Disassembly using IDA

IY3840 Malicious Software - Lecture 5

Daniele Sgandurra
(daniele.sgandurra@rhul.ac.uk)

Systems and Software Security Lab (S3Lab)
Information Security Group
Royal Holloway, University of London

ROYAL HOLLOWAY UNIVERSITY OF LONDON

Systems & Software Security Lab

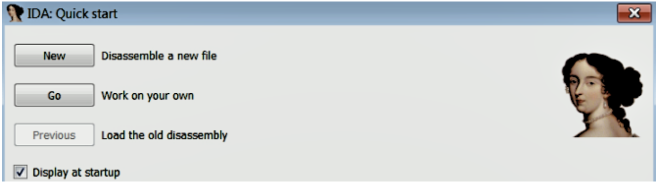
Disassembly using IDA

DISASSEMBLY USING IDA – BASICS

- Hex-Rays IDA is probably the most powerful and popular commercial disassembler/debugger:
  - IDA can run on various platforms (Windows, Linux, and macOS) and supports analysis of various file formats, including the PE/ELF/Macho-O formats
  - commercial version, IDA demo version and IDA Freeware version
  - freeware is IDA 7.0 to disassemble both 32-bit and 64-bit Windows binary (but not to debug it)
- We will learn how to use IDA to perform basic static code analysis (disassembly)
- For more information, it is recommended to the read the book The IDA Pro Book (2nd Edition) by Chris Eagle
- Or, to cover the basics, just Chapter 5 of Learning Malware Analysis (on which this part is based upon)

DISASSEMBLY USING IDA – BASICS II

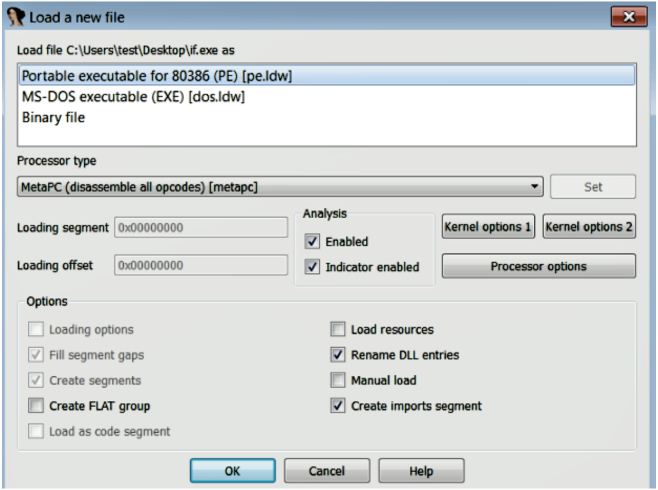
- To load a file, you can either drag and drop or click on `File | Open` and select the file



☑ Display at startup

- The file that you give to IDA will be loaded into the memory (IDA acts like a Windows loader)
- From the file header, IDA determines the processor type that should be used during the disassembly process

DISASSEMBLY USING IDA – BASICS III



Load a new file

Load file C:\Users\test\Desktop\if.exe as

- Portable executable for 80386 (PE) [pe.ldw]
- MS-DOS executable (EXE) [dos.ldw]
- Binary file

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

Loading segment: 0x00000000

Loading offset: 0x00000000

Analysis

- ☑ Enabled
- ☑ Indicator enabled

Options

- □ Loading options
- ☑ Fill segment gaps
- ☑ Create segments
- □ Create FLAT group
- □ Load as code segment
- □ Load resources
- ☑ Rename DLL entries
- □ Manual load
- ☑ Create imports segment

