Outline

• Assembly
• PE and ELF
• The Stack
• Static Vs. Dynamic
• 32 Vs. 64 Bit
• What About Bytecode?
• **4 EXERCISES!**

• So, I had this fancy outline in mind... but there’s so much information, it’s not very fun.
• Instead...
Outline

• Instead, we’re going to start trying to reverse some code, and learn new things along the way
Instruction Format

• Two syntax options
  ▫ ATT
  ▫ Intel

• ATT
  ▫ Opcode source, dest
  ▫ mov %eax, %ecx

• Intel
  ▫ Opcode dest, source
  ▫ mov eax, ecx
...Instruction Format

• It’s know fact that Intel’s syntax > ATT’s, so we’ll use Intel’s ;)
• mov eax, ecx
  ▫ Move the contents of ecx, into eax
• mov eax, [ecx]
  ▫ Move the contents of what ecx points to, into eax
  ▫ In C, this is like a pointer dereference
  ▫ eax = *ecx;
  ▫ [...] means dereference the address between the brackets
...Instruction Format

- Memory values and immediate values can be used as well
- `mov eax, 5`
  - Move into `eax`, the value 5
- `mov eax, [0x12345678]`
  - Move into `eax`, the contents of what `0x12345678` points to
...Instruction Format

- A small handful of instructions will get you a long ways
- **call, mov, cmp, jmp**
- call 0x12345678
  - A function call to 0x1234567
- cmp eax, 8
  - Comparing eax to 8
- jmp 0x12345678 (unconditional)
- jle 0x12345678
  - Jump to 0x1234567 if eax is less than or equal to 8
- jg 0x12345678
  - Jmp to 0x12345678 if eax is greater than 8
Example 1

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
Example 1

- Let’s just focus on the instructions we know
  - mov, cmp, jmp
Example 1

- \([ebp-0x4] = 0x4\)
- \([ebp-0x8] = 0xa\)
- \(eax = [ebp-0x4]\)

- Two values, relative to the pointer contained in ebp have been assigned values
- One register has been assigned a value
Example 1

- \([ebp-0x4] = 0x4\)
- \([ebp-0x8] = 0xa\)
- \(eax = [ebp-0x4]\)
- \(\text{cmp } eax, [ebp-0x8]\)
  - \(eax == [ebp-0x8] \) ?
  - \(4 == 10 \) ?
- \(\text{jge } 0x80483d7\)
  - If 4 was \(>= 10\), jmp
  - Else, continue execution
Example 1

- \([ebp-0x4] = 0x4\)
- \([ebp-0x8] = 0xa\)
- \(eax = [ebp-0x4]\)
- \(cmp eax, [ebp-0x8]\)
  - \(eax == [ebp-0x8]?\)
  - \(4 == 10?\)
- \(jge 0x80483d7\)
  - If 4 was \(\geq 10\), jmp
  - Else, continue execution

False, so execution just continues to the next instruction
Example 1

- 
  - \([\text{ebp-0x4}] = 0x4\)
  - \([\text{ebp-0x8}] = 0xa\)
  - \(\text{eax} = [\text{ebp-0x4}]\)
  - \(\text{cmp} \ \text{eax}, [\text{ebp-0x8}]\)
  - \(\text{jge} \ \text{0x80483d7}\)
  - \(\text{mov} \ \text{eax}, 0x1\)
    - \(\text{eax} = 1\)
  - \(\text{jmp} \ \text{over the mov eax, 0}\)
  - \(\text{leave and return}\)
Example 1

- 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
Example 1

• Let’s dig deeper
• Everything shown in the disassembly has a purpose
• `mov DWORD PTR [ebp-0x4], 0x4`
  ▫ What does DWORD PTR mean?
• We know the brackets [...] mean get the value held at the dereferenced value between them... but DWORD PTR?
Example 1

