Cybersecurity – EC521
Memory Corruption

Manuel Egele
PHO 337
megele@bu.edu
Boston University

Slides from William Robertson @ NEU

Outline

Assembly Review

Vulnerabilities I

Vulnerabilities II

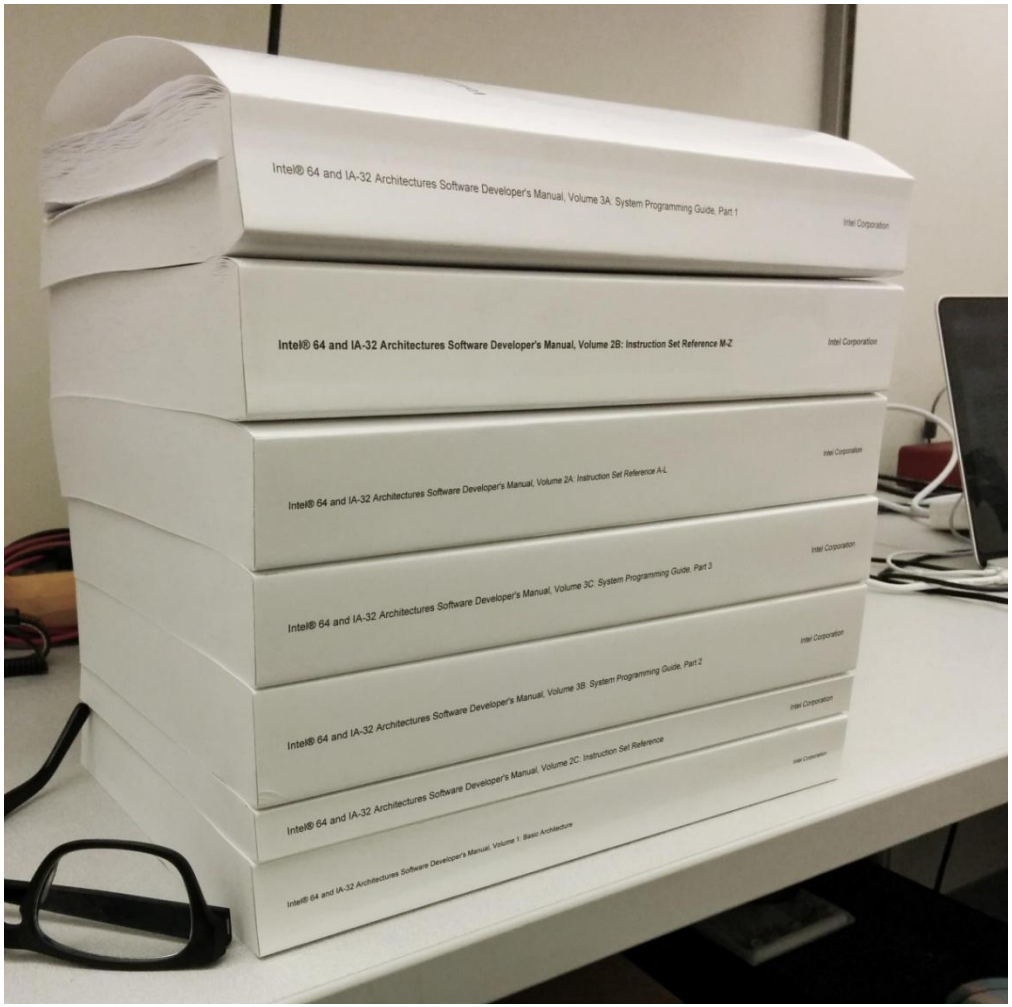
Defenses & Evasion of Defenses

Malware Analysis

Assembly Review

1. Correspondence between a (relatively) high-level language (C) and assembly
2. System components
   - CPU
   - Memory
3. Instructions
   - Formats
   - Classes
   - Control flow
4. Procedures

A Deep Topic


Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1
Intel Corporation
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference M-Z
Intel Corporation
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference A-L
Intel Corporation
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3
Intel Corporation
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2
Intel Corporation
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C: Instruction Set Reference
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture

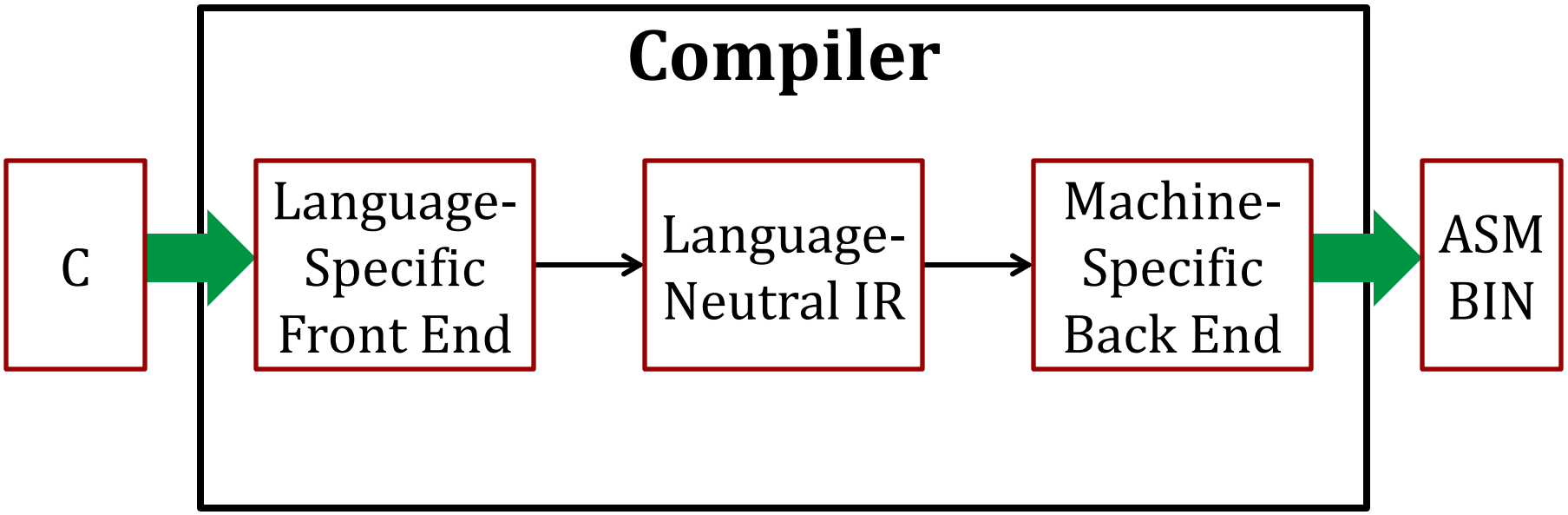
Compilers

```
int abs(int x) {
  if (x < 0)
    return -x;
  return x;
}
```

Computers don't execute source code (doh!)

