any portable .exe (PE) that is 32-bit, so the demo version of IDA Pro can disassemble it. When we open it, IDA Pro begins its disassembly process and displays the information like in the screenshot below.



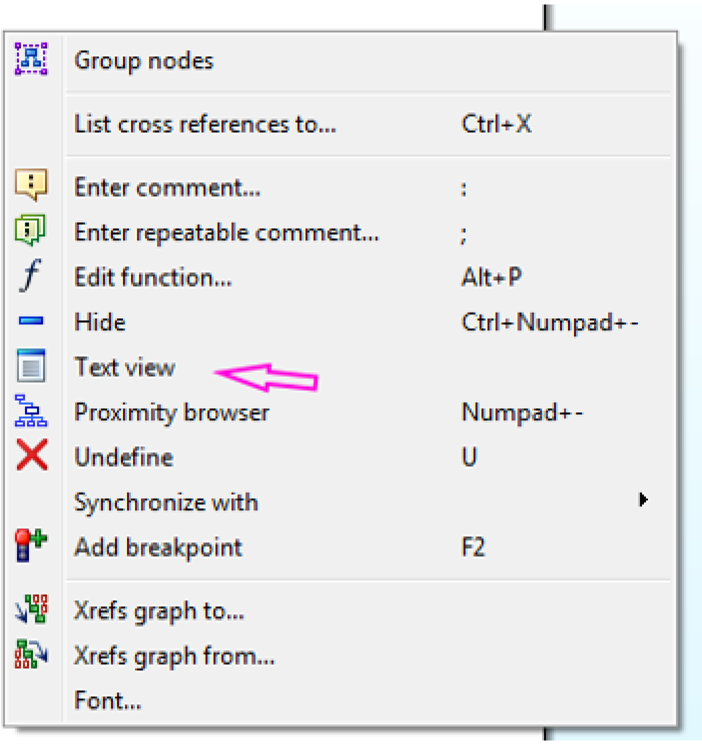
```
CODE:00401000 ;
CODE:00401000 ; +-------------------------------------------------------------------------+
CODE:00401000 ; |   This file has been generated by The Interactive Disassembler (IDA)    |
CODE:00401000 ; |          Copyright (c) 2014 Hex-Rays, <support@hex-rays.com>            |
CODE:00401000 ; |                          Evaluation version                             |
CODE:00401000 ; +-------------------------------------------------------------------------+
CODE:00401000 ;
CODE:00401000 ; Input MD5   : 6A9646F4EF1A7173A76B3FEEC408093A
CODE:00401000 ; Input CRC32 : C568D581
CODE:00401000
CODE:00401000 ; ---------------------------------------------------------------------------
CODE:00401000 ; File Name   : E:\wvs_console.exe
CODE:00401000 ; Format      : Portable executable for 80386 (PE)
CODE:00401000 ; Imagebase   : 400000
CODE:00401000 ; Section 1. (virtual address 00001000)
CODE:00401000 ; Virtual size                  : 00008A18 (  35352.)
CODE:00401000 ; Section size in file          : 00008C00 (  35840.)
CODE:00401000 ; Offset to raw data for section: 00000400
CODE:00401000 ; Flags 60000020: Text Executable Readable
CODE:00401000 ; Alignment     : default
CODE:00401000

00000400 00401000: CODE:loc_401000
```

```
Executing function 'OnLoad'...
IDA is analysing the input file...
You may start to explore the input file right now.
Using FLIRT signature: SEH for vc7-11
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.
```

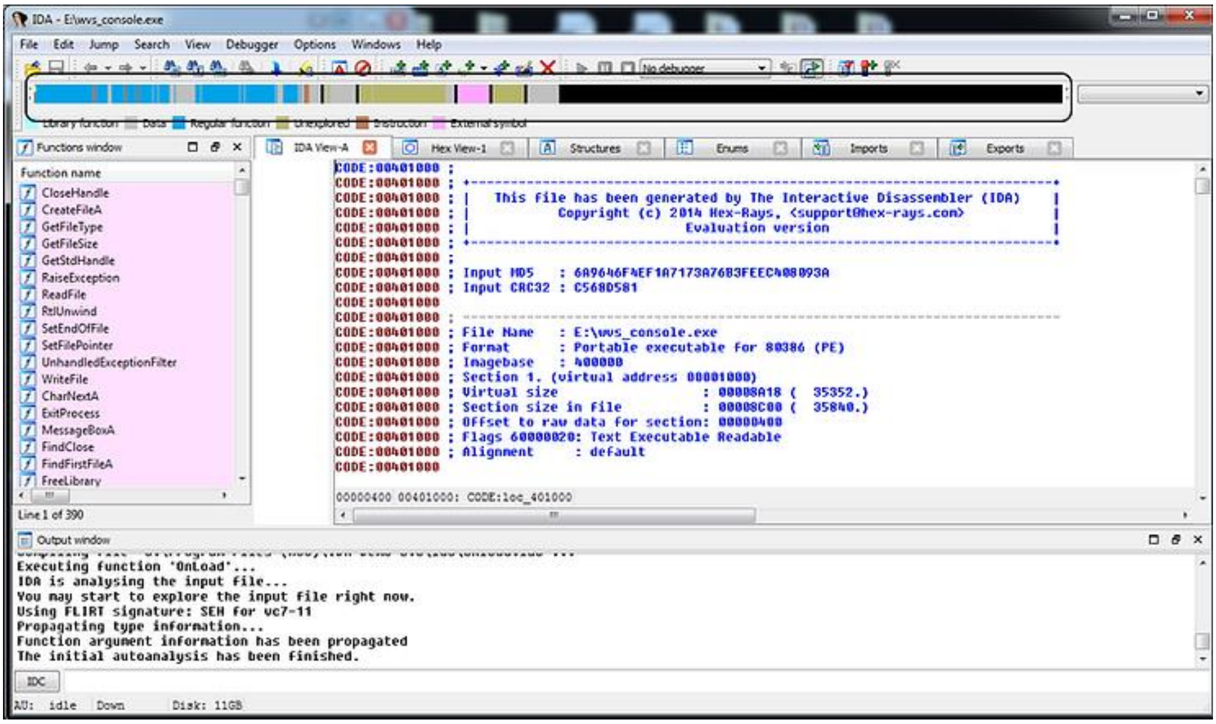
```
AU:  idle   Down     Disk: 11GB
```

As you can see above, IDA provides us with some basic info in the IDA View tab. If we scroll down the IDA View, we can see **every line of code**. This is where we will do most of our work when we begin malware disassembly and analysis.

If we right-click, it displays the window shown below. Note that we can select Text View or any number of other options while in the IDA View. When we begin our analysis later in the course, we will be setting breakpoints in the code, F2.



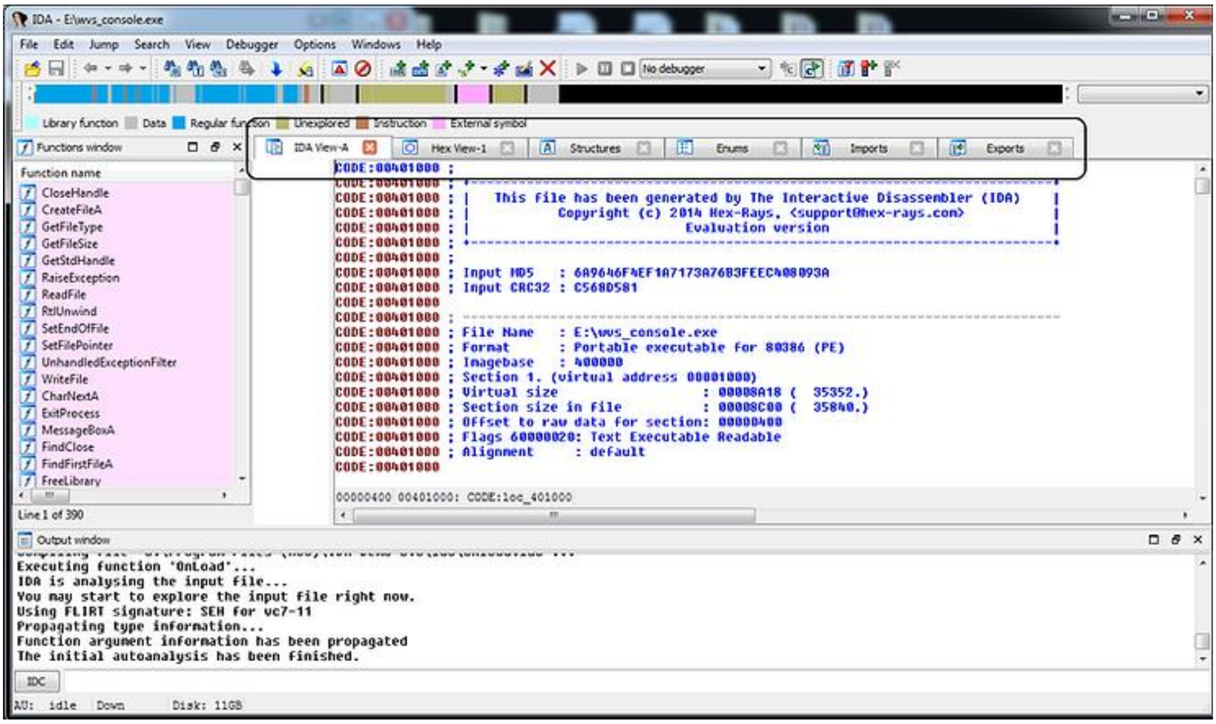
The colorful bar above this IDA View represents the memory that the file is occupying. It color codes for the different parts of the program that are stored in each part of memory. If we right-click any part of the memory bar, we can zoom in to that segment of the code stored in memory. We are capable of zooming in right down to the single byte level.



```
CODE:00401000 ;
CODE:00401000 ; +-------------------------------------------------------------------------+
CODE:00401000 ; |   This file has been generated by The Interactive Disassembler (IDA)    |
CODE:00401000 ; |          Copyright (c) 2014 Hex-Rays, <support@hex-rays.com>            |
CODE:00401000 ; |                          Evaluation version                             |
CODE:00401000 ; +-------------------------------------------------------------------------+
CODE:00401000 ;
CODE:00401000 ; Input MD5    : 6A9646F4EF1A7173A76B3FEEC408093A
CODE:00401000 ; Input CRC32  : C568D581
CODE:00401000
CODE:00401000 ; ---------------------------------------------------------------------------
CODE:00401000 ; File Name    : E:\wvs_console.exe
CODE:00401000 ; Format       : Portable executable for 80386 (PE)
CODE:00401000 ; Imagebase    : 400000
CODE:00401000 ; Section 1. (virtual address 00001000)
CODE:00401000 ; Virtual size                  : 00008A18 (  35352.)
CODE:00401000 ; Section size in file          : 00008C00 (  35840.)
CODE:00401000 ; Offset to raw data for section: 00000400
CODE:00401000 ; Flags 60000020: Text Executable Readable
CODE:00401000 ; Alignment     : default
CODE:00401000

00000400 00401000: CODE:loc_401000
```

```
Executing function 'OnLoad'...
IDA is analysing the input file...
You may start to explore the input file right now.
Using FLIRT signature: SEH for vc7-11
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.
```

We can view the file from many different perspectives by selecting any of these views available. These include the IDA View (as seen here), Hex View, Structures, Enums, Imports, and finally, Exports. By clicking on any one of those tabs, it will give us that particular view of the code (see Import in Step 5 below).

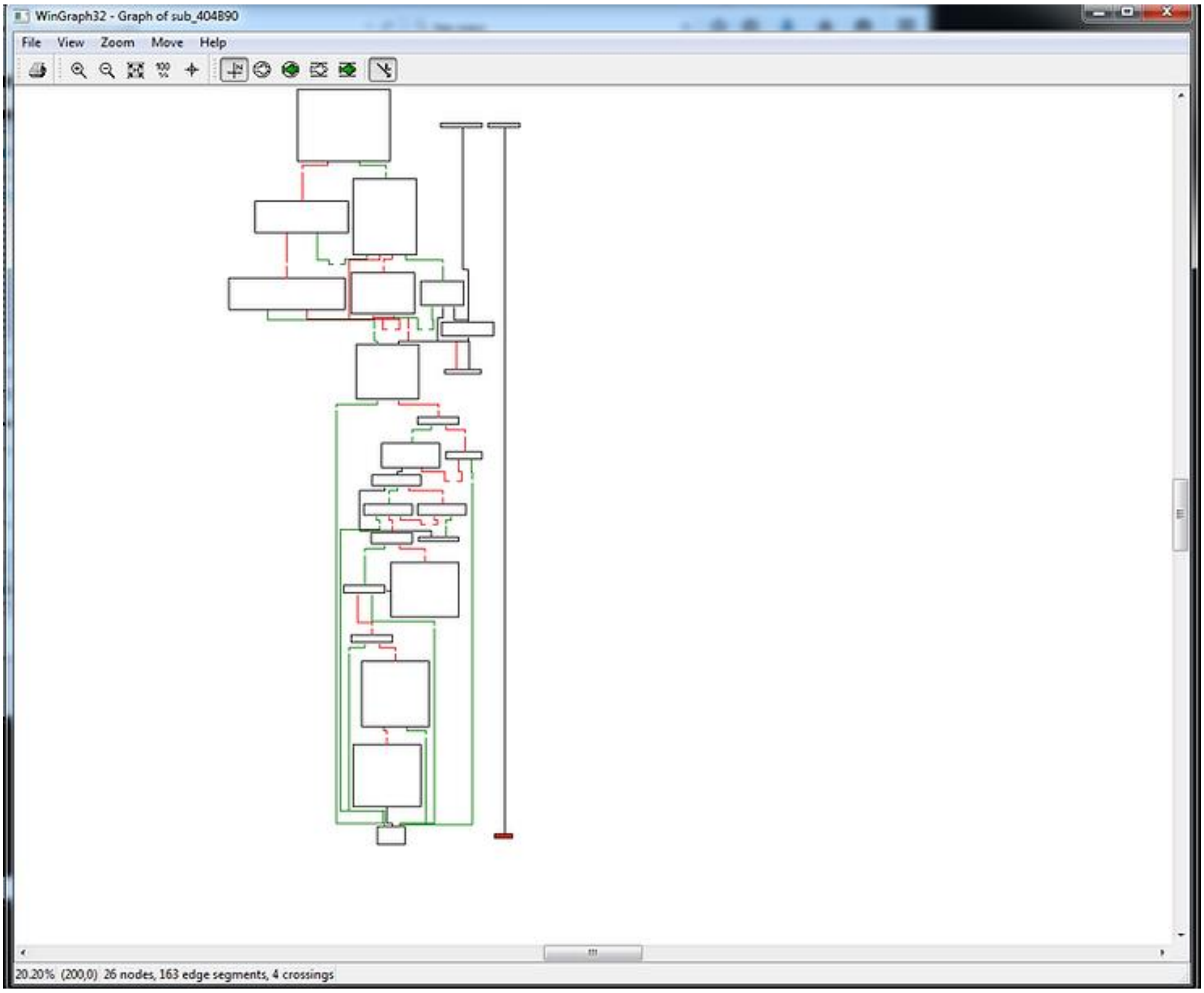


```
CODE:00401000 ;
CODE:00401000 ; +-------------------------------------------------------------------------+
CODE:00401000 ; |   This file has been generated by The Interactive Disassembler (IDA)    |
CODE:00401000 ; |          Copyright (c) 2014 Hex-Rays, <support@hex-rays.com>            |
CODE:00401000 ; |                          Evaluation version                             |
CODE:00401000 ; +-------------------------------------------------------------------------+
CODE:00401000 ;
CODE:00401000 ; Input MD5   : 6A9646F4EF1A7173A76B3FEEC408093A
CODE:00401000 ; Input CRC32 : C568D581
CODE:00401000
CODE:00401000 ; ---------------------------------------------------------------------------
CODE:00401000 ; File Name   : E:\wvs_console.exe
CODE:00401000 ; Format      : Portable executable for 80386 (PE)
CODE:00401000 ; Imagebase   : 400000
CODE:00401000 ; Section 1. (virtual address 00001000)
CODE:00401000 ; Virtual size                  : 00008A18 (  35352.)
CODE:00401000 ; Section size in file          : 00008C00 (  35840.)
CODE:00401000 ; Offset to raw data for section: 00000400
CODE:00401000 ; Flags 60000020: Text Executable Readable
CODE:00401000 ; Alignment     : default
CODE:00401000
```

