Stacks and Buflab

15-213: Introduction to Computer Systems
Recitation 4: Monday, Sept. 23, 2013

Marjorie Carlson
Section A

Agenda

- What's New & Exam Prep
- Bomb Lab (Sorta): Jump Tables vs. Sparse Switches
- Stacks
  - x86-32 Stack Discipline
  - Function Call Walkthrough
  - Back to x86-64
- Buflab
  - Quick Start
  - Miscellany
- Exam Question

What's New (or Not)

- Bomb Lab is due Tuesday (tomorrow), 11:59 PM
  - Your late days are wasted here
  - Student: "But if you wait until the last minute, then it only takes a minute!"
    - Nope.
- Buflab is out Tuesday (tomorrow), 11:59 PM
  - Hacking the stack
- Stacks will be on the exams
  - They're tough at first, but I believe in you ☺

Speaking of the Exam…

- **Midterm: Wed-Sat, 16-19 Oct.** (3.5 weeks from now)

- Covers everything up to, and including, caches:
  - Chapters 1-3 and 6 of textbook.
  - Labs up to and including Cache Lab.
  - Lectures up to and including Caches (1 Oct 2013).

- You will get to bring in one piece of paper (front and back) but it cannot contain preworked problems.

- The recitation three weeks from today will be an exam review… but you should already be reviewing!

Speaking of the Exam…

- How to do well on the exam, according to your professors:
  - Prof. O'Halleron: Read each chapter **3** times and work the practice problems.
  - Prof. Kesden: do **many, many** practice exams. Specifically:
    - Print at least 7 previous exams. Set aside the most recent 2.
    - Divide the other 5 by topic. As we cover each topic, do the associated exam questions until you get good at them.
    - A week or two before the exam, take one of the exams you set aside.
    - Review the areas in which you made mistakes.
    - Take the last exam not long before the real midterm, to double-check your readiness (and timing).

Agenda

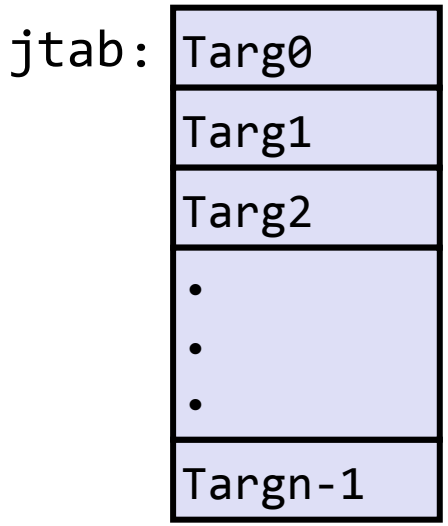
- What's New
- **Bomb Lab (Sorta): Jump Tables vs. Sparse Switches**
- Stacks
  - x86-32 Stack Discipline
  - Function Call Walkthrough
  - Back to x86-64
- Buflab
  - Quick Start
  - Miscellany
- Exam Question

Jump Table Structure

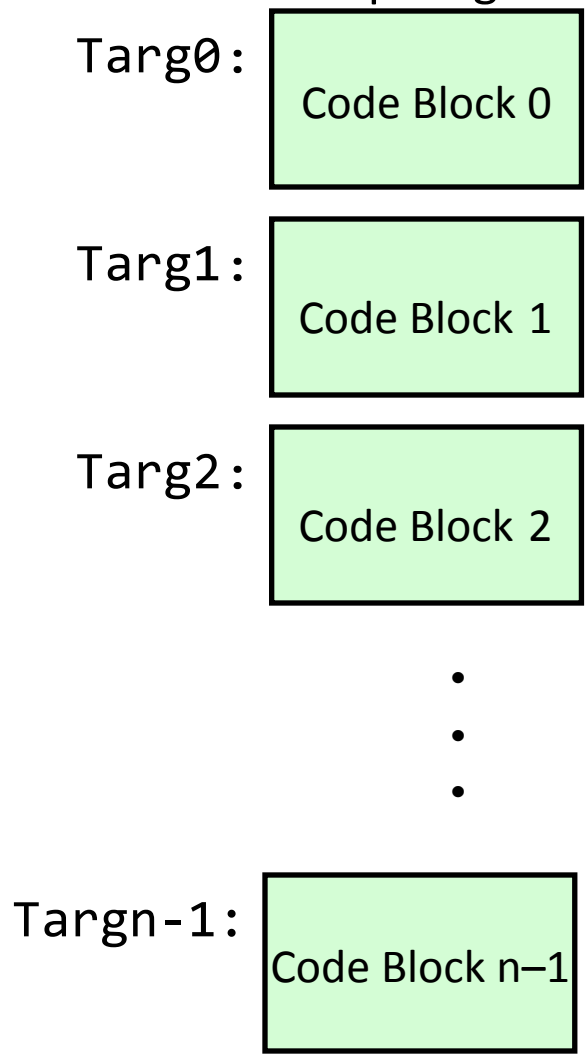
Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table



Jump Targets



Approximate Translation

```
target = JTab[x];
goto *target;
```

Jump Table Example

- The tip-off is something like this:
  - `jmpq *0x400600(,%rax,8)`
    - Empty base means implied 0
    - `%rax` is the "index"
    - 8 is the "scale" (64-bit machine addresses are 8 bytes)
    - * indicates a dereference (like in C notation)
      - Like `leal`: does not do a dereference with parenthesis
  - Put it all together: jump to the address stored in the address 0x400600 + %rax*8
- Using GDB (example output): `x/8g 0x400600`

```
0x400600: 0x00000000004004d1 0x00000000004004c8
0x400610: 0x00000000004004c8 0x00000000004004be
0x400620: 0x00000000004004c1 0x00000000004004d7
0x400630: 0x00000000004004c8 0x00000000004004be
```

Sparse Switches

- Not every switch statement in C compiles to a jump table!
  - Jump tables work if every entry has a jump location.
  - But what if your only cases are x = 1 or x = 100?
  - If the cases are less densely packed (sparse), it's more efficient to implement the switch with a series of comparisons.

Sparse Switch Example

```
int div111(int x) {
  switch(x) {
    case   0: return 0;
    case 111: return 1;
    case 222: return 2;
    case 333: return 3;
    case 444: return 4;
    case 555: return 5;
    case 666: return 6;
    case 777: return 7;
    case 888: return 8;
    case 999: return 9;
    default: return -1;
  }
}
```

```
movl 8(%ebp),%eax # get x
cmpl $444,%eax    # x:444
je L8
jg L16
cmpl $111,%eax    # x:111
je L5
jg L17
testl %eax,%eax   # x:0
je L4
jmp L14
     . . .
```

```
     . . .
L5:
     movl $1,%eax
     jmp L19
L6:
     movl $2,%eax
     jmp L19
L7:
     movl $3,%eax
     jmp L19
L8:
     movl $4,%eax
     jmp L19
     . . .
```

Agenda

- What's New
- Bomb Lab (Sorta): Jump Tables vs. Sparse Switches
- **Stacks**
  - **x86-32 Stack Discipline**
  - Function Call Walkthrough
  - Back to x86-64
- Buflab
  - Quick Start
  - Miscellany
- Exam Question

x86-32 Registers

- x86-64 (used in Bomb Lab) gives you 16 registers:

  `%rax, %rbx, %rcx, %rdx, %rsi, %rdi, %rsp, %rbp,`
  `%r8, %r9, %r10, %r11, %r12, %r13, %r14, %r15`

  - **15** are general-purpose.
  - `%rsp` points at the stack.

