x86 Review: wat dat assembly mean

Week1 – Spring 2015

Presenter: Adam Pond
IRC Nick: xe0
irc.freenode.net #0x4f
Agenda

- Spring Break
- iSEC internship follow-up
- New at UOSec
- x86 is hard
- Stack frames are fun
Spring Break
Spring Break Cont.

Welcome to Raytheon

1220 N. Hwy. A1A, Suite 123 | Indialantic, FL 32903-2858
iSEC Internship Follow-up

- We had some members apply for the iSEC summer internship
iSEC Internship Follow-up Cont.

- We had some members apply for the iSEC summer internship
- Congrats to UOSec member Frank Arana!
New at UOSec
New at UOSec Cont.

- Tuesday walkthroughs
New at UOSec Cont.
New at UOSec Cont.

- Tuesday walkthroughs
- Thursday meetings (what we’re in now)
New at UOSec Cont.

- Tuesday walkthroughs
- Thursday meetings (what we’re in now)
- Saturday CTFs (Capture the Flags)
New at UOSec Cont.
New at UOSec Cont.

- Tuesday walkthroughs
- Thursday meetings (what we’re in now)
- Saturday CTFs (Capture the Flags)
- uosec.net (new format)
And now learning

WHAT IF I TOLD YOU

LEARNING IS GOOD FOR YOU
Intro to Reverse Engineering

- Our definition of RE is going to be taking a binary (program) and disassembling it into assembly code in IDA (or another disassembler) in order to figure out what the program is doing.
Intro to Reverse Engineering Cont.

- But, before we get into RE, we need a primer on x86 assembly.
Nooo Not Assembly
Why Assembly, Why?

High Level Language

```
int c;
printf("Hello.\n");
exit(0);
```

Machine Code

```
55
8B EC
8B EC 40
```

Low-Level Language

```
push ebp
move ebp, esp
sub esp, 0x40
```
x86 - Let's Be Real

- We don’t have time to do a comprehensive coverage of x86
x86 - Lets Be Real

● We don’t have time to do a comprehensive coverage of x86
● Turns out that with a limited number of instructions we can cover a large codebase
x86 - Lets Be Real

- We don’t have time to do a comprehensive coverage of x86
- Turns out that with a limited number of instructions we can cover a large codebase
- Besides, what we’re really interested in is reversing and we can look up instructions we don’t cover in this primer
x86 - Let's Be Real

- We don’t have time to do a comprehensive coverage of x86
- Turns out that with a limited number of instructions we can cover a large codebase
- Besides, what we’re really interested in is reversing and we can look up instructions we don’t cover in this primer
- If you have any questions please ask while they are fresh in your mind
x86 - Let's Talk Registers

- Registers are special memory built into the CPU where data is saved during the operation of a program.
x86 - Lets Talk Registers

- Registers are special memory built into the cpu where data is saved during the operation of a program
- There are 8 general 32 bit registers used in x86: eax, ebx, ecx, edx, edi, esi, ebp, esp
x86 - Lets Talk Registers

- Registers are special memory built into the cpu where data is saved during the operation of a program.
- There are 8 general 32 bit registers used in x86: eax, ebx, ecx, edx, edi, esi, ebp, esp.
- We are also concerned with eip and EFLAGS registers.
x86 - Let's Talk Registers Cont.

General-Purpose Registers:

- EAX
- EBX
- ECX
- EDX
- ESI
- EDI
- ESP (stack pointer)
- EBP (base pointer)

Bytes:

- 8 bits
- 16 bits
- 32 bits
x86 - Lets Talk Registers - Eax

Name: eax
Length: 32bits / 4 bytes
Intel Compiler Suggestion: Store the return value of a called function
x86 - Lets Talk Registers - Ebx

Name: ebx
Length: 32bits / 4 bytes
Intel Compiler Suggestion: Base pointer to the .data section of memory
x86 - Lets Talk Registers - Ecx

Name: ecx
Length: 32bits / 4 bytes
Intel Compiler Suggestion: Counter for string and loop operations
x86 - Lets Talk Registers - Edx

Name: edx
Length: 32bits / 4 bytes
Intel Compiler Suggestion: I/O pointer
x86 - Lets Talk Registers - Esi

Name: esi
Length: 32bits / 4 bytes
Intel Compiler Suggestion: Source pointer for string operations
x86 - Let's Talk Registers - Edi

Name: edi
Length: 32 bits / 4 bytes
Intel Compiler Suggestion: Destination pointer for string operations
x86 - Lets Talk Registers - Esp

Name: esp
Length: 32bits / 4 bytes
Intel Compiler Suggestion: Stack pointer which points to the top of the current stack frame
x86 - Lets Talk Registers - Ebp

Name: ebp
Length: 32bits / 4 bytes
Intel Compiler Suggestion: Base pointer which points to the bottom of the current stack frame
x86 - Lets Talk Registers - Eip

Name: eip
Length: 32bits / 4 bytes
Intel Compiler Suggestion: Pointer to the next instruction for the cpu to execute
Name: EFLAGS
Length: 32bits / 4 bytes
Intel Compiler Suggestion: Register holds many single bit flags which are set and cleared every instruction of the cpu.
x86 - Lets Talk Registers - EFLAGS