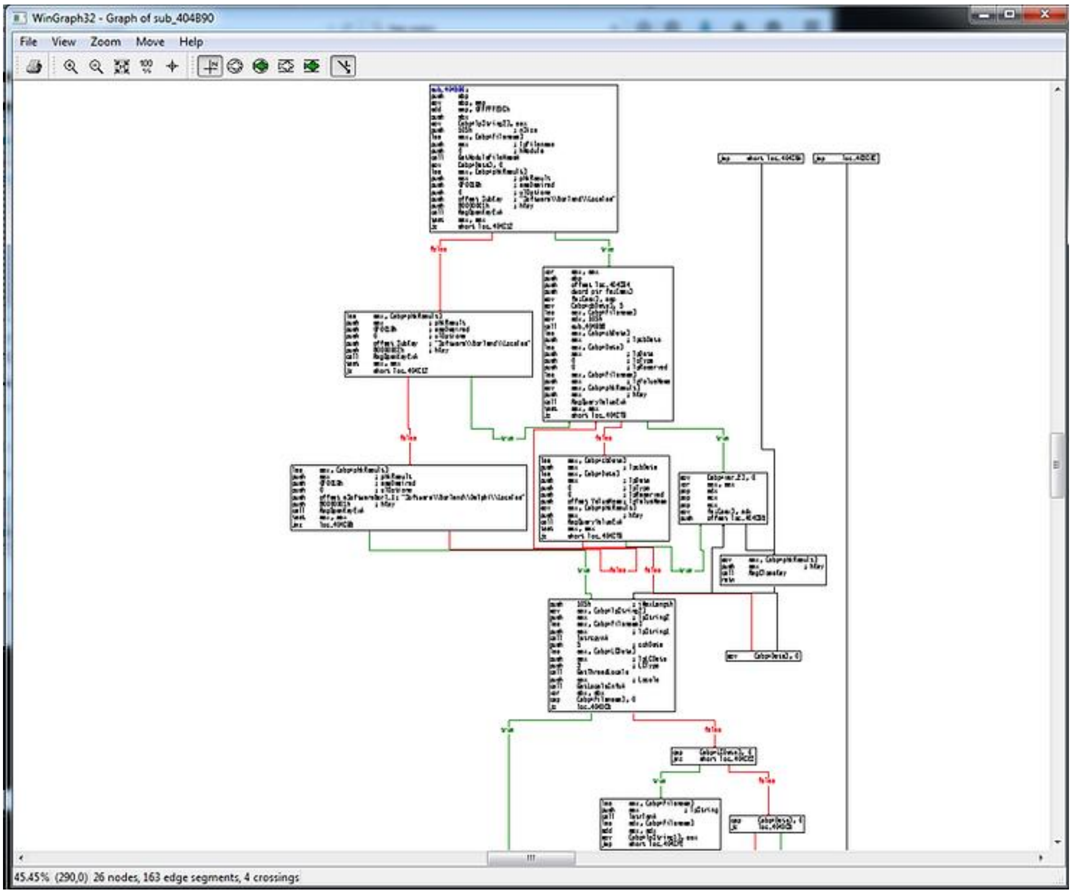
```
00000400 00401000: CODE:loc_401000
```

```
Executing function 'OnLoad'...
IDA is analysing the input file...
You may start to explore the input file right now.
Using FLIRT signature: SEH for vc7-11
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.
```

One of the most interesting and enlightening views that IDA can provide us is the flow chart. The flow chart graphically displays the flow of the execution of the file, making it easier to understand. We can open it by going to the top menu bar and clicking on **View -> Graphs -> Flow Chart**. It will open a Flow Chart of the code similar to that below.

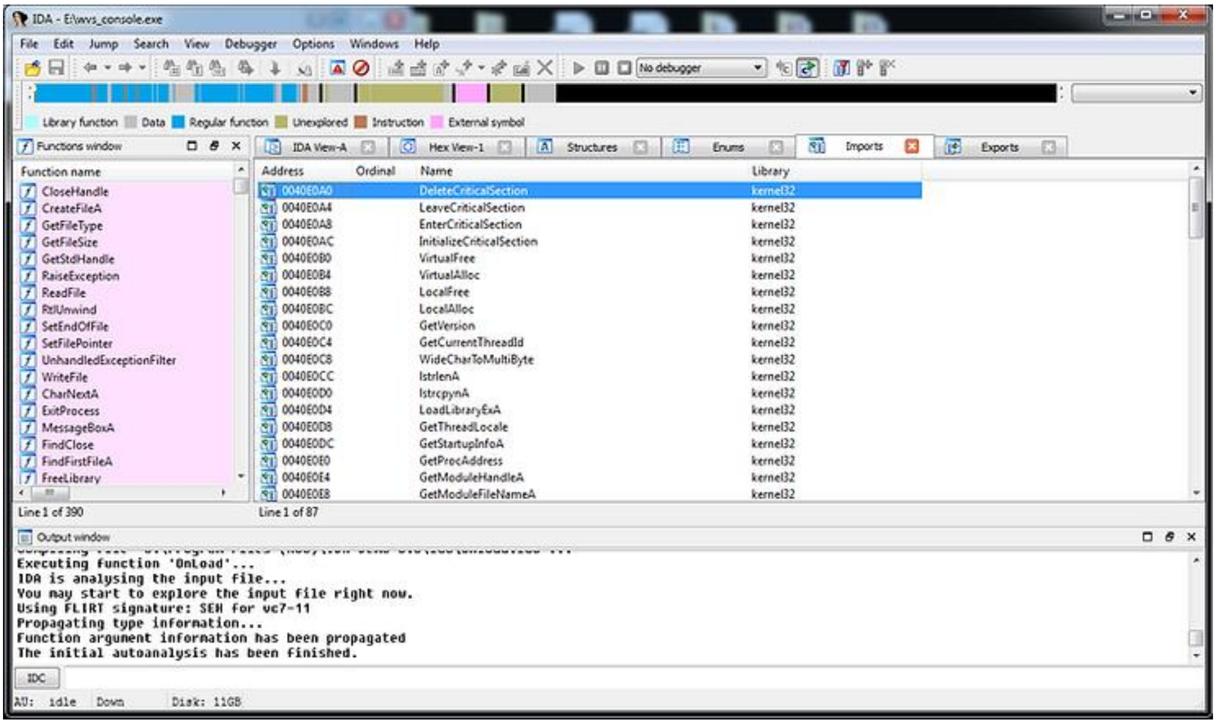


We can zoom in by going to the View menu at the top of the flow chart to get greater detail. In this way, we can view the program flow from each register, subroutine, and function.



Step 5: Show Imports

When we select the Imports view, IDA will show us all the modules that the .exe imported. These imports can give us clues as to the origin of the malware.



Function name
- CloseHandle
- CreateFileA
- GetFileType
- GetFileSize
- GetStdHandle
- RaiseException
- ReadFile
- RtlUnwind
- SetEndOfFile
- SetFilePointer
- UnhandledExceptionFilter
- WriteFile
- CharNextA
- ExitProcess
- MessageBoxA
- FindClose
- FindFirstFileA
- FreeLibrary

Line 1 of 390

Address	Ordinal	Name	Library
0040E0A0		DeleteCriticalSection	kernel32
0040E0A4		LeaveCriticalSection	kernel32
0040E0A8		EnterCriticalSection	kernel32
0040E0AC		InitializeCriticalSection	kernel32
0040E0B0		VirtualFree	kernel32
0040E0B4		VirtualAlloc	kernel32
0040E0B8		LocalFree	kernel32
0040E0BC		LocalAlloc	kernel32
0040E0C0		GetVersion	kernel32
0040E0C4		GetCurrentThreadId	kernel32
0040E0C8		WideCharToMultiByte	kernel32
0040E0CC		lstrlenA	kernel32
0040E0D0		lstrcpynA	kernel32
0040E0D4		LoadLibraryExA	kernel32
0040E0D8		GetThreadLocale	kernel32
0040E0DC		GetStartupInfoA	kernel32
0040E0E0		GetProcAddress	kernel32
0040E0E4		GetModuleHandleA	kernel32
0040E0E8		GetModuleFileNameA	kernel32

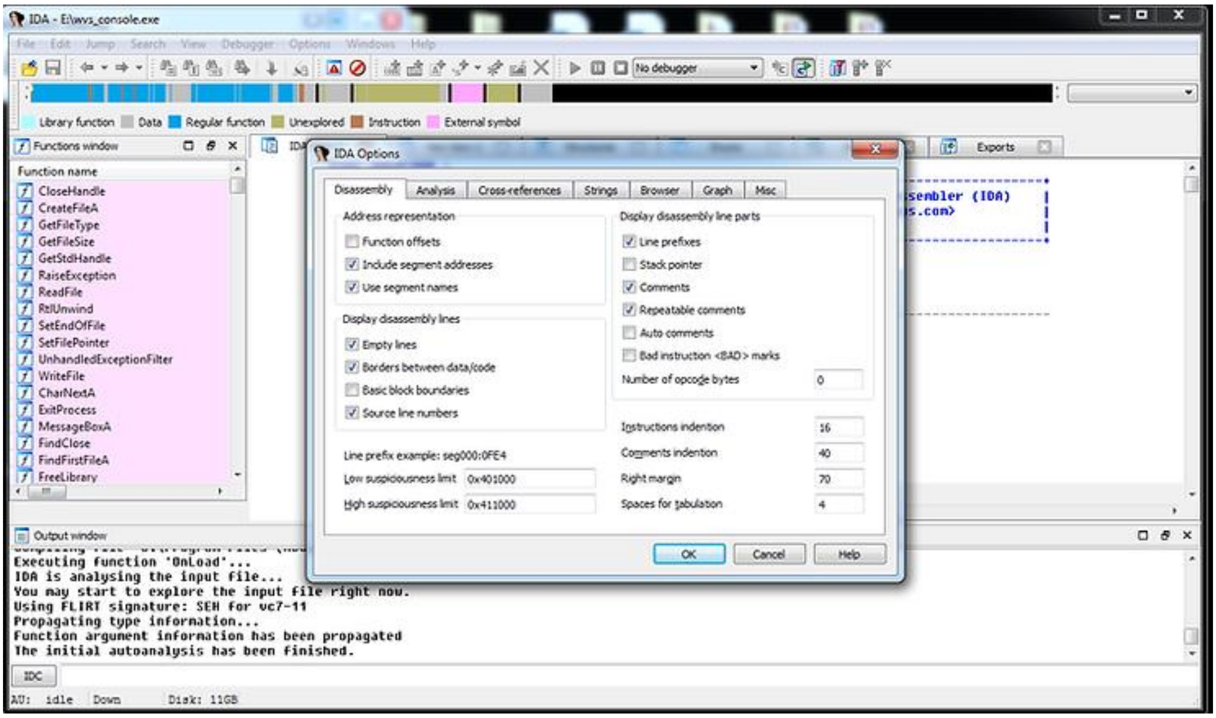
Line 1 of 87

```
Executing function 'OnLoad'...
IDA is analysing the input file...
You may start to explore the input file right now.
Using FLIRT signature: SEH for vc7-11
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.
```

AU: idle Down Disk: 11GB

Step 6: Customize the Analysis

Finally, we can begin to customize what and how IDA displays the code to by going to **Options -> General**. A window like that shown in the screenshot below will enable us to customize our analysis.



Address representation

☐ Function offsets
☑ Include segment addresses
☑ Use segment names

Display disassembly lines

☑ Empty lines
☑ Borders between data/code
☐ Basic block boundaries
☑ Source line numbers

Line prefix example: seg000:0FE4

Low suspiciousness limit: 0x401000
High suspiciousness limit: 0x411000

Display disassembly line parts

☑ Line prefixes
☐ Stack pointer
☑ Comments
☑ Repeatable comments
☐ Auto comments
☐ Bad instruction <BAD> marks

Number of opcode bytes: 0

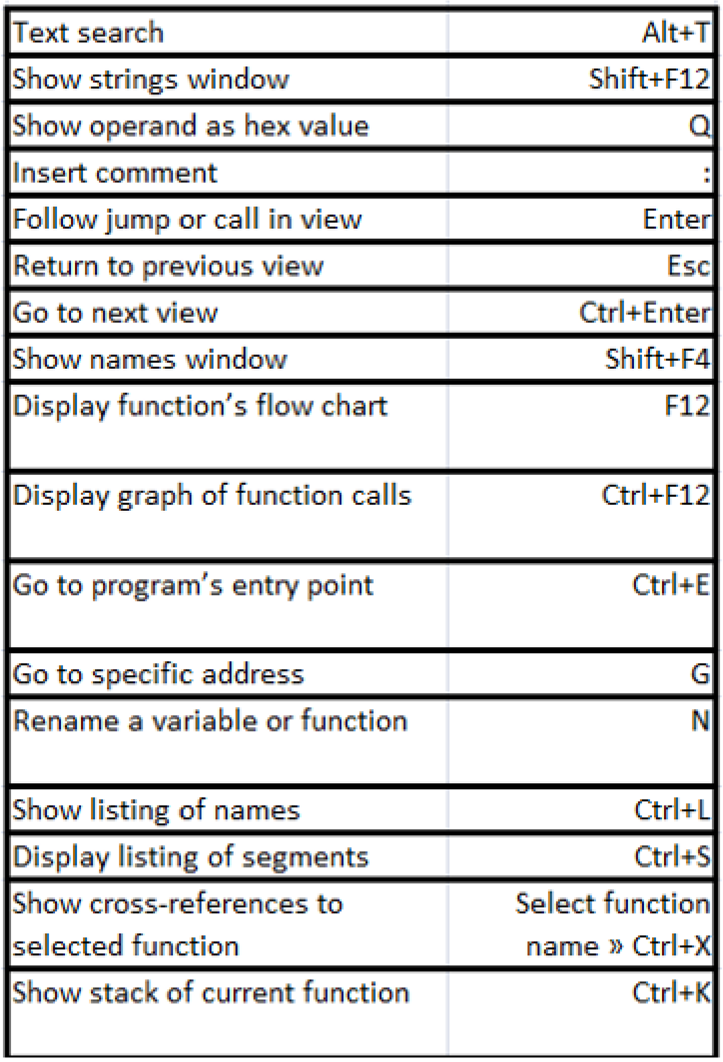
Instructions indention: 16
Comments indention: 40
Right margin: 70
Spaces for tabulation: 4

```
Executing function 'OnLoad'...
IDA is analysing the input file...
You may start to explore the input file right now.
Using FLIRT signature: SEH for vc7-11
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.
```

```
AU:  idle   Down      Disk: 11GB
```

