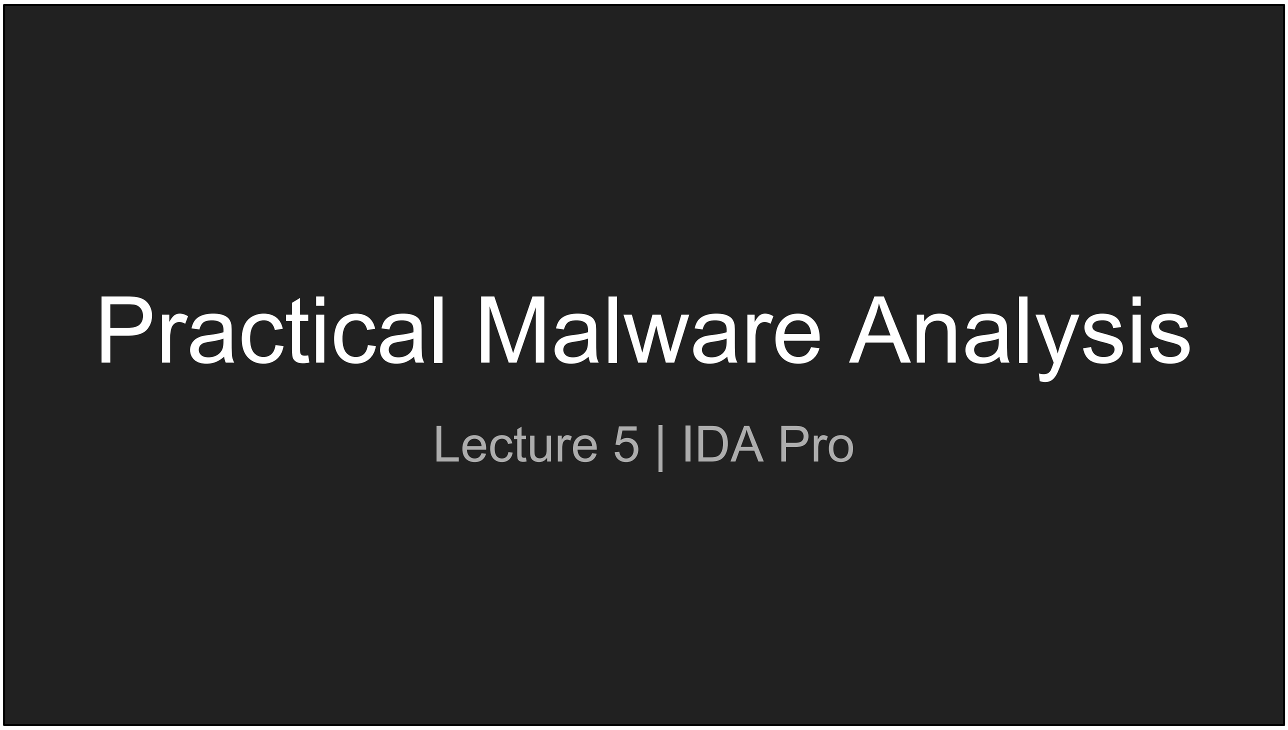
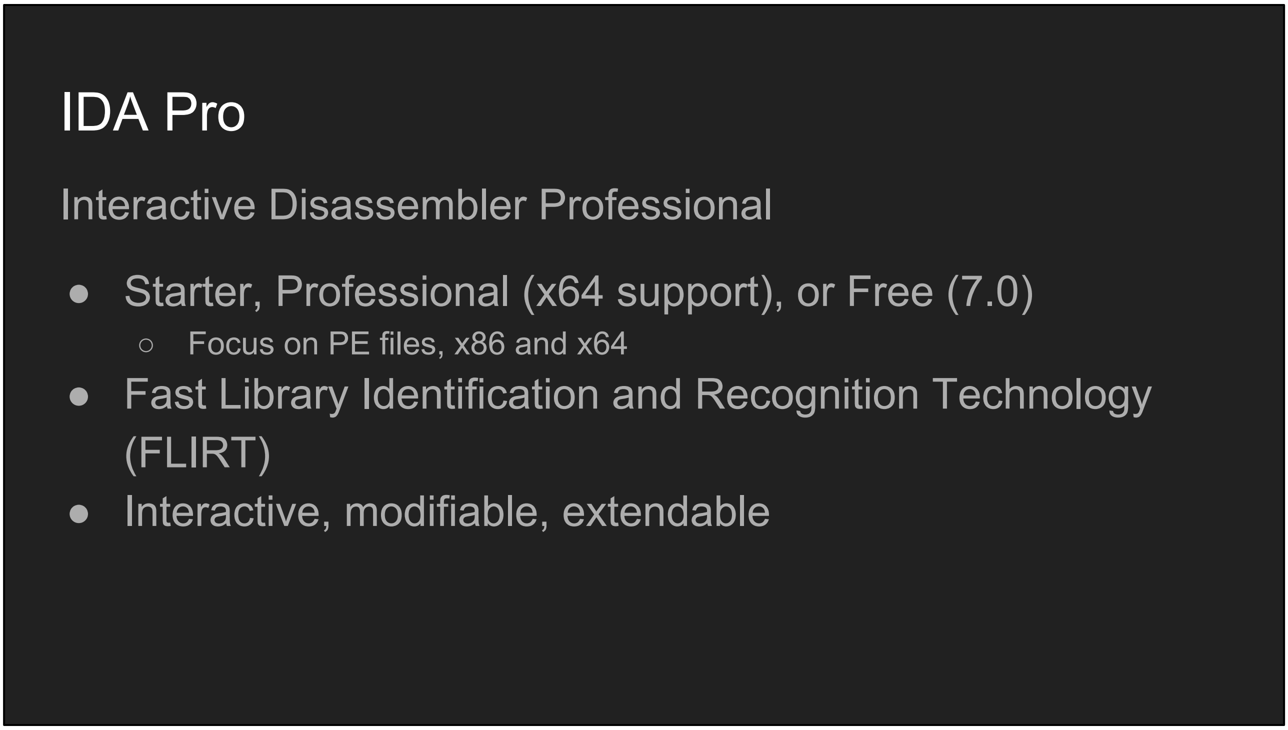


# Practical Malware Analysis

Lecture 5 | IDA Pro



# IDA Pro

Interactive Disassembler Professional

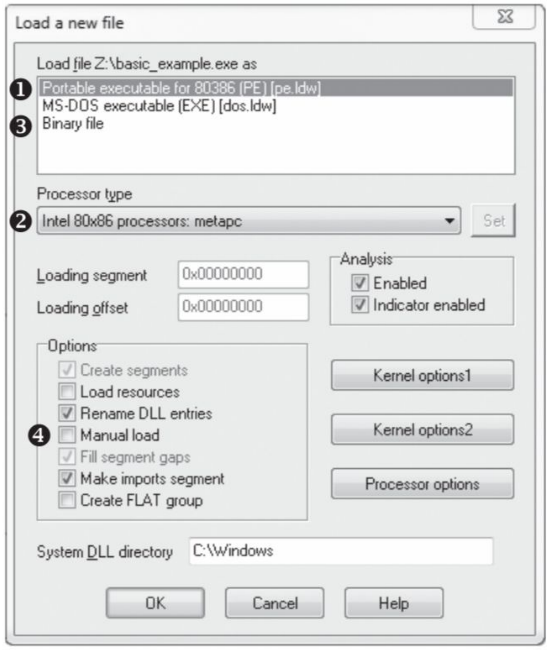
- Starter, Professional (x64 support), or Free (7.0)
  - Focus on PE files, x86 and x64
- Fast Library Identification and Recognition Technology (FLIRT)
- Interactive, modifiable, extendable

The Interactive Disassembler Professional is a popular disassembler distributed by Hex-Rays. We will focus on the commercial version of the software, which supports many processor architectures and file formats, notably x86, x64 and the PE file format. IDA is a powerful tool that can perform tasks like function discovery, stack analysis, and local variable identification automatically to help speed up the analysis process. IDA come with a number of code signatures dubbed FLIRT, allowing it to match and label disassembled functions, especially those belonging to a library added by a compiler.

An analyst may modify, comment, and recolor disassembled code and save any progress to an IDA Pro database file (.idb). IDA Pro also supports plugins for extended functionality.

