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
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?

“We SAW NWWX: What you see is not what you execute”
[Balakrishnan and Reps TOPLAS 2010]

We understand code in this form

```c
int foo()
{
    int a = 0;
    return a + 7;
}
```

Vulnerabilities exploited in this form

```asm
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl $0, -4(%ebp)
movl -4(%ebp), %eax
addl $7, %eax
leave
ret
```
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
Intel attempts to trademark the number 486, gets denied
Process memory layout

- **.text**
  - Machine code of executable
- **.data**
  - Global initialized variables
- **.bss**
  - Below Stack Section
  - Global uninitialized variables
- **heap**
  - Dynamic variables
- **stack**
  - Local variables
  - Function call data
- **Env**
  - Environment variables
  - Program arguments

Low memory addresses Grows upward

High memory addresses Grows downward
## Registers

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

- 32 bits
- Register names in lowercase (EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP)
- Alphabetic order of registers
- Memory locations: (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
Control flow instructions: call

- Saves the current instruction pointer to the stack
- Jumps to the argument value
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:
...  
  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
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
```
void bar(char * in) {
    char name[5];
    strcpy(name, in);
}

```assembly
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
```
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) decompile 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