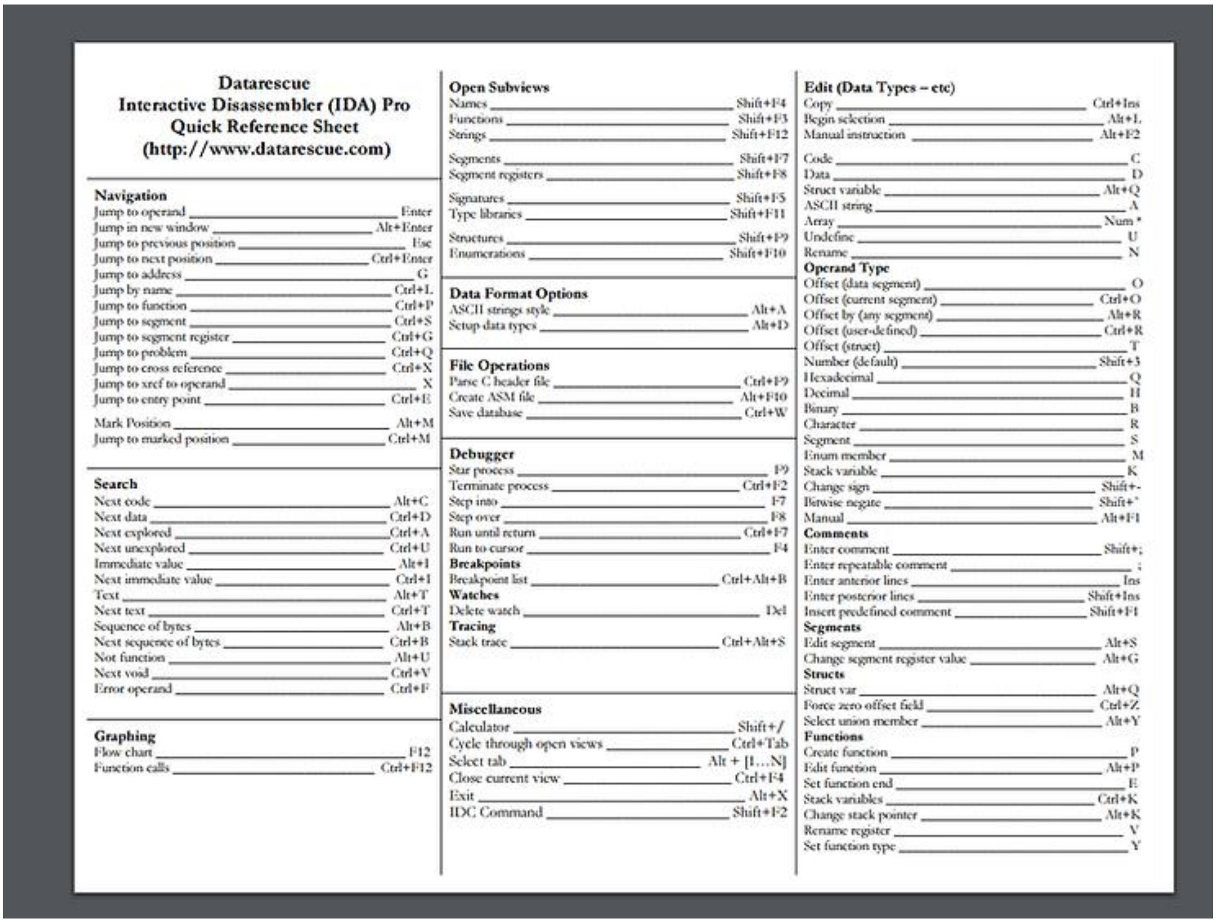
Although this far from a complete and thorough introduction to IDA Pro, we are ready to use IDA Pro for some malware analysis! I will introduce additional concepts and techniques as you need them throughout the course.

Before you proceed, I think it is useful to introduce you to a few key commands and shortcuts in IDA Pro.



Text search	Alt+T
Show strings window	Shift+F12
Show operand as hex value	Q
Insert comment	:
Follow jump or call in view	Enter
Return to previous view	Esc
Go to next view	Ctrl+Enter
Show names window	Shift+F4
Display function's flow chart	F12
Display graph of function calls	Ctrl+F12
Go to program's entry point	Ctrl+E
Go to specific address	G
Rename a variable or function	N
Show listing of names	Ctrl+L
Display listing of segments	Ctrl+S
Show cross-references to selected function	Select function name » Ctrl+X
Show stack of current function	Ctrl+K

Also, please find a complete IDA Pro Quick Reference sheet courtesy of the good folks at www.datarescue.com (the original sales and marketing firm for IDA Pro).



Datarescue
Interactive Disassembler (IDA) Pro
Quick Reference Sheet
(http://www.datarescue.com)

Navigation

Jump to operand	Enter
Jump in new window	Alt+Enter
Jump to previous position	Esc
Jump to next position	Ctrl+Enter
Jump to address	G
Jump by name	Ctrl+L
Jump to function	Ctrl+P
Jump to segment	Ctrl+S
Jump to segment register	Ctrl+G
Jump to problem	Ctrl+Q
Jump to cross reference	Ctrl+X
Jump to xref to operand	X
Jump to entry point	Ctrl+E
Mark Position	Alt+M
Jump to marked position	Ctrl+M

Search

Next code	Alt+C
Next data	Ctrl+D
Next explored	Ctrl+A
Next unexplored	Ctrl+U
Immediate value	Alt+I
Next immediate value	Ctrl+I
Text	Alt+T
Next text	Ctrl+T
Sequence of bytes	Alt+B
Next sequence of bytes	Ctrl+B
Not function	Alt+U
Next void	Ctrl+V
Error operand	Ctrl+F

Graphing

Flow chart	F12
Function calls	Ctrl+F12

Open Subviews

Names	Shift+F4
Functions	Shift+F3
Strings	Shift+F12
Segments	Shift+F7
Segment registers	Shift+F8
Signatures	Shift+F5
Type libraries	Shift+F11
Structures	Shift+F9
Enumerations	Shift+F10

Data Format Options

ASCII strings style	Alt+A
Setup data types	Alt+D

File Operations

Parse C header file	Ctrl+F9
Create ASM file	Alt+F10
Save database	Ctrl+W

Debugger

Star process	F9
Terminate process	Ctrl+F2
Step into	F7
Step over	F8
Run until return	Ctrl+F7
Run to cursor	F4

Breakpoints

Breakpoint list	Ctrl+Alt+B

Watches

Delete watch	Del

Tracing

Stack trace	Ctrl+Alt+S

Miscellaneous

Calculator	Shift+/
Cycle through open views	Ctrl+Tab
Select tab	Alt + [1…N]
Close current view	Ctrl+F4
Exit	Alt+X
IDC Command	Shift+F2

Edit (Data Types – etc)

Copy	Ctrl+Ins
Begin selection	Alt+L
Manual instruction	Alt+F2
Code	C
Data	D
Struct variable	Alt+Q
ASCII string	A
Array	Num *
Undefine	U
Rename	N

Operand Type

Offset (data segment)	O
Offset (current segment)	Ctrl+O
Offset by (any segment)	Alt+R
Offset (user-defined)	Ctrl+R
Offset (struct)	T
Number (default)	Shift+3
Hexadecimal	Q
Decimal	H
Binary	B
Character	R
Segment	S
Enum member	M
Stack variable	K
Change sign	Shift+-
Bitwise negate	Shift+'
Manual	Alt+F1

Comments

Enter comment	Shift+;
Enter repeatable comment	;
Enter anterior lines	Ins
Enter posterior lines	Shift+Ins
Insert predefined comment	Shift+F1

Segments

Edit segment	Alt+S
Change segment register value	Alt+G

Structs

Struct var	Alt+Q
Force zero offset field	Ctrl+Z
Select union member	Alt+Y

Functions

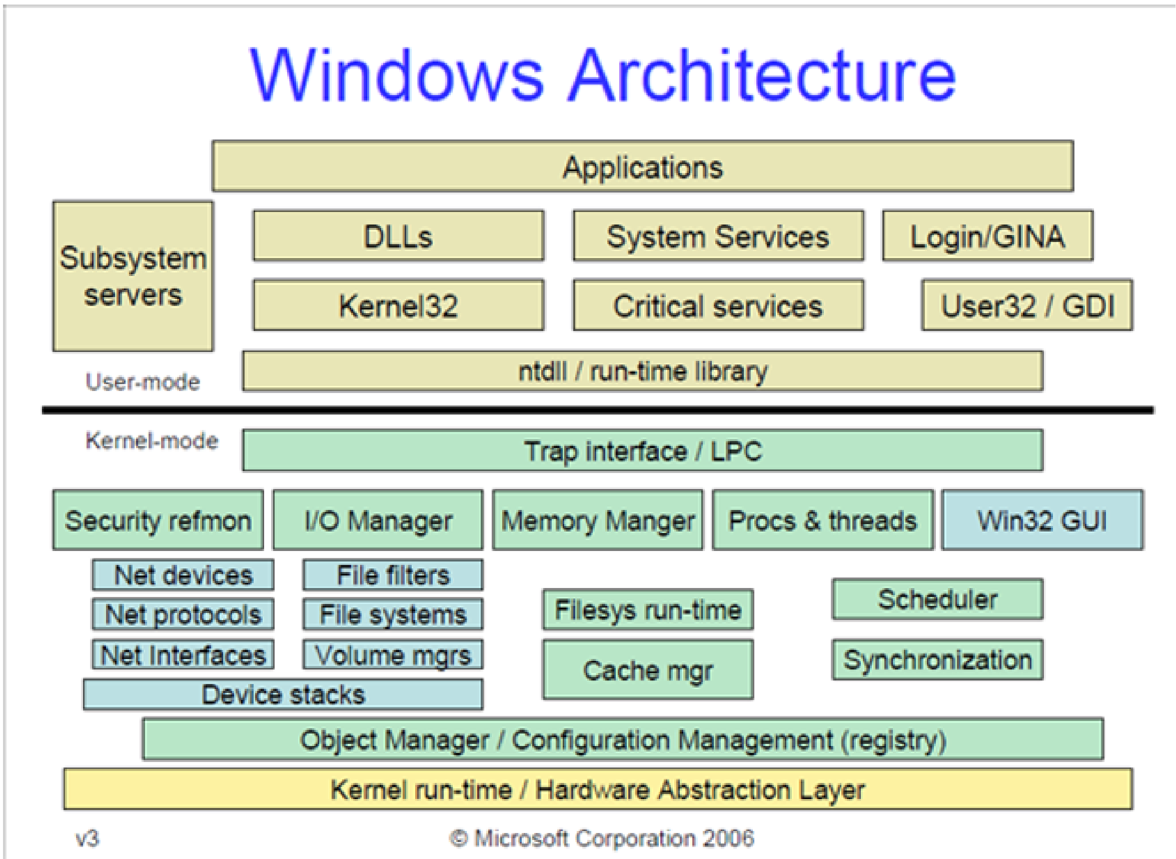
Create function	P
Edit function	Alt+P
Set function end	E
Stack variables	Ctrl+K
Change stack pointer	Alt+K
Rename register	V
Set function type	Y

Reverse Engineering Malware, Part 4: Windows Internals

In general, reverse engineering of malware is done on Windows systems. That's because despite recent inroads by Linux and the Mac OS, Windows systems still comprise over 90% of all computing systems in the world. As such, well over 90% of malware is designed to compromise Windows system. For this reason, it makes sense to focus our attention to Windows operating systems.

When reversing malware, the operating system plays a key role. All applications interact with the operating system and are tightly integrated with the OS. We can gather a significant amount of information on the malware by probing the interface between the OS and the application (malware).

To understand how malware can use and manipulate Windows then, we need to better understand the inner workings of the Windows operating system. In this article, we will examine the inner workings or Windows 32-bit systems so that we can better understand how malware can use the operating system for its malicious purposes.



Windows internals could fill several textbooks (and has), so I will attempt to just cover the most important topics and only in a cursory way. I hope to leave you with enough information though, that you can effectively reverse the malware in the following articles.

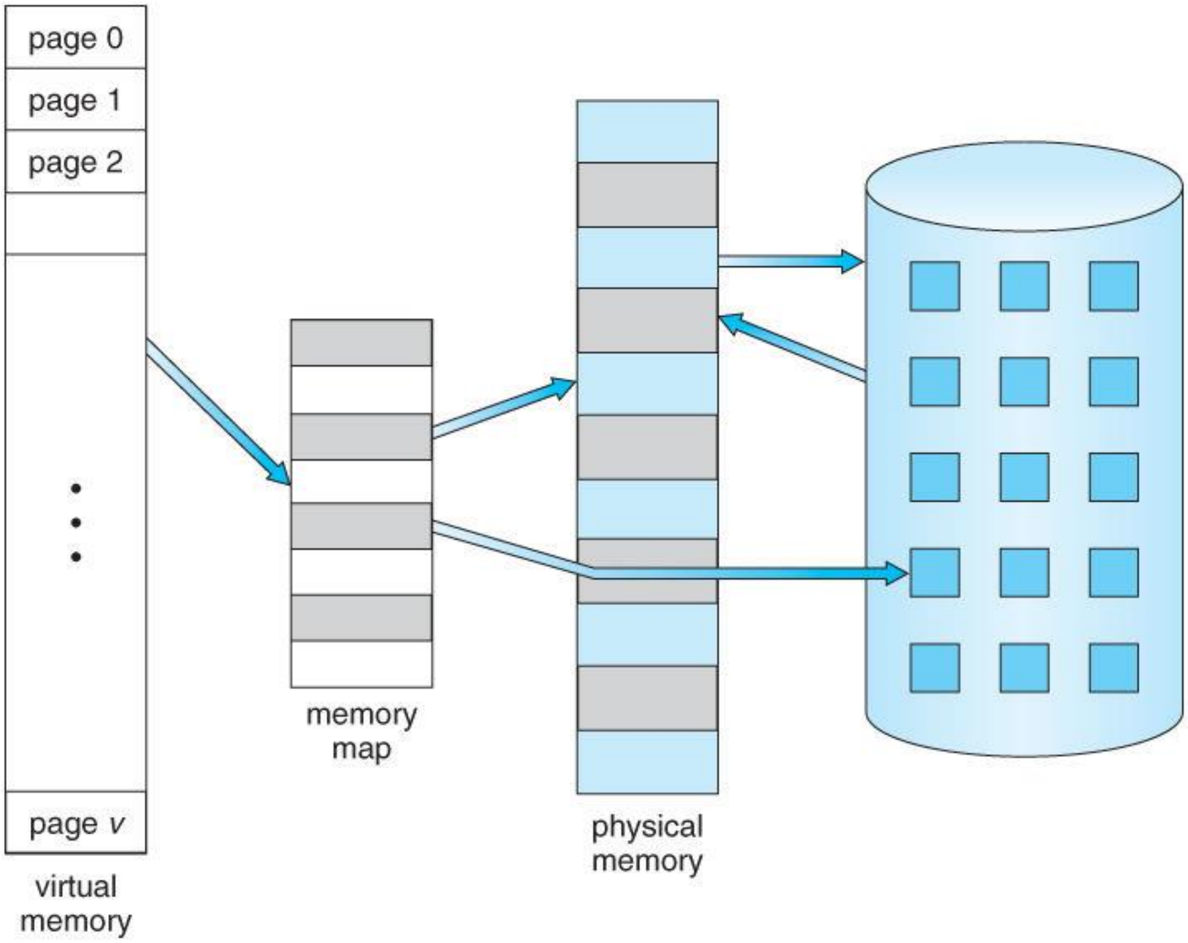
Virtual Memory

Virtual memory is the idea that instead of software directly accessing the physical memory, the CPU and the operating system create an invisible layer between the software and the physical memory.

The OS creates a table that the CPU consults called the **page table** that directs the process to the location of the physical memory that it should use.

Processors divide memory into pages

Pages are fixed sized chunks of memory. Each entry in the page table references one page of memory. In general, 32 -bit processors use 4k sized pages with some exceptions.