EFLAGS Register

- ID Flag (ID)
- Virtual Interrupt Pending (VIP)
- Virtual Interrupt Flag (VIF)
- Alignment Check (AC)
- Virtual-8086 Mode (VM)
- Resume Flag (RF)
- Nested Task (NT)
- I/O Privilege Level (IOPL)
- Overflow Flag (OF)
- Direction Flag (DF)
- Interrupt Enable Flag (IF)
- Trap Flag (TF)
- Sign Flag (SF)
- Zero Flag (ZF)
- Auxiliary Carry Flag (AF)
- Parity Flag (PF)
- Carry Flag (CF)

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

Reserved bit positions. DO NOT USE.
Always set to values previously read.
x86 - Let's Talk Calling Conventions

- Calling convention is an implementation-level scheme for how subroutines receive parameters from their caller and how they return a result.
x86 - Let's Talk Calling Conventions

- Calling convention is an implementation-level scheme for how subroutines receive parameters from their caller and how they return a result.
- We will mostly be using cdecl since we will be using C programs for most examples.
x86 - Let's Talk Calling Conventions

- Calling convention is an implementation-level scheme for how subroutines receive parameters from their caller and how they return a result.
- We will mostly be using cdecl since we will be using C programs for most examples.
- Be aware that other calling conventions exist such as stdcall, and fastcall.
x86 - Let's Talk Cdecl

- Caller saved registers - eax, edx, ecx
  This means before calling a subroutine, the caller must save any value in said registers it doesn’t want clobbered by the called subroutine
x86 - Lets Talk Cdecl

- Caller saved registers - eax, edx, ecx
  This means before calling a subroutine, the caller must save any value in said registers it doesn’t want clobbered by the called subroutine

- Callee saved registers - ebp, ebx, esi, edi
  Same as above, if callee plans on modifying the regs they must be saved and restored
x86 - Let's Talk The Stack

- Conceptual piece of memory that is used by the operating system to manage program data
x86 - Lets Talk The Stack

- Conceptual piece of memory that is used by the operating system to manage program data
- Stack is LIFO data structure
x86 - Lets Talk The Stack

- Conceptual piece of memory that is used by the operating system to manage program data
- Stack is LIFO data structure
- By convention the Stack grows towards lower memory addresses. This seems weird but the “top” of the stack is at a numerically lower address than the bottom
x86 - Lets Talk The Stack Cont.

- It’s important to remember that esp (the stack pointer) is always pointing to the top of the stack (numerically lower)
• It’s important to remember that esp (the stack pointer) is always pointing to the top of the stack (numerically lower)
• The main use of the stack is to keep track of functions being called.
x86 - Lets Talk The Stack Cont.

- It’s important to remember that esp (the stack pointer) is always pointing to the top of the stack (numerically lower)
- The main use of the stack is to keep track of functions being called.
- It always keeps track of local variables, and function parameters are passed (in cdecl) on the stack
* It’s important to remember that esp (the stack pointer) is always pointing to the top of the stack (numerically lower)
* The main use of the stack is to keep track of functions being called.
* It always keeps track of local variables, and function parameters are passed (in cdecl) on the stack
* Read more about the stack on uosec.net
x86 - Let's Talk The Stack Cont.

Stack Growth

Stack

Higher Addresses

saved ESI
saved EDI
local variable 3
local variable 2
local variable 1
saved EBP
return address
parameter 1
parameter 2
parameter 3

ESP

EBP

[ebp]-4
[ebp]+8
[ebp]+12
[ebp]+16
x86 - Lets Talk Instructions

● First a few notes
● We will be covering intel x86 syntax pretty much at all times
● If you’re interested in at&t syntax, I can’t help you (j/k)
x86 - Lets Talk Instructions Cont.

- Arithmetic or bitwise instructions: xor, and, add, sub, dec, inc, div, mul
- Comparison instructions: cmp, test
- Jump instructions: jmp, je, ja, jb, jg, jl, jz
- Misc instructions: mov, lea, pop, push, leave, nop, call, ret
- Intel x86 syntax follows destination, src format i.e. add esp, 0x40 adds 0x40 to the current value of esp and saves result in esp
x86 - xor

Opcode: 0x31
Description: Performs bitwise xor operation
Our use: Usually used to zero out a register
### Registers Before

<table>
<thead>
<tr>
<th>eax</th>
<th>0x00001001</th>
</tr>
</thead>
</table>

### Registers After

<table>
<thead>
<tr>
<th>eax</th>
<th>0x00000000</th>
</tr>
</thead>
</table>

### XOR eax, eax

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A XOR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
x86 - and

**Opcode:** 0x21

**Description:** Performs bitwise and operation

**Our use:** Usually used to perform bitwise masks
and eax, 0x3

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A AND B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
x86 - add

Opcode: 0x01
Description: Performs bitwise add operation
Our use: Usually used for basic addition
### Registers Before

