X86 Review
Process Layout, ISA, etc.

CS642:
Computer Security

Drew Davidson
davidson@cs.wisc.edu
From Last Week

• ACL-based permissions (UNIX style)
  – Read, Write, eXecute can be restricted on users and groups
  – Processes (usually) run with the permissions of the invoking user

• Example:
Processes are the front line of system security

• Control a process and you get the privileges of its UID
• So how do you control a process?
  – Send specially formed input to process

input -> passwd

/etc/shadow

RUID: rist

EUID: root
Roadmap

• Today
  – Enough x86 to understand (some) process vulnerabilities

• Next Time
  – Live demo of an attack
  – How such attacks occur
Why do we need to look at assembly?

“WYSINWYX: What you see is not what you eXecute”
[Balakrishnan and Reps TOPLAS 2010]
x86: Popular but crazy

- CISC (complex instruction set computing)
  - Over 100 distinct opcodes in the set
- Register poor
  - Only 8 registers of 32-bits, only 6 are general-purpose
- Variable-length instructions
- Built of many backwards-compatible revisions
  - Many security problems preventable... in hindsight
A Little History

- **1978**: Intel introduces 8086 (16 bit)
- **1982**: 80186 and 80286
- **1985**: 80386 (32-bit)
- **1989**: i486 (32-bit)
- **1993**: Pentium
- **1995**: Pentium Pro
- **2003**: AMD makes x86-64 (64 bit)

Intel attempts to trademark the number 486, gets denied

“5” Science-y?
Process memory layout

- **.text**
  - Machine code of executable

- **.data**
  - Global initialized variables

- **.bss**
  - Below Stack Section

- **heap**
  - Dynamic variables

- **stack**
  - Local variables
  - Function call data

- **Env**
  - Environment variables
  - Program arguments
# Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>EAX</th>
<th>EBX</th>
<th>ECX</th>
<th>EDX</th>
<th>ESI</th>
<th>EDI</th>
<th>ESP</th>
<th>EBP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AX</td>
<td>BX</td>
<td>CX</td>
<td>DX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AH</td>
<td>BH</td>
<td>CH</td>
<td>DH</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AL</td>
<td>BL</td>
<td>CL</td>
<td>DL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **32 bits**
- **(stack pointer)**
- **(base pointer)**
Instruction Syntax

Examples:

```
subl $16, %ebx

movl (%eax), %ebx
```

- Instruction ends with data length
- opcode, src, dst
- Constants preceded by $
- Registers preceded by %
- Indirection uses ( )
Register Instructions: sub

- Subtract from a register value

```
subl %eax, %ebx
```
The Stack

• Local storage
  – Good place to keep data that doesn’t fit into registers

• Grows from high addresses towards low addresses
Frame Instructions: push

- Put a value on the stack
  - Pull from register
  - Value goes to %esp
  - Subtract from %esp

Example:

```
pushl %eax
```
Frame Instructions: pop

- Take a value from the stack
  - Pull from stack pointer
  - Value goes from %esp
  - Add to %esp
Control flow instructions: jmp

- %eip points to the currently executing instruction (in the text section)
- Has unconditional and conditional forms
- Uses relative addressing

\[
jmp -20
\]

\[
\begin{align*}
\%eip & \quad K \quad \%ebp \quad M \quad \%esp \quad N \\
\text{Frame} & \\
\%eip & \quad \text{K-20} \quad \%ebp \quad M \quad \%esp \quad N \\
\text{Frame} & \\
\end{align*}
\]
Control flow instructions: call

- Saves the current instruction pointer to the stack
- Jumps to the argument value

Registers:
- %eip\text{K}
- %ebp\text{M}
- %esp\text{N}

Memory:
- A: call FOO
- Frame
- Frame
- FOO: (1\text{st} of foo)
- A+2
Control flow instructions: ret

- Pops the stack into the instruction pointer
Stack instructions: leave

• Equivalent to
  movl %ebp, %esp
  popl %ebp
Implementing a function call

(main) (foo)

Stack data

esp

ebp

main:

...  
subl $8, %esp
movl $2, 4(%esp)
movl $1, (%esp)
call foo
addl $8, %esp
...

foo:

pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl $3, -4(%ebp)
movl 8(%ebp), %eax
addl $9, %eax
leave
ret
Implementing a function call

(main) (foo)

Stack data

main:

... subl $8, %esp
movl $2, 4(%esp)
movl $1, (%esp)
call foo
addl $8, %esp

... foo:

pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl $3, -4(%ebp)
movl 8(%ebp), %eax
addl $9, %eax
leave
ret
Implementing a function call

```
(main) (foo)
```

```
main:
  ...
  subl $8, %esp
  movl $2, 4(%esp)
  movl $1, (%esp)
  call foo
  addl $8, %esp
  ...

foo:
  pushl %ebp
  movl %esp, %ebp
  subl $16, %esp
  movl $3, -4(%ebp)
  movl 8(%ebp), %eax
  addl $9, %eax
  leave
  ret
```
Implementing a function call

```
main:
    ...
    subl $8, %esp
    movl $2, 4(%esp)
    movl $1, (%esp)
    call foo
    addl $8, %esp
    ...

foo:
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
    movl $3, -4(%ebp)
    movl 8(%ebp), %eax
    addl $9, %eax
    leave
    ret
```
Implementing a function call

```
main:
  ...
  subl $8, %esp
  movl $2, 4(%esp)
  movl $1, (%esp)
  call foo
  addl $8, %esp
  ...

foo:
  pushl %ebp
  movl %esp, %ebp
  subl $16, %esp
  movl $3, -4(%ebp)
  movl $8(%ebp), %eax
  addl $9, %eax
  leave
  ret
```
Implementing a function call

main:
...
subl $8, %esp
movl $2, 4(%esp)
movl $1, (%esp)
call foo
addl $8, %esp
...