- Instead, they operate on machine code
- *Compilers* translate code from a higher level to a lower level
- Today: C → assembly → machine code

Compilers


Compiler
C
Language-Specific Front End
Language-Neutral IR
Machine-Specific Back End
ASM BIN

Assembly

Human-readable machine code

- Simple translation to machine code

We will focus on x86/x86_64

- (Externally) CISC architecture
- Instructions have side effects

Assembly syntaxes

- Intel: `<mnemonic> <dst>, <src>`
- AT&T: `<mnemoic> <src>, <dst>`
- Examples will be in Intel syntax

Compilers

```
int abs(int x) {
  if (x < 0)
    return -x;
  return x;
}
```

C → ***assembly*** → machine code

Compilation (–O0 vs. –O3)

```
abs:
    pushrbp
    mov rbp, rsp
    mov dword ptr [rbp - 8], edi
    cmp dword ptr [rbp - 8], 0
    jge .LBB0_2

    mov eax, 0
    sub eax, dword ptr [rbp - 8]
    mov dword ptr [rbp - 4], eax
    jmp .LBB0_3
.LBB0_2:
    mov eax, dword ptr [rbp - 8]
    mov dword ptr [rbp - 4], eax
.LBB0_3:
    mov eax, dword ptr [rbp - 4]
    pop rbp
    ret
```

```
abs:
    mov  eax, edi
    neg  eax
    cmovl  eax, edi
    ret
```

Modern compilers are relatively sophisticated

CPU and Memory

Von Neumann Architecture

- Co-mingled code and data in memory
- Shared bus for code and data

CPU has program counter, points to (next) instruction

- Execution through repeated instruction cycles (fetch -- decode -- execute)
- Usually pipelined in modern architectures

Instruction set architectures (ISAs) have operational semantics

- Given an input state (registers, memory), will produce a well-defined output state

CPU and Memory

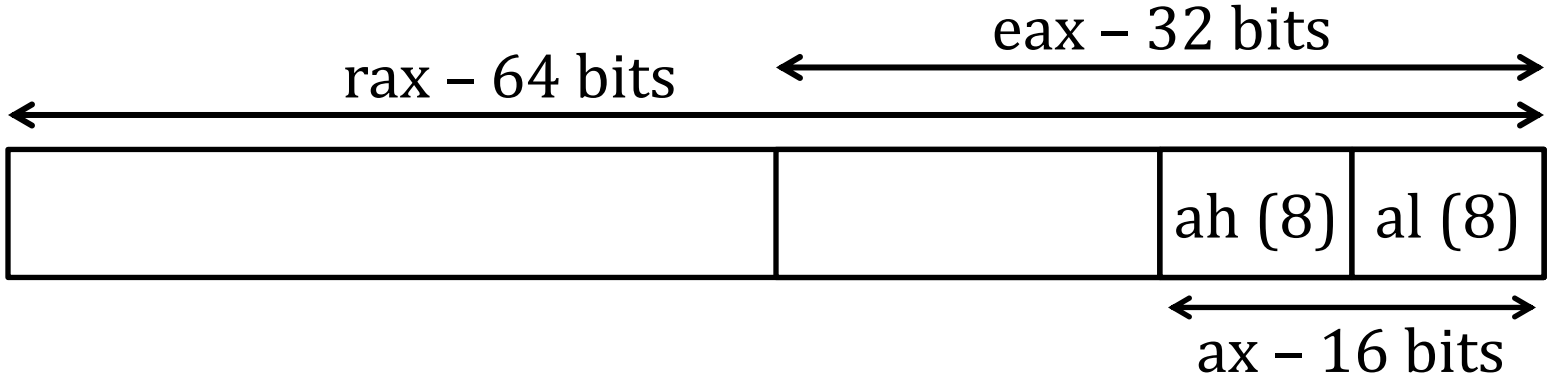
CPU contains registers

- Program counter (rip)
- Stack pointer (rsp)
- Frame pointer (rbp)
- General purpose registers
  - rax, rbx, rcx, rdx, rsi, rdi, r8-r15
- Contition codes (RFLAGS)
  - Affected by arthmetic and logical operations

Memory stores code and data

- Byte-addressable array

Registers



Convention:

- **`rax`** Accumulator
- **`rbx`** Pointer to data
- **`rcx`** Loop counter
- **`rdx`** I/O operations
- **`rdi`** Destination pointer (loops)
- **`rsi`** Source pointer (loops)

Flags

OF Overflow flag
DF Direction flag (loops)
SF Sign flag
ZF Zero flag
PF Parity flag
CF Carry flag

Bit vector of flags (RFLAGS)

- Automatically set and tested by instructions
- Many other fields

Executable File Format

Most common on Linux is ELF (man 5 elf)

- PE on Windows

Header

- Type (executable, library), architecture, offset of segment and section headers, etc.

