Reverse Engineering

Mitchell Adair
1/22/2013
Know Owen from our time at Sandia National Labs
Currently work for Raytheon
Founded UTDallas's Computer Security Group (CSG) in Spring 2010
Reversing, binary auditing, fuzzing, exploit dev, pen testing...
Python
At the end of this, you should feel comfortable

- Being handed a binary
- Examining a binary's sections, imports, strings
- Renaming and simplifying the disassembly
- Converting from assembly to source, where needed
- Understanding process memory layout
- Figuring out function arguments and local variables
  - How many and what types
- Using a debugger to fill in the gaps or manipulate program execution
Outline

- Static vs Dynamic (overview)
- PE and ELF
- Assembly
- Registers
- The Stack
- Functions
- IDA
- Debugging
- Note on Bytecode
- Conclusion
Static
- Looking at the code, figure things out
- It’s all there, but possibly more complicated
- A safer approach
  - Not running the code!

Dynamic
- Examine the process during execution
- Can see the values in real time
  - Registers, memory contents, etc.
- Allows manipulation of the process
- Should run in a VM!
Disassemblers are usually the tool of choice for static
  o IDA Pro, objdump, etc.

Debuggers are used for dynamic analysis
  o Windows
    • WinDBG, Immunity, OllyDBG, IDA
  o Linux
    • GDB
A good disassembler will have several useful features
  - Commenting
  - Renaming variables
  - Changing function prototypes
  - Coloring, grouping and renaming nodes (IDA)
  - ...

A good debugger will have several useful features
  - Set breakpoints
  - Step into / over
  - Show loaded modules, SEH chain, etc.
  - Memory searching
  - ...

Static vs Dynamic - Tools
Okay, no more!

We’ll be going into each of these heavily.

That was just a high level overview to understand

- The difference between static and dynamic analysis
- The general approach taken between the two
PE and ELF
PE (Portable Executable)
- “File format for executables, object code and DLLs, used in 32-bit and 64-bit versions of Windows operating systems” – *wikipedia*

ELF (Executable and Linkable Format)
- “A common standard file format for executables, object code, shared libraries, and core dumps” – *wikipedia*
- Linux, Unix, Apple OS
PE and ELF

ELF File Format

ELF Header
Program Header Table
Section 1
Section 2
...
Section n
Section Header Table (Optional)

PE File Format

MZ - DOS Header
PE Signature
Image File Header
Section Table (Image Section Headers)
Sections 1-n
COFF Debug Sections
We could go very, very deep into file formats... but let’s not

Each format is just a big collection of fields and sections

Fields will have a particular meaning and hold a particular value

- Date created, last modified, number of sections, image base, etc.

A section is, generally, a logical collection of code or data

- Has permissions (read/write/execute)
- Has a name (.text, .bss, etc.)
Okay, so what? Why is this useful?

Can get an overview of what the binary is doing
  o Can look at what libraries the binary is loading
  o Can look at what functions are used in a library
    • Find vulns
  o Can parse data sections for strings
    • Very helpful on CTFs
  o Can help determine if a binary is packed
    • Weird section names or sizes, lack of strings, lack of imports

How do we analyze them?
  o PE : CFF Explorer, IDA, pefile (python library), ...
  o ELF : readelf, objdump, file, ...
This is CFF Explorer looking at calc.exe’s sections headers.

<table>
<thead>
<tr>
<th>Name</th>
<th>Virtual Size</th>
<th>Virtual Ad...</th>
<th>Raw Size</th>
<th>Raw Address</th>
<th>Reloc Address</th>
<th>Linenumbers</th>
<th>Relocations N...</th>
<th>Linenumbers ...</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte[8]</td>
<td>Dword</td>
<td>Dword</td>
<td>Dword</td>
<td>Dword</td>
<td>Dword</td>
<td>Dword</td>
<td>Word</td>
<td>Word</td>
<td>Dword</td>
</tr>
<tr>
<td>.text</td>
<td>00060CC9</td>
<td>00001000</td>
<td>00060E00</td>
<td>00000600</td>
<td>00000000</td>
<td>00000000</td>
<td>0000</td>
<td>0000</td>
<td>60000020</td>
</tr>
<tr>
<td>.rdata</td>
<td>00010EC4</td>
<td>00006200</td>
<td>00011000</td>
<td>00061400</td>
<td>00000000</td>
<td>00000000</td>
<td>0000</td>
<td>0000</td>
<td>40000040</td>
</tr>
<tr>
<td>.data</td>
<td>00004E80</td>
<td>00007300</td>
<td>00004E00</td>
<td>00000400</td>
<td>00000000</td>
<td>00000000</td>
<td>0000</td>
<td>0000</td>
<td>40000040</td>
</tr>
<tr>
<td>.pdata</td>
<td>000064A4</td>
<td>00007800</td>
<td>00006600</td>
<td>00007720</td>
<td>00000000</td>
<td>00000000</td>
<td>0000</td>
<td>0000</td>
<td>40000040</td>
</tr>
<tr>
<td>.rsrc</td>
<td>00062798</td>
<td>00007F00</td>
<td>00062800</td>
<td>0007D800</td>
<td>00000000</td>
<td>00000000</td>
<td>0000</td>
<td>0000</td>
<td>40000040</td>
</tr>
<tr>
<td>reloc</td>
<td>0000037C</td>
<td>000E2000</td>
<td>00000400</td>
<td>000E0000</td>
<td>00000000</td>
<td>00000000</td>
<td>0000</td>
<td>0000</td>
<td>42000040</td>
</tr>
</tbody>
</table>

Represent permissions.
This is CFF Explorer looking at a UPX packed executable from a recent CTF.

