Reverse Engineering with Ida Pro

Chris Eagle
cseagle@redshift.com
Blackhat Training
Federal 2006
Administrivia

• Welcome!
• Please turn in your "A" ticket in exchange for a CD and printed notes
  – WARNING – the CD contains code that will trigger your AV software
Adminstrivia

• Class only wireless (i.e. no internet)
  – Ssid: ctf
  – Wep key:
    0xAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    • i.e. hex key consisting of 26 A's

• Class ftp server
  – 172.16.5.11
Administrivia

- cygwin users
  - Make sure you have gcc/g++ AND make installed before tomorrow
Background

• Personal experience
  – 20+ years assembly/C/C++/…
  – 8 years teaching graduate level CS
    • Programming languages
    • Forensics
    • Computer network attack/defense

• Interests
  – Obfuscated code analysis
Class Background

• Profession
  – Industry?
    • Hardware
    • Software
  – Government?
  – Academic?

• Experience
  – IDA?
  – x86? Other ASM?
  – Windows? Linux?
Expectations/Goals

• Discover how a program works
  – Build compatible software
• Locate hidden functionality
  – Backdoors etc.
• Search for vulnerabilities in closed source software
Introduction

- Reverse engineering with Ida
  - Created by Ilfak Guilfanov
  - Premier disassembly tool available today
    - Interactive
    - Many platforms supported
    - Highly extensible
Basic Disassembly Theory
Disassembly

- Need the proper tools
- Tools must understand executable file format
  - Unless you are dying to parse the file yourself in a hex editor
- Parse machine language op codes back to their assembly language equivalents
  - Must know when to stop, data vs. code
Disassemblers vs. Debuggers

- Debuggers by nature are designed to run code
  - All can disassemble if asked to
    - Single functions
    - From IP forward
  - Most don't do batch disassembly
- Disassemblers aren't interested in running code
Disassemblers

- Two main types
  - Linear sweep
  - Recursive descent
- Output is generally a disassembly listing
  - Can yield extremely large text files
  - Difficult to navigate/change
- Disassembly fails to reveal obfuscated code
Disassembly Tools

- **Linux**
  - objdump
    - Provides a lot of info, see man page for switches
      - objdump -d /bin/cat
  - gdb
    - Can generate disassembly listings but they are cumbersome

- **Windows**
  - Interactive Disassembler Pro (IdaPro)
    - Understands most executable file formats
Binary File Formats
Common Formats

- Executable and Linkable (ELF) Format
  - Found on Linux/Unix
  - Described in file docs/ELF_Format.pdf on the CD

- Portable Executable (PE) Format
  - Windows
  - Several files in the docs directory on the CD
Common Elements

• Each format specifies header fields that describe
  – Characteristics of the executable
  – Point to various portions of the executable
  – Import and export fields
  – Debugging information
  – Others
Essential Information

• Virtual address info
  – Where to load
  – Program entry point

• Relocation information
  – How to modify the memory image if it can't be loaded at its preferred location

• Program section descriptions
  – Where and how large various sections are
Program Sections

• Many different types
  – Code sections contain the executable portions of the program
    • Often named ".text"
  – Data sections contain various types of statically allocated data
    • Read only data - .rodata
    • Read/write initialized data - .data
    • Read/write un-initialized data - .bss
Program Sections (cont)

– Import sections
  • Procedure linkage table - .plt
  • Global offset table - .got
  • Import table - .idata

– Other sections
  • Some sections are required only by the linker and are not used at run time
Ida Pro
IDA Pro

- Interactive Disassembler Professional
  - [http://www.datarescue.com/idabase](http://www.datarescue.com/idabase)
- Recursive descent disassembler
- Premier disassembly tool for reverse engineers
  - Handles many families of assembly language
- Interactive manipulation of disassembly listing
- Scripting and plugins
- Runs on Windows and Linux
IDA Pro Operation

- Load the binary of interest
- IDA builds a database to characterize each byte of the binary
  - All manipulations of the disassembly involve database interactions
- Performs detailed analysis of code
  - Recognizes function boundaries and library calls
  - Recognizes data types for known library calls
Ida Pro Features

- Graph based display of program flow
- Flowchart display of function flow
- Displays data and code cross references
  - List of all locations that refer to a particular piece of data
  - List of all locations that call a particular function
- Automatic recognition of string constants
Ida Pro Features

- Hex display option
- Separate strings window
- Separate list of all symbols in the program
- Very nice stack frame displays
- Allows you to assign your own names to code locations/functions
- Allows you to assign your own names to function locals and parameters
Ida Basics
Assembly Notes

• We will use "intel" syntax throughout
  – MOV <dest>, <src>
    • This is what IDA produces
    • objdump –d –M intel <file>
    • gdb – set disassembly-flavor intel
  – As opposed to "AT&T" syntax
    • MOV <src>, <dest>
    • Default for objdump, gdb
**Stack Terminology/Display**

- For this class
- A is "above" B on stack to the right
  - Though it lies at a lower memory address

```
Stack bottom == higher memory addresses

Lower memory addresses

...  

...  

...  

esp == top of stack

C  

B  

A  

...  
```
Using Ida Pro

• Open Ida
• Choose "New" to start a new project or "Previous" to resume previous work
• If "New" selected, navigate to the file you wish to disassemble and open it
• Ida should recognize the file format and start to analyze your file
  – Displays as much info as possible taking symbol tables and debugging info into account
Basic Ida Walkthrough

- Open the file
  - demos/asm_code_samples_bor.exe
- Observe file type identification
- Ida analyzes file and opens various analysis windows
- The source code for this file can be found in
  - demos/asm_code_samples.c
  - Open it for comparison with the binary
Ida Open File Dialog

![Ida Open File Dialog](image)
Caution

• IDA began life as a DOS application
  – Virtually every action has a hot key sequence
    • Consequently, virtually every key makes something happen
    • THERE IS NO UNDO IN IDA PRO

• Almost all IDA actions are also available via menu items or toolbar buttons
Ida Navigation

- Double click on a reference to a name and IDA jumps to the named location
  - Names can be
    - Function names
    - Local jump targets within a function
    - Global variable names
- IDA maintains a web-browser-like history list
  - The ESC key acts like a back button
  - There are also forward and backward arrows to move back and forth as well
IDA View Window

- This is the main working window
  - Disassembly view
- Disassembly initially positioned at entry point or main
  - Entry point for programs is generally not main but a location named start or _start
    - Start does program setup before calling main
  - If main is present, Ida will position cursor there
Names Window

- Lists all known named locations in program
  - Based on imports, exports, and some analysis
  - F is a function
  - L is a library function
  - C is code/instruction
  - A is a string
  - D is defined data
  - I is an imported function
  - Dynamically linked
<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>004010C</td>
</tr>
<tr>
<td>___GetExceptDLLinfo</td>
<td>004010E</td>
</tr>
<tr>
<td>___isDLL</td>
<td>004010E</td>
</tr>
<tr>
<td>___getHInstance</td>
<td>004010E</td>
</tr>
<tr>
<td>Sysinit::<strong>linkproc</strong> GetTls(void)</td>
<td>0040114</td>
</tr>
<tr>
<td>main</td>
<td>0040130</td>
</tr>
<tr>
<td>_calloc</td>
<td>0040145</td>
</tr>
<tr>
<td>_rtL_close</td>
<td>0040145</td>
</tr>
<tr>
<td>_close</td>
<td>0040145</td>
</tr>
<tr>
<td>@_virt_reserve</td>
<td>0040145</td>
</tr>
<tr>
<td>@_virt_alloc</td>
<td>0040152</td>
</tr>
<tr>
<td>@_virt_commit</td>
<td>0040154</td>
</tr>
<tr>
<td>@_virt_decommit</td>
<td>0040159</td>
</tr>
<tr>
<td>@_virt_release</td>
<td>004015E</td>
</tr>
<tr>
<td>__CRTL_MEM_GetBorMemPtrs</td>
<td>0040158</td>
</tr>
<tr>
<td>__CRTL_MEM_CheckBorMem</td>
<td>0040168</td>
</tr>
</tbody>
</table>
Strings Window