- x86-32 (used in Buflab) gives you only 8 registers:

  `%eax, %ebx, %ecx, %edx, %esi, %edi, %esp, %ebp`

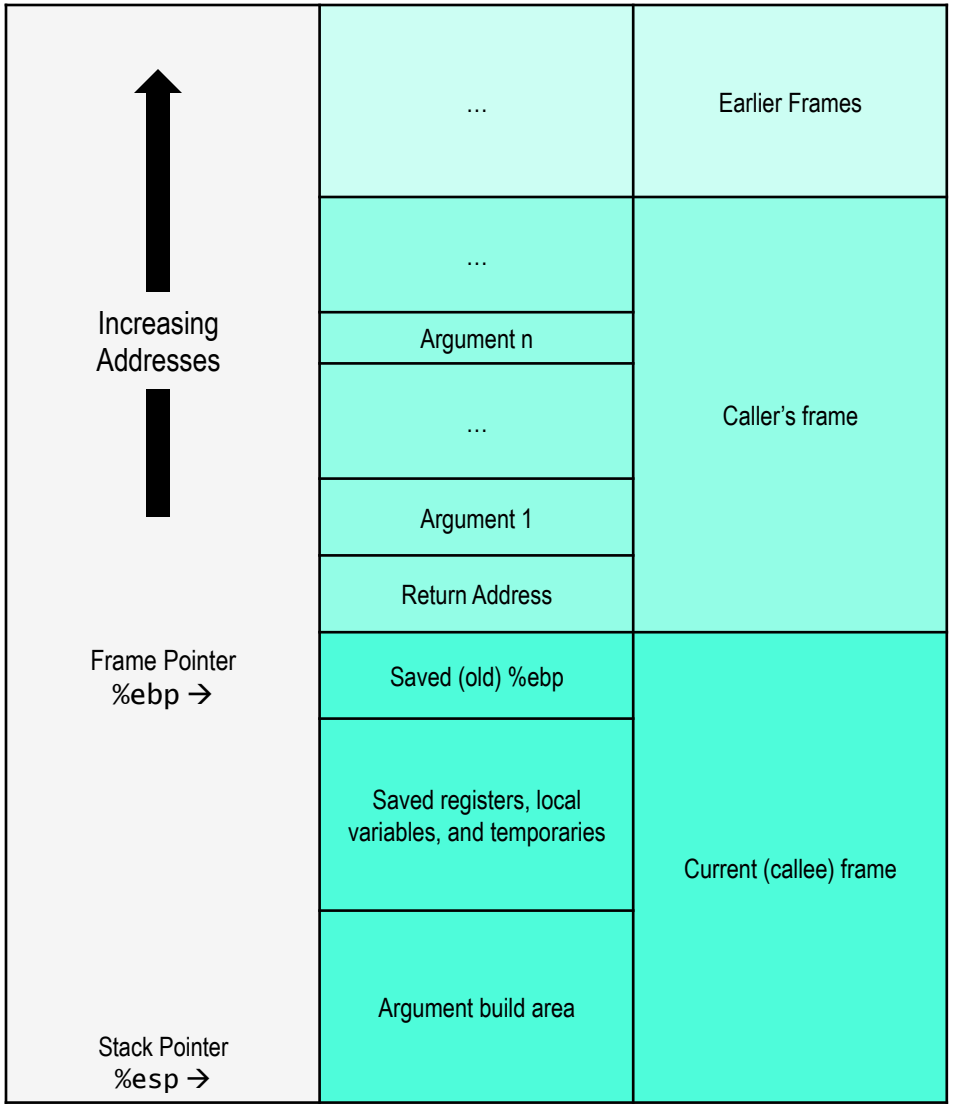
  - **6** are general-purpose.
  - `%ebp` points to the **bottom** of the stack frame.
  - `%esp` points to the **top** of the stack.

Stack Frames

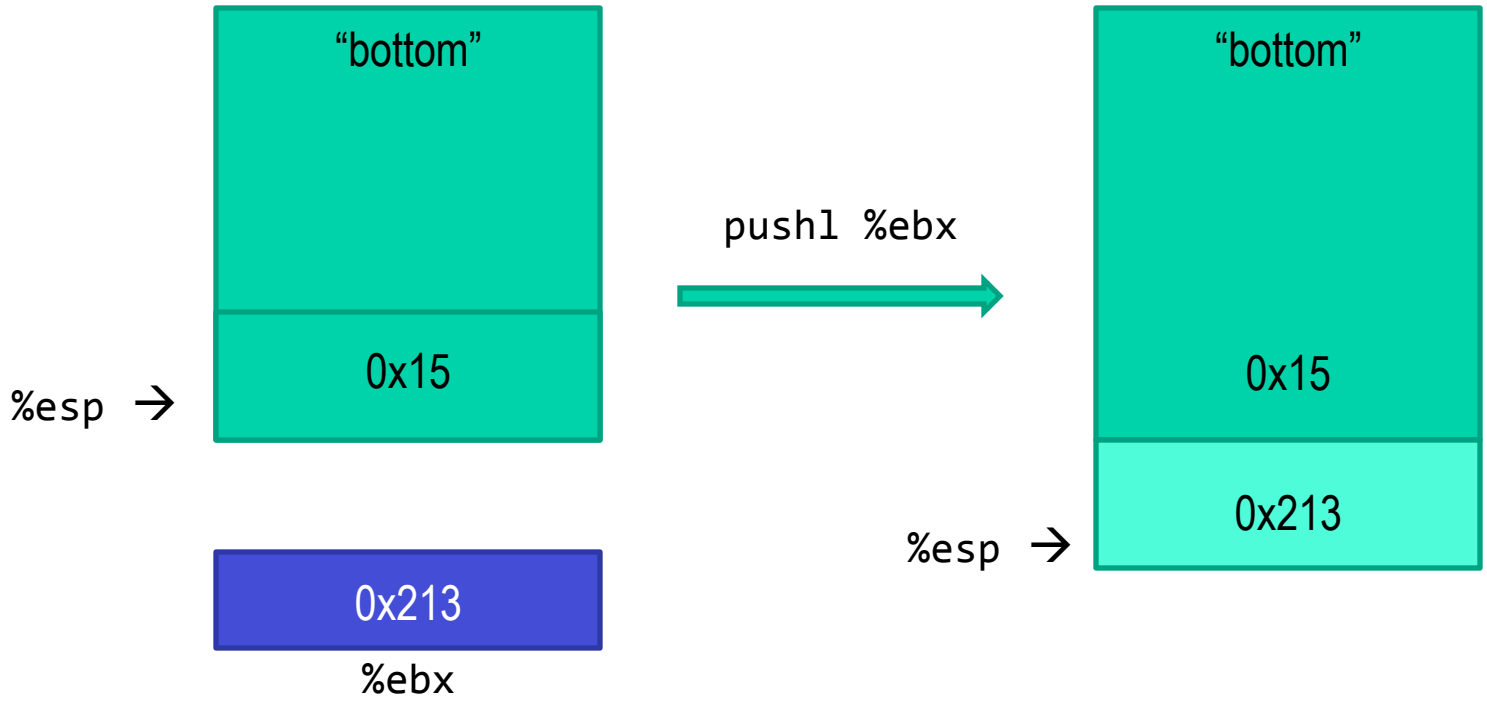
- Every **instance*** of a function gets its own "stack frame" when it's called.
- All the useful stuff can go on the stack!
  - Local variables.
  - Data types that don't fit into registers (structs, arrays …).
  - Things the compiler couldn't fit into registers right now.
  - Arguments for the next function.
  - Callee/caller save registers (more on this in a moment!).