Huge red flag with section names like this.
This is using `readelf` to look at section headers

```
$ readelf -S a.out
```

There are 8 section headers, starting at offset 0x70:

<table>
<thead>
<tr>
<th>Nr</th>
<th>Name</th>
<th>Type</th>
<th>Addr</th>
<th>Off</th>
<th>Size</th>
<th>ES</th>
<th>Flg</th>
<th>Lk</th>
<th>Inf</th>
<th>Al</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>NULL</td>
<td>00000000</td>
<td>000000</td>
<td>000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>text</td>
<td>PROGBITS</td>
<td>00000000</td>
<td>000034</td>
<td>0000a</td>
<td>0</td>
<td>AX</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>.rel.text</td>
<td>REL</td>
<td>00000000</td>
<td>000208</td>
<td>000008</td>
<td>08</td>
<td>6</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>.data</td>
<td>PROGBITS</td>
<td>00000000</td>
<td>000040</td>
<td>000000</td>
<td>0</td>
<td>WA</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>.bss</td>
<td>NOBITS</td>
<td>00000000</td>
<td>000040</td>
<td>000000</td>
<td>0</td>
<td>WA</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>.shstrtab</td>
<td>STRTAB</td>
<td>00000000</td>
<td>000040</td>
<td>000030</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>.symtab</td>
<td>SYMTAB</td>
<td>00000000</td>
<td>0001b0</td>
<td>000050</td>
<td>10</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>.strtab</td>
<td>STRTAB</td>
<td>00000000</td>
<td>000200</td>
<td>000005</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Key to Flags:
- W (write), A (alloc), X (execute), M (merge), S (strings)
- I (info), L (link order), G (group), x (unknown)
- O (extra OS processing required) o (OS specific), p (processor specific)
This is IDA examining what functions are imported

I have filtered using the regular expression .*str.*

<table>
<thead>
<tr>
<th>Address</th>
<th>Function Name</th>
<th>Import Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>011CC54D8</td>
<td>FreeEnvironmentStringsA</td>
<td>KERNEL32</td>
</tr>
<tr>
<td>011CC550</td>
<td>IsBadStringPtrA</td>
<td>KERNEL32</td>
</tr>
<tr>
<td>011CC554</td>
<td>IsBadStringPtrW</td>
<td>KERNEL32</td>
</tr>
<tr>
<td>011CC558</td>
<td></td>
<td>KERNEL32</td>
</tr>
<tr>
<td>011CC564</td>
<td></td>
<td>KERNEL32</td>
</tr>
<tr>
<td>011CC56C</td>
<td></td>
<td>KERNEL32</td>
</tr>
<tr>
<td>011CC57C</td>
<td></td>
<td>KERNEL32</td>
</tr>
<tr>
<td>011CC598</td>
<td></td>
<td>KERNEL32</td>
</tr>
<tr>
<td>011CC5A0</td>
<td></td>
<td>KERNEL32</td>
</tr>
<tr>
<td>011CC5C0</td>
<td></td>
<td>KERNEL32</td>
</tr>
<tr>
<td>011CC5C4</td>
<td></td>
<td>KERNEL32</td>
</tr>
<tr>
<td>011CC5D4</td>
<td></td>
<td>KERNEL32</td>
</tr>
<tr>
<td>011CC644</td>
<td></td>
<td>KERNEL32</td>
</tr>
<tr>
<td>011CC674</td>
<td></td>
<td>KERNEL32</td>
</tr>
<tr>
<td>011CC6A0</td>
<td></td>
<td>KERNEL32</td>
</tr>
<tr>
<td>011CC6B4</td>
<td></td>
<td>KERNEL32</td>
</tr>
<tr>
<td>011CC714</td>
<td></td>
<td>KERNEL32</td>
</tr>
<tr>
<td>011CC724</td>
<td></td>
<td>KERNEL32</td>
</tr>
<tr>
<td>011CC840</td>
<td></td>
<td>OLEAUT32</td>
</tr>
<tr>
<td>011CC844</td>
<td></td>
<td>OLEAUT32</td>
</tr>
<tr>
<td></td>
<td>IstrncpyA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IstrncpyW</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IstrncpiA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IstrncmpW</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IstrncpiW</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetStringTypeExW</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IstrncmpA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IstrlenA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IstrcatW</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetProfileStringW</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WritePrivateProfileStringW</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IstrncpynW</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetPrivateProfileStringW</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IstrlenW</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IstrcatW</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OutputDebugStringW</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Probably worth investigating ;)
This is IDA examining strings it has found for a recent CTF problem

Probably want to start from the “Get your flag %s\n” string and work backwards ;)

