# Disassembly Using IDA

## 1. Code Analysis Tools

- *Dissasembler:* Program that translates machine code back to assembly code
- *Debugger:* Dissasembles code and allows you to execute in a controlled environment
- *Decompiler:* Program that translates machine code into a higher-level pseudo-code

## 2. Static Code Analysis (Disassembly) Using IDA

The recommended version to use for this book is the IDA Commercial version, or, as an alternative, the IDA demo version.

## 2.1 Loading Binary in IDA

This is a guide to open a file. After the file is loaded, the dissasembly engine disassembles the machine code. After the dissasembly, IDA performs an initial analysis to identify the compiler, function arguments, local variables, library functions, and their parameters.

## 2.2 Exploring the IDA Displays

The IDA desktop contains multiple tabs. Clicking on each tab brings up a different window. Each window contains different information extracted by the binary.

### 2.2.1 Disassembly Window

Primary window. Displays the disassembled code. Has two display modes: Graph view and Text view Switch between modes with the space key.

#### Grap Mode

IDA displays only one function at a time. Useful to recognize *branching* and *looping* statements. Red arrow: code executed when condition is false Green arrow: code executed when condition is true Blue arrow: Unconditional jump Backwards blue arrow: loop jump

#### Text Mode

Disassembly in a linear fashon Left of code, the arrows indicate jumps Non-linear arrow indicates conditional jumps Linear arrows indicate unconditional jumps Backwards arrows indicate loops

### Other Windows: 2.2.2 - 2.2.9

- *Functions Window*:
  - Displays all the functions recognized by IDA and their properties.
  - Each function is associated with flags: R, F, L
  - The L flag indicates it's a library function, therefore not written by the malware author
- *Output Window*:
  - Displays the messages generated by IDA and the IDA plugins.
- *Hex View Window*
  - Displays a sequence of bytes in hex-dump and ASCII format.
- *Structures Window*:
  - Lists layout of structures used in program
  - Allows you to create your own
- *Import Window*:
  - Lists functions imported by the binary
- *Export Window*:
  - Lists exported functions
  - Useful when analyzing malicious DLL's
- *Strings Window*:
  - Lists strings and their address
- *Segments Window*:
  - Lists sections (`.text`, `.data`...) in the binary file.

## 2.3 Improving Disassembly Using IDA

When a executable is loaded, IDA performs an analysis on all it's functions, to determine the layout of the stack frame.

If the `ebp` stack register is used, it lets you know, with a comment at the right.

IDA, also knows which function is the `main()` one, based on signatures.

IDA also clarifies local variables, and changes their name to a more clear one. For example, `[ebp-4]` to `[ebp+var_4]`. If it is a variable, it is named with `*var_*` if, instead, it is a argument, it is named `*arg_*`.

### 2.3.1 Renaming Locations

Even though IDA renames the variables or arguments, you should rename them to more meaningful names. Right click and select rename.

### 2.3.2 Commenting in IDA

To comment, place the cursor on any line and press the `:` button. This will bring up the comment entry dialog.

Function comments allow you to group multiple comments and group them at the top of the function. To add a function comment, highlight the function name

### 2.3.3 IDA Database

When a executable is loaded, it creates a database with 5 files: `.id0, .id1, .nam, .id2, .til`

The various displays, are presented as views to the database. Any modification is reflected in the views and stored in the database. IDA uses the `offset` keyword to indicate that addresses of variables are used, rather than the contents of the variables. A address containing a string will have commented at its right the string's value.

### 2.3.4 Formatting Operands

IDA gives you the option to reformat constant values, as a `decimal, octal, binary` value. To do this, right click on the variable and select the option that you prefer.

### 2.3.5 Navigating Locations

Double clicking on the locations, will take you to the selected location. To go to the previous location or the next location (If there is one) press the navigation buttons.

### 2.3.6 Cross-References (X-refs)

The cross-references link relates addresses together. They can be either *data cross-references* or *code cross-references*. A data cross-reference, specifies how data is accessed within a binary. An example of a data cross-reference is:

```
.data:00403374   dword_403374 dd ?  ;  DATA XREF: _main+6w
```

This data cross-reference tells us that this data is referenced by the instruction which is at the offset `0x6`, from the start of the `main()` function. The character `w` indncates a *write cross-reference*; This tells us that the instruction writes contents into this memory's location.

A *read cross-reference* is indicated by the character `r`. A *offset cross-reference* is indicated by the character `o`.

Given the following segment of code (Condition): `if (x==0)`, translated to `jnz short loc_401018` when IDA interprets the code, it creates a XREF in the conditional jump such as: `loc_401018:  ;  CODE XREF: _main+Fj`. This indncates the control is transfered from an instruction, which is at offset `0xF` from the start of the main() function. The character `j` indncates that the control whas transfered as a result of the jump.

