How Software Executes

CS-576 Systems Security
Instructor: Georgios Portokalidis
Spring 2018
Overview

Introduction

Anatomy of a program

Basic assembly

Anatomy of function calls (and returns)

Memory Safety
Programming Languages

C, C++  

Java, C#  

Python, Perl, PHP
C, C++

Compiles to machine code

Typed but weakly enforced

Low-level memory access

User manages memory
Java, C#

Java
- Compiles to bytecode, run by the Java virtual machine
  - Initially interpreted, quickly just-in-time translated

C#
- Mix of compile and JIT

Type safe and strongly typed

Automatic memory management

Implicit memory access
Python, Perl, PHP

Dynamically typed (duck type)
- Types are checked for suitability at run time

Strong typed
- Operations are checked for safety

Interpreted
- PHP now also uses JIT

Automatic memory management
Why Do I Care?

What happens:

- When you have the following array: `char buffer[4];`
- And the following statement: `buffer[4] = 'A';`

Whatever you guessed may be correct!
It’s All C In the End

C, C++      Java, C#      Python, Perl, PHP

C, C++
static int total;
static int total_over_threshold = 0;
static int arbitrary_threshold = 16700;

int simple_function(int a, int b, int x)
{
    int y;

    y = a*x*x + b*x;

    if (y > arbitrary_threshold)
        total_over_threshold++;

    total++;

    return total;
}
Compilation Process

Multiple stages

Usual compiler components

- Front-end parses source code into an internal representation (IR)
- Plenty of optimizations applied on the IR
- Machine code generation back-end generates binary code
Appropriate for performing optimizations

Easy to map to different machine architectures

For example, the LLVM compiler models a machine an infinite number of registers

```
define i32 @simple_function(i32 %a, i32 %b, i32 %x) #0 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    %y = alloca i32, align 4
    store i32 %a, i32* %1, align 4
    store i32 %b, i32* %2, align 4
    store i32 %x, i32* %3, align 4
    %4 = load i32* %1, align 4
    %5 = load i32* %3, align 4
    %6 = mul nsw i32 %4, %5
    %7 = load i32* %3, align 4
    %8 = mul nsw i32 %6, %7
    %9 = load i32* %2, align 4
    %10 = load i32* %3, align 4
    %11 = mul nsw i32 %9, %10
    %12 = add nsw i32 %8, %11
    store i32 %12, i32* %y, align 4
    %13 = load i32* %y, align 4
    %14 = load i32* @arbitrary_theshold, align 4
    %15 = icmp sgt i32 %13, %14
    br i1 %15, label %16, label %19
}
```

Assembly Code

Assembly code is not binary
Low(est) level code
The language corresponds to the actual machine instructions that hardware can execute

AT&T syntax: instr src, dest
Intel syntax: instr dst, src

GAS, AT&T syntax

```
simple_function:
  pushq  %rbp
  movq  %rsp, %rbp
  movl  %edi, -20(%rbp)
  movl  %esi, -24(%rbp)
  movl  %edx, -28(%rbp)
  movl  -20(%rbp), %eax
  imull -28(%rbp), %eax
  movl %eax, %edx
  movl -24(%rbp), %eax
  addl %edx, %eax
  imull -28(%rbp), %eax
  movl %eax, -4(%rbp)
  movl arbitrary_theshold(%rip), %eax
  cmpl %eax, -4(%rbp)
  jle .L2
  movl
  total_over_threshold(%rip), %eax
  addl $1, %eax
  movl %eax,
  total_over_threshold(%rip)
.L2:
  movl total(%rip), %eax
  addl $1, %eax
```

Spring 2018
Stevens Institute of Technology
Programming Models

High-level languages
Variables
Routines
Objects
APIs
Libraries
...

Assembly
Registers
Flat memory model
Routines
Stack
...