* This makes recursion work!

x86-32 Stack Discipline



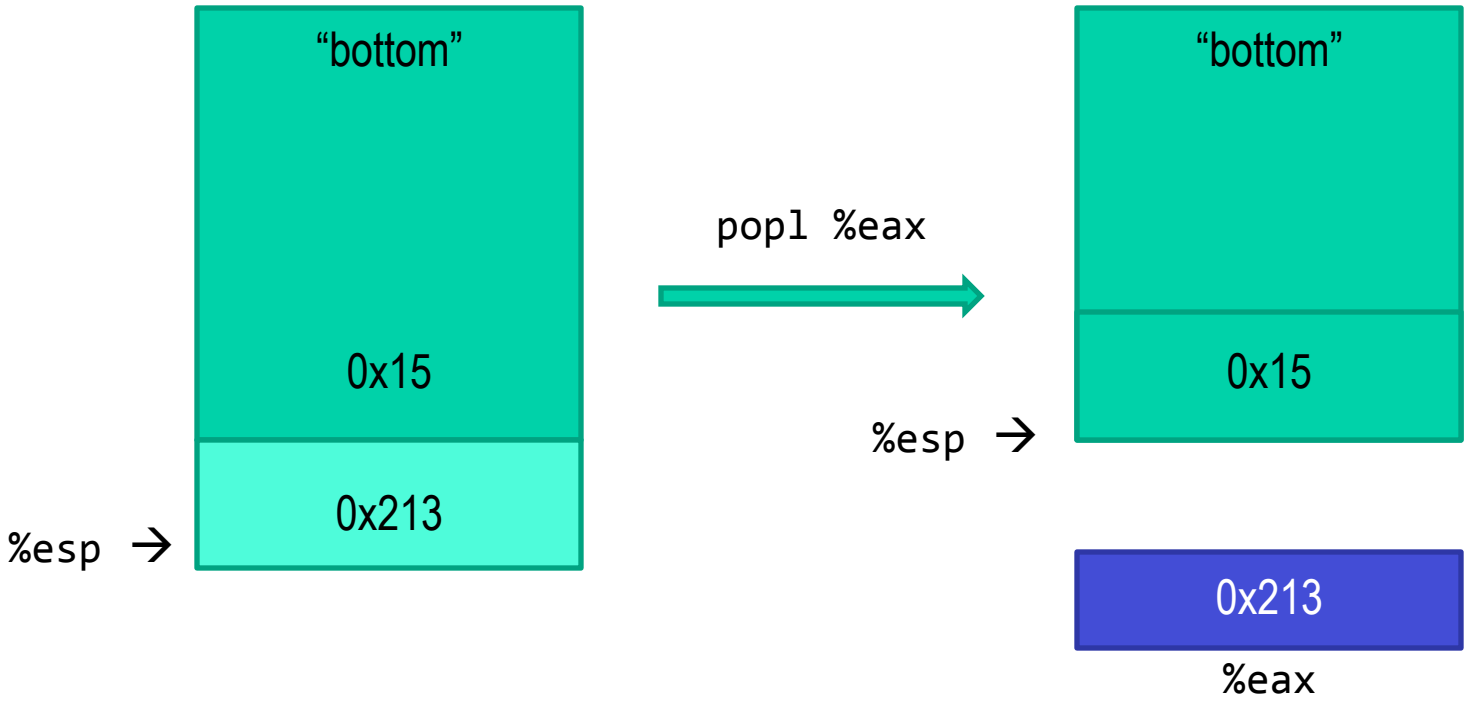
Pushing onto the Stack (i.e., saving)



- In general, `pushl %ebx` translates to:

```
subl $0x4, %esp
movl %ebx, (%esp)
```

Popping off the Stack (i.e., restoring)



- In general, `popl %eax` translates to:

```
movl (%esp), %eax
addl $0x4, %esp
```

Agenda

- What's New
- Bomb Lab (Sorta): Jump Tables vs. Sparse Switches
- Stacks
  - x86-32 Stack Discipline
  - **Function Call Walkthrough**
  - Back to x86-64
- Buflab
  - Quick Start
  - Miscellany
- Exam Question

A Reminder about Registers

- In x86-32, we have to reuse registers more often.
- To reuse a register that's in use, we first have to **save** (push) the old value in the register, then use the register, then **restore** (pop) the old value.
- The question of **who** has to do this is answered by the caller- and callee-save conventions:

Caller-save	`%eax %ecx %edx`
Callee-save	`%ebx %edi %esi`

- What would happen if all registers were callee-save?
- What would happen if all registers were caller-save?

Before, During & After a Function Call

- BEFORE: the **caller…**
  - pushes caller-save registers
  - pushes arguments to the next function
  - `calls` the function; this implicitly pushes the return address
- DURING: the **callee…**
  - pushes %ebp (saves the previous stack frame)
  - copies the value of `%esp` into %ebp
  - pushes any callee-save registers it wants to use
- BEFORE RETURNING: **the callee…**
  - pops relevant callee-save registers
  - copies %ebp into `%esp`
  - pops %ebp
  - `rets`; this pops the return address into `%eip`.
- AFTER: **the caller…**
  - removes arguments (by adding to `%esp` or popping arguments)
  - pops caller-save registers

