

# CSec15233

# Malicious Software Analysis

## IDA Pro



Qasem Abu Al-Haija

# IDA Pro Introduction

- The Interactive Disassembler (IDA) ([Help Index](#)).
  - <https://hex-rays.com/ida-free/#download>
- Extremely powerful disassembler by HexRays.
- The disassembler of choice for:
  - Malware analysts,
  - Reverse engineers, and
  - Vulnerability analysts.

# IDA Pro Introduction

- Supports several PE file formats and some others.
  - Such as Executable and Linking Format (ELF).
- Disassembles an entire program and performs several tasks such as:
  - **Function Discovery, Stack Analysis, Local Variable Identification,...** etc.
- Alternative: radare2

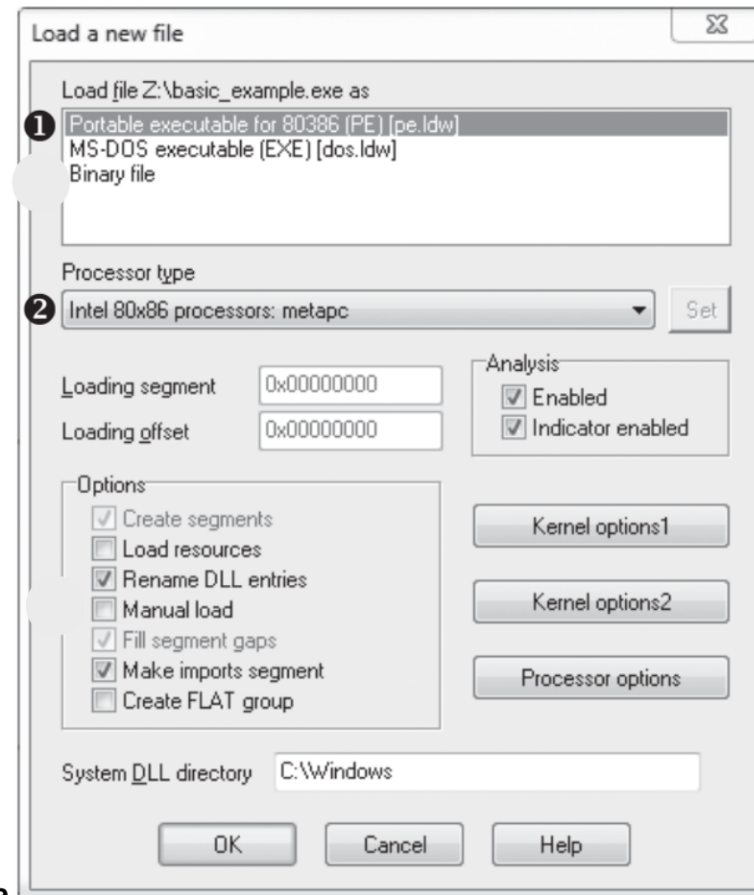
# IDA Pro Versions

- Two versions of IDA Pro are commercially available
  - Full-featured pay version and Old free version
  - Both support x86
  - Pay version supports x64 and other processors, such as cell phone processors.
  - Both have code signatures for common library code in FLIRT
  - FLIRT: Fast Library Identification and Recognition Technology
  - This allows it to recognize and label a disassembled function, especially library code added by a compiler.

# Loading an Executable

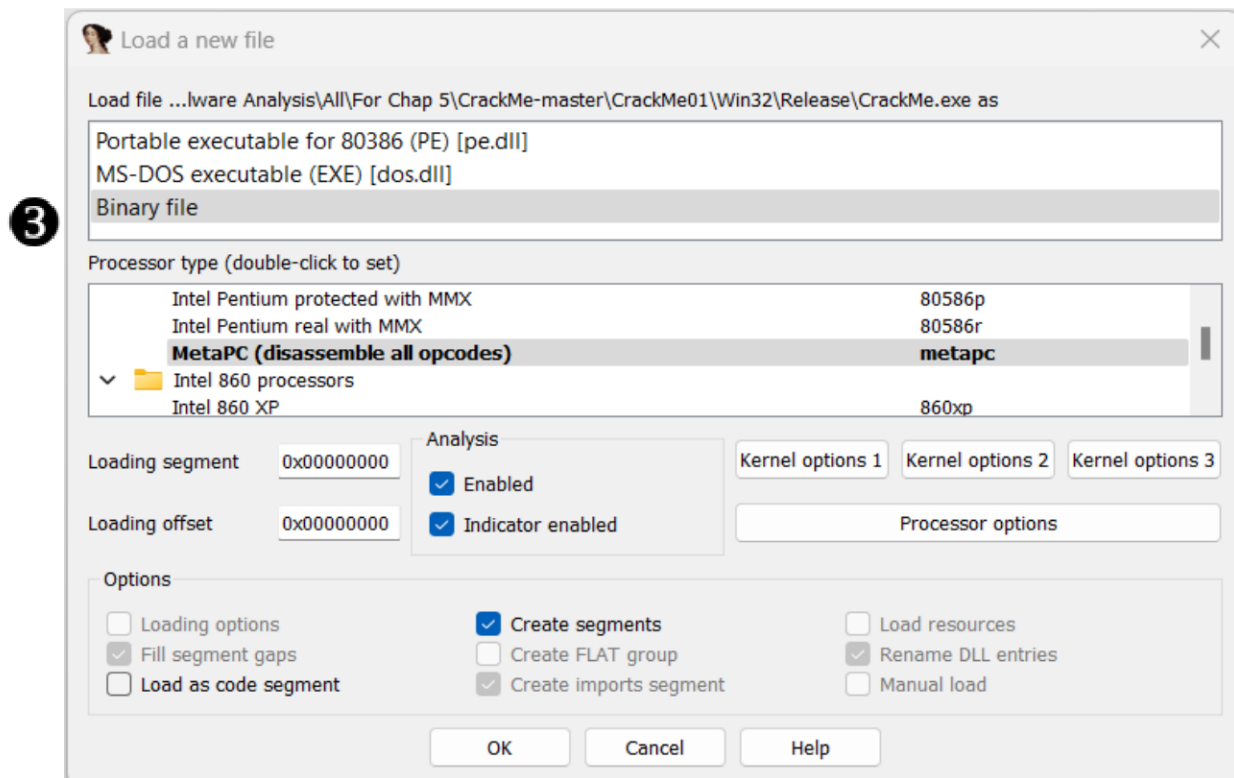
- When you load an executable on IDA Pro:

- IDA Pro will try to recognize the file's format and processor architecture.
- For example (Figure), the file is recognized as having the PE format ❶ with Intel x86 architecture. ❷



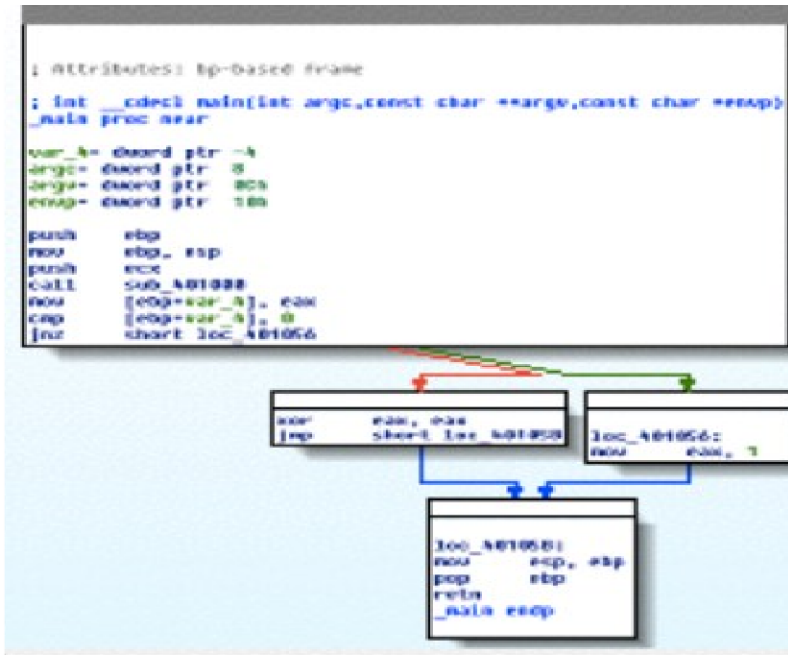
# Loading an Executable

- When loading a file into IDA Pro (such as a PE file), the program maps the file into memory
  - as if the operating system loader had loaded it.
- To have IDA Pro disassemble the file as a raw binary, choose the Binary File option in the top box ③



# IDA Pro - Interface

- Graph and Text Mode
- Spacebar Switches mode

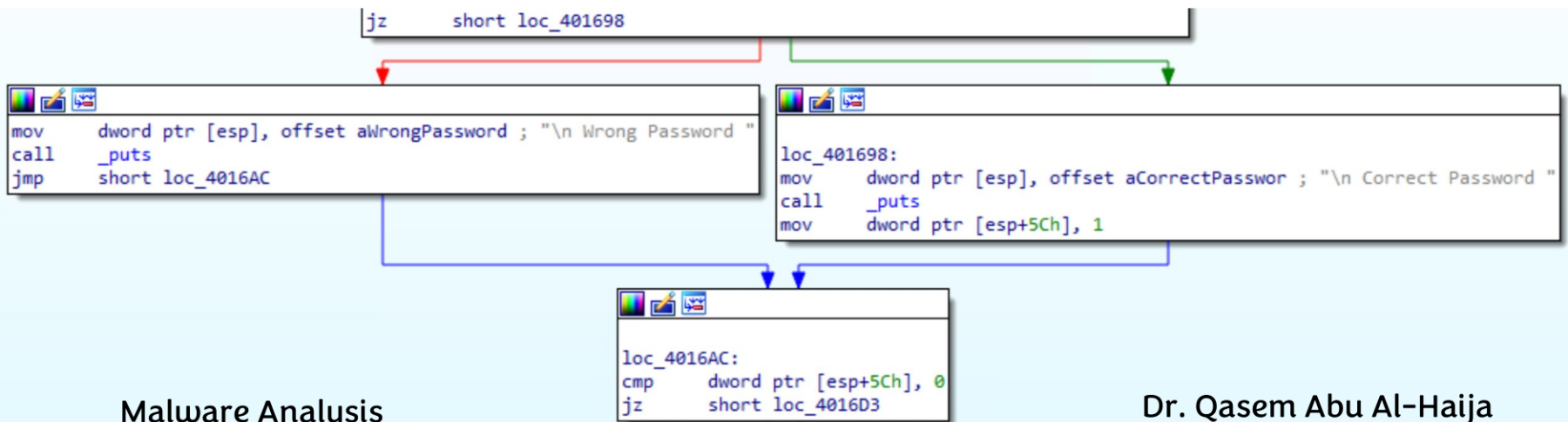


The screenshot shows the IDA View-A window with the following assembly code:

```
.text:00401040 ; Attributes: bp-based frame
.text:00401040 ; int __cdecl main(int argc,const char **argv,const char *envp)
.text:00401040 _main proc near ; CODE XREF: start+AF1p
.text:00401040 var_4 = dword ptr -4
.text:00401040 argc = dword ptr 8
.text:00401040 argv = dword ptr 0Ch
.text:00401040 envp = dword ptr 10h
.text:00401040
.text:00401040 push ebp
.text:00401041 mov ebp, esp
.text:00401043 push ecx
```

# Graph Mode Interface

- Provides a control flow graph to help you understand how the program works
- In graph mode, IDA Pro excludes certain information, such as line numbers and operation codes (i.e., binary code)
  - To change these options, select **Options->General**, and then select **Line prefixes** and set the **Number of Opcode Bytes** to **6**.