Spring 2018
Stevens Institute of Technology
How the World Really Is

PC: Program counter
- Address of next instruction

Register file
- Heavily used program data

Condition codes
- Store status information about most recent arithmetic or logical operation
- Used for conditional branching

Memory
- Byte addressable array
- Code and user data
- Stack to support procedures

CPU
- Registers
- Condition Codes

Addresses
- Data
- Instructions

Memory
- Code
- Data
- Stack
Overview

Introduction

Anatomy of a program

Basic assembly

Anatomy of function calls (and returns)

Memory Safety
Anatomy of a Program in Memory

User/Kernel Space on 64-bit

Kernel space

User space

ffffffffff80000000
00007fffffffffff
0000000000000000
Process Anatomy

- Kernel Space (1GB)
- User Mode Space (Firefox)
<table>
<thead>
<tr>
<th><strong>Kernel space</strong></th>
<th>User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0xc0000000 == TASK_SIZE</td>
</tr>
<tr>
<td></td>
<td>Random stack offset</td>
</tr>
<tr>
<td>Stack (grows down)</td>
<td>RLIMIT_STACK (e.g., 8MB)</td>
</tr>
<tr>
<td></td>
<td>Random mmap offset</td>
</tr>
<tr>
<td>Memory Mapping Segment</td>
<td>File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so</td>
</tr>
<tr>
<td></td>
<td>program break</td>
</tr>
<tr>
<td></td>
<td>brk</td>
</tr>
<tr>
<td>Heap</td>
<td>start_brk</td>
</tr>
<tr>
<td></td>
<td>Random brk offset</td>
</tr>
<tr>
<td>BSS segment</td>
<td>Uninitialized static variables, filled with zeros. Example: static char *userName;</td>
</tr>
<tr>
<td>Data segment</td>
<td>Static variables initialized by the programmer. Example: static char *gonzo = “God’s own prototype”;</td>
</tr>
<tr>
<td>Text segment (ELF)</td>
<td>Stores the binary image of the process (e.g., /bin/gonzo)</td>
</tr>
<tr>
<td></td>
<td>0x08048000</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
### Kernel space
User code CANNOT read from or write to these addresses, doing so results in a Segmentation Fault

<table>
<thead>
<tr>
<th>0xc0000000 == TASK_SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random stack offset</td>
</tr>
<tr>
<td>RLIMIT_STACK (e.g., 8MB)</td>
</tr>
<tr>
<td>Random mmap offset</td>
</tr>
</tbody>
</table>

### Stack (grows down)

### Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

### Heap

### BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;

### Data segment
Static variables initialized by the programmer. Example: static char *gonzo = “God’s own prototype”;

### Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)

0x08048000

---

### Kernel Space (1GB)

### User Mode Space (Firefox)

---

**Read-only**
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

Stack (grows down)

Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;

Data segment
Static variables initialized by the programmer. Example: static char *gonzo = “God’s own prototype”;

Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)

Kernel Space (1GB)

User Mode Space (Firefox)

Read-write
<table>
<thead>
<tr>
<th>Kernel space</th>
</tr>
</thead>
<tbody>
<tr>
<td>User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault</td>
</tr>
</tbody>
</table>

```
0xc0000000 == TASK_SIZE
```

**Stack (grows down)**

**Memory Mapping Segment**

File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

```
program break brk
start_brk
```

**Heap**

**BSS segment**

Uninitialized static variables, filled with zeros. Example: static char *userName;

**Data segment**

Static variables initialized by the programmer. Example: static char *gonzo = “God’s own prototype”;

**Text segment (ELF)**

Stores the binary image of the process (e.g., /bin/gonzo)
Memory vs Disk

- **BSS**: Read/Write (anonymous)
  - (memory filled with zeros)
  - 0x08048f0
  - 10
  - "God's own prototype"
  - jmp here, jmp there
  - mov this, mov that

- **Data**: Read/Write
  - userName
  - cntActiveUsers
  - gonzo
  - cntWorkerBees

- **Text**: Read/Execute
  - Binary image on disk (/bin/gonzo)

Symbol name or address in memory

Stevens Institute of Technology
Stack

Contains arguments, local variables, function return addresses

Read-write
Heap

Contains dynamically allocated data

Allocated by the OS using brk() and sbrk()

Programmers access it usually through malloc(), realloc(), calloc(), free()

Read-write
Large Heap Objects

Allocators may decide to create an anonymous, private mapping to store large objects instead of directly storing into the heap.
Overview

