CSE 351 Section 5

More Stack Stuff

(selected slides by Tom Bergan)

Written HW #2

- Due tomorrow at 5PM
- Try not to use late days on the written assignments, save them for the labs
- Questions?

Stack review

Procedure Call Example

(IA32/Linux)

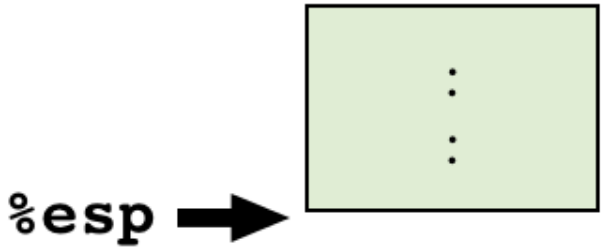
Caller

```
int z = sum(1, 2);
```

Caller in assembly

```
0x8001  pushl $2
0x8005  pushl $1
0x8009  call sum
0x8013  addl $8, %esp
```

The Stack



*note: these instruction addresses are completely made up for this example

Procedure Call Example

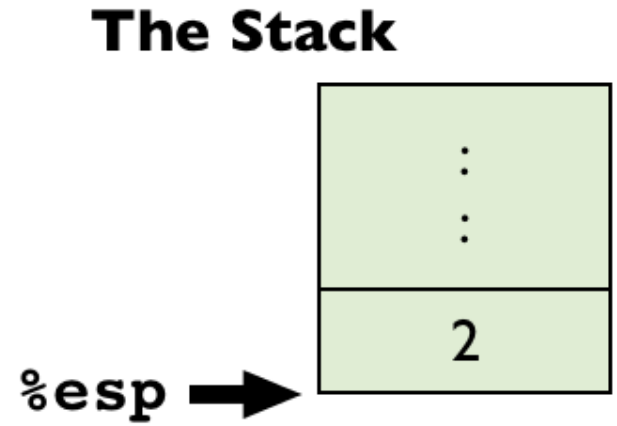
(IA32/Linux)

Caller

```
int z = sum(1, 2);
```

Caller in assembly

```
0x8001   pushl $2
0x8005   pushl $1
0x8009   call sum
0x8013   addl $8, %esp
```



*note: these instruction addresses are completely made up for this example

Procedure Call Example

(IA32/Linux)

Caller

```
int z = sum(1, 2);
```

Caller in assembly

```
    0x8001   pushl $2
→   0x8005   pushl $1
    0x8009   call sum
    0x8013   addl $8, %esp
```

The Stack

*note: these instruction addresses are completely made up for this example

Procedure Call Example

(IA32/Linux)

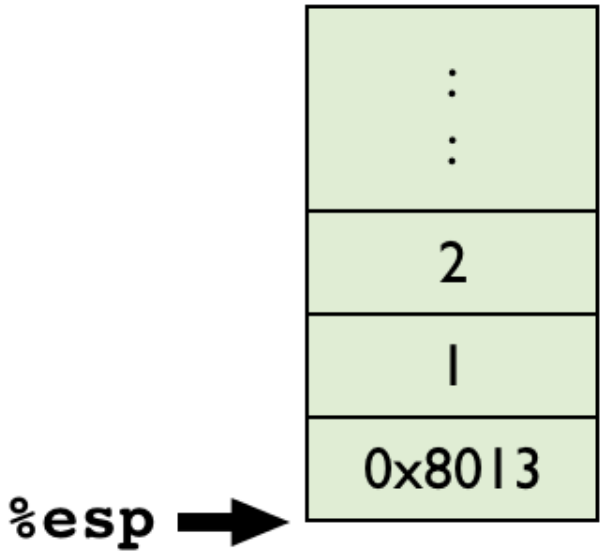
Caller

```
int z = sum(1, 2);
```

Caller in assembly

```
    0x8001   pushl $2
    0x8005   pushl $1
→   0x8009   call sum
    0x8013   addl $8, %esp
```

The Stack



*note: these instruction addresses are completely made up for this example

Procedure Call Example

(IA32/Linux)

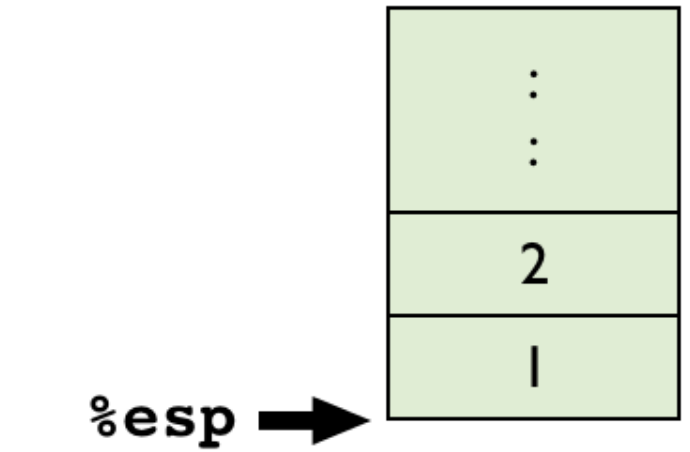
Caller

```
int z = sum(1, 2);
```

Caller in assembly

```
    0x8001   pushl $2
    0x8005   pushl $1
    0x8009   call sum
→   0x8013   addl $8, %esp
```