- Strings window
  - Complete listing of embedded strings within program
  - Configurable
    - Right click in Strings window and choose setup
    - Can change minimum length or style of string to search for
      - IDA rescans for strings if you change settings
<table>
<thead>
<tr>
<th>Address</th>
<th>Length</th>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;...&quot;,data:00... 00000010</td>
<td>C</td>
<td></td>
<td>Bad file number</td>
</tr>
<tr>
<td>&quot;...&quot;,data:00... 00000015</td>
<td>C</td>
<td></td>
<td>Memory arena trashed</td>
</tr>
<tr>
<td>&quot;...&quot;,data:00... 00000012</td>
<td>C</td>
<td></td>
<td>Not enough memory</td>
</tr>
<tr>
<td>&quot;...&quot;,data:00... 0000001D</td>
<td>C</td>
<td></td>
<td>Invalid memory block</td>
</tr>
<tr>
<td>&quot;...&quot;,data:00... 00000014</td>
<td>C</td>
<td></td>
<td>Invalid environment</td>
</tr>
<tr>
<td>&quot;...&quot;,data:00... 0000000F</td>
<td>C</td>
<td></td>
<td>Invalid format</td>
</tr>
<tr>
<td>&quot;...&quot;,data:00... 00000014</td>
<td>C</td>
<td></td>
<td>Invalid access code</td>
</tr>
<tr>
<td>&quot;...&quot;,data:00... 0000000D</td>
<td>C</td>
<td></td>
<td>Invalid data</td>
</tr>
<tr>
<td>&quot;...&quot;,data:00... 0000000C</td>
<td>C</td>
<td></td>
<td>Bad address</td>
</tr>
<tr>
<td>&quot;...&quot;,data:00... 0000000F</td>
<td>C</td>
<td></td>
<td>No such device</td>
</tr>
<tr>
<td>&quot;...&quot;,data:00... 00000026</td>
<td>C</td>
<td></td>
<td>Attempted to remove copy</td>
</tr>
<tr>
<td>&quot;...&quot;,data:00... 00000010</td>
<td>C</td>
<td></td>
<td>Not same device</td>
</tr>
<tr>
<td>&quot;...&quot;,data:00... 0000000E</td>
<td>C</td>
<td></td>
<td>No more files</td>
</tr>
<tr>
<td>&quot;...&quot;,data:00... 00000011</td>
<td>C</td>
<td></td>
<td>Invalid argument</td>
</tr>
<tr>
<td>&quot;...&quot;,data:00... 00000011</td>
<td>C</td>
<td></td>
<td>Arg list too big</td>
</tr>
<tr>
<td>&quot;...&quot;,data:00... 00000012</td>
<td>C</td>
<td></td>
<td>Exec format error</td>
</tr>
</tbody>
</table>
Ida Interaction

• One of the greatest strengths of Ida is the ability to interact with a disassembly
  – Rather than a static disassembly file generated by a tool such as objdump

• Among other things you can do
  – Renaming
  – Reformatting code-data-code
  – Adding comments
  – Many others
Renaming in Ida

• Having source code is cheating
  – But useful today so we can see original names used by the programmer

• Compilation is a lossy operation
  – In a binary we are lucky to get functions names
    • Not always the case
  – Never get local variable names
Ida Names

• Just about anything in Ida can have a name
  – Any address or stack variable
• Ida will assign names based on
  – Symbol table in binary
  – Default generated name
  – User assigned
Ida Default Names

• **sub_xxxx**
  – function starting at address xxxx

• **loc_xxxx**
  – Code at location xxxx that is referenced from elsewhere, generally a branch target

• **byte_xxxx, word_xxxx, dword_xxxx**
  – Byte, word or dword data at location xxxx
Changing/Adding Names

- The name of anything can be set or changed
- Edit/Rename, hotkey is ‘n’
- Place the cursor on the item that you wish to rename and press ‘n’
- Opens dialog to rename variable or address
Example

• In the Ida View window, click on sub_401150 at this line:
  \texttt{.text:004013FA \ call \ sub_401150}
  \begin{itemize}
  \item Press ‘n’ to open a rename window
  \item This particular window applies to renaming addresses
  \end{itemize}
• Enter the new name ‘simple_if’
• Changing a globally scoped name adds it to the Names window
Before Renaming

```
; 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
push ebx
push esi
push edi
xor eax, eax
mov [ebp+var_4]
mov ebx, 1
mov esi, 2
mov edi, 3
push esi
push ebx
call sub_401150
add esp, 8
push edi
push esi
push ebx
```

```
Rename address

Address: 0x401150
Name [simple it]

Maximum length of new names 15
Local name prefix @@

- Local name
- Include in names list
- Public name
- Autogenerated name
- Weak name
- Create name anyway

OK Cancel Help
```
After Renaming

```assembly
.text:004013D0 ; int __cdecl main(int argc, const char **argv, cc
.text:004013DD _main proc near
.text:004013DD proc near
.text:004013DD ; DATA >
.text:004013DD var_4 = dword ptr -4
.text:004013DE argc = dword ptr 8
.text:004013E0 argv = dword ptr 0Ch
.text:004013E4 envp = dword ptr 10h
.text:004013E6
.text:004013E9
.text:004013EE
.text:004013F3
.text:004013F8
.text:004013F9
.text:004013FA
.text:004013FF
.text:00401402
 push ebp
 mov ebp, esp
 push ecx
 push ebx
 push esi
 push edi
 xor eax, eax
 mov [ebp+var_4], eax
 mov ebx, 1
 mov esi, 2
 mov edi, 3
 push esi
 push ebx
 call simple_if
 add esp, 8
 push edi
```
Readability

• Note the improved readability of the code
• The previous name sub_401150 is an example of an Ida default name
  – Not at all descriptive
• When you rename an item, Ida makes the change in all locations that refer to that item
Navigation

• Double click on ‘simple_if’ to jump to the simple_if function
  – Easy navigation reduces the need for search
  – ESC will take you back
    • Careful with ESC, in every window other than the View window, ESC closes the window
    • Recover windows via the View/Open Subviews menu
Renaming Variables

- From the source code we can see that simple_if has two arguments, a and b as well as a local variable result
  - Highlight and press n to rename them
- Ida shows two arguments arg_0 and arg_4, but no local variables
  - Ida default names
    - arg_x an argument x bytes below saved eip
    - var_x a local variable x bytes above saved registers
Renaming a Stack Variable

```
.text:00401150  simple_if   proc near          ; CODE XREF: _main+1D↓p
.text:00401150
.text:00401150  .text:00401150 arg_0     = dword ptr  8
.text:00401150  .text:00401150 arg_4     = dword ptr  0Ch

Please enter a string

Enter stack variable name arg_0

OK  Cancel  Help

.loc 401162: loc_401162:          ; CODE XREF: simple_if+D↑j
.text:00401162
.text:00401162
.text:00401163
.text:00401163
.text:00401163
.text:00401163 simple_if endp
```
After Renaming

Note use of a and b here
Ida Display Elements

View-A

.text:00401150 simple_if proc near
.text:00401150 a = dword ptr 8
.text:00401150 b = dword ptr 0Ch
.text:00401150 push ebp
.text:00401151 mov ebp, esp
.text:00401153 mov ecx, [ebp+b]
.text:00401156 mov edx, [ebp+a]
.text:00401159 xor eax, eax
.text:0040115B cmp ecx, edx
.text:0040115D jge short loc_401162
.text:0040115F lea eax, [ecx+edx]
.text:00401162 loc_401162:
.text:00401162 pop ebp
.text:00401162 push ebp
.text:00401163 simple_if retn
.text:00401163 simple_if endp

section name
virtual addresses
branch indication
stack variables
cross references
Features of Compiled Code
Assembly Constructs

• It is useful to understand what compiled code looks like
• Makes it easier to understand what the source code probably looked like
• Remember, there are always many ways to translate a given sequence of source code into equivalent assembly
Parameter Passing

- Dictated by calling conventions utilized by each function
- Tells you how parameters will be accessed by each function
- Tells you how parameters will be passed to each function
- Tells you whether caller or callee will clean up the stack afterwards
Calling Conventions (i)

- Vary by compiler
  - Visual C++
    - cdecl
      - Push parameters right to left
      - Caller cleans up stack
    - stdcall
      - Push parameters right to left
      - Called function cleans up stack
      - Windows API functions use this calling convention
    -fastcall
      - First two parameters (on the left) go in ECX and EDX
      - Remaining parameters are pushed right to left
  - thiscall
    - For C++ non-static member functions, this is placed in ECX
Calling Conventions (ii)

- gcc
  - Supports cdecl and stdcall
  - cdecl is the default

- g++
  - Pushes "this" as implied first (left most) parameter for non-static member functions

- Others
  - You may see strange things in optimized code
Parameter Access

- Parameters lie beneath return address
  - `call` was last instruction executed prior to function entry
  - Pushes return address
- Parameters accessible at
  - `[esp + 4] ;arg_0`
  - `[esp + 8] ;arg_1`
  - `...`

<table>
<thead>
<tr>
<th>r</th>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg_0</td>
<td>first parameter</td>
</tr>
<tr>
<td>arg_1</td>
<td>second parameter</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Local Variables

• Most functions use local variables
  – Locals are instantiated at time of function call
  – Allocated on the stack upon function entry
    • Explicitly decrement esp to allocate
  – Removed from the stack on function exit
    • Various ways to do this
Local Variable Allocation

```c
void foo(int bar, char *str) {
    int x;
    double y;
    char buf[32];
    //function
}

• This function requires 44 bytes of space for its locals
Local Variable Allocation, asm