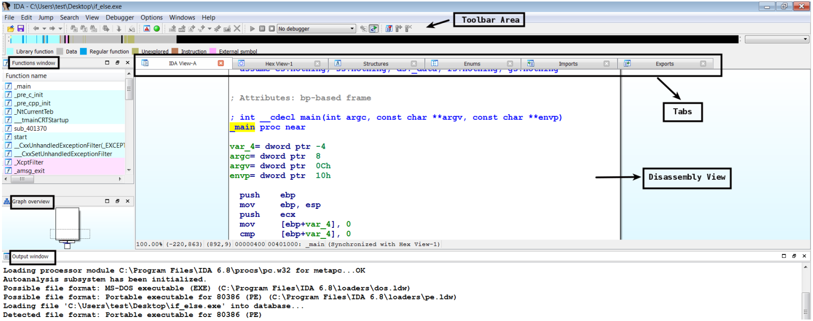
DISASSEMBLY USING IDA – BASICS IV

- From the screenshot, it can be seen that IDA determined the appropriate loaders (pe.ldw and dos.ldw) and the processor type
- The `binary file` option is used by the IDA to load the files that it does not recognize (e.g., shellcode)
- By default, IDA does not load the PE headers and the resource section in the disassembly
- By using the `manual load` checkbox option, you can manually specify the base address where the executable has to be loaded,
    - IDA will prompt you on whether you want to load each section, including the PE headers

DISASSEMBLY USING IDA – BASICS V

- After clicking “OK”, IDA loads the file into memory, and the disassembly engine disassembles the machine code
- After the disassembly, IDA performs an initial analysis to identify the compiler, function arguments, local variables, library functions, and their parameters
- Once the executable has been loaded, you will be taken to the IDA desktop, showing the disassembled output of the program
- The following screenshot shows the IDA desktop after loading an executable file

DISASSEMBLY USING IDA – BASICS VI



Function name

- _main
- _pre_c_init
- _pre_cpp_init
- _NtCurrentTeb
- ___tmainCRTStartup
- sub_401370
- start
- __CxxUnhandledExceptionFilter(_EXCEPT
- ___CxxSetUnhandledExceptionFilter
- _XcptFilter
- _amsg_exit

```
  assume cs:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing


; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near

var_4= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

  push    ebp
  mov     ebp, esp
  push    ecx
  mov     [ebp+var_4], 0
  cmp     [ebp+var_4], 0
```

```
100.00% (-220,863) (892,9) 00000400 00401000: _main (Synchronized with Hex View-1)
```

```
Loading processor module C:\Program Files\IDA 6.8\procs\pc.w32 for metapc...OK
Autoanalysis subsystem has been initialized.
Possible file format: MS-DOS executable (EXE) (C:\Program Files\IDA 6.8\loaders\dos.ldw)
Possible file format: Portable executable for 80386 (PE) (C:\Program Files\IDA 6.8\loaders\pe.ldw)
Loading file 'C:\Users\test\Desktop\if_else.exe' into database...
Detected file format: Portable executable for 80386 (PE)
```

DISASSEMBLY USING IDA – BASICS VII

- The IDA desktop contains multiple tabs (e.g., IDA View-A, Hex View-1):
  - clicking on each tab brings up a different window
  - each window contains different information extracted from the binary.
  - you can also add additional tabs via the `View | Open` Subviews menu
- After the executable has been loaded, you will be presented with the disassembly window (the IDA-view window):
  - this is the primary window, and it displays the disassembled code
  - you will mostly be using this window for analyzing binaries
- IDA can show the disassembled code in two display modes: Graph view and Text view
  - graph view is the default view
  - when the disassembly view (IDA-view) is active, you can switch between the graph and text views by pressing the spacebar button