- `mov DWORD PTR [ebp-0x4], 0x4`
- `DWORD PTR`
  - `DWORD = the size`
  - `PTR = dereference the value, always accompanied by the brackets`
- `We have a few number of sizes allowed`
### Example 1 - Types and Sizes

<table>
<thead>
<tr>
<th>Name</th>
<th>Size (bytes)</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>1 byte</td>
<td>8 bits</td>
</tr>
<tr>
<td>Word</td>
<td>2 bytes</td>
<td>16 bits</td>
</tr>
<tr>
<td>Double word (dword)</td>
<td>4 bytes</td>
<td>32 bits</td>
</tr>
<tr>
<td>Quad word (qword)</td>
<td>8 bytes</td>
<td>64 bits</td>
</tr>
</tbody>
</table>

32 bit architectures
64bit architectures
## Example 1 - Types and Sizes

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

- 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.
- This sounds complicated, but... we know something now
- `[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

- \texttt{mov DWORD PTR [ebp-0x4], 0x4}
- \texttt{mov DWORD PTR [ebp-0x8], 0xa}
- This leaves us with 2 ints being assigned a hard coded value
  - \texttt{int x = 4;}
  - \texttt{int y = 10;}
- Are these locals, globals, static variables???
- We need a little background on process memory layout.
Example 1 - Recap so far

- `int x = 4;`
- `int y = 10;`
  - We don’t know where these are declared
- `if (4 <= 10)`
  - `jmp to main+0x23`
- `main+0x23 :`
  - `eax = 0`

- We take the `jmp` as already discussed.
- It’s starting to look like source code!
Process Memory Layout

• 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?
Virtual Memory

- **Text**: Code segment, machine instr.
- **Data**: Initialized global and static variables
- **BSS**: Uninitialized global and static variables
- **Heap**: Dynamic space. `malloc(...) / free(...) / ~`
- **Stack**: Program scratch space. Local variables, pass arguments, etc.
The Stack

ESP

EBP

local variables
...

EBP

RET

arguments...
previous stack frame

EBP - x

EBP + x

Low

High
# Registers

<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 |
Example 1 - Part 2

- 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
Example 1’s stack

ESP  →  EBP-16

|   10   |
|       |
|       |
|       |
|       |
|       |
|       |
|       |
|       |

args start at EBP+8

No [ebp+x], no arguments to the function

ESP

EBP

RET

EBP-8

EBP-4

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
Example 1 - Part 2

- `int someFunction() {`
- `int x = 4;`
- `int y = 10;`
- `if (4 >= 10)`
  - `jmp to main+0x23`
- `main+0x23 :`
  - `Return 0 (eax is 0)`

```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+0x28>
080483d7: mov eax,0x0
080483dc: leave
080483dd: ret
```
A side note about source to asm

- ‘if’ comparisons get translated opposite from source to assembly
- if x > y
- Will become
  - cmp x, y
  - jle 0x12345678 (jump less than or equal)
- If x <= y
- Will become
  - cmp x, y
  - ja 0x12345678 (jmp above)
Example 1 - Part 2

- int someFunction() {
-     int x = 4;
-     int y = 10;
-     if (4 < 10)
-         Return 1
-     Return 0
- }

- Hey, that's source code!
Exercise 1!

- Produce the source code for the following function

```
080483b4 <sum>:
80483b4:  55  push    ebp
     89 e5  mov      ebp,esp
80483b7:  8b 45 0c  mov      eax,DWORD PTR [ebp+0xc]
80483ba:  8b 55 08  mov      edx,DWORD PTR [ebp+0x8]
80483bd:  8d 04 02  lea      eax,[edx+eax*1]
80483c0:  5d      pop      ebp
80483c1:  c3      ret
```

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

• 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

```plaintext
sum(int x, int y) {
    return x + y;
}

main(void) {
    return sum(5, 7);
}
```
Functions

• Looking at exercise 1 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?
Functions - Calling Conventions