```asm
foo:
    sub esp, 44  ; allocate locals
    ; function body
    add esp, 44  ; deallocate locals
    ret
```

• Every function is similar
  – First step - allocate locals
  – Last step – deallocate locals
### Stack View

Stack **frame** for function foo

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>[esp]</td>
<td>buf</td>
<td>32 bytes</td>
</tr>
<tr>
<td>[esp+32]</td>
<td>y</td>
<td>8 bytes</td>
</tr>
<tr>
<td>[esp+40]</td>
<td>x</td>
<td>4 bytes</td>
</tr>
<tr>
<td></td>
<td>return</td>
<td>4 bytes</td>
</tr>
<tr>
<td>[esp+48]</td>
<td>bar</td>
<td>4 bytes</td>
</tr>
<tr>
<td>[esp+52]</td>
<td>str</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>
Stack Frames in Practice

• esp based stack frames are not always practical
• If the function needs to call other functions it must push parameters, altering esp
  – Any change to esp changes the offsets required to access both locals and arguments
• Solution
  – Use a specific register as a fixed "frame pointer"
  – On the x86 this is ebp by convention
Using ebp as a Frame Pointer

• On entry to a function we must "fix" the frame pointer
  – But there is only one ebp and the function that called us is probably already using it

• Two steps
  – Save the old value of ebp
  – Setup ebp as our frame pointer
Prologues & Epilogues

• A function prologue is the code required to setup a frame pointer and allocate local variables

• A function epilogue is the code required to restore the caller’s frame pointer and deallocate local variables
Revised foo

foo:
    push ebp ; save callers frame pointer
    mov ebp, esp ; setup our frame pointer
    sub esp, 44 ; allocate locals

    ; function body

    mov esp, ebp ; deallocate locals
    pop ebp ; restore caller's fp
    ret
Revised Stack View

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ebp-44]</td>
<td>buf</td>
<td>32 bytes</td>
</tr>
<tr>
<td>[ebp-12]</td>
<td>y</td>
<td>8 bytes</td>
</tr>
<tr>
<td>[ebp-4]</td>
<td>x</td>
<td>4 bytes</td>
</tr>
<tr>
<td></td>
<td>old ebp</td>
<td>4 bytes</td>
</tr>
<tr>
<td></td>
<td>return</td>
<td>4 bytes</td>
</tr>
<tr>
<td>[ebp+8]</td>
<td>bar</td>
<td>4 bytes</td>
</tr>
<tr>
<td>[ebp+12]</td>
<td>str</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>
Other Considerations

- Where to expect return values?
  - Generally returned in EAX
  - 64 bit values in EDX:EAX
Ida and Stack Frames

• Ida provides two views of a function’s stack frame
  – Compressed view
    • Ida shows arguments and local variables inline with the function disassembly
  – Expanded view
    • By double clicking on any stack variable, you get an expanded view of the stack for a given function
Example

- In Ida, ESC back to, or otherwise navigate to main
- Double click on ‘argc’ to obtain the expanded stack frame view for main
- Ida determines the runtime layout of each functions stack by analyzing the use of esp and ebp with each function
Stack Frame of main

- **Saved ebp**: The stack frame's base pointer is saved.
- **Saved eip**: The program's instruction pointer is saved.
- **Function arguments**: Arguments passed to the function are on the stack.
- **Local variable**: A variable declared within the function is on the stack.
- **Function**: The code block for the function is shown.
If Statements

• For a simple binary test
  – Compare two values
  – Jump on the inverse of the condition to the first statement beyond the "if body"
    \[
    \text{if} \ (a > b) \ 
    \begin{array}{l}
    \quad \text{...} \\
    \end{array} \\
    \end{array}
    \]
  – Compare \( a \) to \( b \) and jump if \( a \leq b \)
Simple If Statement (example)

• Conditional test and jump

```
cmp eax, ebx ;if
jle endif ;(eax > ebx) {
 ;if body
 ;}
endif:
```
Example

- In Ida, close the stack layout for main by using the ESC key
- Double click on ‘simple_if’ to navigate back to that function
- The disassembled if statement is visible
```assembly
.text:00401150 simple_if proc near ; CODE XREF: _main+1D↓p
.text:00401150 .text:00401150 a = dword ptr 8
.text:00401150 b = dword ptr 0Ch
.text:00401150
* .text:00401150 push ebp
* .text:00401151 mov ebp, esp
* .text:00401153 mov ecx, [ebp+b]
* .text:00401156 mov edx, [ebp+a]
* .text:00401159 xor eax, eax
* .text:0040115B cmp ecx, edx
* .text:0040115D jge short loc_401162
* .text:0040115F lea eax, [ecx+edx]
* .text:00401162 loc_401162: ; CODE XREF: simple_if+D↑j
* .text:00401162 pop ebp
* .text:00401163 retn
* .text:00401163 simple_if endp
```

- Dashed line indicates a conditional branch.
- Solid line indicates an unconditional branch.
- The `if` body is indicated by the line `jge short loc_401162`.
Ida Flowcharting

- For the graphically oriented, Ida also offers some interesting graphing capabilities
- The first that we will look at is flowcharting
- Available for current function only
- Graphs are not interactive
  - That will change in Ida 5.0
Flowchart of simple_if

• Position the cursor on any statement of the simple_if function
• Select
  – View/Graphs/Flowchart, or F12, or
Flowchart Example

• Result is a flowchart that makes it clear that there is some conditionally executed code
  – ESC will close the WinGraph32 window
• This one is easy to interpret because the function is so small
• Complex functions far more difficult
Compound Condition - OR

• For all but the last condition
  – Test and jump if the condition is true to the first statement of the if body
    • i.e. if any part is true proceed to the body
• For the last term in the OR
  – Test and jump if NOT true to the first statement following the if body
• This implements "short circuit" evaluation
Compound OR

cmp eax, ebx ;if
jg body ;(eax > ebx) ||
cmp eax, ecx ;
jnz body ;(eax != ecx) ||
cmp ebx, ecx
jne endif ;(ebx == ecx) {

body:
    ;if body

} endif:
Example

• In Ida, ESC back to or otherwise navigate to main
• The second function main calls is ‘compound_or’
  – Rename it if you like
• Navigate to compound_or
Either test true jumps to body

Both tests false bypasses body

if body
Compound Condition - AND

• For all terms
  – Test for the opposite of the condition and jump to the first statement beyond the if body
    • i.e. if any part is false skip the body

• This implements "short circuit" evaluation
Compound AND

cmp eax, ebx ;if
jle endif ;(eax > ebx) &&
cmp ebx, ecx ;
jle endif ;(ebx > ecx) &&
cmp ecx, edx ;(ecx > edx) {
    body:
    ;if body
    
    endif:
Example

• In Ida, ESC back to main or otherwise navigate to main
• The third function called is ‘compound_and’
  – Rename it if you like
• Navigate to compound_and
Any failure bypasses body → if body
Commenting in Ida

- Ida allows insertion of several different types of comments
- Comments entered by placing the cursor on the line you wish to comment, then selecting a comment type
  - Edit/Comments menu
- Basic comment hot key is colon i.e. Shift-;
Commented compound_and

.text:00401189     compound_and  proc near                    ; CODE XREF: _main+36↓p
.text:00401189
.text:00401189     a                      = dword ptr  8
.text:00401189     b                      = dword ptr 0Ch
.text:00401189     c                      = dword ptr 10h
.text:00401189     d                      = dword ptr 14h
.text:00401189
.text:00401189     push                    ebp
.text:00401189
.text:0040118A     mov         ebp, esp
.text:0040118C     push                    esi
.text:0040118D
.text:0040118E     mov         esi, [ebp+d]
.text:00401191     mov         edx, [ebp+c]
.text:00401194     mov         eax, [ebp+b]
.text:00401197     mov         ebx, [ebp+a]
.text:0040119A
.text:0040119C     xor         ecx, ecx          ; result = 0
.text:0040119E     cmp         eax, ebx          ; if (a > b)
.text:004011A0     jge         short loc_4001AF
.text:004011A2
.text:004011A4     cmp         edx, eax          ; & & (b > c)
.text:004011A6     jge         short loc_4001AF
.text:004011A8
.text:004011A6     lea         ecx, [eax+ebx]   ; result = b + a
.text:004011A8     add         ecx, edx          ; result += c
.text:004011AB     add         ecx, esi          ; result += d
.text:004011AF
.text:004011AF     loc_4001AF:                           ; CODE XREF: compound_and+15↑j
.text:004011AF
.text:004011AF     mov         eax, ecx          ; return result
.text:004011AF
.text:004011B1     pop         esi
.text:004011B2     pop         ebx
.text:004011B3     pop         ebp
.text:004011B4     ret
.text:004011B4     compound_and  endp
If/Else Statement

- All conditional tests that evaluate to false jump to the first statement of the else body
- The last statement of the if body is an unconditional jump past the else body
Simple If/Else Statement (example)

- Conditional test and jump