*note: these instruction addresses are completely made up for this example



Registers

%eax	3
%edi	2
%eip	0x8013

Procedure Call Example

(IA32/Linux)

Caller

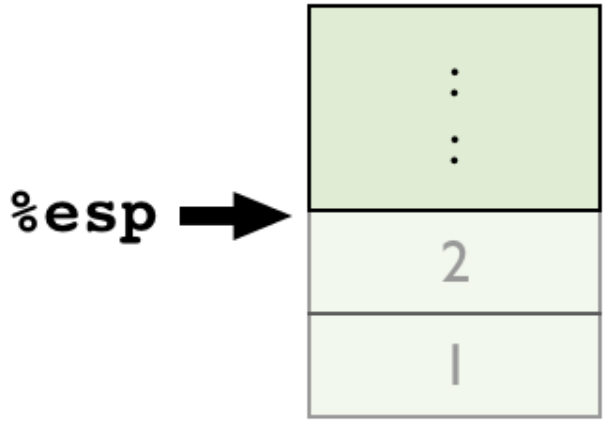
```
int z = sum(1, 2);
```

Caller in assembly

```
    0x8001   pushl $2
    0x8005   pushl $1
    0x8009   call sum
→   0x8013   addl $8, %esp
```

*note: these instruction addresses are completely made up for this example

The Stack



Registers

%eax	3
%edi	2
%eip	0x8013

Procedure Call Example

(IA32/Linux)

Caller

```
int z = sum(1, 2);
```

Problem:

- **What if Caller used %edi before making the call?**

Caller in assembly

```
    0x8001   pushl $2
    0x8005   pushl $1
    0x8009   call sum
➡   0x8013   addl $8, %esp
```

Registers

%eax	3
%edi	2
%eip	0x8013

*note: these instruction addresses are completely made up for this example

Procedure Call Example

(IA32/Linux)

Caller

```
int d = 5;
int z = sum(1, 2);
```

Problem:

- **What if Caller used %edi before making the call?**

Caller in assembly

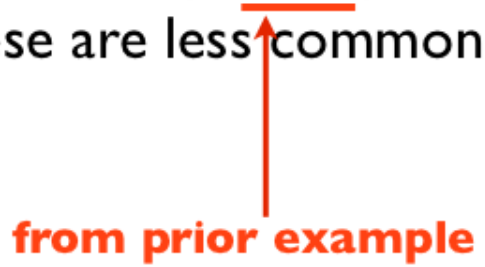
```
0x7fff   movl $5, %edi
0x8001   pushl $2
0x8005   pushl $1
0x8009   call sum
0x8013   addl $8, %esp
```

sum() overwrote %edi!
Need to save ...

*note: these instruction addresses are completely made up for this example

Saving Registers

- Some are ***caller save***
  - IA32: `%eax, %edx, %ecx`
  - These are very commonly used
    (caller should expect they will be clobbered)
- Some are ***callee save***
  - IA32: `%ebx, %edi, %esi`
  - These are less commonly used



Procedure Call Example

(IA32/Linux)

Callee

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

Callee in assembly (better version)

setup

```
pushl %ebp
movl  %esp, %ebp
pushl %edi
```

---

body

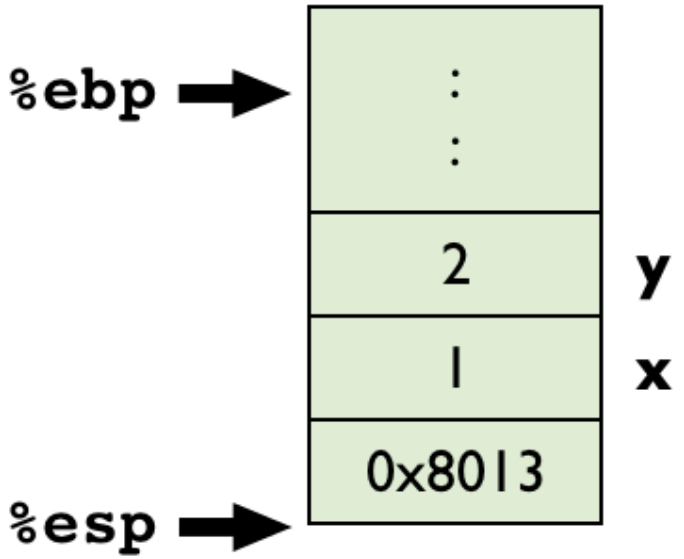
```
movl 12(%ebp), %edi
movl 8(%ebp), %eax
addl %edi, %eax
```