When a function is called from inside another, IDA automatically prepends `sub_` to the address, to indicate a subroutine, or function.

### 2.3.7 Listing all cross-references

There is a display limit of 2 cross-references. If there are more cross-references, it is indicated by .... To list all the cross-references, click on the location and press `X` key.

If you want to see all the references and calls a function makes, highlight the function and choose *view*.

### 2.3.8 Proximity view and graphs

To view a callgraph of a function, while placing the cursor inside the function, click on view -> Open subviews -> Proximity browser.

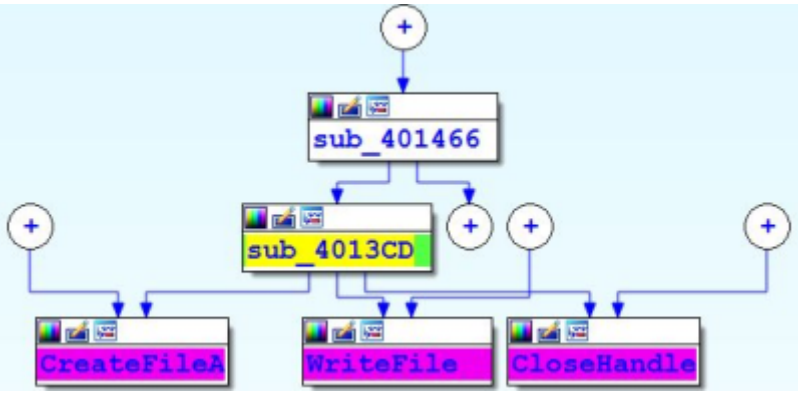


Figure 1: Proximity browser

Apart from this viewing resource, there are more, even provided by third-parties.

## 3. Dissasembling Windows API

Malware uses Windows API. Executables import and call these API functions from various DLLs, which provide different functionalities.

- `Kernel32.dll`: Exports functions related to process, memory, hardware and file system operations.
- `Advapi.dll`: Information relating to service and registry.
- `Dgi.dll`: Exports functions relating to graphics
- `User32.dll`: Create and manipulate Windows user interface components (Desktop, windows, menus, message boxes, prompts...)
- `MSVCRT.dll`: Implementations of C standard library
- `WS2_32.dll` and `WSock32.dll`: Network communication
- `Wininet.dll`: High-level functions to interact with HTTP and FTP
- `Urlmon.dll`: Wrapper around `WinInet.dll`. Responsible for MIME-type handling and downloading web content

- `NTDLL.dll`: Exports windows Native API and acts as a interface between the user mode programs and the Kernel.

## 3.1 Understanding Windows API

The Windows API uses Hungarian notation for naming variables. (Variable prefixed with abbreviation of its data type)

Some data types supported by the Windows API

- `BYTE (b)`: Unsigned 8-bit value
- `WORD (w)`: Unsigned 16-bit value
- `DWORD (dw)`: Unsigned 32-bit value
- `QWORD (qw)`: Unsigned 64-bit value
- `Char (c)`: 8-bit ANSI character
- `WCHAR`: 16-bit Unicode character
- `TCHAR`: Generic character
- `Long Pointer (LP)`: Pointer to another data type (e.g. LPWORD is a pointer to DWORD
- `Handle (H)`: Represents the `handle` data type. A handle is a reference to an object.

Annotations:

- `_In_`: Specifies it is an input parameter
- `_Out_`: Specifies it is an output parameter
- `_Inout_`: Parameter passes vale to the function and receives the output from the function

IDA has a feature called *Fast library Identification and Recognition Technology* which contains pattern-matching algorithms to identify wither the disassembled function is a library or a imported function. (e.g. CreateFile (From Windows API)) IDA also adds names of parameters as comments to indicate which parameter is being pushed at each instruction leading up to the CreateFile Windows API call.

IDA also mantains a list of symbolic constants (e.g. `GENERIC_WRITE`) for the Windows API or the C standard library function.

### 3.1.1 ANSI and Unicode API Functions

Many functions that take a string as a argument include an `A` or `W` at the end of their names (e.g. CreateFileA)

- `A` for ANSI
- `W` for Unicode

### 3.1.2 Extended API Functions

Extended functions are updates of previous functions. When extended, they have the suffix: `Ex` in their name. (e.g. `RegCreateKeyEx` is an extended version of `RegCreateKey`.)

## 3.2 Windows API 32-bit and 64-bit Comparison

We will interpret dissasembly code to understand the operations performed by the malware.