- 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
Exercise 1 - Main

- 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
Exercise 1 - Main

Now that the stack is setup, sum is called
Stack Frames

• Functions reference local variables and arguments via their stack frame pointer, esp and ebp
• So, every function has it’s 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
Stack Frames - Exercise 1

- The call instruction pushes EIP onto the stack

```assembly
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
```
Stack Frames - Exercise 1

- EBP is saved

```assembly
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
```
• EBP has the same value as ESP now
• EAX gets the value of arg 2
Stack Frames - Exercise 1

- EDX gets the value of arg 1

```
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
```
• EAX contains a new value now, not what was in arg2
Stack Frames - Exercise 1

- 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     ebp
ret
```
Stack Frames - Exercise 1

• Ret is the same as: pop EIP
• Control flow returns to the next instruction in the caller
Recognizing Patterns

- \texttt{for(i = 0; i < 10; i++)}
Recognizing Patterns

- 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

- 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
IDA

- 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*
  - Go over variable renaming, function protocol modification, comments, coloring, grouping, sections, string, imports, etc.
Exercise 2

- Can you crack the super secret key checking algorithm?
- Grab the executable (or .idb file) from utdcsg.org
Dynamic Analysis

- Everything so far has been static
  - Look at the code, figure things out
  - A safer approach
  - Everything is there, but possibly more complicated
- Dynamic
  - Run the code, debug it during execution
  - Can see the value of things in real time
  - Some find it easier
  - Safer (you’re not running the code on your box!)
Dynamic Analysis - Debuggers

- **Windows**
  - WinDBG, Immunity, OllyDBG, IDA, ...
- **Linux**
  - GDB, IDA+GDB, ?
Dynamic Analysis - Debuggers

- Every debugger has a common set of features
  - And they usually share the same hotkeys as well
- Set breakpoints
- Single step into
- Single step over
- Continue
- And helpful things like
  - Show SHE chain, loaded modules, pattern searching, etc.
Dynamic Analysis - Quick Note

• 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).
Dynamic Analysis - IDA

- 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
Dynamic Analysis - IDA

- 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
);
```
Dynamic Analysis - IDA

- 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.
Exercise 3

- Download the Windows version of the key_checker program from utdcs.org
- Can you bypass the key check entirely?

- In CTFs a lot of times we can see where the key get’s 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 ;)


Exercise 3 - Solution

- Set a breakpoint at the beginning of the function
Exercise 3 - Solution

- When execution is stopped, find where you want to jump to, and right click -> set ip
Dynamic Analysis - Debuggers

- 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
Dynamic Analysis - GDB

- \texttt{gdb -q ./my\_program}
  - \texttt{-q} specifies quietly, no banner printed
- \texttt{run}
- \texttt{break my\_function, break \*0x12345678}
- \texttt{info breakpoints}
- \texttt{disassemble my\_function}
- \texttt{info registers}
  - Print the registers, “i r eax” for just one
- \texttt{print 0x123 – 0x567}
  - Will print simple mathematical results
Dynamic Analysis - GDB

- Okay, this is as hard as it gets... and it’s not that bad
- x/s 0x12345678
  - Examine the string at 0x12345678
- x/4xw 0x12345678
  - Examine 4 hex words at 0x...
- x/xw $esp
  - Examine a hex word at esp
- x/10i $eip
  - Examine 10 instructions at eip
**Dynamic Vs. Static**

- 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.
Last Exercise

- We’ll do a real crackme
- Crackme at
- Solution at
- This might be a little tricky, that’s okay.
One quick note

- 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...
Conclusion

• 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

• If you’re interested in some packing / obfuscation techniques check out the 3 sections on the utdcsg.org site we did a while back over that topic
Conclusion

- Fantastic books
  - Reversing: The secrets of reverse engineering
  - The IDA Pro book
- Challenges
  - Crackmes.de
  - Woodmann.com