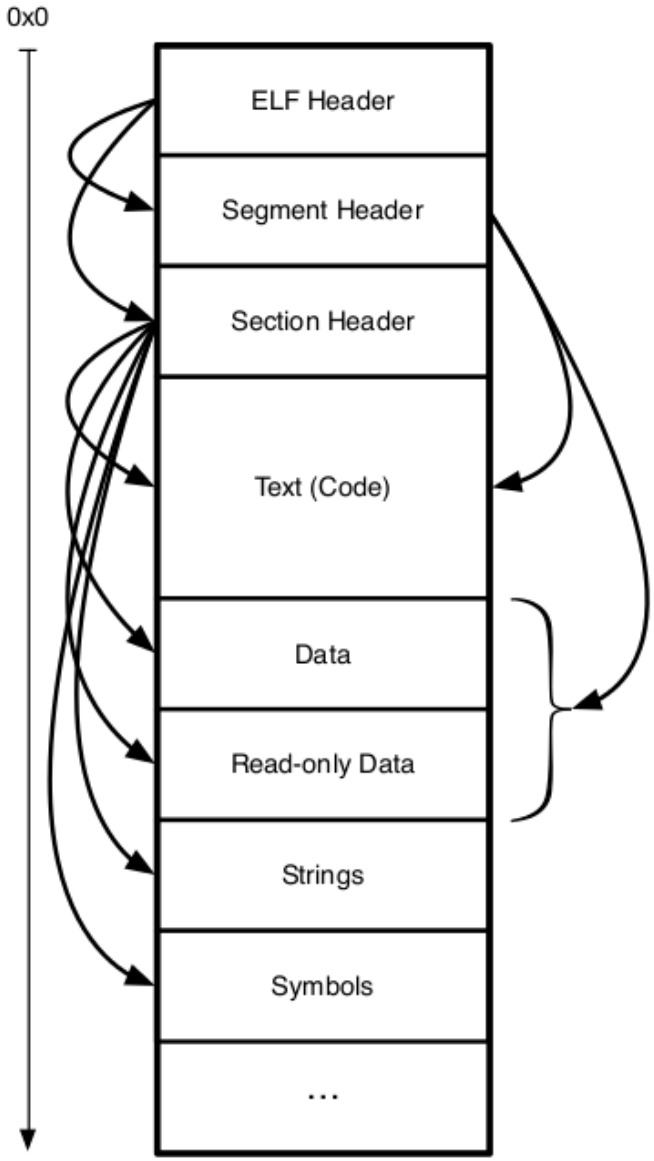
Segments

- Chunk of code, data necessary for execution
- Offset, size, type, virtual address

Sections

- Code, data, relocation info, symbols, debug info, etc.
- Offset, size, type

Structure of an ELF File



Memory Layout

Runtime loader is responsible for loading (usually) multiple binary objects into virtual memory

- Executable
- Libraries

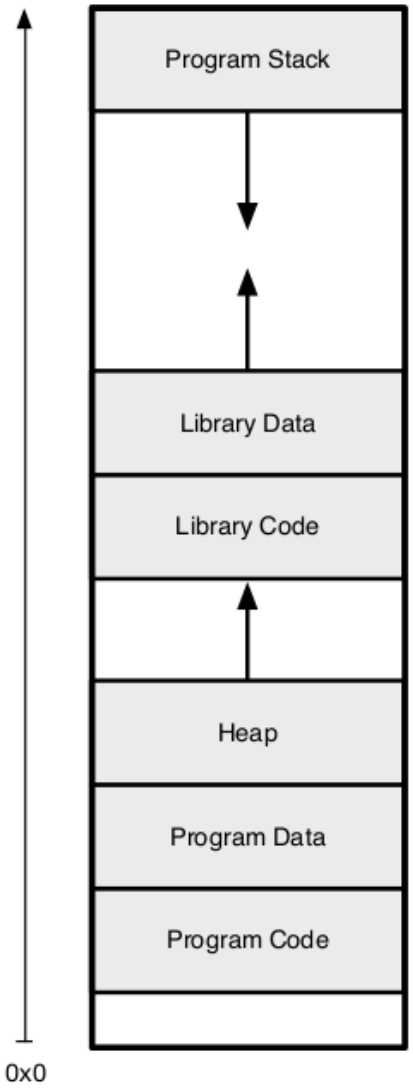
Heap

- Data segment for dynamic data
- Grows upwards, limit controlled by brk()

Stack

- Data segment for procedure-local data, control information
- Grows downwards (on the x86 family)

Memory Layout



Instruction Components

Mnemonic	Operands
nop	empty
neg	rax
add	rax, 0x10
mov	rax, rdx
mov	rax, byte[rdx]
mov	rax, dword[rdx+rcx*4]
jmp	0x08042860
jmp	[rdi]

Operand Types

Literal

- Integer constant directly encoded into the instruction
- e.g., `mov rax, 0x0` (constant 0 moved into rax)

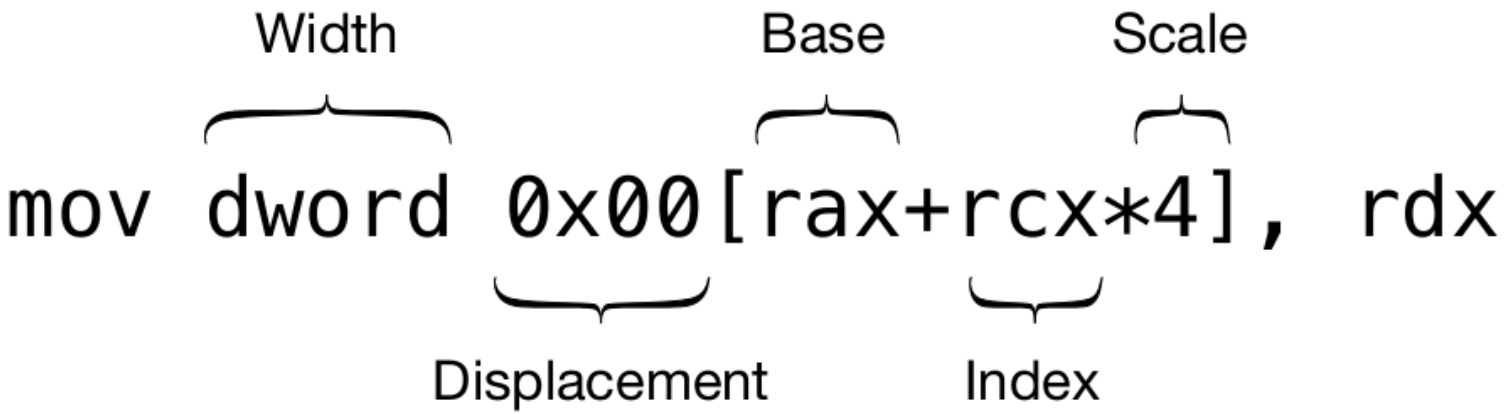
Register

- Contents of a named register
- e.g., `mov rax, rdx` (rdx is moved into rax)