<table>
<thead>
<tr>
<th>Address</th>
<th>Length</th>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>.rdata:004020D6 00000004</td>
<td>unico... @</td>
<td></td>
<td>HoppaKey</td>
</tr>
<tr>
<td>.rdata:004020E6 00000004</td>
<td>unico... @</td>
<td></td>
<td>Ups, some calls are wrong or missing =|</td>
</tr>
<tr>
<td>.rdata:0040210C 00000009</td>
<td>C</td>
<td></td>
<td>Get your flag %s\n</td>
</tr>
<tr>
<td>.rdata:00402118 00000028</td>
<td>C</td>
<td></td>
<td>load_me</td>
</tr>
<tr>
<td>.rdata:00402140 00000012</td>
<td>C</td>
<td></td>
<td>Kernel32.dll</td>
</tr>
<tr>
<td>.rdata:00402154 00000008</td>
<td>C</td>
<td></td>
<td>LoadLibraryA</td>
</tr>
<tr>
<td>.rdata:0040215C 000000D</td>
<td>C</td>
<td></td>
<td>GetProcAddress</td>
</tr>
<tr>
<td>.rdata:0040216C 000000D</td>
<td>C</td>
<td></td>
<td>KERNEL32.DLL</td>
</tr>
<tr>
<td>.rdata:0040217C 000000F</td>
<td>C</td>
<td></td>
<td>MSVCR90.dll</td>
</tr>
</tbody>
</table>
Open number_checker.exe and number_checker_packed.exe

Compare these two!

In CFF Explorer
  - Look at different fields in the PE format
  - Look at sections
  - Just explore

In IDA
  - Look at strings (shift+f12)
  - Look at imports (view->open subviews->imports)
  - Look at sections (shift+f7)
Assembly
Two syntax options

- ATT
  - instruction source, dest
  - mov %eax, %edx
  - “Move eax into edx”

- Intel
  - instruction dest, source
  - mov edx, eax
  - “Move into edx, eax”
It’s a known fact that Intel’s syntax > ATT’s, so we’ll be using Intels ;)

**mov eax, ecx**
- Move into eax, the contents of ecx

**mov eax, [ecx]**
- Move into eax, the contents of what ecx **points to**
- The brackets, [...], mean dereference the value between them
- In C, this is like a pointer dereference
- eax = *ecx
Memory values and immediates can be used as well

**mov eax, 5**
- Move into eax, the value 5

**mov edx, [0x12345678]**
- Move into edx, what 0x12345678 points to
Assembly

- A very small handful of instructions will get you a long way
  - call, mov, cmp, jmp
- call 0x12345678
  - Call the function at 0x12345678
- cmp eax, 8
  - Compare eax to 8
  - Compare left to right
- jmp 0x12345678
  - Unconditional jump to 0x12345678
- jle 0x12345678
  - Jump to 0x12345678 if eax is less than or equal to 8
- jg 0x12345678
  - Jump to 0x112345678 if eax is greater than 8
Assembly — Example

080483b4 <main>:
80483b4:      55          push    ebp
80483b5:      89 e5       mov      ebp,esp
80483b7:      83 ec 10    sub      esp,0x10
80483ba:      c7 45 fc 04 00 00 00 mov    DWORD PTR [ebp-0x4],0x4
80483c1:      c7 45 f8 0a 00 00 00 mov    DWORD PTR [ebp-0x8],0xa
80483c8:      8b 45 fc     mov      eax,DWORD PTR [ebp-0x4]
80483cb:      3b 45 f8     cmp      eax,DWORD PTR [ebp-0x8]
80483ce:      7d 07       jge      80483d7 <main+0x23>
80483d0:      b8 01 00 00 00 mov      eax,0x1
80483d5:      eb 05       jmp      80483dc <main+0x28>
80483d7:      b8 00 00 00 00 mov      eax,0x0
80483dc:      c9          leave
80483dd:      c3          ret
Let’s focus on the instructions we know

- mov, cmp, jmp, call
- \([\text{ebp}-\text{0x4}] = \text{0x4}\)
- \([\text{ebp}-\text{0x8}] = \text{0xa}\)
- \(\text{eax} = [\text{ebp}-\text{0x4}]\)

- Two values, relative to the pointer contained in ebp have been assigned values
- One register has been assigned a value
[ebp-0x4] = 0x4
[ebp-0x8] = 0xa
eax = [ebp-0x4]
cmp eax, [ebp-0x8]
  o  eax == [ebp-0x8] ?
  o  4 == 10 ?
jge 0x80483d7
  o  If 4 was >= 10, jmp
  o  Else, continue execution
[ebp-0x4] = 0x4
[ebp-0x8] = 0xa
eax = [ebp-0x4]
cmp eax, [ebp-0x8]
  o eax == [ebp-0x8]?
  o 4 == 10?
jge 0x80483d7
  o If 4 was >= 10, jmp
  o Else, continue execution

False, so execution just continues to the next instruction
[ebp-0x4] = 0x4
[ebp-0x8] = 0xa
eax = [ebp-0x4]
cmp eax, [ebp-0x8]
jge 0x80483d7
mov eax, 0x1
  o eax = 1
jmp over the mov eax, 0
leave and return
So two memory addresses, relative to the pointer contained in ebp, have values. One has 4, one has 10.