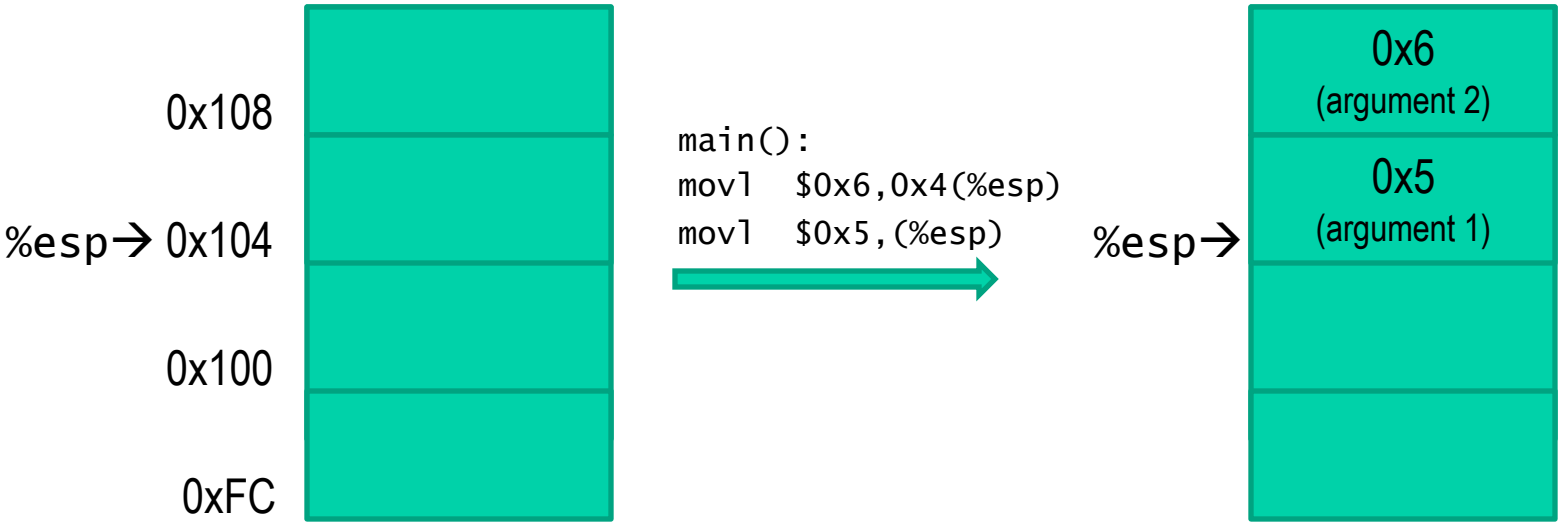
Stack Frames in Action

C Code	Disassembly

`int main() {`
`  return addition(5, 6);`
`}`
 	
`int addition(int x, int y) {`
`  return x+y;`
`}`
	`08048394 <main>:`
	`8048394: 55 push %ebp`
	`8048395: 89 e5 mov %esp,%ebp`
	`8048397: 83 e4 f0 and $0xfffffff0,%esp`
	`804839a: 83 ec 10 sub $0x10,%esp`
	`804839d: c7 44 24 04 06 00 00 movl $0x6,0x4(%esp)`
	`80483a4: 00`
	`80483a5: c7 04 24 05 00 00 00 movl $0x5,(%esp)`
	`80483ac: e8 02 00 00 00 call 80483b3 <addition>`
	`80483b1: c9 leave`
	`80483b2: c3 ret`
	
	`080483b3 <addition>:`
	`80483b3: 55 push %ebp`
	`80483b4: 89 e5 mov %esp,%ebp`
	`80483b6: 8b 45 0c mov 0xc(%ebp),%eax`
	`80483b9: 8b 55 08 mov 0x8(%ebp),%edx`
	`80483bc: 8d 04 02 lea (%edx,%eax,1),%eax`
	`80483bf: c9 leave`
	`80483c0: c3 ret`

Breakdown: Argument Build

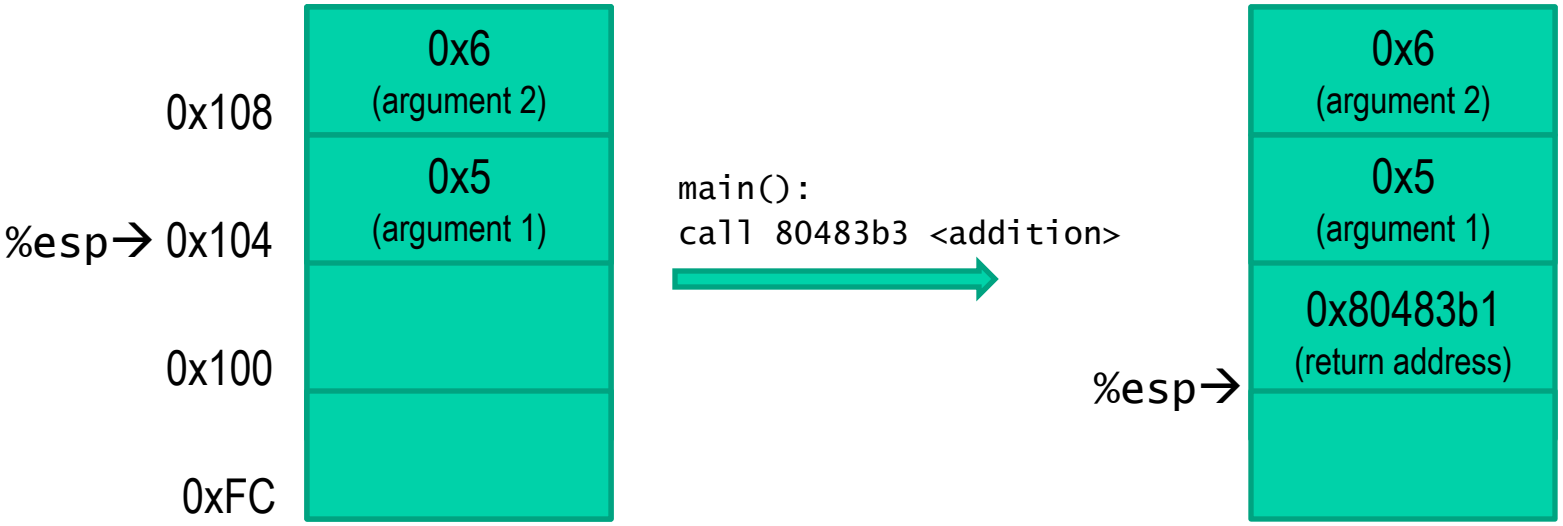
- Build the arguments (note: 2 instructions are executed in this example)



Before	After
%esp = 0x104 %ebp = 0x200 %eip = 0x804839d	%esp = 0x104 %ebp = 0x200 %eip = 0x80483ac