Memory

- Memory reference
- e.g., `mov rax, dword [rdx]`

Memory References



Base Base of reference, register

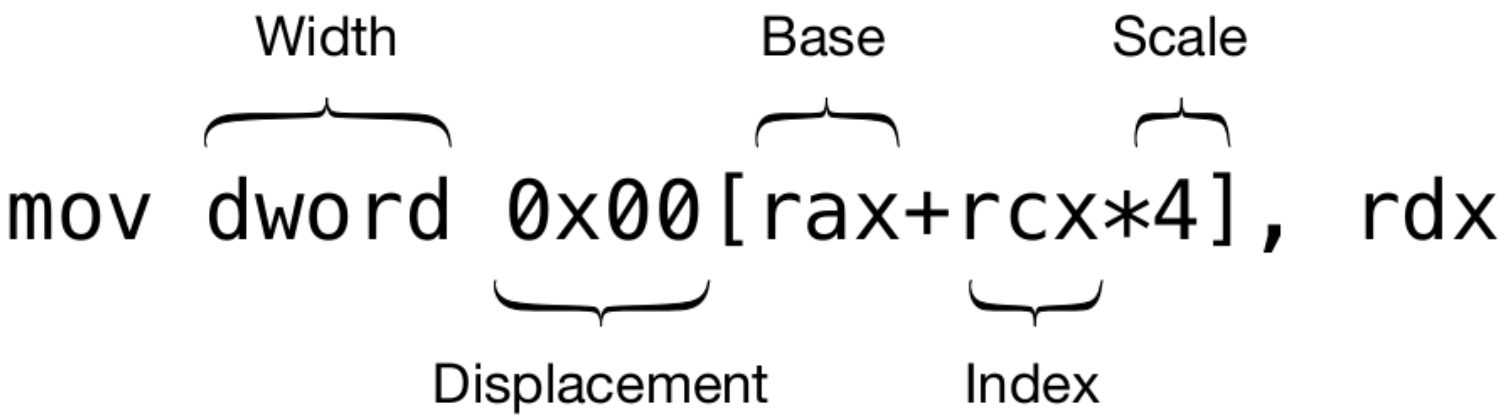
Index Offset from base

Scale Constant that scales index from base

Disp. Base of reference, constant

Width Scales reference

Common Widths



byte Obvious

word 16 bits

dword 32 bits (double word)

qword 64 bits (quad word)

Memory References

For example, this C snippet

```
int data[8]
…
data[1] = 4;
```

might translate to

```
lea rax, [rbp-0x40]
mov rdx, 0x04
mov rcx, 0x01
mov dword [rax + rcx * 4], rdx
```

Instruction Classes

Instructions groups into different classes

- Load/store
- Arithmetic
- Logic
- Comparison
- Control transfer

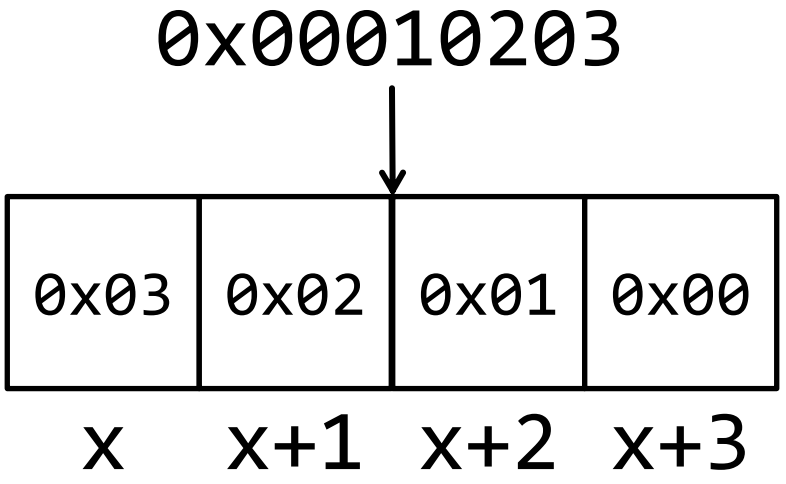
We'll go through a few common examples for each

- Impossible to cover everything here
- Compile programs, disassemble the output or capture assembly, and investigate *yourself*!
- RTFM! (read the ***fine*** manual)

Common Loads, Stores

Instruction	Effect	Description
`mov y, x`	y ← x	Move x to y
`movsx y, x`	y ← SignEx(x)	Move sign-extended x to y
`movzx y, x`	y ← ZeroEx(x)	Move zero-extended x to y
`push x`	rsp ← rsp – 8	Decrement rsp by 8
	Mem(rsp) ← x	Store x on stack
`pop x`	x ← Mem(rsp)	Load top of stack in x
	rsp ← rsp + 8	Increment rsp by 8
`lea y, x`	y ← Addr(x)	Store address of x to y

Endian-ness



The x86 family is a little-endian architecture

- Multi-byte values stored least-significant byte first

If you have a `uint8_t*` pointer to address x

- What is the value for x[0]? x[3]?

Types and the Lack Thereof

Memory at this level: Just a chunk of bytes

- No primitive types, structs

Data can have multiple interpretations

- Signed or unsigned?
- Store a 32-bit value, read back a 16-bit value
- Overlapping loads, stores

Common Arithmetic

Instruction	Effect	Description
`add y, x`	$y \leftarrow y + x$	Add x to y
`sub y, x`	$y \leftarrow y - x$	Subtract x from y
`mul x`	$y \leftarrow rax \times x$	Signed multiply of rax and x
	$rdx \leftarrow High(r)$	High bits stored in rdx
	$rax \leftarrow Low(r)$	Low bits stored in rax
`div x`	$r \leftarrow \frac{rdx:rax}{x}$	Divides rdx:rax by x
	$rdx \leftarrow Rem(r)$	Remainder stored in rdx
	$rax \leftarrow Quo(r)$	Quotient stored in rax

Common Logic

Instruction	Effect	Description
`and y, x`	$y \leftarrow y \wedge x$	Logical AND stored in y
`or y, x`	$y \leftarrow y \vee x$	Logical OR stored in y
`xor y, x`	$y \leftarrow x \oplus y$	Logical XOR stored in y
`shl y, x`	y ← ShiftLeft(y, x)	Shift y left by x bits
`shr y, x`	y ← ShiftRight(y,x)	Shift y right by x bits
`sal x`	y ← SShiftLeft(y,x)	Signed left shift
`rol y, x`	y ← RotateLeft(y,x)	Rotates y left by x bits

Comparison

Instruction	Effect	Description
`test y, x`	$t \leftarrow y \wedge x$	Performs logical AND
	$SF \leftarrow MSB(t)$	Sets SF if MSB set in result
	$ZF \leftarrow t \stackrel{?}{=} 0$	Sets ZF if result is 0
	…	
`cmp y, x`	$t \leftarrow y - x$	Performs signed subtraction
	$SF \leftarrow MSB(t)$	Sets SF if MSB set in result
	$ZF \leftarrow t \stackrel{?}{=} 0$	Sets ZF if result is 0
	…	

Control Transfers

Control transfers change control flow of programs

- Can be predicated on results of a previous comparison (flag bits)
- Arithmetic, logic instructions also set flags (omitted before for brevity)