<table>
<thead>
<tr>
<th>eax</th>
<th>0x00000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>edx</td>
<td>0x0000001F</td>
</tr>
</tbody>
</table>

### Registers After

<table>
<thead>
<tr>
<th>eax</th>
<th>0x00000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>edx</td>
<td>0x00000020</td>
</tr>
</tbody>
</table>

### add edx, eax

\[
\begin{align*}
0x00000001 & + 0x0000001F = 0x00011111 \\
0x100000 & = 0x100000
\end{align*}
\]
x86 - sub

Opcode: 0x29
Description: Performs bitwise sub operation
Our use: Usually used for basic subtraction
Registers Before

<table>
<thead>
<tr>
<th>eax</th>
<th>0x000000FD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td>0x000000FF</td>
</tr>
</tbody>
</table>

Registers After

<table>
<thead>
<tr>
<th>eax</th>
<th>0x000000FD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td>0x00000002</td>
</tr>
</tbody>
</table>

\[
\text{sub ebx, eax}
\]

\[
0x000000FF - 0x000000FD = 0x11111111010x00000010
\]
x86 - dec

Opcode: 0x48
Description: Decrements value by 1
Our use: Usually used in loops
dec ecx
x86 - inc

Opcode: 0x40
Description: Increments value by 1
Our use: Usually used in loops
### Registers Before

<table>
<thead>
<tr>
<th>eax</th>
<th>0x000000FD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecx</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

### Registers After

<table>
<thead>
<tr>
<th>eax</th>
<th>0x000000FD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecx</td>
<td>0x00000001</td>
</tr>
</tbody>
</table>

**inc ecx**
x86 - div

Opcode: 0xF76
Description: Unsigned divide operation on eax which stores quotient in eax and the remainder in edx
Our use: Used on unsigned values when division is needed (use idiv for signed)
### Registers Before

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x00000008</td>
</tr>
<tr>
<td>ebx</td>
<td>0x00000000</td>
</tr>
<tr>
<td>ecx</td>
<td>0x00000005</td>
</tr>
<tr>
<td>edx</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

### Registers After

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x00000001</td>
</tr>
<tr>
<td>ebx</td>
<td>0x00000008</td>
</tr>
<tr>
<td>ecx</td>
<td>0x00000005</td>
</tr>
<tr>
<td>edx</td>
<td>0x00000003</td>
</tr>
</tbody>
</table>

### div ecx

```
64 bit dividend

\[
\begin{align*}
\text{EDX} & & \text{EAX} \\
\hline
32 \text{ bit Divisor} & = & \text{Quotient} \\
\text{EAX} & & \text{And} \\
\text{EDX} & & \text{Remainder}
\end{align*}
\]
```
x86 - mul

Opcode: 0xF74

Description: Unsigned multiply operation on eax with another register and stores the product in edx:eax

Our use: Used on unsigned values when multiplication is needed (use imul for signed)
### Registers Before

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x00000003</td>
</tr>
<tr>
<td>ebx</td>
<td>0x00000004</td>
</tr>
<tr>
<td>ecx</td>
<td>0x00000000</td>
</tr>
<tr>
<td>edx</td>
<td>0x00000001</td>
</tr>
</tbody>
</table>

### Registers After

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x0000000C</td>
</tr>
<tr>
<td>ebx</td>
<td>0x00000004</td>
</tr>
<tr>
<td>ecx</td>
<td>0x00000000</td>
</tr>
<tr>
<td>edx</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

### mul ebx

```
EAX  ×  32 Bit Source = EDX  EAX
```
x86 - cmp

Opcode: 0x39
Description: Compares two values and sets EFLAGS register based on result which is used for conditional execution
Our use: Used to make jumps based on result of cmp
Registers Before

<table>
<thead>
<tr>
<th>eax</th>
<th>0x00000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td>0x00000001</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

Registers After

<table>
<thead>
<tr>
<th>eax</th>
<th>0x00000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td>0x00000001</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>0x00000006</td>
</tr>
</tbody>
</table>

cmp eax, ebx

Note: cmp is just like sub except it doesn’t store the result of the operation, it just sets flags.
x86 - test

**Opcode:** 0x85

**Description:** Works just like bitwise and operator except it doesn’t save the result

**Our use:** Used to set flags and determine conditional behavior
### Registers Before

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x00000001</td>
</tr>
<tr>
<td>ecx</td>
<td>0x00000000</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

### Registers After

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x00000001</td>
</tr>
<tr>
<td>ecx</td>
<td>0x00000000</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>0x00000006</td>
</tr>
</tbody>
</table>

**test ecx, ecx**

Note: test is just like and except it doesn’t store the result of the operation, it just sets flags.
x86 - jmp

 Opcode: 0xE9
 Description: Unconditional jump to whatever relative address is provided as an argument
 Our use: jmp esp commonly used to jump to shellcode stored where esp is pointing
**Registers Before**

<table>
<thead>
<tr>
<th>eax</th>
<th>0x00000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>eip</td>
<td>0x08040125</td>
</tr>
</tbody>
</table>