# Loading an Executable

- (2) for mobile malware
- Program mapped into memory as if by OS loader
- (3) for data appended to PE
  - Shellcode
- Rebasing (DLLs)
  - DLL loads into different process than what is seen in IDA
  - (4) to specify virtual base address
- (4) also for PE header, .rsrc



Load a new file

Load file Z:\basic_example.exe as

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

Processor type: Intel 80x86 processors: metapc

Loading segment: 0x00000000

Loading offset: 0x00000000

Analysis
☑ Enabled
☑ Indicator enabled

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

System DLL directory: C:\Windows

*Figure 5-1: Loading a file in IDA Pro*

When you load a file with IDA Pro, it will automatically try to recognize the file format and processor architecture. If you are analysing mobile malware, you may need to manually select the processor type, as mobile malware exists for many platforms. IDA Pro will load a program into memory as if it were being loaded by the Operating System loader. If previous basic analysis has indicated that there may be extra data or code appended to the PE file, or any shellcode in the binary, telling IDA to load the file as a raw Binary file will force it to load any extra data that it would not otherwise.

Portable executable files are compiled to load at a preferred base address in memory, unless the address is already taken, in which case the loader will perform a rebasing operation. This is common with dynamic-link libraries, and manifests as a DLL loaded into a process different than what it observed in IDA Pro. Using the Manual load option allows us to specify a new virtual base address. IDA also does not load the PE header or resource sections, use the Manual load option to include these in analysis.

# The Interface

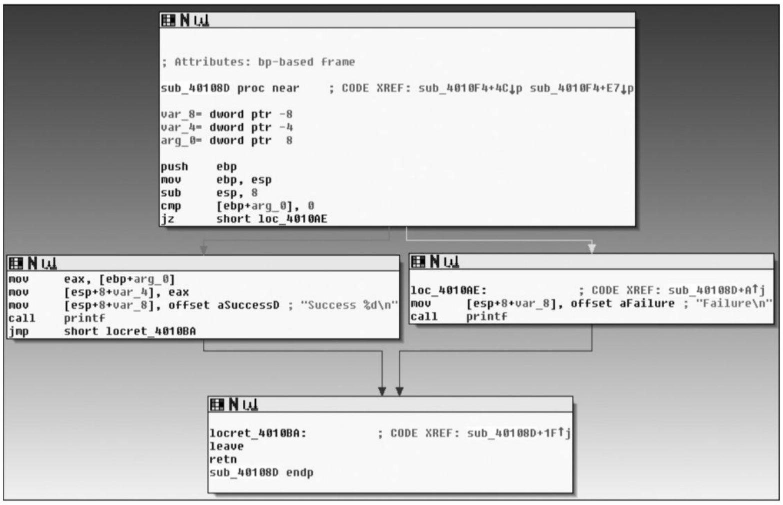
## Graph Mode

- Line numbers & Opcodes

Options > General > Line prefixes
Number of Opcode Bytes = 6

(Optional) Instruction Indentation = 8

- Program flow, arrows
  - ○ conditional jump not taken
  - ○ jump is taken
  - ○ unconditional jump



```
; Attributes: bp-based frame

sub_40108D proc near    ; CODE XREF: sub_4010F4+4C↓p sub_4010F4+E7↓p

var_8= dword ptr -8
var_4= dword ptr -4
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 8
cmp     [ebp+arg_0], 0
jz      short loc_4010AE
```

```
mov     eax, [ebp+arg_0]
mov     [esp+8+var_4], eax
mov     [esp+8+var_8], offset aSuccessD ; "Success %d\n"
call    printf
jmp     short locret_4010BA
```

```
loc_4010AE:                ; CODE XREF: sub_40108D+A↑j
mov     [esp+8+var_8], offset aFailure ; "Failure\n"
call    printf
```

```
locret_4010BA:             ; CODE XREF: sub_40108D+1F↑j
leave
retn
sub_40108D endp
```

*Figure 5-2: Graph mode of the IDA Pro disassembly window*

Disassembly is available in two interfaces: graph mode and text mode. Use the spacebar to quickly switch between the two.

In graph mode, to view memory addresses and opcode values for each instruction in the disassembly, set Options > General > Line prefixes
Number of Opcode Bytes = 6. If this pushes stuff off-screen, try setting Instruction Indentation = 8.

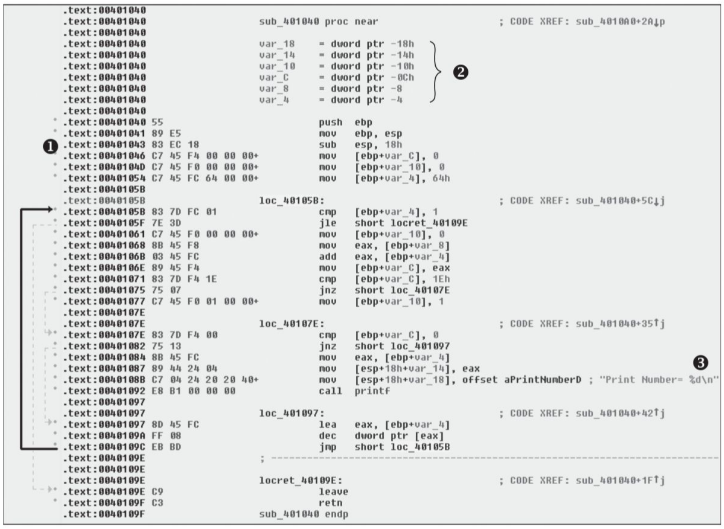
Arrows between code blocks indicate the program's flow, with upwards arrows usually indicating a loop. Arrows will be red to indicate the path given an untaken conditional jump, green if taken, or blue if the jump is unconditional. Any code highlighted in graph mode will also be highlighted in text mode.

# The Interface

## Text Mode

- View data regions
- Arrows window
  - Solid = unconditional
  - Dashed = conditional