Distinction between *jumps* and *calls*

- Jumps simply transfer control with no side effects
- Calls used to implement procedures

Distinction between *direct* and *indirect* transfers (*indirect* also known as computed transfers)

- Direct transfers use relative offsets, indirect transfers are absolute (through a register or memory reference)

Instruction Side Effects

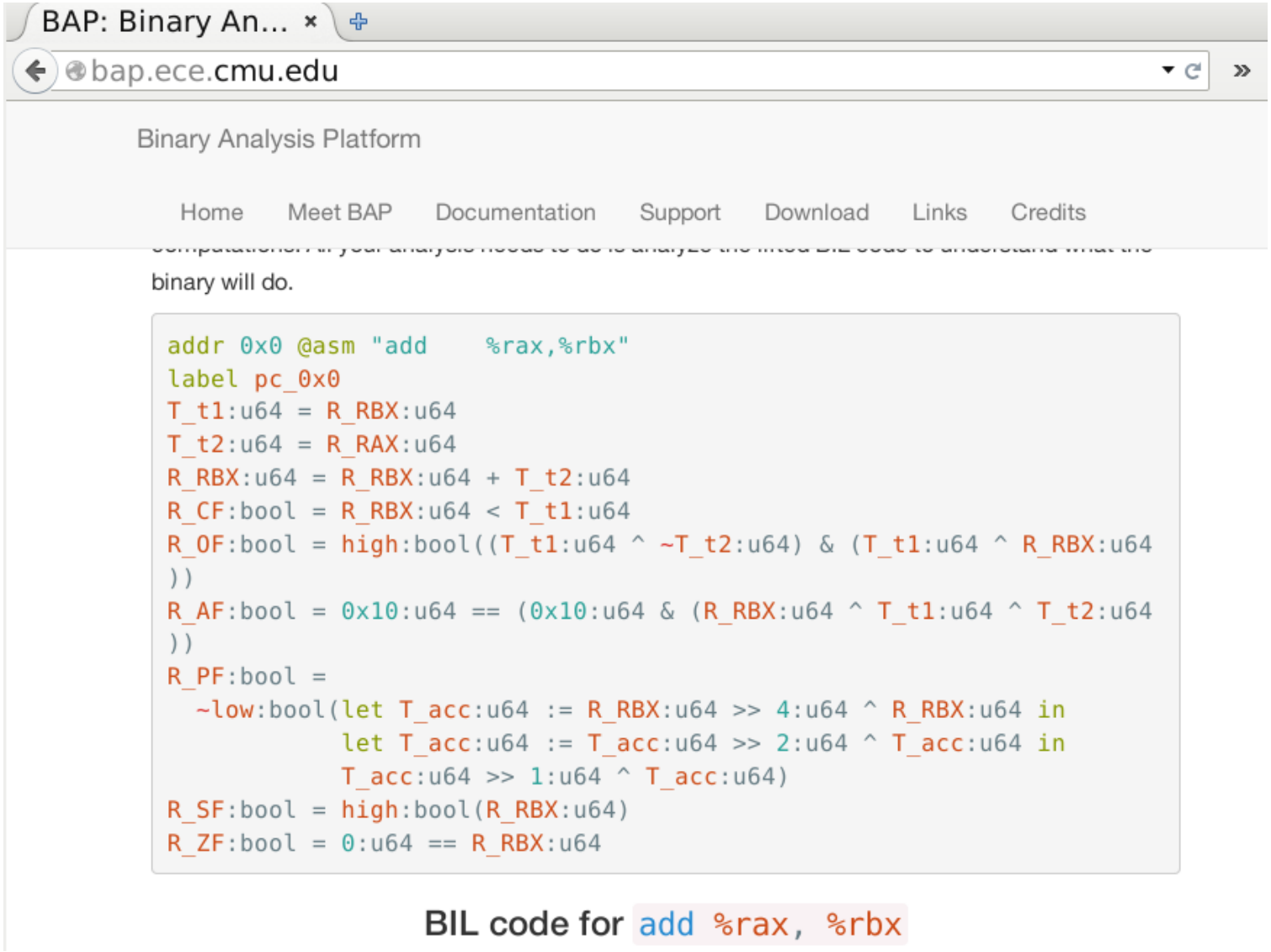
Described in the Intel Programmers Manual

- Human readable text
- Hard to parse/understand for computer programs
- Sometimes incomplete

Use the BAP (Binary Analysis Platform)

- BAP features the BAP intermediate language (BIL)
- Make all side effects explicit
- Supports most x86/x86_64 instructions
- Lacks support for some “obscure” SSE instructions

BIL Example


BAP: Binary An...
bap.ece.cmu.edu
Binary Analysis Platform
Home
Meet BAP
Documentation
Support
Download
Links
Credits
binary will do.
addr 0x0 @asm "add %rax,%rbx"
label pc_0x0
T_t1:u64 = R_RBX:u64
T_t2:u64 = R_RAX:u64
R_RBX:u64 = R_RBX:u64 + T_t2:u64
R_CF:bool = R_RBX:u64 < T_t1:u64
R_OF:bool = high:bool((T_t1:u64 ^ ~T_t2:u64) & (T_t1:u64 ^ R_RBX:u64
))
R_AF:bool = 0x10:u64 == (0x10:u64 & (R_RBX:u64 ^ T_t1:u64 ^ T_t2:u64
))
R_PF:bool =
~low:bool(let T_acc:u64 := R_RBX:u64 >> 4:u64 ^ R_RBX:u64 in
let T_acc:u64 := T_acc:u64 >> 2:u64 ^ T_acc:u64 in
T_acc:u64 >> 1:u64 ^ T_acc:u64)
R_SF:bool = high:bool(R_RBX:u64)
R_ZF:bool = 0:u64 == R_RBX:u64
BIL code for add %rax, %rbx

Control Transfers (Part I)

Instruction	Condition	Description
`jmp x`	unconditional	Direct or indirect jump
`je/jz x`	ZF	Jump if equal
`jne/jnz x`	¬ ZF	Jump if not equal
`jl x`	SF ⊕ OF	Jump if less (signed)
`jle x`	(SF ⊕ OF) ∨ ZF	Jump if less or equal
`jg x`	¬ (SF ⊕ OF) ∧ ¬ ZF	Jump if greater (signed)
`jb x`	CF	Jump if below (unsigned)
`ja x`	¬ CF ∧ ¬ ZF	Jump if above (unsigned)
`js x`	SF	Jump if negative