**Jmp esp**

**Registers After**

<table>
<thead>
<tr>
<th>eax</th>
<th>0x00000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>eip</td>
<td>0xDEADBEEF</td>
</tr>
</tbody>
</table>
x86 - je/jz

**Opcode:** 0x0F84

**Description:** Jump to relative address if the zero flag of the EFLAGS register is set

**Our use:** Part of the conditional jump family of instructions. Used to jump to relative addresses under certain conditions
Note: jz and je work exactly the same. The rest of the jump instructions work similar look em up.
x86 - mov

Opcode: 0x89
Description: Move from memory to register, register to register, or immediate to memory or register.
Our use: Mov is used often to do what you’d expect, move data into registers or memory.
`mov eax, ebx`
`mov ecx, [eax]`
`mov ebx, 0x20`

Note: You can never move memory to memory
x86 - push

**Opcode:** 0x50

**Description:** Push a word (2bytes), dword(4bytes), or quadword(8bytes) onto the stack. Automatically decrements the stack pointer (esp) by number of bytes pushed.

**Our use:** Push values to the stack before function calls.
### Registers Before

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td>0x0804108C</td>
</tr>
<tr>
<td>ecx</td>
<td>0x00000000A</td>
</tr>
</tbody>
</table>

### Registers After

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td>0x08041088</td>
</tr>
<tr>
<td>ecx</td>
<td>0x00000000A</td>
</tr>
</tbody>
</table>

### push ecx

**Stack Before**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08041090</td>
<td>0x00001234</td>
</tr>
<tr>
<td>0x08041088</td>
<td>0x00002345</td>
</tr>
<tr>
<td>0x08041084</td>
<td>undefined</td>
</tr>
<tr>
<td>0x08041080</td>
<td>undefined</td>
</tr>
<tr>
<td>0x0804107C</td>
<td>undefined</td>
</tr>
</tbody>
</table>

**Stack After**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08041090</td>
<td>0x00001234</td>
</tr>
<tr>
<td>0x08041088</td>
<td>0x00002345</td>
</tr>
<tr>
<td>0x08041084</td>
<td>0x00000000A</td>
</tr>
<tr>
<td>0x08041080</td>
<td>undefined</td>
</tr>
<tr>
<td>0x0804107C</td>
<td>undefined</td>
</tr>
</tbody>
</table>
x86 - pop

Opcode: 0x58

Description: Pop a word (2bytes), dword(4bytes), or quadword(8bytes) off the stack into a register. Automatically increments the stack pointer (esp) by number of bytes popped.

Our use: Pop values off the stack to restore values previous to function calls.
Registers Before

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td>0x0804108C</td>
</tr>
<tr>
<td>ecx</td>
<td>0x0000000A</td>
</tr>
</tbody>
</table>

Registers After

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td>0x08041090</td>
</tr>
<tr>
<td>ecx</td>
<td>0x00002345</td>
</tr>
</tbody>
</table>

**pop ecx**

Stack Before

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08041090</td>
<td>0x00001234</td>
</tr>
<tr>
<td>0x0804108C</td>
<td>0x00002345</td>
</tr>
<tr>
<td>0x08041088</td>
<td>undefined</td>
</tr>
<tr>
<td>0x08041084</td>
<td>undefined</td>
</tr>
<tr>
<td>0x08041080</td>
<td>undefined</td>
</tr>
<tr>
<td>0x0804107C</td>
<td>undefined</td>
</tr>
</tbody>
</table>

Stack After

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08041090</td>
<td>esp → 0x08041090</td>
</tr>
<tr>
<td>0x00001234</td>
<td>0x0804108C</td>
</tr>
<tr>
<td>0x0804108C</td>
<td>undefined</td>
</tr>
<tr>
<td>0x08041088</td>
<td>undefined</td>
</tr>
<tr>
<td>0x08041084</td>
<td>undefined</td>
</tr>
<tr>
<td>0x08041080</td>
<td>undefined</td>
</tr>
<tr>
<td>0x0804107C</td>
<td>undefined</td>
</tr>
</tbody>
</table>
x86 - call

Opcode: 0xE8

Description: Transfers control to a different function such that control can be returned back to the function that invoked call at a later time. call will push the next instruction in the current function to the stack and then places address of called function in eip.

Our use: calls functions, pretty self-explanatory.
**call getchar**

### Registers Before

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td>0x0804107C</td>
</tr>
<tr>
<td>eip</td>
<td>0x0804108C</td>
</tr>
</tbody>
</table>

### Registers After

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td>0x08041078</td>
</tr>
<tr>
<td>eip</td>
<td>0x0BF0F123</td>
</tr>
</tbody>
</table>

### Stack Before

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08041090</td>
<td>0x00001234</td>
</tr>
<tr>
<td>0x0804108C</td>
<td>call getchar 0x0BF0F123</td>
</tr>
<tr>
<td>0x08041088</td>
<td>jmp 0x0804107C</td>
</tr>
<tr>
<td>0x08041084</td>
<td>0xBFFFF0121</td>
</tr>
<tr>
<td>0x08041080</td>
<td>0x00000000</td>
</tr>
<tr>
<td>esp   → 0x0804107C</td>
<td>ret</td>
</tr>
</tbody>
</table>