Breakdown: Function Call

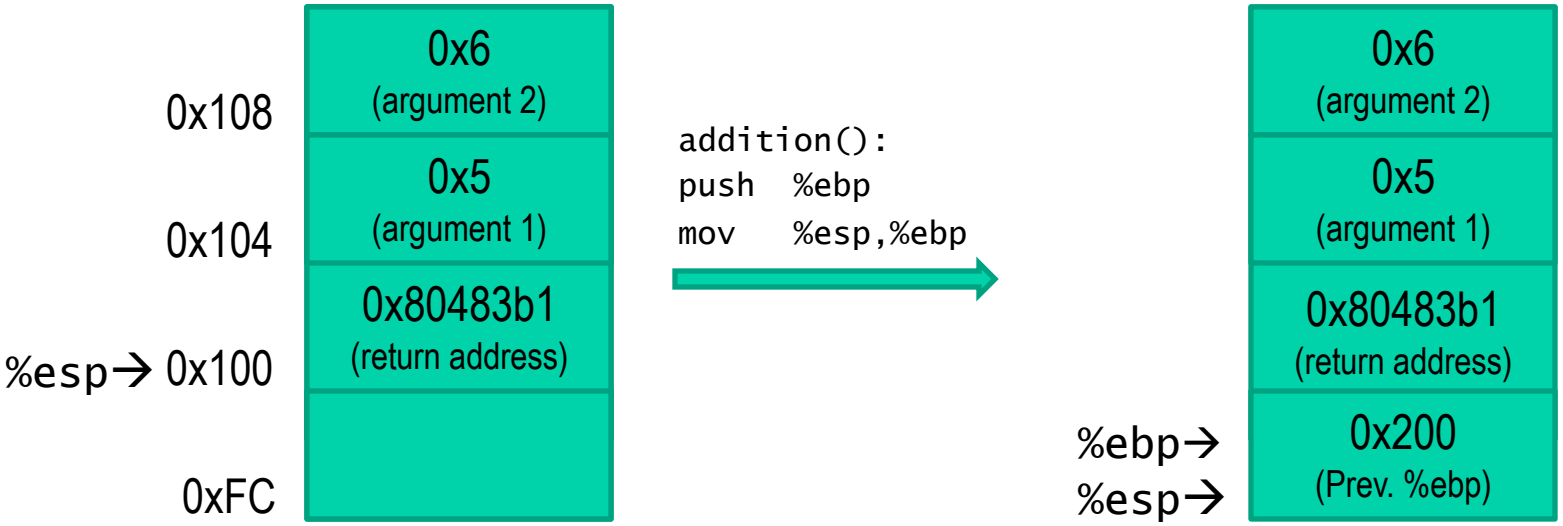
- Call the function



Before	After
%esp = 0x104 %ebp = 0x200 %eip = 0x80483ac	%esp = 0x100 %ebp = 0x200 %eip = 0x80483b3

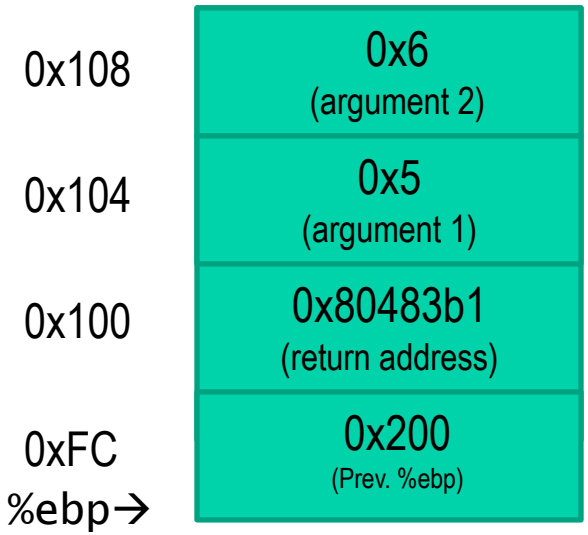
Breakdown: Callee Setup

- Stack frame set up (note: 2 instructions are executed in this example)



Before	After
%esp = 0x100 %ebp = 0x200 %eip = 0x80483b3	%esp = 0xFC %ebp = 0xFC %eip = 0x80483b6

Breakdown: Accessing Arguments



Argument	Location
Argument 2	`0xC(%ebp)`
Argument 1	`0x8(%ebp)`

- In the current frame, arguments are accessed via references to `%ebp`
  - Notice how argument 1 is at `0x8(%ebp)`, not `0x4(%ebp)`
- How does this compare to x86-64?

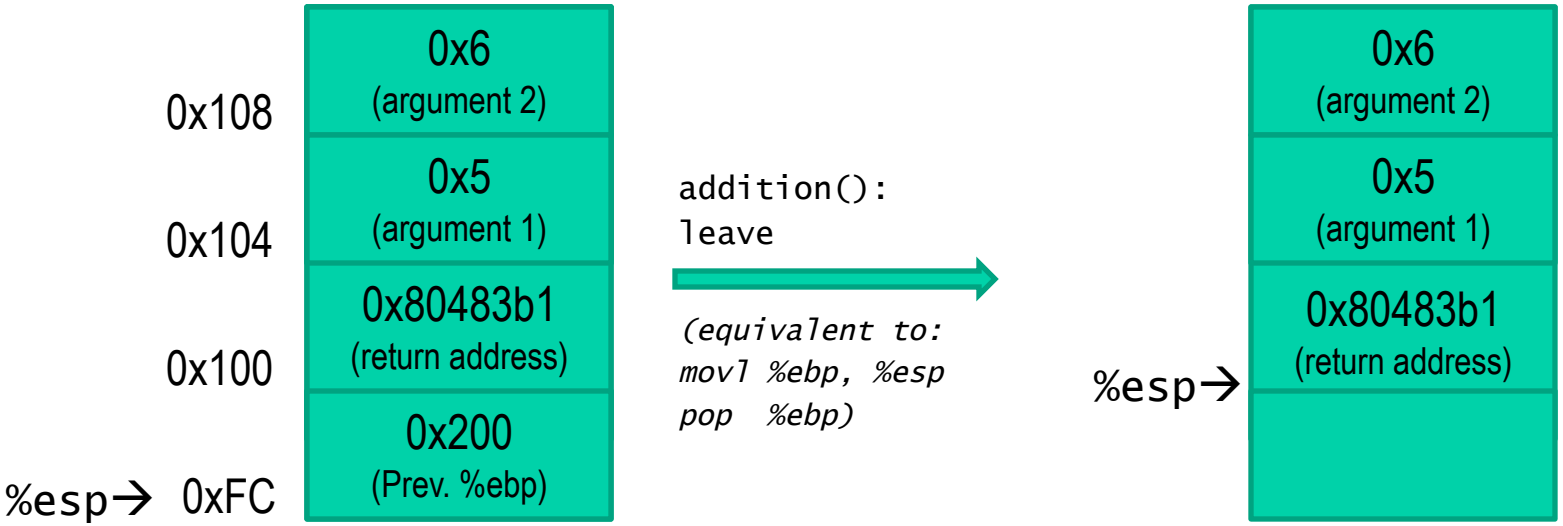
Let's Review the Code Again

C Code	Disassembly

`int main() {`
`  return addition(5, 6);`
`}`
 	
`int addition(int x, int y)`
`{`
`  return x+y;`
`}`
	`08048394 <main>:`
	`8048394: 55 push %ebp`
	`8048395: 89 e5 mov %esp,%ebp`
	`8048397: 83 e4 f0 and $0xfffffff0,%esp`
	`804839a: 83 ec 10 sub $0x10,%esp`
	`804839d: c7 44 24 04 06 00 00 movl $0x6,0x4(%esp)`
	`80483a4: 00`
	`80483a5: c7 04 24 05 00 00 00 movl $0x5,(%esp)`
	`80483ac: e8 02 00 00 00 call 80483b3 <addition>`
	`80483b1: c9 leave`
	`80483b2: c3 ret`
	