  ```
  cmp eax, ebx ;if
  jle else ;(eax > ebx) {
    ;if body
  jmp endif ;}
  else: ;else {
    ;else body
  }
endif:
  ```
Example

- Navigate back to main
- The next function called is named if_else
- Navigate to if_else and create a flow chart
  - The if/else structure is clear from the flow chart
  - Executes code in either case
  - Compare this to the graph for simple_if
Loops

- Although x86 offers the `loop` instruction, it is not always practical
  - Only useful if you can test a counter against zero
  - Doesn't work when you want to count up
    - For tests with a fixed start value against a fixed end value, the compiler may be able to compute the count and use the `loop` instruction

```c
for (i = 0; i < 10; i++) {
    // But only if i is not used in the loop body
}```
Loops (cont)

- In high level languages most loops appear to test at the top
  - Conditional jump exits loop when test fails or falls through to continue loop
- End of loop body requires unconditional jump back to top
- Most compilers rearrange loops to contain only a single conditional jump
  - Unconditional jump factored out
While Loop

- Test condition
- Jump if false to first statement following loop body
- Last statement in loop body jumps back to test
While (naïve example)

top:
    cmp eax, ebx        ; while
    jge end_loop       ; (eax < ebx) {

    ; loop body

    jmp top            ;}

end_loop:
While (common example)

cmp eax, ebx ;pretest allows
jge end_loop ;case of 0 passes
top:

;loop body
cmp eax, ebx ;}
jl top ; (eax < ebx);
end_loop:
Example

- Navigate back to main
- The function called after if _else is named while_loop (sub_4011D1)
- Navigate to the while_loop function
- Note the use of heavier lines for backward jumps
  - This is how ida tries to point out a potential loop
Loop Caution

• Don’t assume that a register will contain your loop variable for the duration of a loop
• In a long loop body, the registers involved in the original test may be reused for other purposes.
• Registers need to get reloaded prior to performing loop continuation test
For Loops

- Loop initialization performed immediately prior to the top of the loop
- Counting statements placed at the end of the loop body immediately prior to the jump back to the top
- Test usually takes place at the bottom of the loop
For (example)

```assembly
xor ebx, ebx       ; for (j = 0;
 top:
    cmp ebx, 10    ;
    jge end_loop   ; j < 10;
 ; loop body
    inc ebx        ; j++
    jmp top       ;
end_loop:
```

Alternative For (example)

xor ebx, ebx ; for (j = 0;
jmp test

top:
    ; loop body
inc ebx ; j++)

test:
    cmp ebx, 10 ;
    jl top ; j < 10;

end_loop:
Examples

• The next two functions called from main contain for loops
• The functions are named for_loop and for_loop_down respectively
• In each you can see loop initialization, the testing, and the increment phases
Ida and Strings

• Strings can be very useful in determining the behavior of a binary
  – If nothing else they reveal the use of a char* data type
• When Ida recognizes strings in the data section of a binary, it groups all characters of the string together into a static string variable
Ida String Example

• The function for_loop_down (sub_4011FE) references a string variable
• Note what Ida has done with the string
  – Automatically names the string variable
    • aZZZZZ where ZZZZ are the characters in the string
  – Adds a comment that shows the content of the string
Switch Statements

- Can be done in many ways
- The slowest way
  - A sequence of tests against each case
    - Break statements translate to jumps to first statement after switch
  - If no match found must result in default case or end of switch
- The fastest way
  - Vectored jump based on the switch variable
  - Wastes space if cases are not entirely sequential
Example

- Navigate back to main
- sub_40121D corresponds to switch_small
- Navigate to switch_small
  - Small number of consecutive cases
  - Successive decrement and test
- Take a look at the flowchart
  - Doesn’t necessarily suggest a switch
switch_small proc near ; CODE XREF: _main+63
arg_0 = dword ptr 8
arg_4 = dword ptr 0Ch
arg_8 = dword ptr 10h
arg_C = dword ptr 14h
push ebp
mov ebp, esp
xor eax, eax
mov edx, [ebp+arg_0]
dec edx
jz short case_1
dec edx
jz short case_2
dec edx
jz short case_3
jmp short default_or_end_switch

case_1: ; CODE XREF: switch_small+9
mov eax, [ebp+arg_4]
jmp short default_or_end_switch

case_2: ; CODE XREF: switch_small+1C
mov eax, [ebp+arg_8]
jmp short default_or_end_switch

case_3: ; CODE XREF: switch_small+F
mov eax, [ebp+arg_C]
default_or_end_switch:
pop ebp
ret
switch_small endp
Larger Switches

- Consecutive case handled with jump tables
- Non-consecutive cases handled with subtract and test
  - Subtract smallest constant test for zero
  - Subtract delta to next smallest, test for zero
  - Repeat
Jump Table

- Assume eax holds switch variable which ranges from 0..N

  mov    ebx, jump_table ; address of table
  jmp    [ebx + eax * 4]

- jump_table is the address of the first entry (item 0) in a list of addresses for each case
  – Each address occupies 4 bytes, hence eax * 4
Jump Tables (cont)

- Jump tables can be used for any consecutive range of values, simply normalize to zero
- In this example, the cases run from 32..64

```assembly
mov ebx, jump_table; address of table
sub eax, 32
jmp [ebx + eax * 4]
```
Example

• Navigate to function switch_large (sub_41023F)
• In this case, Ida recognizes the jump tables and labels things accordingly
  – This is Borland code which Ida knows well
• Ida does not always do so well
  – You need to recognize it on your own in those cases
switch_variable_test

Ida recognizes case 1
Switch Weirdness

• Apparent optimization for non-linear cases
  – Successive subtraction
    • Subtract smallest case value
      – If zero, then it’s a match
      – If non-zero, then subtract delta to next smallest and so on
    • If any cases are consecutive, then simply use dec rather than sub
Example

- Navigate to function `switch_gaps` (sub_4102E2)
- In this case, Ida recognizes the consecutive cases and uses a jump table
- Non-consecutive tables handle using subtraction
Reversing Function Calls

- call statement easily recognized in disassembly
- Usually preceded by a series of push operations to get parameters on the stack
  - Sometimes "mov" is used rather than push
    - In this case, space must have been pre-allocated for the parameters
    - Compare with asm_code_samples_gcc.exe
Pushing Parameters - Example

```c
fprintf(stdout, "This program has \%d ...", arg_0);

.text:0804848D push [ebp+arg_0]
.text:08048490 push offset aThisProgramHas
; "This program has \%d command line argument"
...  
.text:08048495 push ds:stdout
.text:0804849B call _fprintf
```
Push via mov Example

```
sub_804844C(getenv("HELLOWORLD"));

.text:080484AE   mov [esp+8+var_8], offset aHelloworld
                 ; "HELLOWORLD"
.text:080484B5   call _getenv
.text:080484BA   mov [esp+8+var_8], eax
.text:080484BD   call sub_804844C
```
Linux System Calls

• Invoked using an `int 0x80`
  – This is a software interrupt
  – Transfers control to the kernel
    • Transitions to kernel stack so we can't pass our parameters on the user stack
      – We could but would need to perform a user to kernel space copy operation
  – Parameters passed in various CPU registers
Linux System Calls (ii)

• There are about 190 different system calls
  – But there is only one `int 0x80`

• Specify which system call you wish to make by placing the syscall number into `eax` before executing `int 0x80`

• Not well documented
  – `http://www.linuxassembly.org/syscall.html`
Linux System Calls (iii)

- Like a function call, each system call expects zero or more parameters
- System calls expect their parameters in very specific registers
Linux System Calls (iv)

• Syscall parameters (if necessary)
  - ebx – first parameter
  - ecx – second parameter
  - edx – third parameter
  - esi – fourth parameter
  - edi – fifth parameter
# Useful System Calls

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>ebx</th>
<th>ecx</th>
<th>edx</th>
</tr>
</thead>
<tbody>
<tr>
<td>sys_exit</td>
<td>1</td>
<td>int retval</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sys_read</td>
<td>3</td>
<td>int fd</td>
<td>char *buf</td>
<td>size_t len</td>
</tr>
<tr>
<td>sys_write</td>
<td>4</td>
<td>int fd</td>
<td>char *buf</td>
<td>size_t len</td>
</tr>
<tr>
<td>sys_open</td>
<td>5</td>
<td>char *name</td>
<td>int flags</td>
<td>int mode</td>
</tr>
<tr>
<td>sys_close</td>
<td>6</td>
<td>int fd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sys_execve</td>
<td>11</td>
<td>char *file</td>
<td>char **argv</td>
<td>char **envp</td>
</tr>
<tr>
<td>sys_socketcall</td>
<td>102</td>
<td>int call</td>
<td>ulong *args</td>
<td></td>
</tr>
</tbody>
</table>
Syscalls and Reverse Engineering

- You will usually only see systems calls in two types of code
  - Shellcode
    - Allow for smallest possible shellcode with no need to link to compiled libraries
  - Statically linked code
    - All library functions linked in with user code to form stand alone executable
    - Makes code independent of installed libraries
Ida Pro