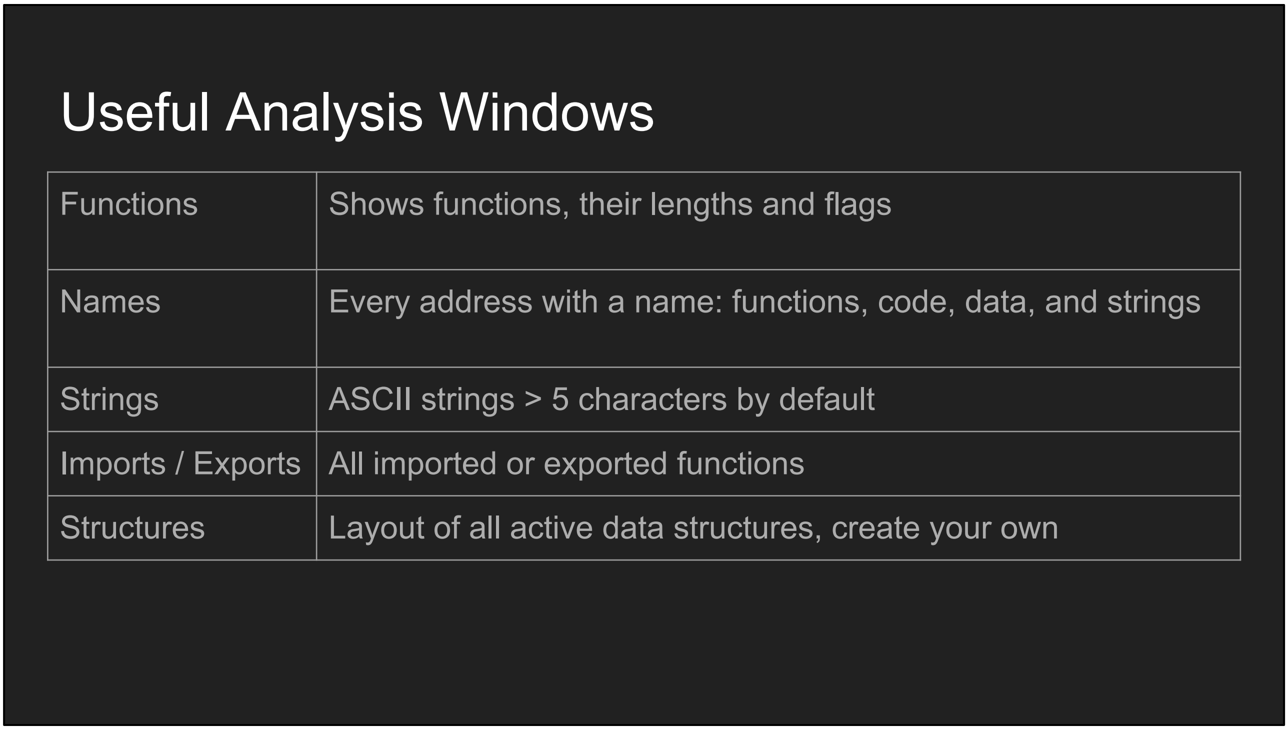
1. Section, address
2. Stack layout
3. Auto-comment
   a. Options > General > Auto comments



```
.text:00401040
.text:00401040                    sub_401040 proc near                     ; CODE XREF: sub_4010A0+2A↓p
.text:00401040
.text:00401040                    var_18     = dword ptr -18h
.text:00401040                    var_14     = dword ptr -14h
.text:00401040                    var_10     = dword ptr -10h
.text:00401040                    var_C      = dword ptr -0Ch
.text:00401040                    var_8      = dword ptr -8
.text:00401040                    var_4      = dword ptr -4
.text:00401040
.text:00401040 55                            push   ebp
.text:00401041 89 E5                         mov    ebp, esp
.text:00401043 83 EC 18                      sub    esp, 18h
.text:00401046 C7 45 F4 00 00 00+            mov    [ebp+var_C], 0
.text:0040104D C7 45 F0 00 00 00+            mov    [ebp+var_10], 0
.text:00401054 C7 45 FC 64 00 00+            mov    [ebp+var_4], 64h
.text:0040105B
.text:0040105B                    loc_40105B:                              ; CODE XREF: sub_401040+5C↓j
.text:0040105B 83 7D FC 01                   cmp    [ebp+var_4], 1
.text:0040105F 7E 3D                         jle    short locret_40109E
.text:00401061 C7 45 F0 00 00 00+            mov    [ebp+var_10], 0
.text:00401068 8B 45 F8                      mov    eax, [ebp+var_8]
.text:0040106B 03 45 FC                      add    eax, [ebp+var_4]
.text:0040106E 89 45 F4                      mov    [ebp+var_C], eax
.text:00401071 83 7D F4 1E                   cmp    [ebp+var_C], 1Eh
.text:00401075 75 07                         jnz    short loc_40107E
.text:00401077 C7 45 F0 01 00 00+            mov    [ebp+var_10], 1
.text:0040107E
.text:0040107E                    loc_40107E:                              ; CODE XREF: sub_401040+35↑j
.text:0040107E 83 7D F4 00                   cmp    [ebp+var_C], 0
.text:00401082 75 13                         jnz    short loc_401097
.text:00401084 8B 45 FC                      mov    eax, [ebp+var_4]
.text:00401087 89 44 24 04                   mov    [esp+18h+var_14], eax
.text:0040108B C7 04 24 20 20 40+            mov    [esp+18h+var_18], offset aPrintNumberD ; "Print Number= %d\n"
.text:00401092 E8 B1 00 00 00                call   printf
.text:00401097
.text:00401097                    loc_401097:                              ; CODE XREF: sub_401040+42↑j
.text:00401097 8D 45 FC                      lea    eax, [ebp+var_4]
.text:0040109A FF 08                         dec    dword ptr [eax]
.text:0040109C EB BD                         jmp    short loc_40105B
.text:0040109E                    ; ---------------------------------------------------------------
.text:0040109E
.text:0040109E                    locret_40109E:                           ; CODE XREF: sub_401040+1F↑j
.text:0040109E C9                            leave
.text:0040109F C3                            retn
.text:0040109F                    sub_401040 endp
```

*Figure 5-3: Text mode of IDA Pro's disassembly window*

Text mode is necessary for viewing the data regions of a binary, but also presents a non-linear view of program flow via the leftmost arrows window, with solid lines representing unconditional jumps and dashed lines for conditional jumps. As with graph mode, upwards arrows typically indicate a loop. Also displayed in text mode is the name of the PE section and memory addresses in which the opcodes reside in memory, the stack layout for the function, and any comments added by IDA (which is a useful feature that can be turned on in Options > General > Auto comments).



## Useful Analysis Windows

| | |
|---|---|
| Functions | Shows functions, their lengths and flags |
| Names | Every address with a name: functions, code, data, and strings |
| Strings | ASCII strings > 5 characters by default |
| Imports / Exports | All imported or exported functions |
| Structures | Layout of all active data structures, create your own |

Other useful windows for analysis include

Functions window:

Lists all functions and their lengths.

Sort by function length and filter for large, interesting functions.

The L function flag can save you time during analysis by allowing you to identify and skip library functions.

Names window:

Lists every address with a name, including functions, named code, named data, and strings.

Strings window:

Shows all ASCII strings > 5 characters. Adjust by right-clicking in the window > Setup.

Imports window:

Lists all imports for a file.

Exports window:

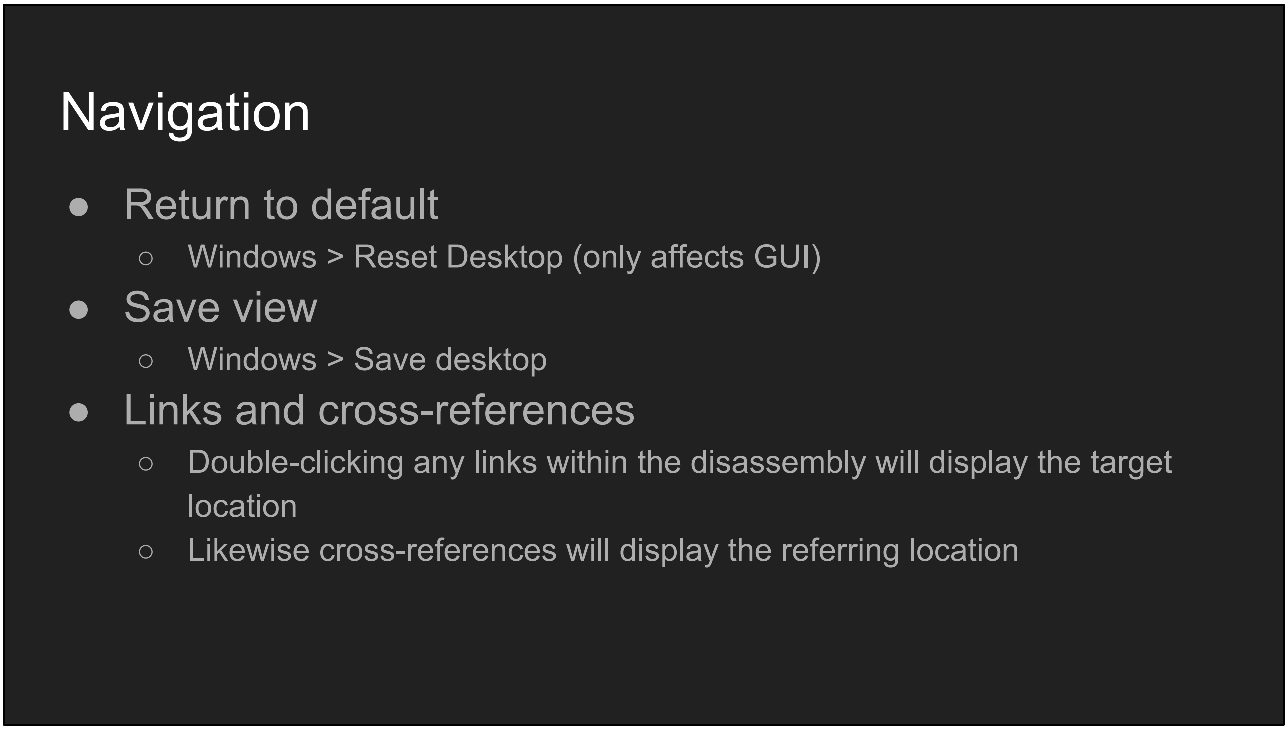
Lists all the exported functions for a file, useful for DLLs.

Structures window:

Lists the layout of all active data structures.

Create your own data structures for use as memory layout templates.

“Use the cross-reference feature of these windows to locate interesting code, for example, to find all code locations that call an imported function, use the import window, double- click the imported function of interest, and then use the cross-reference feature to locate the import call in the code listing.”