Introduction

Anatomy of a program

Basic assembly

Anatomy of function calls (and returns)

Memory Safety
Intel x86 Processors

Dominate laptop/desktop/server market

Evolutionary design
- Backwards compatible up until 8086, introduced in 1978
- Added more features as time goes on

Complex instruction set computer (CISC)
- Many different instructions with many different formats
  - But, only small subset encountered with Linux programs
- Hard to match performance of Reduced Instruction Set Computers (RISC)
- But, Intel has done just that!
  - In terms of speed. Less so for low power.
Intel x86 Processors

Machine Evolution

- 386 1985 0.3M
- Pentium 1993 3.1M
- Pentium/MMX 1997 4.5M
- PentiumPro 1995 6.5M
- Pentium III 1999 8.2M
- Pentium 4 2001 42M
- Core 2 Duo 2006 291M
- Core i7 2008 731M

Added Features

- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 bits to 64 bits
- More cores
## x86 Integer Registers

**General purpose registers**
- On 32-bit architectures
  - EAX, EBX, ECX, EDX, EDI, ESI, ESP, EBP

**The instruction pointer (IP)**
- Also referred to as program counter (PC)
- EIP on 32-bit

**FLAGS register**
- Used for control flow operations, etc.
- EFLAGS

### Table: x86 Integer Registers

<table>
<thead>
<tr>
<th>Register Encoding</th>
<th>High 8-bit</th>
<th>Low 8-bit</th>
<th>16-bit</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AH (4)</td>
<td>AL</td>
<td>AX</td>
<td>EAX</td>
</tr>
<tr>
<td>3</td>
<td>BH (7)</td>
<td>BL</td>
<td>BX</td>
<td>EBX</td>
</tr>
<tr>
<td>1</td>
<td>CH (5)</td>
<td>CL</td>
<td>CX</td>
<td>ECX</td>
</tr>
<tr>
<td>2</td>
<td>DH (6)</td>
<td>DL</td>
<td>DX</td>
<td>EDX</td>
</tr>
<tr>
<td>6</td>
<td>SI</td>
<td>SI</td>
<td>SI</td>
<td>ESI</td>
</tr>
<tr>
<td>7</td>
<td>DI</td>
<td>DI</td>
<td>DI</td>
<td>EDI</td>
</tr>
<tr>
<td>5</td>
<td>BP</td>
<td>BP</td>
<td>BP</td>
<td>EBP</td>
</tr>
<tr>
<td>4</td>
<td>SP</td>
<td>SP</td>
<td>SP</td>
<td>ESP</td>
</tr>
</tbody>
</table>

### Diagram: x86 Integer Registers
# x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>
x86-64 Integer Registers

Can reference low-order bytes too
- d suffix for lower 32-bits (%r8d)
- w suffix for lower 16-bits (%r8w)
- b suffix for lower 8-bits (%r8b)
## Typical Register Uses

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>accumulator</td>
</tr>
<tr>
<td>EBX</td>
<td>Pointer to data</td>
</tr>
<tr>
<td>ECX</td>
<td>Counter for string operations and loops</td>
</tr>
<tr>
<td>EDX</td>
<td>I/O Operations</td>
</tr>
<tr>
<td>EDI</td>
<td>Destination for string operations</td>
</tr>
<tr>
<td>ESP</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>EBP</td>
<td>Frame pointer</td>
</tr>
</tbody>
</table>