There is a comparison

If operand 1 >= operand 2, take the jump

If not, continue execution

Eax gets assigned the value of 1

The function returns
Let’s dig deeper

Everything shown in the disassembly has a purpose

mov DWORD PTR [ebp-0x4], 0x4
  o What does DWORD PTR mean?

We know the brackets [...] mean get the value held at the dereferenced value between them... but DWORD PTR?
mov DWORD PTR [ebp-0x4], 0x4

DWORD PTR
  o DWORD = the size
  o PTR = dereference the value, accompanied by the brackets

We have a few number of sizes allowed
# Example 1 — Types and Sizes

<table>
<thead>
<tr>
<th>Type</th>
<th>Size (bytes)</th>
<th>Size (bits)</th>
<th>ASM</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 byte</td>
<td>8 bits</td>
<td>BYTE</td>
<td>char c;</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>16 bits</td>
<td>WORD</td>
<td>short s;</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
<td>32 bits</td>
<td>DWORD</td>
<td>int i;</td>
</tr>
<tr>
<td>long long</td>
<td>8 bytes</td>
<td>64 bits</td>
<td>QWORD</td>
<td>long long l;</td>
</tr>
</tbody>
</table>
So...

`mov DWORD PTR [ebp-0x4], 0x4`

The address pointed to by the dereferenced value of `[ebp-4]` is getting 4 bytes moved into it, with the value of 4.

`[ebp-4]` is an int

So our source code probably has some int value and hard codes a value of 4 to it
Example 1

- `mov DWORD PTR [ebp-0x4], 0x4`
- `mov DWORD PTR [ebp-0x8], 0xa`
- This leaves us with 2 ints being assigned a hard coded value
  - `int x = 4;`
  - `int y = 10;`
- Are these locals, globals, static variables???
- We need a little background on process memory layout.


```plaintext
int x = 4;
int y = 10;
  o  We don’t know where these are declared
if (4 >= 10)
  o  jmp to main+0x23
eax = 1
jmp to main+0x28
main+0x23:
  o  eax = 0
main+0x28:
  o  ret

We don’t take the jmp as already discussed.
It’s starting to look like source code!
```
Let’s do a quick introduction to process memory layout, then we’ll continue with the first example

We want to know

- Why things are relative to esp/ebp?
- What are the push/pop instructions doing?
- What about the leave/ret instructions?
Process Memory Layout - Windows

- **Stack**: When something is pushed onto the stack, the stack pointer decrements. Stack size: fixed allocation.

- **Heap**: Heap grows down (to higher address).

- **Program Image**:
  - PE header
  - .text (code)
  - .rdata (imports)
  - .data (data)
  - .rsrc (resources)

- **Memory space**: Can be allocated as heap, or stack for other threads in the process, etc.

- **DLL**
  - DLL's have a header, .text, .data, .rsrc and .reloc segment

- **PEB (data block of main thread)**

- **Shared user page**

- **No access**

- **No access**
<table>
<thead>
<tr>
<th>Higher memory address</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
</tr>
<tr>
<td><strong>Stack</strong></td>
</tr>
<tr>
<td>env argv argc</td>
</tr>
<tr>
<td>Auto variables for main()</td>
</tr>
<tr>
<td>Auto variable for func()</td>
</tr>
<tr>
<td>Available for stack growth</td>
</tr>
<tr>
<td><strong>Shared libraries</strong></td>
</tr>
<tr>
<td>malloc.o (lib*.so)</td>
</tr>
<tr>
<td>printf.o (lib*.so)</td>
</tr>
<tr>
<td>Available for heap growth</td>
</tr>
<tr>
<td><strong>data</strong></td>
</tr>
<tr>
<td>Heap (malloc(), calloc(), new)</td>
</tr>
<tr>
<td>Global variables</td>
</tr>
<tr>
<td>int y = 100;</td>
</tr>
<tr>
<td><strong>Text (Compiled code, a.out)</strong></td>
</tr>
<tr>
<td>malloc.o (lib*.a)</td>
</tr>
<tr>
<td>printf.o (lib*.a)</td>
</tr>
<tr>
<td>file.o</td>
</tr>
<tr>
<td>main.o</td>
</tr>
<tr>
<td>func()</td>
</tr>
<tr>
<td><strong>Lower memory address</strong></td>
</tr>
</tbody>
</table>

- `main()` frame pointer (EBP)
- Stack pointer (ESP), points at the top of the stack - grows downward
- Library functions if dynamically linked - usual case
- `brk()` point
- Uninitialized data - bss
- Initialized data - data
- Library functions if statically linked - not usual case
- The return address
Virtual Memory

Low

Text
Code segment, machine instr.