# Default Graph Mode Display

```

; Attributes: bp-based frame

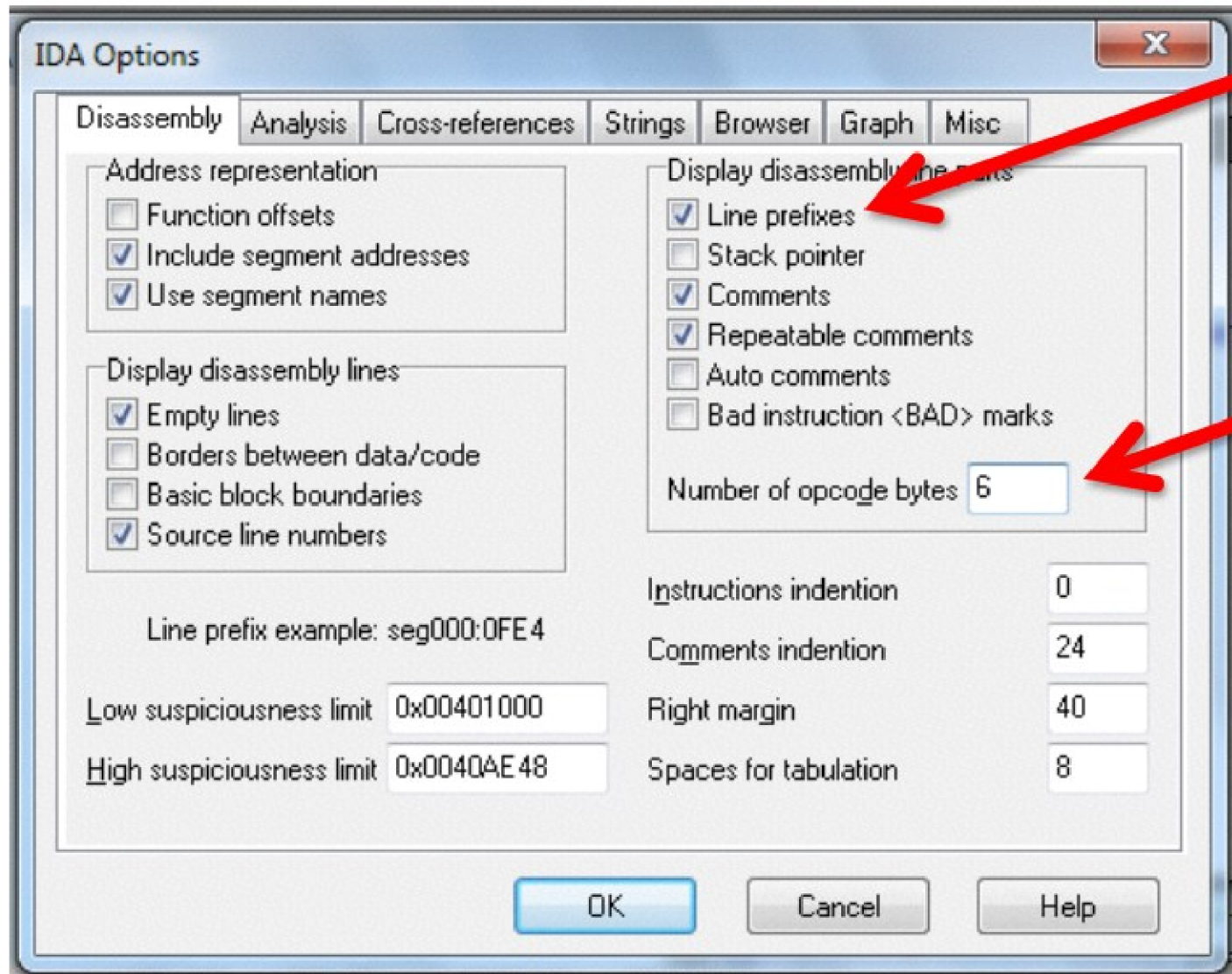
; int __cdecl main(int argc,const char **argv,const char *envp)
_main proc near

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

push    ebp
mov     ebp, esp
push    ecx
call   sub_401000
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 0
jnz    short loc_401056

```

# Options, General



# Better Graph Mode Display

```
00401040
00401040
00401040      ; Attributes: bp-based frame
00401040
00401040      ; int __cdecl main(int argc,const char **argv,const char *envp)
00401040      _main proc near
00401040
00401040      var_4= dword ptr -4
00401040      argc= dword ptr  8
00401040      argv= dword ptr  0Ch
00401040      envp= dword ptr  10h
00401040
00401040 55          push    ebp
00401041 8B EC      mov     ebp, esp
00401043 51          push    ecx
00401044 E8 B7 FF FF FF  call   sub_401000
00401049 89 45 FC      mov     [ebp+var_4], eax
0040104C 83 7D FC 00    cmp     [ebp+var_4], 0
00401050 75 04        jnz    short loc_401056
```

# Graph Mode: Notes

- The color and direction of the arrows help show the program's flow during analysis.
- The arrow's color tells you whether the path is based on a particular decision having been made:
  - **red** if a conditional **jump is not taken**,
  - **green** if the **jump is taken**, and
  - **blue** for an **unconditional** jump.
- The arrow direction shows the program's flow;
  - upward arrows typically denote a loop situation.

IDA - C:\f\w\CAP4145\Practical Malware Analysis Labs\BinaryCollection\Chapter\_1L\Lab01-04.exe

File Edit Jump Search View Debugger Options Windows Help

1234h

Open subviews

- Graphs
- Toolbars
- Calculator...
- Print segment registers
- Print internal flags
- Hide
- Unhide
- Hide all
- Unhide all
- Delete hidden area
- Setup hidden items

IDA View-A

```

_main proc near
var_145C= dword ptr -145Ch
ExistingFileName= byte ptr -145h
NewFileName= byte ptr -1348h
var_1238= dword ptr -1238h
var_1234= dword ptr -1234h
var_1230= dword ptr -1230h
var_122C= dword ptr -122Ch
var_1228= dword ptr -1228h
dwProcessId= dword ptr -1224h
Buffer= byte ptr -224h
var_114= dword ptr -114h
var_110= dword ptr -110h

push    ebp
mov     ebp, esp
mov     eax, 145Ch
call   __alloca_probe
push    edi
mov     [ebp+var_114], 0
mov     byte ptr [ebp+var_110]
mov     ecx, 43h
xor     eax, eax

```

Graph overview

Names window

Name

- F \_main
- L \_\_alloca\_probe
- L start
- F \_XcptFilter
- F \_initterm
- L \_setdefaultprecision
- F nullsub\_1
- L \_except\_handler3

Line 1 of 60

Strings window

Address	Length	T...	String
... .rdata:0...	0000000C	C	CloseHan
... .rdata:0...	0000000C	C	OpenProc
... .rdata:0...	00000012	C	GetCurren
... .rdata:0...	00000013	C	CreatePer
... .rdata:0...	0000000F	C	GetProcA
... .rdata:0...	0000000D	C	LoadLibr
... .rdata:0...	00000008	C	WinExec
... .rdata:0...	0000000A	C	WriteFile

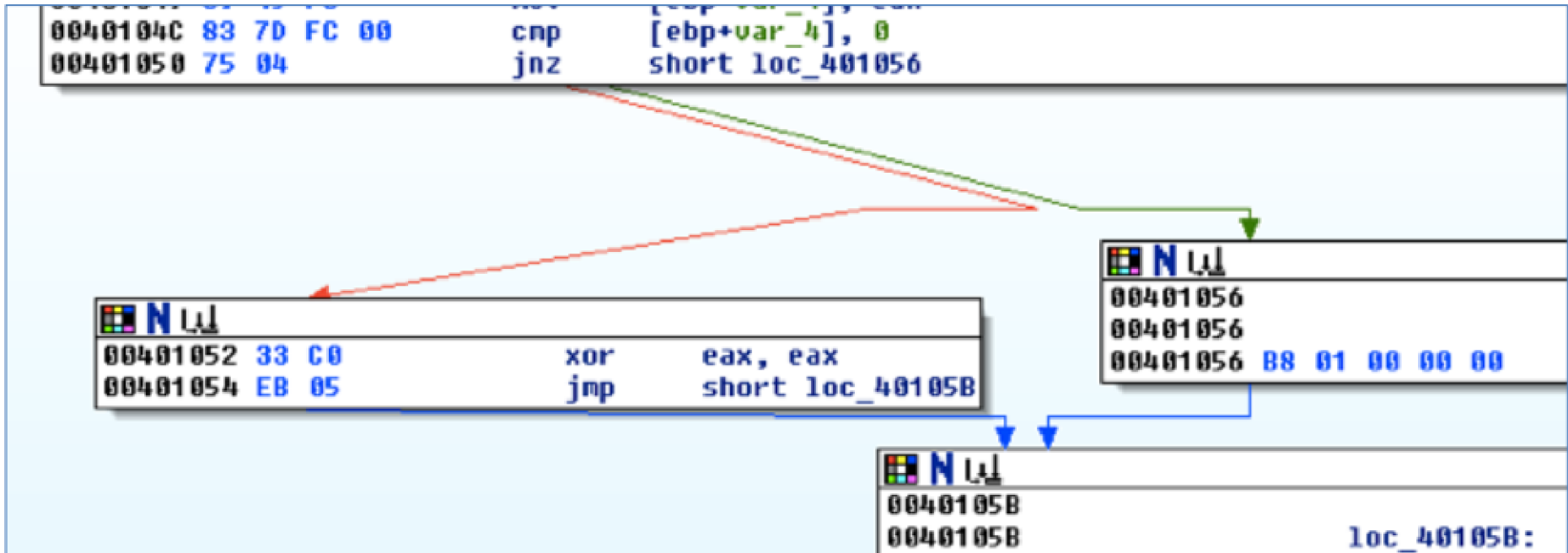
Compiling file 'C:\Program Files\IDA Free\idc\ida.idc'...  
 Executing function 'main'...  
 Compiling file 'C:\Program Files\IDA Free\idc\onload.idc'...  
 Executing function 'OnLoad'...  
 IDA is analysing the input file...  
 You may start to explore the input file right now.  
 Using FLIRT signature: Microsoft VisualC 2-8/net runtime  
 Propagating type information...  
 Function argument information is propagated  
 The initial autoanalysis has been finished.

AU: idle | Down | Disk: 120GB

start | Help | IDA - C:\f\w\CAP414...

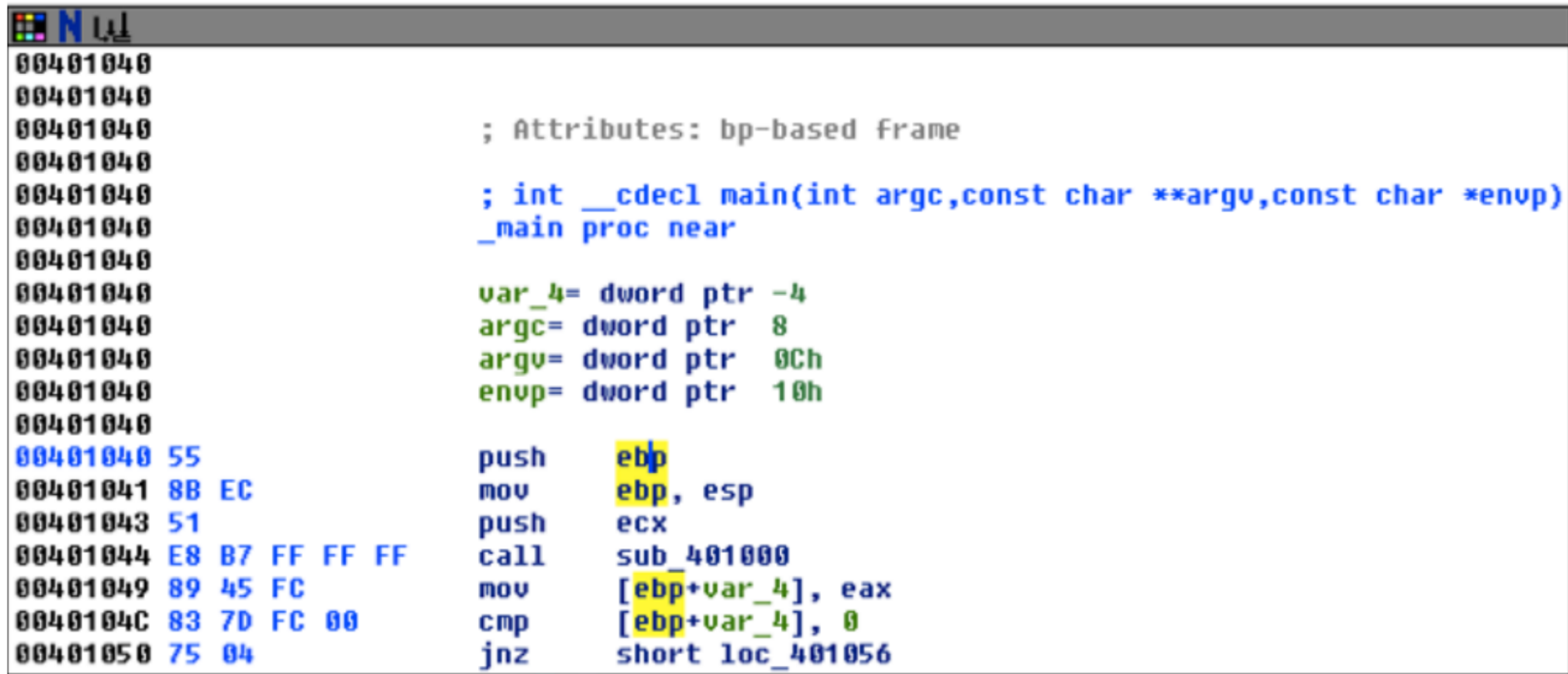
10:41 AM

# Arrow Color Example



# Highlighting

- Highlighting text in graph mode highlights every instance of that text

A screenshot of a debugger window showing assembly code. The window has a title bar with a small icon and the text 'N'. The code is displayed in a monospaced font. The first few lines are comments: '; Attributes: bp-based frame' and '; int \_\_cdecl main(int argc,const char \*\*argv,const char \*envp) \_main proc near'. The next four lines are variable declarations: 'var\_4= dword ptr -4', 'argc= dword ptr 8', 'argv= dword ptr 0Ch', and 'envp= dword ptr 10h'. The final four lines are instructions: 'push ebp', 'mov ebp, esp', 'push ecx', and 'call sub\_401000'. The instruction 'push ebp' is highlighted in yellow. The address column on the left shows addresses from 00401040 to 00401050. The instruction column shows mnemonics and operands. The operand 'ebp' in the 'push ebp' instruction is highlighted in yellow. The operand '[ebp+var\_4]' in the 'mov' and 'cmp' instructions is also highlighted in yellow. The operand '0' in the 'cmp' instruction is highlighted in yellow. The operand 'short loc\_401056' in the 'jnz' instruction is highlighted in yellow. The instruction 'push ebp' is highlighted in yellow. The instruction 'mov ebp, esp' is highlighted in yellow. The instruction 'push ecx' is highlighted in yellow. The instruction 'call sub\_401000' is highlighted in yellow. The instruction 'mov [ebp+var\_4], eax' is highlighted in yellow. The instruction 'cmp [ebp+var\_4], 0' is highlighted in yellow. The instruction 'jnz short loc\_401056' is highlighted in yellow.

```
00401040
00401040
00401040      ; Attributes: bp-based frame
00401040
00401040      ; int __cdecl main(int argc,const char **argv,const char *envp)
00401040      _main proc near
00401040
00401040      var_4= dword ptr -4
00401040      argc= dword ptr 8
00401040      argv= dword ptr 0Ch
00401040      envp= dword ptr 10h
00401040
00401040 55          push     ebp
00401041 8B EC      mov     ebp, esp
00401043 51          push     ecx
00401044 E8 B7 FF FF FF  call   sub_401000
00401049 89 45 FC      mov     [ebp+var_4], eax
0040104C 83 7D FC 00    cmp     [ebp+var_4], 0
00401050 75 04        jnz     short loc_401056
```

