How Software Executes

CS-576 Systems Security
Instructor: Georgios Portokalidis
Fall 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
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_theshold = 16700;

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

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

    if (y > arbitrary_theshold)
        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
Compiler IR

Appropriate for performing optimizations

Easy to map to different machine architectures

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

High-level languages

Variables
Routines
Objects
APIs
Libraries
...

Assembly

Registers
Flat memory model
Routines
Stack
...

Fall 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

Memory
- Code
- Data
- Stack

Addresses
Data
Instructions
Overview

Introduction

Anatomy of a program

Basic assembly

Anatomy of function calls (and returns)

Memory Safety
Anatomy of a Program in Memory

32-bit OS

User/Kernel Space on 64-bit

Kernel space

User space
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

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)

0xc0000000 == TASK_SIZE
Random stack offset
RLIMIT_STACK (e.g., 8MB)
Random mmap offset

program break
brk
start_brk
Random brk offset

data
end_data
start_data
end_code
0x08048000
0

Kernel Space (1GB)

User Mode Space (Firefox)
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

0xc0000000 == TASK_SIZE
Random stack offset

Stack (grows down)

RLIMIT_STACK (e.g., 8MB)
Random mmap offset

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

program break
brk
start_brk
Random brk offset

Heap

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

end_data
start_data

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

end_code

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

0x08048000
0
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

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)

0xc0000000 == TASK_SIZE
Random stack offset
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
program break brk
start_brk
Random brk offset
data
end_data
start_data
end_code
0x08048000 0
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

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)

0xc0000000 == TASK_SIZE
Random stack offset
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
program break brk
start_brk Random brk offset
end_data
start_data
end_code
0x08048000 0
Initialized vs Uninitialized

- **BSS (anonymous)**: Read/Write
- **Data**: Read/Write
- **Text**: Read/Execute

### Memory
- **Symbol name or address in memory**
  - `userName`
  - `cntActiveUsers`
  - `gonzo`
  - `cntWorkerBees`
  - `jmp here, jmp there, mov this, mov that`

- **Binary image on disk**: `/bin/gonzo`

### Example
- `0x080484f0`
- `10`
- `"God's own prototype"`
- `0x080484f0`
- `0x08048000`
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
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
## x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<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

EAX: accumulator
EBX: Pointer to data
ECX: Counter for string operations and loops
EDX: I/O Operations
EDI: Destination for string operations
ESP: Stack pointer
EBP: Frame pointer
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

**Pushq**

Register operands are prefixed with a `%`

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

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

Intel syntax does not include a suffix, size depends on the size of the operand
Parentheses indicate a memory operand

Each memory address can be defined as:
Base + Index * Scale + Disp

- In AT&T syntax: 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) \quad \text{Mem}[\text{Reg}[R]]
- Register R specifies memory base address
- Pointer dereferencing in C

\textbf{movq} (\%rcx),\%rax

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

\textbf{movq} 8(\%rbp),\%rdx
Memory Addressing Modes

Most General Form

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

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

\text{movq} 8(%rbp, %rax, 4),%rdx