Procedures

```
int f(int x) {return   x + 1; }
int g(int x) {return    f(x); }
int h(int x) {return f(x *2); }
```

Procedures (functions) are intrinsically linked to the stack

- Provides space for local variables
- Records where to return to
- Used to pass arguments (sometimes)

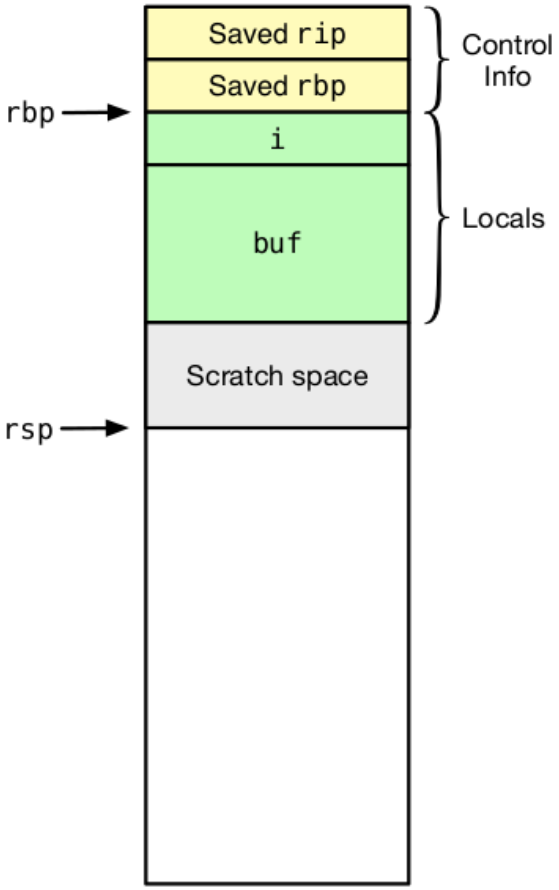
Implemented using stack frames

- Also known as activation records

Control Transfers (Part II)

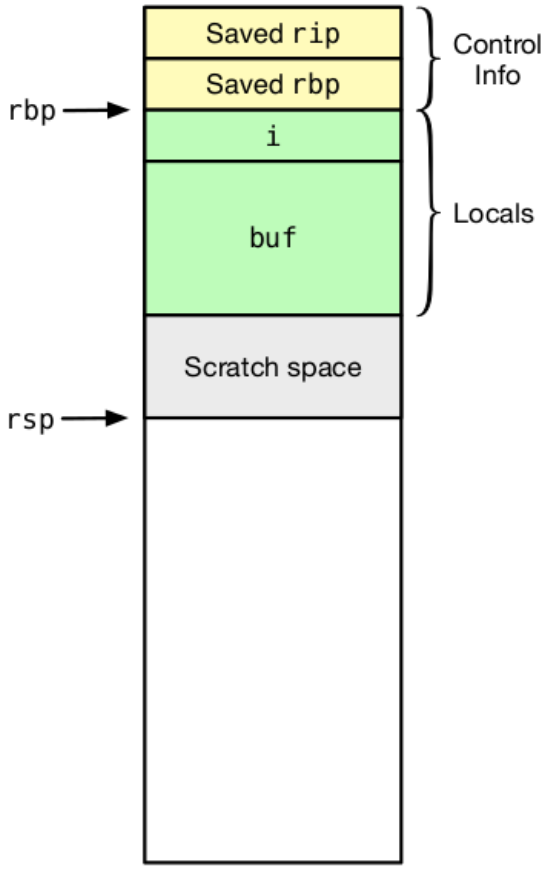
Instruction	Effect	Description
`call x`	rsp ← rsp - 8	Decrement rsp by 8
	Mem(rsp) ← Succ(rip)	Store successor
	rip ← Addr(x)	Jump to address
`ret`	rip ← Mem(rsp)	Pop successor into rip
	rsp ← rsp + 8	Increment rsp by 8

Stack Frame



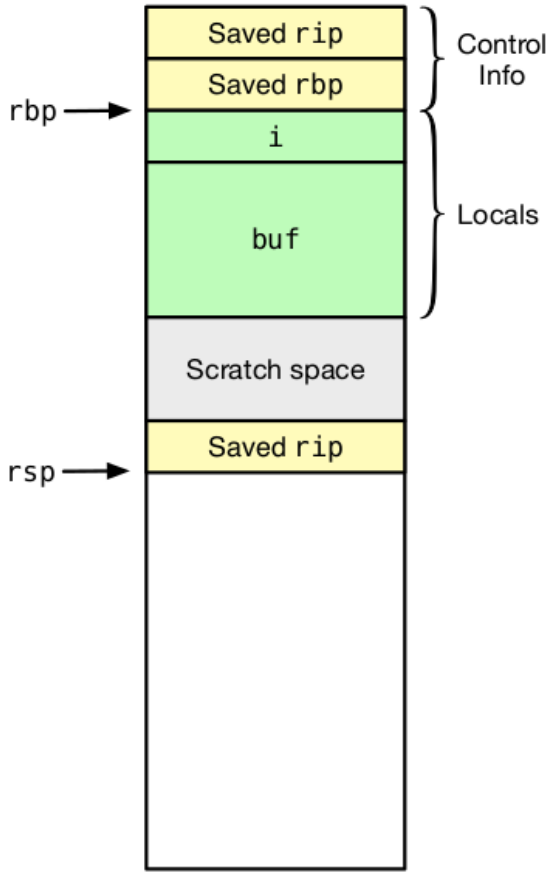

```
1  int auth(const char* user) {
2      size_t i;
3      char buf[16];
4      strncpy(buf, user, sizeof(buf));
5      buf[sizeof(buf) - 1] = '\0';
6      for (i = 0; i < sizeof(buf); i++)
7          buf[i] ^= 0xe5;
8      return !memcmp(buf, "secret", 6);
9  }
```

