



IA-32 Architecture

CS 4430/7430 Compiler Construction

Intel x86 Architecture

- ▶ Security professionals constantly analyze assembly language code
- ▶ Many exploits are written in assembly
- ▶ Source code for applications and malware is not available in most cases
- ▶ We cover only the modern 32-bit view of the x86 architecture
- ▶ Why go through this in 4430/7430?
 - ▶ Introduction to Implementing Functions
 - ▶ Security Dimension to the Course
 - ▶ Return-oriented Programming
 - ▶ Buffer overflows & Stack Smashing

x86 Primer

- ▶ CISC architecture

- ▶ Lots of instructions and addressing modes
- ▶ Operands can be taken from memory
- ▶ Instructions are variable length
 - ▶ Depends on operation
 - ▶ Depends on addressing modes

- ▶ Architecture manuals at:

<http://www.intel.com/products/processor/manuals/index.htm>

x86 Registers

- ▶ Eight 32-bit general registers:
 - ▶ EAX, EBX, ECX, EDX, ESI, EDI,
 - ▶ ESP (stack pointer),
 - ▶ EBP (base pointer, a.k.a. frame pointer)
- ▶ Names are not case-sensitive and are usually lower-case in assembly code (e.g. `eax`, `ecx`)

x86 Registers

- ▶ 8 general-purpose 32-bit registers
- ▶ ESP is the stack pointer; EBP is the frame pointer
- ▶ Not all registers can be used for all operations
 - ▶ Multiplication, division, shifting use specific registers

EAX	AH	AX	AL
EDX	DH	DX	DL
ECX	CH	CX	CL
EBX	BH	BX	BL
EBP	BP		
ESI	SI		
EDI	DI		
ESP	SP		

x86 Floating-point Registers

- ▶ Floating-point unit uses a stack
- ▶ Each register is 80-bits wide (doesn't use IEEE FP standard)

	79	78	64	63	0
R0	SIGN	EXPONENT	SIGNIFICAND		
R1					
R2					
R3					
R4					
R5					
R6					
R7					

x86 Instructions

- ▶ In MASM (Microsoft Assembler), the first operand is usually a destination, and the second operand is a source:

```
mov eax,ebx    ; eax := ebx
```

- ▶ Two-operand instructions are most common, in which first operand is both source and destination:

```
add eax,ecx    ; eax := eax + ecx
```

- ▶ Semicolon begins a comment

x86 Data Declarations

- ▶ Must be in a data section
- ▶ Give name, type, optional initialization:

`.DATA`

`count DW 0 ; 16-bit, initialized to 0`

`answer DD ? ; 32-bit, uninitialized`

- ▶ Can declare arrays:

`array1 DD 100 DUP(0) ; 100 32-bit values,
 ; initialized to zero`

x86 Memory Operations

- ▶ “lea” instruction means “load effective address:

```
lea  eax,[count]  ; eax := address of count
```

- ▶ Can move through an address pointer

```
lea  ebx,[count]  ; ebx := address of count
```

```
mov  [ebx],edx    ; count := edx
```

```
                ; ebx is a pointer
```

```
                ; [ebx] dereferences it
```

- ▶ We also will see the stack used as memory

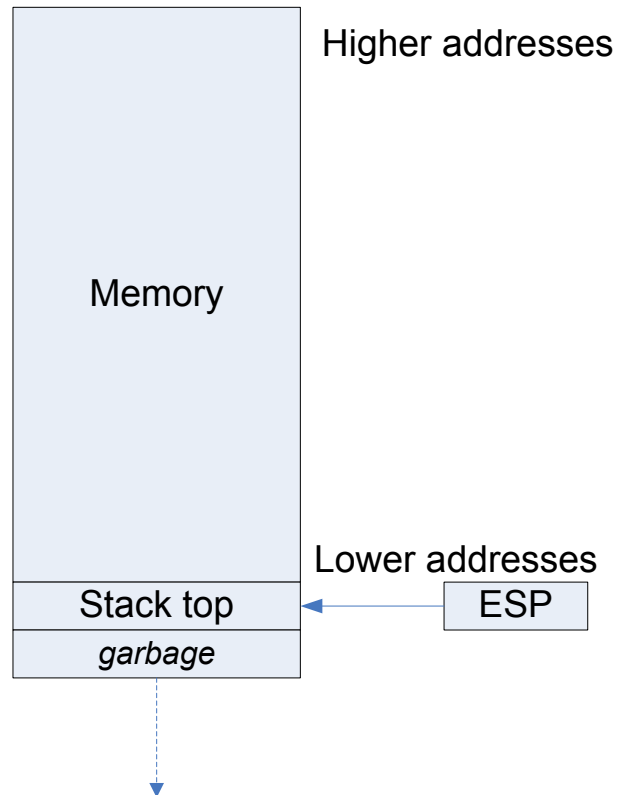
x86 Stack Operations

- ▶ The x86 stack is managed using the ESP (stack pointer) register, and specific stack instructions:
 1. `push ecx` ; push ecx onto stack
 2. `pop ebx` ; pop top of stack into register ebx
 3. `call foo` ; push address of next instruction on stack, then jump to label foo
 4. `ret` ; pop return address off stack, then jump to it