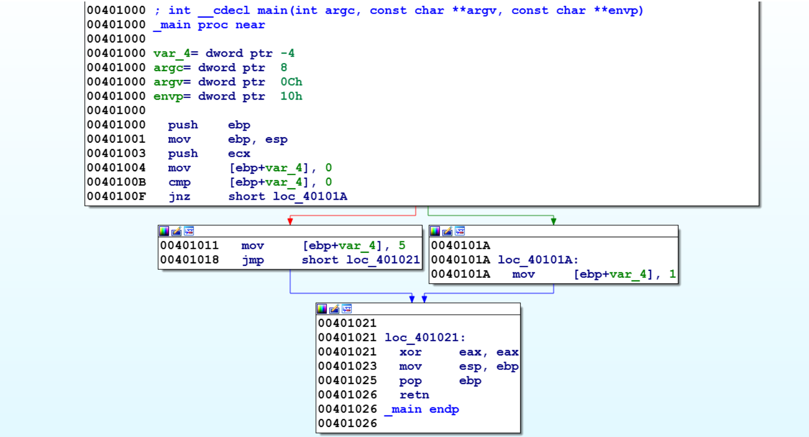
DISASSEMBLY USING IDA – BASICS VIII

- In the graph view mode, IDA displays only one function at a time, in a flowchart-style graph, and the function is broken down into basic blocks:
  - useful to quickly recognize branching and looping statements
- In the graph view mode, the color and the direction of the arrows indicate the path that will be taken, based on a particular decision:
  - the conditional jumps use green and red arrows:
    - the green arrow indicates that the jump will be taken if the condition is true
    - the red arrow indicates that the jump will not be taken (normal flow)
  - the blue arrow is used for an unconditional jump, and the loop is indicated by the upward (backward) blue arrow
- In the graph view, the virtual addresses are not displayed by default (to minimize the amount of space to display basic blocks)
  - to display virtual address information, click on `Options | General` and enable line prefixes

DISASSEMBLY USING IDA – BASICS IX

- The following screenshot shows the disassembly of the `main` function in the graph view mode
- Notice the conditional check at the addresses 0x0040100B and 0x0040100F:
    - if the condition is true, then the control is transferred to the address 0x0040101A (indicated by a green arrow)
    - if the condition is false, the control gets transferred to 0x00401011 (indicated by a red arrow)
    - in other words, the green arrow indicates jump and the red arrow indicates the normal flow

DISASSEMBLY USING IDA – BASICS X



```
00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401000 _main proc near
00401000
00401000 var_4= dword ptr -4
00401000 argc= dword ptr  8
00401000 argv= dword ptr  0Ch
00401000 envp= dword ptr  10h
00401000
00401000   push    ebp
00401001   mov     ebp, esp
00401003   push    ecx
00401004   mov     [ebp+var_4], 0
0040100B   cmp     [ebp+var_4], 0
0040100F   jnz     short loc_40101A
```

```
00401011   mov     [ebp+var_4], 5
00401018   jmp     short loc_401021
```

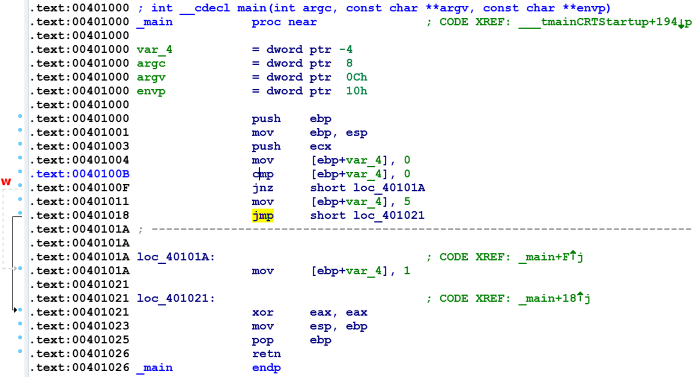
```
0040101A
0040101A loc_40101A:
0040101A   mov     [ebp+var_4], 1
```

```
00401021
00401021 loc_401021:
00401021   xor     eax, eax
00401023   mov     esp, ebp
00401025   pop     ebp
00401026   retn
00401026 _main endp
00401026
```

DISASSEMBLY USING IDA – BASICS XI

- In the text view mode, the entire disassembly is presented in a linear fashion
- The following screenshot shows the text view of the same program
- The virtual addresses are displayed by default, in the `<section name>:<virtual address>` format
- The left-hand portion of the text view window is called the arrows window:
  - it is used to indicate the program's nonlinear flow
  - the dashed arrows represent conditional jumps
  - the solid arrows indicate unconditional jumps
  - the backward arrows (arrows facing up) indicate loops