Stack Frame



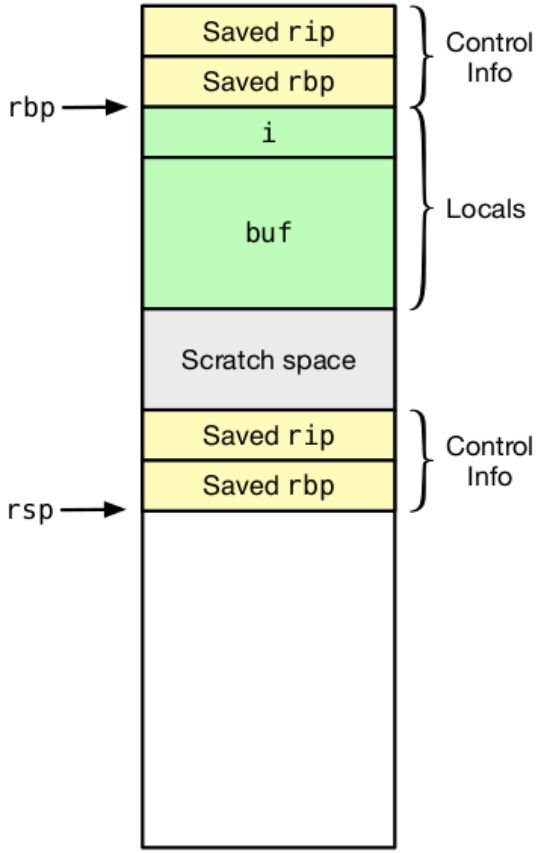

```
1   auth:
2       ...
3       mov rdi, rax
4       call strncpy
5       mov byte [rbp-0x11], 0x00
6       ...
7
8   strncpy:
9       push rbp
10      mov rbp, rsp
11      sub rsp, 0x30
12      ...
13      add rsp, 0x30
14      pop rbp
15      ret
```

Stack Frame



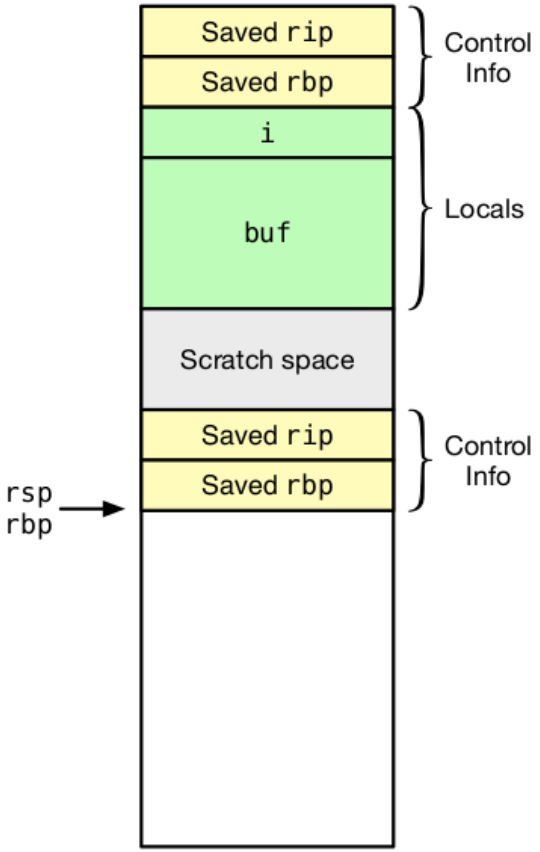

```
1  auth:
2      ...
3      mov rdi, rax
4      call strncpy
5      mov byte [rbp-0x11], 0x00
6      ...
7
8  strncpy:
9      push rbp
10     mov rbp, rsp
11     sub rsp, 0x30
12     ...
13     add rsp, 0x30
14     pop rbp
15     ret
```

Stack Frame



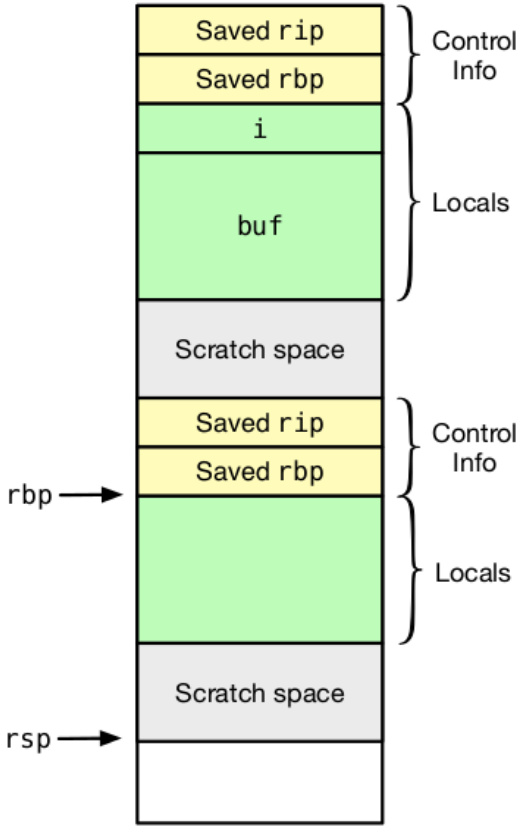

```
1   auth:
2       ...
3       mov rdi, rax
4       call strncpy
5       mov byte [rbp-0x11], 0x00
6       ...
7
8   strncpy:
9       push rbp
10      mov rbp, rsp
11      sub rsp, 0x30
12      ...
13      add rsp, 0x30
14      pop rbp
15      ret
```

Stack Frame



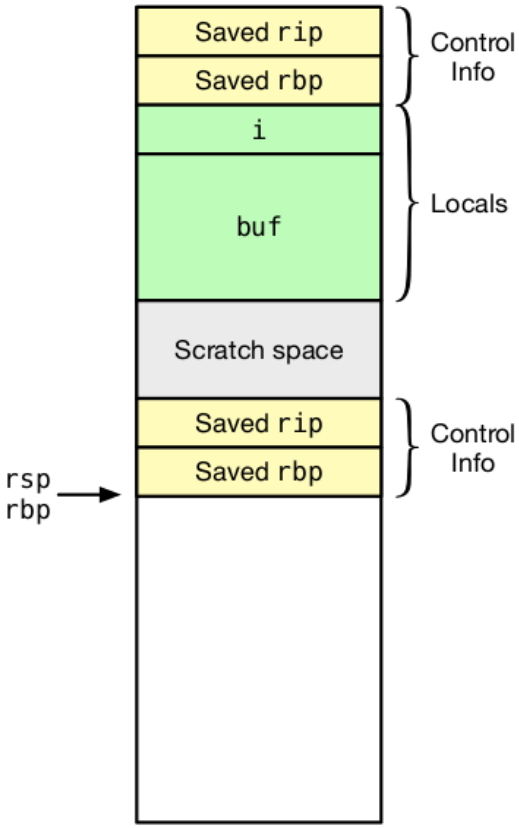

```
1   auth:
2       ...
3       mov rdi, rax
4       call strncpy
5       mov byte [rbp-0x11], 0x00
6       ...
7
8   strncpy:
9       push rbp
10      mov rbp, rsp
11      sub rsp, 0x30
12      ...
13      add rsp, 0x30
14      pop rbp
15      ret
```

