System Model: Source Code to Execution

18-732
Spring 2015
Lujo Bauer
From Concept to Execution

Program idea → Program source code → Compiled program
Why Programs Have (Security) Bugs

- Bad idea
- Program idea
- Idea poorly translated to source code
- Compiler bugs; abstractions not preserved
- Code tampered with before execution
- Code or machine state tampered with during execution
- Compiled program
Today

- Understanding how code executes

- Why?
  - So that we can understand what can go wrong
  - Go wrong ⇒ possible attacks
  - Go wrong ⇒ fixes / defenses

- Next lecture: attacks
Key Concepts

- Compilation steps
- x86 execution model
- Endian
- Registers
- Stack
- Heap
- Basic instructions
What Will **Executing** This Program Do?

```c
#include <stdio.h>
void answer(char *name, int x){
    printf("%s, the answer is: %d\n", name, x);
}
void main(int argc, char *argv[]){
    int x;
    x = 40 + 2;
    answer(argv[1], x);
}
```

42.c
```c
void answer(char *name, int x){
    printf("%s, the answer is: %d\n", name, x);
}
void main(int argc, char *argv[]){
    int x;
    x = 40 + 2;
    answer(argv[1], x);
}
```

David, the answer is 42
1. How is C code translated to executable code?
2. What is the machine model for executing code?
Source Language
42.c in C

Pre-processor (cpp)

Compilation

Compiler (cc1)

Assembler (as)

Linker (ld)

Target Language
42 in x86

42.c
```c
#include <stdio.h>
void answer(char *name, int x){
    printf(“%s, the answer is: %d\n”, name, x);
}
...
```

#include expansion
#define substitution
#include <stdio.h>

void answer(char *name, int x) {
    printf("%s, the answer is: %d\n", name, x);
}

...
gcc –S 42.c outputs 42.s

... 
answer:

pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $.LC0, %eax
subl $4, %esp
movl 12(%ebp)
pushl 8(%ebp)
pushl %eax
...
$ as <options>

pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $.LC0, %eax
subl $4, %esp
pushl 12(%ebp)
pushl 8(%ebp)
pushl %eax
...

Creates object code
Pre-processor (cpp) → Compiler (cc1) → Assembler (as) → Linker (ld)

Links with other files and libraries to produce an exe

$ ld <options>

```
0101100101010101101010101101101010101
1010101010101010101111111111100
0011010101101010100101011011101
01011110101001011000101010111101
```

42.o
The program **binary** (aka executable)

Final executable consists of several segments

- Text for code written
- Read-write data for constants such as “hello world” and globals
BINARY CODE EXECUTION
Basic Execution

- Binary
  - Code
  - Data
  - ...

- File system

- Process Memory
  - Stack
  - Heap

- Processor

- Fetch, decode, execute
- read and write
x86 Processor

- EIP: Address of next instruction
- EFLAGS: Condition codes
- General Purpose: EAX, EDX, ECX, EBX, ESP, EBP, ESI,EDI
Registers Have Up to 4 Addressing Modes

1. Lower 8 bits
2. Mid 8 bits
3. Lower 16 bits
4. Full register

EAX
EDX
ECX
EBX
ESP
EBP
ESI
EDI
EAX, EDX, ECX, and EBX

- 32 bit registers (three letters)
- Lower bits (bits 0-7) (two letters, ending with L)
- Mid-bits (bits 8-15) (two letters, ending with H)

- Lower 16 bits (bits 0-15) (2 letters, ending with X)
## ESP, EBP, ESI, and EDI

<table>
<thead>
<tr>
<th>EAX</th>
<th>AH</th>
<th>AL</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDX</td>
<td>DH</td>
<td>DL</td>
</tr>
<tr>
<td>ECX</td>
<td>CH</td>
<td>CL</td>
</tr>
<tr>
<td>EBX</td>
<td>BH</td>
<td>BL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ESP</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBP</td>
<td>BP</td>
</tr>
<tr>
<td>ESI</td>
<td>SI</td>
</tr>
<tr>
<td>EDI</td>
<td>DI</td>
</tr>
</tbody>
</table>