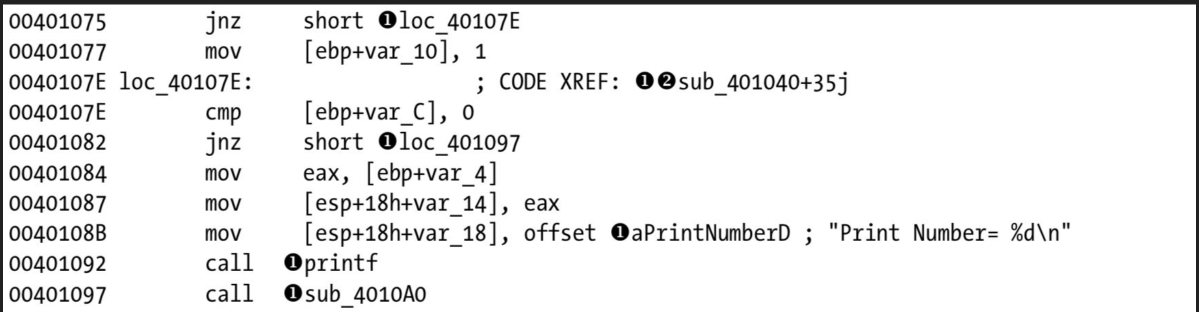


## Navigation

- Return to default
  - Windows > Reset Desktop (only affects GUI)
- Save view
  - Windows > Save desktop
- Links and cross-references
  - Double-clicking any links within the disassembly will display the target location
  - Likewise cross-references will display the referring location

Return to default
Windows > Reset Desktop (only affects GUI)
Save view
Windows > Save desktop
Links and cross-references
Double-clicking any links within the disassembly will display the target location
Likewise cross-references will display the referring location

# Navigational Links



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

Sub links - links to the start of functions (printf and sub_4010A0).

Loc links - links to jump destinations (loc_40107E and loc_401097).

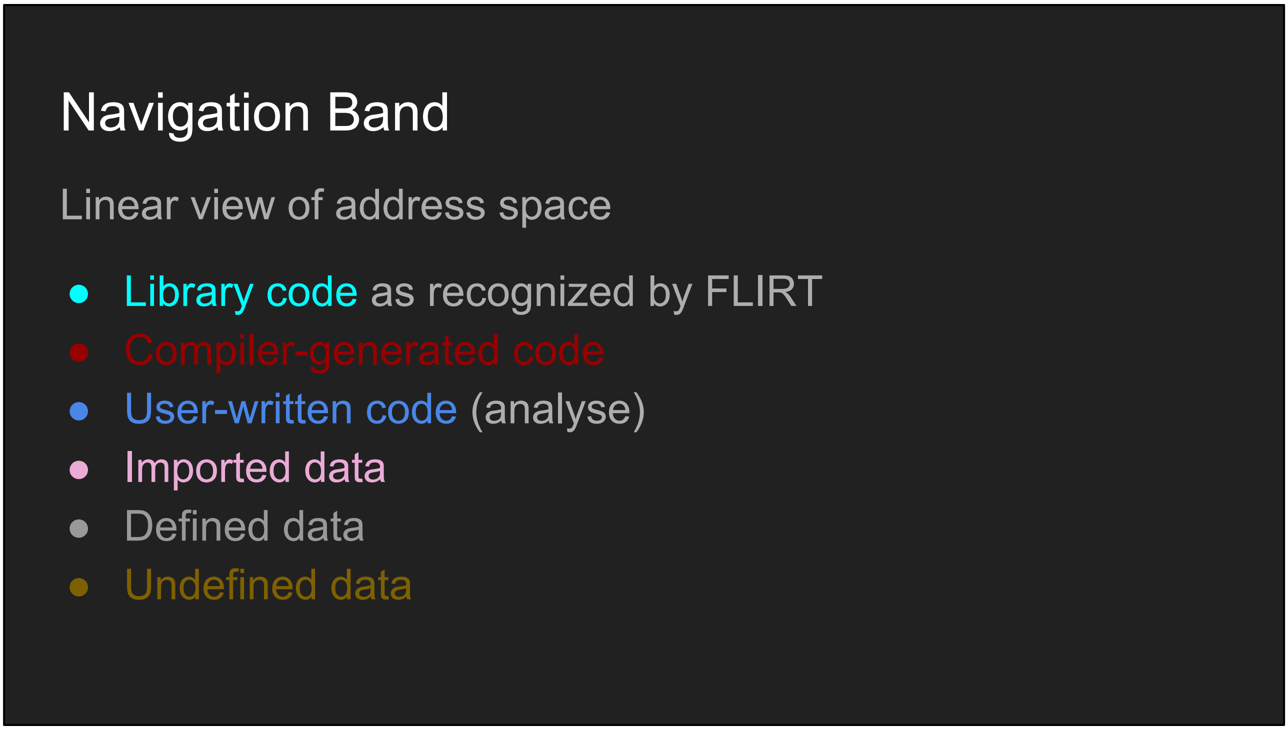
Offset links are links to an offset in memory.

The most common types of links include:

Sub links - links to the start of functions (printf and sub_4010A0).
Loc links - links to jump destinations (loc_40107E and loc_401097).
Offset links are links to an offset in memory.

(2) would display 0x401075

Strings are typically navigational links that will display where they are defined in memory

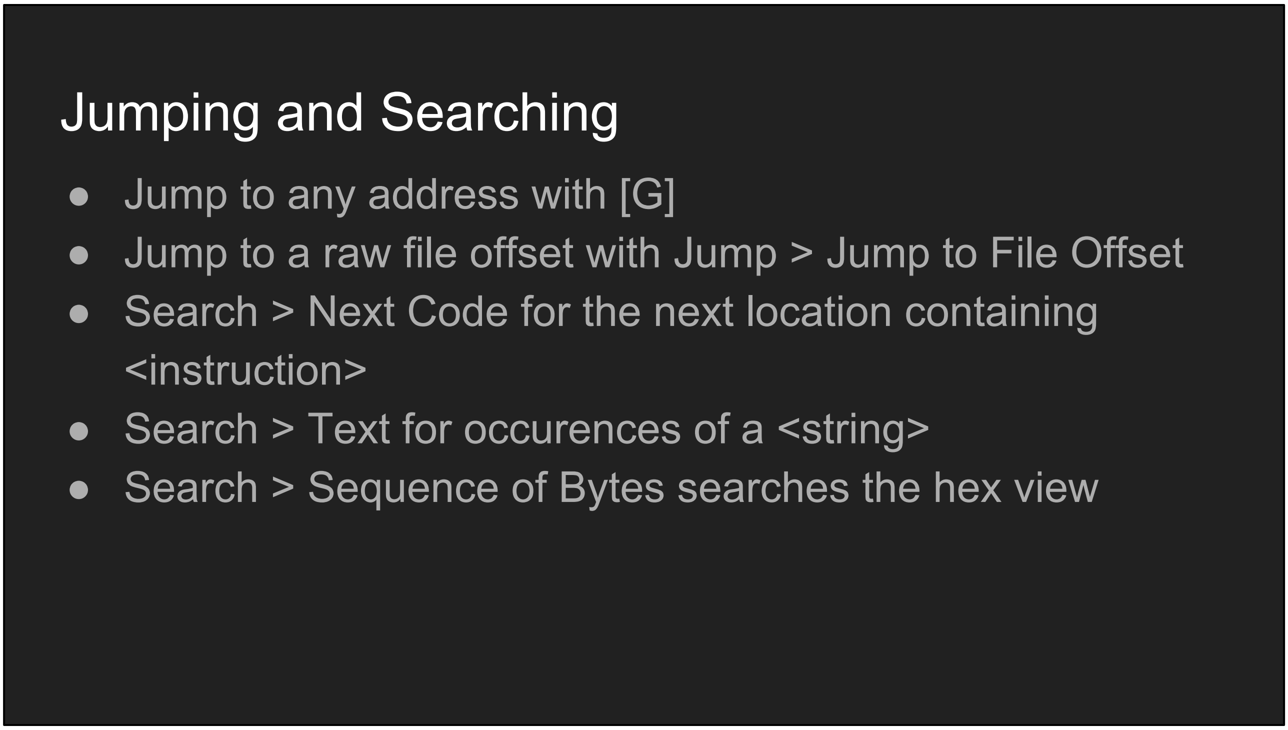


## Navigation Band

Linear view of address space

- Library code as recognized by FLIRT
- Compiler-generated code
- User-written code (analyse)
- Imported data
- Defined data
- Undefined data

The navigation band provides a color-coded linear view of the binary's address space.
Light blue is library code as recognized by FLIRT.
Red is compiler-generated code.
Dark blue is user-written code.
Pink is imported data
Grey is defined data
Brown is undefined data