# Text Mode Interface

- **Text Mode** – the traditional view
  - Use it to view data regions of a binary

**Arrows**  
Solid = Unconditional  
Dashed = Conditional  
Up = Loop

**Section**  
**Address**

## Text Mode

**Comment Generated by IDA Pro**

```

-- .text:00401015      jz      short loc_40102B
. .text:00401017      push   offset aSuccessInterne ; "Success: Internet Connection\n"
. .text:0040101C      call   sub_40105F
. .text:00401021      add    esp, 4
. .text:00401024      mov    eax, 1
. .text:00401029      jmp    short loc_40103A
. .text:0040102B      ; -----
. .text:0040102B      loc_40102B: ; CODE XREF: sub_401000+15↑j
. .text:0040102B      push   offset aError1_1NoInte ; "Error 1.1: No Internet\n"
. .text:00401030      call   sub_40105F
. .text:00401035      add    esp, 4
. .text:00401038      xor    eax, eax
. .text:0040103A      loc_40103A: ; CODE XREF: sub_401000+29↑j
. .text:0040103A      mov    esp, ebp
. .text:0040103C      pop    ebp

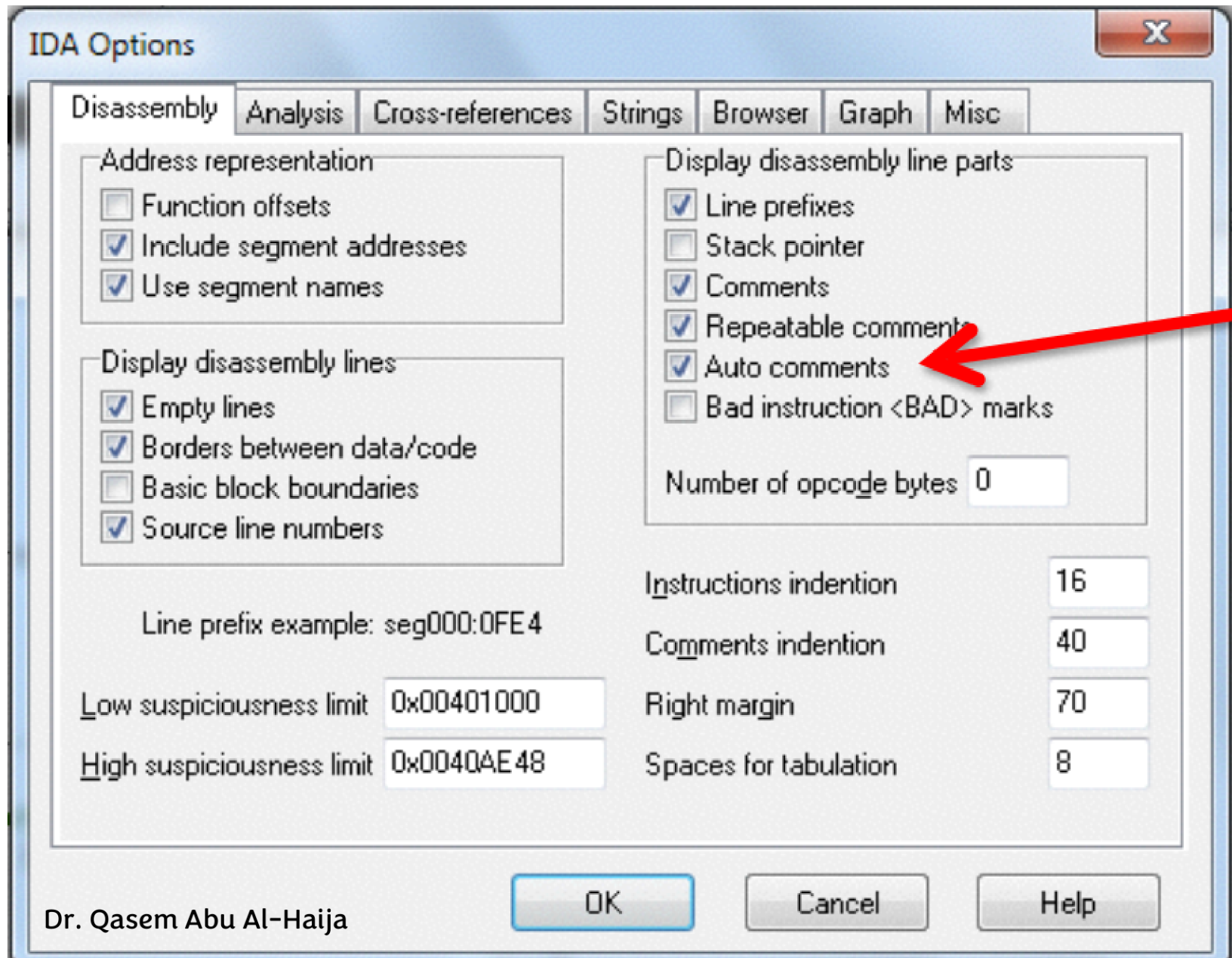
```



# Text Mode: Notes

- The left portion of the text-mode display is known as the arrows window and shows the program's *nonlinear flow*.
  - Solid lines mark unconditional jumps
  - Dashed lines mark conditional jumps
  - **Arrows facing up** indicate a loop
- A comment (beginning with a semicolon) that IDA Pro automatically added

# Options, General



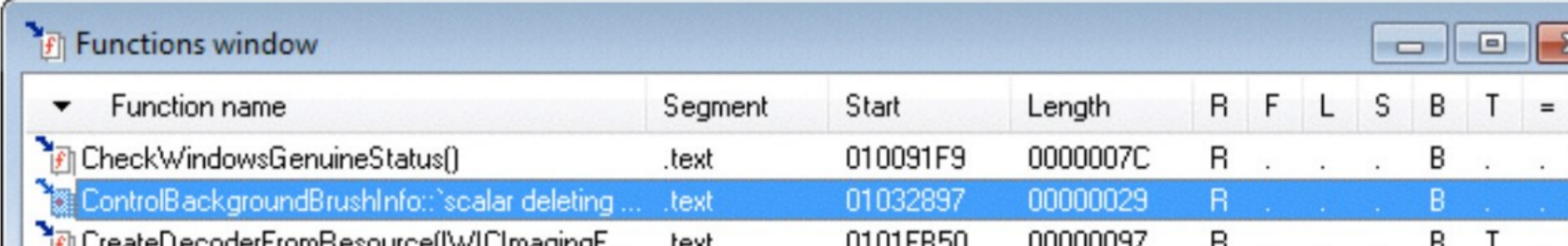
# Adds Comments to Each Instruction

```
• .text:00401015      jz      short loc_40102B ; Jump if Zero (ZF=1)
• .text:00401017      push   offset aSuccessInterne ; "Success: Internet Connection\n"
• .text:0040101C      call   sub_40105F          ; Call Procedure
• .text:00401021      add    esp, 4              ; Add
• .text:00401024      mov    eax, 1              ; Add
• .text:00401029      jmp    short loc_40103A    ; Jump
• .text:0040102B ; -----
• .text:0040102B      loc_40102B:                ; CODE XREF: sub_401000+15↑j
• .text:0040102B      push   offset aError1_1NoInte ; "Error 1.1: No Internet\n"
• .text:00401030      call   sub_40105F          ; Call Procedure
• .text:00401035      add    esp, 4              ; Add
• .text:00401038      |                          ; Logical Exclusive OR
• .text:0040103A      xor    eax, eax
• .text:0040103A      loc_40103A:                ; CODE XREF: sub_401000+29↑j
• .text:0040103A      mov    esp, ebp
• .text:0040103C      pop    ebp
```

# Useful Windows for Analysis

# Functions window

- Shows every function, its length, and flags.
- The most useful flag is L: **indicates library functions.**
  - The L flag can save you time during analysis because you can identify and skip these compiler-generated functions.
- Sortable: Large functions usually more important

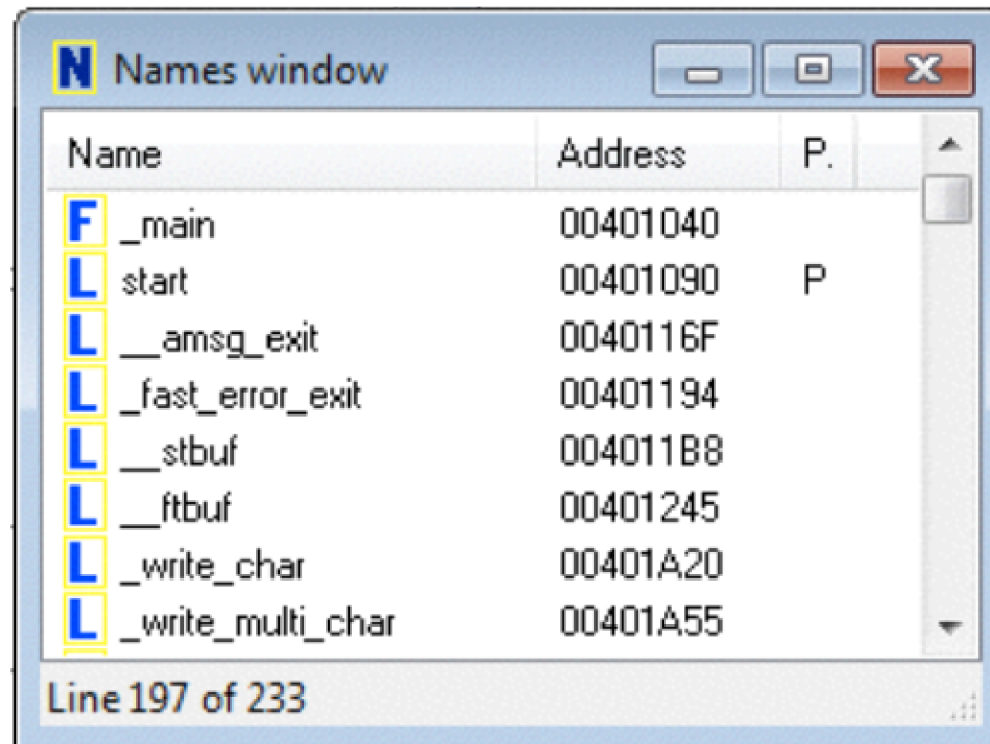


The screenshot shows a window titled "Functions window" with a table of functions. The table has columns for Function name, Segment, Start, Length, and flags (R, F, L, S, B, T, =). The function "ControlBackgroundBrushInfo: `scalar deleting ..." is highlighted in blue and has the L flag set.

Function name	Segment	Start	Length	R	F	L	S	B	T	=
CheckWindowsGenuineStatus()	.text	010091F9	0000007C	R	.	.	.	B	.	.
ControlBackgroundBrushInfo: `scalar deleting ...	.text	01032897	00000029	R	.	.	.	B	.	.
CreateDecoderFromResource(IWICImagingF...	.text	0101FB50	00000097	R	.	.	.	B	T	.