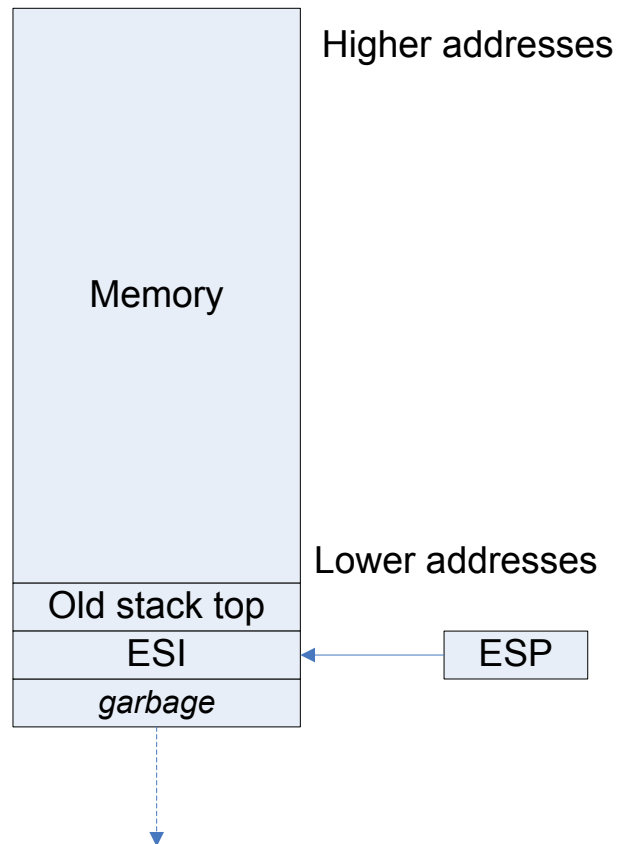
x86 Hardware Stack

- ▶ The x86 stack grows downward in memory addresses
- ▶ Decrementing ESP increases stack size;
 - ▶ incrementing ESP reduces it

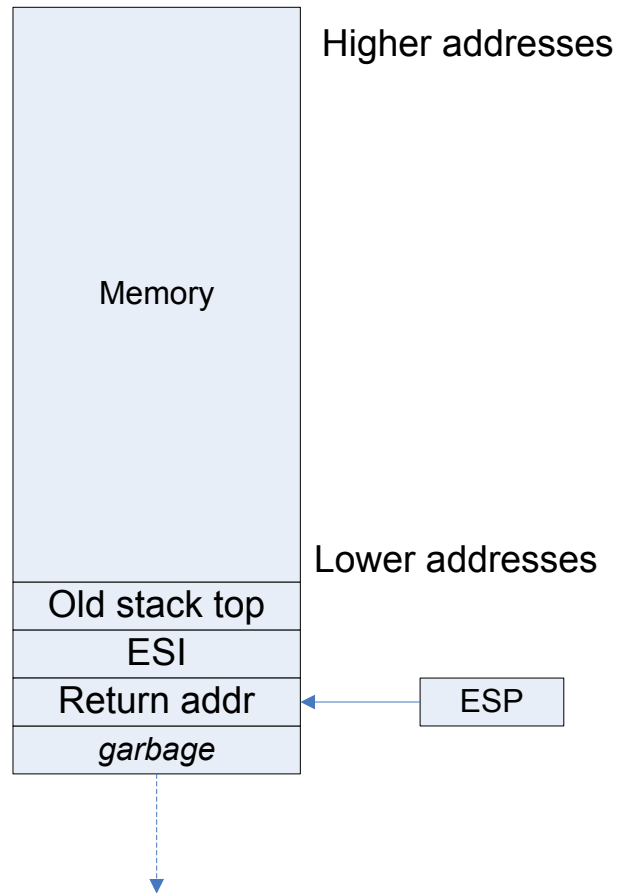
x86 Hardware Stack



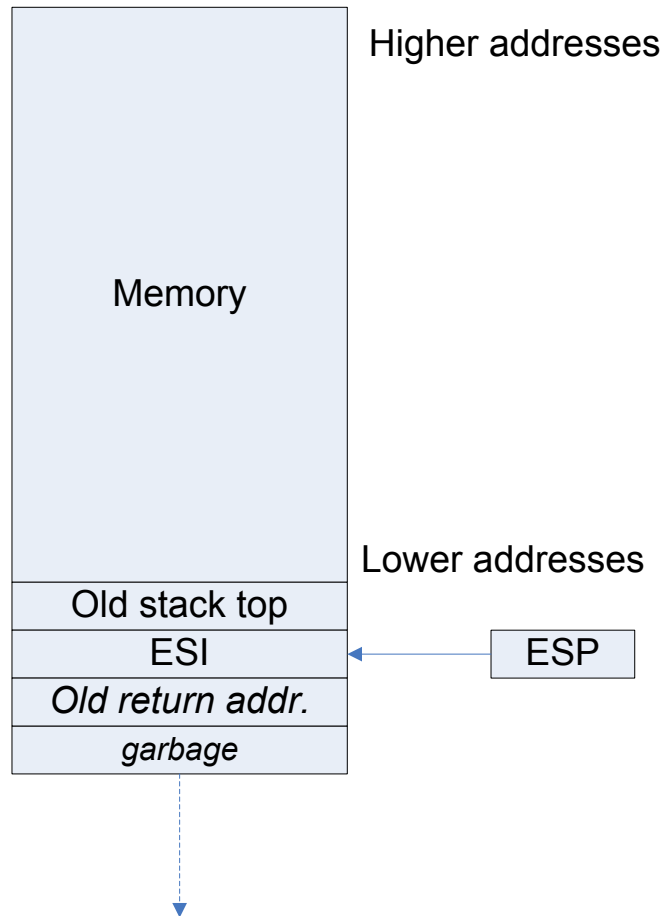
x86 Stack after "push ESI"



x86 Stack after call



x86 Stack after `ret`



x86 C Calling Convention

- ▶ A **calling convention** is an *agreement* among software designers
 - ▶ (e.g. of compilers, compiler libraries, assembly language programmers) on how to use registers and memory in subroutines
 - ▶ NOT enforced by hardware!
- ▶ Allows software pieces to interact compatibly,
 - ▶ e.g. a C function can call an ASM function, and vice versa

C Calling Convention cont.

- Questions answered by a calling convention:
 1. How are parameters passed?
 2. How are values returned?
 3. Where are local variables stored?
 4. Which registers must the caller save before a call, and which registers must the callee save if it uses them?