### Stack After

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08041090</td>
<td>0x08041090</td>
</tr>
<tr>
<td>0x0804108C</td>
<td>0x0804108C</td>
</tr>
<tr>
<td>0x08041088</td>
<td>call getchar</td>
</tr>
<tr>
<td>0x08041084</td>
<td>jmp 0x0804107C</td>
</tr>
<tr>
<td>0x08041080</td>
<td>0xBFFFF0121</td>
</tr>
<tr>
<td>0x08041080</td>
<td>0x00000000</td>
</tr>
<tr>
<td>esp   → 0x0804107C</td>
<td>ret</td>
</tr>
<tr>
<td>esp   → 0x08041078</td>
<td>0x08041088</td>
</tr>
</tbody>
</table>
x86 - ret

Opcode: 0xC3

Description: Returns control to previous called function. Pops address on top of stack into eip.

Our use: Used to return from called functions. Used extensively in return oriented programming (roping) to build a chain of instructions.
### Registers Before

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td>0x0804107C</td>
</tr>
<tr>
<td>eip</td>
<td>0x0804107C</td>
</tr>
<tr>
<td>ebp</td>
<td>0x08041090</td>
</tr>
</tbody>
</table>

### Registers After

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td>0x0804107C</td>
</tr>
<tr>
<td>eip</td>
<td>0x08041094</td>
</tr>
<tr>
<td>ebp</td>
<td>0x08041090</td>
</tr>
</tbody>
</table>

### Stack Before

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08041094</td>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>ebp  → 0x08041090</td>
<td>0x00001234</td>
</tr>
<tr>
<td>0x0804108C</td>
<td>call getchar 0xBF0F123</td>
</tr>
<tr>
<td>0x08041088</td>
<td>jmp 0x0804107C</td>
</tr>
<tr>
<td>0x08041084</td>
<td>0xBFFF0121</td>
</tr>
<tr>
<td>0x08041080</td>
<td>0x00000000</td>
</tr>
<tr>
<td>esp  → 0x0804107C</td>
<td>ret</td>
</tr>
</tbody>
</table>

### Stack After

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08041094</td>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>ebp  → 0x08041090</td>
<td>0x00001234</td>
</tr>
<tr>
<td>0x0804108C</td>
<td>call getchar 0xBF0F123</td>
</tr>
<tr>
<td>0x08041088</td>
<td>jmp 0x0804107C</td>
</tr>
<tr>
<td>0x08041084</td>
<td>0xBFFF0121</td>
</tr>
<tr>
<td>0x08041080</td>
<td>0x00000000</td>
</tr>
<tr>
<td>esp  → 0x0804107C</td>
<td>ret</td>
</tr>
</tbody>
</table>
x86 - leave

Opcode: 0xC9

Description: leave is used as part of function cleanup/function epilogue. It is equivalent to mov esp, ebp pop ebp

Our use: As said above it is used at the end of a function call right before the ret instruction. Also used in exploitation to pivot the stack sometimes.
x86 - lea

Opcode: 0x8D

Description: lea or load effective address is generally used for pointer arithmetic and loading addresses of arrays, structs

Our use: Same as description
### Registers Before

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x00000000</td>
</tr>
<tr>
<td>ebx</td>
<td>0x08040102</td>
</tr>
<tr>
<td>ecx</td>
<td>0x00000003</td>
</tr>
</tbody>
</table>

### Registers After

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x08040105</td>
</tr>
<tr>
<td>ebx</td>
<td>0x08040102</td>
</tr>
<tr>
<td>ecx</td>
<td>0x00000003</td>
</tr>
</tbody>
</table>

```
lea eax, [ebx+ecx]
```
x86 - nop

Opcode: 0x90

Description: No Operation, effectively does nothing

Our use: Padding exploits when offsets aren’t exactly known
x86 - Stack Behavior

The following slides were created by xenokovah of opensecuritytraining and all creative commons licenses still apply.
General Stack Frame Operation

We are going to pretend that main() is the very first function being executed in a program. This is what its stack looks like to start with (assuming it has any local variables).

[Diagram of a stack frame with local variables and a main() frame with undefined values and ellipses at the bottom.]

Book p. 306
General Stack Frame Operation 2

When main() decides to call a subroutine, main() becomes “the caller”. We will assume main() has some registers it would like to remain the same, so it will save them. We will also assume that the callee function takes some input arguments.

<table>
<thead>
<tr>
<th>Local Variables</th>
<th>stack bottom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caller-Save Registers</td>
<td>main() frame</td>
</tr>
<tr>
<td>Arguments to Pass to Callee</td>
<td>undef</td>
</tr>
<tr>
<td></td>
<td>undef</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>stack top</td>
</tr>
</tbody>
</table>
General Stack Frame Operation 3

When main() actually issues the CALL instruction, the return address gets saved onto the stack, and because the next instruction after the call will be the beginning of the called function, we consider the frame to have changed to the callee.