Kernel v User Mode

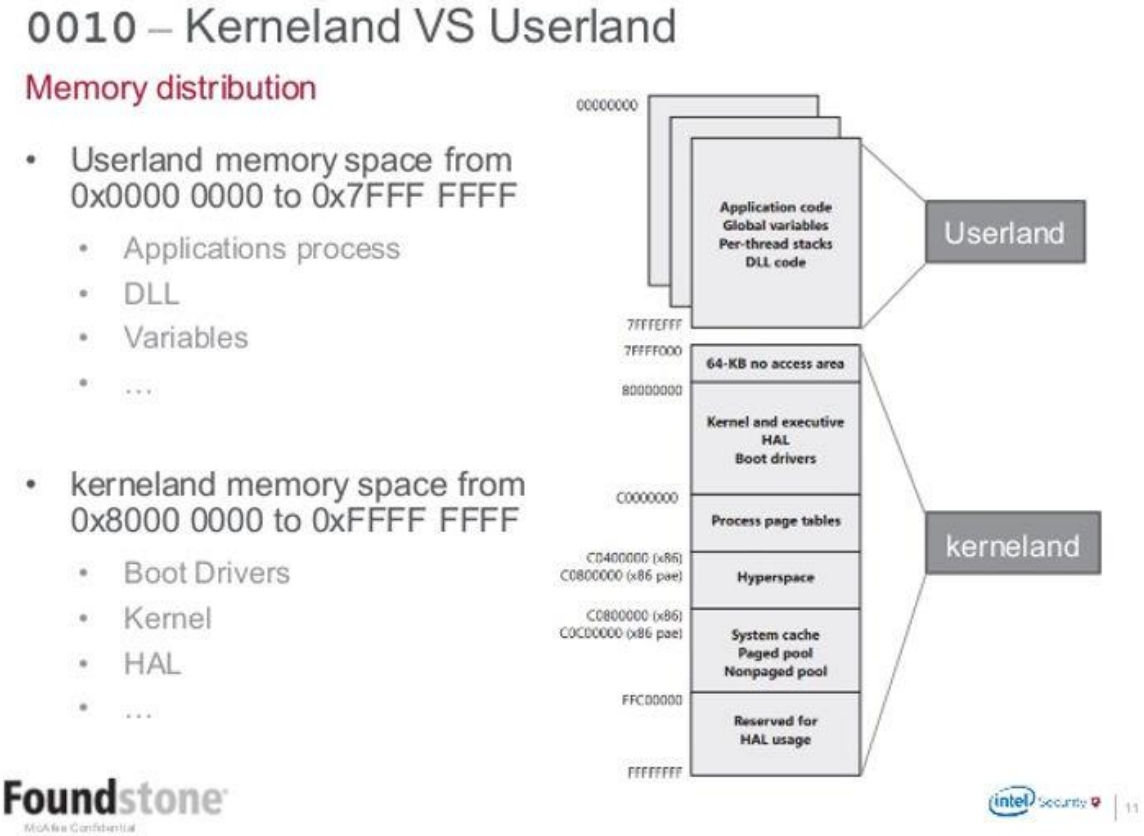
Having a page table enables the processor to enforce rules on how memory will be accessed. For instance, page table entries often have flags that determine whether the page can be accessed from a non-privileged mode (user mode).

In this way, the operating system's code can reside inside the process's address space without concern that it will be accessed by non-privileged processes. This protects the operating system's sensitive data.

This distinction between privileged vs. non-privileged mode becomes kernel (privileged) and non-privileged (user) modes.

Kernel memory Space

The kernel reserves 2gb of address space for itself. This address space contains all the kernel code, including the kernel itself and any other kernel components such as device drivers.



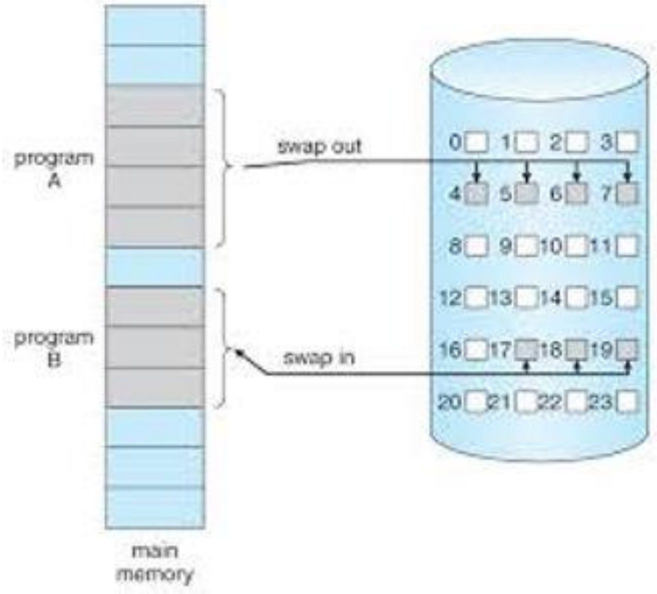
0010 – Kerneland VS Userland

Memory distribution

- Userland memory space from 0x0000 0000 to 0x7FFF FFFF
  - Applications process
  - DLL
  - Variables
  - ...
- kerneland memory space from 0x8000 0000 to 0xFFFF FFFF
  - Boot Drivers
  - Kernel
  - HAL
  - ...

Paging

Paging is the process where memory regions are temporarily flushed to the hard drive when they have not been used recently. The processor tracks the time since a page of memory was last used and the oldest is flushed. Obviously, physical memory is faster and more expensive than space on the hard drive.



The windows operating system tracks when a page was last accessed and then uses that information to locate pages that haven't been accessed in a while. Windows then flushes their content to a file. The contents of the flushed pages can then be discarded and the space used by other information. When the operating system needs to access these flushed pages, a page fault will be generated and then system then does that the information has "paged out" to a file. Then, the operating system will access the page file and pull the information back into memory to be used.

Objects and Handles

The Windows kernel manages objects using a centralized object manager component. This object manager is responsible for all kernel objects such as sections, files, and device objects, synchronization objects, processes and threads. It ONLY manages kernel objects.

GUI-related objects are managed by separate object managers that are implemented inside WIN32K.SYS.

Kernel code typically accesses objects using direct pointers to the object data structures. Applications use handles for accessing individual objects.

Handles

A handle is process specific numeric identifier which is an index into the processes private handle table. Each entry in the handle table contains a pointer to the underlying object, which is how the system associates handles with objects. Each handle entry also contains an access mask that determines which types of operations that can be performed on the object using this specific handle.

Processes

A process is really just an isolated memory address space that is used to run a program. Address spaces are created for every program to make sure that each program runs in its own address space without colliding with other processes. Inside a processes' address space the system can load code modules, but must have at least one thread running to do so.

Process Initialization

The creation of the process object and the new address space is the first step. When a new process calls the Win32 API **CreateProcess**, the API creates a process object and allocates a new memory address space for the process.

CreateProcess maps NTDLL.DLL and the program executable (the .exe file) into the newly created address space. **CreateProcess** creates the process's first thread and allocates stack space it. The processes first thread is resumed and starts running in the **LdrpInitialization** function inside NTDLL.DLL

LdrpInitialization recursively traverses the primary executable's import tables and maps them to memory every executable that is required.

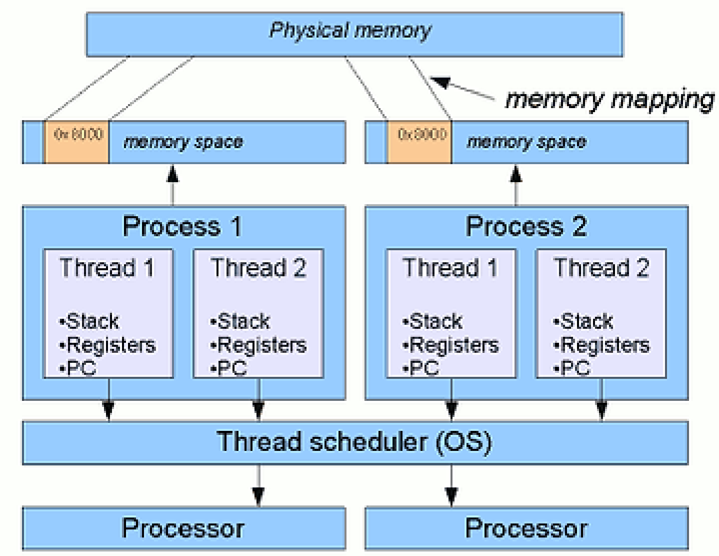
At this point, control passes into **LdrpRunInitializeRoutines**, which is an internal NTDLL routine responsible for initializing all statically linked DLL's currently loaded into the address space. The initialization process consists of a link each DLL's entry point with the **DLL_PROCESS_ATTACH** constant. Once all the DLL's are initialized, **LdrpInitialize** calls the thread's real initialization routine, which is the **BaseProcessStart** function from KERNELL32.DLL. This function in turn calls the executable's WinMain entry point, at which point the process has completed it's initialization sequence.

Threads

At ant given moment, each processor in the system is running one thread. Instead of continuing to run a single piece of code until it completes, Windows can decide to interrupt a running thread at given given time and switch to execution of another thread.

A thread is a data structure that has a CONTEXT data structure. This CONTEXT includes;

(1) the state of the processor when the thread last ran
(2) one or two memory blocks that are used for stack space
(3) stack space is used to save off current state of thread when context switched
(4) components that manage threads in windows are the scheduler and the dispatcher
(5) Deciding which thread get s to run for how long and perform context switch



Context Switch

Context switch is the thread interruption. In some cases, threads just give up the CPU on their own and the kernel doesn't have to interrupt. Every thread is assigned a quantum, which quantifies has long the the thread can run without interruption. Once the quantum expires, the thread is interrupted and other threads are allowed to run. This entire process is transparent to thread. The kernel then stores the state of the CPU registers before suspending and then restores that register state when the thread is resumed.

Win32 API

An API is a set of functions that the operating system makes available to application programs for communicating with the OS. The Win32 API is a large set of functions that make up the official low-level programming interface for Windows applications. The MFC is a common interface to the Win32 API.

The three main components of the Win 32 API are;

(1) **Kernel or Base API's**: These are the non GUI related services such as I/O, memory, object and process an d thread management

(2) **GDI API's** : these include low-level graphics services such a s those for drawing a line, displaying bitmap, etc.

(3) **USER API's** : these are the higher level GUI-related services such as window management, menus, dialog boxes, user-interface controls.

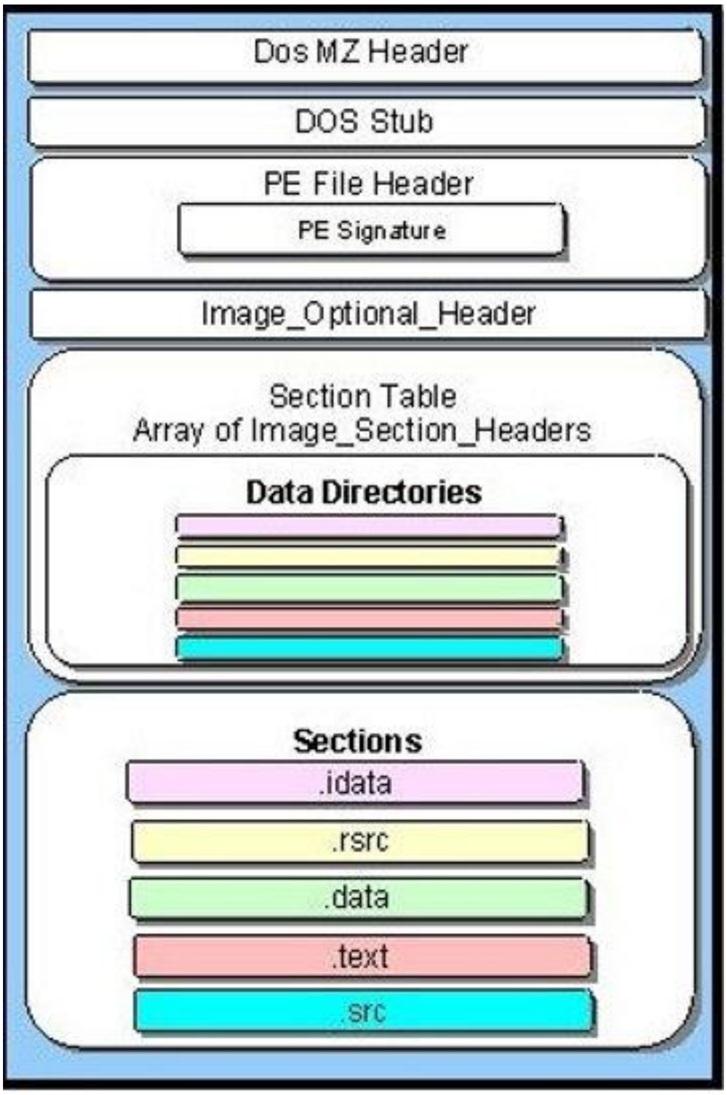
System Calls

A system call is when a user mode code needs to cal a kernel mode function. This usually happens when an application calls an operating system API. User mode code invokes a special CPU instruction that tells the processor to switch to its privileged mode and call a dispatch routine. This dispatch routine then calls the specific system function requested from user mode.

PE Format

The Windows executable format is a PE (Portable Executable). The term "portable" refers to format's versatility in numerous environments and architectures.

Executable files are relocatable. This means that they could be loaded at a different virtual address each time they are loaded. An executable must coexist with other executables that are loaded in the same memory address. Other than the main executable, every program has a certain number of additional executables loaded into its address space regardless of whether it has DLL's of its own or not.



Relocation Issues

If two executables attempt to be loaded into the same virtual space, one must be relocated to another virtual space. each executable is module is assigned a base address and if something is already there, it must be relocated.

There are never absolute memory addresses in executable headers, those only exist in the code. To make this work, whenever there is a pointer inside the executable header, it is always a relative virtual address (RVA). Think of this as simply an offset. When the file is loaded, it is assigned a virtual address and the loaded calculates real virtual addresses out of RVA's by adding the modules base address to an RVA.

Image Sections

An executable section is divided into individual sections in which the file's contents are stored. Sections are needed because different areas in the file are treated differently by the memory manager when a module is loaded. This division takes place in the code section (also called text) containing the executable's code and a data section containing the executable's data.

When loaded, the memory manager sets the access rights on memory pages in the different sections based on their settings in the section header.

Section Alignment

Individual sections often have different access settings defined in the executable header. The memory manager must apply these access settings when an executable image is loaded. Sections must typically be page aligned when an executable is loaded into memory. It would take extra space on disk to page align sections on disk. Therefore, the PE header has two different kinds of alignment fields, section alignment and file alignment.

DLL's

DLL's allow a program to be broken into more than one executable file. In this way, overall memory consumption is reduced, executables are not loaded until features they implement are required. Individual components can be replaced or upgraded to modify or improve a certain aspect of the program.

DLL's can dramatically reduce overall system memory consumption because the system can detect that a certain executable has been loaded into more than one address space, then map it into each address space instead of reloading it into a new memory location. DLL's are different from static libraries (.lib) which linked to the executable.

Loading DLL's

Static Linking is implemented by having each module list the the modules it uses and the functions it calls within each module. This is known as an import table (see IDA Pro tutorial). Run time linking refers to a different process whereby an executable can decide to load another executable in runtime and call a function from that executable.

PE Headers

A Portable Executable (PE) file starts with a DOS header.
"This program cannot be run in DOS mode"

```
typedef struct _IMAGE_NT_HEADERS {
        DWORD Signature;
        IMAFE_FILE_HEADER Fileheader;
        IMAGE_OPTIONAL_HEADER32 OptionHeader;
} Image_NT_HEADERS32, *PIMAGE_NT_HEADERS32
```

This data structure references two data structures that contain the actual PE header.

Imports and Exports

Imports and Exports are the mechanisms that enable the dynamic linking process of executables. The compiler has no idea of the actual addresses of the imported functions, only in runtime will these addresses be known. To solve this issue, the linker creates a import table that lists all the functions imported by the current module by their names.

Reverse Engineering Malware, Part 5: OllyDbg Basics

In this series, we are examining how to reverse engineer malware to understand how it works and possibly re-purposing it. Hackers and espionage agencies such as the **CIA and NSA,** regularly re-purpose malware for other purpose.