How Are Parameters Passed?

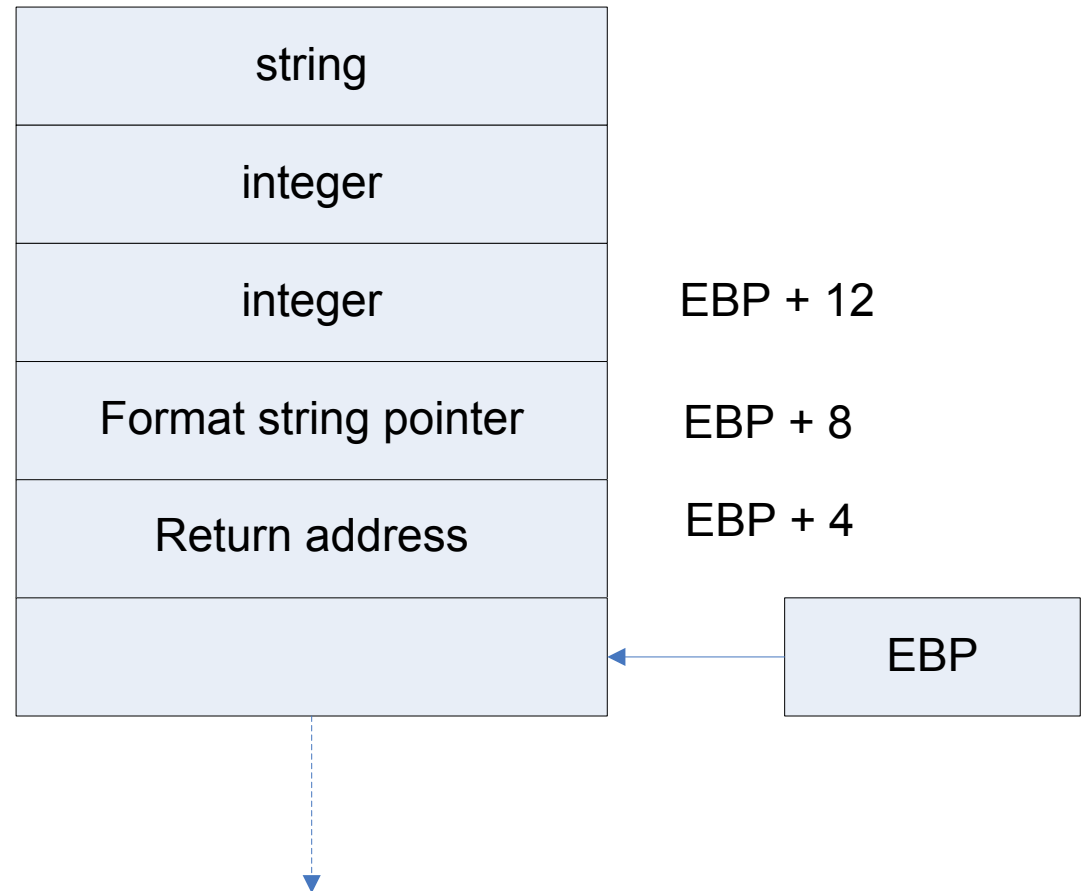
- Most machines use registers, because they are faster than memory
 - x86 has too few registers to do this
- Therefore, the stack must be used to pass parameters
- Parameters are pushed onto the stack in reverse order

Why Pass Parameters in Reverse Order?

- Some C functions have a variable number of parameters
 - First parameter determines the number of remaining parameters!
- Example: `printf("%d %d %s\n", ...);`
- `printf()` library function
 - reads first parameter, then
 - determines that the number of remaining parameters is 3

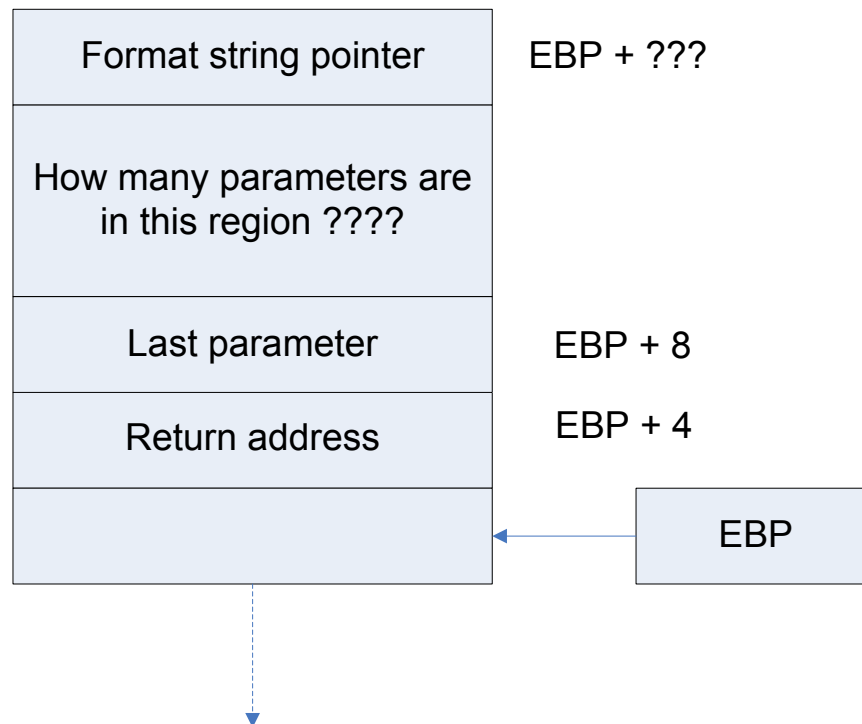
Reverse Order Parameters cont.

- **printf()** will always find the first parameter at $[EBP + 8]$



What if Parameter Order was NOT Reversed?