Stack Frame



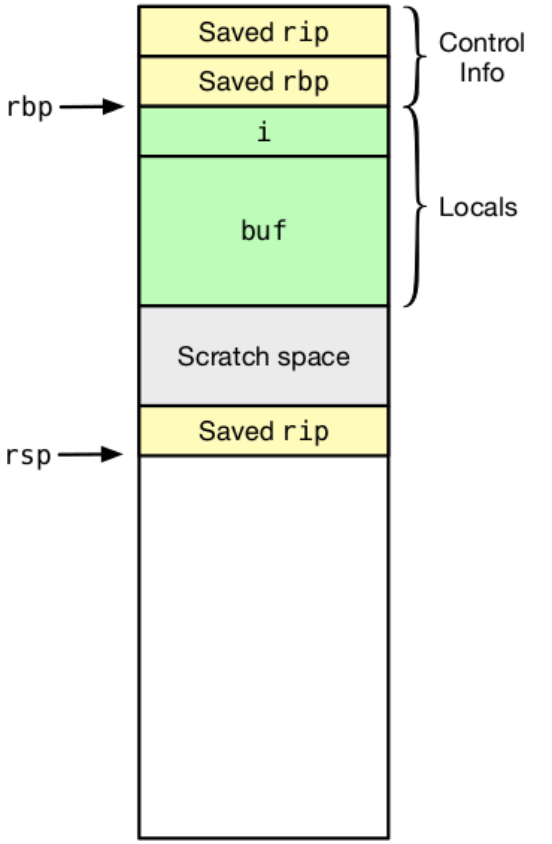

```
1   auth:
2       ...
3       mov rdi, rax
4       call strncpy
5       mov byte [rbp-0x11], 0x00
6       ...
7
8   strncpy:
9       push rbp
10      mov rbp, rsp
11      sub rsp, 0x30
12      ...
13      add rsp, 0x30
14      pop rbp
15      ret
```

Stack Frame



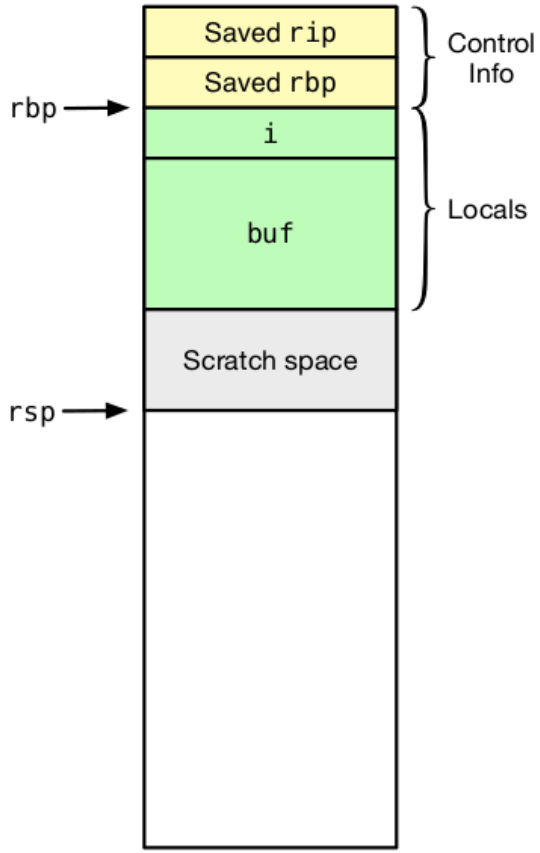

```
1   auth:
2       ...
3       mov rdi, rax
4       call strncpy
5       mov byte [rbp-0x11], 0x00
6       ...
7
8   strncpy:
9       push rbp
10      mov rbp, rsp
11      sub rsp, 0x30
12      ...
13      add rsp, 0x30
14      pop rbp
15      ret
```

Stack Frame




```
1  auth:
2      ...
3      mov rdi, rax
4      call strncpy
5      mov byte [rbp-0x11], 0x00
6      ...
7
8  strncpy:
9      push rbp
10     mov rbp, rsp
11     sub rsp, 0x30
12     ...
13     add rsp, 0x30
14     pop rbp
15     ret
```

Stack Frame




```
1   auth:
2       ...
3       mov rdi, rax
4       call strncpy
5       mov byte [rbp-0x11], 0x00
6       ...
7
8   strncpy:
9       push rbp
10      mov rbp, rsp
11      sub rsp, 0x30
12      ...
13      add rsp, 0x30
14      pop rbp
15      ret
```

Procedure Arguments

Standards (*calling conventions*) exist for argument passing

- Specify where arguments are passed (registers, stack)
- Specify the caller and callee's responsibilities
  - Who deallocates argument space on the stack?
  - Which registers can be clobbered, and who must save them?

Why do we need standards?

- There are many ways to pass arguments
- How would code compiled by different developers and toolchains interoperate?

Calling Conventions

We often speak of *callers* and *callees*

- Caller: Code that invokes a procedure
- Callee: Procedure invoked by another function

Conventions must specify how registers must be dealt with

- Could always save them, but that is inefficient (why?)
- Usually, some registers can be overwritten (clobbered), others cannot
- Registers that can be clobbered: *caller* saved
- Registers that must not be clobbered: *callee* saved

cdecl

We've been concentrating on x86_64, but cdecl is important to know

- Linux 32 bit calling convention

Arguments

- Passed on the stack
- Pushed *right to left* (reverse order)

Registers

- eax, edx, ecx are *caller* saved
- Remainder are *callee* saved

Return value in eax

Caller deallocates arguments on stack after return

stdcall

```
stdcall_fn:
   …
   pop ebp
   ret 0x10 ; return with an operand
```

- Calling convention used by the Win32 API
- Almost identical to cdecl
- **But**, *callee* deallocates arguments on the stack
  - Can you think of a reason why this is better or worse then cdecl? (Hint: `printf()`)

SysV AMD64 ABI

x86_64 calling convention used on Linux, Solaris, FreeBSD, Mac OS X

- This is what you'll see most often in this course

First six arguments passed in registers

- rdi, rsi, rdx, rcx, r8, r9
  - Except syscalls, rcx → r10
- Additional arguments spill to stack

Return value in rax