IA-32 Stack and Calling Convention

Wei Wang

Why learn stack and calling convention

- Process stack is one of the most frequently attacked memory data structure
- Stack is primarily used for function calls.
- Calling convention defines how stack is used

Road Map

- x86 hardware stack
- C calling conventions
  - Stack pointers, return address
  - Function parameters and local variables
  - Return from a function
- Put it all together
- Reference Documents
- Summary

x86 Stack Operations

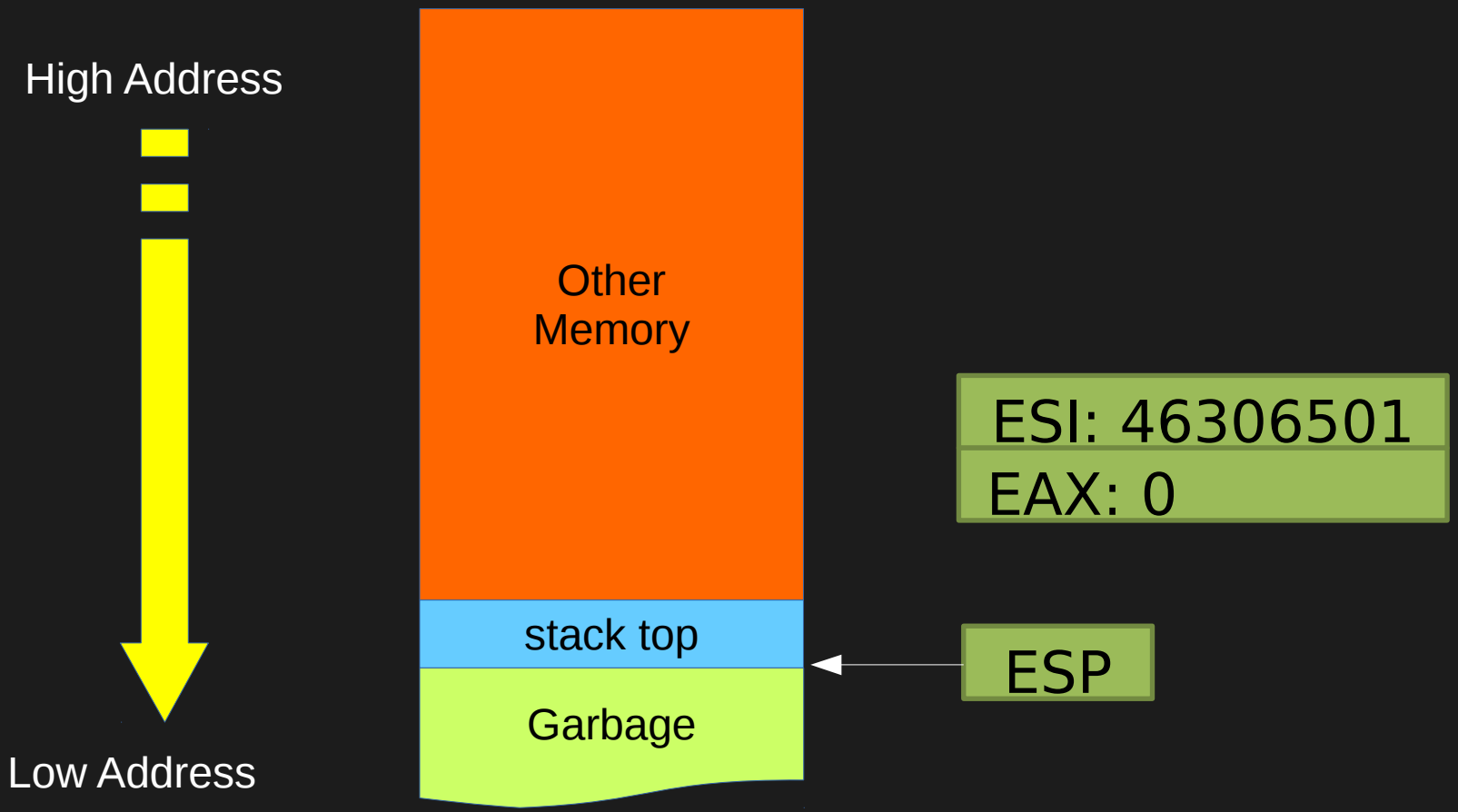
- The x86 stack is managed using the ESP (stack pointer) register, and specific stack instructions:

```
pushl   %ecx ; push ecx onto stack
popl    %ebx ; pop top of stack into register ebx
call    bar  ; push address of next instruction on
             ; stack, then jump to label bar
ret          ; pop return address off stack, and
             ; jump to it
```