---

cleanup

```
movl (%esp), %edi
movl %ebp, %esp
popl %ebp
ret
```

The Stack



Procedure Call Example

(IA32/Linux)

Callee

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

Callee in assembly (better version)

setup

```
pushl %ebp
movl  %esp, %ebp
pushl %edi
```

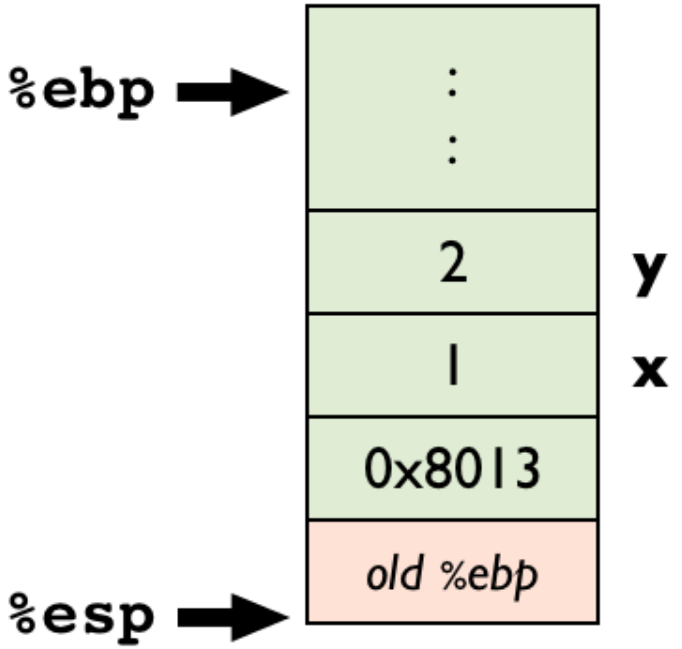
body

```
movl 12(%ebp), %edi
movl 8(%ebp), %eax
addl %edi, %eax
```

cleanup

```
movl (%esp), %edi
movl %ebp, %esp
popl %ebp
ret
```

The Stack



Procedure Call Example

(IA32/Linux)

Callee

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

Callee in assembly (better version)

setup

```
pushl %ebp
movl  %esp, %ebp
pushl %edi
```

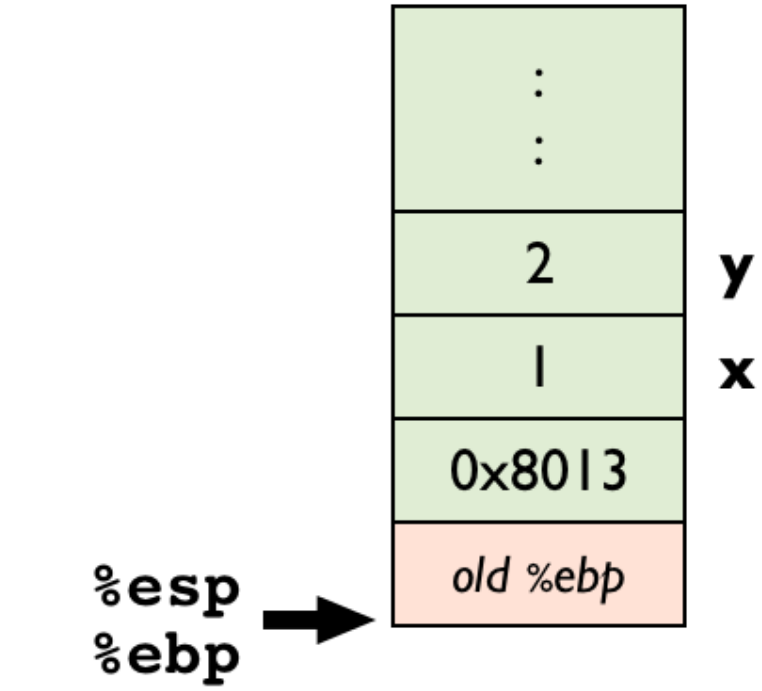
body

```
movl 12(%ebp), %edi
movl 8(%ebp), %eax
addl %edi, %eax
```

cleanup

```
movl (%esp), %edi
movl %ebp, %esp
popl %ebp
ret
```

The Stack



Procedure Call Example

(IA32/Linux)