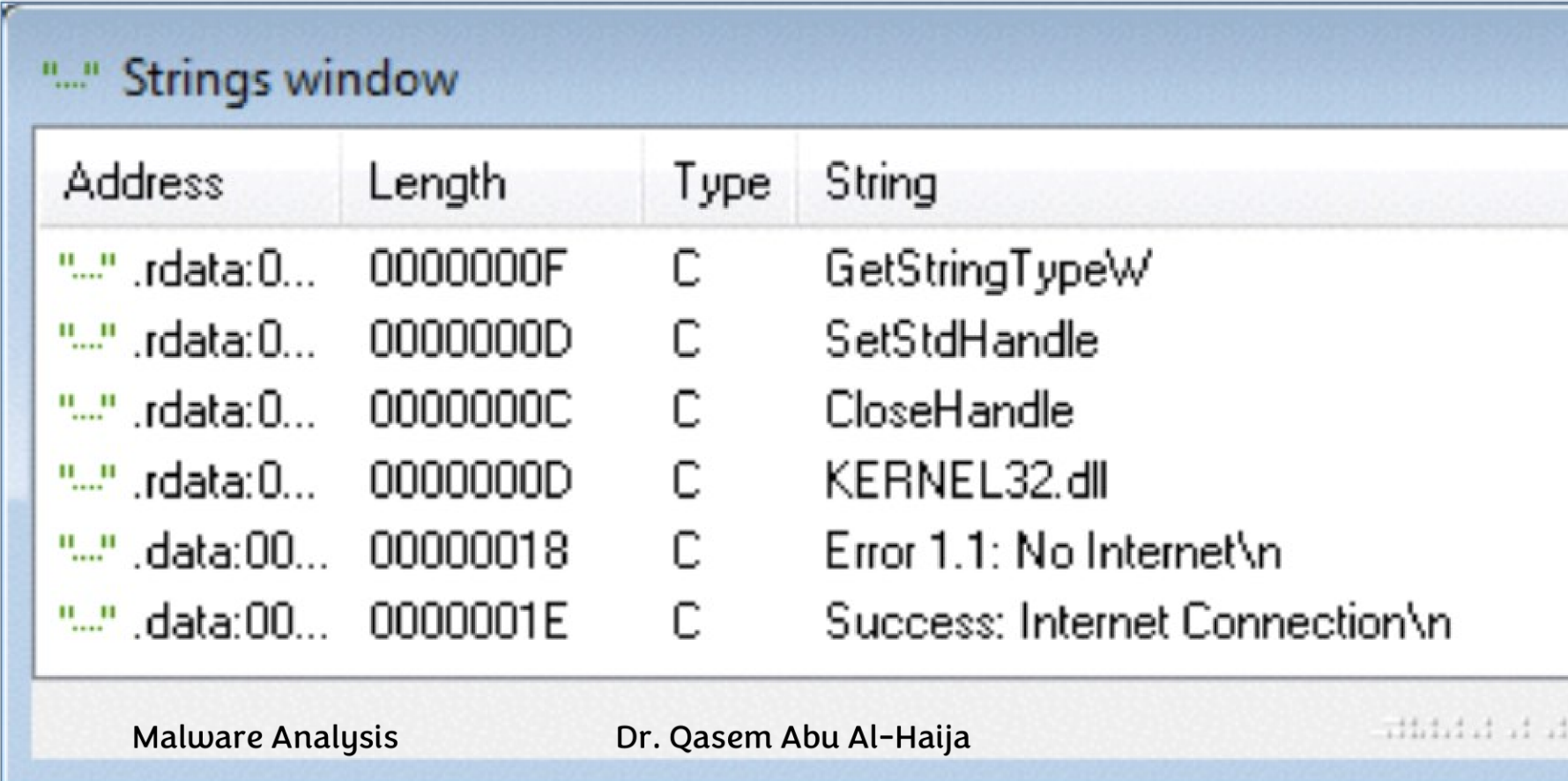
# Names window

- Lists every address with a name, including:
  - Functions, named code, named data, and strings.



# Strings window

- Shows all strings.
  - Default: shows only ASCII strings longer than five characters.
  - You can change this: by right-clicking in Strings window → Setup.



Address	Length	Type	String
"..." .rdata:0...	0000000F	C	GetStringTypeW
"..." .rdata:0...	0000000D	C	SetStdHandle
"..." .rdata:0...	0000000C	C	CloseHandle
"..." .rdata:0...	0000000D	C	KERNEL32.dll
"..." .data:00...	00000018	C	Error 1.1: No Internet\n
"..." .data:00...	0000001E	C	Success: Internet Connection\n

Malware Analysis Dr. Qasem Abu Al-Haija

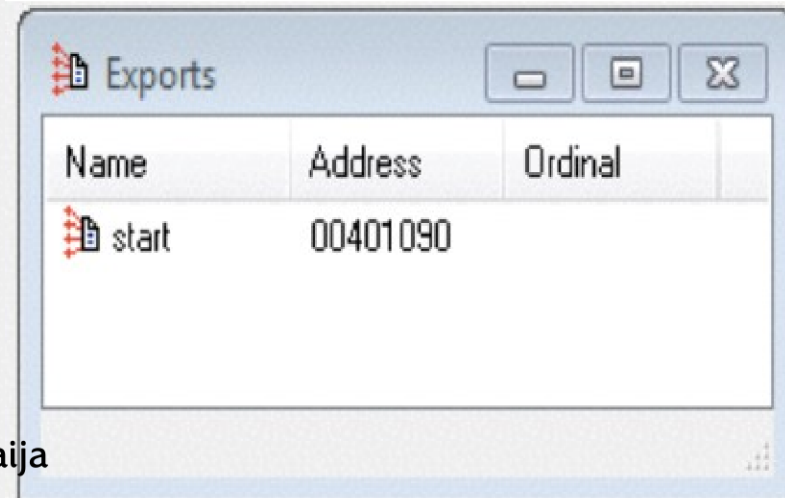
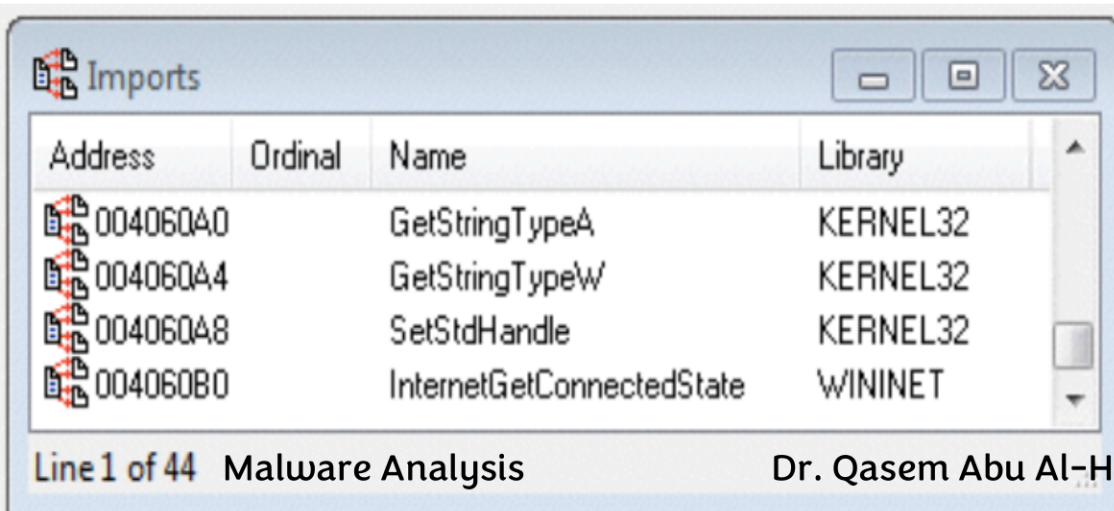
# Imports & Exports

- **Imports window**

- Lists all imports for a file.
- This window is useful when you're analyzing EXE.

- **Exports window**

- Lists all the exported functions for a file.
- This window is useful when you're analyzing DLLs.

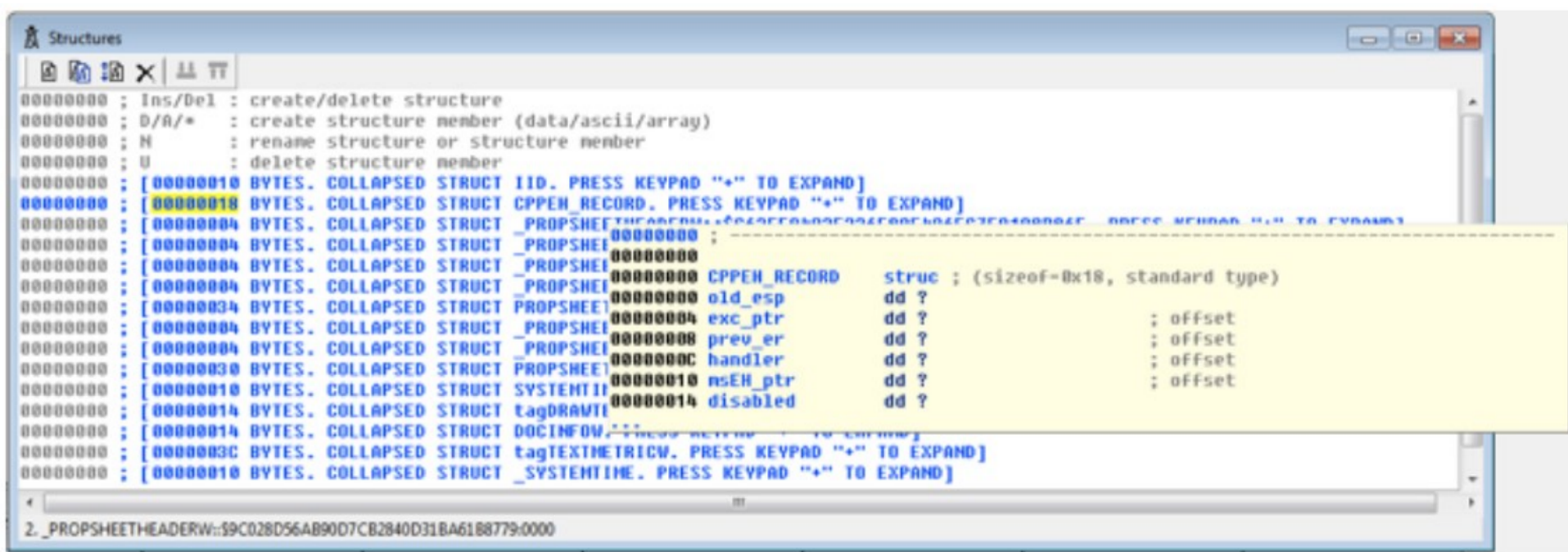




# Structures window

Lists the layout of all active data structures.

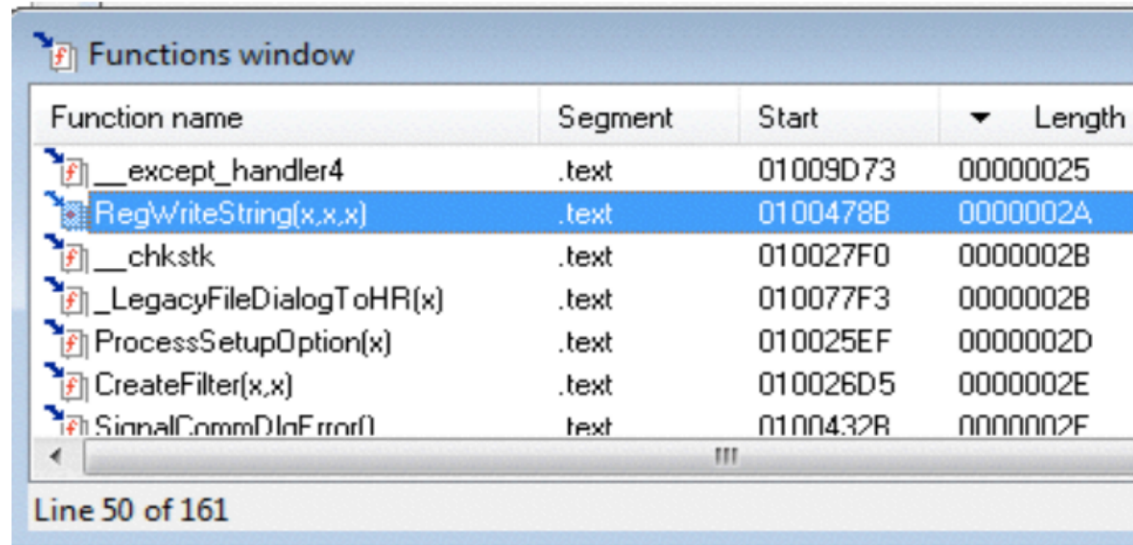
- Hover to see a yellow pop-up window



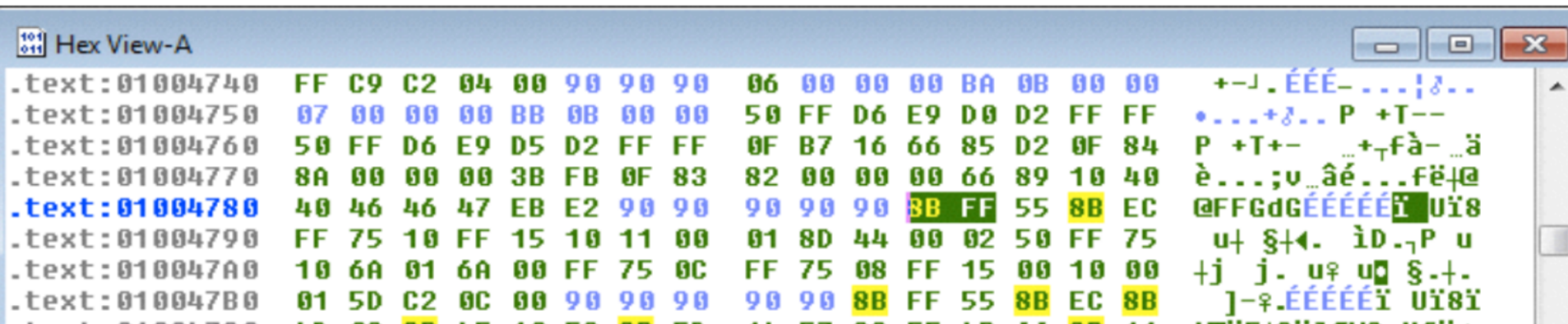
# Cross-References and Links

## Cross-Reference

- Double-click function
- Jump to code in other views



Links are **function names**, **locations**, and **offsets**.  
Clicking a link and pressing [x] (key x) allows you to see everywhere that link is referenced and jump around



# Function Call

- Parameters pushed onto the stack
- CALL to start the function



```
0100478B
0100478B
0100478B      ; Attributes: bp-based frame
0100478B
0100478B      ; int __stdcall RegWriteString(HKEY hKey,LPCWSTR lpValueName,BYTE *lpData)
0100478B      _RegWriteString@12 proc near
0100478B
0100478B      hKey= dword ptr  8
0100478B      lpValueName= dword ptr  0Ch
0100478B      lpData= dword ptr  10h
0100478B
0100478B  8B FF      mov     edi, edi
0100478D  55        push   ebp
0100478E  8B EC      mov     ebp, esp
01004790  FF 75 10   push   [ebp+lpData] ; lpString
01004793  FF 15 10 11 00 01 call   ds:__imp__lstrlenW@4 ; lstrlenW(x)
```