Data
Initialized global and static variables

BSS
Uninitialized global and static variables

Heap
Dynamic space.
malloc(...) / free(...) 
new(...) / ~

High

Stack
Program scratch space.
Local variables, pass arguments, etc..
<table>
<thead>
<tr>
<th>Register Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| EIP          | Next instruction executed  
*Want to hijack during exploitation |
| ESP          | Stack pointer |
| EBP          | Base pointer |
| EAX          | Accumulation  
*Holds the return value, usually. |
| EBX          | Base |
| ECX          | Counter |
| EDX          | Data |
| ESI          | Source index |
| EDI          | Destination index |
The Stack
The Stack

local variables
...

arguments...
previous stack frame

EBP + x

EBP - x

ESP

EBP

EBP

RET
Okay, we have some background on the registers, the stack, and process layout.

Let’s try to figure out what this code’s stack layout would look like.

Then, we’ll look back at the code and what we know.
Example 1 — Part 2

- `sub esp, 0x10`
  - There is room for 16 bytes of locals, or 4 ints
- `[ebp-4]` is a local
- `[ebp-8]` is a local
- Return value, `eax`, is either 1 or 0 depending on the comparison

```assembly
080483b4  push ebp
080483b5  mov ebp,esp
080483b7  sub esp,0x10
080483ba  mov DWORD PTR [ebp-0x4],0x4
080483c1  mov DWORD PTR [ebp-0x8],0xa
080483c8  mov eax,DWORD PTR [ebp-0x4]
080483cb  cmp eax,DWORD PTR [ebp-0x8]
080483ce  jge 80483d7 <main+0x23>
080483d0  mov eax,0x1
080483d5  jmp 80483dc <main+0x29>
080483d7  mov eax,0x0
080483dc  leave
080483dd  ret
```
Example 1’s stack

ESP →

EBP-16

push ebp
mov ebp, esp
sub esp, 0x10
mov DWORD PTR [ebp-0x4], 0x4
mov DWORD PTR [ebp-0x8], 0xa
mov eax, DWORD PTR [ebp-0x4]
cmp eax, DWORD PTR [ebp-0x8]
jge 80483d7 <main+0x23>
mov eax, 0x1
jmp 80483dc <main+0x28>
mov eax, 0x0
leave
ret

No [ebp+x], no arguments to the function
```c
int someFunction() {
    int x = 4;
    int y = 10;
    if (4 >= 10) {
        jmp to main+0x23
    }
    eax = 1
    jmp to main+0x28
    main+0x23 :
    eax = 0
    main+0x28:
    return
```
‘if’ comparisons get translated opposite from source to assembly

if x > y
Will become
  o cmp x, y
  o jle 0x12345678 (jump less than or equal)
  o If some condition is *not true*, jump over it

if x <= y
Will become
  o cmp x, y
  o ja 0x12345678 (jmp above)
int someFunction() {
    int x = 4;
    int y = 10;
    if (4 < 10)
        Return 1
    Return 0
}

Hey, that’s source code!
Produce the source code for the following function

```
push ebp
mov ebp, esp
mov eax, DWORD PTR [ebp+0xc]
mov edx, DWORD PTR [ebp+0x8]
lea eax, [edx+eax*1]
pop ebp
ret
```

How many local variables, how many arguments, what types?

Hint: lea eax, [edx+eax*1] is the same thing as

- eax = edx+eax
What we just saw was the sum function.
The compiler used lea edx+eax for efficiency
It could have similarly used the add instruction
eax contains the return value
No local variables were used (no [ebp-x]), just arguments ([ebp+x])
Functions
Looking at the previous exercise introduces a question about how function calls are handled.

We know:
- eax holds the return value
- Arguments (from the functions point of view) begin at ebp+8

But how do those arguments get there, and how are they removed?
Two main calling conventions are commonly used

- **CDECL**
  - Originates from C
  - Args pushed on the stack, right to left (reverse)
  - Calling function cleans up

- **STDCALL**
  - Originates from Microsoft
  - Args pushed on the stack, right to left (reverse)
  - Called function cleans up
    - Must know how many bytes ahead of time
GCC tends to use: move [esp+x], arg
Visual studio tends to use: push arg
Regardless, we’re putting args on top of the stack
Now that the stack is setup, sum is called
Functions reference local variables and arguments via their stack frame pointers, esp and ebp.

So, every function has its own prolog and epilog to adjust esp and ebp to contain the correct values.
Stack Frames

- **Prolog** – push ebp to save it on the stack, then move ebp to the top of the stack, then make room for locals
  - Push ebp
  - mov ebp, esp
  - sub esp, x

- **Epilog** – move esp back to ebp, pop the top of the stack into ebp, return to the address on top of the stack
  - add esp, x
  - pop ebp
  - ret

- **Epilog 2** – leave is equivalent to: mov esp, ebp; pop ebp
  - leave
  - ret
The call instruction pushes EIP onto the stack
EBP is saved
EBP has the same value as ESP now
EAX gets the value of arg 2
EDX gets the value of arg 1
EAX contains a new value now, not what was in arg2
In the epilog now, set EBP back to the callers value

```
push ebp
mov ebp, esp
mov eax, DWORD PTR [ebp+0xc]
mov edx, DWORD PTR [ebp+0x8]
lea eax, [edx+eax*1]
pop edx
ret
```
Ret is the same as: pop EIP
Control flow returns to the next instruction in the caller
Quick Exercise — 5 minutes