Callee

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

Callee in assembly (better version)

setup

```
pushl %ebp
movl  %esp, %ebp
pushl %edi
```

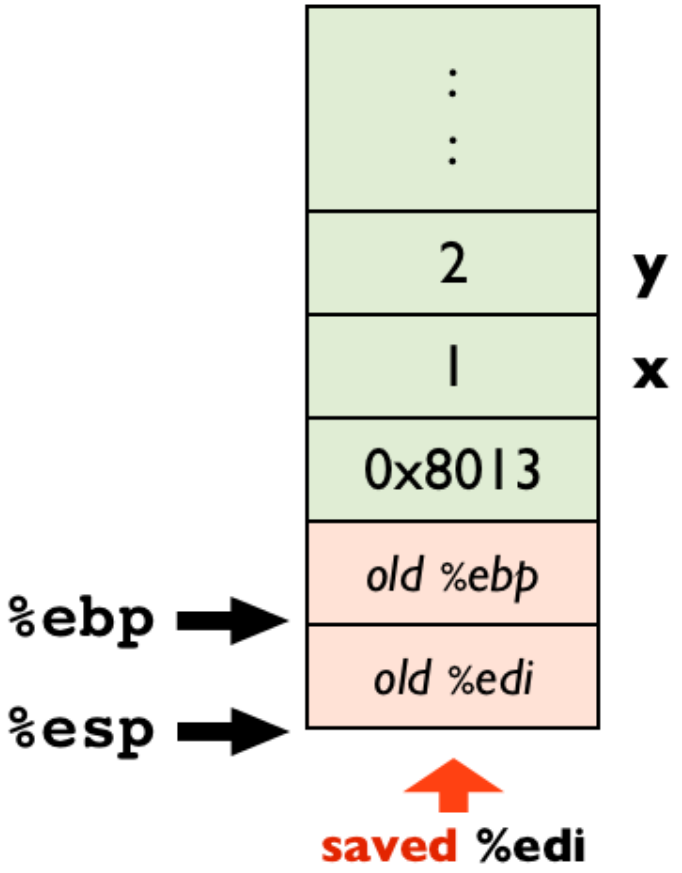
body

```
movl 12(%ebp), %edi
movl 8(%ebp), %eax
addl %edi, %eax
```

cleanup

```
movl (%esp), %edi
movl %ebp, %esp
popl %ebp
ret
```

The Stack



Procedure Call Example

(IA32/Linux)

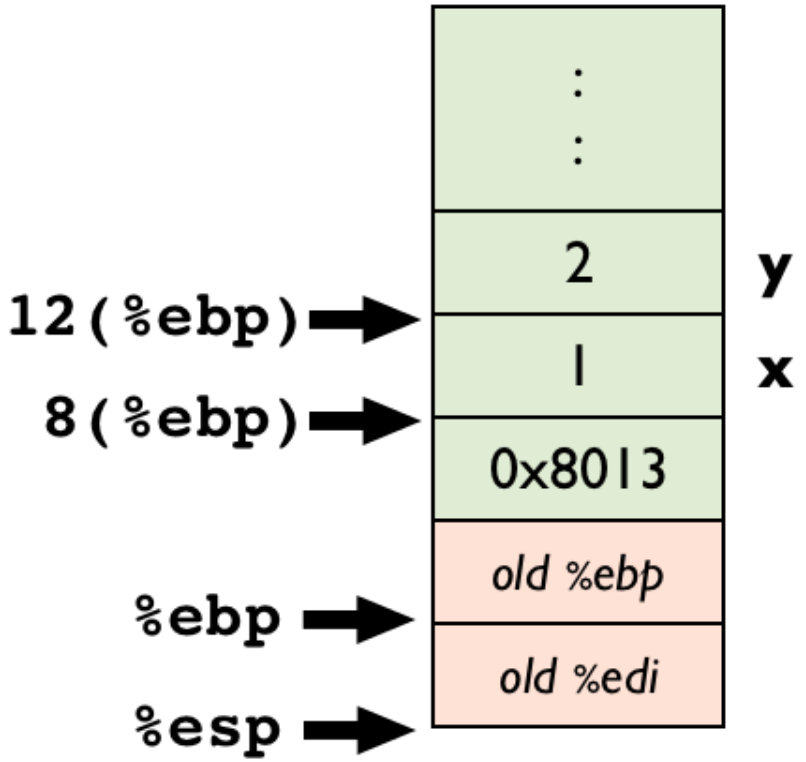
Callee

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

Callee in assembly (better version)

```
setup     pushl %ebp
          movl  %esp, %ebp
          pushl %edi
-------------------------------
          movl 12(%ebp), %edi
body      movl 8(%ebp), %eax
          addl %edi, %eax
-------------------------------
          movl (%esp), %edi
cleanup   movl %ebp, %esp
          popl %ebp
          ret
```

The Stack



Key: %ebp is fixed for the entire function

Procedure Call Example

(IA32/Linux)