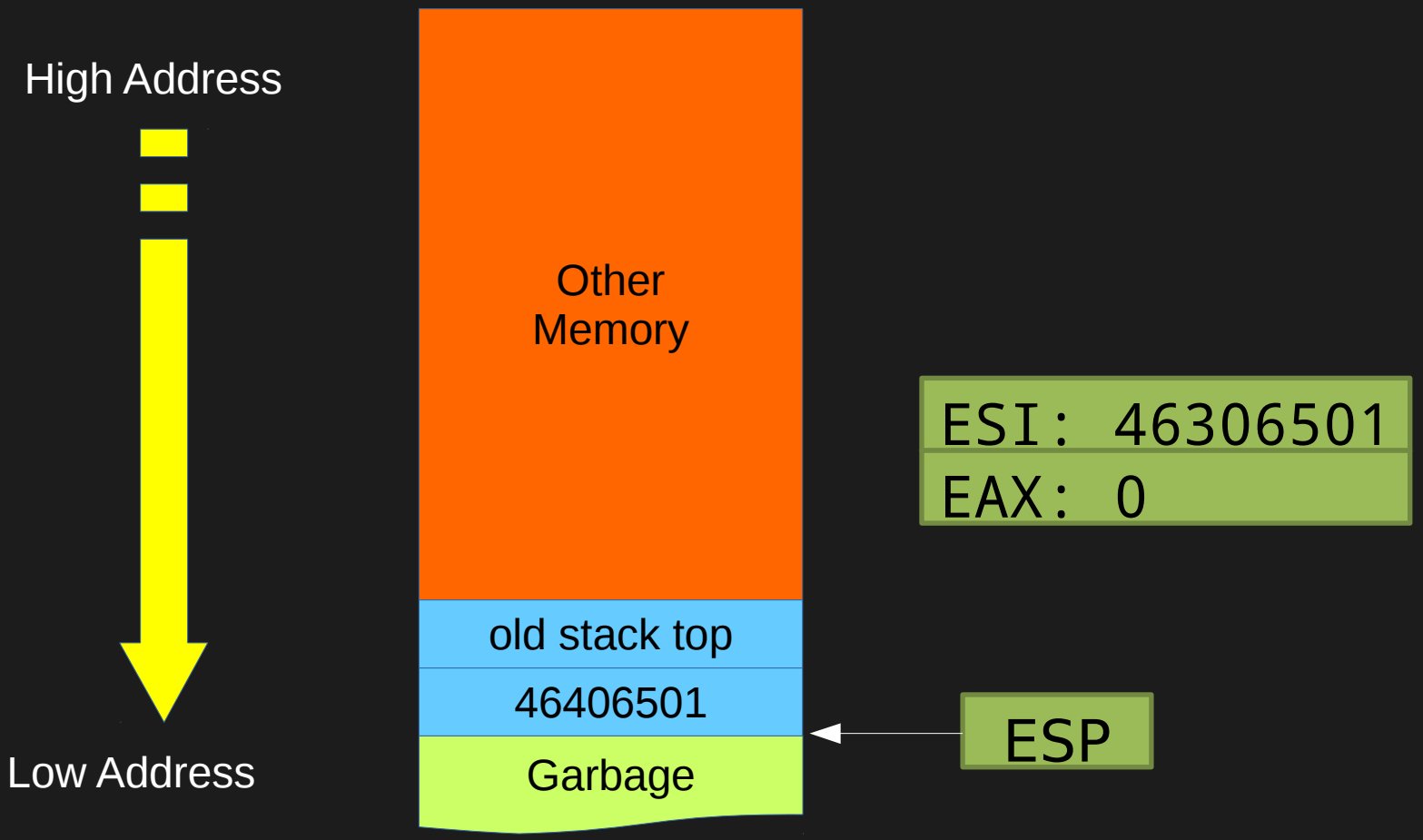
x86 Hardware Stack

- The x86 stack grows downward in memory addresses
- Register ESP is used to hold the memory address of the top of the stack
- Decrementing ESP increases stack size; incrementing ESP reduces it

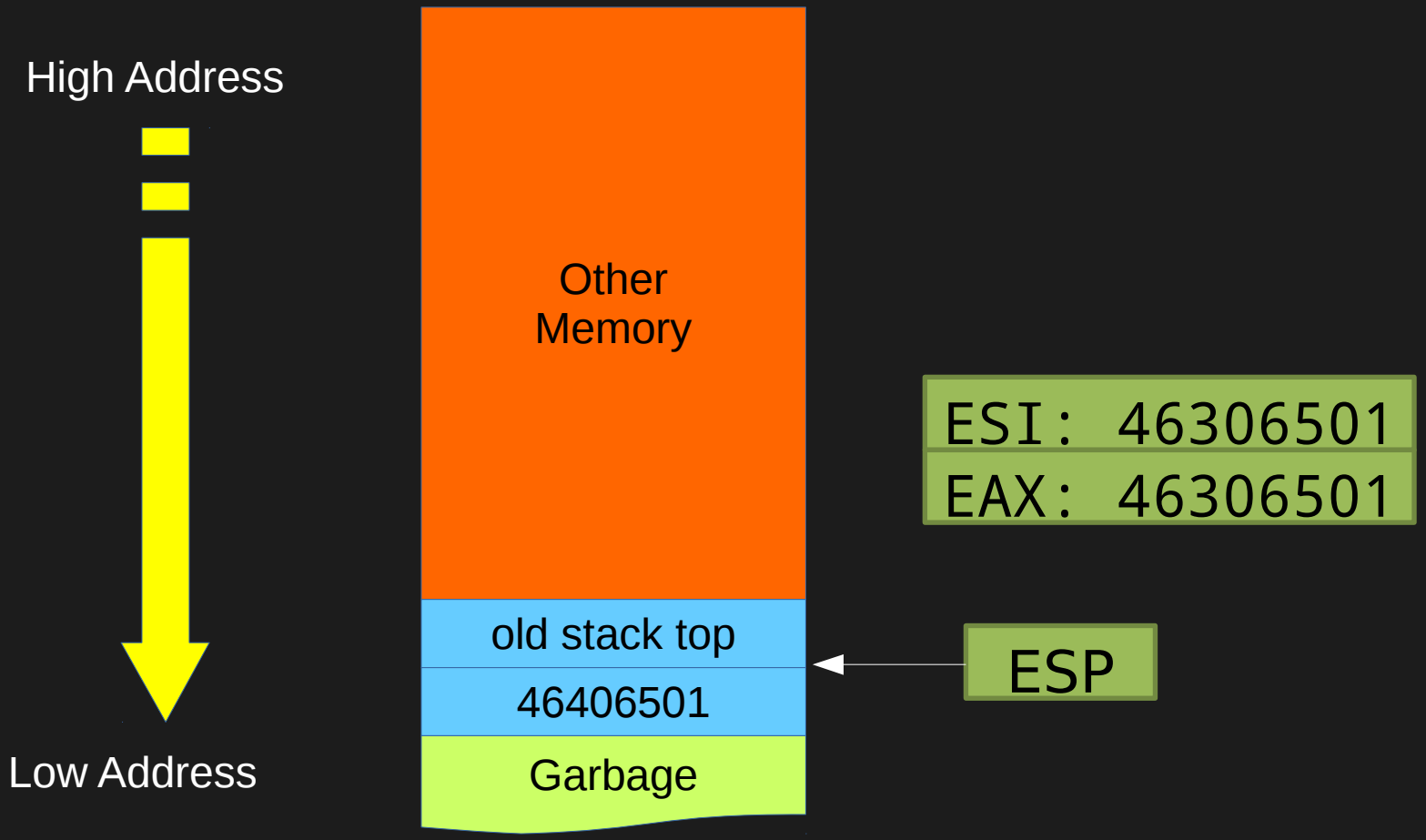
x86 Hardware Stack



x86 Hardware Stack: push %ESI



x86 Hardware Stack: popl %eax



Road Map

- x86 hardware stack
- C calling conventions
  - Stack pointers, return address
  - Function parameters and local variables
  - Return from a function
- Put it all together
- Reference Documents
- Summary

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
  - There exist many calling conventions, we will learn the most common one
- 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:
  - How to preserve old context and create new context?
  - How to return values?
  - How to pass parameters?
  - How to store local variables?

What have to be preserved

- Program counter (PC), i.e., register EIP on x86
- General purpose registers
- Local variables
  - Local variables are saved on stack. Preserving local variables is essentially persevering the stack status
  - A stack frame contains a function calls' data
  - Two registers recording stack status
    - EBP: stack frame pointer, the beginning of the stack frame for the current function
    - ESP: stack pointer, the current top of the stack, end of the stack frame for the current function

Preserving the caller context

- In short, just two things to preserve
  - Register values have to saved
    - General purpose registers, EIP, ESP, EBP
  - Data on stack should not be changed
    - You won't change it as long as you have the correct stack pointer ESP
- How to preserve the context
  - Push register values on the stack
  - E.g. push %eax

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
- PC or EIP is automatically saved by CPU
- Remember: Just a convention, or agreement, among software designers

What Does Caller Do?