	`080483b3 <addition>:`
	`80483b3: 55 push %ebp`
	`80483b4: 89 e5 mov %esp,%ebp`
	`80483b6: 8b 45 0c mov 0xc(%ebp),%eax`
	`80483b9: 8b 55 08 mov 0x8(%ebp),%edx`
	`80483bc: 8d 04 02 lea (%edx,%eax,1),%eax`
	`80483bf: c9 leave`
	`80483c0: c3 ret`

Breakdown: Preparing to Return

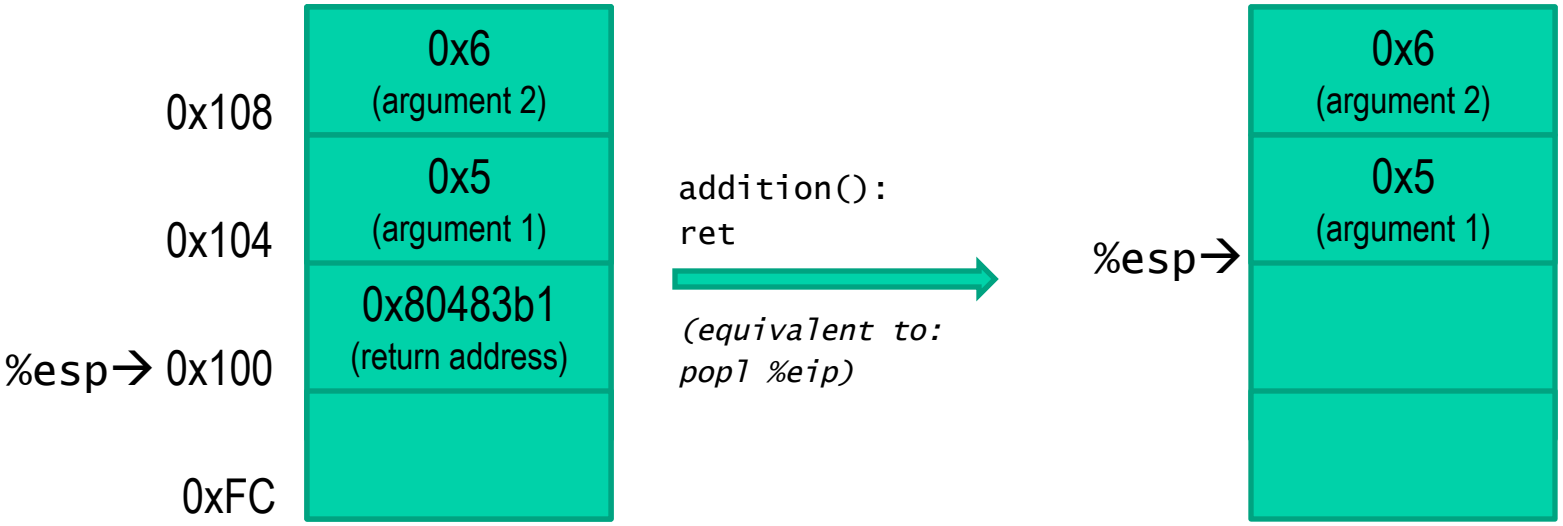
- Preparing to return from a function



Before	After
%esp = 0xFC %ebp = 0xFC %eip = 0x80483bf	%esp = 0x100 %ebp = 0x200 %eip = 0x80483c0

Breakdown: Return

- Return from a function



Before	After
%esp = 0xFC %ebp = 0x200 %eip = 0x80483c0	%esp = 0x104 %ebp = 0x200 %eip = 0x80483b1

Agenda

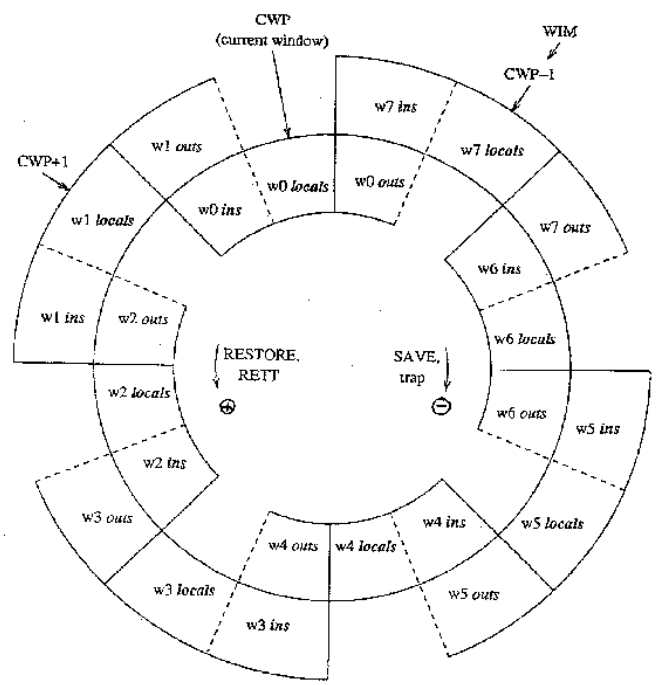
- What's New & Exam Prep
- Assembly Review: Jump Tables vs. Sparse Switches
- Stacks
  - x86-32 Stack Discipline
  - Function Call Walkthrough
  - **Extras on x86(-64) stacks**
- Buflab Quick Start
  - Essential Items of Business
  - Miscellany
- Exam Question

Stacks on x86-64

- Arguments (≤ 6) are passed via registers
  - `%rdi, %rsi, %rdx, %rcx, %r8, %r9`
  - Extra arguments passed via stack!
    - X86-32 stack knowledge still matters!
- Don't need `%ebp` as the base pointer
  - Compilers are smarter now
- Overall less stack use
  - Potentially better performance

Aside: Technology Note

- This class is strictly x86(-64).
- Other architectures may have different conventions
  - x86 stacks always grow down.
  - ARM is configurable: stacks can grow up **or** down.
  - SPARC stacks…



Agenda

- What's New
- Bomb Lab (Sorta): Jump Tables vs. Sparse Switches
- Stacks
  - x86-32 Stack Discipline
  - Function Call Walkthrough
  - Back to x86-64
- **Buflab**
  - Quick Start
  - Miscellany
- Exam Question

Buflab

- A series of exercises asking you to overflow the stack and change execution
  - You do this with inputs that are super long and write over key stack values
  - **Incorrect inputs will not hurt your score**

- Seminal paper on stack corruption
  - *Smashing the Stack for Fun and Profit* (1996)

Basic Approach

1) Examine the provided C code/disassembly
   - Disassembling

     ```
     > objdump -d bufbomb > outfile
     ```
   - Don't forget that GDB is still used for this lab!