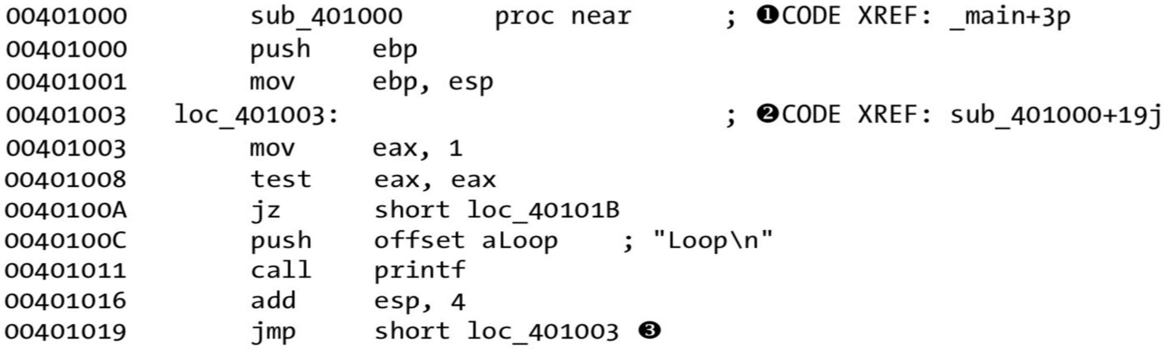
## Jumping and Searching

- Jump to any address with [G]
- Jump to a raw file offset with Jump > Jump to File Offset
- Search > Next Code for the next location containing <instruction>
- Search > Text for occurences of a <string>
- Search > Sequence of Bytes searches the hex view

Jump to any address with [G]
Jump to a raw file offset with Jump > Jump to File Offset
Search > Next Code for the next location containing <instruction>
Search > Text for occurences of a <string>
Search > Sequence of Bytes searches the hex view

# Code Cross-References

(xref) - where a function is called or a string is used



```
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
```

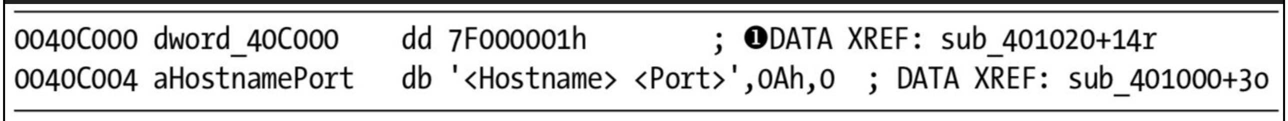
[X] to show all xrefs for a given function

“A code cross-reference at (1) tells us that this function (sub_401000) is called from inside the main function at offset 0x3 into the main function.
The code cross-reference for the jump at (2) tells us which jump takes us to this location, which in this example corresponds to the location marked at (3).
We know this because at offset 0x19 into sub_401000 is the jmp at memory address 0x401019.”

To view all xrefs for a given function, click the function name and press [x]

# Data Cross-References

Track the way data (any byte referenced in code via a memory reference) is accessed within a binary

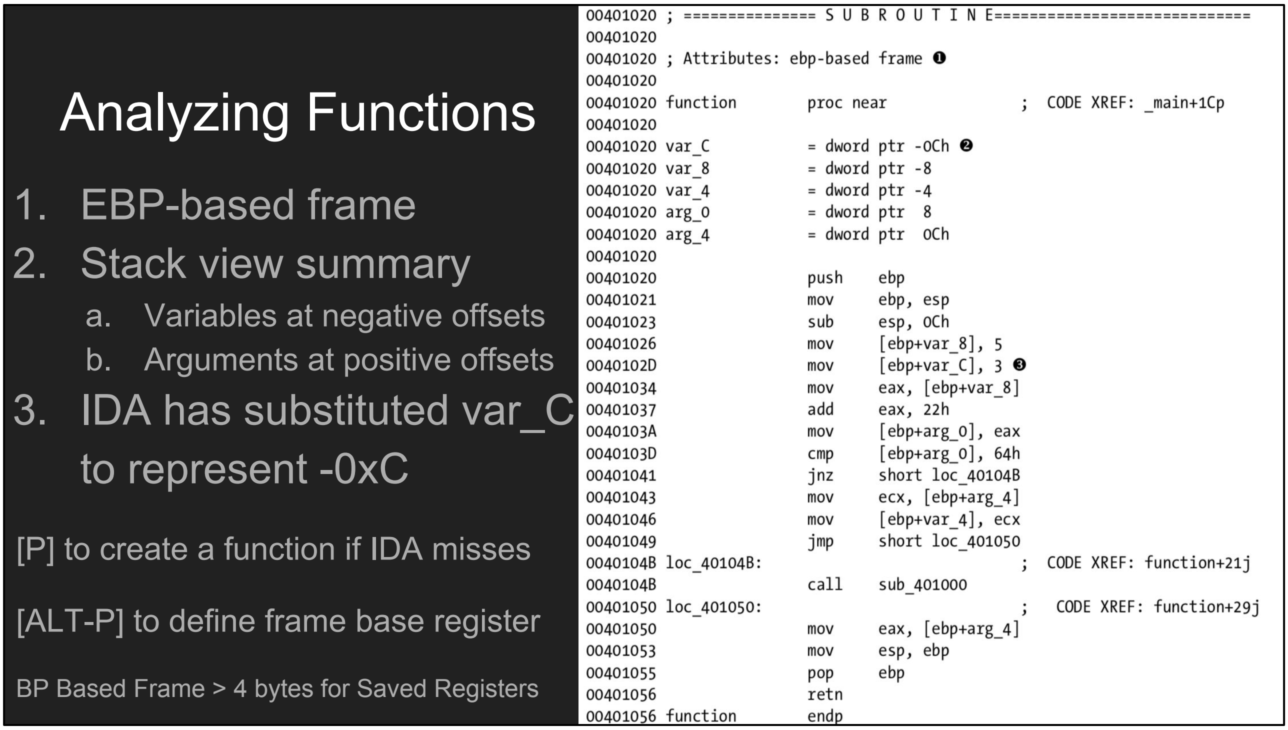


```
0040C000 dword_40C000    dd 7F000001h        ; ❶DATA XREF: sub_401020+14r
0040C004 aHostnamePort   db '<Hostname> <Port>',0Ah,0  ; DATA XREF: sub_401000+3o
```

*Listing 5-3: Data cross-references*

“Data cross-references are used to track the way 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 cross-reference to the DWORD 0x7F000001 at (1). The corresponding cross-reference 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>.”



## Analyzing Functions

1. EBP-based frame
2. Stack view summary
   a. Variables at negative offsets
   b. Arguments at positive offsets
3. IDA has substituted var_C to represent -0xC

[P] to create a function if IDA misses

[ALT-P] to define frame base register

BP Based Frame > 4 bytes for Saved Registers

```
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
```

Here we can see an example where IDA has recognized a function, labeled it, and shows the local variables and parameters.

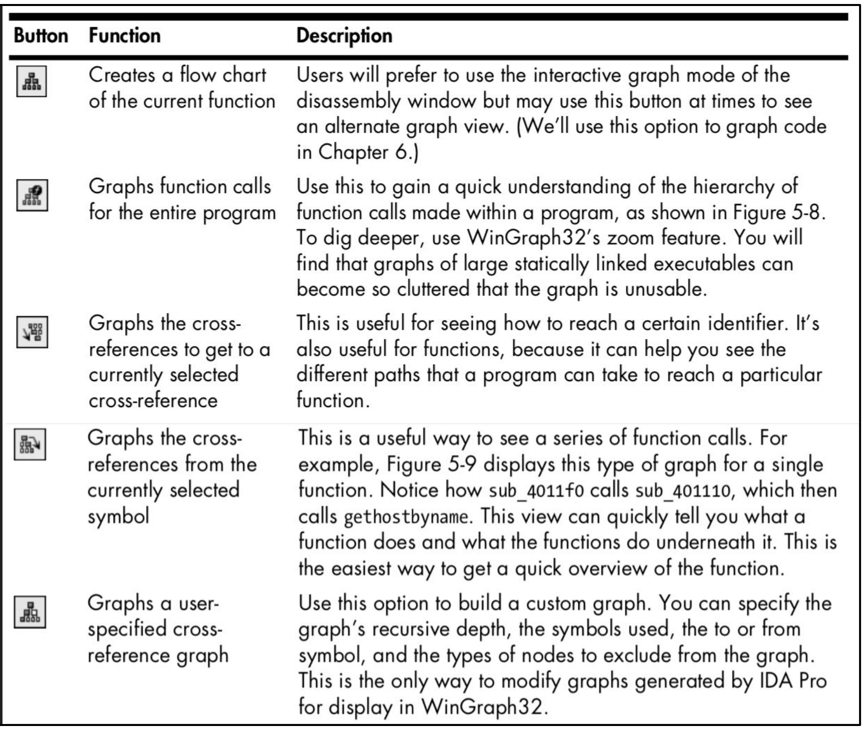
To start, IDA notes that this is an EBP-based stack frame, meaning that local variables and parameters will be referenced relative to the EBP register throughout this function. Note that IDA will only label things it finds, but it may not find everything from the original source.