0.00% (-30,-41) (788,342) 00003B8B 0100478B: RegWriteString(x,x,x)

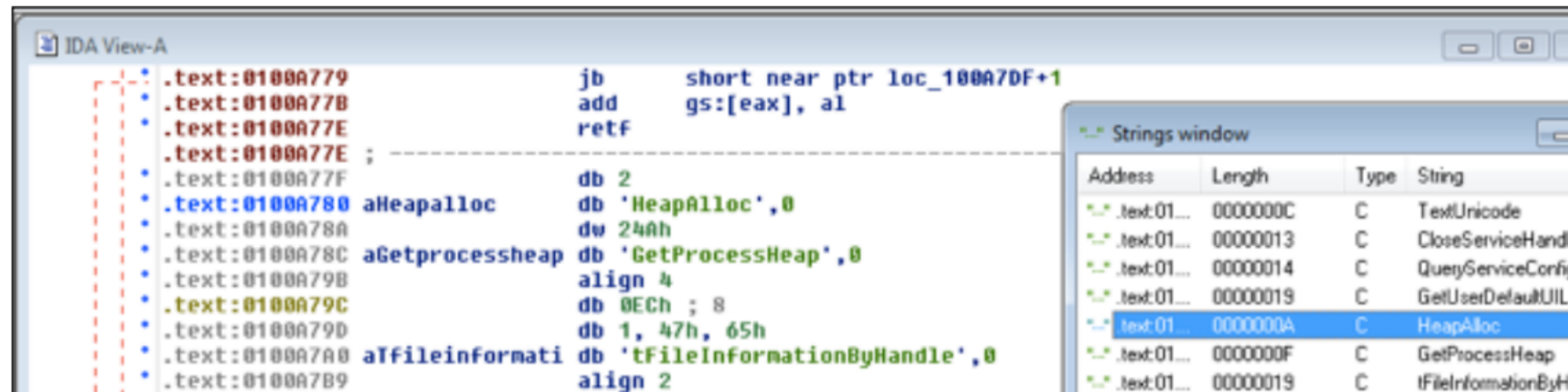
# Returning to the Default View

- **Windows, Reset Desktop**
  - Restores GUI elements to their defaults
- **Windows, Save Desktop**
  - To save a new view

# IDA Pro - Navigation

# Imports or Strings

- Double-click any entry to display it in the disassembly window



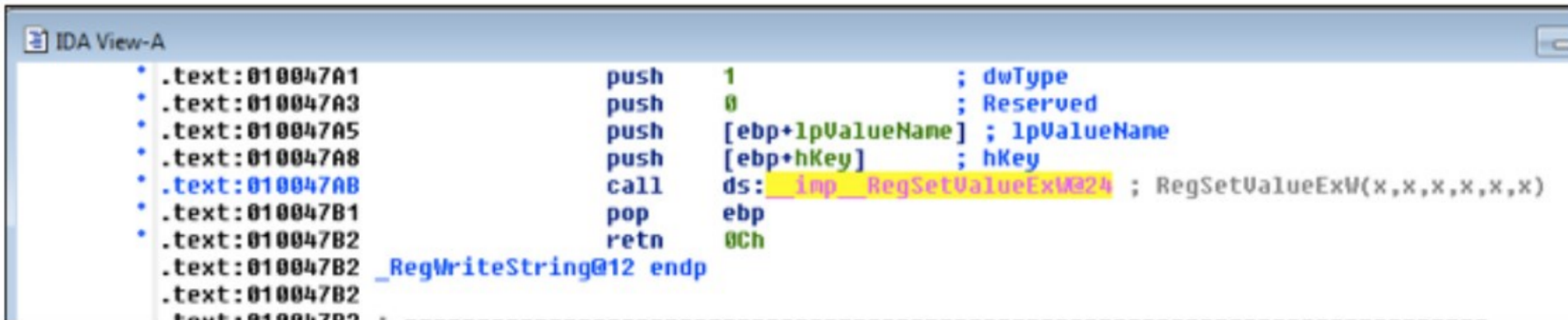
The screenshot shows the IDA View-A interface. The main window displays assembly code for a function. A red dashed box highlights the first four lines of code. The Strings window is open on the right, showing a list of strings with their addresses, lengths, and types. The string 'HeapAlloc' is highlighted in blue.

```
.text:0100A779  jb     short near ptr loc_100A7DF+1
.text:0100A77B  add   gs:[eax], al
.text:0100A77E  retf
.text:0100A77E  ; -----
.text:0100A77F  db  2
.text:0100A780  aHeapalloc  db  'HeapAlloc',0
.text:0100A78A  dw  24Ah
.text:0100A78C  aGetprocessheap  db  'GetProcessHeap',0
.text:0100A79B  align 4
.text:0100A79C  db  0ECh ; 8
.text:0100A79D  db  1, 47h, 65h
.text:0100A7A0  aTfileinformati  db  'tFileInformationByHandle',0
.text:0100A7B9  align 2
```

Address	Length	Type	String
.text:01...	0000000C	C	TextUnicode
.text:01...	00000013	C	CloseServiceHandl
.text:01...	00000014	C	QueryServiceConfig
.text:01...	00000019	C	GetUserDefaultUIL
.text:01...	0000000A	C	HeapAlloc
.text:01...	0000000F	C	GetProcessHeap
.text:01...	00000019	C	tFileInformationByH

# Using Links

- Double-click any address in the disassembly window to display that location

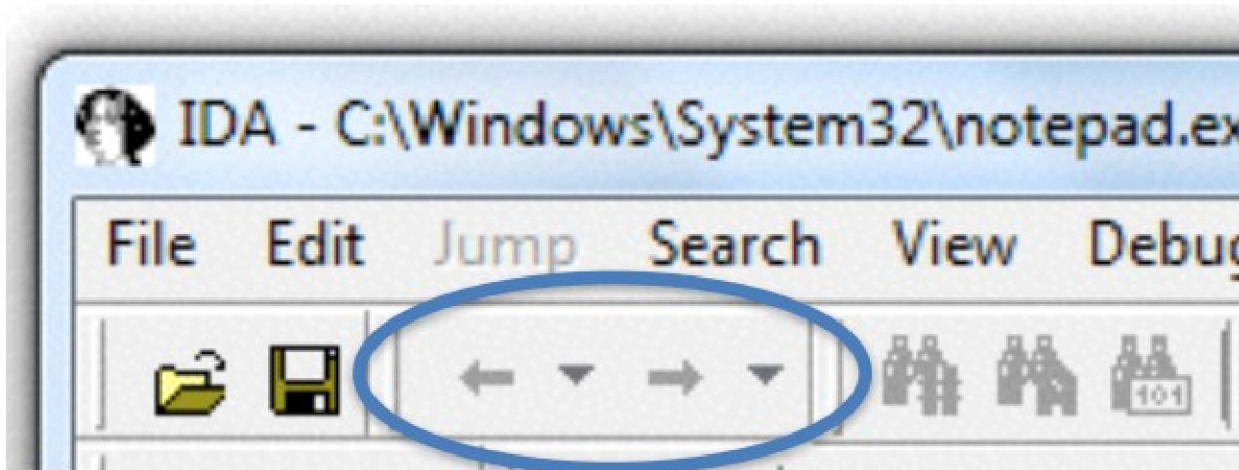


The screenshot shows the IDA View-A window displaying assembly code. The instruction at address 010047AB is highlighted in yellow: `call ds:inp__RegSetValueExW@24 ; RegSetValueExW(x,x,x,x,x,x)`. The code includes several push and pop instructions, and a return instruction.

```
IDA View-A
• .text:010047A1      push    1           ; dwType
• .text:010047A3      push    0           ; Reserved
• .text:010047A5      push    [ebp+lpValueName] ; lpValueName
• .text:010047A8      push    [ebp+hKey]   ; hKey
• .text:010047AB      call   ds:inp__RegSetValueExW@24 ; RegSetValueExW(x,x,x,x,x,x)
• .text:010047B1      pop     ebp
• .text:010047B2      retn    0Ch
.text:010047B2      _RegWriteString@12 endp
.text:010047B2
.text:010047B2
```

# Using Links

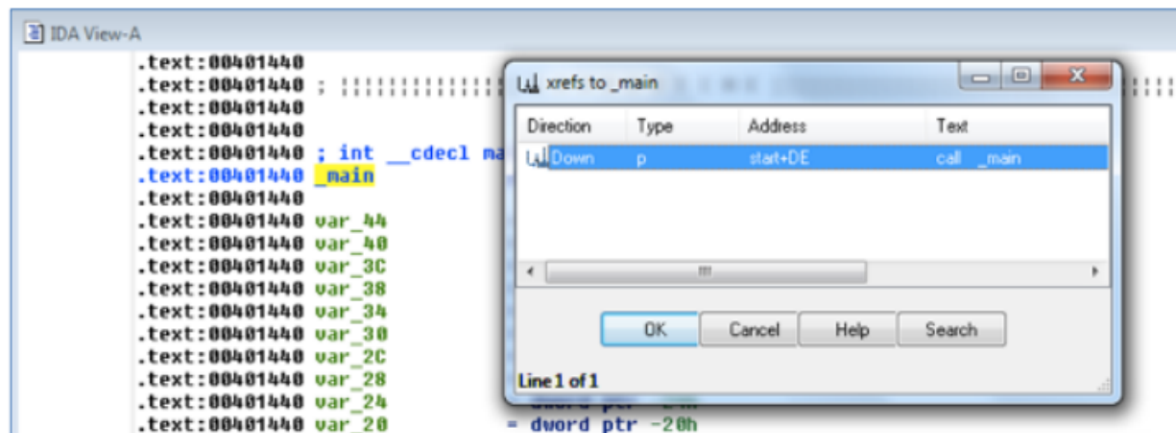
- Forward and Back buttons work like a Web browser





# Cross-References and Links

- Links are function names, locations, and offsets
- Clicking a link and pressing [x] (key x) allows you to see everywhere that link is referenced and jump around



# Common types of links

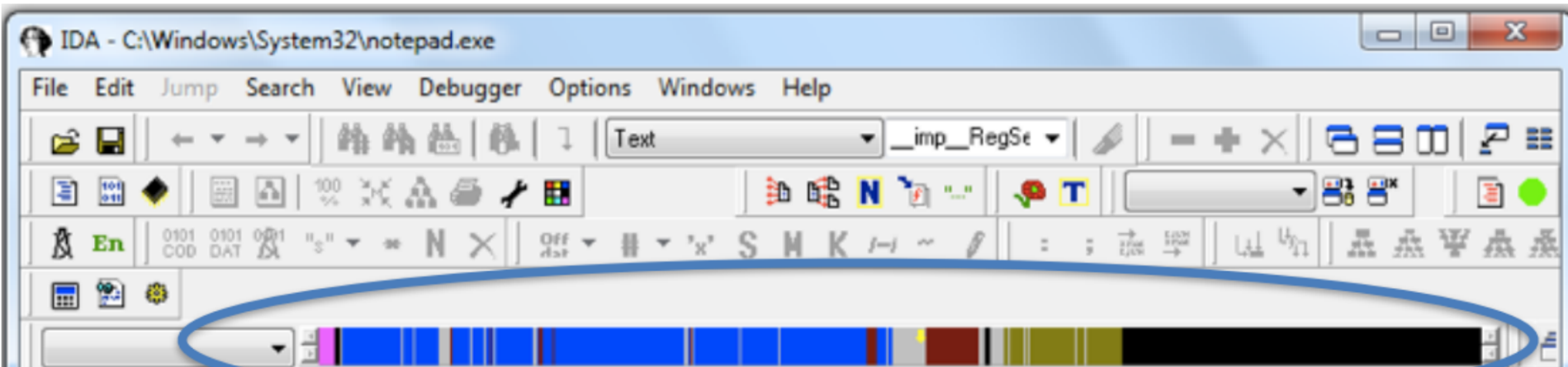
- **Sub** (*subroutine*) links are links to the start of functions such as **printf** and **sub\_4010A0**.
- **Loc** (*location*) links are links to jump destinations such as **loc\_40107E** and **loc\_401097**.
- **Offset** links are links to an offset in memory.

---

```
00401075      jnz      short ①loc_40107E
00401077      mov     [ebp+var_10], 1
0040107E loc_40107E:      ; CODE XREF: ①②sub_401040+35j
0040107E      cmp     [ebp+var_C], 0
00401082      jnz     short ①loc_401097
00401084      mov     eax, [ebp+var_4]
00401087      mov     [esp+18h+var_14], eax
0040108B      mov     [esp+18h+var_18], offset ①aPrintNumberD ; "Print Number= %d\n"
00401092      call   ①printf
00401097      call   ①sub_4010A0
```

# Navigation Band

- Color-coded linear view of the loaded binary's address space
  - **Light blue** is library code (recognized by FLIRT signatures)
  - **Red** is compiler-generated code
  - **Dark blue** is user-written code
  - **Pink** is for imports
  - Gray is for defined data
  - **Brown** is for undefined data
- You should perform malware analysis in the dark-blue region.