<table>
<thead>
<tr>
<th>Local Variables</th>
<th>stack bottom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caller-Save Registers</td>
<td>main() frame</td>
</tr>
<tr>
<td>Arguments to Pass to Callee</td>
<td>undef</td>
</tr>
<tr>
<td>Caller's saved return address</td>
<td>undef</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

stack top
General Stack Frame Operation 4

When foo() starts, the frame pointer (ebp) still points to main()'s frame. So the first thing it does is to save the old frame pointer on the stack and set the new value to point to its own frame.
General Stack Frame Operation 5

Next, we'll assume the the callee foo() would like to use all the registers, and must therefore save the callee-save registers. Then it will allocate space for its local variables.

<table>
<thead>
<tr>
<th>Local Variables</th>
<th>stack bottom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caller-Save Registers</td>
<td>main() frame</td>
</tr>
<tr>
<td>Arguments to Pass to Callee</td>
<td>foo()'s frame</td>
</tr>
<tr>
<td>Caller's saved return address</td>
<td>undef</td>
</tr>
<tr>
<td>Saved Frame Pointer</td>
<td>...</td>
</tr>
<tr>
<td>Callee-Save Registers</td>
<td>stack top</td>
</tr>
<tr>
<td>Local Variables</td>
<td>5</td>
</tr>
</tbody>
</table>
General Stack Frame Operation 6

At this point, foo() decides it wants to call bar(). It is still the callee-of-main(), but it will now be the caller-of-bar. So it saves any caller-save registers that it needs to. It then puts the function arguments on the stack as well.

![Diagram of stack frame operation]

- Saved Frame Pointer
- Callee-Save Registers
- Local Variables
- Caller-Save Registers
- Arguments to Pass to Callee

stack bottom

<table>
<thead>
<tr>
<th>main() frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo()'s frame</td>
</tr>
<tr>
<td>undef</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

stack top
General Stack Frame Layout

Every part of the stack frame is technically optional (that is, you can hand code asm without following the conventions.) But compilers generate code which uses portions if they are needed. Which pieces are used can sometimes be manipulated with compiler options. (E.g. omit frame pointers, changing calling convention to pass arguments in registers, etc.)
Stack Frames are a Linked List!

The ebp in the current frame points at the saved ebp of the previous frame.

```
stack bottom

main() frame

foo()'s frame

bar()'s frame

...

stack top
```
Example1.c

The stack frames in this example will be very simple. Only saved frame pointer (ebp) and saved return addresses (eip).

```c
//Example1 - using the stack
//to call subroutines
//New instructions:
//push, pop, call, ret, mov
int sub(){
    return 0xbeef;
}

text main(){
    sub();
    return 0xf00d;
}
```
Example1.c 1:
EIP = 00401010, but no instruction yet executed

```
sub:
00401000  push  ebp
00401001  mov  ebp,esp
00401003  mov  eax,0BEEFh
00401008  pop  ebp
00401009  ret
main:
00401010  push  ebp
00401011  mov  ebp,esp
00401013  call  sub (401000h)
00401018  mov  eax,0F00Dh
0040101D  pop  ebp
0040101E  ret
```

Belongs to the frame *before* main() is called
Example 1.c 2

<table>
<thead>
<tr>
<th>eax</th>
<th>0x003435C0 ⚫</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FFB8 ⚫</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF68 ⚫</td>
</tr>
</tbody>
</table>

Key:
- ⚫ executed instruction,
- ⚫ modified value
- ✶ start value

sub:
- 00401000 push ebp
- 00401001 mov ebp, esp
- 00401003 mov eax, 0BEEFh
- 00401008 pop ebp
- 00401009 ret

main:
- 00401010 push ebp  ⚫
- 00401011 mov ebp, esp
- 00401013 call sub (401000h)
- 00401018 mov eax, 0F00Dh
- 0040101D pop ebp
- 0040101E ret
Example1.c 3

<table>
<thead>
<tr>
<th>eax</th>
<th>0x003435C0 ✘</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FF68 ☹</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF68</td>
</tr>
</tbody>
</table>

sub:
00401000 push ebp
00401001 mov ebp,esp
00401003 mov eax,0BEF0h
00401008 pop ebp
00401009 ret

main:
00401010 push ebp
00401011 mov ebp,esp ✘
00401013 call sub (401000h)
00401018 mov eax,0F00Dh
0040101D pop ebp
0040101E ret

Key:
- ☒ executed instruction
- ☹ modified value
- ✘ start value

0x0012FF6C 0x004012E8 ✘
0x0012FF68 0x0012FFB8
def
0x0012FF64 undef
0x0012FF60 undef
0x0012FF5C undef
0x0012FF58 undef

4
Example1.c 4

<table>
<thead>
<tr>
<th>eax</th>
<th>0x003435C0 ✗</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FF68</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF64 ✷</td>
</tr>
</tbody>
</table>

**Key:**
- ✗ executed instruction,
- ✷ modified value
- ✱ start value

**sub:**
- 0x00401000 push ebp
- 0x00401001 mov ebp, esp
- 0x00401003 mov eax, 0BEEFH
- 0x00401008 pop ebp
- 0x00401009 ret