What is the stack going to look like at the printf call?

```c
int function1(int age, char *name) {
    int real_age = age + 2;
    printf("Hi %s, I bet you are *really* %d years old ;)\n", name, real_age);
    return real_age;
}

int main(void) {
    function1(29, "Sally");
    return 0;
}
```
Solution

.data section char *
name
real_age
real_age

function1

Main

ESP
EBP
for (i = 0; i < 10; i++)

push ebp
mov ebp, esp
sub esp, 0x10
mov DWORD PTR [ebp-0x8], 0x0
mov DWORD PTR [ebp-0x4], 0x0
jmp 80483d4 <main+0x20>
mov eax, DWORD PTR [ebp-0x4]
add DWORD PTR [ebp-0x8], eax
add DWORD PTR [ebp-0x4], 0x1
cmp DWORD PTR [ebp-0x4], 0x9
jle 80483ca <main+0x16>
mov eax, DWORD PTR [ebp-0x4]
leave
ret
nop
Without a single instruction, it’s clear what is happening at a high level here.

This common “stair step” graph structure is a series of calls/checks that error out on failure.
IDA rocks...

- We can do many things, including grouping a set of nodes, color coding them, and renaming them.
- Knowing that all these checks error out on failure we can simplify the graph.
IDA — Simplifying the graph
I could spend on all day on IDA, too much information to put into slides without making it a pure IDA talk

*Live demo goes here*

- How to use IDA
- Go over variable renaming, function protocol modification, comments, coloring, grouping, sections, string, imports, etc.
Can you figure out the correct input to get the key program to print the key?

Use the executable number_checker.exe
Everything covered so far has been static analysis
Now we’ll cover dynamic analysis through debugging
Remember

A good debugger will have several useful features
  - Set breakpoints
  - Step into / over
  - Show loaded modules, SEH chain, etc.
  - Memory searching
  - ...

WinDBG, OllyDBG, Immunity, IDA, GDB, etc. are good debuggers
Keep in mind...

*You control everything!*

If you want to skip over an instruction, or a function call, do it!

If you want to bypass the “authentication” method or make it return true... you can!

You can change register contents and memory values, whatever you want.

You can even patch programs (make changes and save it to a new executable).
F2 will set a breakpoint in IDA, Olly, Immunity
The breakpoint has been hit, execution is stopped

- The registers
- The stack
The breakpoint has been hit, execution is stopped

- The registers
- The stack
We can now see the function call is

InterlockedCompareExchange(__native_startup_lock, 0x47000, 0)

Looking at the MSDN site for the prototype:

```c
LONG InterlockedCompareExchange(
    LPLONG Destination,
    LONG Exchange,
    LONG Comperand
);
```
Knowing the data types of the parameters, we can trace back up through the program where the values in ebx, esi and edi came from.

Then we can rename those values to something useful.

Just looking at calls, figuring out their arguments, and tracing back to fill in the data types can really help figure out most of the functions.
We’ll again use the number_checker.exe binary for this exercise.

Can you bypass the key check entirely?

In CTFs a lot of times we can see where the key gets printed, and we’ll try to just jump directly to that function, or make checks return True/False depending on where we want to go.

- Usually can get a quick low point problem this way ;}
Set a breakpoint at the beginning of the function (f2)
When execution is stopped, find where you want to jump to, and right click -> set ip
Most of the Windows debuggers are similar
  - Same windows, same hotkeys, etc.
  - Except WinDBG, WinDBG is more GDB like
GDB is similar, but is command line
We’ll cover some simple GDB usage
Starting GDB and launching the application
  - With and without arguments

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gdb ./my_program</td>
<td>Launch gdb, debug my_program</td>
</tr>
<tr>
<td>gdb --args ./my_program arg1 arg2</td>
<td>Launch gdb, debug my_program, passing two arguments</td>
</tr>
<tr>
<td>run</td>
<td>Run the application</td>
</tr>
<tr>
<td>run arg1 arg2</td>
<td>Run the application, pass two args</td>
</tr>
<tr>
<td>run $(python –c “print ‘A’*1000”)</td>
<td>Run the application, pass one arg, just like regular shell execution</td>
</tr>
</tbody>
</table>
1. Launch GDB with the program we want to debug
2. Run it

Hmm... we need more information
  ○ (I would just open it in IDA, but we’re trying to learn GDB here!)
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set disassembly-flavor intel</code></td>
<td>Use Intel syntax</td>
</tr>
<tr>
<td><code>disas [function_name]</code></td>
<td>Disassemble the chosen function</td>
</tr>
</tbody>
</table>

```
(gdb) set disassembly-flavor intel
(gdb) disass main
Dump of assembler code for function main:
0x08048434 <main+0>: push ebp
0x08048435 <main+1>: mov ebp,esp
0x08048437 <main+3>: and esp,0xffffffff0
0x0804843a <main+6>: sub esp,0x50
0x0804843d <main+9>: cmp DWORD PTR [ebp+0x8],0x3
0x08048441 <main+13>: je 0x8048456 <main+34>
0x08048443 <main+15>: mov DWORD PTR [esp],0x8048590
0x0804844a <main+22>: call 0x8048364 <puts@plt>
0x0804844f <main+27>: mov eax,0xffffffff
0x08048454 <main+32>: jmp 0x80484c6 <main+146>
```
### Dynamic Analysis - GDB

