X86 Review
Process Layout, ISA, etc.

CS642:
Computer Security

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

- 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

```
input
```

```
passwd
```

```
RUID: ace
```

```
EUID: root
```

```
/write
```

```
/etc/shadow
```

```
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
  RUID: ace
  EUID: root
write → /etc/shadow
Privilege Escalation

article published last Thursday!

Linux Kernel Bug Allows Local-To-Root Privilege Escalation

Tens of millions of Linux servers, desktops, plus 66 percent of Android devices affected.

Tens of millions of Linux PCs and servers and 66% of all Android devices are impacted by a vulnerability in the Linux kernel that allows privilege escalation from local to root via a use-after-free attack, according to the research team at Perception Point.

Although no exploits for the bug have been seen in the wild yet, the
Lecture Roadmap

• Today
  – Enough x86 to understand (some) process vulnerabilities
    • Memory Layout
    • Some x86 instruction semantics
    • Tools for inspecting assembly

• Next Time
  – 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: The De Facto Standard

- Extremely popular for desktop computers
- Alternatives
  - ARM: popular on mobile
  - MIPS: very simple
  - Itanium: ahead of its time
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 introduces 8086 (16-bit)
1978

1982
80186 and 80286

1985
80386 (32-bit)

1989
i486 (32-bit)

Intel attempts to trademark the number 486, gets denied

1993
Pentium

1995
Pentium Pro

2003
AMD makes x86-64 (64-bit)

...
Let’s Dive in To X86!
## Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>AX</th>
<th>AH</th>
<th>AL</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EBX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDX</td>
<td></td>
<td></td>
<td></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
- (stack pointer)
- (base pointer)
Process memory layout

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

Low memory addresses: Grows upward

Free memory: Grows upward

High memory addresses: Grows downward
• Allow for more efficient use of finite free memory
  – Growing in opposite directions allows extra flexibility at runtime

• Stack
  – Local variables, function bookkeeping

• Heap
  – Dynamic memory
Heap and Stack Design

- Allow for more efficient use of finite free memory
  - Growing in opposite directions allows extra flexibility at runtime
- Stack
  - Local variables, function bookkeeping
- Heap
  - Dynamic memory
Heap and Stack Design

- Allow for more efficient use of finite free memory
  - Growing in opposite directions allows extra flexibility at runtime

- Stack
  - Local variables, function bookkeeping

- Heap
  - Dynamic memory
Heap and Stack use: Example

main():
   call foo()
   call bar()

foo():
   f_glob = malloc(0x100)
   call bar()

bar()
   b_loc = 7;
Reminder: These are conventions

• Dictated by compiler
• Only instruction support by processor
  – Almost no structural notion of memory safety
    • Use of uninitialized memory
    • Use of freed memory
    • Memory leaks

• So how are they actually implemented?
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

```
%eax  7  %ebx  9  2

subl %eax, %ebx
```
Frame Instructions: push

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

Example:

```assembly
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

%eip K %ebp M %esp N

A: call FOO

%eip FOO %ebp M %esp N-4

FOO: (1st of foo) A+2 Frame
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

%eax 0xdeadbeef

4 bytes

esp

0xef 0xbe 0xad 0xde

esp

1 1 1 1
void bar(char * in){
    char name[5];
    strcpy(name, in);
}

Arrays

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

Esp

Hep

.text .data

HEAP

in

ebp

eip+2

D 0x44
r 0x72
w 0x65
\0 0x00

char name[5];
strcpy(name, in);
Assembly Code Tools

• Let’s look at some programs for observing these phenomena
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
Memory Safety: Why and Why Not

• The freedom from these shenanigans

• X86 has little *inbuilt* notion of memory safety
  – Compiler or analysis can
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

• Next time: exploiting these vulnerabilities