• When analyzing Linux binaries, Ida recognizes the int 0x80 instruction and attempts to comment the preceding instructions based on current value in eax
• Not always possible for Ida to know eax value
Additional Ida Features
Reverse Engineering Goals

• Discover how a program works
  – Build compatible software
• Locate hidden functionality
  – Backdoors etc.
• Search for vulnerabilities in closed source software
• All start with a quality disassembly
  – We will assume Ida is used for this class
Analysis

• Trace code to understand how it works
  – Could generate your own high level code as you go
• Observe/Understand function call tree
• Understand data types
  – Everything looks the same in assembly
    • Is a 4 byte quantity an int, float, or pointer?
    • Depends on how it is used
Analyzing Functions

- **Two approaches**
  - Breadth first
    - Understand a function, then try to understand the functions that are called
  - Depth first
    - Descend into each function as it is called
      - At some point you will get to a function that calls no others or invokes only system/api calls
      - If the former, attempt to figure out what the function does
      - If the later make note of the data passed to the system calls and bubble the types back out toward your initial function
Analyzing Data

• Determining data types used in a program helps determine its functionality

• One of the best ways to determine data types is to look for calls to known functions
  – C standard library calls
  – O/S API calls

• Observe the parameters passed to these functions and name them accordingly
Automated Analysis

• The quality of your disassembler makes a big difference
• IdaPro contains signatures for most of the standard library calls made in C programs
• When Ida sees a call to a known function it annotates your code with known variable type and parameter name information
Ida Pro Strengths

• GUI provides easy navigation and multiple windows of useful info
  – Graphical display of control flow
  – Double click navigation

• Understands many library calls and data types
  – Particularly strong against Windows binaries

• Allows you to annotate your disassemblies
Various Other Windows

• Hex view
  – Raw hex display, tracks along with IDA View

• Segments
  – Breakdown of program segments and virtual addresses assigned to each

• All accessible via View/Open subviews menu item
Ida Subwindows

Open/reopen other windows from the View menu

Windows opened at startup
Revisiting Ida Stack Displays

• Ida only assigns names to locations that are actually referenced in a function
• s and r are Ida standard names for the saved register space and saved return address respectively
• Accounts for every byte on stack
• Data sizes
  – db = byte
  – dw = word = 2 bytes
  – dd = double word = 4 bytes
Stack Based Arrays

- Open demos/proj4 binary
  - Probably need to set file type filter to*.*
  - Note that Ida properly identifies it as an ELF binary
- Code for main begins:

```c
int main(int argc, char **argv, char **envp) {
    unsigned int index;
    char buf[32];

    // 36 bytes of stack locals
```
Example

• Double click on var_38 to bring up the stack frame view
Stack Frame View

• Stack based arrays consume a lot of space in the view
  – Ida often identifies start as dd
  – Many unnamed db lines – why?
• Ida allows you to group consecutive memory locations into arrays
  – Find the start of the array
  – Set the data size (d key toggles between db, dw, dd)
  – Select (Num *) key or click the * tool button to create an array
  – Ida guesses at a proper size
Creating an Array
Creating an Array

• Ida collapses the array variable and all accompanying bytes into a single statement
Collapsing Arrays

• Two notes
  – Compilers often insert padding bytes after arrays
    • Hence the 44 byte array rather than the 32 bytes we asked for
  – The disassembly shows us the exact number of bytes that would be required to overflow the buffer and corrupt other data
Control Flow

• In the left margin, Ida indicates control flow for jumps with arrows/lines showing the direction and target of jumps
  – Conditional jumps – dashed
  – Unconditional jumps – solid
  – Backward jumps – heavier line

• Very useful in identifying branching and looping constructs
Sample (proj3a)

• In this case a loop is shown as flow is backwards

```assembly
.text:000484A3  mov   [ebp+var_4], 0
.text:000484AA  loc_80484AA:  ; CODE XREF: main+77↓j
.text:000484AA  mov   eax, [ebp+var_4]
.text:000484AD  cmp   eax, [ebp+arg_0]
.text:000484B0  jl    short loc_80484B4
.text:000484B2  jmp   short loc_80484DE

.text:000484B4  loc_80484B4:  ; CODE XREF: main+48↑j
.text:000484B4  sub   esp, 4
.text:000484B7  mov   eax, [ebp+var_4]
.text:000484BA  lea   edx, ds:0[eax*4]
.text:000484C1  mov   eax, [ebp+arg_4]
.text:000484C4  push  dword ptr [eax+edx]
.text:000484C7  push  [ebp+var_4]
.text:000484CA  push  offset aArgvDS ; "argv[%d]: %s\n"
.text:000484CF  call  _printf
.text:000484D4  add   esp, 10h
.text:000484D7  lea   eax, [ebp+var_4]
.text:000484DA  inc   dword ptr [eax]
.text:000484DC  jmp   short loc_80484AA

.text:000484DE  ;
.text:000484DE  loc_80484DE:  ; CODE XREF: main+4D↑j
.text:000484DE  sub   esp, 0Ch
```
Data Display

- Ida allows selection of alternate data displays
  - Hex, octal, decimal, binary, ASCII

```
lea    edx, [ebp+var_8]
add    eax, edx
sub    eax, 110h
mov    byte ptr [eax], 25h
lea    eax, [ebp+var_10]
inc    dword ptr [eax]
mov    eax, [ebp+var_10]
lea    edx, [ebp+var_8]
add    eax, edx
sub    eax, 110h
mov    byte ptr [eax], '2'
lea    eax, [ebp+var_10]
inc    dword ptr [eax]
mov    eax, [ebp+var_10]
lea    edx, [ebp+var_8]
add    eax, edx
sub    eax, 110h
```
Changing Data Format

- Right click on data item or choose Edit/Operand Type

```
sub    eax, 110h
mov    byte ptr [eax], 110h
lea    eax, [ebp+var_10]
inc    dword ptr [eax]
mov    eax, [ebp+var_10]
lea    edx, [ebp+var_8]
add    eax, edx
sub    eax, 110h
mov    byte ptr [eax], 110h
lea    eax, [ebp+var_10]
inc    dword ptr [eax]
mov    eax, [ebp+var_10]
lea    edx, [ebp+var_8]
add    eax, edx
sub    eax, 110h
mov    byte ptr [eax], 110h
lea    eax, [ebp+var_10]
inc    dword ptr [eax]
lea    eax, [ebp+var_C]
inc    dword ptr [eax]
jmp    short loc_804837F
```
Ida Cross Referencing

- On initial analysis, Ida creates cross references every chance it gets
- Cross references are displayed as comments in the right margin of the disassembly
- Cross references indicate what other lines of code refer to the current line
  - Very useful for understanding control flow
Ida Graphing

- Cross references form the foundation for a very useful feature of Ida Pro, graphing
- The following graphs can be generated
  - Function flow charts
  - The entire function call tree (forest) for a program
  - All xrefs from a function
    - Who do I call?
  - All xrefs to a function
    - Who calls me?
Flow Chart

- demos/stage4, sub_804844C
- View/Graphs/Flowchart (F12)
Function Call Graph (stage4)
Xrefs From (stage4, sub_804849E)
Graphing Limitations

• Graphs are not interactive
  – Not navigable, collapsible or editable
  – Lose address information
  – Can’t prune

• Often too much information to be useful

• Graphing utility is stand alone app

• No access to generated graph source code or graphing functionality via api
Graphing Improvements

• Third party developers have filled a need
  – Sabre’s BinDiff, BinNavi
  – Pedram Amini’s ProcessStalker

• Ida 5.0 will introduce many new features
  – Integrating graphing
  – Graphing api directly accessible to plugin developers
Ida Comments

• There are several types of comments you can add to a disassembly
  – Access via Edit/Comments menu or hotkeys
  – We have already seen standard comments

• Three additional types
  – Anterior lines
    • Entire preceding line dedicated to comment text
  – Posterior lines
    • Entire succeeding line dedicated to comment text
  – Repeatable comments
Repeatable Comments

• Repeatable comments are repeated at any location that refers to the original commented location
• Entered with ; hotkey
• Useful, for example, when you have commented a data item and you wish to see that comment where the data item is referenced
Commented Code

- Note that Ida uses comments itself to display things like references and function header info
Data Types and Data Structures
Ida Structures

- User defined/complex data type are used frequently in programming
  - C struct for example
- Tough to disassemble because field access is a complex operation in assembly
- Ida allows you to define struct data types and refer to the offsets in your disassembly
Example

• Open demos/fetch
• The call to connect requires a sockaddr_in, so var_28 must be one

```assembly
.push    eax          ; struct in_addr *
.push    offset a205_155_71_181 ; "205.155.71.181"
call    inet_pton
add     esp, 10h
sub     esp, 4
.push    0           ; protocol
.push    1           ; type
.push    2           ; family
call    socket
.add     esp, 10h
.mov     [ebp+var_C], eax
.sub     esp, 4
.push    10h         ; int
.lea     eax, [ebp+var_28]
push    eax          ; int
.push    [ebp+var_C] ; struct sockaddr *
call    connect
.add     esp, 10h
.push    0           ; flags
```
Sidenotes