Callee

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

Callee in assembly (better version)

setup

```
pushl %ebp
movl  %esp, %ebp
pushl %edi
```

body

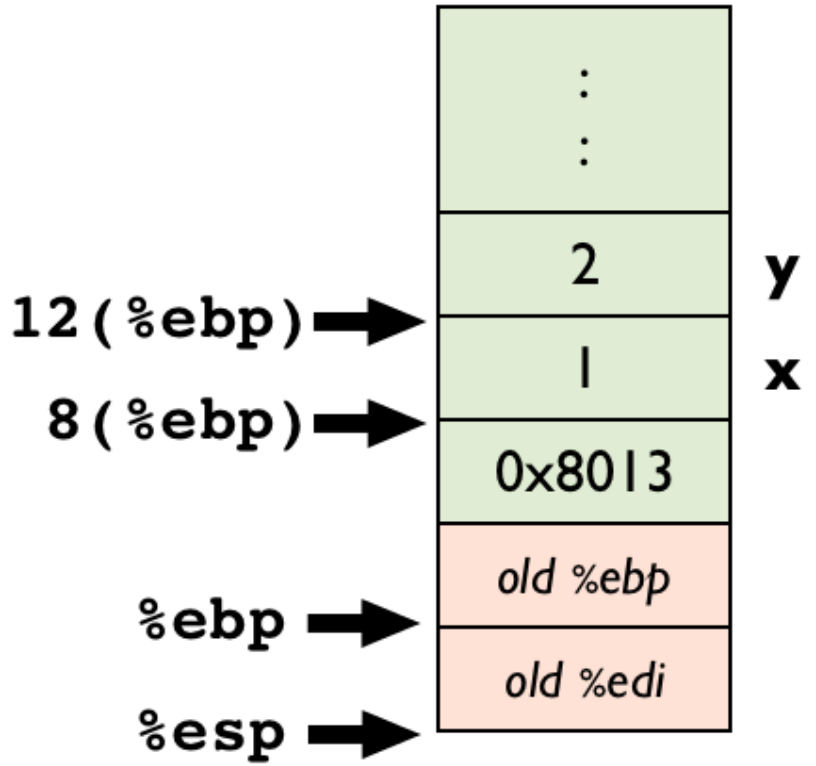
```
movl 12(%ebp), %edi
movl 8(%ebp), %eax
addl %edi, %eax
```

cleanup

```
movl (%esp), %edi
movl %ebp, %esp
popl %ebp
ret
```

restoring %edi

The Stack



Procedure Call Example

(IA32/Linux)

Callee

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

Callee in assembly (better version)

setup

```
pushl %ebp
movl  %esp, %ebp
pushl %edi
```

body

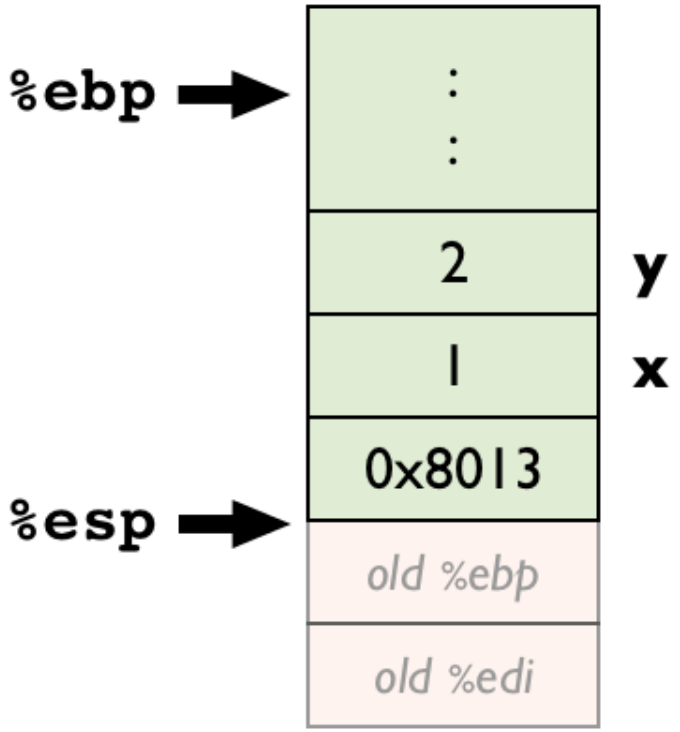
```
movl 12(%ebp), %edi
movl 8(%ebp), %eax
addl %edi, %eax
```

cleanup

```
movl (%esp), %edi
movl %ebp, %esp
popl %ebp
ret
```

restoring %ebp

The Stack



Why use a frame pointer?