In the stack view summary starting at (2) we can see that local variables are at negative offsets relative to EBP, and arguments at positive offsets. We can also see that IDA has substituted the label var_C for the value -0xC and uses this name in place of the value in a mov instruction at (3) to set var_C = 3.

If IDA fails to identify a function, you can manually define one with [P]. If IDA fails to identify the base register of the frame you may see instructions mov [ebp-0Ch], eax and push dword ptr [ebp-010h] instead of IDA’s labeling. You can fix this with [ALT-P] > BP Based Frame > 4 bytes for Saved Registers

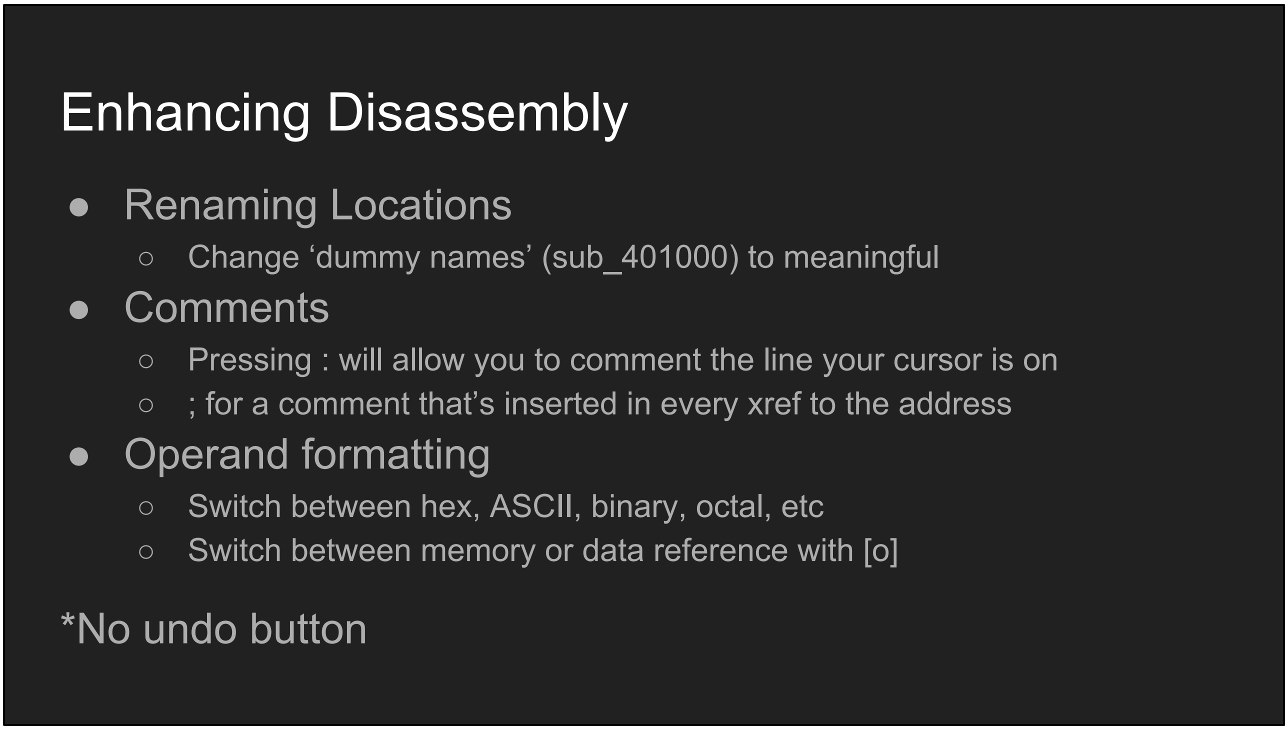
# Graphing Options

- Cannot be modified



| 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. |
| | 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 `gethostbyname`. 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. |

IDA supports five graphing options to build legacy graphs that cannot be manipulated but may be useful for viewing various cross-references



# Enhancing Disassembly

- Renaming Locations
  - Change ‘dummy names’ (sub_401000) to meaningful
- Comments
  - Pressing : will allow you to comment the line your cursor is on
  - ; for a comment that’s inserted in every xref to the address
- Operand formatting
  - Switch between hex, ASCII, binary, octal, etc
  - Switch between memory or data reference with [o]

*No undo button

The interactive component of IDA allows us to make multiple modifications to the disassembly in order to make it more human-readable and easier to process

Renaming Locations
    Change ‘dummy names’ (sub_401000) to meaningful
Comments
    Pressing : will allow you to comment the line your cursor is on
    ; for a comment that’s inserted in every xref to the address
Operand formatting
    Switch between hex, ASCII, binary, octal, etc (IDA usually marks things as hex)
    Switch between memory or data reference with [o] (IDA may misrepresent data as a memory address, for example)

*No undo button so be careful with any changes

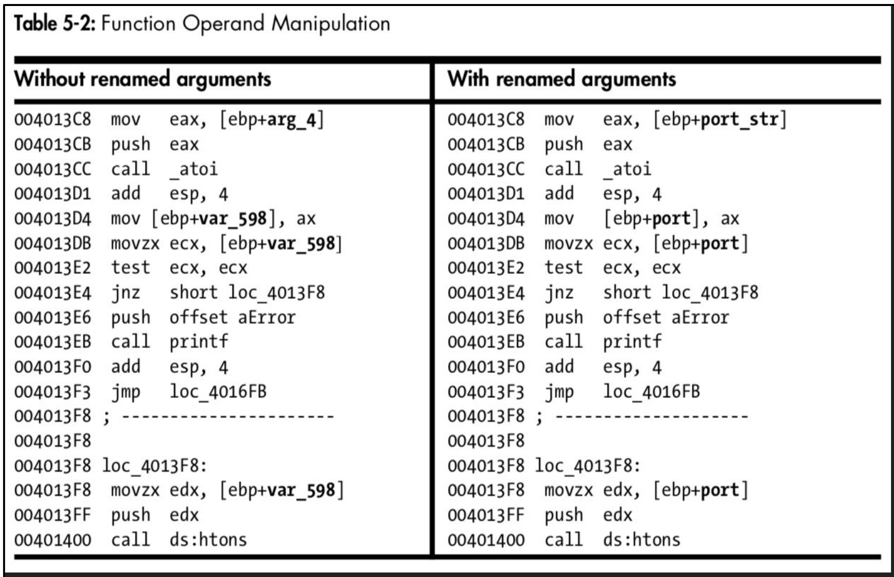
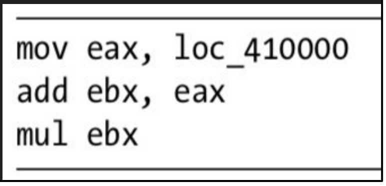