• Ida 4.9 does a better job of automatically applying type information to disassemblies than previous versions
• The snippet on the previous slide was generated with 4.9
• The same binary loaded in 4.8 will show no type info at all
Structures Window

- Bring up from View/Open Subviews/Structures
- This is where you create and edit structures
Creating a new struct

- Press the Insert key
- Give the structure a name
  - Or add a standard struct
  - For Windows binaries, Ida has a large number of predefined standard structs
  - For Linux/Unix you may need to add a type library
- Add new fields using the d key
- Name the fields using the n key
New Struct
Adding Fields

- Add fields based on what you see or what you know (if you have the source)
Naming Fields

- Name the fields (n key)

I padded the struct to its known size of 16 bytes by adding an 8 byte array
Applying Struct Templates

- In your disassembly, click on the variable that is to become a struct
  - If it is a stack variable, you should be in stack view
- Select the Edit/Struct var…menu option
- Double click on the name of the desired structure
Select Struct Variable
Select Struct

• Note: Ida 4.9 users should redesignate var_28 as a sockaddr_in
Result

```
Stack of main

FFFD堆7    db ? ; undefined
FFFD堆8   var_28  sockaddr_in ?
FFFD堆8    db ? ; undefined
FFFD堆9    db ? ; undefined
FFFD堆A    db ? ; undefined
FFFD堆B    db ? ; undefined
FFFD堆C    db ? ; undefined
FFFD堆D    db ? ; undefined
FFFD堆E    db ? ; undefined
FFFD堆F    db ? ; undefined
FFFD堆0    db ? ; undefined
FFFD堆1    db ? ; undefined
FFFD堆2    db ? ; undefined
FFFD堆3    db ? ; undefined
FFFD堆4   var_C  dd ?
FFFD堆8    db ? ; undefined
FFFD堆9    db ? ; undefined
FFFD堆A    db ? ; undefined
FFFD堆B    db ? ; undefined
FFFD堆C    db ? ; undefined
FFFD堆D    db ? ; undefined
FFFD堆E    db ? ; undefined
FFFD堆F    db ? ; undefined
FFFD堆0    db ? ; undefined
00000000  s     db 4 dup(?)
00000004  r     db 4 dup(?)
00000008 ; end of stack variables

SP+000000230
```
Using Struct Fields

- In your disassembly, struct field names are now available for cleaning up structure member access
- Ida will apply names where it can
- You can right click on constant values to change numbers to a struct field name
Example (fetch)

- Right click on offset to access struct field renaming options

```
push    eax                ; void *
call    _memset
add     esp, 10h
mov     [ebp+var_28.sin_family], 2
sub     esp, 0Ch
push    50h                ; unsigned __int16
call    _htons
add     esp, 10h
mov     [ebp+var_28.sin_port], ax
sub     esp, 8
lea     eax, [ebp+var_28]
add     eax, 4
push    eax                ; struct in_addr *
push    offset a205_155_71_181 ; "205.155.71.181"
call    _inet_pton
add     esp, 10h
sub     esp, 4
push    0                   ; protocol
```
Black Hat Training

Example (fetch)

```
push     3on            ; unsigned __int32
call    _htons
add     esp, 10h
mov     [ebp+var_28.sin_port], ax
sub     esp, 8
lea     eax, [ebp+var_28]
add     eax, sockaddr_in.sin_addr
push    eax             ; struct in_addr *
push    offset a205_155_71_181 ; "205.155.71.181"
call    _inet_aton
add     esp, 10h
sub     esp, 4
push    0               ; protocol
push    1               ; type
push    2               ; family
call    _socket
add     esp, 10h
mov     [ebp+var_C], eax
sub     esp, 4
push    10h             ; int
lea     eax, [ebp+var_28]
push    eax             ; struct sockaddr *
push    [ebp+var_C]     ; int
call    _connect
add     esp, 10h
```
Type Libraries

- Ida offers standard data types when it recognizes the compiler used to create the binary
- For Linux/Unix binaries it often fails to recognize the compiler (does better job in 4.9)
  - Thus no data types are offered
- You can force Ida to show you data types
  - View/Open Subview/Type Libraries
  - Which will get you a warning and an empty window
Type Library Example

• Close the fetch demo, choosing the "DO NOT SAVE" option in the close dialog
• Reopen demos/fetch
• Choose
  – View/Open Subview/Type Libraries
Type Libraries (cont)

- Press the insert key to add a library
  - 4.9 users will see an entry here already
Type Libraries (cont)

- Choose an appropriate library (GNU C++ unix)
Type Libraries (cont)

• Once a type library is selected, Ida will apply function signatures from the library to your disassembly
• Note the change in the disassembly listing (versions < 4.9)
• Try to change the type of var_28 from sockaddr to sockaddr_in
Adding a Standard Struct

- Navigate to the Structures window
- Press the insert key and choose "Add standard structure"
Choosing a Standard Struct

- Scroll to and highlight the sockaddr_in struct, then click OK
Change var_28

- Return to the IDA View window
- Double click on var_28 to get a stack frame view
- Highlight var_28
- Use the Edit/Struct_var menu to change var_28 to a sockaddr_in
Ida Customization Part 1
Basic Configuration

- Ida contains many configuration files in its cfg subdirectory
- Three files of interest
  - ida.cfg
  - idagui.cfg
  - idauser.cfg
  - User specified options (create this yourself)
ida.cfg

- Many parameters to affect basic behavior
  - Whether to create backups
  - Formatting options
  - Default maximum name length
idagui.cfg

• Hotkey assignments
  – Can add or change mappings

• Presence or absence of “Patch” submenu
  – DISPLAY_PATCH_SUBMENU = NO
  – Set to yes for access to patch dialog
    • Allows modification of database bytes
User Defined Macros

• Ida has a built in scripting language called IDC

• Allows scripting of complex actions
  – Virtually anything you can do with hotkeys or menus
  – Cursor control
  – Opening input dialogs

• We will cover IDC later
Running Macros

• Macro options
  – Run once via File/IDC Command
  – Save macro as stand alone file and run via File/IDC File
  – Assign macro to hotkey by editing idc/ida.idc
    • This file is executed at Ida startup
• We will do all of these later
Advanced Binary Analysis
Stripped Binaries

- Contain no symbol table information
- Generally the only names that get recovered are imports
- Look at the difference between demos/proj3a and demos/proj3b for example
Windows

- Windows binaries import a lot of extra stuff
  - Compare the proj3c, "Debug" version to proj3a
  - Compare the proj3d, "Release" version to proj3b
Analyzing Statically Linked Binaries

- Statically linked binaries can be challenging
- No import tables
- Large amounts of code
- Most of it is library code
  - Don't want to reverse known library functions
  - Must recognize them somehow
Statically Linked Binaries

• Linked to library code at build time
  – As opposed to runtime which would be dynamic linking
• Contain no external dependencies
• Usually much larger files
• Much more stuff to sift through
  – See demos/proj3e
Statically Linked, Stripped Binaries

• Biggest hassle to reverse
  – demos/proj3f

• Difficult to tell user code from library code
  – Could look for syscalls and go from there
  – Much more to libraries than just syscalls

• Ida has a tool to help
FLAIR

- Fast Library Acquisition for Identification and Recognition
- Examines a library and creates signatures for each exported function
- Then you can match signatures against functions within a binary
- Not well documented
  - See top level readme and sigmake.txt
FLAIR Installation

• Ida 4.8 users
  – Create a Flair48 subdirectory in your main IdaPro directory
  – Unzip extras/flair48.zip into your newly created subdirectory

• Ida 4.9 users
  – Create a Flair49 subdirectory in your main IdaPro directory
  – Unzip extras/flair49.zip into your newly created subdirectory
FLAIR Demo

• Copy demos/libc_6.a into your Flair4x/bin directory
• Open a command window and cd into the Flair4x/bin directory
• Our demo library is an ELF binary so we will use the pelf tool
Creating Flair Signatures

- `pelf libc_6.a libc_6.pat`
  - Parse the library and create a pattern file
- `sigmake libc_6.pat libc_6.sig`
  - Create signatures from a pattern file, this will yield errors
- Delete the commented lines at the top of the file `libc_6.exc` and rerun `sigmake`
- `sigmake libc_6.pat libc_6.sig`
Applying Flair Signatures

- Close IdaPro
- Copy the file libc_6.sig from the Flair4x/bin directory into your <idabase>/sig directory
- Restart IdaPro
- Open demos/proj3f
- Choose file/Load file/Flirt signature file
  - Choose LIBC_6 "Unnamed sample library"
- Many though not all functions are now recognized
Extending Ida's Capabilities
Ida Scripting