DISASSEMBLY USING IDA – BASICS XII



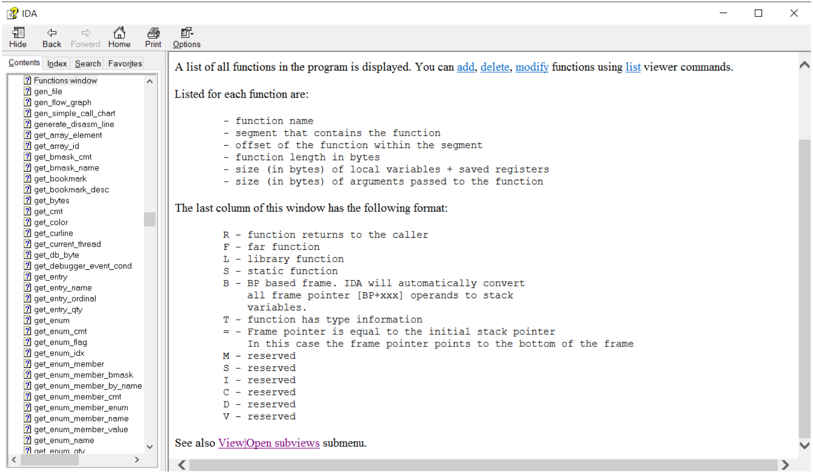
Arrows Window

```
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 _main           proc near               ; CODE XREF: ___tmainCRTStartup+194↓p
.text:00401000
.text:00401000 var_4           = dword ptr -4
.text:00401000 argc            = dword ptr  8
.text:00401000 argv            = dword ptr  0Ch
.text:00401000 envp            = dword ptr  10h
.text:00401000
.text:00401000                 push    ebp
.text:00401001                 mov     ebp, esp
.text:00401003                 push    ecx
.text:00401004                 mov     [ebp+var_4], 0
.text:0040100B                 cmp     [ebp+var_4], 0
.text:0040100F                 jnz     short loc_40101A
.text:00401011                 mov     [ebp+var_4], 5
.text:00401018                 jmp     short loc_401021
.text:0040101A ; ---------------------------------------------------------------------------
.text:0040101A
.text:0040101A loc_40101A:                             ; CODE XREF: _main+F↑j
.text:0040101A                 mov     [ebp+var_4], 1
.text:00401021
.text:00401021 loc_401021:                             ; CODE XREF: _main+18↑j
.text:00401021                 xor     eax, eax
.text:00401023                 mov     esp, ebp
.text:00401025                 pop     ebp
.text:00401026                 retn
.text:00401026 _main           endp
```

DISASSEMBLY USING IDA – BASICS XIII

- The functions window displays all the functions recognized by IDA
  - as well as the virtual address, their size and various other properties
- You can double-click on any of these functions to jump to a selected function
- Each function is associated with various flags (such as R, F, L, and so on):
  - e.g., L flag indicates that the function is a library function
  - you can get more information about these flags in the help file (by pressing F1), e.g. see next screenshot

DISASSEMBLY USING IDA – BASICS XIV



A list of all functions in the program is displayed. You can add, delete, modify functions using list viewer commands.

Listed for each function are:

```
        - function name
        - segment that contains the function
        - offset of the function within the segment
        - function length in bytes
        - size (in bytes) of local variables + saved registers
        - size (in bytes) of arguments passed to the function
```

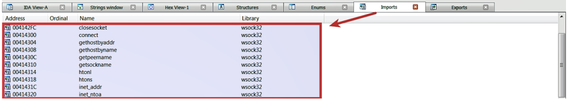
The last column of this window has the following format:

```
        R - function returns to the caller
        F - far function
        L - library function
        S - static function
        B - BP based frame. IDA will automatically convert
            all frame pointer [BP+xxx] operands to stack
            variables.
        T - function has type information
        = - Frame pointer is equal to the initial stack pointer
            In this case the frame pointer points to the bottom of the frame
        M - reserved
        S - reserved
        I - reserved
        C - reserved
        D - reserved
        V - reserved
```

See also View|Open subviews submenu.