Table 5-2: Function Operand Manipulation

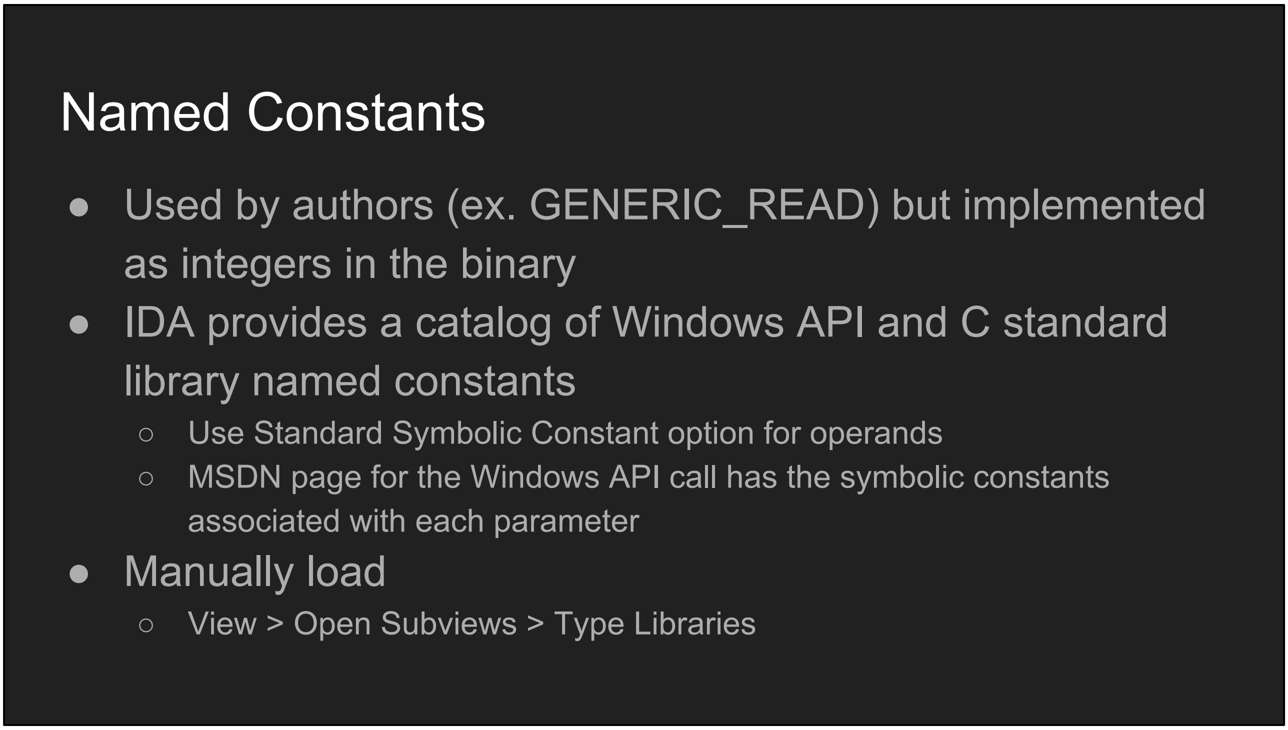
| Without renamed arguments | With renamed arguments |
|---|---|
| `004013C8  mov   eax, [ebp+`**`arg_4`**`]` | `004013C8  mov   eax, [ebp+`**`port_str`**`]` |
| `004013CB  push  eax` | `004013CB  push  eax` |
| `004013CC  call  _atoi` | `004013CC  call  _atoi` |
| `004013D1  add   esp, 4` | `004013D1  add   esp, 4` |
| `004013D4  mov [ebp+`**`var_598`**`], ax` | `004013D4  mov   [ebp+`**`port`**`], ax` |
| `004013DB  movzx ecx, [ebp+`**`var_598`**`]` | `004013DB  movzx ecx, [ebp+`**`port`**`]` |
| `004013E2  test  ecx, ecx` | `004013E2  test  ecx, ecx` |
| `004013E4  jnz   short loc_4013F8` | `004013E4  jnz   short loc_4013F8` |
| `004013E6  push  offset aError` | `004013E6  push  offset aError` |
| `004013EB  call  printf` | `004013EB  call  printf` |
| `004013F0  add   esp, 4` | `004013F0  add   esp, 4` |
| `004013F3  jmp   loc_4016FB` | `004013F3  jmp   loc_4016FB` |
| `004013F8 ; ----------------------` | `004013F8 ; --------------------` |
| `004013F8` | `004013F8` |
| `004013F8 loc_4013F8:` | `004013F8 loc_4013F8:` |
| `004013F8  movzx edx, [ebp+`**`var_598`**`]` | `004013F8  movzx edx, [ebp+`**`port`**`]` |
| `004013FF  push  edx` | `004013FF  push  edx` |
| `00401400  call  ds:htons` | `00401400  call  ds:htons` |



```
mov eax, loc_410000
add ebx, eax
mul ebx
```

The top left table shows noticeable readability improvements made by assigning names to variables and arguments in the disassembly

The bottom right image shows an example where IDA has misrepresented a value as a memory address, which needs modified to correctly show the value 4259840 (0x410000 in hex)



## Named Constants

- Used by authors (ex. GENERIC_READ) but implemented as integers in the binary
- IDA provides a catalog of Windows API and C standard library named constants
  - Use Standard Symbolic Constant option for operands
  - MSDN page for the Windows API call has the symbolic constants associated with each parameter
- Manually load
  - View > Open Subviews > Type Libraries

Used by authors (ex. GENERIC_READ) but implemented as integers in the binary
IDA provides a catalog of Windows API and C standard library named constants
    Use Standard Symbolic Constant option for operands
    MSDN page for the Windows API call has the symbolic constants associated with each parameter
Manually load
    View > Open Subviews > Type Libraries

*“Normally, mssdk and vc6win will automatically be loaded, but if not, you can load them manually (as is often necessary with malware that uses the Native API, the Windows NT family API). To get the symbolic constants for the Native API, load ntapi (the Microsoft Windows NT 4.0 Native API). In the same vein, when analyzing a Linux binary, you may need to manually load the gnuunx (GNU C++ UNIX) libraries.”

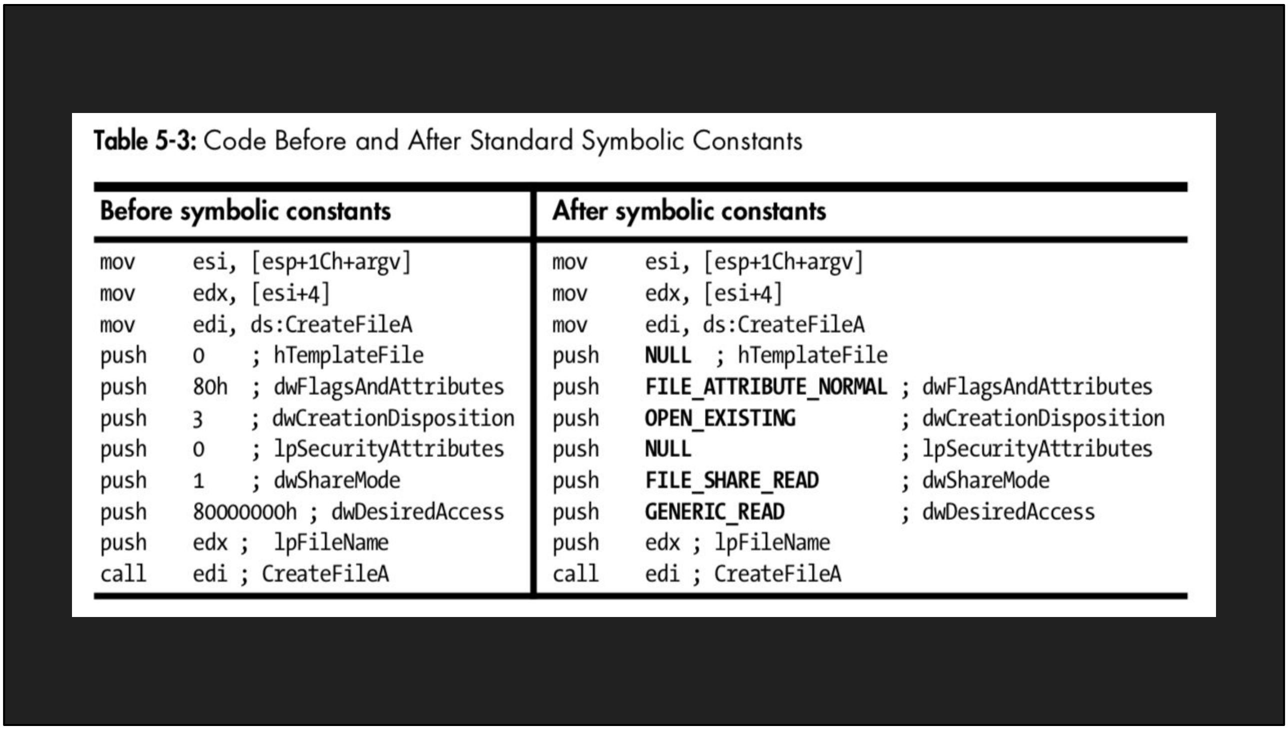


**Table 5-3:** Code Before and After Standard Symbolic Constants

| Before symbolic constants | After symbolic constants |
|---|---|
| mov esi, [esp+1Ch+argv] | mov esi, [esp+1Ch+argv] |
| mov edx, [esi+4] | mov edx, [esi+4] |
| mov edi, ds:CreateFileA | mov edi, ds:CreateFileA |
| push 0 ; hTemplateFile | push **NULL** ; hTemplateFile |
| push 80h ; dwFlagsAndAttributes | push **FILE_ATTRIBUTE_NORMAL** ; dwFlagsAndAttributes |
| push 3 ; dwCreationDisposition | push **OPEN_EXISTING** ; dwCreationDisposition |
| push 0 ; lpSecurityAttributes | push **NULL** ; lpSecurityAttributes |
| push 1 ; dwShareMode | push **FILE_SHARE_READ** ; dwShareMode |
| push 80000000h ; dwDesiredAccess | push **GENERIC_READ** ; dwDesiredAccess |
| push edx ; lpFileName | push edx ; lpFileName |
| call edi ; CreateFileA | call edi ; CreateFileA |