foo:
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl $3, -4(%ebp)
movl 8(%ebp), %eax
addl $9, %eax
leave
ret
Implementing a function call

```
main:
  ...
  subl $8, %esp
  movl $2, 4(%esp)
  movl $1, (%esp)
  call foo
  addl $8, %esp
  ...

foo:
  pushl %ebp
  movl %esp, %ebp
  subl $16, %esp
  movl $3, -4(%ebp)
  movl 8(%ebp), %eax
  addl $9, %eax
  leave
  ret
```
Implementing a function call

main:
...  
subl $8, %esp  
movl $2, 4(%esp)  
movl $1, (%esp)  
call foo  
addl $8, %esp
...  

foo:
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl $3, -4(%ebp)  
movl 8(%ebp), %eax  
addl $9, %eax  
leave  
ret
Implementing a function call

```
main:
  ...
  subl $8, %esp
  movl $2, 4(%esp)
  movl $1, (%esp)
  call foo
  addl $8, %esp
  ...

foo:
  pushl %ebp
  movl %esp, %ebp
  subl $16, %esp
  movl $3, -4(%ebp)
  movl 8(%ebp), %eax
  addl $9, %eax
  leave
  ret
```
Implementing a function call

main:
  ...
  subl $8, %esp
  movl $2, 4(%esp)
  movl $1, (%esp)
  call foo
  addl $8, %esp
  ...

foo:
  pushl %ebp
  movl %esp, %ebp
  subl $16, %esp
  movl $3, -4(%ebp)
  movl 8(%ebp), %eax
  addl $9, %eax
  leave
  ret
Implementing a function call

```
main:
    ...
    subl $8, %esp
    movl $2, 4(%esp)
    movl $1, (%esp)
    call foo
    addl $8, %esp
    ...

foo:
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
    movl $3, -4(%ebp)
    movl 8(%ebp), %eax
    addl $9, %eax
    leave
    ret
```
Implementing a function call

```
main:
  ...
  subl $8, %esp
  movl $2, 4(%esp)
  movl $1, (%esp)
  call foo
  addl $8, %esp
  ...

foo:
  pushl %ebp
  movl %esp, %ebp
  subl $16, %esp
  movl $3, -4(%ebp)
  movl 8(%ebp), %eax
  addl $9, %eax
  leave
  ret
```
Implementing a function call

main:

... 
subl $8, %esp
movl $2, 4(%esp)
movl $1, (%esp)
call foo
addl $8, %esp
... 

foo:

pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl $3, -4(%ebp)
movl 8(%ebp), %eax
addl $9, %eax
leave
ret
Implementing a function call

```assembly
main:
  ...
  subl $8, %esp
  movl $2, 4(%esp)
  movl $1, (%esp)
  call foo
  addl $8, %esp
  ...

foo:
  pushl %ebp
  movl %esp, %ebp
  subl $16, %esp
  movl $3, -4(%ebp)
  movl 8(%ebp), %eax
  addl $9, %eax
  leave
  ret
```
Implementing a function call

main:

...  
subl $8, %esp
movl $2, 4(%esp)
movl $1, (%esp)
call foo
addl $8, %esp
...

foo:

pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl $3, -4(%ebp)
movl $8(%ebp), %eax
addl $9, %eax
leave
ret
Implementing a function call

main:

... subl $8, %esp
movl $2, 4(%esp)
movl $1, (%esp)
call foo
addl $8, %esp

foo:

pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl $3, -4(%ebp)
movl 8(%ebp), %eax
addl $8, %esp
leave
ret
Function Calls: High level points

• Locals are organized into stack frames
  – Callees exist at lower address than the caller

• On call:
  – Save %eip so you can restore control
  – Save %ebp so you can restore data

• Implementation details are largely by convention
  – Somewhat codified by hardware
Data types / Endianness

• x86 is a little-endian architecture

pushl %eax

%eax: 0xdeadbeef

4 bytes

esp: 0xdede 0xad 0xbe 0xef
Arrays

void bar(char * in){
    char name[5];
    strcpy(name, in);
}

bar:
    pushl %ebp
    movl %esp, %ebp
    subl $5, %esp
    movl 8(%ebp), %eax
    movl %eax, 4(%esp)
    leal -5(%ebp), %eax
    movl %eax, (%esp)
    call strcpy
    leave
    ret
What if you have a long name?

```c
void bar(char * in){
    char name[5];
    strcpy(name, in);
}
```
Buffer Overflow!

- You know it!
- I know it
- C doesn’t know it
  - In this case, the return is likely to cause a catastrophic failure upon return (seg fault)
  - If you are “lucky” the overflow might only overwrite unused locals and never manifest a bug
- Java *does* know it’s a buffer overflow
  - ArrayIndexOutOfBoundsException
Next Time

Exploiting buffer overflows
Tools: GCC

gcc -O0 -S program.c -o program.S -m32

gcc -O0 -g program.c -o program -m32
Tools: GDB

gdb program
(gdb) run
(gdb) disas foo
(gdb) quit
Tools: objdump

objdump -Dwrt program
Tools: od

od  -x  program
Summary

• Basics of x86
  – Process layout
  – ISA details
  – Most of the instructions that you’ll need

• Introduced the concept of a buffer overflow

• Some tools to play around with x86 assembly