- **printf()** will always find the LAST parameter at **[EBP + 8]** ; not helpful



C Calling Convention cont.

- Questions answered by a calling convention:
 1. How are parameters passed?
 2. How are values returned?
 3. Where are local variables stored?
 4. Which registers must the caller save before a call, and which registers must the callee save if it uses them?

How are Values Returned?

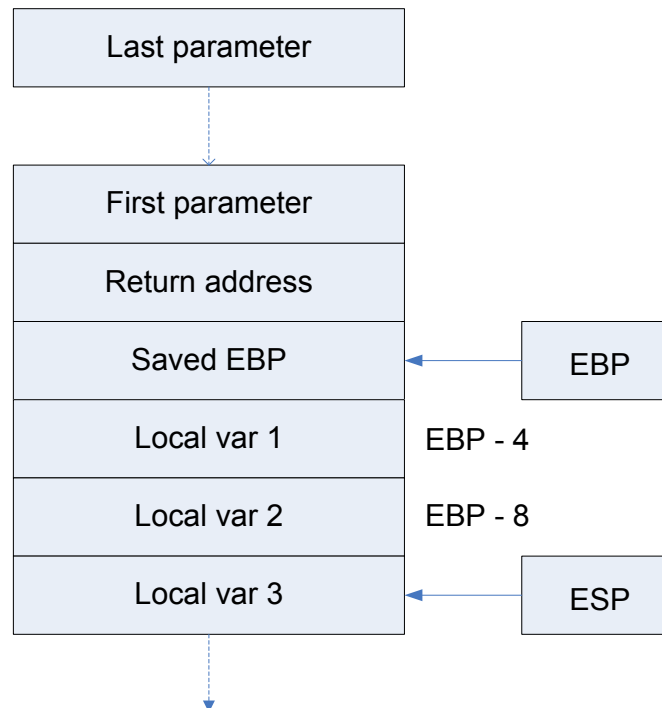
- Register `eax` contains the return value
- This means x86 can only return a 32-bit value from a function
- Smaller values are zero extended or sign extended to fill register `eax`
- If a programming language permits return of larger values (structures, objects, arrays, etc.),
 - a pointer to the object is returned in register `eax`

C Calling Convention cont.

- Questions answered by a calling convention:
 1. How are parameters passed?
 2. How are values returned?
 3. Where are local variables stored?
 4. Which registers must the caller save before a call, and which registers must the callee save if it uses them?

Where are Local Variables Stored?

- **Stack frame** for the currently executing function is between where EBP and ESP point in the stack



C Calling Convention cont.

- Questions answered by a calling convention:
 1. How are parameters passed?
 2. How are values returned?
 3. Where are local variables stored?
 4. Which registers must the caller save before a call, and which registers must the callee save if it uses them?

Who Saves Which Registers?

- It is efficient to have the caller save some registers before the call, leaving others for the callee to save
- x86 only has 8 general registers; 2 are used for the stack frame (ESP and EBP)
- The other 6 are split between callee-saved (ESI, EDI) and caller-saved
- Remember: Just a convention, or agreement, among software designers

What Does the Caller Do?

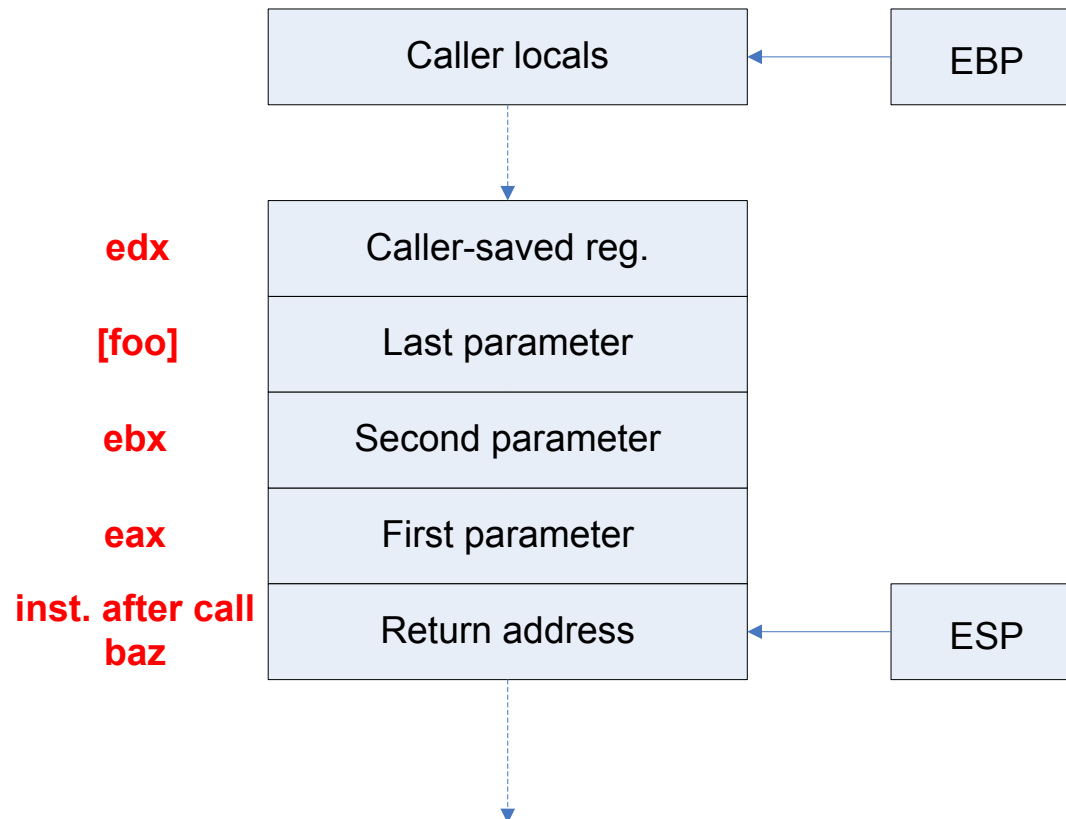
- Example: Call a function and pass 3 integer parameters to it

```
push edx      ; caller-saved register
push [foo]    ; Var foo is last parameter
push ebx      ; ebx is second parameter
push eax      ; eax is first parameter
call baz      ; push return address, jump
add esp,12    ; toss old parameters
pop edx       ; restore caller-saved edx
              ; eax holds return value
```