DISASSEMBLY USING IDA – BASICS XV

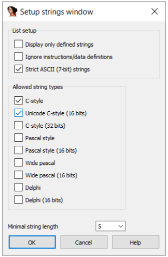
- The output window displays the messages generated by IDA (e.g., various operations performed when an executable is loaded)
- The hex window displays a sequence of bytes in a hex dump and the ASCII format:
    - useful to inspect the contents of the memory address
    - by default, synchronized with the disassembly window (e.g., the corresponding bytes are highlighted)
- The structures window lists the layout of the standard data structures used in the program
- The imports window lists all of the functions imported, while the exports window lists all of the exported functions (e.g., in a DLL)



Address	Ordinal	Name	Library
004142FC		closesocket	wsock32
00414300		connect	wsock32
00414304		gethostbyaddr	wsock32
00414308		gethostbyname	wsock32
0041430C		getpeername	wsock32
00414310		getsockname	wsock32
00414314		htonl	wsock32
00414318		htons	wsock32
0041431C		inet_addr	wsock32
00414320		inet_ntoa	wsock32

DISASSEMBLY USING IDA – BASICS XVI

- The strings window can be brought up by clicking on `View | Open Subviews | Strings` (or `Shift + F12`):
    - displays the list of strings extracted from the binary and the address where these strings can be found
    - by default, only null-terminated ASCII strings of at least five characters
    - to configure IDA to, e.g., show UNICODE strings, right-click on Setup (or `Ctrl + U`), check `Unicode C-style (16 bits)`, and click OK



Setup strings window

List setup

☐ Display only defined strings
☐ Ignore instructions/data definitions
☑ Strict ASCII (7-bit) strings

Allowed string types

☑ C-style
☑ Unicode C-style (16 bits)
☐ C-style (32 bits)
☐ Pascal style
☐ Pascal style (16 bits)
☐ Wide pascal
☐ Wide pascal (16 bits)
☐ Delphi
☐ Delphi (16 bits)

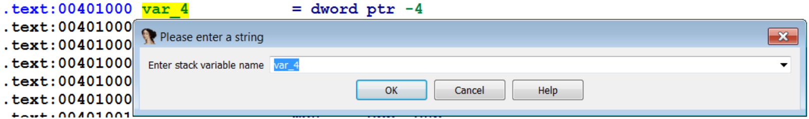
Minimal string length: 5

DISASSEMBLY USING IDA – BASICS XVII

- Finally, the segments window is available via `View | Open Subviews | Segments` (or `Shift + F7`)
- It lists the sections (`.text`, `.data`, etc) in the binary file
- It contains the start address, the end address, and the memory permissions of each section:
  - the start and end address specify the virtual address of each section that is mapped into memory during runtime

DISASSEMBLY USING IDA – BASICS XVIII

- When analysing malware, you should change the variable/function names to more meaningful names
- To rename a variable or an argument, right-click on the variable name or argument and select `rename` (or press N):
    - IDA will propagate the new name to wherever that item is referenced
    - you can rename functions and variables



```
.text:00401000 var_4           = dword ptr -4
.text:00401000
.text:00401000
.text:00401000
.text:00401000
.text:00401000
```