- Steps for caller
  1. Push any caller-saved registers that it used
  2. Call the callee (return address automatically pushed, i.e., PC is saved)

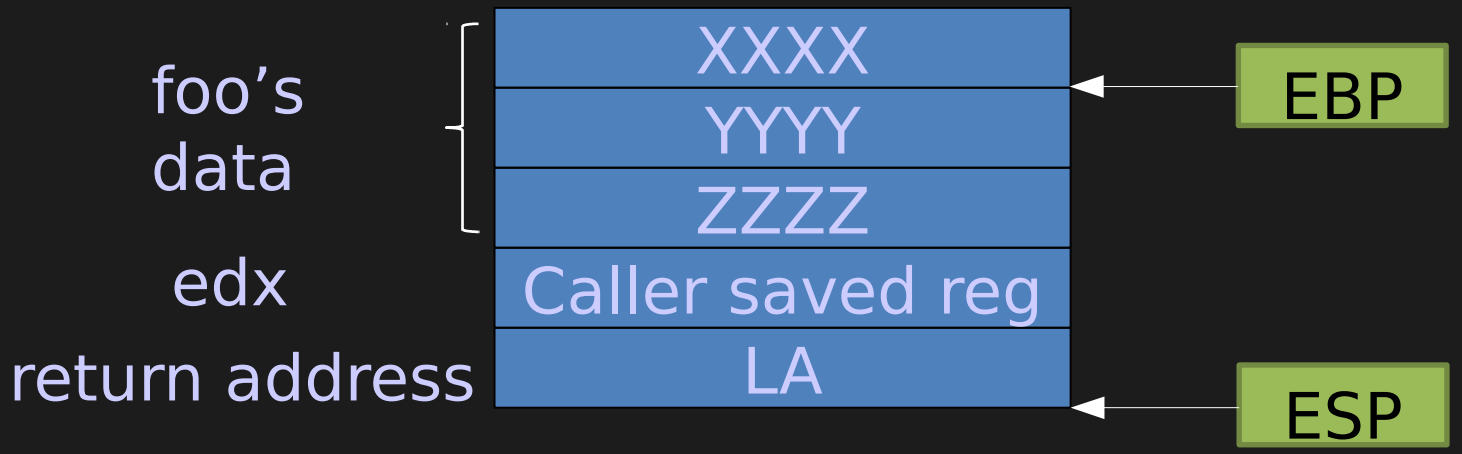
What Does the Caller Do?

- Example: Call a function and pass 3 integer parameters to it; caller is foo

```
    pushl %edx      ; caller-saved register
    call  bar       ; push return address, jump
LA: popl  %edx      ; restore caller-saved edx
                    ; eax holds return value
```

Stack after “Call bar”

- x86 stack immediately after “call bar”



What does Callee do?

- Steps for callee
  1. Push caller's EBP (preserves the address of the beginning of caller's stack frame)
  2. move current ESP to EBP; this is the beginning of callee's stack frame (also preserves the address of the end of caller's stack frame)
  3. Push any callee-saved registers that it will use
  4. Execute callee function

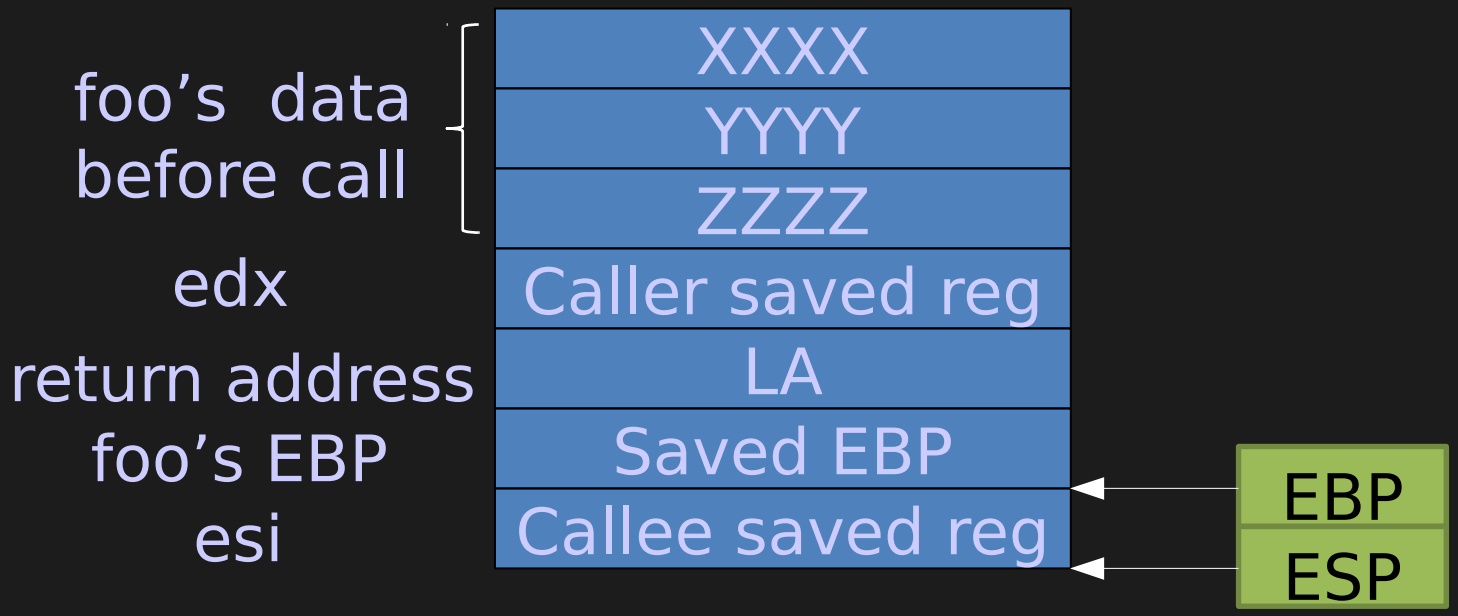
Callee Stack Frame Setup

- To preserve that stack status
- The standard subroutine prologue code sets up the new stack frame:

```
; Prologue code at top of function bar
pushl %ebp          ; save old base pointer
movl  %esp,%ebp     ; Set new base pointer
pushl %esi          ; bar uses ESI, so save
```

Stack After Prologue Code

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



What does Callee do After It Finish Execution? Cleanup Its Stack Frame

- Steps for callee
  1. Push caller's EBP
  2. move current ESP to EBP
  3. Push any callee-saved registers that it will use
  4. Execute callee function
  5. Pop any callee-saved registers that it used
  6. Move current EBP to ESP
  7. Pop caller's EBP
  8. Return to caller