2) Find out how long to make your inputs

3) Write exploits to divert program execution.

4) Profit!

Buflab Tools

- `./makecookie` *`andrewID`*
  - Makes a unique "cookie" based on your Andrew ID
- `./hex2raw`
  - Use the hex generated from assembly to pass raw strings into bufbomb
  - Use with –n in the last stage
- `./bufbomb -t` *`andrewID`*
  - The actual program to attack
  - Always pass in with your Andrew ID so your score is logged
  - Use with –n in the last stage

How to Input Answers

- Earlier stages:
  - Put your byte code exploit into a text file.
  - Feed it through `hex2raw`.

- Later stages:
  - Write (corruption) assembly
  - Compile it.

    > `gcc -m32 -c` *`example.S`*
  - Get the byte codes.

    > `objdump -d` *`example.o`* `>` *`outfile`*
  - Then feed it through `hex2raw`.

Ways to Feed Byte Codes

- Option 1: Pipes

```
> cat exploitfile | ./hex2raw | ./bufbomb -t andrewID
```

- Option 2: Redirects

```
> ./hex2raw < exploitfile > exploit-rawfile
> ./bufbomb -t andrewID < exploit-rawfile
```

- Option 3: Redirects in GDB

```
> gdb bufbomb
(gdb) run -t andrewID < exploit-rawfile
```

Potential Points of Failure

- Don't use byte value `0A` in your exploit
  - ASCII for newline
  - `Gets()` will terminate early if it sees this
- When multiple exploits submitted for the same level, it always takes the latest submission. So, if you accidentally pass the wrong exploit later, remember to pass the correct one again. (Yay version control!)
- If you manage to execute your exploit….
  - GDB will say weird things
    - "Can't access memory…" etc.
    - Just ignore it and keep going
- Don't forget the –n flag on the last level

Buflab

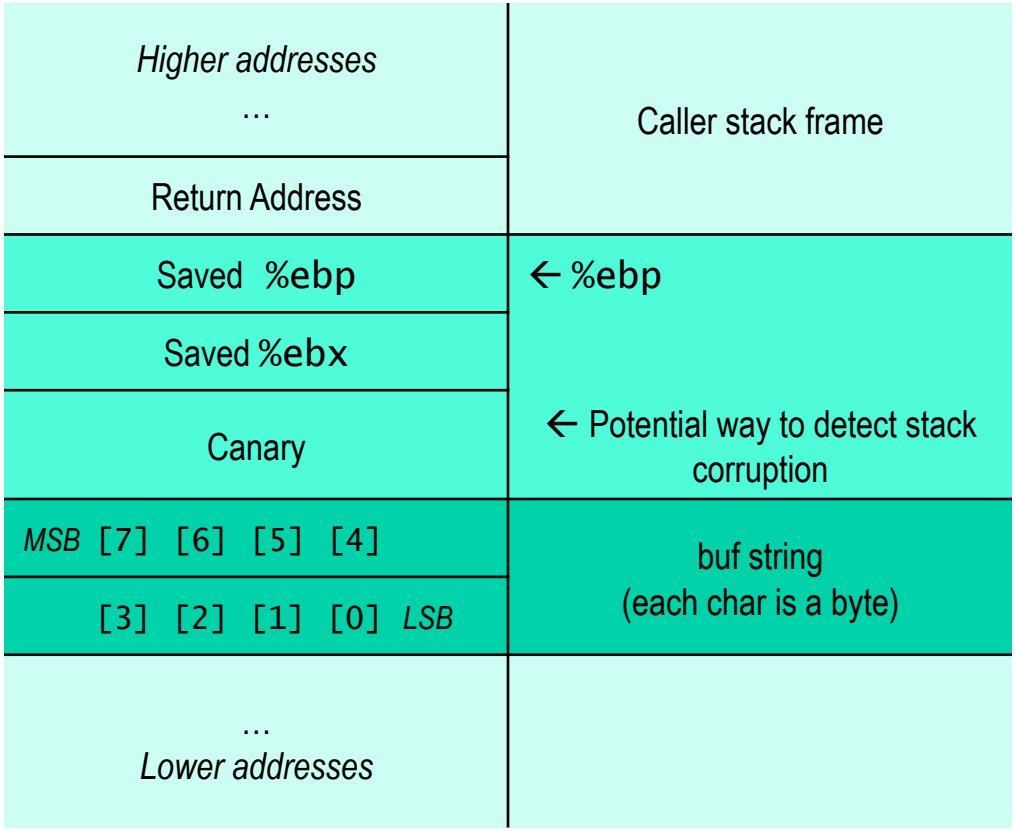
- As always, the writeup is on Autolab, a couple links down from the handout.
- The writeup contains all lab knowledge!!
  - How to use the tools
  - How to write corruption code
  - It even tells you how to solve the level (at a high level)!
- Please don't ask questions answered by the writeup. ☹

Miscellany but Necessary

- **Canaries**
  - Detect overrun buffers
  - Sit at the end of the buffer (array)
  - If the array overflows, *hopefully* we detect this with a change in the canary value….
- **NOP sleds**
  - The nop instruction means "no-op/ no operation"
  - Used to "pad" instructions (or exploits!)
    - Place your exploits at the end of the nop sled.
    - Allows you to be "sloppier" in providing the return address of your exploit.

A Lesson on Endianness

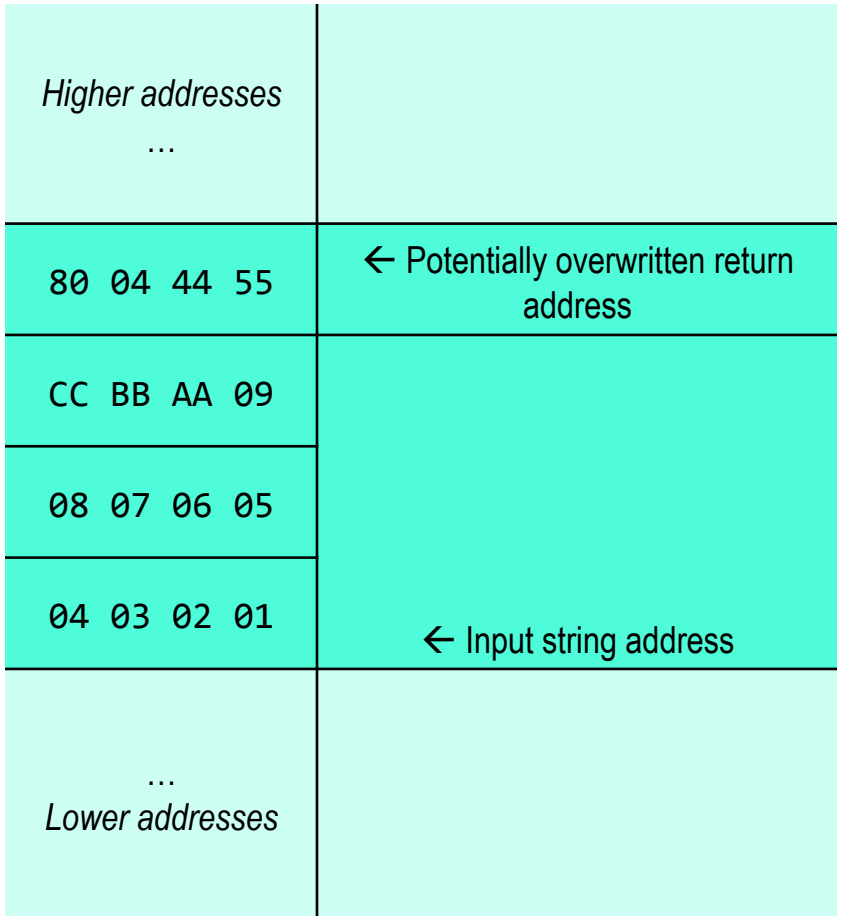
- We're working with **little-endian** machines
  - For primitive data types of more than one byte (e.g., ints), the least significant byte is at the lowest address.