- Scripting in Ida is done using the IDC scripting language
  - C like
  - No data types
  - Declare all variables at beginning of functions
    • No globals
  - Arrays are cumbersome at best, no C style array syntax
IDC Documentation

- Some help actually included with IDA!
- Look for topics
  - "IDC Language"
    - Expressions
    - Statements
    - Variables
    - Functions
  - "Index of IDC Functions"
IDC Variables

- Local variables only
- Declare first in function
  - No initialization with declaration
- Not typed
  - auto is the keyword that introduces a variable declaration
  - Example
    ```
    auto count, index, i;
    ```
- Functions generally expect int, float or string data
IDC Functions

- All are defined with the "static" keyword
- Argument list does not require any type info or the auto keyword
- Return type never specified
  - Just return whatever you want
Example IDC Function

• Example function

```cpp
static demoFunc(arg1, arg2) {
    auto var1;
    var1 = arg1 * arg2;
    return var1;
}
```
IDC Expressions

- Use C style operators except op=
- ints promote to floats as required
- + with strings performs string concatenation
- Comparisons work for string operands

```c
if ("cat" == "dog") {
}
```
IDC Statements

- Most C statements available
  - No switch statement
  - No goto
  - Loops
    - for, while, do all available
    - break and continue available
  - Bracing { } used as in C
Accessing the Ida database

- Data read functions
  - long Byte(long addr);
  - long Word(long addr);
  - long Dword(long addr);
- Read 1, 2, 4 bytes from indicated database location
  - Address should be a virtual address
- Return -1 if address is invalid
  - Outside any defined program section
Modifying an Ida Database

• Data writing functions
  – void PatchByte(long addr, long val);
  – void PatchWord(long addr, long val);
  – void PatchDword(long addr, long val);
  – Change 1, 2, or 4 bytes at the indicated virtual address

• Useful when working with self modifying code
Interactive Scripting

• Interface functions
  – void Message(string format, ...);
    • Print a message to the message area
    • format is printf style format string
  – void Warning(string format, ...);
    • Show a warning dialog box
  – void Fatal(string format, ...);
    • Show a fatal dialog box and quit IDA
Interactive Scripting

- User query functions
  - long AskYN(long default, string prompt);
    - Ask a yes or no question in a dialog box
    - Returns
      - Cancel == -1
      - No == 0
      - Yes == 1
  - string AskStr(string default, string prompt);
    - Ask the user for a string
Interactive Scripting

• File selection dialog
  – string AskFile(bool forsave, string mask, string prompt);
    • forsave – 0 -> open, 1 -> save
    • mask such as "*.*"

• Several other "Ask" function for requesting various data types
Cursor Control

• Read current cursor location
  – long ScreenEA();
    • Returns the virtual address of the cursor location

• Jump display to new location
  – long Jump(long addr);
    • Set cursor to indicated virtual address
Persistant Data

• IDC Arrays
  – The only way to have anything resembling global data
  – long CreateArray(string name);
    • Create a named array, return its "id"
  – void DeleteArray(long id);
    • Clear all elements from and array
  – long SetArrayLong(long id, long idx, long val);
  – long SetArrayString(long id, long idx, string str);
  – string or long GetArrayElement(long tag, long id, long idx);
    • Tag is either AR_LONG or AR_STR
Script Execution

- **File/IDC Command**
  - Type or paste IDC commands into an edit dialog
  - Can execute single statements without need to wrap within a function

- **File/IDC File**
  - Used to execute a stored IDC "program"
    - Program needs a "main" function
Script Execution

- Macro hotkey execution
  - Create function and save in idc/ida.idc
  - Need not be named main (in fact can’t be named main)
  - Use AddHotKey function to map macro to a hot key sequence
    - AddHotkey("Shift-Z", "MyMacro");
    - Add this statement in ida.idc main function
Example IDC Commands

- Open demos/proj3a
- Double click on the string "SECRET="
- Select File/IDC Command…
- Enter the following
  ```c
  auto i, val;
  i = ScreenEA();
  while ((val = Byte(i)) != '=')) {
    PatchByte(i, val + 32);
    i++;
  }
  ```
Stored IDC Programs

- Must have a "main" function
- Stored programs must
  ```
#include <idc.idc>
  ```
- `#define` is understood as well
- `/* ... */` or `//` comments understood
Uses for Scripts

• De-obfuscating obfuscated code
• Finding and labeling uses of insecure functions
• Analyzing stack frames for presence of stack allocated buffers
• Automatically recognize and create data structures
• Infinite possibilities
Example IDC Program

- On your CD
  extras/scripts/n2b_d32.idc
- This script mimics the UPX decompression algorithm to decompress a UPX packed binary
- Also rebuilds import table
Example IDC Program

- Using lida, open demos/proj3_upx.exe
- This is a UPX packed executable
  - It IS NOT hostile, but your AV software might think it is
- Position the cursor at start
- Select File/IDC File…
- Open extras/scripts/n2b_d32.idc
- Click through any warnings
- Notice the appearance of many more Names in the Names window
- Right click in the Strings window and choose setup, then Ok
IDC Programs

• Once you run an IDC program a small "recent IDC scripts" window will appear
• Click on the sheet of paper to edit a script in notepad or the gear to run the script
  – Open n2b_d32.idc in notepad to view the script
Advanced Scripting
IDC Iterator Functions

- IDC offers iterator functions
  - Iterate through code xrefs
    - Rfirst, Rnext, RfirstB, RnextB
  - Iterate through data xrefs
    - Dfirst, Dnext, DfirstB, DnextB
  - Iterate through segments
    - FirstSeg, NextSeg
  - Iterate through functions
    - NextFunction
IDAPython

- Author: Gergely Erdélyi
- Allows scripts to be authored in Python
- Scripts have access to full IDA API as well as full Python API
- http://d-dome.net/idapython/
IDA Plugins

- Integrate directly into IDA
  - Essentially a dll that IDA automatically loads
  - Loaded from <ida dir>/plugins when IDA starts

- Compiled C/C++
  - Can access IDA api
  - Can access Windows API
  - Samples provided as Visual C++ projects or gcc makefile
IDA Plugins

• IDA SDK is required to build plugins
• Essentially no documentation
  – SDK is not supported by DataRescue
• Best, though not great, source of info are the hpp header files in `<sdkdir>/include`
  – All plugin files should #include `<ida.hpp>`
Plugin Writers Guide

- Author: Steve Micallef
- Included on CD
  - docs/ida_plugin_writing.pdf
- Online version at
  - http://www.binarypool.com/idapuginwriting/
- Hyperlinked version at
Plugin Architecture

• All plugins need an init function
  – Called by IDA at startup
  – Instructs IDA whether to load the plugin or not

• Plugin exports: plugin_t PLUGIN
  – Struct that describes various plugin options including
    • Name of the init function
    • Name of the term(inate) function
    • Name of the run function
    • Desired hotkey to activate the plugin
Plugin Architecture

- Termination function is called when IDA is closing to offer plugin a chance to cleanup after itself
- Run is called by IDA whenever user enters hotkey sequence
  - Can do just about anything you want here
Basic Plugin

- Distributed with SDK
- In `<sdkdir>/plugins/vcsample`
- Demonstrates some basic plugin concepts
IDA API

- C functions offered that do almost all of the things you can do in the IDC language
  - Unfortunately function names are not always the same
  - Can interact with status window or open basic dialog boxes
- Significantly more functions available for lower level interaction with IDA database
Plugin Demo

• x86 emulator plugin
• untar extras/ida-x86emu-0.9.tgz into <sdkdir>/plugins
• Shutdown IDA, DO NOT SAVE your proj3_upx.exe work
Build w/ Visual C++ 6.0

- Using MSCV++, open
  `<sdkdir>/plugins/ida-x86emu/x86Emulator.dsw`
- Choose Build/build x86emu.plw
- Copy
  `<sdkdir>/plugins/ida-x86emu/Debug/x86emu.plw`
  To
  `<idadir>/plugins`
Build w/ cygwin

- Open cygwin terminal
- cd to `<sdkdir>/plugins/ida-x86emu/
- make –f makefile.gcc
- cd to `<sdkdir>/plugins/bin
- Copy
  `<sdkdir>/plugins/bin/x86emu.plw
  To
  `<idadir>/plugins
Plugin Demo

• Restart IDA
• IDA should load the plugin automatically
• Reopen proj3_upx.exe
• Position the cursor at start
• Type Alt-F8
  – Which happens to be the hot key sequence for the x86emu plugin
X86 Emulator Plugin

- Provides a virtual CPU
- Allows emulated execution of instructions
- Uses the IDA database as its RAM
  - Provides its own heap and stack
- Fetches instructions from the IDA database and executes them
  - If an instruction modifies other instructions, then the plugin updates the IDA database accordingly
X86 Emulator Plugin

• Every time an instruction is fetched, the plugin tells IDA to turn that location into code
  – Even if IDA previously thought it was data
  – May require undefining existing instructions
• Useful for working through self modifying code
• Custom dialog boxes can be used in plugins because full Windows API is available
Collaborative Reversing