Previously, we looked at the basics of **IDA Pro,** the most widely used disassembler in our industry. In this tutorial, we will look at one of the most widely used and free debuggers, OllyDbg.

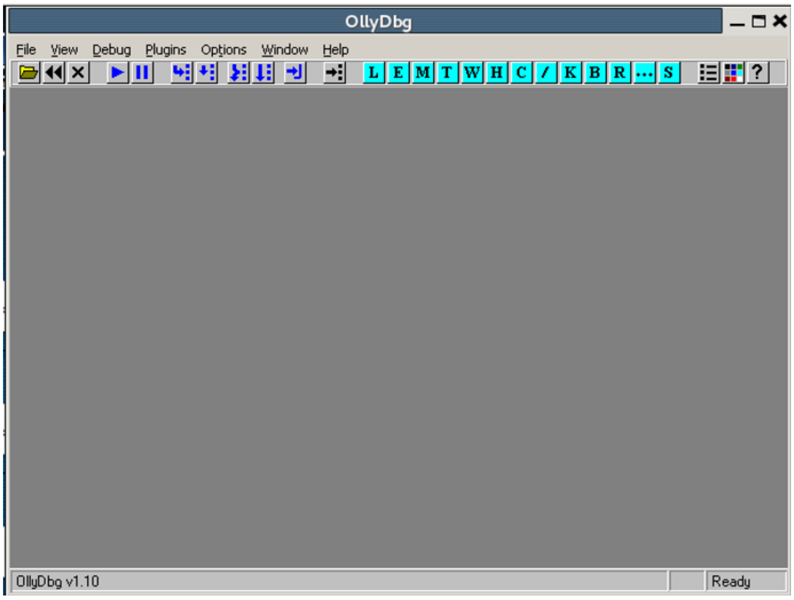
OllyDbg is a general purpose Win32 user-land debugger. It has an easy-to-use and fairly intuitive GUI making it a relatively quick study. Although OllyDbg is free, it is NOT open source as we do not have access to the source code. Despite this, OllyDbg has a well-defined plug-in architecture making it easily extensible to developers who want add capabilities to this powerful tool.

If you are using Kali or another security distribution, it is usually installed on your system. OllyDbg will run in either Windows or Linux and, in fact, it requires WINE to run in Linux. If you do not have OllyDbg on your system, you **can download OllyDbg here.**

Step #1: Starting OllyDbg

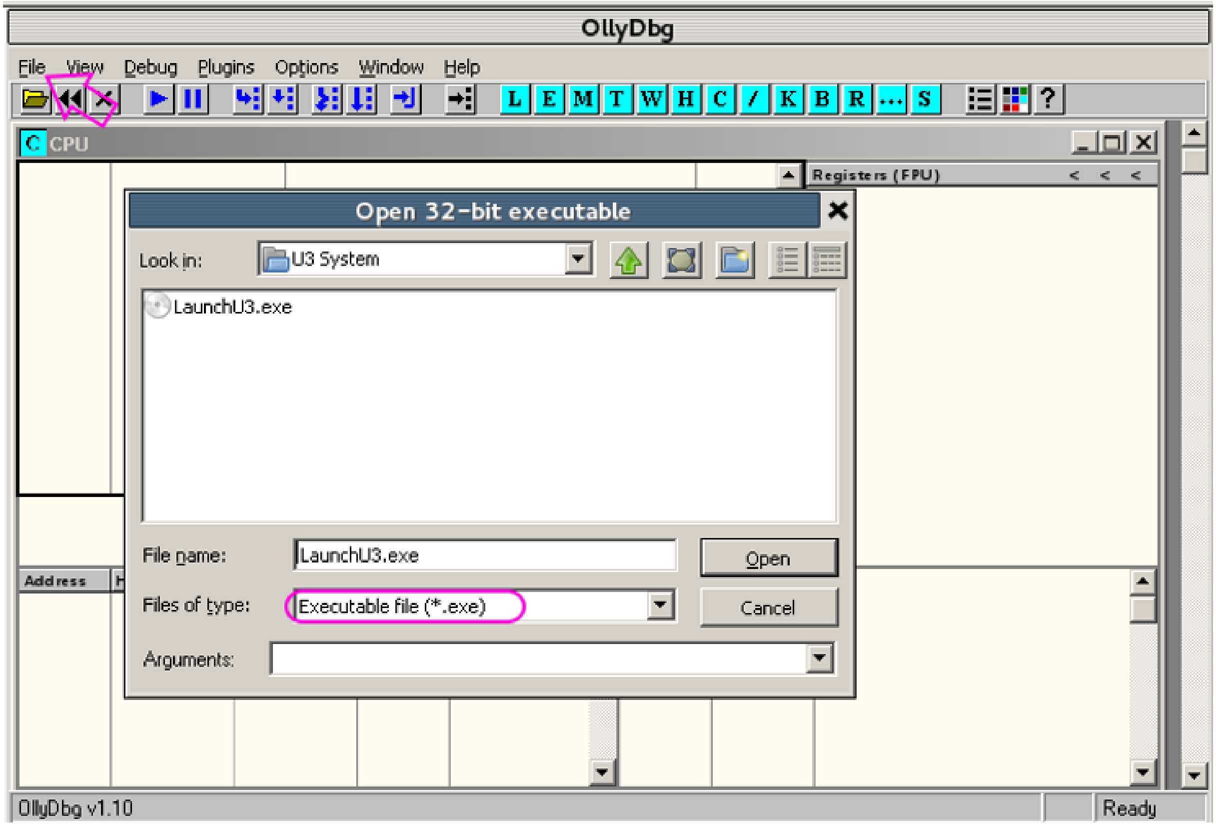
To start OllyDbg in Kali, go to **Applications, then Reverse Engineering** and finally **ollydbg,** as seen in this screenshot below.

When you do, it will open a screen like that below. Note that OllyDbg has the familiar pull-down menu system along the top of the GUI.



Step #2: Loading a File into OllyDbg

The next step is to load an .exe file into Ollydbg. You can do that by dragging and dropping the file into the work area of Olly or go to the File menu at the top and select Open. Note that the open window specifies that it **must** be an executable file.

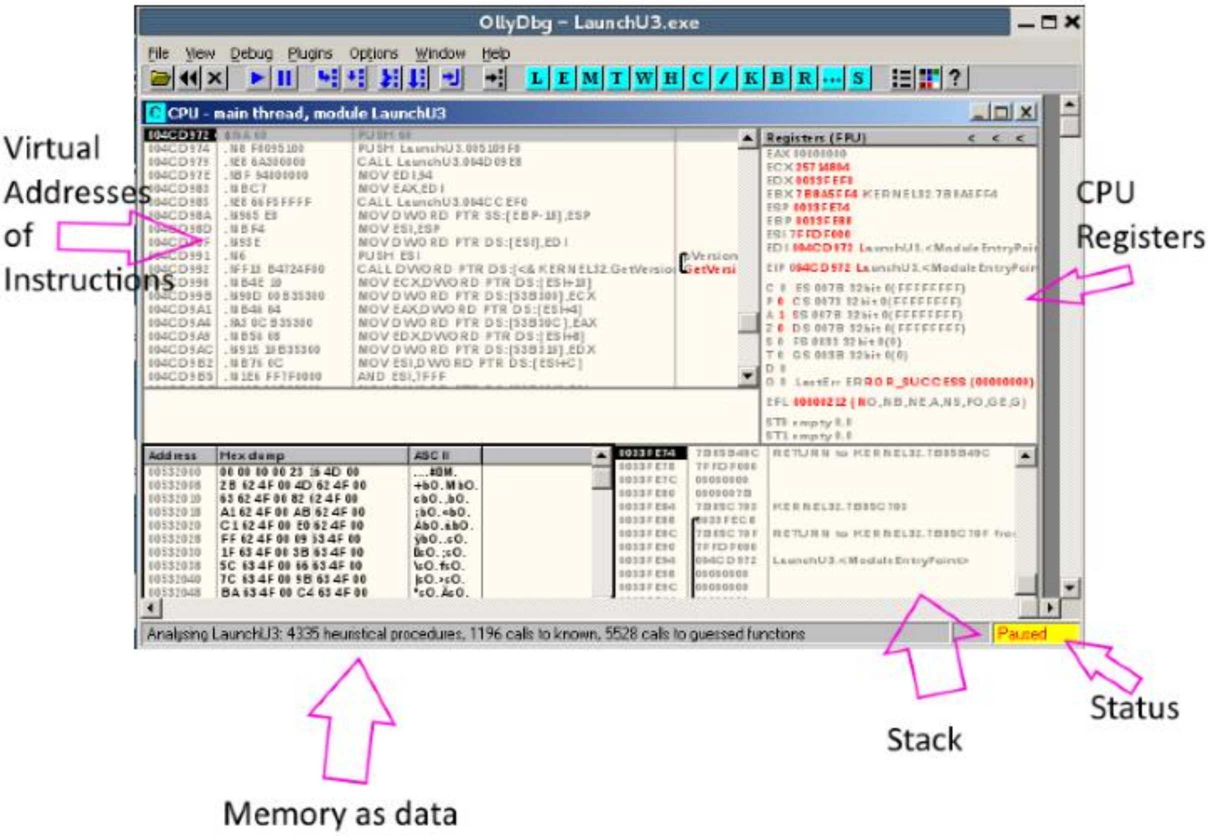


File name: LaunchU3.exe

Files of type: Executable file (*.exe)

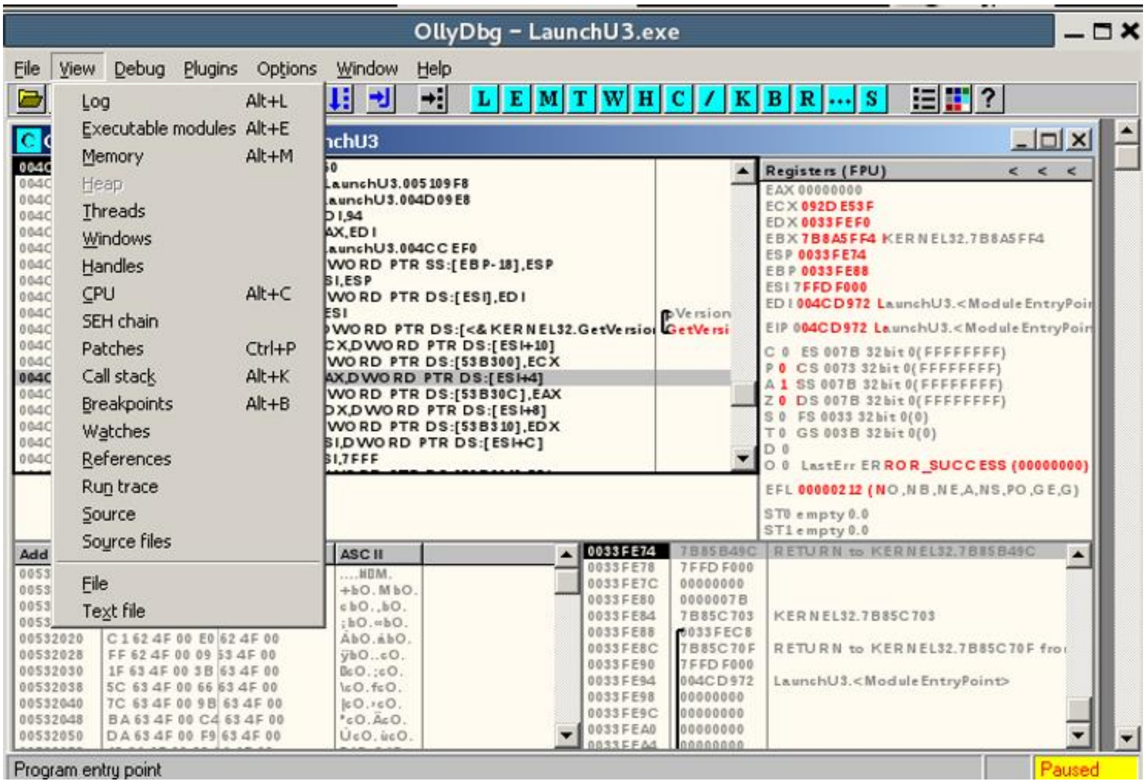
When you click open, Ollydbg will begin the process of analyzing your code. In this case, I used a simple .exe that comes pre-installed on my flash drive named LaunchU3.exe for demonstration purposes only. Obviously, it is NOT malware. In future tutorials, we will use both malware and non-malware to debug and analyze. Debuggers such as OllyDbg are also useful for analyzing errors (bugs) in code for developers and also breaking authentication schemes that prevent piracy.

As you can see below, Olly, takes the code and breaks into several windows. In the upper left window we have the virtual addresses of the instructions, in the upper right window the CPU registers, in the lower left we have the data residing in memory and finally in the lower right window, we have the stack. Also, please note that in the lower right, highlighted in yellow, we have the status. In this case, it indicates that we are in "pause" status.



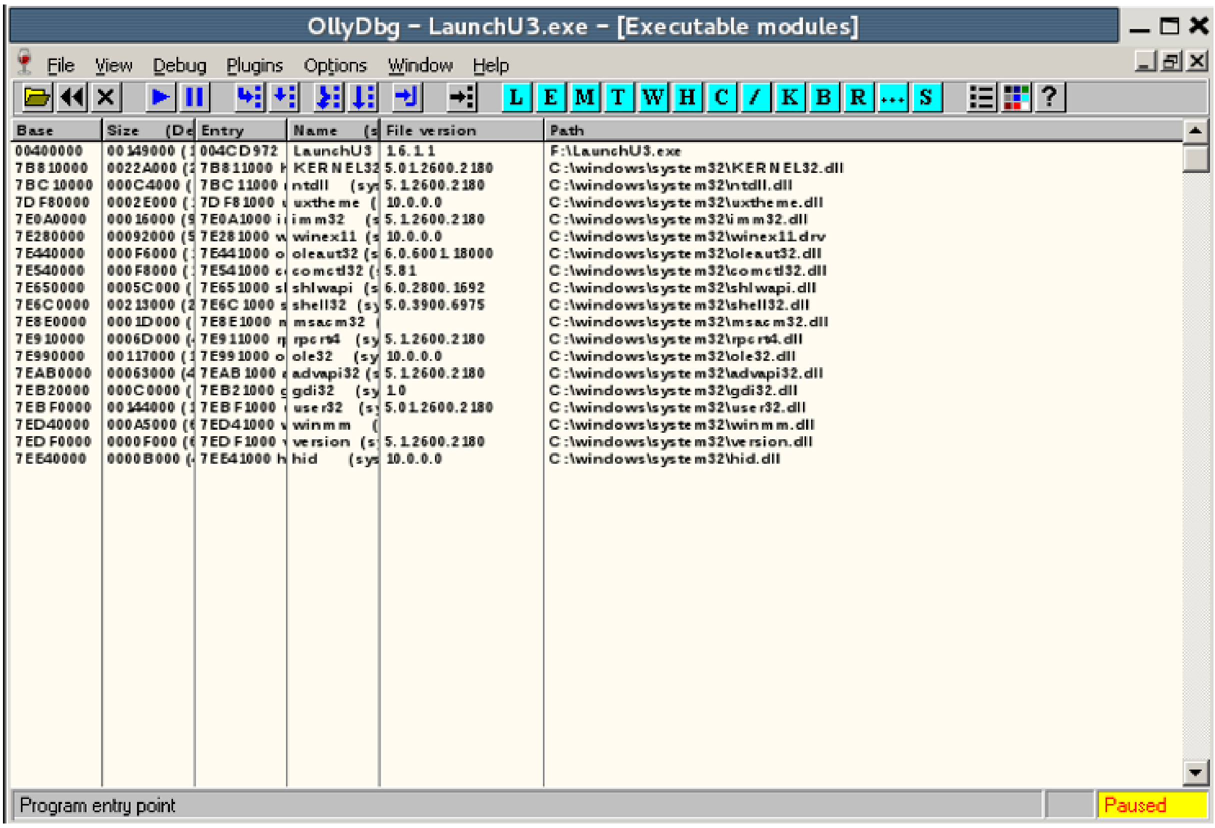
Step #3: Different Views of the Code

We can get different views of our data by clicking on the view button on the top menu. Note that each view is associated with a hotkey that is preceded by the Alt key with the exception of "patches" which uses the Ctrl key.