- eax, ebx did not need to be saved here

Stack after Call

- x86 stack immediately after `call baz`



Callee Stack Frame Setup

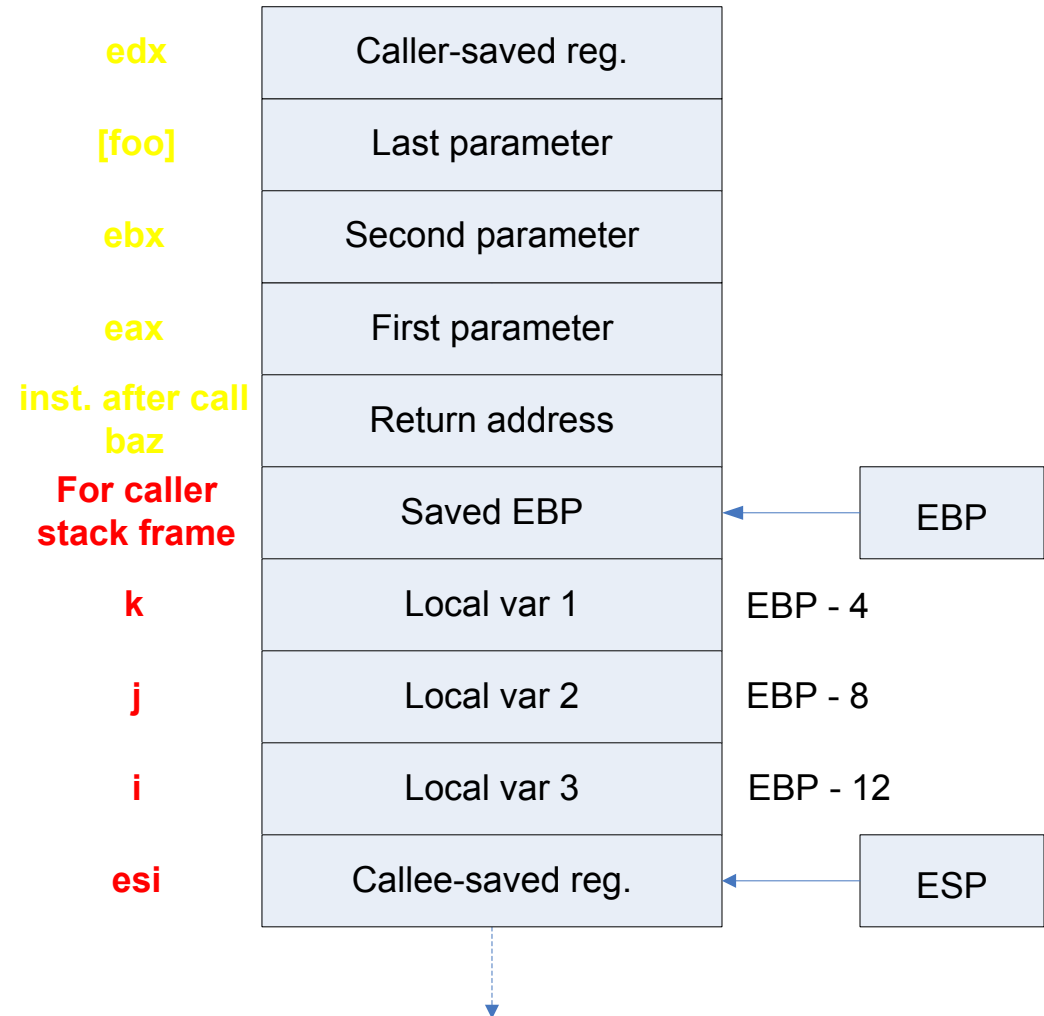
- The standard subroutine prologue code sets up the new stack frame:

```
; Prologue code at top of function
push ebp          ; save old base pointer
move ebp,esp      ; Set new base pointer
sub esp,12        ; Make room for locals
push esi          ; Func uses ESI, so save
:
:
```

This code sets up the stack frame of the callee

Stack After Prologue Code

- After the prologue code sets up the new stack frame:



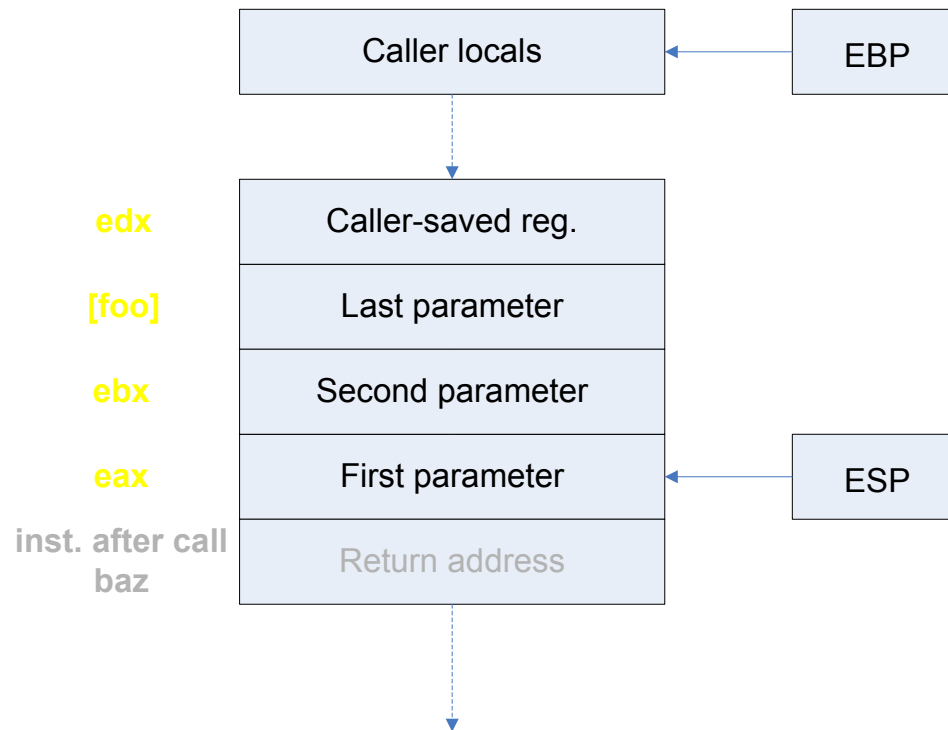
Callee Stack Frame Cleanup

- ▶ Epilogue code at end cleans up frame (mirror image of prologue):

```
    ; Epilogue code at bottom of function  
pop esi           ; Restore callee-saved ESI  
move esp,ebp      ; Deallocate stack frame  
pop ebp          ; Restore caller's EBP  
ret              ; return
```


Stack After Return

► After epilogue code and return:



Caller Stack Cleanup

- ▶ After the return, caller has a little cleanup code:

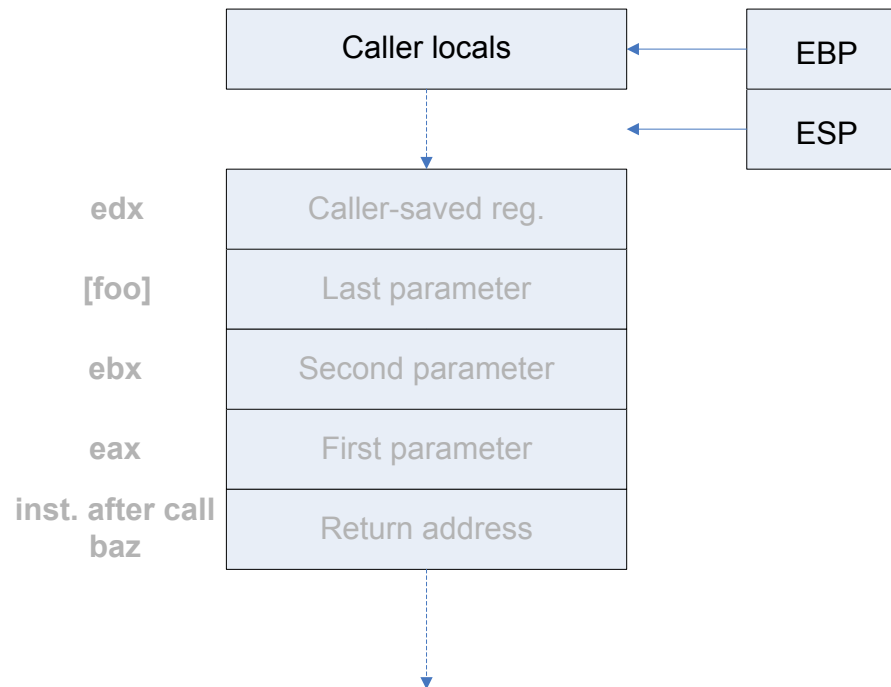
```
add esp,12    ; deallocate parameter space
pop edx       ; restore caller-saved register
```

Today

- ▶ Finish covering x86 background
- ▶ Reading Assignment
 - ▶ Szor, Chapter 2 (if you haven't already)
 - ▶ “Smashing the Stack for Fun and Profit”
- ▶ We will cover some details of the PE file format
 - ▶ Szor, pp. 160-172, section 4.3.2.1, describes PE format
 - ▶ Pay special attention to pp. 163-165, where the fields of interest to virus creators are discussed

Caller Stack After Cleanup

- ▶ After the caller's cleanup code:



Register Save Question

- ▶ Why would it be less efficient to have all registers be callee-saved, rather than splitting the registers into caller-saved and callee-saved? (Just think of one example of inefficiency.)

Call Stack: Virus Implications

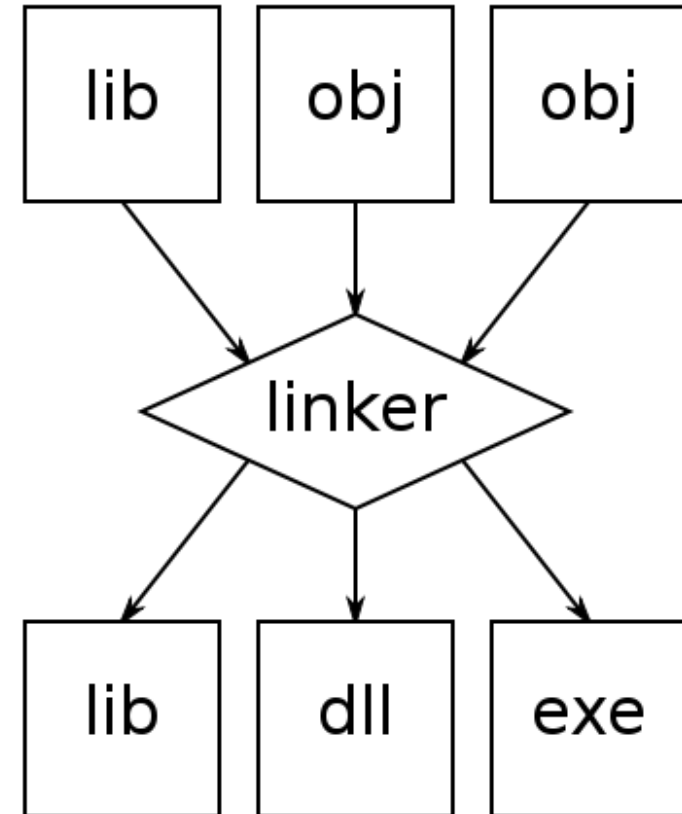
- ▶ The return address is a primary target of malware
- ▶ If the malware can change the return address on the stack, it can cause control to pass to malware code
- ▶ We saw an example with buffer overflows

x86 Executable Files

- ▶ The standard format of a *.exe file, produced by compiling and linking, is the PE (Portable Executable) file
- ▶ Also called PE32 (because it is 32-bit code); newer format is PE64, and PE32+ is a transitional format
- ▶ Older formats exist for 16-bit DOS and Windows 3.1
- ▶ We will stick to the PE32 format, calling it PE for brevity