(%ebp)

Callee

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

To make debugging easier

- %esp may move
- %ebp is fixed

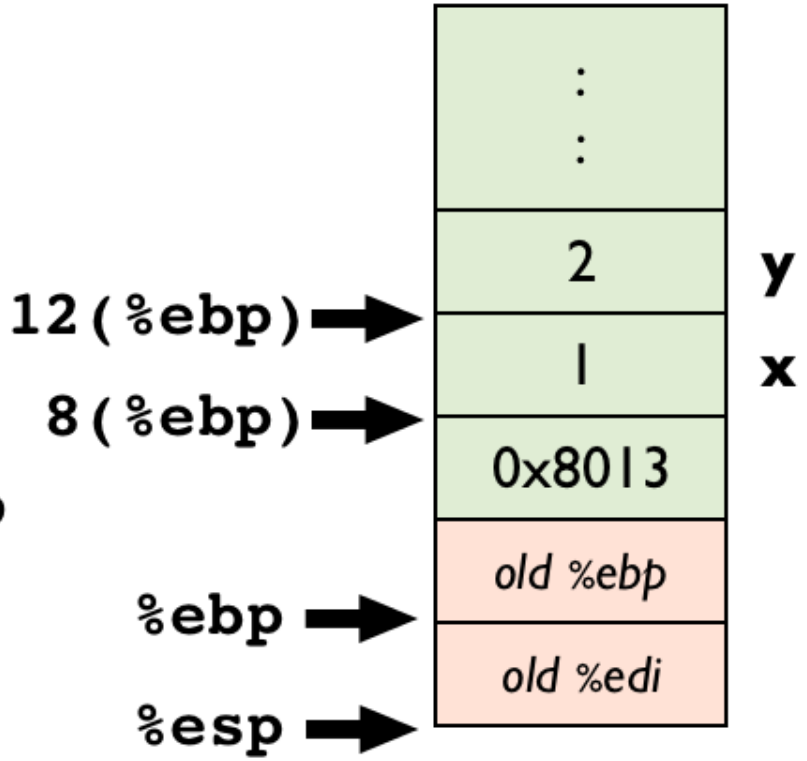
Your compiler emits a symbol map

```
y → 12(%ebp)
x → 8(%ebp)
```

gdb uses this map when you write

```
print x
```

The Stack



How is x86-64 different?

- Pass the first six arguments *in registers*
  - In this order: `%rdi,%rsi,%rdx,%rcx,%r8,%r9`
- New register save convention
  - *Callee save*: `%rbx,%rbp,%r12,%r13,%r14,%r15`
  - Others are *caller save*
- By default, gcc omits the frame pointer
  - It has to emit more complex debug info
    (e.g., the location of argument x relative to %esp can change)

Procedure Call Example

(x86-64/Linux)

Caller

```
int z = sum(1, 2);
```

Caller in assembly

```
movl $1, %edi
movl $2, %esi
call sum
```

edi not rdi because int is 32-bits

Callee

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

Callee in assembly

```
addl %esi, %edi
movl %edi, %eax
ret
```

x86-64 with gcc does not use a frame pointer

Tip: you can force gcc to emit code with a frame pointer using

```
gcc -fno-omit-frame-pointer
```

Lab 3 - Buffer Overflows

Bufbomb Introduction

- Several stages
- Practice analyzing stack organization
- Practice with buffer overflows

Bufbomb Introduction

GDB commands from today:

```
gdb bufbomb
set args -u <username>
break getbuf
run
disas
display $rsp
stepi
x /40x ($rsp - 40)        (show the 40 bytes above rsp)
break *(&getbuf + 12)   (create a breakpoint at 12 bytes
                        after the start of getbuf)
continue
```

Lab 3: Buffer Overflow

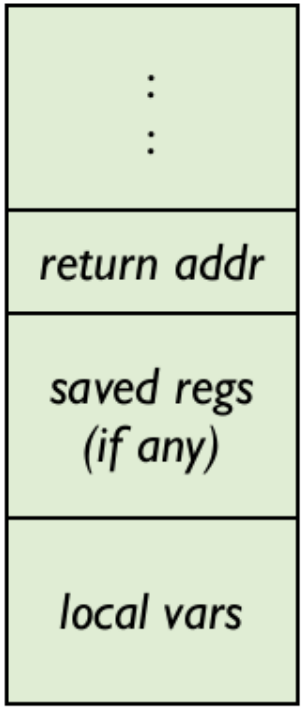
This has a buffer overflow

```
int getbuf() {
  char buf[36];
  Gets(buf);
  return 1;
}
```

Why?

- `Gets()` doesn't check the length of the buffer

The Stack in getbuf()