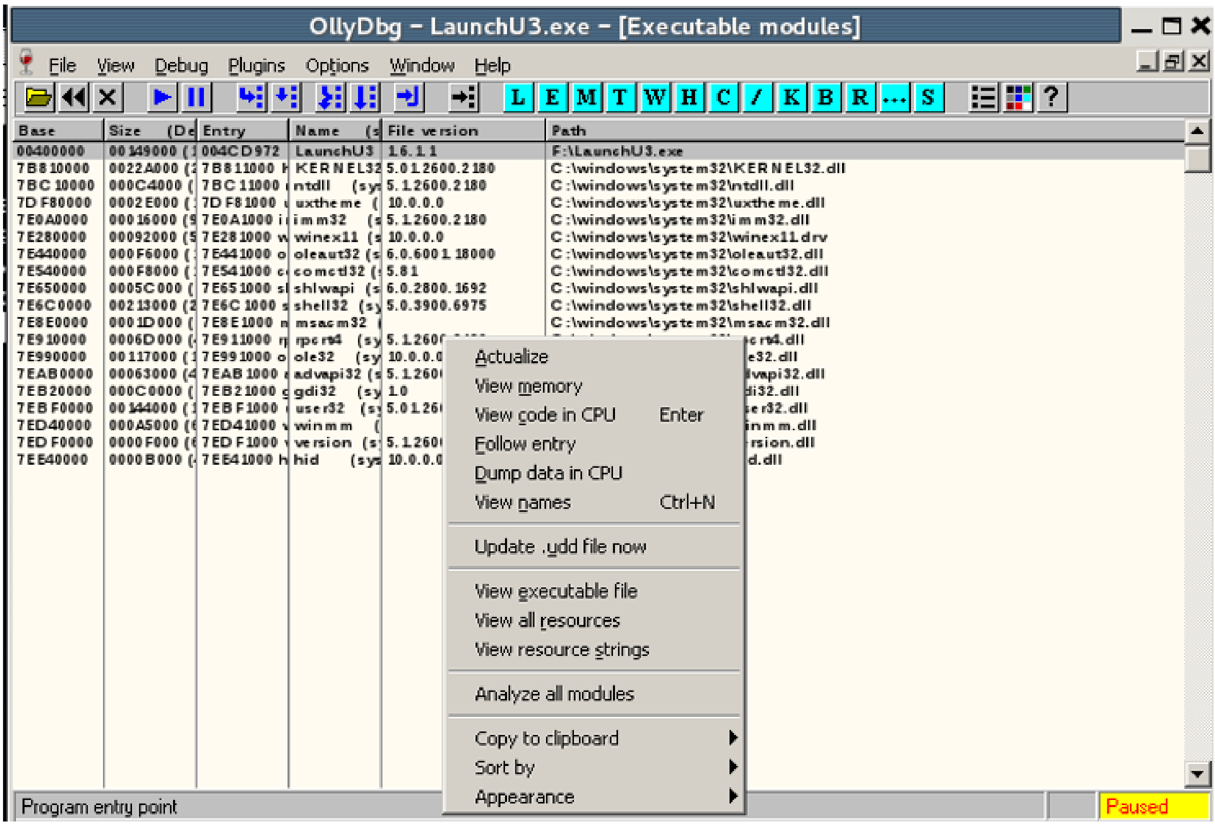
From here we can open a processes' logs (Alt+L), executables (Alt+E), memory layout (Alt +M), windows, handles and and its breakpoints (Alt+B). Note that each of these is also represented in the blue letters on the menu bar as shortcuts.

If we select the Executable modules (Alt+E) or the blue "E", we open a window with all the files executables like below.The Executable Modules Window shows the base virtual address to the far right, the virtual size of the binary in memory in the second column, the Entry Point’s virtual address in the third column, the name of the module in the fourth column, file version, and file path for each module loaded in the process. If the text appears in Red, that means the module was loaded dynamically.



```
Base      Size     (De Entry      Name     (s File version     Path
00400000  00149000 (1 004CD972   LaunchU3   1.6.1.1          F:\LaunchU3.exe
7B810000  0022A000 (2 7B811000 K KERNEL32  5.01.2600.2180   C:\windows\system32\KERNEL32.dll
7BC10000  000C4000 (  7BC11000 n ntdll   (sy 5.1.2600.2180   C:\windows\system32\ntdll.dll
7DF80000  0002E000 (1 7DF81000 u uxtheme  (  10.0.0.0        C:\windows\system32\uxtheme.dll
7E0A0000  00016000 (9 7E0A1000 i imm32    (s 5.1.2600.2180   C:\windows\system32\imm32.dll
7E280000  00092000 (5 7E281000 w winex11  (s 10.0.0.0        C:\windows\system32\winex11.drv
7E440000  000F6000 (1 7E441000 o oleaut32 (s 6.0.6001.18000  C:\windows\system32\oleaut32.dll
7E540000  000F8000 (1 7E541000 c comctl32 (: 5.81            C:\windows\system32\comctl32.dll
7E650000  0005C000 (  7E651000 sl shlwapi (s 6.0.2800.1692   C:\windows\system32\shlwapi.dll
7E6C0000  00213000 (2 7E6C1000 s shell32  (sy 5.0.3900.6975  C:\windows\system32\shell32.dll
7E8E0000  0001D000 (  7E8E1000 m msacm32  (                  C:\windows\system32\msacm32.dll
7E910000  0006D000 (4 7E911000 r rpcrt4   (sy 5.1.2600.2180  C:\windows\system32\rpcrt4.dll
7E990000  00117000 (1 7E991000 o ole32    (sy 10.0.0.0       C:\windows\system32\ole32.dll
7EAB0000  00063000 (4 7EAB1000 a advapi32 (s 5.1.2600.2180   C:\windows\system32\advapi32.dll
7EB20000  000C0000 (  7EB21000 g gdi32    (sy 1.0            C:\windows\system32\gdi32.dll
7EBF0000  00144000 (1 7EBF1000 u user32   (sy 5.01.2600.2180 C:\windows\system32\user32.dll
7ED40000  000A5000 (6 7ED41000 w winmm    (                  C:\windows\system32\winmm.dll
7EDF0000  0000F000 (6 7EDF1000 v version  (s 5.1.2600.2180   C:\windows\system32\version.dll
7EE40000  0000B000 (4 7EE41000 h hid     (sys 10.0.0.0       C:\windows\system32\hid.dll
```

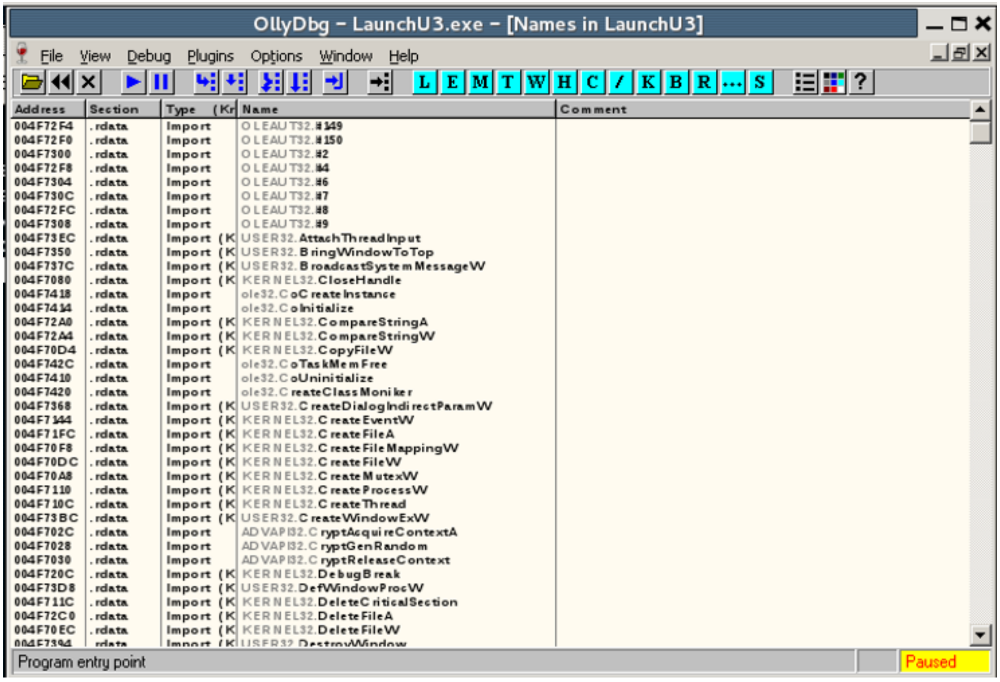
From the executables window, we can right click and pull up a context sensitive window. From here we can do a number of things, but let's take a look at the "View names" window.



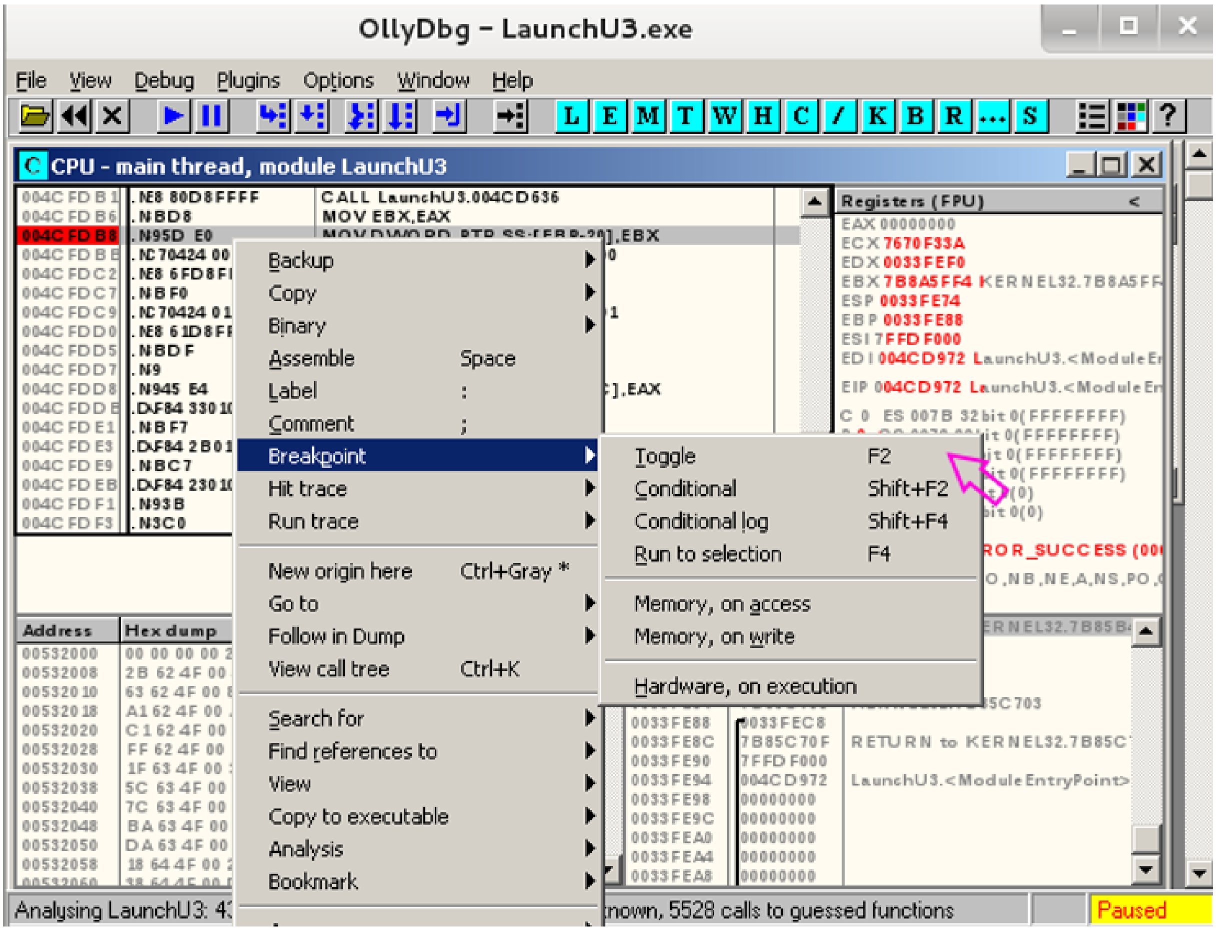
```
Base      Size     (De Entry      Name      (s File version     Path
00400000  00149000 (1  004CD972   LaunchU3     1.6.1.1          F:\LaunchU3.exe
7B810000  0022A000 (2  7B811000 K KERNEL32     5.0.1.2600.2180  C:\windows\system32\KERNEL32.dll
7BC10000  000C4000 (   7BC11000 n ntdll     (sy 5.1.2600.2180   C:\windows\system32\ntdll.dll
7DF80000  0002E000 (1  7DF81000 u uxtheme   (   10.0.0.0        C:\windows\system32\uxtheme.dll
7E0A0000  00016000 (9  7E0A1000 i imm32     (s 5.1.2600.2180    C:\windows\system32\imm32.dll
7E280000  00092000 (5  7E281000 w winex11   (s 10.0.0.0         C:\windows\system32\winex11.drv
7E440000  000F6000 (1  7E441000 o oleaut32  (s 6.0.6001.18000   C:\windows\system32\oleaut32.dll
7E540000  000F8000 (1  7E541000 c comctl32  (s 5.81             C:\windows\system32\comctl32.dll
7E650000  0005C000 (   7E651000 sl shlwapi  (s 6.0.2800.1692    C:\windows\system32\shlwapi.dll
7E6C0000  00213000 (2  7E6C1000 s shell32   (sy 5.0.3900.6975   C:\windows\system32\shell32.dll
7E8E0000  0001D000 (   7E8E1000 n msacm32   (                   C:\windows\system32\msacm32.dll
7E910000  0006D000 (4  7E911000 r rpcrt4    (sy 5.1.260                             crt4.dll
7E990000  00117000 (1  7E991000 o ole32     (sy 10.0.0.0                            e32.dll
7EAB0000  00063000 (4  7EAB1000 a advapi32  (s 5.1.260                              vapi32.dll
7EB20000  000C0000 (   7EB21000 g gdi32     (sy 1.0                                 di32.dll
7EBF0000  00144000 (1  7EBF1000 u user32    (sy 5.0.1.26                            er32.dll
7ED40000  000A5000 (6  7ED41000 v winmm     (                                       inmm.dll
7EDF0000  0000F000 (6  7EDF1000 v version   (s 5.1.260                              rsion.dll
7EE40000  0000B000 (4  7EE41000 h hid       (sys 10.0.0.0                           d.dll
```

Here we see all the functions and imported functions used in the program. We can also access this window by using the Ctrl+N. By examining the executable's imported functions we can often decipher the malware's functionality. Microsoft's MSDN API documentation site (www.MSDN.microsoft.com) can be a useful resource for finding out what these functions do, the parameter's these functions take in, and what these functions return.