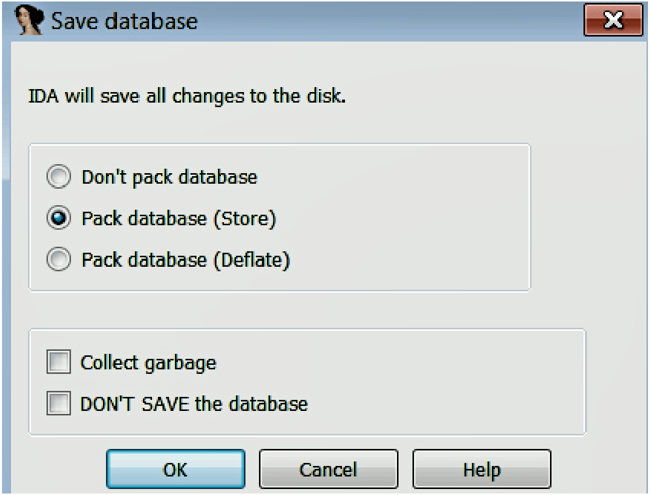
Please enter a string

Enter stack variable name var_4

DISASSEMBLY USING IDA – BASICS XIX

- When an executable is loaded, it creates a database consisting of five files (extensions: `.id0`, `.id1`, `.nam`, `.id2`, and `.til`)
- Each of these files stores various information and has a base name that matches the selected executable
- Upon loading the executable, the database is created and populated with the information from the executable files
- The various displays that are presented to you are simply views into the database
- Any modifications (e.g., renaming) are reflected in the views and saved in the database:
  - these changes do not modify the original executable file
  - when you close IDA, you will be presented with a `Save database` dialog
  - the `Pack database` option (default) archives all of the files into a single IDB (`.idb`) or i64 (`.i64`) file

DISASSEMBLY USING IDA – BASICS XX



Save database

IDA will save all changes to the disk.

○ Don't pack database

● Pack database (Store)

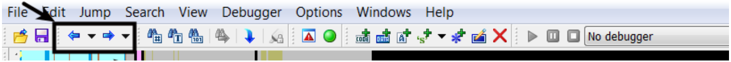
○ Pack database (Deflate)

□ Collect garbage

□ DON'T SAVE the database

DISASSEMBLY USING IDA – BASICS XXI

- When a program is disassembled, IDA labels every location in the program:
  - double-clicking on the locations will jump the display to the selected location
- IDA keeps track of your navigation history:
  - any time you navigate to a new location and would like to go back to your original position, you can use the navigation buttons



DISASSEMBLY USING IDA – BASICS XXII

- Another way to navigate is by using cross-references (also referred to as Xrefs)
- The cross-references link relates addresses together
- Cross-references can be either data cross-references or code cross-references
- A data cross-reference specifies how the data is accessed within a binary:
    - write cross-reference (w)
    - read cross-reference (r)
    - offset cross-reference (o)
- A code cross-reference indicates the control flow from one instruction to an another (such as jump or function call)

DISASSEMBLY USING IDA – BASICS XXIII

Example:

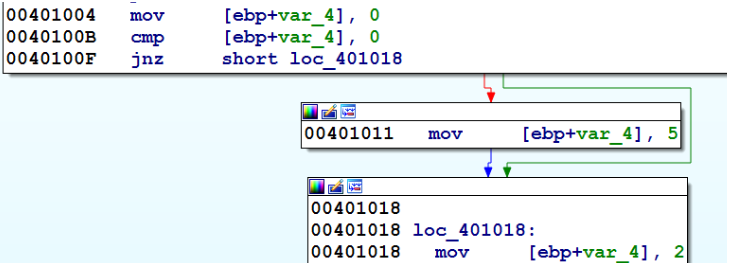
```
int x = 0;
if (x == 0)
{
    x = 5;
}
x = 2;
```

```
.text:00401004     mov [ebp+var_4], 0
.text:0040100B     cmp [ebp+var_4], 0
.text:0040100F     jnz short loc_401018 ①
.text:00401011     mov [ebp+var_4], 5
.text:00401018
.text:00401018     loc_401018:  ③; CODE XREF: _main+Fj
.text:00401018     ② mov [ebp+var_4], 2
```

- The jump cross-reference comment is shown at the jump target ③
- It indicates that the control is transferred from an instruction, which is at the offset `0xF` from the start of the `main` function (in other words, ①)

DISASSEMBLY USING IDA – BASICS XXIV

The preceding listing can be viewed in the graph view mode by pressing the spacebar key