- Lower 16 bits (bits 0-15) (2 letters)
## Register to Register Operations

AT&T syntax, which is the default for objdump, uses `<src>`, `<dst>`

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov %ebx, %eax</td>
<td>Move contents of ebx into eax</td>
</tr>
<tr>
<td>add %ebx, %eax</td>
<td>Calculate eax = eax + ebx</td>
</tr>
<tr>
<td>shl $2, %ecx</td>
<td>Calculate ecx = ecx &lt;&lt; 2</td>
</tr>
</tbody>
</table>

Constants, also called “immediates”, are specified with “$”
### AT&T vs Intel

AT&T syntax, which is the default for objdump, uses `<src>, <dst>`

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov %ebx, %eax</td>
<td>Move contents of ebx into eax</td>
</tr>
<tr>
<td>add %ebx, %eax</td>
<td>Calculate eax = eax + ebx</td>
</tr>
<tr>
<td>shl $2, %ecx</td>
<td>Calculate ecx = ecx &lt;&lt; 2</td>
</tr>
</tbody>
</table>

AT&T syntax, which is the default for objdump, uses `<src>, <dst>`.

We will use the AT&T syntax in lecture.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov eax, ebx</td>
<td>Move contents of ebx into eax</td>
</tr>
<tr>
<td>add eax, ebx</td>
<td>Calculate eax = eax + ebx</td>
</tr>
<tr>
<td>shl ecx, 2</td>
<td>Calculate ecx = ecx &lt;&lt; 2</td>
</tr>
</tbody>
</table>
x86: **Byte Addressable Memory**

It’s convention: lower address at the bottom

Address 3 holds 1 byte
Address 2 holds 1 byte
Address 1 holds 1 byte
Address 0 holds 1 byte

I can get
- Address 0
- Address 5
- Address 7

Memory is just like using an array!

Alternative: **Word addressable**
For a 32-bit word size, I could fetch 4 bytes from address 0, but not from address 6 since it’s not a multiple of the word size
x86: Addressing Bytes

Addresses are indicated by operands that have a bracket “[]” or paren “()”, depending on Intel vs. AT&T.

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0x0</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>

What does `mov (%al), %dl` do?

Moves 0xccc into dl
Addresses are indicated by operands that have a bracket “[]” or paren “()”, depending on Intel vs. AT&T.

What does `mov (%eax), %edx` do?

Tricky Question! registers are 32-bits but memory is byte addressable.
mov (%eax), %edx

calculate the 32-bit address

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0xcc</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>

Move 4 bytes into edx

Two questions:
1) Which 4 bytes do we load?
2) How are those 4 bytes put a register?

4-byte load of address a is shorthand for a, a+1, a+2, a+3.

“endianess”
mov (%eax), %edx

**Endianess:** Ordering of individually addressable units

**Little Endian:** Least significant byte first

... so ...

address \( a \) goes in the least significant byte (the **littles**t bit) \( a+1 \) goes into the next byte, and so on.
mov (%eax), %edx

Endianess: the ordering of individually addressable units

Little Endian: least significant byte first
... so ...
address \( a \) goes in the least significant byte (the li\_tt\_lest bit) \( a+1 \) goes into the next byte, and so on.

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0xccc</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>

EDX holds 0xffeeddcc!
mov %ebx, (%eax)

**Endianess:** Ordering of individually addressable units

**Little Endian:** Least significant byte first

... so ...

address \(a\) goes in the least significant byte (the *littlest* bit) \(a+1\) goes into the next byte, and so on.

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>0x3</td>
</tr>
<tr>
<td>edx</td>
<td>0xccc</td>
</tr>
<tr>
<td>ebx</td>
<td>0x5</td>
</tr>
</tbody>
</table>
There are other ways to address memory than just (register).