**main:**
- 0x00401010 push ebp
- 0x00401011 mov ebp, esp
- 0x00401013 call sub (401000h) ✱
- 0x00401018 mov eax, 0F00Dh
- 0x0040101D pop ebp
- 0x0040101E ret

**Key:**
- 0x0012FF6C
- 0x0012FF68
- 0x0012FF64
- 0x0012FF60
- 0x0012FF5C
- 0x0012FF58
- 0x004012E8 ✗
- 0x004012F8
- 0x00401018 ✷
- 0x00401018
- 0x00401018
- undef
- undef
- undef
- 5
### Example1.c 5

<table>
<thead>
<tr>
<th>eax</th>
<th>0x003435C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FF68</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key:</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
</tr>
<tr>
<td>✧</td>
</tr>
<tr>
<td>✶</td>
</tr>
</tbody>
</table>

#### sub:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>00401000</td>
<td>push</td>
<td>ebp ✗</td>
</tr>
<tr>
<td>00401001</td>
<td>mov</td>
<td>ebp,esp</td>
</tr>
<tr>
<td>00401003</td>
<td>mov</td>
<td>eax,0BEEFh</td>
</tr>
<tr>
<td>00401008</td>
<td>pop</td>
<td>ebp</td>
</tr>
<tr>
<td>00401009</td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>

#### main:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>00401010</td>
<td>push</td>
<td>ebp</td>
</tr>
<tr>
<td>00401011</td>
<td>mov</td>
<td>ebp,esp</td>
</tr>
<tr>
<td>00401013</td>
<td>call</td>
<td>sub (401000h)</td>
</tr>
<tr>
<td>00401018</td>
<td>mov</td>
<td>eax,0F00Dh</td>
</tr>
<tr>
<td>0040101D</td>
<td>pop</td>
<td>ebp</td>
</tr>
<tr>
<td>0040101E</td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>
Example1.c 6

<table>
<thead>
<tr>
<th>eax</th>
<th>0x003435C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FF60</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF60</td>
</tr>
</tbody>
</table>

**Key:**
- ✗ executed instruction,
- 📅 modified value
- ✈️ start value

**sub:**
- 0x00401000 push ebp
- 0x00401001 mov ebp,esp ✗
- 0x00401003 mov eax,0BEEFh
- 0x00401008 pop ebp
- 0x00401009 ret

**main:**
- 0x00401010 push ebp
- 0x00401011 mov ebp,esp
- 0x00401013 call sub (401000h)
- 0x00401018 mov eax,0F00Dh
- 0x0040101D pop ebp
- 0x0040101E ret
Example1.c 6
STACK FRAME TIME OUT

```
sub    ebp
mov    ebp, esp
mov    eax, 0BEEFH
pop    ebp
retn
main
push   ebp
mov    ebp, esp
call   _sub
mov    eax, 0F00Dh
pop    ebp
retn
```

"Function-before-main"'s frame

main's frame
(saved frame pointer and saved return address)

sub's frame
(only saved frame pointer, because it doesn't call anything else, and doesn't have local variables)
Example1.c 7

<table>
<thead>
<tr>
<th>eax</th>
<th>0x0000BEEF</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FF60</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF60</td>
</tr>
</tbody>
</table>

Key:
- ✗ executed instruction,
- ⬤ modified value
- ✄ start value

```
sub:
0x00401000 push ebp
0x00401001 mov ebp, esp
0x00401003 mov eax, 0BEEFh ✗
0x00401008 pop ebp
0x00401009 ret

main:
0x00401010 push ebp
0x00401011 mov ebp, esp
0x00401013 call sub (0x401000h)
0x00401018 mov eax, 0F00Dh
0x0040101D pop ebp
0x0040101E ret

0x0012FF6C ✄
0x0012FF68      ✗
0x0012FF64      ⬤
0x0012FF60      ⬤
0x0012FF5C      undef
0x0012FF58      undef
```
Example1.c 8

<table>
<thead>
<tr>
<th>eax</th>
<th>0x0000BEEF</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FF68</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF64</td>
</tr>
</tbody>
</table>

sub:
00401000 push ebp
00401001 mov ebp, esp
00401003 mov eax, 0BEEFh
00401008 pop ebp
00401009 ret

main:
00401010 push ebp
00401011 mov ebp, esp
00401013 call sub (401000h)
00401018 mov eax, 0F00Dh
0040101D pop ebp
0040101E ret

Key:
- ✗ executed instruction,
- ✔ modified value
- ✗ start value

```
0x0012FF6C  0x004012E8 ✗
0x0012FF68   0x0012FFB8
0x0012FF64   0x00401018
0x0012FF60
0x0012FF5C
0x0012FF58
```

10
Example1.c 9

<table>
<thead>
<tr>
<th>eax</th>
<th>0x0000BEEF</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FF68</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF68</td>
</tr>
</tbody>
</table>

**Key:**
- ❌ executed instruction
- m modified value
- ※ start value

**Sub:**
- 0x00401000 push ebp
- 0x00401001 mov ebp,esp
- 0x00401003 mov eax,0BEEFH
- 0x00401008 pop ebp
- 0x00401009 ret ❌