File Edit Jump Search View Debugger Options Windows Help

Text

En 0101 COD 0101 DAT 0101 DA "s" \* N X Off dat # % S M K /- ~ :

IDA View-A Hex View-A Exports Imports Names Functions Strings Structures En Enums

← Navigation Band

```

IDA View-A
. .data:00403138 dword_403138 dd 0 ; DATA XREF: start+55f1r
. .data:0040313C dword_40313C dd 0 ; DATA XREF: start+47f1r
. .data:00403140 dword_403140 dd 0 ; DATA XREF: start+33f1w
. .data:00403144 dword_403144 dd 0 ; DATA XREF: start+3Af1w
. .data:00403148 dword_403148 dd 0 ; DATA XREF: start+64f1w
. .data:0040314C
. |
. .data:0040314C _data align 1000h
. ends
. .data:0040314C
. .data:0040314C
. .data:0040314C
. .data:0040314C
. .data:0040314C
. end start
    
```

Names window

Name

- \_main
- \_\_alloca\_probe
- start
- \_\_XcptFilter
- \_\_initterm
- \_\_setdefaultprecision
- nullsub\_1
- \_\_except\_handler3

1 of 60

Strings window

Address	Length	T...	String
.rdata:0...	0000000C	C	CloseHan
.rdata:0...	0000000C	C	OpenProc
.rdata:0...	00000012	C	GetCurren
.rdata:0...	00000013	C	CreatePer
.rdata:0...	0000000F	C	GetProcA
.rdata:0...	0000000D	C	LoadLibra
.rdata:0...	00000008	C	WinExec
.rdata:0...	0000000A	C	WriteFile

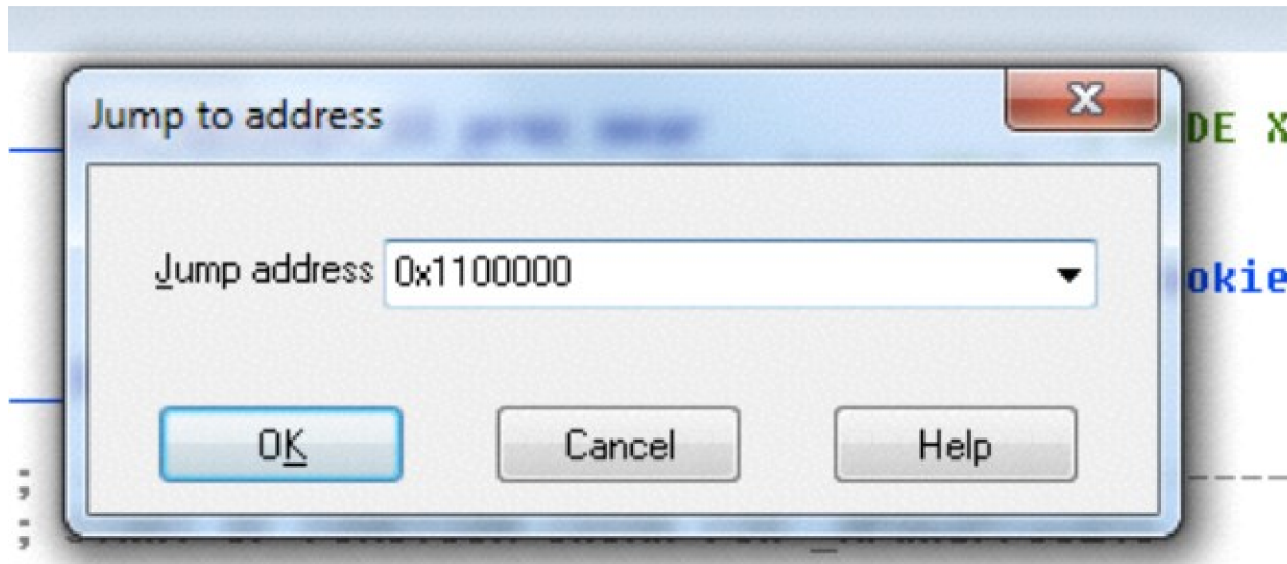
```

262144 32 8192 allocating memory for name pointers...
-----
589824 total memory allocated

Loading IDP module C:\Program Files\IDA Free\procs\pc.w32 for processor metapc...OK
Loading type libraries...
Autoanalysis subsystem has been initialized.
Database for file 'Lab01-04.exe' is loaded.
Compiling file 'C:\Program Files\IDA Free\idc\ida.idc'...
Executing function 'main'...
    
```

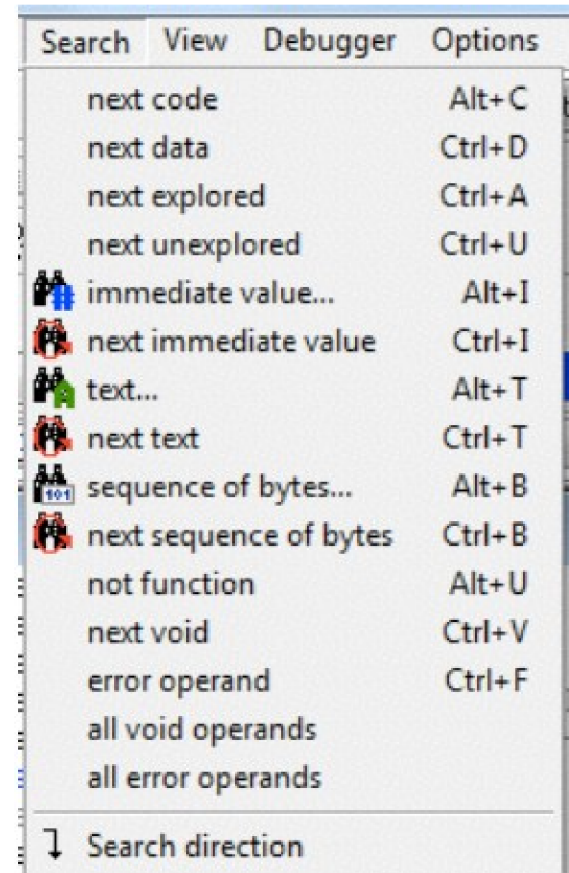
# Jump to Location

- The [g] key lets you jump to a specific address or named location
  - ex. sub\_401730 or printf



# Searching

- You can also use the search option in the toolbar
  - Search → Next Code
  - Search → Text
  - Search → Sequence of Bytes



# Using Cross-References

# Using Cross-References (xref)

- **xref** in IDA Pro can tell you
  - where a function is called or
  - where a string is used.
- If you identify a useful function and want to know the parameters with which it is called,
  - you can use **xref to navigate** quickly to the location where the parameters are placed on the stack.
  - Interesting graphs can also be generated based on **xref**, which are helpful for performing analysis.



# Code Cross-References

- XREF comment shows where this function is called
- But it only shows a couple of XREF by default.
- Click function name and press X

```
.text:00401440
.text:00401440 ; :::::::::::::::::::: S U B R O U T I N E ::::::::::::::::::::
.text:00401440
.text:00401440 ; int __cdecl main(int argc,const char **argv,const char *envp)
.text:00401440 _main          proc near          ; CODE XREF: start+0E1p
.text:00401440
.text:00401440 var_44          = dword ptr -44h
.text:00401440 var_40          = dword ptr -40h
.text:00401440 var_3C          = dword ptr -3Ch
.text:00401440 var_38          = dword ptr -38h
.text:00401440 var_34          = dword ptr -34h
.text:00401440 var_30          = dword ptr -30h
.text:00401440 var_2C          = dword ptr -2Ch
.text:00401440 var_28          = dword ptr -28h
.text:00401440 var_24          = dword ptr -24h
.text:00401440 var_20          = dword ptr -20h
.text:00401440 var_1C          = dword ptr -1Ch
.text:00401440 var_18          = dword ptr -18h
                push    offset unk_403000
                call    _inittern
                call    ds:__p__initenv
                mov     ecx, [ebp+envp]
                mov     [eax], ecx
                push   [ebp+envp]      ; envp
                push   [ebp+argv]     ; argv
                push   [ebp+argc]    ; argc
                call   main
                add    esp, 30h
```

# Code Cross-References

- A code XREF at ❶ tells us that this function (sub\_401000) is called from inside the main function at offset 0x3 into the main function.
- The code XREF for the jump at ❷ tells us which jump takes us to this location, which in this example corresponds to the location marked at ❸

---


```
00401000      sub_401000      proc near      ; ❶CODE XREF: _main+3p
00401000      push     ebp
00401001      mov      ebp, esp
00401003  loc_401003:      ; ❷CODE XREF: sub_401000+19j
00401003      mov      eax, 1
00401008      test     eax, eax
0040100A      jz       short loc_40101B
0040100C      push    offset aLoop      ; "Loop\n"
00401011      call    printf
00401016      add     esp, 4
00401019      jmp     short loc_401003 ❸
```

---

# Data Cross-References

## Data Cross-References (XREF)

- Start with strings
- Double-click an interesting string
- Hover over DATA XREF to see where that string is used
- X shows all references



```
.data:0040304C ; char NewFileName[]
.data:0040304C NewFileName db 'C:\\windows\\system32\\kerne132.dll',0
.data:0040304C ; DATA XREF: _main+30A7o
.data:00403060 align 10h
.data:00403070 dword_403070 dd 6E726540h ; 0
.data:00403074 dword_403074 dd 32306C65h ; 0
.data:00403078 byte_403078 db 2Eh ; 0
.data:00403079 align 4
.data:0040307C ; char ExistingFileName[]
.data:0040307C ExistingFileName db 'Lab01-01.dll',0 ; 0
.data:0040307C ; 0
.data:00403089 align 4
.data:0040308C ; char FileName[]
.data:0040308C FileName db 'C:\\Windows\\System32\\Ker
.data:0040308C ; DATA XREF: _main+677o

mov ecx, [esp+54h+hObject]
mov esi, ds:CloseHandle
push ecx ; hObject
call esi ; CloseHandle
mov edx, [esp+54h+var_4]
push edx ; hObject
call esi ; CloseHandle
push 0 ; bFailIfExists
push offset NewFileName ; "C:\\windows\\system32\\kerne132.dll"
push offset ExistingFileHane ; "Lab01-01.dll"
```

# Data Cross-References

- Data XREF is used to track **how data is accessed within a binary**.
  - Data references can be associated with any byte of data that is referenced in code via a memory reference.
- For example, you can see the data XREF to DWORD 0x7F000001 at **①**.
  - The corresponding XREF tells us that this data is used in the function located at 0x401020.
  - The following line shows a data cross-reference for the string <Hostname> <Port>.

---

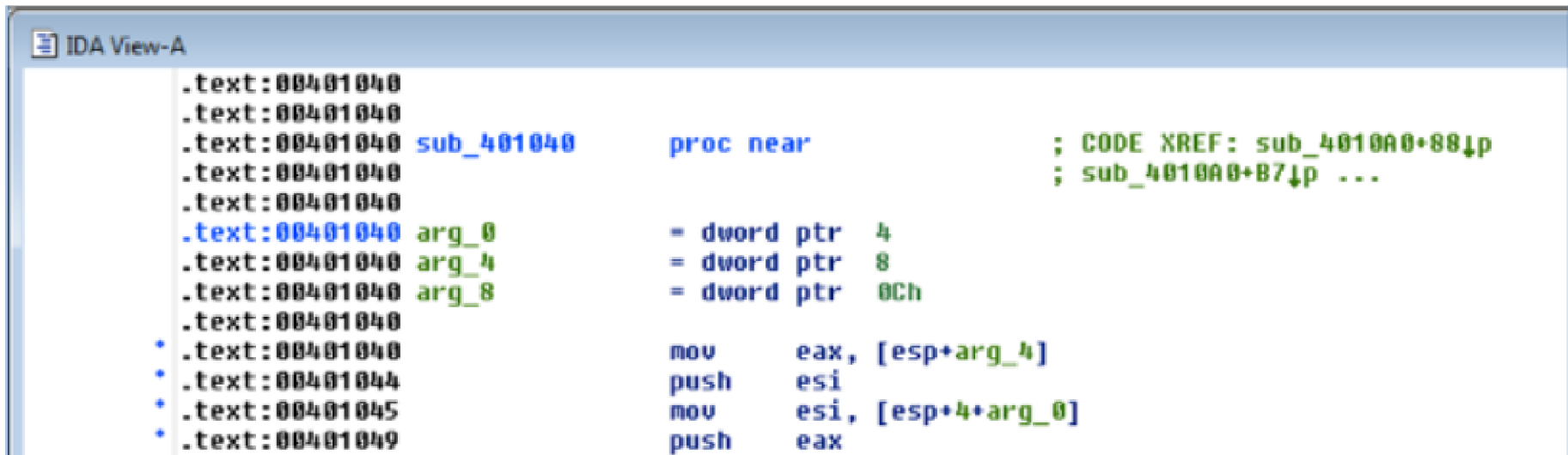
```
0040C000 dword_40C000    dd 7F000001h          ; ①DATA XREF: sub_401020+14r
0040C004 aHostnamePort      db '<Hostname> <Port>',0Ah,0 ; DATA XREF: sub_401000+30
```

---

# Analyzing Functions

# Function and Argument Recognition

- One powerful aspect of IDA Pro is its ability to:
  - recognize functions,
  - label them (name them), and
  - break down local variables and parameters (name them).
  - It's not always correct



```
IDA View-A
.text:00401040
.text:00401040
.text:00401040 sub_401040 proc near ; CODE XREF: sub_4010A0+88↓p
.text:00401040 ; sub_4010A0+B7↓p ...
.text:00401040 arg_0 = dword ptr 4
.text:00401040 arg_4 = dword ptr 8
.text:00401040 arg_8 = dword ptr 0Ch
* .text:00401040 mov eax, [esp+arg_4]
* .text:00401044 push esi
* .text:00401045 mov esi, [esp+4+arg_0]
* .text:00401049 push eax
```