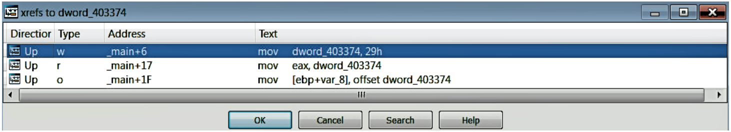
```
00401004    mov     [ebp+var_4], 0
0040100B    cmp     [ebp+var_4], 0
0040100F    jnz     short loc_401018
```

```
00401011    mov     [ebp+var_4], 5
```

```
00401018
00401018 loc_401018:
00401018    mov     [ebp+var_4], 2
```

DISASSEMBLY USING IDA – BASICS XXV

- Cross-references are very useful when analyzing malicious binary
- If you come across a string or a useful function, you can use cross-references to quickly navigate to the location where the string or function is referenced
- To list all of the cross-references, click on the named location, such as `dword_403374`, and press the X key

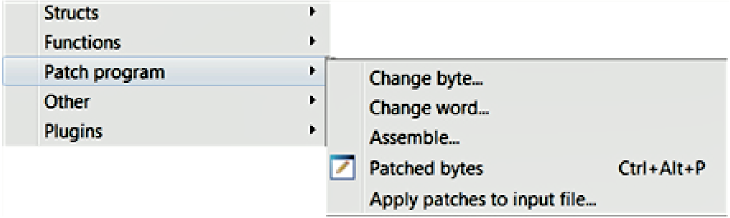


xrefs to dword_403374

Direction	Type	Address	Text
Up	w	_main+6	mov dword_403374, 29h
Up	r	_main+17	mov eax, dword_403374
Up	o	_main+1F	mov [ebp+var_8], offset dword_403374

DISASSEMBLY USING IDA – BASICS XXVI

- When performing malware analysis, you may want to modify the binary to change its inner workings or reverse its logic to suit your needs
- Using IDA, it is possible to modify the data or instructions of a program
- You can perform patching by selecting `Edit | Patch` program menu
  - using the submenu items, you can modify a byte, word, or assembly instructions



DISASSEMBLY USING IDA – BASICS XXVII

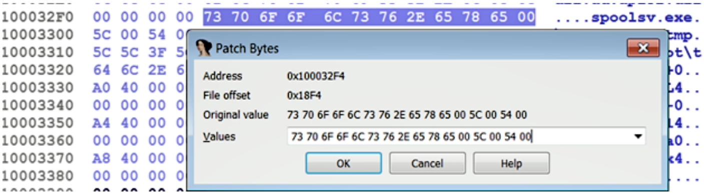
- Consider the code excerpt from the 32-bit malware DLL (TDSS rootkit)
    - it performs a check to make sure that it is running under `spoolsv.exe`
- This check is performed using string comparison at ①
    - if the string comparison fails, then the code jumps to end of the function ②
    - it generates malicious behavior only when it is loaded by `spoolsv.exe`

```
10001BF2      push offset aSpoolsv_exe  ; "spoolsv.exe"
10001BF7      push edi                  ; char *
10001BF8      call _stricmp  ①
10001BFD      test eax, eax
10001BFF      pop ecx
10001C00      pop ecx
10001C01      jnz loc_10001CF9

[REMOVED]

10001CF9 loc_10001CF9: ②      ; CODE XREF: DllEntryPoint+10j
10001CF9       xor  eax, eax
10001CFB       pop  edi
10001CFC       pop  esi
10001CFD       pop  ebx
10001CFE       leave
10001CFF       retn 0Ch
```

DISASSEMBLY USING IDA – BASICS XXVIII

- Suppose you want the malicious DLL to generate the behavior on any other process, such as `notepad.exe`
- You can change the hardcoded string from `spoolsv.exe` to `notepad.exe`
- To do that, navigate to the hardcoded address by clicking on `aSpoolsv_exe`:
  - place your mouse cursor on the variable name (`aSpoolsv_exe`)
  - the hex view window should be synchronized with this address
  - clicking on the Hex View-1 tab displays the hex and ASCII dump of this memory address
- To patch the bytes, select `Edit | Patch program | Change byte`
  - this will bring up the patch bytes dialog shown in the following screenshot