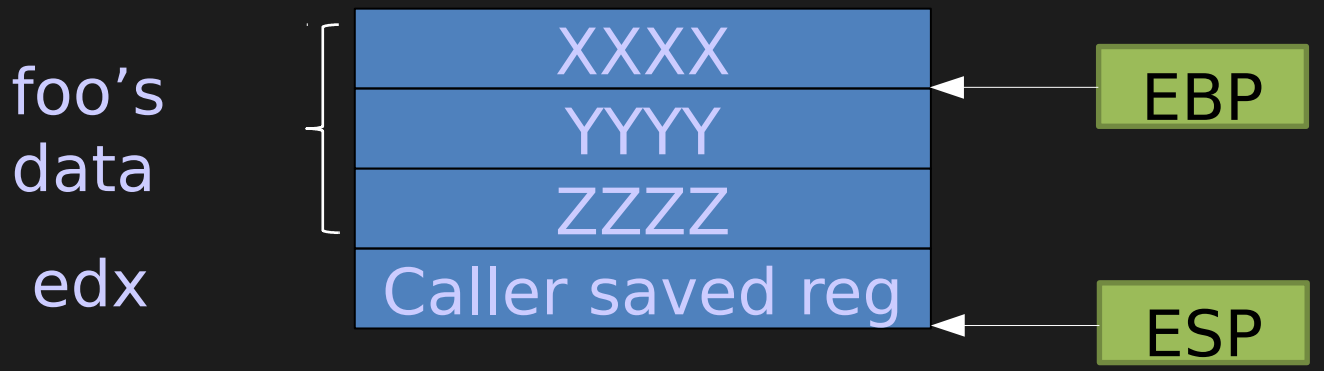
Callee Stack Frame Cleanup

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

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

Stack After Return

- After epilogue code and return:



What Does Caller Do After Callee Returns? Restored its Registers

- Steps for caller
  1. Push any caller-saved registers that it used
  2. Call the callee (return address automatically pushed)
  3. (after callee returns) Pop the general purpose register it saved

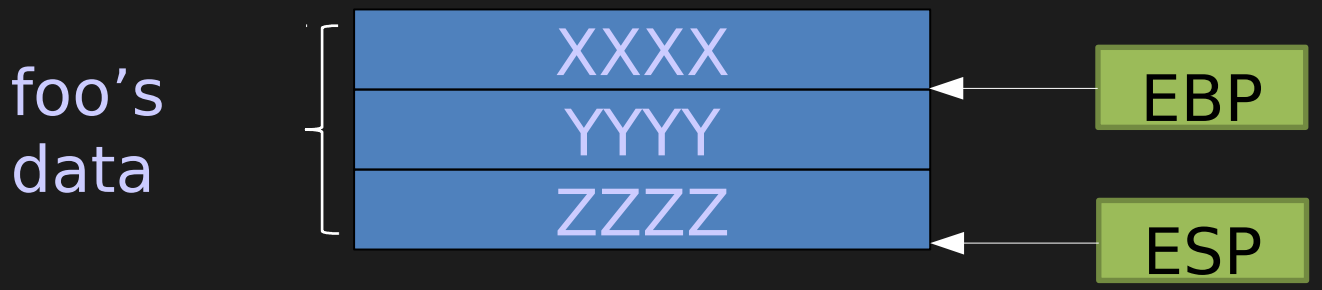
Caller Stack Cleanup

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

```
    pushl %edx      ; caller-saved register
    call  bar       ; push return address, jump
LA: popl  %edx      ; restore caller-saved edx
                    ; eax holds return value
```

Stack After Caller Cleanup

- After epilogue code and return:



Who Saved Which Register

Register	Who Saves	How to save
EAX,ECX,EDX	Caller	Considered volatile, caller must save; Manually push to stack; e.g., push %eax
EBX, EDI, ESI	Callee	Considered non-volatile, callee save if use; Manually push to stack; e.g., push %edi
EBP	Callee	Stack frame pointer; caller stack status; Manually push to stack; push %ebp
ESP	Callee	Stack Pointer; Current top of the stack; Must reset to return address before “ret”
EIP	CPU	Automatically push to stack; “call” pushes EIP, while “ret” popes EIP

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.)

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.)
- Answer: If all registers are callee-saved, then callee may save some registers that caller does not use; if all registers are caller-saved, then caller may save some registers that callee does not use

C Calling Convention cont.

- Questions answered by a calling convention:
  - How to preserve old context and create new context?
  - How to return values?
  - How to pass parameters?
  - How to store local variables?

How are Values Returned?

- For integer return value, 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

What does Callee do to Return Value

- Steps for callee
  1. Push caller's EBP
  2. move current ESP to EBP
  3. Push any callee-saved registers that it will use
  4. Execute callee function
  5. Pop any callee-saved registers that it used
  6. Move return value to EAX
  7. Move current EBP to ESP
  8. Pop caller's EBP
  9. Return to caller

Return value example

- At the epilogue of the callee:

```
; Epilogue code at bottom of function
popl %esi           ; Restore callee-saved ESI
movl $1, %eax       ; return 1, save the value to EAX
movl %ebp,%esp      ; Deallocate stack frame
popl %ebp           ; Restore caller's EBP
ret                 ; return
```

What Does Caller Do to Retrieve Return Value?

- Steps for caller
  1. Push any caller-saved registers that it used
  2. Call the callee (return address automatically pushed)
  3. Examine or save or ignore the return value from EAX depending on the source code
  4. (after callee returns) Pop the general purpose register it saved

Return a Floating Point Value

- A floating point value is return in the ST0 floating point register
- Use flds instruction to load value from memory to ST0

C Calling Convention cont.

- Questions answered by a calling convention:
  - How to preserve old context and create new context?
  - How to return values?
  - How to pass parameters?
  - How to store local variables?

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

What Does Caller Do to Pass Parameters?

- Steps for caller
  1. Push any caller-saved registers that it used
  2. Push the parameters in reverse order
  3. Call the callee
  4. Examine or save or ignore the return value from EAX depending on the source code
  5. (after callee returns) Pop the general purpose register it saved