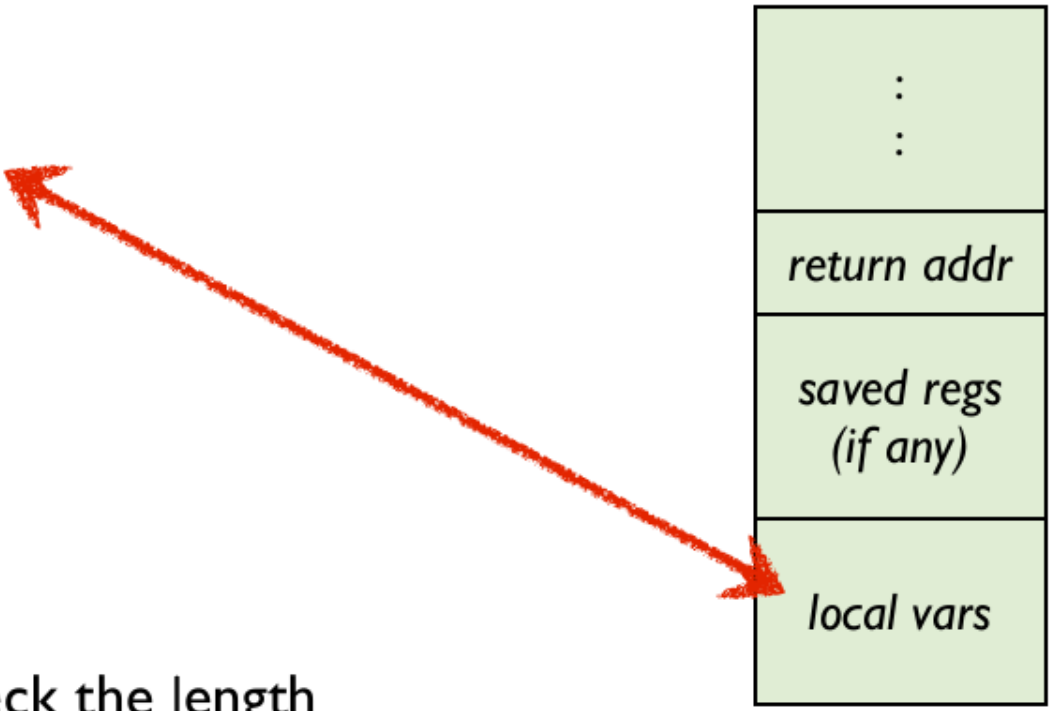


Lab 3: Buffer Overflow

This has a buffer overflow

```
int getbuf() {
  char buf[36];
  Gets(buf);
  return 1;
}
```

The Stack in getbuf()



Why?

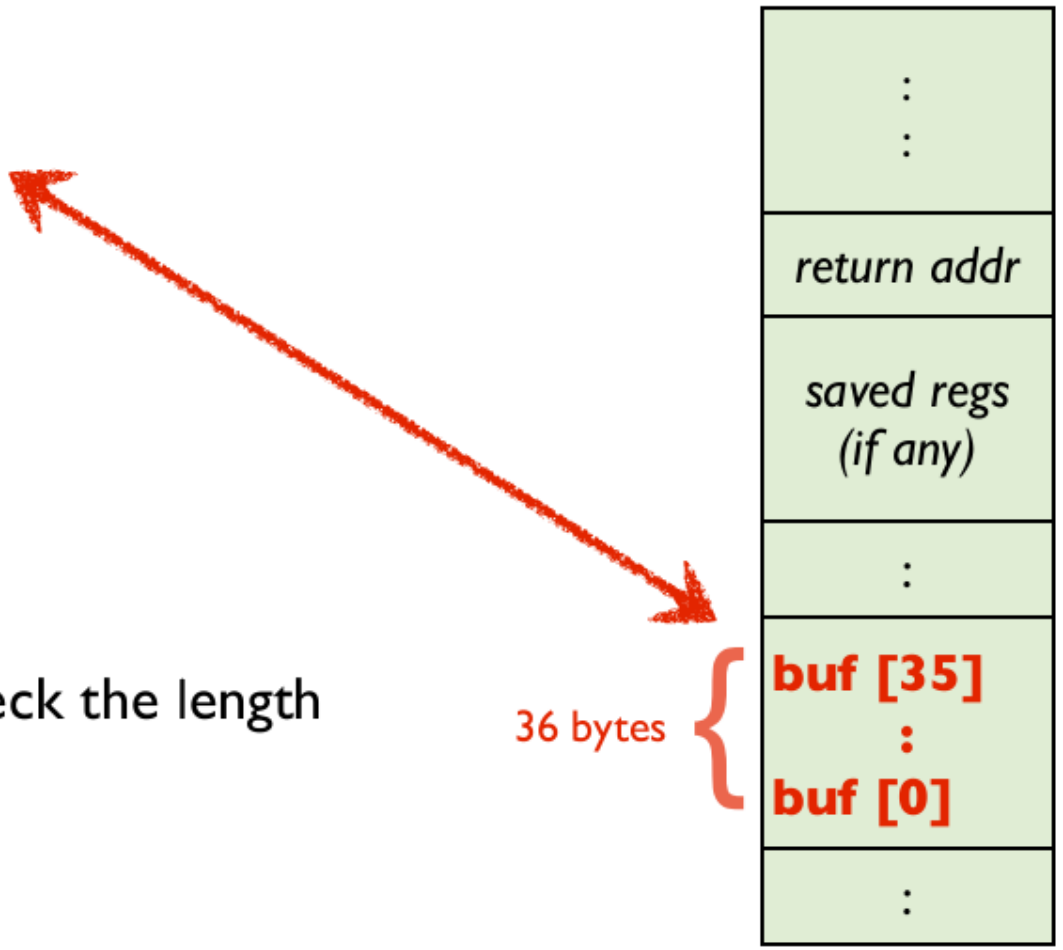
- `Gets()` doesn't check the length of the buffer

Lab 3: Buffer Overflow

This has a buffer overflow

```
int getbuf() {
  char buf[36];
  Gets(buf);
  return 1;
}
```

The Stack in getbuf()



Why?