“Common addressing modes”
void int_buffer_index() {
    int buf[10];
    unsigned int i, ptraddr;
    for (i = 0; i < 10; i++){
        buf[i] = i;
    }
    ptraddr = (unsigned int) buf;
    for (i = 0; i < 10; i++){
        printf("buf[%d] = %d.\t", i, buf[i]);
        printf("*(buf+%d) = %d\t", i, *(buf+i));
        printf("*(ptraddr+sizeof(int)*%d = %d\n", i,
                *((int *) (ptraddr+sizeof(int)*i)));
    }
}
Specifically,

```c
int buf[10];
buf[2];
*(buf+2);
```

is shorthand (syntactic sugar) for

```c
int buf[10];
*(buf + sizeof(int)*2);
```

Generally,

```c
Type buf[s];
buf[index];
```

```c
Type buf[s];
*(buf + sizeof(Type)*index);
*(buf + sizeof(int)*2);
```
Say at \( r_1 + \text{imm} \)

\[(\text{buf} + \text{sizeof(\text{Type})}*\text{index})\]

Want address: \( r_1 + \text{imm} + r_2*s \)

Constant *scaling* factor \( s \), typically 1, 2, 4, or 8

Say in Register \( r_2 \)

Must be 1, 2, 4, or 8

Assembly: \( \text{imm} (r_1, r_2, s) \)
... = (unsigned int) buf;

Where buf is at address
register + imm
becomes

lea    -0x38(%ebp),%eax

Load address of buf,
which is at ebp-0x38, into eax
Suppose instead of

```c
... = (unsigned int) buf;
```

We had

```c
... = *buf;
```

Now, we want to dereference buf in addition to getting the address

```assembly
lea -0x38(%ebp),%eax
```

Becomes

```assembly
mov -0x38(%ebp),%eax
```
Suppose I want to access address 0xdeadbeef directly

Loads the address

```
lea    0xdeadbeef,%eax
```

Deref the address

```
mov    0xdeadbeef,%eax
```

Note missing $. This distinguishes the address from the value.
# Common Addressing Modes

<table>
<thead>
<tr>
<th>Form</th>
<th>Meaning on memory M</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm (r)</td>
<td>M[r + imm]</td>
</tr>
<tr>
<td>imm (r₁, r₂)</td>
<td>M[r₁ + r₂ + imm]</td>
</tr>
<tr>
<td>imm (r₁, r₂, s)</td>
<td>M[r₁ + r₂*s + imm]</td>
</tr>
<tr>
<td>imm</td>
<td>M[imm]</td>
</tr>
</tbody>
</table>

What do you think (r₁, r₂, s) means?

M[r₁ + r₂*s]
Done:

- Register to register moves
- Register to memory and memory to register moves

Next: Control flow
Then: Program memory organization and functions
Assembly Is “Spaghetti Code”

Nice C Abstractions
- if-then-else
- while
- for loops
- do-while

Assembly
- Jump
- Branch
Jumps
- `jmp 0x45`, called a *direct jump*
- `jmp *%eax`, called an *indirect jump*

Branches
- `if (EFLAG) jmp x`
  Use one of the 32 EFLAG bits to determine if jump taken

*Note:* No direct way to get or set EIP
Implementing “if”

C
1. if(x <= y)
2.   z = x;
3. else
4.   z = y;

Pseudo-Assembly
A. Test if x <= y by computing x – y. Set EFLAGS.
   a. CF set if x < y
   b. ZF set if x==y
B. Test EFLAGS. If both CF and ZF not set, branch to E
C. Fall-through: mov x, z
D. Jump to F
E. mov y, z
F. <end of if-then-else>

Implemented using 2 instructions
1. Set eflag to conditional
2. Test eflag and branch
if \((x \leq y)\)

\(eax\) holds \(x\) and \(0xc(\%ebp)\) holds \(y\)