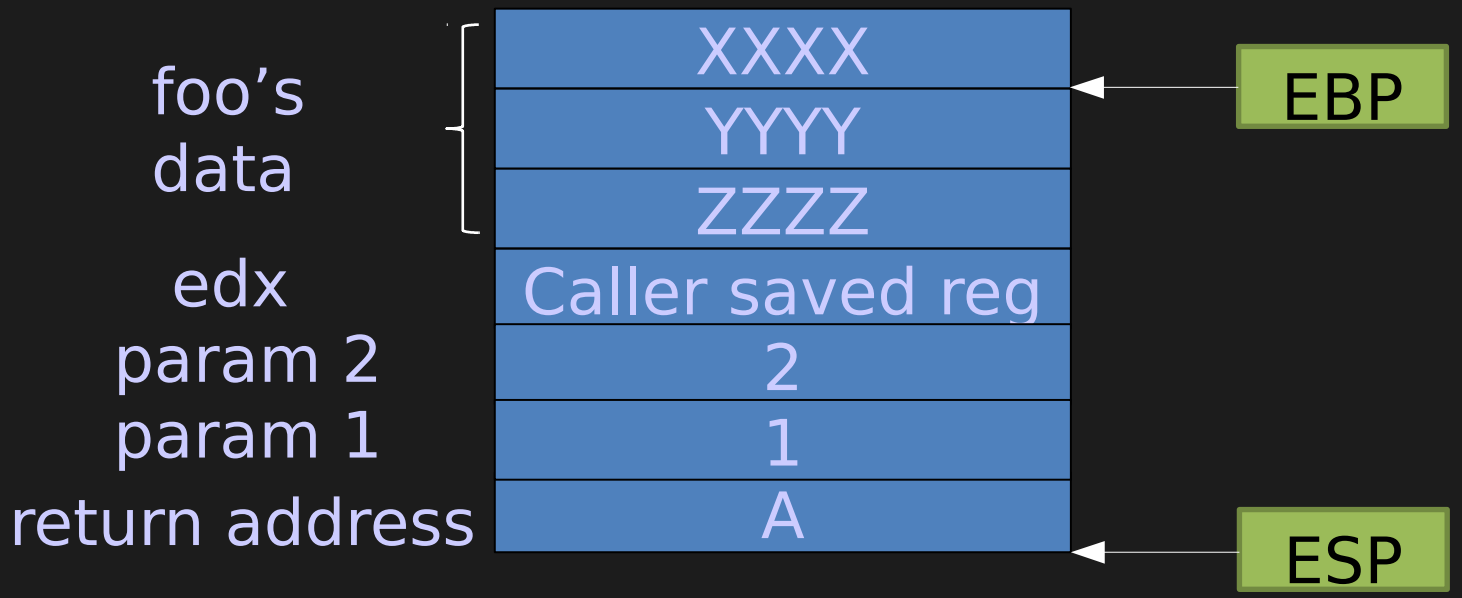
Make the call with parameters

- Example: Call a function and pass 2 integer parameters to it;
- caller is foo, calling bar(1,2)

```
    pushl %edx      ; caller-saved register
    push  $2        ; push param 2
    push  $1        ; push param 1
    call  bar       ; push return address, jump
LA: popl  %edx      ; restore caller-saved edx
                    ; eax holds return value
```

Stack after “Call bar” with parameters

- x86 stack immediately after “call bar”



Callee Stack Frame Setup w/ parameters

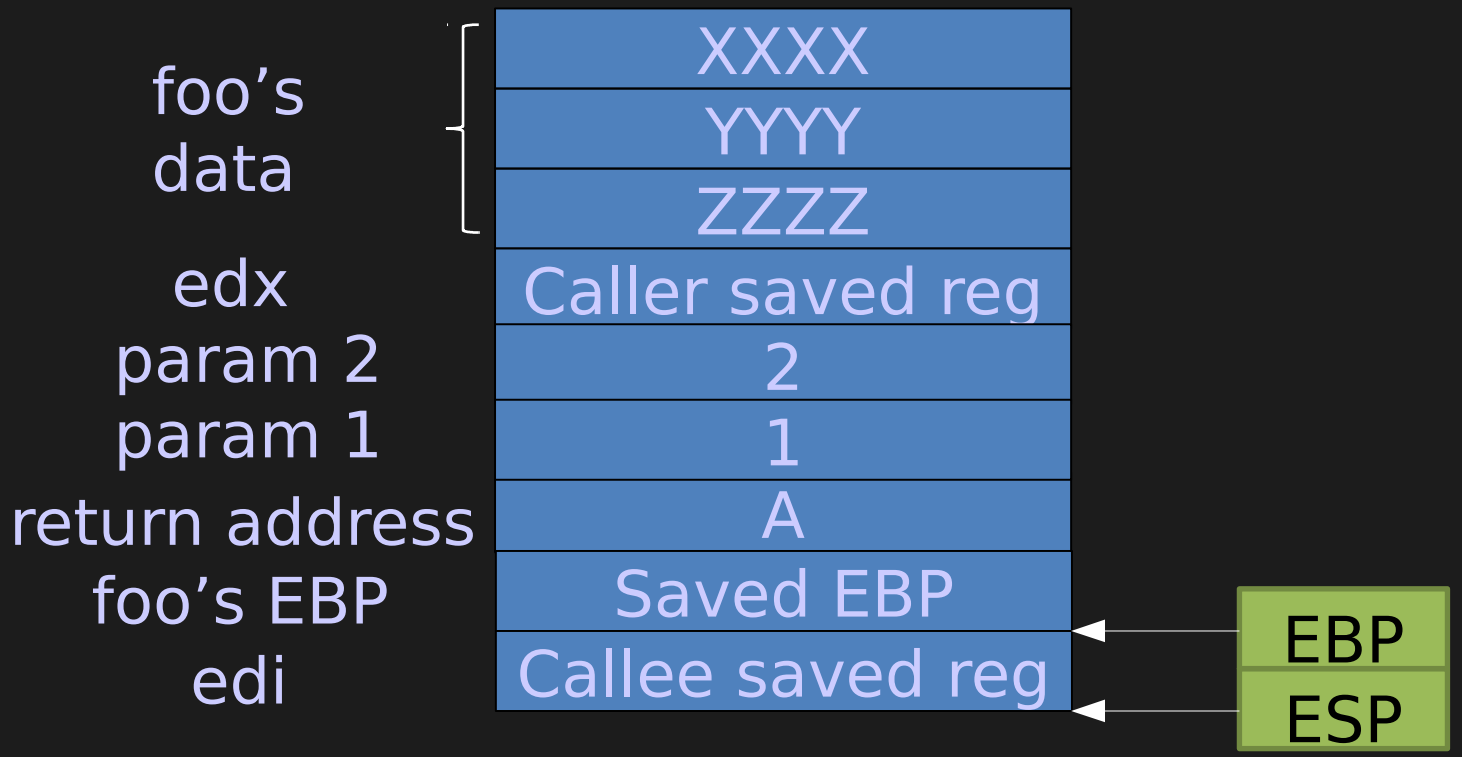
- Nothing different than no-parameter case
- The standard subroutine prologue code sets up the new stack frame:

```
; Prologue code at top of function bar
pushl %ebp          ; save old base pointer
movl  %esp,%ebp     ; Set new base pointer
pushl %esi          ; bar uses ESI, so save
```

- Callee reference first parameter with %ebp+8, second parameter with %ebp+12, and go on…