### Register Encoding Table

<table>
<thead>
<tr>
<th>Register Encoding</th>
<th>8-bit High</th>
<th>8-bit Low</th>
<th>16-bit</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AH (4)</td>
<td>AL</td>
<td>AX</td>
<td>EAX</td>
</tr>
<tr>
<td>3</td>
<td>BH (7)</td>
<td>BL</td>
<td>BX</td>
<td>EBX</td>
</tr>
<tr>
<td>1</td>
<td>CH (5)</td>
<td>CL</td>
<td>CX</td>
<td>ECX</td>
</tr>
<tr>
<td>2</td>
<td>DH (6)</td>
<td>DL</td>
<td>DX</td>
<td>EDX</td>
</tr>
<tr>
<td>6</td>
<td>SI</td>
<td>SI</td>
<td>SI</td>
<td>ESI</td>
</tr>
<tr>
<td>7</td>
<td>DI</td>
<td>DI</td>
<td>DI</td>
<td>EDI</td>
</tr>
<tr>
<td>5</td>
<td>BP</td>
<td>BP</td>
<td>BP</td>
<td>EBP</td>
</tr>
<tr>
<td>4</td>
<td>SP</td>
<td>SP</td>
<td>SP</td>
<td>ESP</td>
</tr>
<tr>
<td>31-0</td>
<td>Flags</td>
<td>IP</td>
<td>IP</td>
<td>EIP</td>
</tr>
</tbody>
</table>
Assembly Syntax

Intel: OP dest, src

AT&T: OP src, dest

Unix systems prefer AT&T

- We are going to use the same as the GNU assembler (gas syntax)
Assembly Instructions

**pushq**: push quad word to stack

**movq**: Move quad word

**imull**: Signed multiply long

**addl**: Add long

```
pushq  %rbp
movq   %rsp, %rbp
movl   %edi, -20(%rbp)
movl   %esi, -24(%rbp)
movl   %edx, -28(%rbp)
movl   -20(%rbp), %eax
imull  -28(%rbp), %eax
movl   %eax, %edx
movl   -24(%rbp), %eax
addl   %edx, %eax
imull  -28(%rbp), %eax
```
Operand Sizes

Instructions include a suffix that indicates the size of the operand(s)

Register operands are prefixed with a %

Register operands must match size
For example,
- quad → rax
- long → eax
- word → ax
- byte → ah or al

Intel syntax does not include a suffix, size depends on the size of the operand
Memory Operands

Parentheses indicate a memory operand

Each memory address can be defined as:

\( \text{Base} + \text{Index} \times \text{Scale} + \text{Disp} \)

- In AT&T syntax:
  
  \( \text{disp(base, index, scale)} \)

- disp, index, and scale are optional

```
pushq  %rbp
movq   %rsp, %rbp
movl   %edi, -20(%rbp)
movl   %esi, -24(%rbp)
movl   %edx, -28(%rbp)
movl   -20(%rbp), %eax
imull  -28(%rbp), %eax
movl   %eax, %edx
movl   -24(%rbp), %eax
addl   %edx, %eax
imull  -28(%rbp), %eax
```
Memory Addressing Modes

Normal (B) Mem[Reg[R]]
- Register R specifies memory base address
- Pointer dereferencing in C

\[
\text{movq } (\%rcx), \%rax
\]

Displacement D(B) Mem[Reg[R]+D]
- Register R specifies start of memory region
- Constant displacement D specifies offset

\[
\text{movq } 8(\%rbp), \%rdx
\]
Memory Addressing Modes

Most General Form

\[ D(B, I, S) \quad \text{Mem}[\text{Reg}[R_b] + S \times \text{Reg}[R_i] + D] \]

- **D**: Constant “displacement” 1, 2, or 4 bytes
- **Rb**: Base register
- **Ri**: Index register: Any, except for `%rsp`
- **S**: Scale: 1, 2, 4, or 8

```assembly
movq 8(%rbp, %rax, 4), %rdx
```
Immediates

Constants or immediates are defined using $

In decimal, unless:

- 0x prefix is used → hexadecimal
- 0 prefix is used → octal

addl $1, %eax

Immediates can help you identify the syntax
Endianness

Memory representation of multi-byte integers
For example the integer: 0x0A0B0C0Dh

Big-endian $\leftrightarrow$ highest order byte first
  0A 0B 0C 0D

Little-endian $\leftrightarrow$ lowest order byte first (X86)
  0D 0C 0B 0A
The diagram illustrates the difference between big-endian and little-endian memory access.