- Ida-sync plugin allows multiple users to share work on a single binary
- Client/server architecture
- Server - Python based server
  - Stores user, database, and database change records on central server
- Client – Ida plugin
  - forwards some user actions to server for distribution to other clients
Vulnerability Scanning

- Halvar Flake's BugScam
  - Set of IDC scripts
  - Iterates through calls to unsafe functions
  - Analyzes arguments to each call for possible unsafe use
  - Generates html reports pointing to possible problems
  - http://sourceforge.net/projects/bugscam
Vulnerability Discovery with Ida Pro
Vulnerability Discovery

- Ida does not automate the vulnerability discovery process
- Its capabilities may make the process easier
Stack Analysis

- Accurate stack display
  - Required for determining proper placement in return address in exploit buffer
  - Clear picture of what variables may get clobbered during an overflow
- Is there buffer in this stack frame?
- What is the exact distance from the buffer start to overwrite the saved eip?
- What variables lie between the buffer and eip?
Function Xrefs

- Cross reference lists
  - Clean display of all calls to specified functions
- Xrefs To
  - What are possible execution paths to arrive at a specific location
- Xrefs From
  - Where might this data get passed
Virtual Address Layout

• Ida acts like a loader when it analyzes a binary for the first time
• Maps the binary to virtual addresses just as actual loaders do
• Easy to determine useful address when write anywhere vulnerabilities are discovered
GOT Layout

• .got is just another section to ida and easy to view
Binary Patching
Why Patch

- Add/Delete/Modify existing behavior
  - Fix vulnerabilities in closed source binary
  - Bypass existing behavior
    - Common among crackers
  - Customize strings
    - Hex editor may be just as easy in this case
Ida Patching Features

• Patch submenu
  – Enabled by editing cfg/idagui.cfg
    • DISPLAY_PATCH_SUBMENU = YES

• Produce file options
  – File/Produce File submenu
    • Looks promising
      – Especially “Create EXE file …”
        » Not supported for most formats
    • “Create DIF file …” is best option
      – Non-standard diff format
Patch Submenu
Using the Patch Submenu

• Change byte and change word are just shortcuts to idc PatchByte and PatchWord functionality
  – Opens dialog to changes values starting at cursor address

• Assemble
  – Opens dialog to enter new instruction at cursor location
Assemble Dialog

- Replaces cursor instruction with user specified instruction
  - Users responsibility to make sure instruction alignment is maintained
Instruction Alignment

- nop below only takes one byte
  - Bytes a 08048496-A remain unchanged
Ida Dif Files

- Most practical way to export changes
- Only output changes made via PatchByte/Word/Dword
- Simple text file
  - Must apply changes to transform original binary
Example Ida Dif File

This difference file is created by The Interactive Disassembler

proj3a
0000005C0: 53 73
0000005C1: 45 65
0000005C2: 43 63
0000005C3: 52 72
0000005C4: 45 65
0000005C5: 54 74
Patching Challenges

• Changing a few bytes is relatively simple
• Careful when changing any relative offset
  – Make sure you compute correct new offset
• Adding code is more challenging
  – Tough to change function calls
    • Must already link to desired function
    • Need space for code to push additional parameters
Adding Code to a Binary

• Can’t simply insert new code
  – Impact on binary file header values
• Moving code changes relative/absolute offsets
  – Must propagate changes through entire binary
• Best option is to patch into available holes
Binary “Caves”

- Requires detailed understanding of binary format
- Binary sections often have alignment requirements
- Subsequent section must begin with specific alignment
- May offer “slack space” opportunities at end of each section
  - Size on disk vs. size in memory
• Sections align to 256 byte boundaries

Can fit 224 bytes in here, but must adjust .data and .bss pointers
Analyzing Obfuscated Code
Background

- What is obfuscated code?
  - Program transformation to reduce "readability"
    - Performed at source or binary level
    - This talk deals with binary obfuscation
  - Preserves original behavior of program

- Why obfuscate code?
  - Protect intellectual property
  - Hide malicious intent
Background

• Why analyze obfuscated code?
  – To understand functionality in order to interoperate
  – To access malicious program within for further analysis
  – To understand state of the art in code obfuscation
Obfuscation Basics

- Program written and tested using standard methods
- Compiled program is fed to an obfuscator
- Obfuscator typically "encrypts" the original program
- Obfuscator combines encrypted data block with a "decryption" stub to create a new executable
Obfuscation Basics

- Program entry point changed to point to decryption stub
- Decryption stub executes and decrypts original binary
- Once decrypted, stub transfers control to original entry point and original binary executes
Simple Obfuscation

Entry point

Header
Code
Data

Header
Obfuscated Code
Obfuscated Data
De-obfuscation stub
Types of Analysis

• Black Box/Dynamic
  – Observe the behavior of the program in an instrumented environment
  – Difficult to test all code paths

• White Box/Static
  – Deduce behavior by analyzing the code
  – Requires high quality disassembly

• Hybrid/Gray Box
Anti-Reverse Engineering

- **Anti-disassembly**
  - Efforts to prevent proper disassembly
    - Encrypted code
    - Jumps to middle of instructions
      - Violates assumption of sequential execution

- **Anti-debugging**
  - Debugger detection
  - Timing checks
  - Self-debugging
  - Virtual machine environment checks
Anti-Reverse Engineering

- **Anti-Analysis**
  - Intentional exceptions to modify execution path
  - On demand decryption of code blocks
    - Entire executable is never decrypted at once
    - Defeats memory snap-shotting
  - Instruction replacement/emulation
    - Instructions replaced with software interrupt
    - Interrupt handler does table lookup and emulates the instruction
Analysis Techniques

- Generally running malicious code is a bad thing
- Static analysis requires a high quality disassembly
Obfuscated Code Analysis

- Hand tracing assembly language is tedious and error prone
- Anti-reverse engineering techniques obfuscate code paths
- Obfuscated binaries require de-obfuscation before their code can be analyzed
Obfuscated Code Analysis

• The challenge in static analysis is to get at the obfuscated code
• Essentially need to perform the function of the de-obfuscation stub
• Requires running the code
  – By hand
  – Debugger
  – Emulation
Dynamic De-obfuscation

• First step – understand de-obfuscation algorithm
• Second step – mimic the algorithm
  – Can be scripted in IDA
    • Requires unique script for each de-obfuscation technique
  – Alternatively mimic the CPU
    • Add an execution engine to IDA
One Method

• x86 emulator plugin for IDA
• Lightweight emulator
  – Maintains CPU state
  – 'Fetches' instructions by querying IDA database
  – Emulates the instruction
  – Updates IDA database if required
    • Self modifying code for example
Emulator Console
Results

• No need to develop scripts or even perform detailed analysis of de-obfuscation layer
  – The emulator is the script
• Allows safe, automated unpacking/decrypting of "protected" binaries
  – UPX, burneye, shiva, tElock, ASPack, …
Emulator Features

• Similar to a debugger in many ways
• IDA database serves as instruction and static data memory space
• Emulator supplies its own stack space
• Emulator supplies its own heap
  – Redirect library functions to plugin provided equivalents
Emulator Memory

- Code and static data must be fetched from IDA database
- Other references must be directed to either stack or heap
  - Every memory reference checked
  - Could easily add comprehensive memory usage analysis
Limitations

• Slow
  – Because of emulated execution and IDA interactions

• Instruction set emulator only
  – Not an O/S emulator
  – Can't follow calls into dynamically linked functions
  – Can't follow system calls in statically linked functions
O/S Interface Issues

• Generally need to provide some basic services to the de-obfuscation routine
  – Memory allocation
  – Exception handling
  – Linking services

• Minimal set of functions provided by the plugin
  – Heap management
  – Windows Exception Frames
Morphine Demo

• Morphine is an obfuscator used on some windows rootkits
• Available in demos/rootkit/avg.exe
  – Load into IDA
  – Use emulator to unpack and extract
Contact Info

- Chris Eagle
  - cseagle@redshift.com
Resources

- **IDA Downloads**
  - [http://www.datarescue.com/idabase/idadown.htm](http://www.datarescue.com/idabase/idadown.htm)
  - Halvar Flake's structure reconstructor

- **Interesting IDC scripts**
  - Halvar Flake's script based "security scanner"
    - [http://sourceforge.net/projects/bugscam](http://sourceforge.net/projects/bugscam)
      - Scans for use of strcpy, printf, etc

- **x86 Emulator plugin**
  - [http://sourceforge.net/projects/ida-x86emu](http://sourceforge.net/projects/ida-x86emu)
References

- Pentium reference manuals
  - [http://developer.intel.com/design/Pentium4/documentation.htm#manuals](http://developer.intel.com/design/Pentium4/documentation.htm#manuals)
- Others on CD in docs directory
  - File format references
- API references are always handy
  - MSDN