Example of Endianness in Buf Lab

- If we want to input an array of four one-byte characters and later have them interpreted as one four-byte integer, we need to put the bytes in the "opposite" order than we might expect.
- Example byte code input:

```
01 02 03 04
05 06 07 08
09 AA BB CC
55 44 04 80
```



Agenda

- What's New
- Bomb Lab (Sorta): Jump Tables vs. Sparse Switches
- Stacks
  - x86-32 Stack Discipline
  - Function Call Walkthrough
  - Back to x86-64
- Buflab
  - Quick Start
  - Miscellany
- **Exam Question**

Stacks on the Midterm

Problem 8. (10 points):

Stack discipline. Consider the following C code and its corresponding 32-bit x86 machine code. Please complete the stack diagram on the following page.

```
int bar (int a, int b) {
    return a + b;
}

int foo(int n, int m, int c) {
    c += bar(m, n);
    return c;
}
```

```
08048374 <bar>:
 8048374:       55                      push   %ebp
 8048375:       89 e5                   mov    %esp,%ebp
 8048377:       8b 45 0c                mov    0xc(%ebp),%eax
 804837a:       03 45 08                add    0x8(%ebp),%eax
 804837d:       5d                      pop    %ebp
 804837e:       c3                      ret

0804837f <foo>:
 804837f:       55                      push   %ebp
 8048380:       89 e5                   mov    %esp,%ebp
 8048382:       83 ec 08                sub    $0x8,%esp
 8048385:       8b 45 08                mov    0x8(%ebp),%eax
 8048388:       89 44 24 04             mov    %eax,0x4(%esp)
 804838c:       8b 45 0c                mov    0xc(%ebp),%eax
 804838f:       89 04 24                mov    %eax,(%esp)
 8048392:       e8 dd ff ff ff          call   8048374 <bar>
 8048397:       03 45 10                add    0x10(%ebp),%eax
 804839a:       c9                      leave
 804839b:       c3                      ret
```

Stacks on the Midterm

A. Draw a detailed picture of the stack, starting with the caller invoking `foo(3, 4, 5)`, and ending immediately **before** execution of the `ret` instruction in `bar`.

- The stack diagram should begin with the three arguments for `foo` that the caller has placed on the stack. To help you get started, we have given you the first one.
- Use the actual values for function arguments, rather than variable names. For example, use 3 or 4 instead of `n` or `m`.
- Always label `%ebp` and give its value when it is pushed to the stack, e.g., `%ebp: 0xffff1400`.
- You may not need to fill in all of the boxes in the diagram.

```
Value of %ebp when foo is called: 0xffffd858
Return address in function that called foo: 0x080483c9

Stack      The diagram starts with the
addresss   arguments for foo()
           +-----------------------------------+
0xffffd850 |                5                  |
           +-----------------------------------+
0xffffd84c |                                   |
           +-----------------------------------+
0xffffd848 |                                   |
           +-----------------------------------+
0xffffd844 |                                   |
           +-----------------------------------+
0xffffd840 |                                   |
           +-----------------------------------+
0xffffd83c |                                   |
           +-----------------------------------+
0xffffd838 |                                   |
           +-----------------------------------+
0xffffd834 |                                   |
           +-----------------------------------+
0xffffd830 |                                   |
           +-----------------------------------+
```

B. What is the final value of `%ebp`, immediately **before** execution of the `ret` instruction in `bar`?

`%ebp=0x`_____

C. What is the final value of `%esp`, immediately **before** execution of the `ret` instruction in `bar`?

`%esp=0x`_____

Stacks on the Midterm

A. Draw a detailed picture of the stack, starting with the caller invoking `foo(3, 4, 5)`, and ending immediately **before** execution of the `ret` instruction in `bar`.

- The stack diagram should begin with the three arguments for `foo` that the caller has placed on the stack. To help you get started, we have given you the first one.
- Use the actual values for function arguments, rather than variable names. For example, use 3 or 4 instead of `n` or `m`.
- Always label `%ebp` and give its value when it is pushed to the stack, e.g., `%ebp: 0xffff1400`.
- You may not need to fill in all of the boxes in the diagram.

```
Value of %ebp when foo is called: 0xffffd858
Return address in function that called foo: 0x080483c9

Stack      The diagram starts with the
addresss   arguments for foo()
           +-----------------------------------+
0xffffd850 |                5                  |
           +-----------------------------------+
0xffffd84c |                4                  |
           +-----------------------------------+
0xffffd848 |                3                  |
           +-----------------------------------+
0xffffd844 |   Return address: 0x080483c9      |
           +-----------------------------------+
0xffffd840 |      Old ebp: 0xffffd858          |   ← %ebp
           +-----------------------------------+
0xffffd83c |                3                  |
           +-----------------------------------+
0xffffd838 |                4                  |
           +-----------------------------------+
0xffffd834 |    Return address: 08048397       |   ← %esp
           +-----------------------------------+
0xffffd830 |      Old ebp: 0xffffd840          |   ←OK to omit since it's no longer part of the stack
           +-----------------------------------+      after the "leave" instruction
```

B. What is the final value of `%ebp`, immediately **before** execution of the `ret` instruction in `bar`?

`%ebp=0x`_____

C. What is the final value of `%esp`, immediately **before** execution of the `ret` instruction in `bar`?

`%esp=0x`_____

Stolen Credits & Questions Slide

- This whole slide deck was mostly cribbed from Anita Zhang's
- xkcd: Tabletop Roleplaying
- StackOverflow: Supporting Recursion
- StackOverflow: Direction of Stack Growth
- Understanding the SPARC Architecture
- CS:APP p. 220 – Stack Frame Structure
- *Smashing the Stack for Fun and Profit*
- Wikipedia's Big- vs. Little-Endian Graphic
- Cannock Chase Heritage Trail (coal mine canary)
- CS:APP p.262 – NOP sleds
- CS:APP p.263 – Stack Frame with a canary