Stack After Prologue Code

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



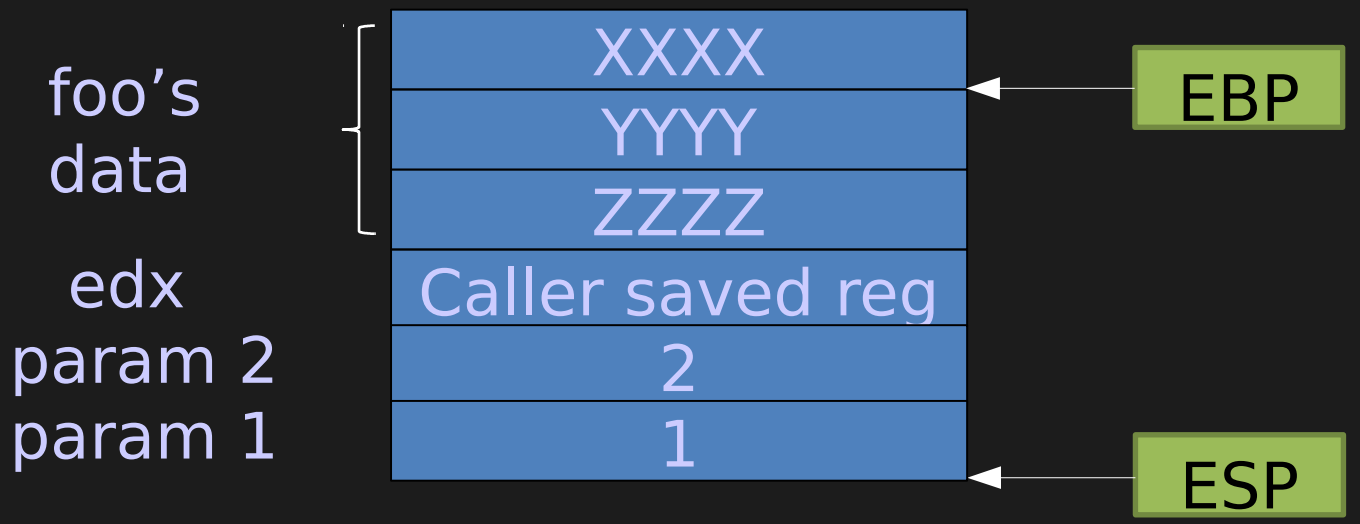
Callee Stack Frame Cleanup w/. params

- no thing different than non-parameter case
- Epilogue code at end cleans up frame (mirror image of prologue):

```
; Epilogue code at bottom of function
popl %esi           ; Restore callee-saved ESI
movl $1, %eax       ; return 1, save the value to EAX
movl %ebp,%esp      ; Deallocate stack frame
popl %ebp           ; Restore caller's EBP
ret                 ; return
```

Stack After Return

- After epilogue code and return:



What Does Caller Do to Pass Parameters?

- Steps for caller
  1. Push any caller-saved registers that it used
  2. Push the parameters in reverse order
  3. Call the callee
  4. Examine or save or ignore the return value from EAX depending on the source code
  5. Release the stack memory used for parameters
  6. (after callee returns) Pop the general purpose register it saved

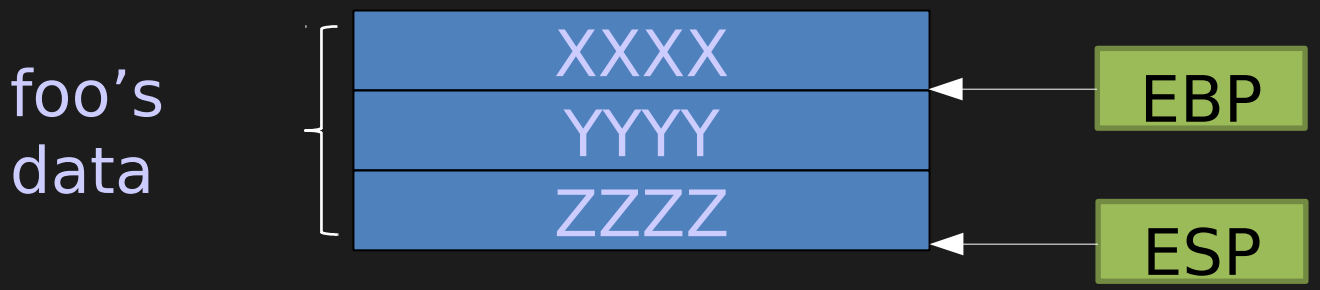
Caller Stack Cleanup with parameters

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

```
    pushl %edx      ; caller-saved register
    push  $2        ; push param 2
    push  $1        ; push param 1
    call  bar       ; push return address, jump
LA: addl  $8, %esp; deallocate parameters
    popl  %edx      ; restore caller-saved edx
                    ; eax holds return value
```

Caller Stack Cleanup with parameters

- After caller stack cleanup:

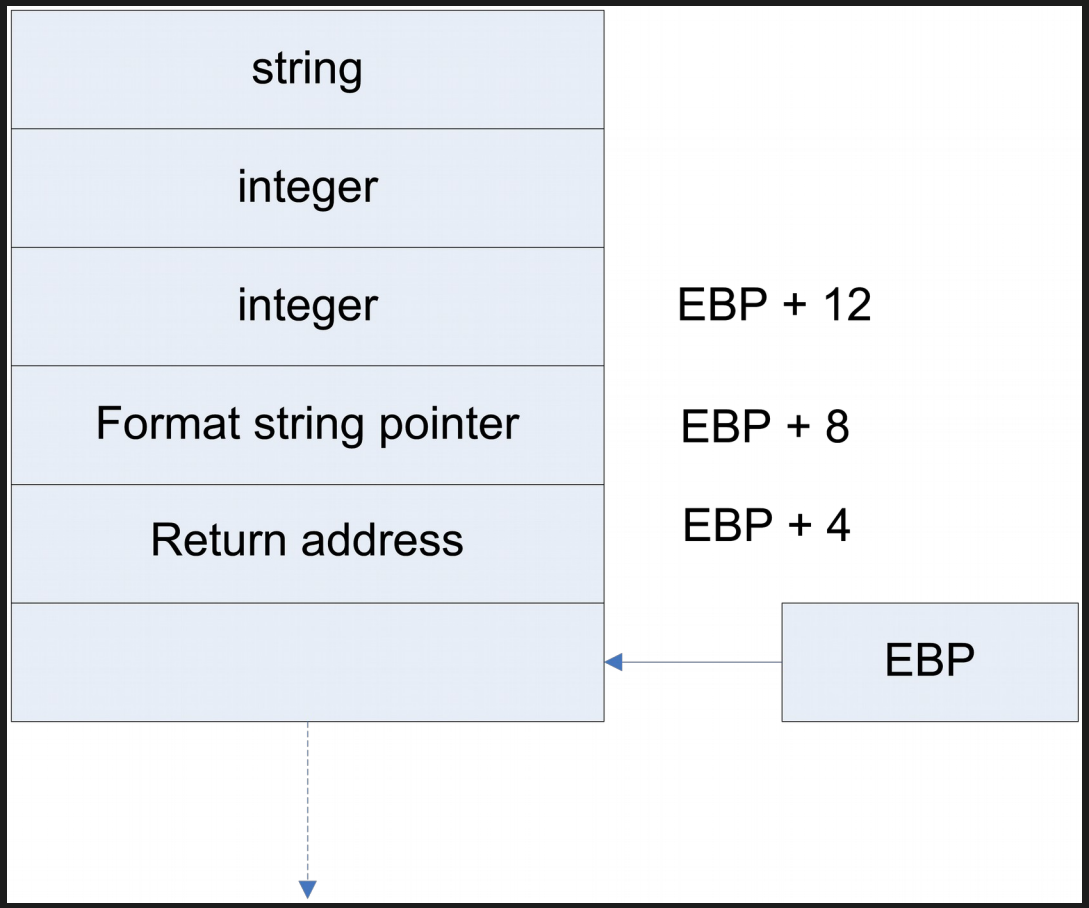


Why Pass Parameters in Reverse Order?

- 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

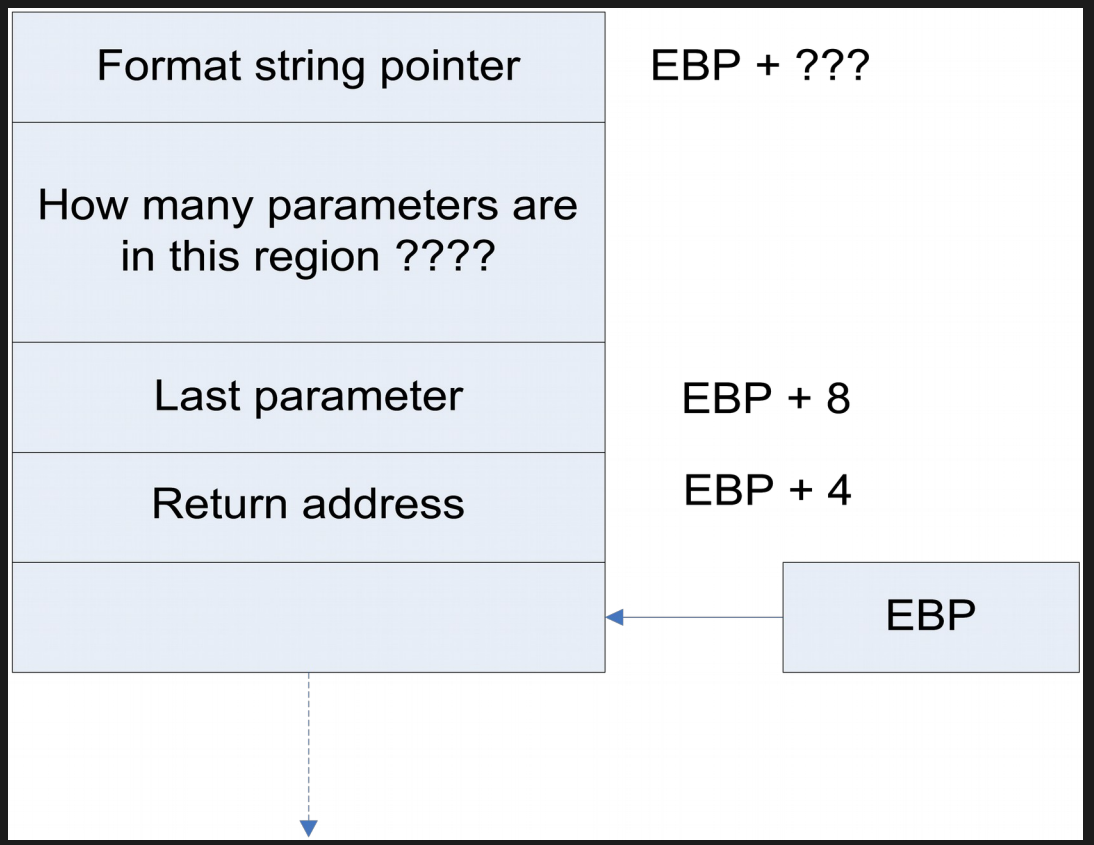
Reverse Order Parameters cont.

- 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 to preserve old context and create new context?
  2. How to return value?
  3. How to pass parameters?
  4. How to store local variables?

What does Callee do to Allocate Local Variables

- Steps for callee
  1. Push caller's EBP
  2. Move current ESP to EBP
  3. Advance stack pointer ESP to make room for locals
  4. Push any callee-saved registers that it will use
  5. Execute callee function
  6. Pop any callee-saved registers that it used
  7. Move current EBP to ESP
  8. Pop caller's EBP
  9. Move return value to EAX
  10. Return to caller

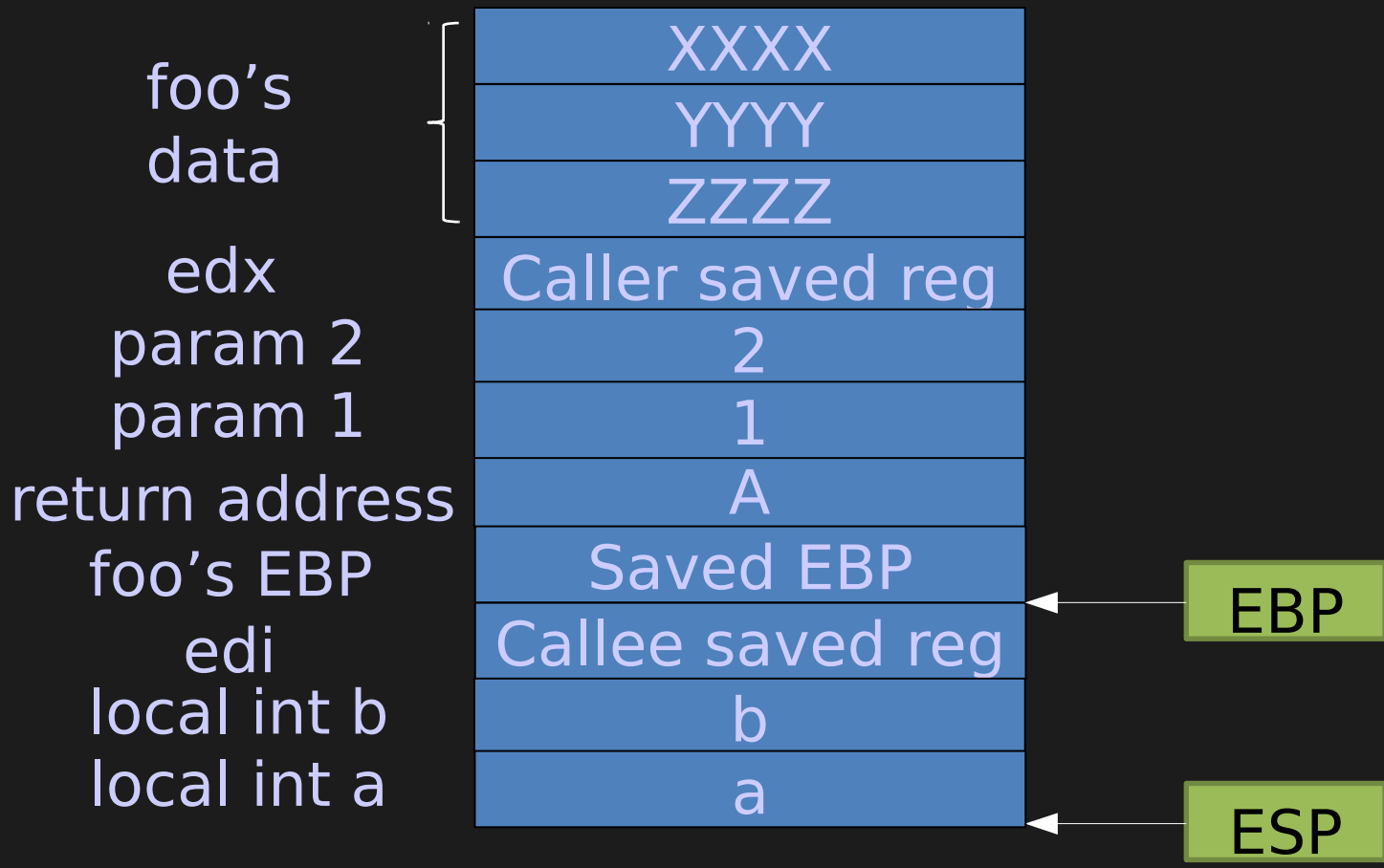
Where are Local Variables Stored?