# Function and Argument Recognition

- IDA Pro says this is an EBP-based stack frame used in the function.
- This means the local variables and parameters will be referenced via the EBP register throughout the function.

```
00401020 ; ===== S U B R O U T I N E =====
00401020
00401020 ; Attributes: ebp-based frame ❶
00401020
00401020 function      proc near          ; CODE XREF: _main+1Cp
00401020
00401020 var_C          = dword ptr -0Ch ❷
00401020 var_8          = dword ptr -8
00401020 var_4          = dword ptr -4
00401020 arg_0          = dword ptr 8
00401020 arg_4          = dword ptr 0Ch
00401020
00401020 push          ebp
00401021 mov           ebp, esp
00401023 sub           esp, 0Ch
00401026 mov           [ebp+var_8], 5
0040102D mov           [ebp+var_C], 3 ❸
00401034 mov           eax, [ebp+var_8]
00401037 add           eax, 22h
0040103A mov           [ebp+arg_0], eax
0040103D cmp           [ebp+arg_0], 64h
00401041 jnz          short loc_40104B
00401043 mov           ecx, [ebp+arg_4]
00401046 mov           [ebp+var_4], ecx
00401049 jmp          short loc_401050
0040104B loc_40104B:          ; CODE XREF: function+21j
0040104B call         sub_401000
00401050 loc_401050:          ; CODE XREF: function+29j
00401050 mov           eax, [ebp+arg_4]
00401053 mov           esp, ebp
00401055 pop           ebp
00401056 retn
00401056 function      endp
```

Listing 5-4: Function and stack example

# Local Variables and Argument Recognition

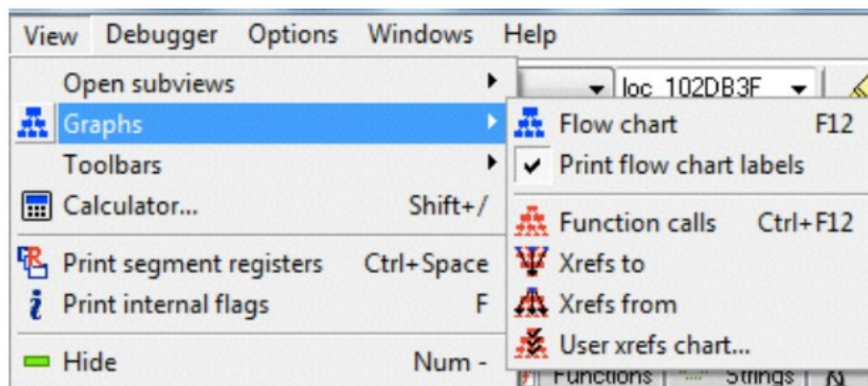
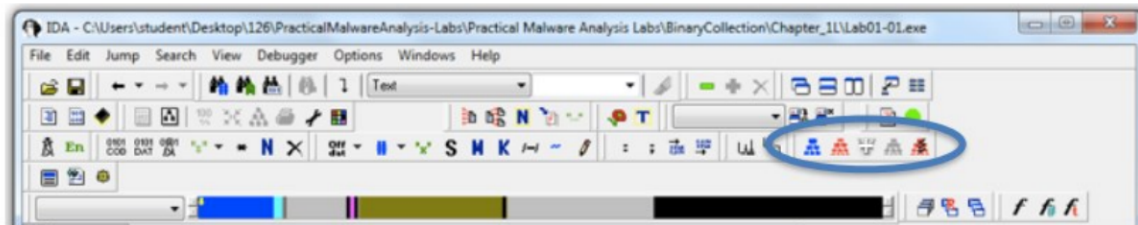
- IDA Pro has successfully discovered all local variables and parameters in this function.
- It has labeled:
  - Local variables with the prefix `var_` and suffix corresponding to their offset relative to EBP
  - parameters with the prefix `arg_`, and suffix corresponding to their offset relative to EBP



# Using Graph Options

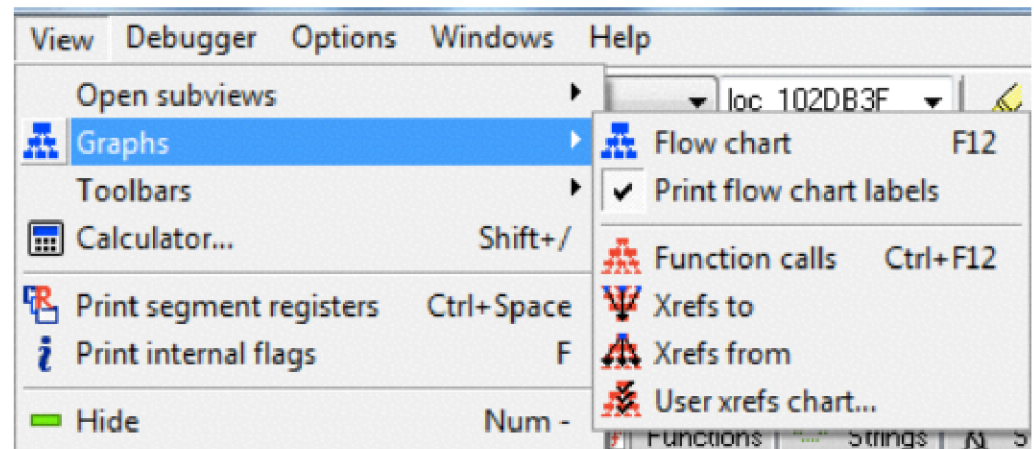
# Graphing Options

- IDA Pro can generate other graphs
  - Flow chart of the current function
  - Function calls for the entire program
  - Xrefs to/from a currently selected xref
  - User-specified xref graphs



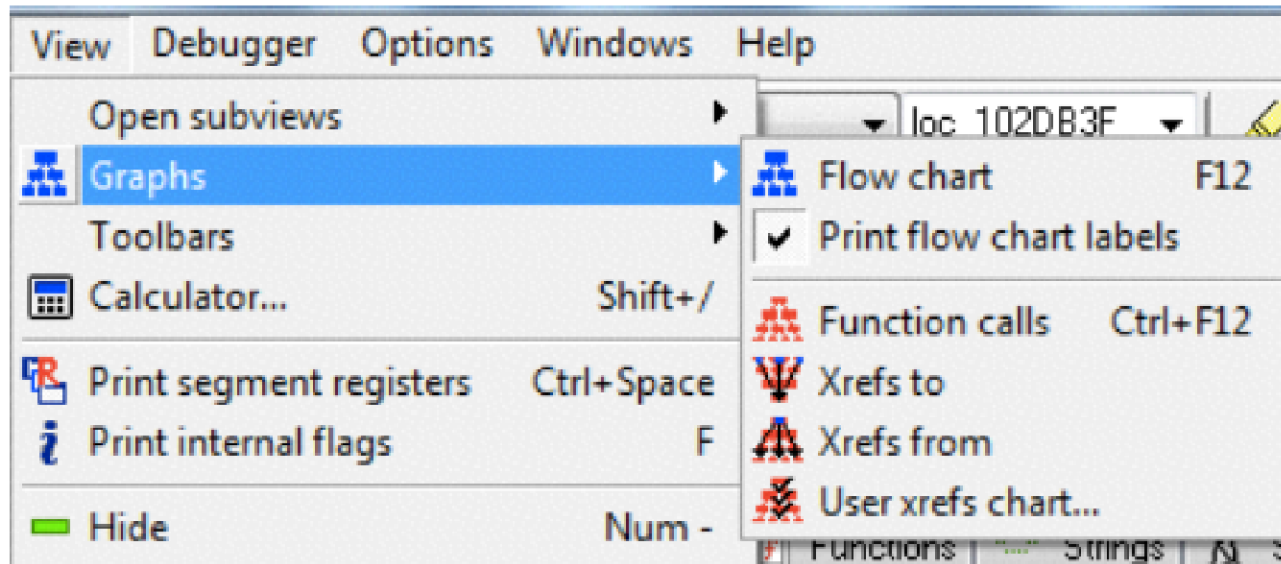
# Graphing Options

- These are "Legacy Graphs" and cannot be manipulated with IDA
- The first two seem obsolete
  - Flow chart
    - Create a flow chart of the current function
  - Function calls
    - Graph function calls for the entire program



# Graphing Options

- Xrefs to
  - Graphs XREFs to get to selected XREF
  - Can show all the paths that get to a function



# Windows Genuine Status in Calc.exe

The image displays two windows from a malware analysis tool. The left window is IDA Pro, showing the assembly code for the `_WinMainCRTStartup` function. The right window is WinGraph32, showing a call graph for the same function.

**IDA View-A Assembly Code:**

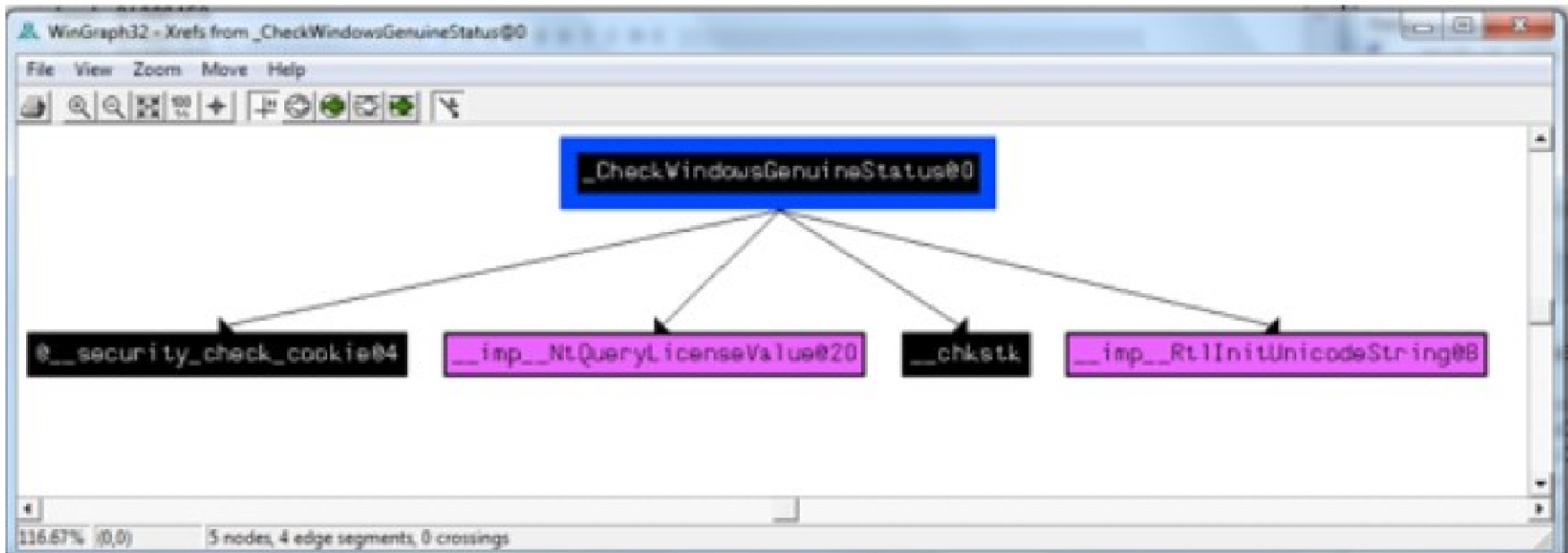
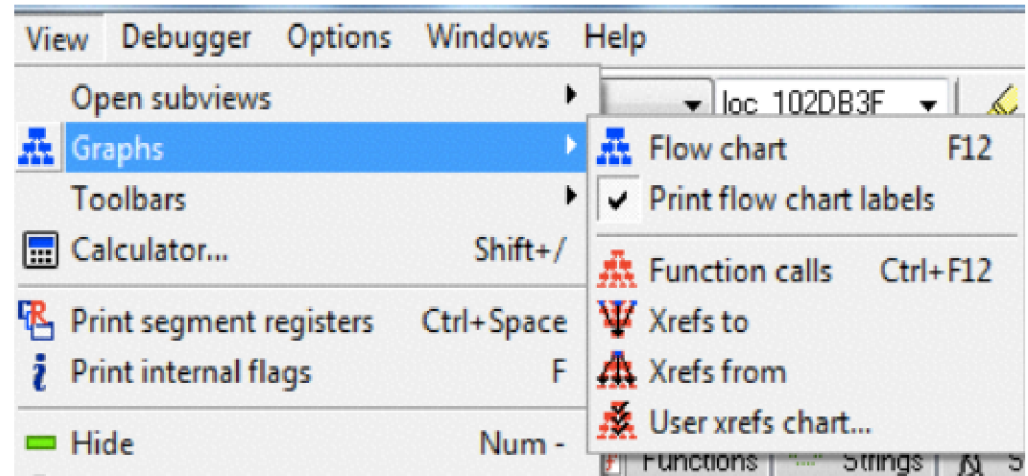
```
.text:010091F9  
.text:010091F9 : ..... SUBROUTINE .....  
.text:010091F9 : Attributes: bp-based frame  
.text:010091F9 : stocall CheckWindowsGenuineStatus()  
.text:010091F9 : CheckWindowsGenuineStatus@0 proc near ; CODE  
.text:010091F9  
.text:010091F9 var_1204 = dword ptr -1204h  
.text:010091F9 var_1204 = dword ptr -1204h  
.text:010091F9 var_1208 = dword ptr -1208h  
.text:010091F9 var_1294 = dword ptr -1294h  
.text:010091F9 var_4 = dword ptr -4  
.text:010091F9  
* .text:010091F9 mov edi, edi  
* .text:010091F8 push ebp  
* .text:010091FC mov ebp, esp  
* .text:010091FE mov eax, 1320h  
* .text:01009200 call ___chkstk  
* .text:01009208 mov eax, ___security_cookie  
* .text:01009200 xor eax, ebp
```