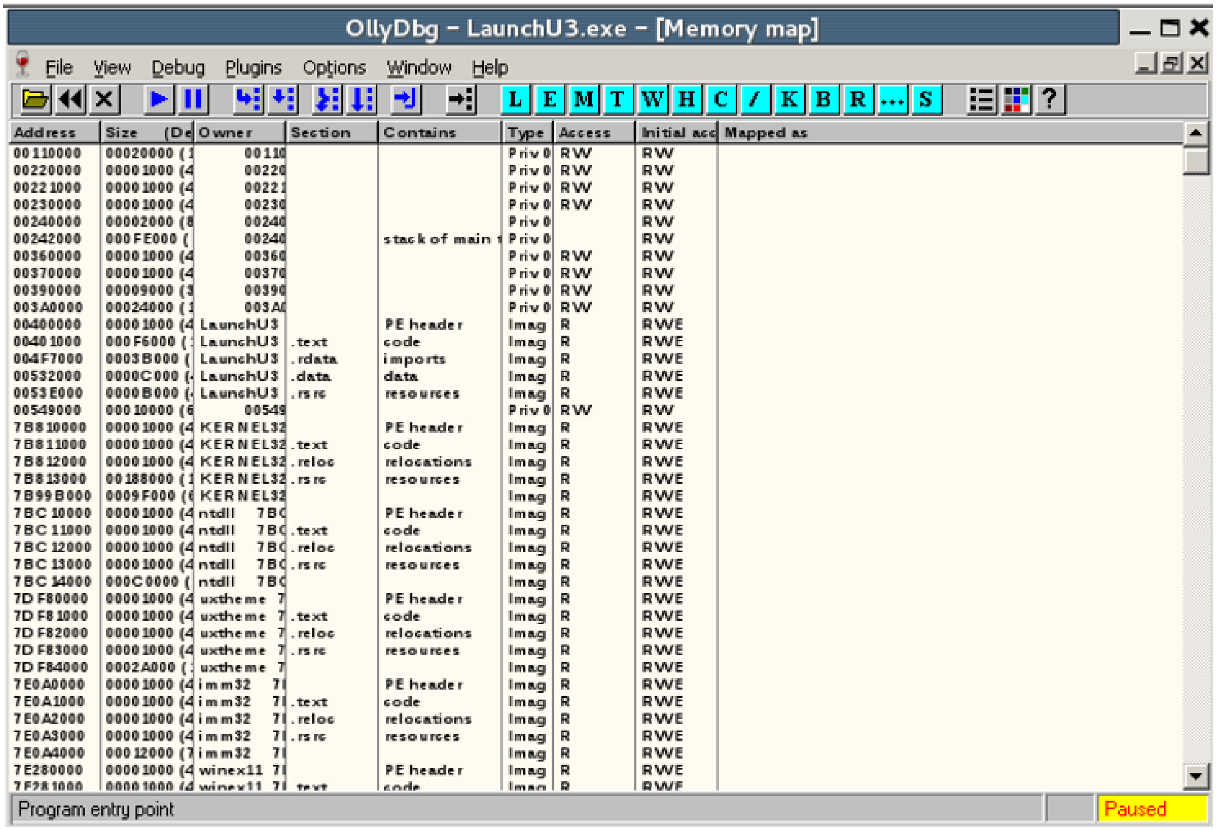
From the Names window, if we right click on the function names we can set a breakpoint by clicking on Toggle Breakpoint or F2.



```
Address   Section  Type   (Kr Name                                      Comment
004F72F4  .rdata   Import     OLEAUT32.#149
004F72F0  .rdata   Import     OLEAUT32.#150
004F7300  .rdata   Import     OLEAUT32.#2
004F72F8  .rdata   Import     OLEAUT32.#4
004F7304  .rdata   Import     OLEAUT32.#6
004F730C  .rdata   Import     OLEAUT32.#7
004F72FC  .rdata   Import     OLEAUT32.#8
004F7308  .rdata   Import     OLEAUT32.#9
004F73EC  .rdata   Import  (K USER32.AttachThreadInput
004F7350  .rdata   Import  (K USER32.BringWindowToTop
004F737C  .rdata   Import  (K USER32.BroadcastSystemMessageW
004F7080  .rdata   Import  (K KERNEL32.CloseHandle
004F7418  .rdata   Import     ole32.CoCreateInstance
004F7414  .rdata   Import     ole32.CoInitialize
004F72A0  .rdata   Import  (K KERNEL32.CompareStringA
004F72A4  .rdata   Import  (K KERNEL32.CompareStringW
004F70D4  .rdata   Import  (K KERNEL32.CopyFileW
004F742C  .rdata   Import     ole32.CoTaskMemFree
004F7410  .rdata   Import     ole32.CoUninitialize
004F7420  .rdata   Import     ole32.CreateClassMoniker
004F7368  .rdata   Import  (K USER32.CreateDialogIndirectParamW
004F7144  .rdata   Import  (K KERNEL32.CreateEventW
004F71FC  .rdata   Import  (K KERNEL32.CreateFileA
004F70F8  .rdata   Import  (K KERNEL32.CreateFileMappingW
004F70DC  .rdata   Import  (K KERNEL32.CreateFileW
004F70A8  .rdata   Import  (K KERNEL32.CreateMutexW
004F7110  .rdata   Import  (K KERNEL32.CreateProcessW
004F710C  .rdata   Import  (K KERNEL32.CreateThread
004F73BC  .rdata   Import  (K USER32.CreateWindowExW
004F702C  .rdata   Import     ADVAPI32.CryptAcquireContextA
004F7028  .rdata   Import     ADVAPI32.CryptGenRandom
004F7030  .rdata   Import     ADVAPI32.CryptReleaseContext
004F720C  .rdata   Import  (K KERNEL32.DebugBreak
004F73D8  .rdata   Import  (K USER32.DefWindowProcW
004F711C  .rdata   Import  (K KERNEL32.DeleteCriticalSection
004F72C0  .rdata   Import  (K KERNEL32.DeleteFileA
004F70EC  .rdata   Import  (K KERNEL32.DeleteFileW
004F7394  .rdata   Import  (K USER32.DestroyWindow
```

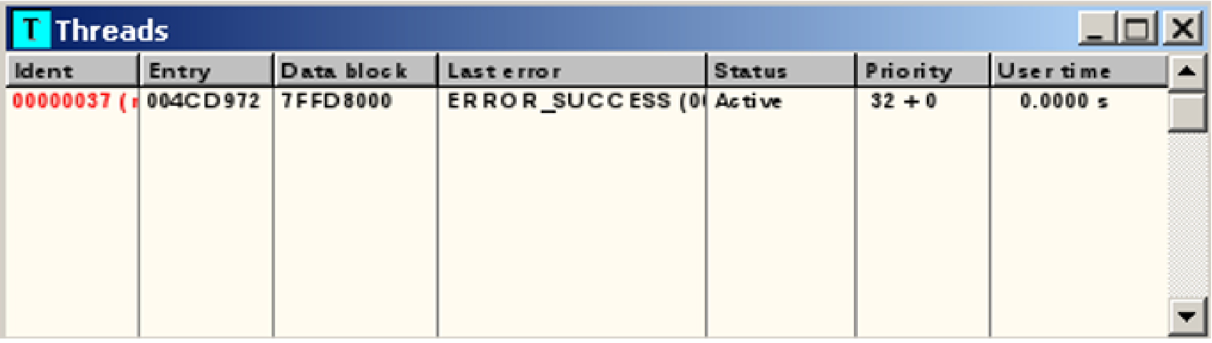


OllyDbg’s Memory Map window shows the virtual address, the virtual size, the owner module, section names, memory allocation type and memory protection for each allocated region of memory in the process.



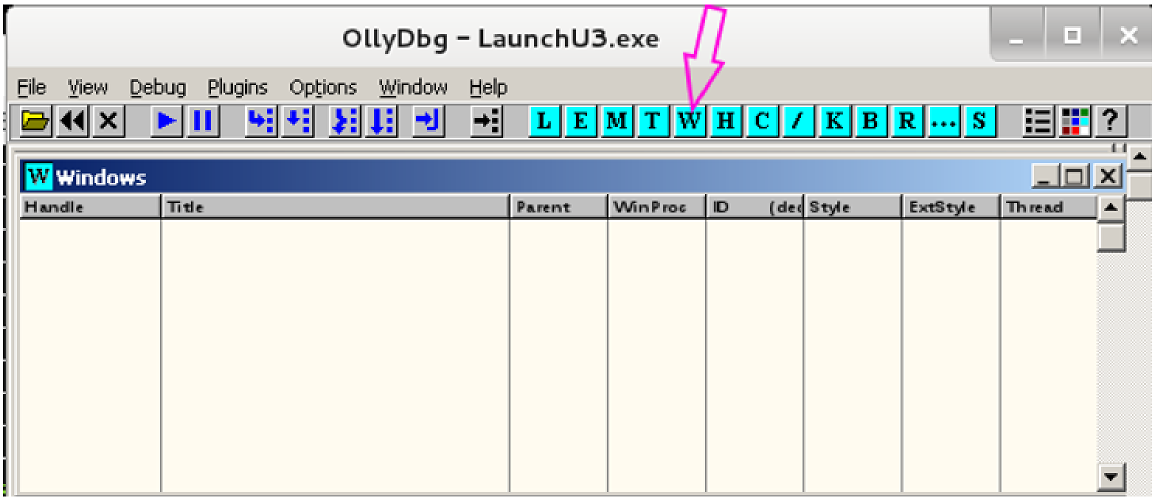
Address	Size (De	Owner	Section	Contains	Type	Access	Initial acc	Mapped as
00110000	00020000 (1	00110			Priv 0	RW	RW	
00220000	00001000 (4	00220			Priv 0	RW	RW	
00221000	00001000 (4	00221			Priv 0	RW	RW	
00230000	00001000 (4	00230			Priv 0	RW	RW	
00240000	00002000 (8	00240			Priv 0		RW	
00242000	000FE000 (	00240		stack of main t	Priv 0		RW	
00360000	00001000 (4	00360			Priv 0	RW	RW	
00370000	00001000 (4	00370			Priv 0	RW	RW	
00390000	00009000 (3	00390			Priv 0	RW	RW	
003A0000	00024000 (1	003A0			Priv 0	RW	RW	
00400000	00001000 (4	LaunchU3		PE header	Imag	R	RWE	
00401000	000F6000 (1	LaunchU3	.text	code	Imag	R	RWE	
004F7000	0003B000 (	LaunchU3	.rdata	imports	Imag	R	RWE	
00532000	0000C000 (4	LaunchU3	.data	data	Imag	R	RWE	
0053E000	0000B000 (4	LaunchU3	.rsrc	resources	Imag	R	RWE	
00549000	00010000 (6	00549			Priv 0	RW	RW	
7B810000	00001000 (4	KERNEL32		PE header	Imag	R	RWE	
7B811000	00001000 (4	KERNEL32	.text	code	Imag	R	RWE	
7B812000	00001000 (4	KERNEL32	.reloc	relocations	Imag	R	RWE	
7B813000	00188000 (1	KERNEL32	.rsrc	resources	Imag	R	RWE	
7B99B000	0009F000 (6	KERNEL32			Imag	R	RWE	
7BC10000	00001000 (4	ntdll 7BC		PE header	Imag	R	RWE	
7BC11000	00001000 (4	ntdll 7BC	.text	code	Imag	R	RWE	
7BC12000	00001000 (4	ntdll 7BC	.reloc	relocations	Imag	R	RWE	
7BC13000	00001000 (4	ntdll 7BC	.rsrc	resources	Imag	R	RWE	
7BC14000	000C0000 (	ntdll 7BC			Imag	R	RWE	
7DF80000	00001000 (4	uxtheme 7		PE header	Imag	R	RWE	
7DF81000	00001000 (4	uxtheme 7	.text	code	Imag	R	RWE	
7DF82000	00001000 (4	uxtheme 7	.reloc	relocations	Imag	R	RWE	
7DF83000	00001000 (4	uxtheme 7	.rsrc	resources	Imag	R	RWE	
7DF84000	0002A000 (1	uxtheme 7			Imag	R	RWE	
7E0A0000	00001000 (4	imm32 7I		PE header	Imag	R	RWE	
7E0A1000	00001000 (4	imm32 7I	.text	code	Imag	R	RWE	
7E0A2000	00001000 (4	imm32 7I	.reloc	relocations	Imag	R	RWE	
7E0A3000	00001000 (4	imm32 7I	.rsrc	resources	Imag	R	RWE	
7E0A4000	00012000 (7	imm32 7I			Imag	R	RWE	
7E280000	00001000 (4	winex11 7I		PE header	Imag	R	RWE	
7E281000	00001000 (4	winex11 7I	.text	code	Imag	R	RWE	

OllyDbg’s Threads window shows the thread ID, Entry Point virtual address, the Thread Environment Block (TEB) virtual address, the last-error value, status such as, active or suspended, the priority, and the timing information for each thread in the process.

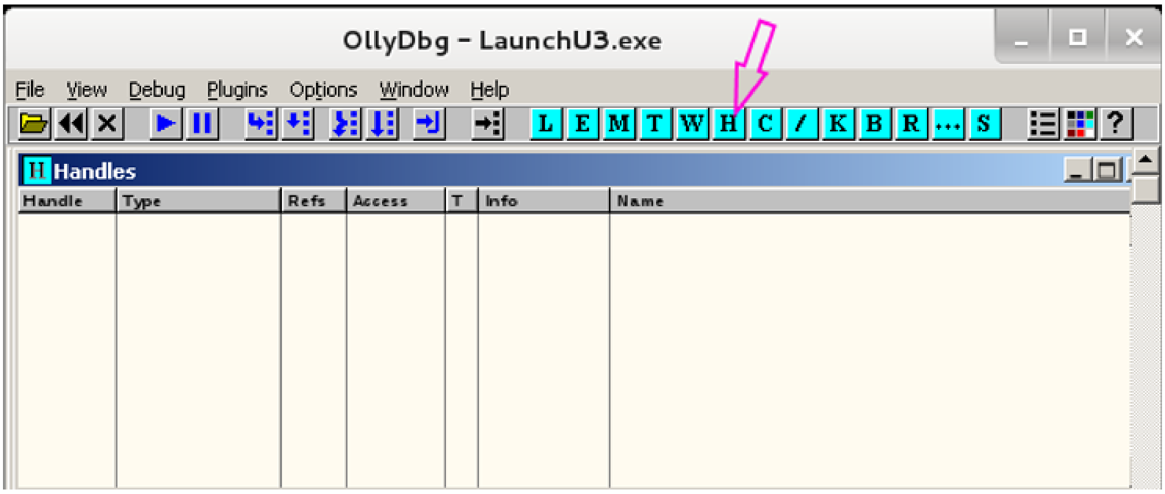


Ident	Entry	Data block	Last error	Status	Priority	User time
00000037 (r	004CD972	7FFD8000	ERROR_SUCCESS (0(	Active	32 + 0	0.0000 s

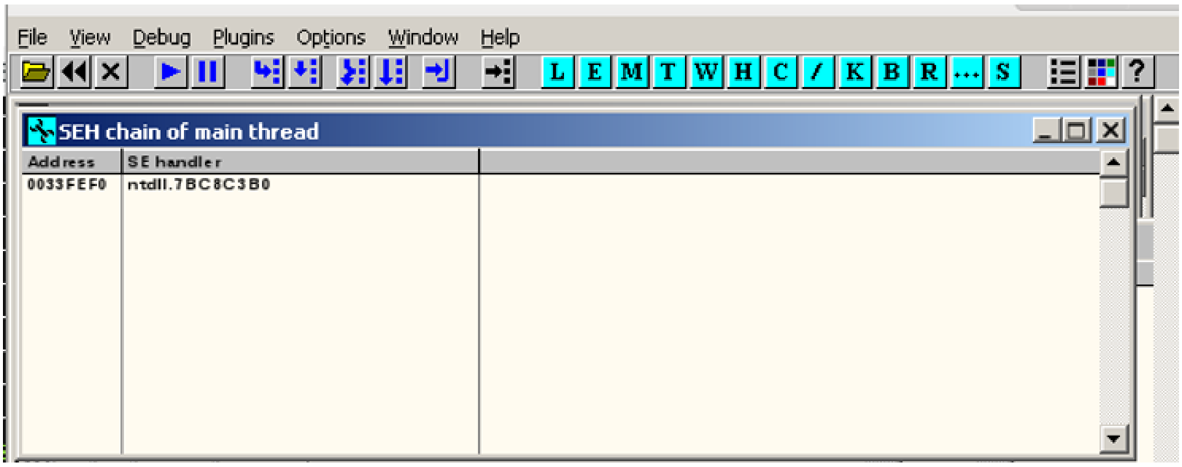
The Windows window displays the Handle, Title, Parent Window, Window ID, Window Style, and Window Class Information for each window owned by the process.