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>break main</td>
<td>Set a breakpoint on the function “main”</td>
</tr>
<tr>
<td>break *0x12345678</td>
<td>Set a breakpoint on the address 0x...</td>
</tr>
<tr>
<td>info breakpoints</td>
<td>Show information regarding breakpoints</td>
</tr>
<tr>
<td>delete breakpoint 2</td>
<td>Delete breakpoint 2</td>
</tr>
<tr>
<td>delete breakpoints</td>
<td>Delete all breakpoints</td>
</tr>
</tbody>
</table>

---

```
(gdb) break main
Breakpoint 1 at 0x8048437
(gdb) run
Starting program: /home/nomnom/FSU_Reversing/a.out
Breakpoint 1, 0x08048437 in main ()
(gdb) 
```
<table>
<thead>
<tr>
<th>Commands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>si</td>
<td>Step Instruction. Execute to next instruction, go <em>into</em> functions</td>
</tr>
<tr>
<td>ni</td>
<td>Next Instruction. Execute to next instruction, go <em>over</em> functions</td>
</tr>
</tbody>
</table>

- Look at the addresses
- We’re manually stepping through the instructions

```
(gdb) si
0x0804843a in main ()
(gdb) ni
0x0804843d in main ()
(gdb) ni
0x08048441 in main ()
(gdb) ni
0x08048443 in main ()
(gdb) ni
0x08048444 in main ()
(gdb) ni
0x0804844a in main ()
(gdb) ni
Missing something?
0x0804844f in main ()
```
<table>
<thead>
<tr>
<th>Commands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>si</td>
<td>Step Instruction. Execute to next instruction, go <em>into</em> functions</td>
</tr>
<tr>
<td>ni</td>
<td>Next Instruction. Execute to next instruction, go <em>over</em> functions</td>
</tr>
</tbody>
</table>

- Look at the addresses
- We’re manually stepping through the instructions

This still isn’t helping us though!
We can disassemble, set breakpoints, and step through the program... but

We need to
  - See the contents of registers
  - See the contents of memory
  - Modify (if desired)
x/nfu <address>

Print memory.

n: How many units to print (default 1).
f: Format character (like „print“).
u: Unit.

Unit is one of:

b: Byte,
h: Half-word (two bytes)
w: Word (four bytes)
g: Giant word (eight bytes).
### Command Table

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x/5i $eip</td>
<td>Examine 5 instructions at EIP</td>
</tr>
<tr>
<td>x/4xw $esp</td>
<td>Examine 4 hex words at ESP</td>
</tr>
<tr>
<td>x/s 0x12345678</td>
<td>Examine the string at 0x12345678</td>
</tr>
<tr>
<td>x/5b $ecx</td>
<td>Examine 5 bytes at ECX</td>
</tr>
<tr>
<td>i r</td>
<td>“info register”, show the values of all registers</td>
</tr>
<tr>
<td>i r esp ebp ecx</td>
<td>Show the values of registers ESP, EBP, and ECX</td>
</tr>
</tbody>
</table>
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/nomnom/FSU_Reversing/a.out

Breakpoint 1, 0x08048437 in main ()
(gdb) x/5i $eip
0x08048437 <main+3>:      and  esp,0xfffffffff0
0x0804843a <main+6>:      sub  esp,0x50
0x0804843d <main+9>:      cmp  DWORD PTR [ebp+0x8],0x3
0x08048441 <main+13>:      je   0x8048456 <main+34>
0x08048443 <main+15>:      mov  DWORD PTR [esp],0x8048590
(gdb) ni
0x0804843a in main ()
(gdb) ni
0x08048443d in main ()
(gdb) x/5i $eip
0x0804843d <main+9>:      cmp  DWORD PTR [ebp+0x8],0x3
0x08048441 <main+13>:      je   0x8048456 <main+34>
0x08048443 <main+15>:      mov  DWORD PTR [esp],0x8048590
0x804844a <main+22>:      call 0x8048364 <puts@plt>
0x804844f <main+27>:      mov  eax,0xffffffff
(gdb) x/xw $ebp+0x8
0xbffffffcd0:  0x00000001
(gdb) ni
0x8048441 in main ()
(gdb) ni
0x8048443 in main ()
(gdb) x/i $eip
0x8048443 <main+15>:      mov  DWORD PTR [esp],0x8048590
(gdb) x/s 0x8048590
0x8048590:  "Missing something?"
(gdb) ni
0x8048444a in main ()
(gdb) ni
Missing something?
0x804844f in main ()
1. Run the program