The 32-bit malware calls the `RegOpenKeyEx` API to open a handle to the `Run` registry key. The output parameter `phkResult` is a pointer variable that receives the handle to the opened registry key after the function call.

```
lea ecx, [esp+7E8h+phkResult]
push ecx                    ; phkResult
push 20006h                 ; samDesired
push 0                      ; ulOptions
push offset aSoftwareMicros ; Software\Microsoft\Windows\CurrentVersion\Run
push HKEY_CURRENT_USER      ; hKey
call ds:RegOpenKeyExW
```

Notice all the parameters for the method call, are stored in the stack.

After the malware opens the handle to the `run` registry key by calling `RegOpenKeyEx`, the returned handle (stored in the `phkResult` variable) is moved into the `ecx` register and then passed as the first parameter to `RegSetValueExW`. This sets a value in the `Run` registry. (for persistance) The value is passed as the second parameter, which is the string `System`. The data that it adds to the registry can be datermined by examining the fifth parameter, which is passed in the `eax` register. The `pszPath` varieble is populated with some content during runtime; so, we can't know what this value is.

```
mov ecx, [esp+7E8h+phkResult]
sub eax, edx
sar eax, 1
lea edx, ds:4[eax*4]
push edx              ; cbData
lea eax, [esp+7ECh+pszPath]
push eax              ; lpData
push REG_SZ           ; dwType
```

```
push 0                ; Reserved
push offset ValueName ; "System"
push ecx              ; hKey
call ds:RegSetValueExW
```

After adding the entry to the registry key, the malware closes the handle to the registry key by passing the handle it acquired previously (which was stored in the `phResult` variable) to the `RegCloseKey` API function.

```
mov edx, [esp+7E8h+phkResult]
push edx    ; hKey
call esi    ; RegCloseKey
```

The preceding example demonstrates how malware makes use of multiple Windows API functions to add an entry into the registry key, which will allow it to run automatically when the computer reboots.

In the x64 architecture, the first 4 function parameters are passed to the specialised registers, and the others are passed through the stack. As follows

```
xor r9d, r9                                ; lpSecurityAttributes
lea rcx, [rsp+3B8h+FileName                ; lpFileName
lea r8d, [r9+1]                            ; dwShareMode
mov edx, 40000000h                         ; dwDesiredAccess
mov [rsp+3B8h+dwFlagsAndAttributes], 80h ; dwFlagsAndAttributes
mov rsp+3B8h+dwCreationDisposition], 2   ; lpOverlapped
call cs:createFileW
mov rsi, rax
```

## 4. Patching Binary Using IDA

To modify the program, to reverse-engineer or suit your needs. To save the changes, you have to apply patches. Otherwise, they are stored in the IDA database.

### 4.1 Patching Program Bytes

Consider a code excerpt from the 32-bit malware DLL (TDSS rootkit). This rootkir only executes when it is loaded under the `spoolsv.exe` process. To change this, in order to control better when it is esecuted, you could change the string, from `spool.exe` to `notepad.exe`. This way, you could control when the malware is executed for better debugging.

## 4.2 Patching Instructions

To expand on the previous example imagine we want the malware to execute always. This way, we dont have to wait until the condition is met. To do this, we can change the `jnz` instruction, for the `jz`.


Figure 2: Patching Instructions

# 5. IDA Scripting and Plugins

IDA offers scripting capabilities that give you access to the contents of a database. IDA supports two scripting languages: *IDC* and *Python*.

## 5.1 Executing IDA Scripts

- Directly from `File -> Script command` if you only want to execute a couple of lines.
- Directly from IDA's Output window.

## 5.2 IDAPython

Consists of 3 modules:

- `idaapi`: Provides access to the IDA API
- `idautils`: Provides high-level utility functions for IDA
- `ICD`: An IDC compatibility module

### 5.2.1 Checking The Presence Of CreateFile API

To iterate over the list of functions IDA has observed:

```
import idautils

for addr, name in idautils.Names():
  if "CreateFile" in Name:
    print hex(addr), name
```

Another way of checking the presence of the file:

```
import idc
import idautils
```

```
ea = idc.get_name_ea_simple("CreateFileA")

if ea != idaapi.BADADDR:
  print hex(ea), idc.generate_dissasm_line(ea,0)
else:
  print "Not Found"
```

### 5.2.2 Code Cross-References to Create File Using IDAPython

Retreaving all the addresses where `CreateFileA` is called from:

```
import idc
import idautils

ea = idc.get_name_ea_simple("CreateFileA")

if ea != idaapi.BADADDR:
  for ref in idautils.CodeRefsTo(ea, 1):
    print hex(ref, idc.generate_dissasm_line(ref,0)
```

## 5.3 IDA Plugins

A commercial plugin that is of great value to a malware analyst and reverse engineer is the *Hex-Rays Decompiler* This decompiles the processor code into human-readable C-like pseudocode.