- `Gets()` doesn't check the length of the buffer

Level 0: Call `smoke()`

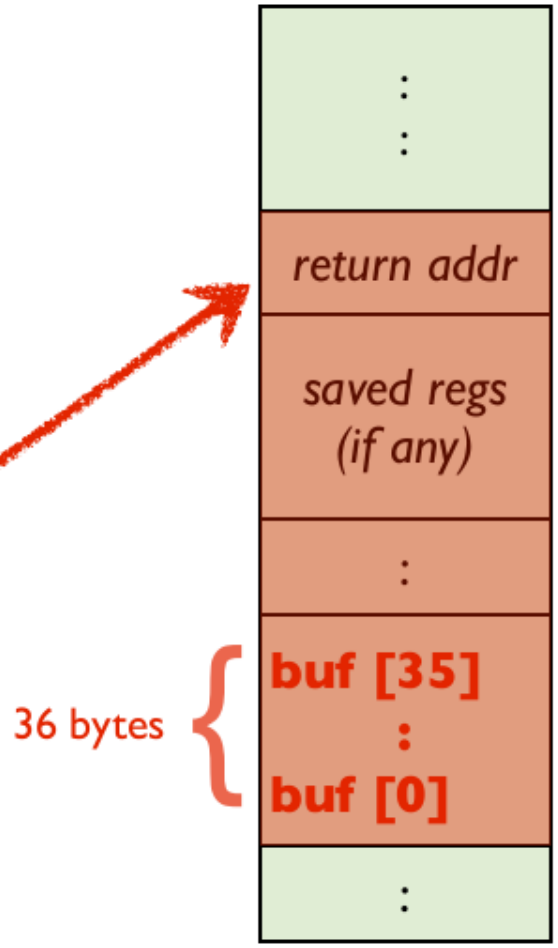
Goal: call the smoke() function from getbuf()

```
int getbuf() {
  char buf[36];
  Gets(buf);
  return 1;
}
```

How?

- overwrite the return address so we "return" to smoke()

The Stack in getbuf()

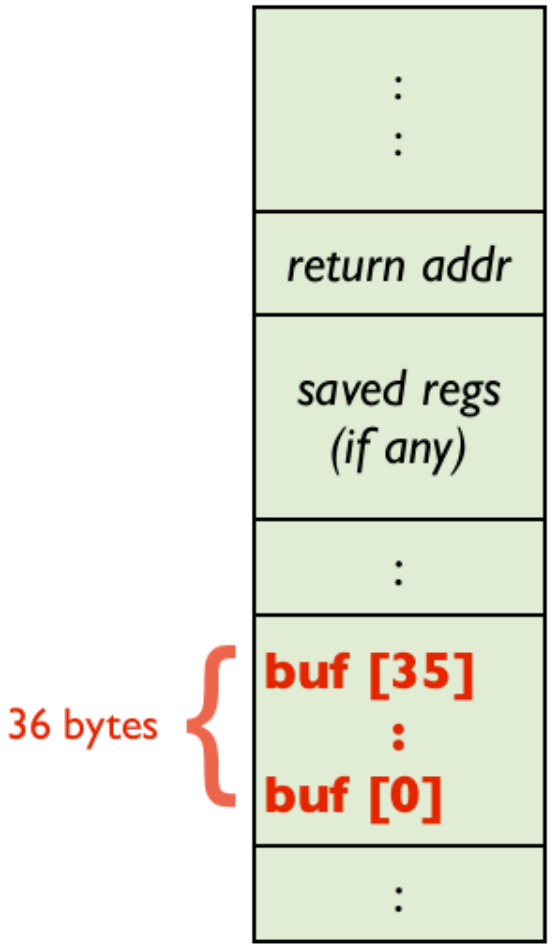


Level 1: Call `fizz()`

Goal: call fizz() with a special parameter (your "cookie")

```
int getbuf() {
  char buf[36];
  Gets(buf);
  return 1;
}
```

The Stack in getbuf()



Level 2: Call `bang()`

Goal: call bang() after writing your "cookie" to a global variable

```
int getbuf() {
  char buf[36];
  Gets(buf);
  return 1;
}
```

How?

1. overwrite the return address
2. jump *inside the buffer*
3. write x86 code in the buffer

The Stack in getbuf()