Breakpoint 1, 0x08048437 in main ()
(gdb) x/5i $eip
0x08048437 <main+3>: and esp,0xfffffffff0
0x0804843a <main+6>: sub esp,0x50
0x0804843d <main+9>: cmp DWORD PTR [ebp+0x8],0x3
0x08048441 <main+13>: je 0x8048456 <main+34>
0x08048443 <main+15>: mov DWORD PTR [esp],0x8048590
(gdb) ni
0x0804843a in main ()
(gdb) ni
0x0804843d in main ()
(gdb) x/5i $eip
0x0804843d <main+9>: cmp DWORD PTR [ebp+0x8],0x3
0x08048441 <main+13>: je 0x8048456 <main+34>
0x08048443 <main+15>: mov DWORD PTR [esp],0x8048590
0x0804844a <main+22>: call 0x8048364 <puts@plt>
0x0804844f <main+27>: mov eax,0xffffffff
(gdb) x/xw $ebp+0x8
0xbfffffffcd0: 0x00000001
(gdb) ni
0x08048441 in main ()
(gdb) ni
0x08048443 in main ()
(gdb) x/i $eip
0x08048443 <main+15>: mov DWORD PTR [esp],0x8048590
(gdb) x/s 0x8048590
0x8048590: "Missing something?"
(gdb) ni
0x0804844a in main ()
(gdb) ni
Missing something?
0x0804844f in main ()
1. Run the program

2. Where are we? Check out EIP
1. Run the program

The program being debugged has been started already. Start it from the beginning? (y or n) y
Starting program: /home/nomnom/FSU_Reversing/a.out

Breakpoint 1, 0x0048437 in main ()

(gdb) x/i $eip
0x0048437 <main+3>: and esp,0xffffffff0
0x004843a <main+6>: sub esp,0x50
0x004843d <main+9>: cmp DWORD PTR [ebp+0x8],0x3
0x0048441 <main+13>: je 0x0048456 <main+34>
0x0048443 <main+15>: mov DWORD PTR [esp],0x0048590
(gdb) ni
0x0048443a in main ()
(gdb) ni
0x0048443d in main ()
(gdb) x/i $eip
0x0048443d <main+9>: cmp DWORD PTR [ebp+0x8],0x3
0x00484441 <main+13>: je 0x0048456 <main+34>
0x00484443 <main+15>: mov DWORD PTR [esp],0x0048590
0x0048444a <main+22>: call 0x0048364 <puts@plt>
0x0048444f <main+27>: mov eax,0xffffffff
(gdb) x/w $ebp+0x8
0xbffffcd0: 0x00000001
(gdb) ni
0x00484441 in main ()
(gdb) ni
0x00484443 in main ()
(gdb) x/i $eip
0x00484443 <main+15>: mov DWORD PTR [esp],0x0048590
(gdb) x/s 0x0048590
0x0048590: "Missing something?"
(gdb) ni
0x0048444a in main ()
(gdb) ni
Missing something?
0x0048444f in main ()

2. Where are we? Check out EIP

3. Continue until we hit an instruction of interest
1. Run the program

2. Where are we? Check out EIP

3. Continue until we hit an instruction of interest

4. Let’s see what’s being compared – we can see this jump is not taken
1. Run the program

2. Where are we? Check out EIP

3. Continue until we hit an instruction of interest

4. Let's see what's being compared – we can see this jump is not taken

5. Check out the argument passed to puts
1. Run the program

2. Where are we? Check out EIP

3. Continue until we hit an instruction of interest

4. Let’s see what’s being compared – we can see this jump is not taken

5. Check out the argument passed to puts

Aha! We don’t satisfy the compare (1 != 3), and call puts, then exit!
Think about the function protocol for main
- int main (int argc, char *argv[])

In main, [ebp+8] would reference the first argument,argc

We aren’t passing any arguments, besides argv[0], the program name, hence why [ebp+8] has the value 1
Haha, passing the program 2 more arguments (3 total) does in fact satisfy the first `cmp` instruction.

A new code path is taken!
Try to figure out the correct input that will cause the program to print message, "Congrats, you did it!"

Use IDA and GDB!

- Hey, we’ve seen this graph pattern before!
Everyone has their own preferences

But the combination of the two will undoubtedly yield the best results

IDA, WinDBG, Immunity, GDB all have scripting

- In fact, they all use Python except WinDBG*
- There are awesome scripts that will import results from debuggers into IDA’s view, filling in all the registers/operands for each instruction.
key_checker.exe or
We’ll do a real crackme
Crackme at
This might be a little tricky, that’s okay.
What about bytecode?
  - .NET applications, java, python, etc.

Just download a disassembler

You’ll get near complete source code back

It’s really that easy...
Hopefully you feel comfortable

- Opening up and examining a binary and looking at it's sections to get a feel for it
- Renaming and simplifying the disassembly
- Converting back to source code where needed
- Using a debugger to fill in the gaps or manipulate program execution
Conclusion

- **Fantastic books**
  - Reversing: The secrets of reverse engineering
  - The IDA Pro book
  - The Art of Exploitation

- **Challenges**
  - Crackmes.de
  - Woodmann.com
  - Smashthestack.org (plenty of debugging involved ;))

- **Links**
  - CSG: csg.utdallas.edu and irc.oftc.net #utdcsg (everyone is welcome)
  - IDA: hex-rays.com
  - CFF Explorer: ntcore.com/exsuite.php
  - Immunity Debugger: immunityinc.com