Linker

- A **linker** is a program that
- takes one or more object files generated by a compiler and
 - combines them into a single executable file, library file, or another object file.

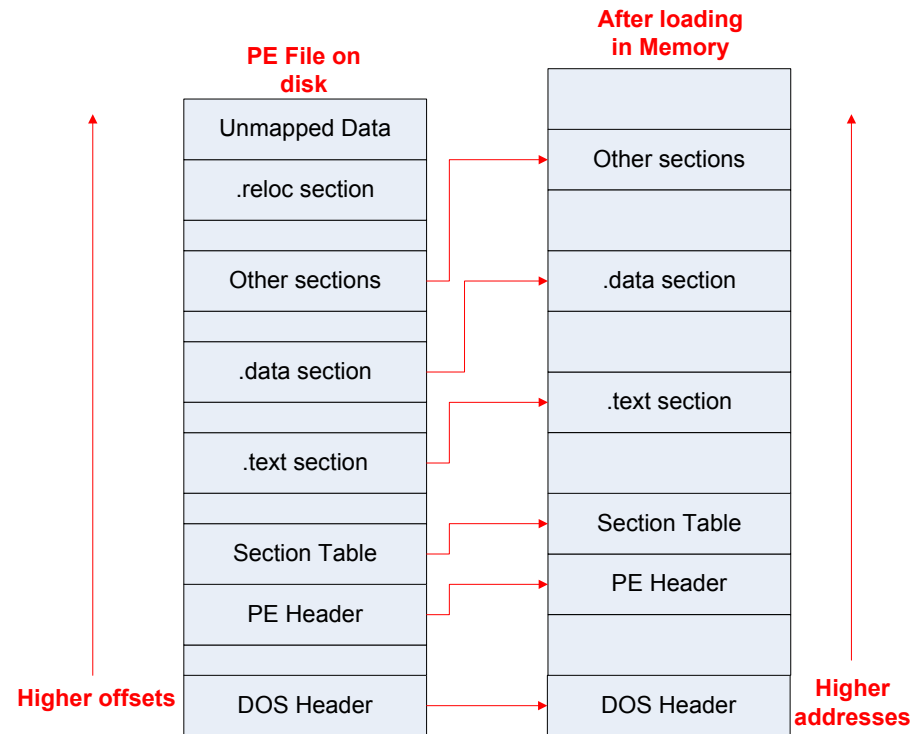


Loader

- ▶ **Loader:** the part of an OS responsible for loading programs and libraries into memory.
- ▶ **Loading:** an essential stage in the process of starting a program, as it places programs into memory and prepares them for execution.
- ▶ **Loading a program** involves
 - ▶ reading the contents of the executable file containing the program instructions into memory, and then
 - ▶ carrying out other required tasks to initialize the executable for running.
- ▶ Once loading complete, OS passes control to the loaded program code.

PE File Format

- ▶ Important to know how to analyze PE files when analyzing malware
- ▶ Overview:

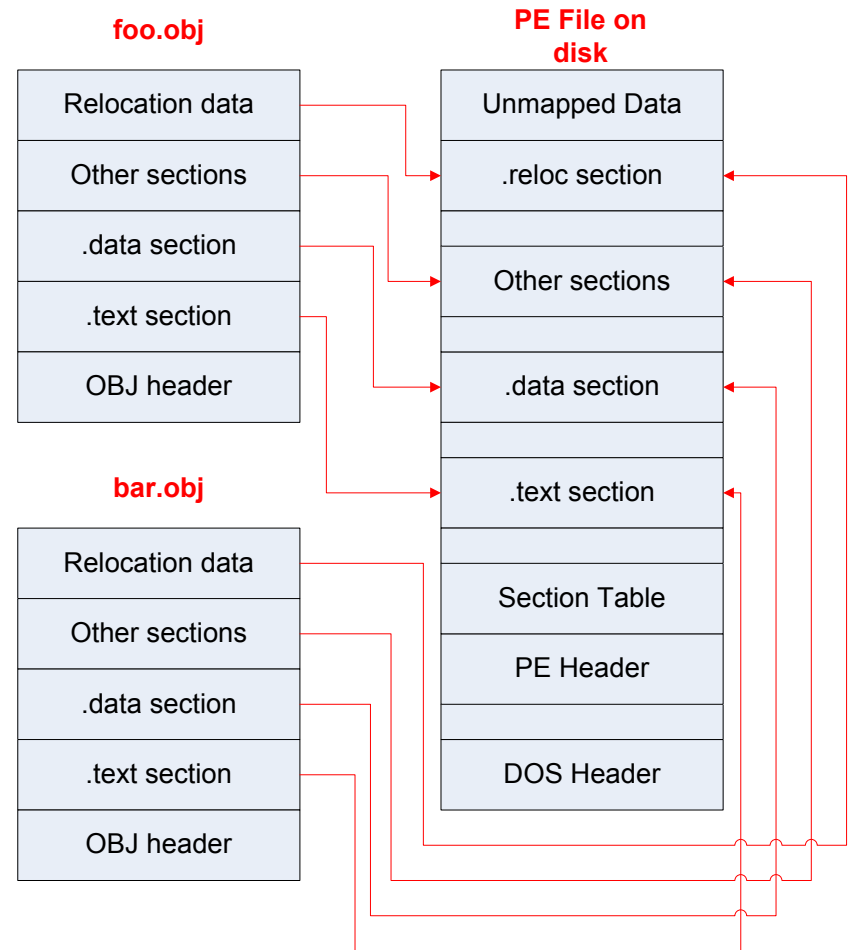


PE File Format

- ▶ **Why the dead spaces?**
 - ▶ Alignment restrictions
 - ▶ Loader increases dead spaces to use page boundaries (4KB), while alignment is to a lesser size (e.g. 128B) in the PE file on disk
- ▶ **Some linkers make PE file align to page boundaries**
 - ▶ Simplifies the loader's job
 - ▶ Make PE file bigger on disk

Producing a PE File

- ▶ Compiler produces multiple *.obj files
 - ▶ i.e., in case of separate compilation
- ▶ Linker groups like sections together and creates headers and section tables:



Detour: Motivation for Learning File Formats

- ▶ Question: Why do we care about the details of the PE file format?
- ▶ Answer: Because a virus writer will try to infect the PE file in such a way as to make the virus code execute, while making the PE file look as it would normally look. The job of anti-virus software is to find well-disguised viruses.

Next time

- ▶ Finish x86 slides
- ▶ Learn about “Obfuscated Tricky Jumps”

Virtual Addresses

- ▶ Addresses within *.obj and PE files are RVA (Relative Virtual Addresses)
- ▶ They are offsets relative to the eventual base address that the loader chooses at load time
- ▶ $VA \text{ (virtual address)} = \text{base address (load point for section)} + RVA$
- ▶ Physical address is wherever the VA is mapped by the OS to actual RAM
- ▶ Linker cannot know final VA, as loading has not happened yet; must deal with RVA

Loading the PE File

- ▶ OS provides kernel32.dll, which is linked with almost every PE file
 - ▶ Application might also make use of other DLLs, such as compiler libraries, etc.
 - ▶ Loader must ensure that all dependent DLLs are loaded and ready to use
 - ▶ Linker cannot know where in memory the library functions, etc., will be loaded
- ▶ **Therefore**, PE file code calls external API functions through function pointers

PE Function Pointers

- ▶ For each DLL from which the PE file imports (uses) API functions, the linker creates an IAT (Import Address Table) in the PE
 - ▶ The Import Address Table is a table of function pointers into another DLL
 - ▶ Function calls from your application to the DLL your application depends on are made through these function pointers
- ▶ Linker initializes the IAT to RVAs
- ▶ Loader fills in the virtual addresses at load time

PE Function Pointers Example

- ▶ Your C code: `call printf(...)`
- ▶ Compiler records in the OBJ header the need to import `printf()` from the DLL that contains `stdio`
- ▶ Compiler produces initial IAT for `stdio` in the OBJ header
- ▶ Linker merges IATs from all *.obj files
 - ▶ Offset (RVA) of `printf()` within `stdio` DLL is fixed and can be determined by the linker simply by looking at the `stdio` library object code

PE Function Pointers Example cont'd.

- ▶ Linker patches new IAT RVA for **printf()** into your object code:
 - ▶ `call dword ptr 0x41003768`
 - ▶ This is an indirect call through a pointer
- ▶ Address **0x41003768** is an IAT entry that will be filled in by the loader
- ▶ Loader replaces IAT entry with VA at load time; it knows where **stdio** DLL is loaded

Import Address Table

.idata name

25000 virtual address (00425000 to 00425B72)

C0000040 flags

Initialized Data

Read Write

Section contains the following imports:

KERNEL32.dll

425 IEC Import Address Table

110 GetCommandLineA

216 HeapFree

1E9 GetVersionExA

210 HeapAlloc

1A3 GetProcessHeap



Using the IAT

In the .text segment:

__GetCommandLineA@0:

00415C7E: FF 25 EC 51 42 00 jmp dword ptr

[__imp__GetCommandLineA@0]

__HeapFree@12:

00415C84: FF 25 F0 51 42 00 jmp dword ptr

[__imp__HeapFree@12]

__GetVersionExA@4:

00415C8A: FF 25 F4 51 42 00 jmp dword ptr

[__imp__GetVersionExA@4]

__HeapAlloc@12:

00415C90: FF 25 F8 51 42 00 jmp dword ptr

[__imp__HeapAlloc@12]

__GetProcessHeap@0:

00415C96: FF 25 FC 51 42 00 jmp dword ptr

[__imp__GetProcessHeap@0]



DOS Header

- ▶ If a program is invoked within a DOS command prompt window, it starts executing here
- ▶ For most PE32 executables, the DOS header contains a tiny executable that prints: “This application must be run from Windows”, then exits

PE Header

- ▶ DOS Header points to PE header
- ▶ PE header points to IATs and the section table, which points to all code and data sections
- ▶ Viruses use the PE Header to navigate around the PE file, find where to insert virus code, etc.

PE Sections

- ▶ Common sections are `.text` (for code), `.data` (read/write data), `.rdata` (read-only data), `.reloc` (relocation data used to build IATs)
- ▶ The attribute bits determine whether a section can be read, written, executed, etc., **NOT** the section name; viruses might modify the attribute bits so that a `.text` section becomes writable!
- ▶ Class web page links to more details on PE files

Analyzing PE Files

- ▶ DUMPBIN tool produces readable printout of a PE file
- ▶ **DUMPBIN /ALL /RAWDATA:NONE** is most common usage
- ▶ **/DISASM** switch also useful: disassembles the code sections
- ▶ Class web page links to more details on DUMPBIN tool

See you later