In big-endian architecture, the memory is read as follows:
- $a$: OA
- $a+1$: OB
- $a+2$: OC
- $a+3$: OD

In little-endian architecture, the memory is read as follows:
- $a$: OD
- $a+1$: OC
- $a+2$: OB
- $a+3$: OA
Load Effective Address

`leaq Src, Dst`
- Src is address mode expression
- Set Dst to address denoted by expression

Computing addresses without a memory reference
- E.g., translation of `p = &x[i];`

Computing arithmetic expressions of the form `x + k*y`
- `k = 1, 2, 4, or 8`

Example

`leaq (%rdi,%rdi,2), %rax`
Control Flow

if (a > b) {
    c = d;
} else {
    d = c;
}
EFLAGS Register

- **ID Flag (ID)**
- **Virtual Interrupt Pending (VIP)**
- **Virtual Interrupt Flag (VIF)**
- **Alignment Check (AC)**
- **Virtual-8086 Mode (VM)**
- **Resume Flag (RF)**
- **Nested Task (NT)**
- **I/O Privilege Level (IOPL)**
- **Overflow Flag (OF)**
- **Direction Flag (DF)**
- **Interrupt Enable Flag (IF)**
- **Trap Flag (TF)**
- **Sign Flag (SF)**
- **Zero Flag (ZF)**
- **Auxiliary Carry Flag (AF)**
- **Parity Flag (PF)**
- **Carry Flag (CF)**

- **S** Indicates a Status Flag
- **C** Indicates a Control Flag
- **X** Indicates a System Flag

Reserved bit positions. DO NOT USE.
Always set to values previously read.
Condition Codes Set by Compare

Explicit Setting by Compare Instruction

- `cmpq Src2, Src1`
- `cmpq b, a` like computing `a - b` without setting destination

- **CF set** if carry out from most significant bit (used for unsigned comparisons)
- **ZF set** if `a == b`
- **SF set** if `(a - b) < 0` (as signed)
- **OF set** if two’s-complement (signed) overflow
  
  
  \[(a > 0 \land b < 0 \land (a - b) < 0) \lor (a < 0 \land b > 0 \land (a - b) > 0)\]
Condition Codes (Explicit Setting: Test)

Explicit Setting by Test instruction

- \texttt{testq \ Src2, Src1}  
  - \texttt{testq \ b, a} like computing \texttt{a\&b} without setting destination

- Sets condition codes based on value of \texttt{Src1 \& Src2}
- Useful to have one of the operands be a mask

- \texttt{ZF set when \ a\&b == 0}
- \texttt{SF set when \ a\&b < 0}
# Common Conditional Jumps

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>\sim\ ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>jg</td>
<td>\sim (SF^OF) &amp; \sim\ ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>\sim (SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
</tbody>
</table>
Overview

Introduction

Anatomy of a program

Basic assembly

Anatomy of function calls (and returns)

Memory Safety
Function Calls

Function calls transfer control and use the stack to keep track of callees

- `call <address>` Transfer control to address and save the address of the next instruction on the stack
- `ret` Pop the address from the stack and transfer control to it

`call` and `ret` implicitly use the RSP register
- So does `push/pop`
Simple Function Call
Simple Function Call

```c
void F1()
{
    ....
    F2();
    ...
}
```
Simple Function Call

```c
void F1()
{
    ....
    F2();
    ...
}
```

```c
void F2()
{
    ...
}
```
Simple Function Call

```c
void F1()
{
    ...
    F2();
    ...
}

void F2()
{
    ...
}
```
Simple Function Call

```c
void F1()
{
    ...
    F2();
    ...
}

void F2()
{
    ...
}
```

```
RSP

RETADDR

RETADDR

STACK

Bottom

High address

Low address

TOP

Spring 2018

Stevens Institute of Technology
Simple Function Call

```c
void F1()
{
    ....
    F2();
    ....
}

void F2()
{
    ...
}
```

Diagram:
- Call from F1 to F2.
- Return from F2 to F1.
- Stack layout with RSP and RETADDR.
Local Variables

```c
void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ...
    F2();
    ...
}
```
Local Variables

```c
void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ...
    F2();
    ...
}
```
Local Variables

void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ....
    F2();
    ...
}

void F2()
{
    int i;
    ...
}
void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ....
    F2();
    ...
}

void F2()
{
    int i;
    ...
}
RBP == Frame Base Pointer