- Variables are added during prologue. E.g., we have two integer local variables

```
; Prologue code at top of function bar
pushl %ebp          ; save old base pointer
movl  %esp,%ebp     ; Set new base pointer
subl  %8, $esp      ; make room for locals
pushl %esi          ; bar uses ESI, so save
```

Stack After Prologue Code with locals

- After the prologue code sets up the new stack frame (order of the locals is compiler-dependent):



Note that there is no specification on the order (i.e., which local should be on top of stack) of the local variables

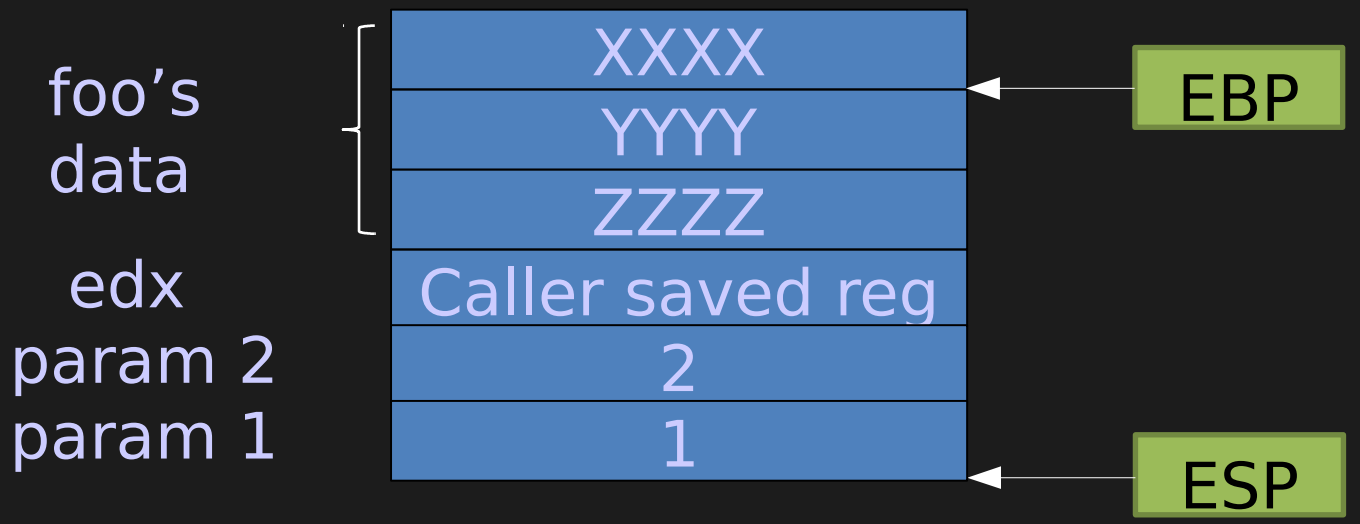
Callee Stack Frame Cleanup w/. locals

- No extra things to do for callee
- Epilogue code at end cleans up frame (mirror image of prologue):

```
; Epilogue code at bottom of function
popl %esi           ; Restore callee-saved ESI
movl $1, %eax       ; return 1, save the value to EAX
movl %ebp,%esp      ; Deallocate stack frame
popl %ebp           ; Restore caller's EBP
ret                 ; return
```

Stack After Return

- After epilogue code and return:



Summary of Caller's Steps

- Steps for caller
  - Push any caller-saved registers that it used
  - Push the parameters in reverse order
  - Call the callee
  - Examine or save or ignore the return value from EAX depending on the source code
  - Release the stack memory used for parameters
  - (after callee returns) Pop the general purpose register it saved

Summary of Callee Steps

- Steps for callee
  1. Push caller's EBP
  2. Move current ESP to EBP
  3. Advance stack pointer ESP to make room for locals
  4. Push any callee-saved registers that it will use
  5. Execute callee function
  6. Pop any callee-saved registers that it used
  7. Move return value to EAX
  8. Pop caller's EBP
  9. Move current EBP to ESP
  10. Return to caller

Road Map

- x86 hardware stack
- C calling conventions
  - Stack pointers, return address
  - Function parameters and local variables
  - Return from a function
- Put it all together
- Reference Documents
- Summary

Put it all together

- Let's see a real example
- GCC may manipulate ESP slightly different then discussed here, but essentially doing the same thing
- "leave" instruction is a high level procedure exit operation
  - leave = mov %ebp, %esp; pop %ebp;
  - For faster functions call

Call Stack: Virus Implications

- The return address is the 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
- Read “Tricky Jump” document on syllabus for another virus technique

Reference Documents

- A Tiny Guide to Programming in 32-bit x86 Assembly Language
- Intel Software Developer's manual
  - Vol 1 Chapter 6

Summary

- x86 hardware stack
  - Grow downward
  - ESP
- Calling conventions
  - It is a software agreement
  - Preserve registers (caller and callee)
  - Preserve stack frame pointer: EBP (callee)
  - Return value: eax (callee)
  - Passing parameters (caller)
  - Allocate local variables (callee)
  - Steps for caller and callee