**Main:**
- 0x00401010 push ebp
- 0x00401011 mov ebp,esp
- 0x00401013 call sub (401000h)
- 0x00401018 mov eax,0F00DH
- 0x0040101D pop ebp
- 0x0040101E ret
Example1.c 9

<table>
<thead>
<tr>
<th>eax</th>
<th>0x0000F00D</th>
<th>ebp</th>
<th>0x012FF68</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td>0x012FF68</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Key:
- ✗ executed instruction,
- ▮ modified value
- ☣ start value

sub:
00401000 push ebp
00401001 mov ebp, esp
00401003 mov eax, 0BEEFH
00401008 pop ebp
00401009 ret

main:
00401010 push ebp
00401011 mov ebp, esp
00401013 call sub (401000h)
00401018 mov eax, 0F00Dh ✗
0040101D pop ebp
0040101E ret
Example 1.c 10

<table>
<thead>
<tr>
<th>eax</th>
<th>0x0000F00D</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FFB8 ⬤</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF6C ⬤</td>
</tr>
</tbody>
</table>

Key:
- ▼ executed instruction
- ⬤ modified value
- ⬤ start value

```
sub:
00401000 push ebp
00401001 mov ebp, esp
00401003 mov eax, 0BEFh
00401008 pop ebp
00401009 ret

main:
00401010 push ebp
00401011 mov ebp, esp
00401013 call sub (401000h)
00401018 mov eax, 0F00Dh
0040101D pop ebp ▼
0040101E ret
```

Diagram:
```
0x004012E8 ▼
0x0012FF6C
0x0012FF68
0x0012FF64
0x0012FF60
0x0012FF5C
0x0012FF58
```

```
13
```
Example1.c 11

<table>
<thead>
<tr>
<th>eax</th>
<th>0x0000F00D</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp</td>
<td>0x0012FFB8</td>
</tr>
<tr>
<td>esp</td>
<td>0x0012FF70</td>
</tr>
</tbody>
</table>

**Key:**
- ✗ executed instruction,
- ☛ modified value
- ☼ start value

```
sub:
00401000 push    ebp
00401001 mov ebp,esp
00401003 mov eax,0BEEFh
00401008 pop ebp
00401009 ret

main:
00401010 push    ebp
00401011 mov ebp,esp
00401013 call sub (401000h)
00401018 mov eax,0F00Dh
0040101D pop ebp
0040101E ret ✗
```

Execution would continue at the value ret removed from the stack: 0x004012E8
Example 1 Notes

- sub() is deadcode - its return value is not used for anything, and main always returns 0xF00D. If optimizations are turned on in the compiler, it would remove sub()
- Because there are no input parameters to sub(), there is no difference whether we compile as cdecl vs stdcall calling conventions
x86 - So Much Information

BOY,

THAT ESCALATED QUICKLY
Below is a simple C program, I want you to draw a stack frame of it and if you think you can, the corresponding assembly.

```c
int main() {
    int a = 0;
    int b = 1;
    return a+b;
}
```
x86 - Exercise 1 - Lets Draw Stacks

*assuming main is first function called*
push ebp // save old frame pointer
mov ebp, esp // start new stack frame
sub esp, 0x8
mov dword ptr [ebp-4], 0 // a = 0
mov dword ptr [ebp-8], 1 // b = 1
mov ebx, [ebp-4]
mov eax, [ebp-8]
add eax, ebx // a += b
leave // mov esp, ebp & pop ebp
ret // pop eip
x86 - Exercise 2 - Lets Draw Stacks

Below is a simple C program, I want you to draw a stack frame of it and if you think you can, the corresponding assembly.

```c
int getAnswer(int first, int last) {
    int num = 6;
    return (num + first) - last;
}

int main() {
    int a = 0;
    int b = 1;
    int answer = getAnswer(a,b);
    return answer;
}
```
```assembly
getAnswer()
push ebp
mov ebp, esp
sub esp, 0x4
mov dword ptr [ebp-4], 6 // num = 6
mov ecx, [ebp-0x4]
mov eax, [ebp+0x8] // first
add eax, ecx // first + num
mov ecx, [ebp+0xC] // last
sub eax, ecx // (first+num) - last
leave
ret

main()
push ebp // save old frame pointer
mov ebp, esp // start new stack frame
sub esp, 0x8
mov dword ptr [ebp-0x8], 1 // b = 1
mov eax, [ebp-0x8]
push eax
mov dword ptr [ebp-0x4], 0 // a = 0
mov ebx, [ebp-0x4]
push ebx
call getAnswer
add esp, 0x8 // fix the stack
leave // mov esp, ebp & pop ebp
ret // pop eip
```

| saved ebp for getAnswer
| saved ebp
| saved eip for main
| saved eip

| num
| b
| a
x86 - Summary

- Don’t worry about learning all the instructions.
- Most important thing is understanding how stack frames work.
- Goal should be able to understand assembly well enough to draw stack frames for simple C programs.