**WinGraph32 Call Graph:**

```
graph TD; A[_WinMainCRTStartup] --> B[_WinMain@16]; B --> C[_CheckWindowsGenuineStatus@0];
```

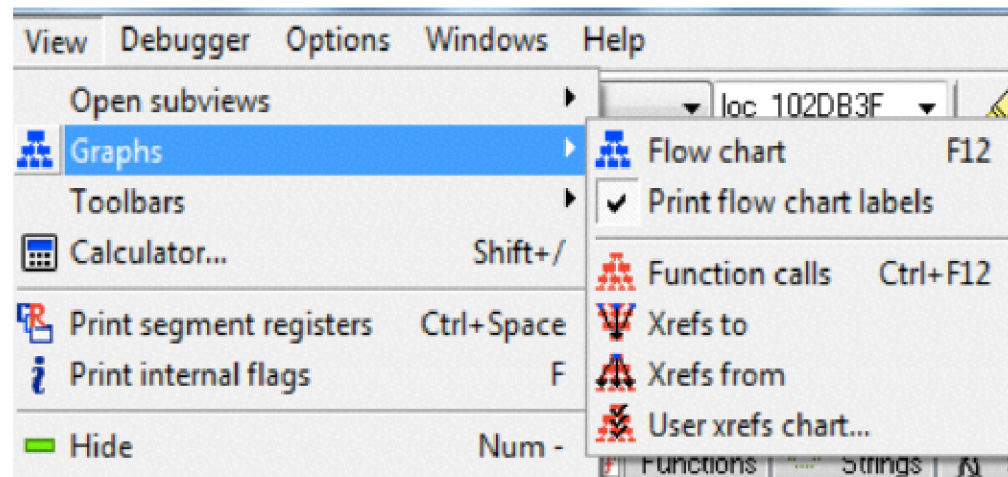
# Graphing Options

- **Xrefs from**
  - Graphs XREFs from selected XREF
  - Can show all the paths that exit from a function








# Graphing Options

- User xrefs chart...
  - Customize graph's recursive depth, symbols used, to or from symbol, etc.
  - The only way to modify legacy graphs



# Using Graphing Options

Button	Function	Description
	Creates a flow chart of the current function	Users will prefer to use the interactive graph mode of the disassembly window but may use this button at times to see an alternate graph view. (We'll use this option to graph code in Chapter 6.)
	Graphs function calls for the entire program	Use this to gain a quick understanding of the hierarchy of function calls made within a program, as shown in Figure 5-8. To dig deeper, use WinGraph32's zoom feature. You will find that graphs of large statically linked executables can become so cluttered that the graph is unusable.
	Graphs the cross-references to get to a currently selected cross-reference	This is useful for seeing how to reach a certain identifier. It's also useful for functions, because it can help you see the different paths that a program can take to reach a particular function.

Button	Function	Description
	Graphs the cross-references from the currently selected symbol	This is a useful way to see a series of function calls. For example, Figure 5-9 displays this type of graph for a single function. Notice how sub_4011f0 calls sub_401110, which then calls gethostname. This view can quickly tell you what a function does and what the functions do underneath it. This is the easiest way to get a quick overview of the function.
	Graphs a user-specified cross-reference graph	Use this option to build a custom graph. You can specify the graph's recursive depth, the symbols used, the to or from symbol, and the types of nodes to exclude from the graph. This is the only way to modify graphs generated by IDA Pro for display in WinGraph32.



# Enhancing Disassembly

# Warning

- There's no Undo, so if you make changes and mess them up, you may be sorry
  - But you manually change it back

# Renaming Locations

- Renaming functions & variables by pressing key **[n]**
  - sub\_401000 or arg\_4 don't tell you much
  - Rename them to something more useful!
  
- Example: You can change a name like sub\_401000 to ReverseBackdoorThread
  - Change it in one place; IDA will change it everywhere else

# Renaming Locations

*Table 6-2. Function Operand Manipulation*

## Without renamed arguments

```
004013C8 mov    eax, [ebp+arg_4]
004013CB push  eax
004013CC call  _atoi
004013D1 add   esp, 4
004013D4 mov   [ebp+var_598], ax
004013DB movzx ecx, [ebp+var_598]
004013E2 test  ecx, ecx
004013E4 jnz   short loc_4013F8
004013E6 push  offset aError
004013EB call  printf
004013F0 add   esp, 4
004013F3 jmp   loc_4016FB
004013F8 ; -----
004013F8
004013F8 loc_4013F8:
004013F8 movzx edx, [ebp+var_598]
004013FF push  edx
00401400 call  ds:htons
```

## With renamed arguments

```
004013C8 mov    eax, [ebp+port_str]
004013CB push  eax
004013CC call  _atoi
004013D1 add   esp, 4
004013D4 mov   [ebp+port], ax
004013DB movzx ecx, [ebp+port]
004013E2 test  ecx, ecx
004013E4 jnz   short loc_4013F8
004013E6 push  offset aError
004013EB call  printf
004013F0 add   esp, 4
004013F3 jmp   loc_4016FB
004013F8 ; -----
004013F8
004013F8 loc_4013F8:
004013F8 movzx edx, [ebp+port]
004013FF push  edx
00401400 call  ds:htons
```

# Comments

- You can add comments to lines too!
- Press the colon (:) to add a single comment
- Press semicolon (;) to echo this comment to all Xrefs

# Formatting Operands

- Hexadecimal by default
- Right-click to use other formats
  - You can change the format of the data
  - ex. 0x61 → 'a' → 97
  - Hex: [h/q]
  - decimal: [d] ; key [h] toggles hex and decimal
  - char: [r]
  - binary: [b]

```
mov     edi, edi
push   ebp
mov     ebp, esp
mov     eax, 1320h
call   __chkstk
mov     eax, ___se
xor     eax, ebp
mov     [ebp+var_4], eax
push   offset aSe
```

Use standard symbolic constant		
#10	4896	H
#8	11440o	
#2	1001100100000b	B

# Using Named Constants

- Makes Windows API arguments clearer

## Before symbolic constants

```
mov     esi, [esp+1Ch+argv]
mov     edx, [esi+4]
mov     edi, ds:CreateFileA
push    0      ; hTemplateFile
push    80h    ;
dwFlagsAndAttributes
push    3      ;
dwCreationDisposition
push    0      ;
lpSecurityAttributes
push    1      ; dwShareMode
```

## After symbolic constants

```
mov     esi, [esp+1Ch+argv]
mov     edx, [esi+4]
mov     edi, ds:CreateFileA
push    NULL   ; hTemplateFile
push    FILE_ATTRIBUTE_NORMAL ;
dwFlagsAndAttributes
push    OPEN_EXISTING ;
dwCreationDisposition
push    NULL   ;
lpSecurityAttributes
push    FILE_SHARE_READ ; dwShareMode
```

# Using Named Constants

- Makes Windows API arguments clearer

## Before symbolic constants

```
mov     esi, [esp+1Ch+argv]
mov     edx, [esi+4]
mov     edi, ds:CreateFileA
push    0      ; hTemplateFile
push    80h    ;
dwFlagsAndAttributes
push    3      ;
dwCreationDisposition
push    0      ;
lpSecurityAttributes
push    1      ; dwShareMode
```


## After symbolic constants

```
mov     esi, [esp+1Ch+argv]
mov     edx, [esi+4]
mov     edi, ds:CreateFileA
push    NULL  ; hTemplateFile
push    FILE_ATTRIBUTE_NORMAL ;
dwFlagsAndAttributes
push    OPEN_EXISTING ;
dwCreationDisposition
push    NULL  ;
lpSecurityAttributes
push    FILE_SHARE_READ ; dwShareMode
```



# Extending IDA with Plug-ins

- IDA Pro has plenty of plugins
  - Interface by using the IDA API
    - IDC
    - IDAPython (not available in IDA Free )
  - Some Useful Plugins:
    - Fentanyl - Patch Assembly in IDA
    - Hex-Rays Decompiler (Costs fat stacks of cash)
    - KANAL - Krypto Analyzer



The screenshot shows a web browser window with the address bar containing the URL [www.openrce.org/downloads/browse/IDA\\_Scripts](http://www.openrce.org/downloads/browse/IDA_Scripts). The main content area displays a table of IDA scripts with the following entries:

Script Name	Author	Description
<b>Decrypt Data</b>	Unknown	IDA script to decipher data from HCU Millenium strainer stage 1 (AESCUL.EXE)
<b>Delphi RTTI script</b>	RedPlait	This script deals with Delphi RTTI structures
<b>Export To Lib</b>	Unknown	This script exports all functions to a lib file
<b>Find Format String Vulnerabilities</b>	Unknown	A small IDC script hacked from sprintf.idc to detect format bugs currently ...

# X-86 Assemble/Disassemble

- Online x86 / x64 Assembler and Disassembler

- <https://defuse.ca/online-x86-assembler.htm#disassembly>

- Online x86 / x64 Assembler and Disassembler

- <https://defuse.ca/online-x86-assembler.htm#disassembly>

Enter your assembly code (intel syntax):

```
mov eax,1  
ret
```

Architecture:  x86  x64

Assembly:

Raw Hex (zero bytes in bold):

```
B801000000C3
```

String Constant:

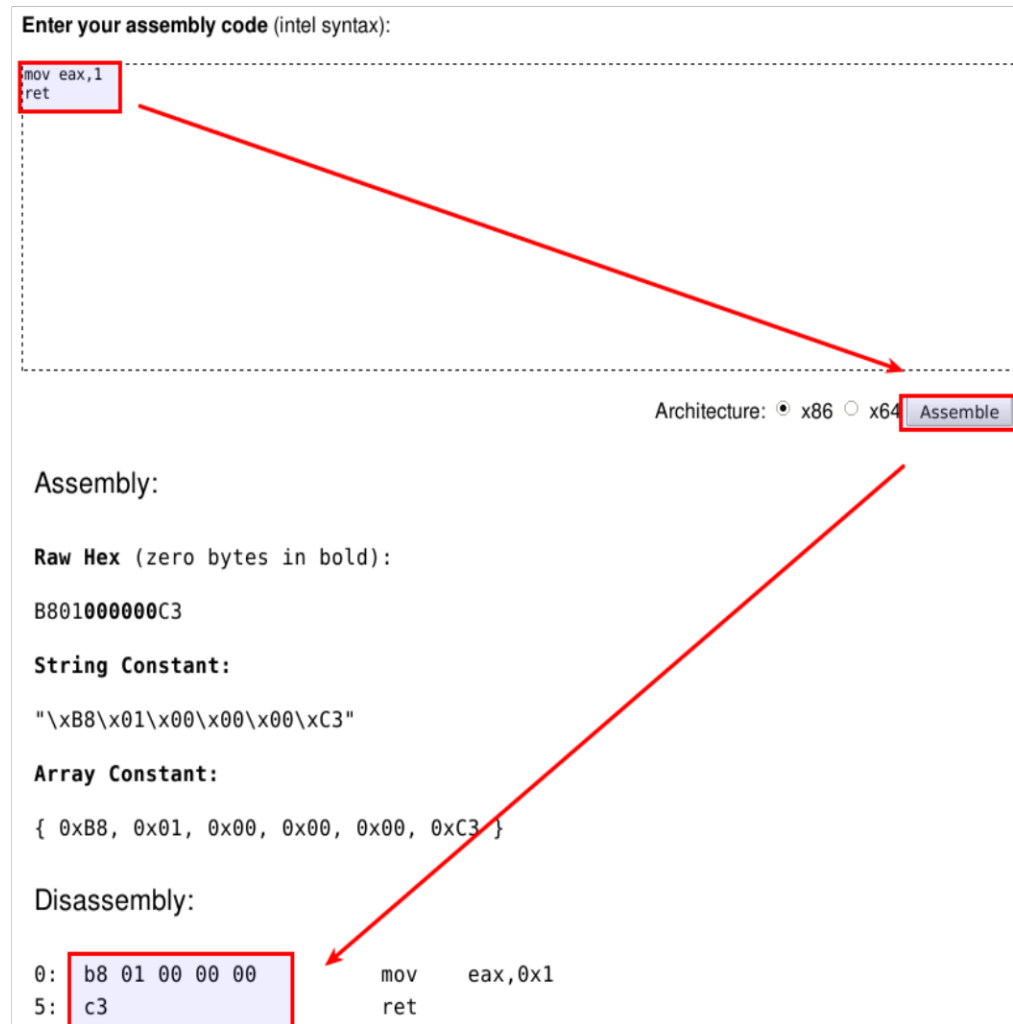
```
"\xB8\x01\x00\x00\x00\xC3"
```

Array Constant:

```
{ 0xB8, 0x01, 0x00, 0x00, 0x00, 0xC3 }
```

Disassembly:

```
0: b8 01 00 00 00    mov    eax,0x1  
5: c3                ret
```



# Main Sources for these slides

- *Michael Sikorski and Andrew Honig, "Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software"; ISBN-10: 1593272901.*
- *Xinwen Fu, "Introduction to Malware Analysis," University of Central Florida*
- *Sam Bowne, "Practical Malware Analysis," City College San Francisco*
- *Abhijit Mohanta and Anoop Saldanha, "Malware Analysis and Detection Engineering: A Comprehensive Approach to Detect and Analyze Modern Malware," ISBN: 1484261925.*

Thank you