The Handles window shows the object type, reference count, access flags, and the object name for each handle owned by the process.



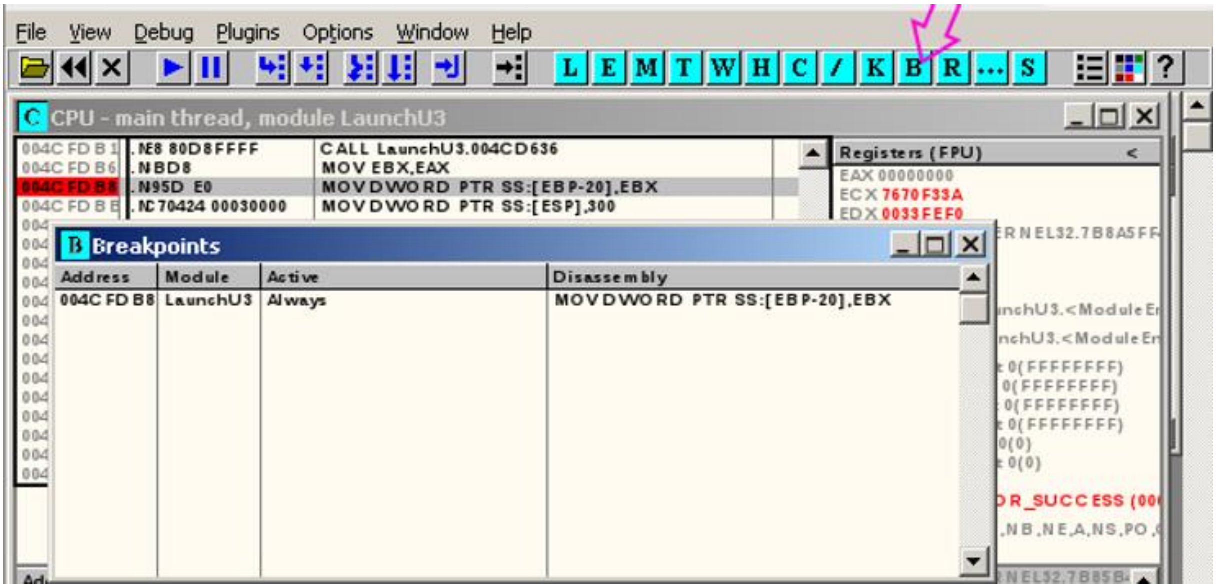
The SEH (Structured Exception Handler) chain window shows the Structured Exception Handler functions for the current thread.



```
Address   SE handler
0033FEF0  ntdll.7BC8C3B0
```

Breakpoints

One of key features of any debugger is the ability to set breakpoints. A breakpoint enables us to stop the execution of a program at a specified address or instruction. There are two primary types of breakpoints (1) software and (2) hardware. OllyDbg provides a way to view and turn on and off breakpoints via the breakpoints window with Alt+B



```
004C FD B1 .E8 80D8FFFF        CALL LaunchU3.004CD636
004C FD B6 .8BD8               MOV EBX,EAX
004C FD B8 .895D E0            MOV DWORD PTR SS:[EBP-20],EBX
004C FD BE .C70424 00030000    MOV DWORD PTR SS:[ESP],300
```

```
EAX 00000000
ECX 7670F33A
EDX 0033FEF0
```

Address	Module	Active	Disassembly
004C FD B8	LaunchU3	Always	MOV DWORD PTR SS:[EBP-20],EBX

OllyDbg Frequently Used Shortcuts

UI

Open new program F3
Close program Alt+F2
Maximize/restore active windows F5
Make OllyDbg topmost window Alt+F5
Close OllyDbg Alt+X

Windows

Open breakpoints window Alt+B
Open CPU window Alt+C
Open modules window Alt+E
Open log window Alt+L
Open memory window Alt+M

Editing

Add label : (Colon)
Add comment ; (Semicolon)
Edit memory Ctrl+
Assemble Space
Undo changes Alt+BkSp

Execution

Step into F7
Animate into Ctrl+F7
Step over F8
Animate over Ctrl+F8
Run application F9
Pass exception handler and run Shift+F9
Execute till return Ctrl+F9
Execute till user code Alt+F9
Trace into Ctrl+F11
Trace over Ctrl+F12
Pause F12
Pause trace conditional Ctrl+T
Run to selection F4

Breakpoints

Set/Unset breakpoint F2
Set/Edit conditional breakpoint Shift+F2
Set/Edit conditional log breakpoint Shift+F4
Temporarily disable/restore BP Space

Data

Analyze executable code Ctrl+A
Scan object files Ctrl+O
Display symbolic names Ctrl+N

Searching

Find selected address xrefs Ctrl+R
Find jumps to line Ctrl+J
Search for sequence Ctrl+S
Search allocated memory Ctrl+L
Search binary Ctrl+B
Search for a command Ctrl+F
Repeat last search Ctrl+L

Navigation

Go to origin * (Asterisk)
Go to address of expression Ctrl+G
Go to previous address - (Minus)
Go to next address + (Plus)
Go to previous procedure Ctrl+-
Go to next procedure Ctrl++
Go to previous reference Alt+F7
Go to next reference Alt+F8
Follow expression Ctrl+G
Follow jump or call Enter
View call tree Ctrl+K

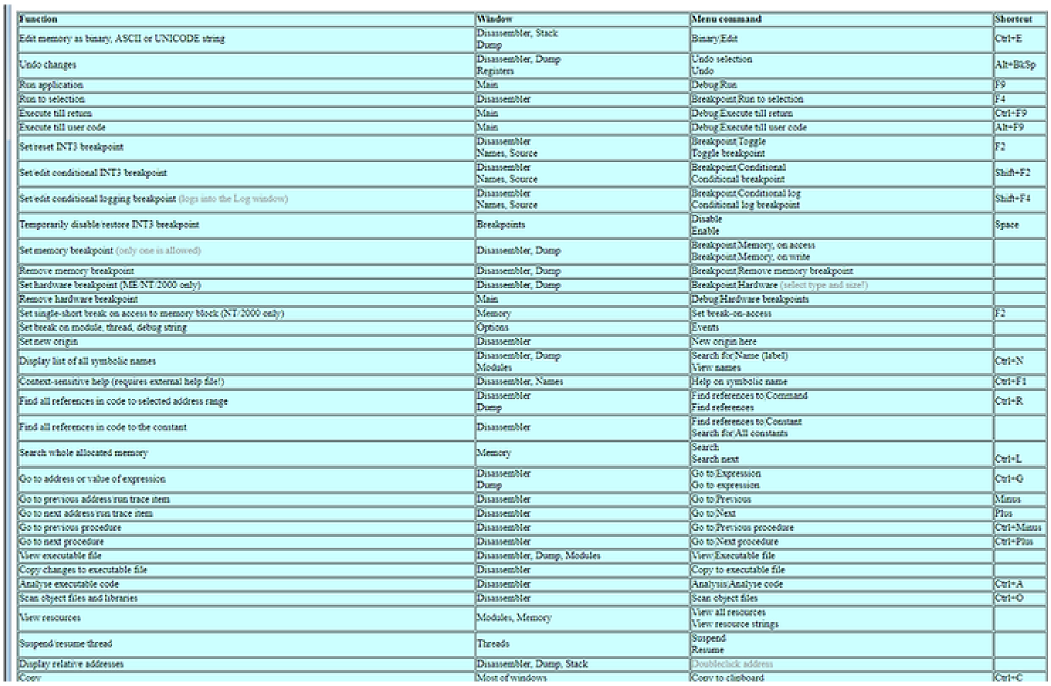
Miscellaneous

Context sensitive help Ctrl+F

Complete List of Shortcuts

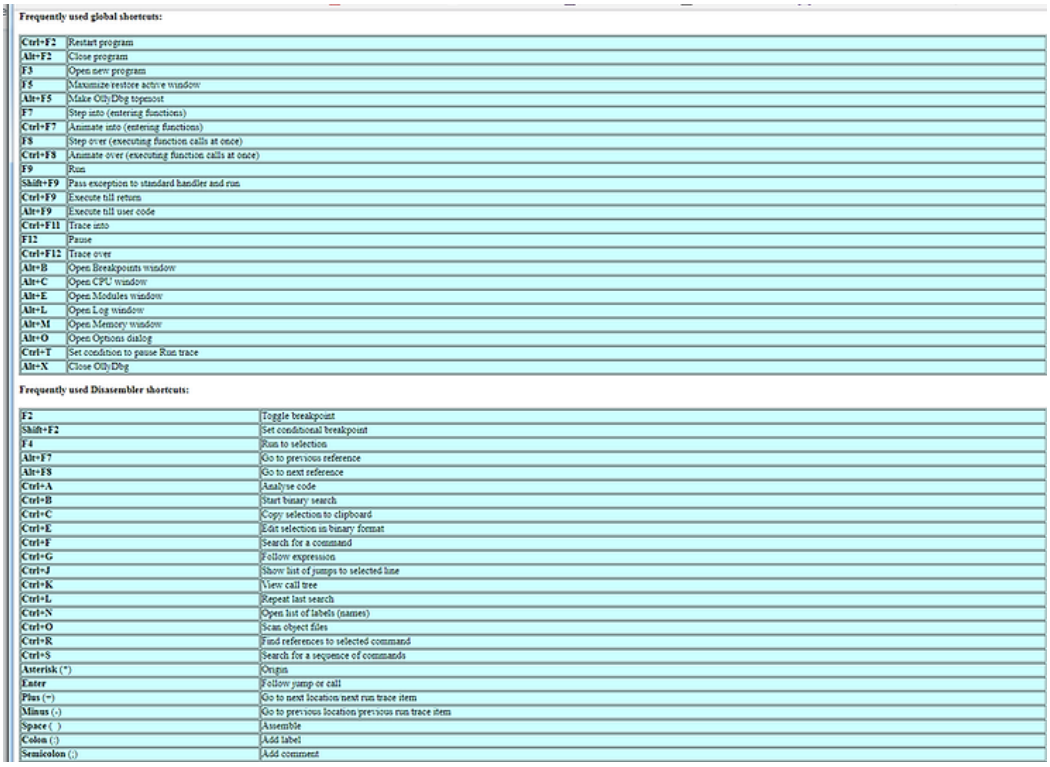
The following is a complete list of OllyDbg shortcuts from OllyDbg's official website www.ollydbg.de and visit the [Quick start]() section.

Functions



Function	Window	Menu command	Shortcut
Edit memory as binary, ASCII or UNICODE string	Disassembler, Stack Dump	Binary|Edit	Ctrl+E
Undo changes	Disassembler, Dump Registers	Undo selection Undo	Alt+BkSp
Run application	Main	Debug|Run	F9
Run to selection	Disassembler	Breakpoint|Run to selection	F4
Execute till return	Main	Debug|Execute till return	Ctrl+F9
Execute till user code	Main	Debug|Execute till user code	Alt+F9
Set/reset INT3 breakpoint	Disassembler Names, Source	Breakpoint|Toggle Toggle breakpoint	F2
Set/edit conditional INT3 breakpoint	Disassembler Names, Source	Breakpoint|Conditional Conditional breakpoint	Shift+F2
Set/edit conditional logging breakpoint (logs into the Log window)	Disassembler Names, Source	Breakpoint|Conditional log Conditional log breakpoint	Shift+F4
Temporarily disable/restore INT3 breakpoint	Breakpoints	Disable Enable	Space
Set memory breakpoint (only one is allowed)	Disassembler, Dump	Breakpoint|Memory, on access Breakpoint|Memory, on write	
Remove memory breakpoint	Disassembler, Dump	Breakpoint|Remove memory breakpoint	
Set hardware breakpoint (ME/NT/2000 only)	Disassembler, Dump	Breakpoint|Hardware (select type and size!)	
Remove hardware breakpoint	Main	Debug|Hardware breakpoints	
Set single-short break on access to memory block (NT/2000 only)	Memory	Set break-on-access	F2
Set break on module, thread, debug string	Options	Events	
Set new origin	Disassembler	New origin here	
Display list of all symbolic names	Disassembler, Dump Modules	Search for|Name (label) View names	Ctrl+N
Context-sensitive help (requires external help file!)	Disassembler, Names	Help on symbolic name	Ctrl+F1
Find all references in code to selected address range	Disassembler Dump	Find references to|Command Find references	Ctrl+R
Find all references in code to the constant	Disassembler	Find references to|Constant Search for|All constants	
Search whole allocated memory	Memory	Search Search next	 Ctrl+L
Go to address or value of expression	Disassembler Dump	Go to|Expression Go to expression	Ctrl+G
Go to previous address/run trace item	Disassembler	Go to|Previous	Minus
Go to next address/run trace item	Disassembler	Go to|Next	Plus
Go to previous procedure	Disassembler	Go to|Previous procedure	Ctrl+Minus
Go to next procedure	Disassembler	Go to|Next procedure	Ctrl+Plus
View executable file	Disassembler, Dump, Modules	View|Executable file	
Copy changes to executable file	Disassembler	Copy to executable file	
Analyse executable code	Disassembler	Analysis|Analyse code	Ctrl+A
Scan object files and libraries	Disassembler	Scan object files	Ctrl+O
View resources	Modules, Memory	View all resources View resource strings	
Suspend/resume thread	Threads	Suspend Resume	
Display relative addresses	Disassembler, Dump, Stack	Doubleclick address	
Copy	Most of windows	Copy to clipboard	Ctrl+C

Global Shortcuts



Frequently used global shortcuts:

Ctrl+F2	Restart program
Alt+F2	Close program
F3	Open new program
F5	Maximize/restore active window
Alt+F5	Make OllyDbg topmost
F7	Step into (entering functions)
Ctrl+F7	Animate into (entering functions)
F8	Step over (executing function calls at once)
Ctrl+F8	Animate over (executing function calls at once)
F9	Run
Shift+F9	Pass exception to standard handler and run
Ctrl+F9	Execute till return
Alt+F9	Execute till user code
Ctrl+F11	Trace into
F12	Pause
Ctrl+F12	Trace over
Alt+B	Open Breakpoints window
Alt+C	Open CPU window
Alt+E	Open Modules window
Alt+L	Open Log window
Alt+M	Open Memory window
Alt+O	Open Options dialog
Ctrl+T	Set condition to pause Run trace
Alt+X	Close OllyDbg

Frequently used Disassembler shortcuts:

F2	Toggle breakpoint
Shift+F2	Set conditional breakpoint
F4	Run to selection
Alt+F7	Go to previous reference
Alt+F8	Go to next reference
Ctrl+A	Analyse code
Ctrl+B	Start binary search
Ctrl+C	Copy selection to clipboard
Ctrl+E	Edit selection in binary format
Ctrl+F	Search for a command
Ctrl+G	Follow expression
Ctrl+J	Show list of jumps to selected line
Ctrl+K	View call tree
Ctrl+L	Repeat last search
Ctrl+N	Open list of labels (names)
Ctrl+O	Scan object files
Ctrl+R	Find references to selected command
Ctrl+S	Search for a sequence of commands
Asterisk (*)	Origin
Enter	Follow jump or call
Plus (+)	Go to next location/next run trace item
Minus (-)	Go to previous location/previous run trace item
Space ( )	Assemble
Colon (:)	Add label
Semicolon (;)	Add comment