```c
void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ....
    F2();
    ...
}
```
RBP == Frame Base Pointer

```c
void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ...
    F2();
    ...
}
```

**Function entry (ENTER)**
- `pushq %rbp`
- `movq %rsp, %rbp`

**Function exit (LEAVE)**
- `movq %rbp, %rsp`
- `pop %rbp`
Frame pointers are optional

gcc -fomit-frame-pointer test.c
void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ....
    F2(a, buf, l);
    ...
}

void F2(int A, char *BUF unsigned long L)
{
    int i;
    ...
}
Calling Conventions

Defines the standard for passing arguments

Caller and callee need to agree

Enforced by compiler

Important when using 3rd party libraries

Different styles ↔ different advantages
Popular Conventions

cdecl (mostly 32-bit)
Arguments are passed on the stack
- Pushed right to left
  - eax, edx, ecx are caller saved
  - callee can overwrite without saving

ebx, esi, edi are callee saved
- callee must ensure they have same value on return

eax used for function return value

System V AMD64 ABI
Arguments are passed using registers
- First 6 integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, and R9

RBP, RBX, and R12–R15 are callee saved

RAX used for function return

https://en.wikipedia.org/wiki/X86_calling_conventions
cdecl Example

void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ...
    F2(a, buf, l);
    ...
}

void F2(int A, char *BUF unsigned long L)
{
    int i;
    ...
}
cdecl Example

```c
void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ...
    F2(a, buf, l);
    ...
}

void F2(int A, char *BUF unsigned long L)
{
    int i;
    ...
}
```

Spring 2018
Stevens Institute of Technology
void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ...
    F2(a, buf, l);
    ...
}

void F2(int A, char *BUF unsigned long L)
{
    int i;
    ...
}
void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ....
    F2(a, buf, l);
    ...
}

void F2(int A, char *BUF unsigned long L)
{
    int i;
    ...
}
Variable Number of Arguments

Used in variadic functions, like

```c
int printf(const char *format, ...);
```

Arguments passed in the stack
- Order right-to-left

Only caller knows exact number of arguments
- Caller responsible for cleaning
void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ...
    printf("%d %s\n", a, buf);
    ...
}

void printf(const char *fmt, ...)
{
    ...
}
void F1() {
    int a;
    char buf[16];
    unsigned long l;
    ....
    printf("%d %s\n", a, buf);
    ...
}

void printf(const char *fmt, ...) {
    ...
}
Alignment

CPUs like aligned data
- Better performance

Compilers try to align data

```
0x00  0x07  0x10
int   long
```
Accessing Stack Variables

With frame pointer

```c
mov -0x18(%rbp),%eax
```

With stack pointer

```c
mov 0xc(%rsp),%eax
```
Overview

Introduction

Anatomy of a program

Basic assembly

Anatomy of function calls (and returns)

Memory Safety
Flat Memory Model
Paging and Permissions

Memory is organized in pages (usually 4KB, but can also be 2MB)
Page permissions: readable, writable, executable

- RW:
- R--:
- RX:
- RWX:

The hardware enforces these permissions

Processes can change (their) page permissions
int n = 0xdeadbeef;
char buffer[16] = "Hello";
int n = 0xdeadbeef;
char buffer[16] = “Hello”;
Memory Safety

int n = 0xdeadbeef;
char buffer[16] = “Hello”;

Little endian byte order

| 0xef | 0xbe | 0xed | 0xde |
| 'H'  | 'e'  | 'l'  | 'l'  |
| 'o'  | 0x00 | ~~~  | ~~~  |

10 uninitialized bytes

Data are not isolated by hardware, it is up to the software

Memory safe languages introduce checks to ensure memory is not corrupted (Python, Java, etc.)

Other languages depend on the programmer to check when accessing memory (C, C+)

Spring 2018
Stevens Institute of Technology
int simple_function(char *str)
{
    char *c;

    for (c = str; c != '\0'; c++) {
        if (*c == '0')
            *c = '1';
    }
}

http://www.felixcloutier.com/x86/
https://en.wikibooks.org/wiki/X86_Assembly

https://en.wikipedia.org/wiki/X86_calling_conventions