Example of named constant modifications to positively affect readability

# Redefining Code and Data

[U] to undefine incorrectly defined code/data/etc

- [C] to define raw bytes as code
- [D] for data
- [A] for ASCII

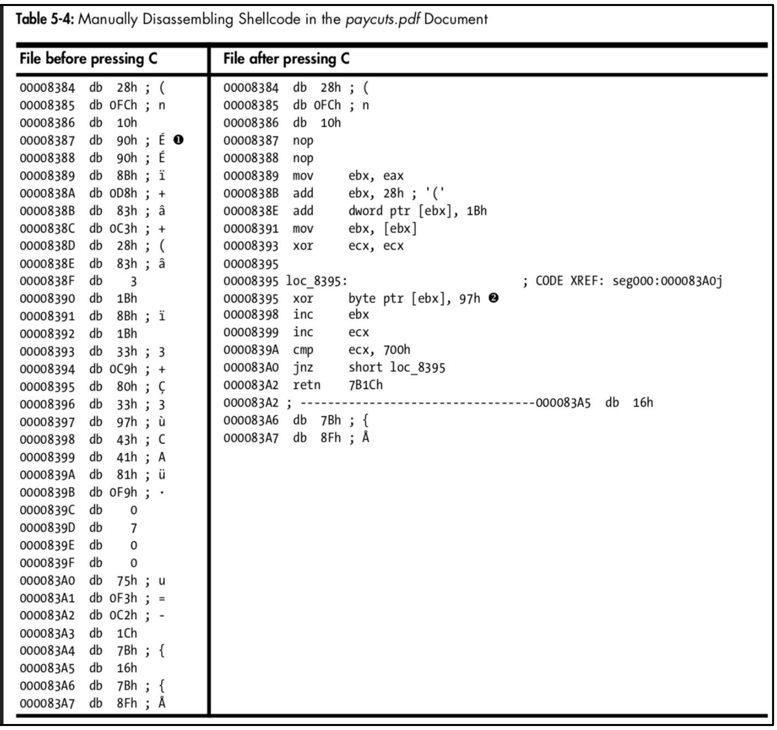


**Table 5-4:** Manually Disassembling Shellcode in the *paycuts.pdf* Document

**File before pressing C**

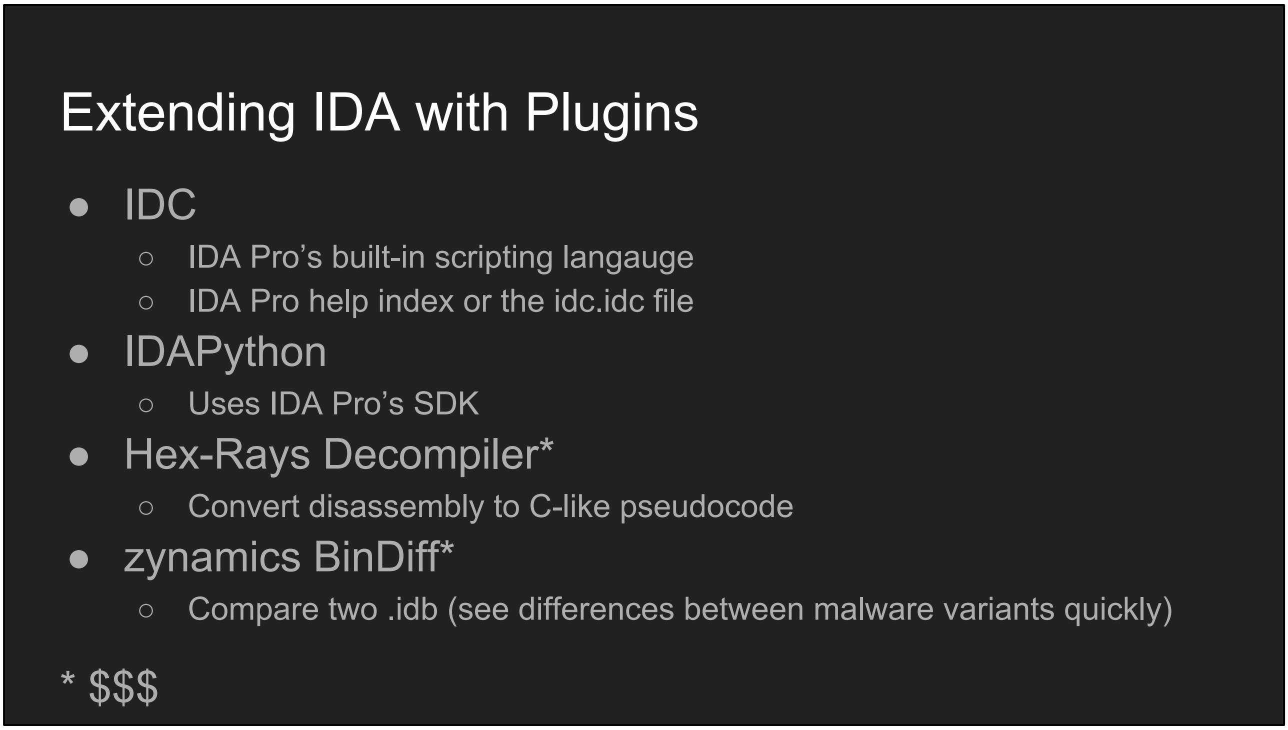
```
00008384  db  28h ; (
00008385  db 0FCh ; n
00008386  db  10h
00008387  db  90h ; É ❶
00008388  db  90h ; É
00008389  db  8Bh ; ï
0000838A  db 0D8h ; +
0000838B  db  83h ; â
0000838C  db 0C3h ; +
0000838D  db  28h ; (
0000838E  db  83h ; â
0000838F  db    3
00008390  db  1Bh
00008391  db  8Bh ; ï
00008392  db  1Bh
00008393  db  33h ; 3
00008394  db 0C9h ; +
00008395  db  80h ; Ç
00008396  db  33h ; 3
00008397  db  97h ; ù
00008398  db  43h ; C
00008399  db  41h ; A
0000839A  db  81h ; ü
0000839B  db 0F9h ; ·
0000839C  db    0
0000839D  db    7
0000839E  db    0
0000839F  db    0
000083A0  db  75h ; u
000083A1  db 0F3h ; =
000083A2  db 0C2h ; -
000083A3  db  1Ch
000083A4  db  7Bh ; {
000083A5  db  16h
000083A6  db  7Bh ; {
000083A7  db  8Fh ; Å
```

**File after pressing C**

```
00008384  db  28h ; (
00008385  db 0FCh ; n
00008386  db  10h
00008387  nop
00008388  nop
00008389  mov     ebx, eax
0000838B  add     ebx, 28h ; '('
0000838E  add     dword ptr [ebx], 1Bh
00008391  mov     ebx, [ebx]
00008393  xor     ecx, ecx
00008395
00008395 loc_8395:                              ; CODE XREF: seg000:000083A0j
00008395  xor     byte ptr [ebx], 97h ❷
00008398  inc     ebx
00008399  inc     ecx
0000839A  cmp     ecx, 700h
000083A0  jnz     short loc_8395
000083A2  retn    7B1Ch
000083A2 ; ----------------------------------000083A5  db  16h
000083A6  db  7Bh ; {
000083A7  db  8Fh ; Å
```

When IDA miscategorizes code as data, data as code, etc you can undefine with [U] and redefine the raw bytes as [C]ode, [D]ata, or [A]SCII strings.

In the image we can see some shellcode defined as raw bytes on the left, and redefined as code on the right to reveal an XOR decoding loop using 0x97



## Extending IDA with Plugins

- IDC
  - IDA Pro’s built-in scripting langauge
  - IDA Pro help index or the idc.idc file
- IDAPython
  - Uses IDA Pro’s SDK
- Hex-Rays Decompiler*
  - Convert disassembly to C-like pseudocode
- zynamics BinDiff*
  - Compare two .idb (see differences between malware variants quickly)

* $$$

Run external scripts using File > Script File or one of the Command options under File, see the log output for debugging or status messages. See pages 103 - 106 of the book if you’re really interested in IDC or IDAPython specifics and examples.

The Hex-Rays Decompiler and zynamics BinDiff extensions are popular and useful, but can be quite expensive for personal use

# Resources

IDA Free v7.0

The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler, 2nd Edition (No Starch Press, 2011)