DISASSEMBLY USING IDA – BASICS XXIX



```
100032F0  00 00 00 00 73 70 6F 6F  6C 73 76 2E 65 78 65 00  ....spoolsv.exe.
10003300  5C 00 54 0                                         tmp.
10003310  5C 5C 3F 5                                         ot\t
10003320  64 6C 2E 6                                         +0..
10003330  A0 40 00 0                                         L4..
10003340  00 00 00 0                                         +0..
10003350  A4 40 00 0                                         14..
10003360  00 00 00 0                                         a0..
10003370  A8 40 00 0                                         x4..
10003380  00 00 00 0                                         ....
```

Patch Bytes

Address 0x100032F4

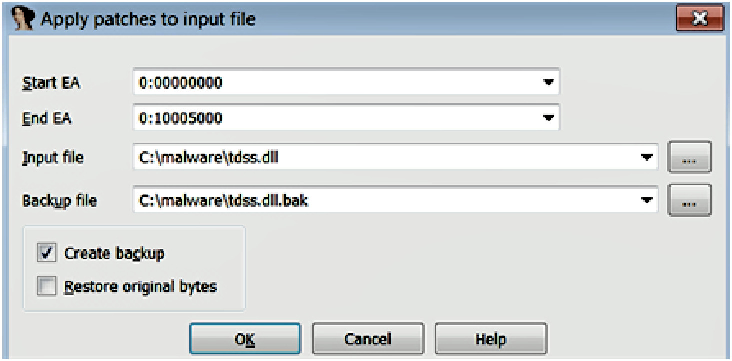
File offset 0x18F4

Original value 73 70 6F 6F 6C 73 76 2E 65 78 65 00 5C 00 54 00

Values 73 70 6F 6F 6C 73 76 2E 65 78 65 00 5C 00 54 00

- The modification that you make is applied to the IDA database:
    - to apply the changes to the original executable file, you can select `Edit | Patch program | Apply` patches to the input file

DISASSEMBLY USING IDA – BASICS XXX



Start EA: 0:00000000

End EA: 0:10005000

Input file: C:\malware\tdss.dll

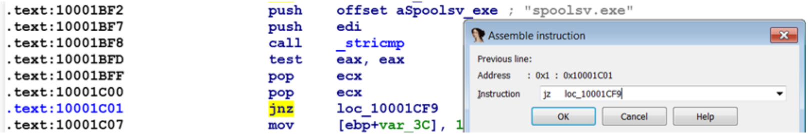
Backup file: C:\malware\tdss.dll.bak

☑ Create backup

☐ Restore original bytes

DISASSEMBLY USING IDA – BASICS XXXI

- Similarly, we can change the `jnz` instruction to `jz` by selecting `Edit | Patch program | Assemble`, as shown in the following screenshot



```
.text:10001BF2                 push    offset aSpoolsv_exe ; "spoolsv.exe"
.text:10001BF7                 push    edi
.text:10001BF8                 call    _stricmp
.text:10001BFD                 test    eax, eax
.text:10001BFF                 pop     ecx
.text:10001C00                 pop     ecx
.text:10001C01                 jnz     loc_10001CF9
.text:10001C07                 mov     [ebp+var_3C], 1
```

Assemble instruction

Previous line:

Address : 0x1 : 0x10001C01

Instruction jz loc_10001CF9

- Please note that, when patching an instruction, care needs to be taken to make sure that the instruction alignment is correct:
    - otherwise, the patched program may exhibit unexpected behavior
    - If the new instruction is shorter than the instruction you are replacing, then `nop` instructions can be inserted to keep the alignment intact

References

REFERENCES

[1] Learning Malware Analysis. Monnappa K A. June. 2018
Chapter 5 (*Available on Safari Online and library*)