\(\text{cmp } 0xc(\%ebp), %eax\)

\(\text{ja } addr\)

Same as “sub” instruction

\(r = %eax - M[ebp+0xc]\), i.e., \(x - y\)

Jump if CF=0 and ZF=0

\((x \geq y) \land (x \neq y) \Rightarrow x > y\)
Setting EFLAGS

- Instructions may set an eflag, e.g.,
- “cmp” and arithmetic instructions most common
  - Was there a carry (CF Flag set)
  - Was the result zero (ZF Flag set)
  - What was the parity of the result (PF flag)
  - Did overflow occur (OF Flag)
  - Is the result signed (SF Flag)
Aside: Although the x86 processor knows every time integer overflow occurs, C does not make this result visible.

From the Intel x86 manual
### Instr. Description Condition

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>JO</td>
<td>Jump if overflow</td>
<td>OF == 1</td>
</tr>
<tr>
<td>JNO</td>
<td>Jump if not overflow</td>
<td>OF == 0</td>
</tr>
<tr>
<td>JS</td>
<td>Jump if sign</td>
<td>SF == 1</td>
</tr>
<tr>
<td>JZ</td>
<td>Jump if zero</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>JE</td>
<td>Jump if equal</td>
<td>ZF == 1</td>
</tr>
<tr>
<td>JL</td>
<td>Jump if less than</td>
<td>SF &lt;&gt; OF</td>
</tr>
<tr>
<td>JLE</td>
<td>Jump if less than or equal</td>
<td>ZF == 1 or SF &lt;&gt; OF</td>
</tr>
<tr>
<td>JB</td>
<td>Jump if below</td>
<td>CF == 1</td>
</tr>
<tr>
<td>JP</td>
<td>Jump if parity</td>
<td>PF == 1</td>
</tr>
</tbody>
</table>
All C control flow is implemented by

- A test on operands
- A branch to a location if true
- A fall-through to the next instruction if not true
Done:
– Register to register moves
– Register to memory and memory to register moves
– Control flow

Next:
Program memory organization and functions
The stack grows down towards lower addresses

- Stack grows down
- Heap grows up

Memory
- Object Code
- Program Data
- OS Data

%esp

brk

User stack

Shared libraries

Run time heap

0xC0000000
(3GB)

0x00000000

The stack grows down towards lower addresses
Variables

- **On the stack**
  - Local variables

- **On the heap**
  - Dynamically allocated via new/malloc/etc.
Procedures

- Procedures/functions are not native to assembly
- Compilers *implement* procedures
  - On the stack
  - Following the call/return stack discipline
Procedures/Functions

- We need to address several issues:
  1. How to allocate space for local variables
  2. How to pass parameters
  3. How to pass return values
  4. How to share 8 registers with an infinite number of local variables

- A stack frame provides space for these values
  - Each procedure invocation has its own stack frame
  - Stack discipline is LIFO
    - If procedure A calls B, B’s frame must exit before A’s
orange(…)
{
    ...
    red()
    ...
}

red(…)
{
    ...
    green()
    ...
    green()
}

green(…)
{
    ...
    green()
    ...
}
Frame for
- locals
- pushing parameters
- temporary space

Call to red "pushes"
new frame

When green returns it "pops" its frame

Function Call Chain

orange
↓
red
↓
green
↓
green
↓
...

Call to red "pushes" new frame

Pushes new frame
int orange(int a, int b) {
    char buf[16];
    int c, d;
    if(a > b) {
        c = a;
        d = red(c, buf);
    } else {
        c = b;
        d = red(c, buf);
    }
    return d;
}

Calling convention determines the above features

Need to access arguments

Need space to store local vars (buf, c, and d)

Need space to put arguments for callee

Need a way for callee to return values
cdecl – the default for Linux & gcc

```c
int orange(int a, int b) {
  char buf[16];
  int c, d;
  if(a > b) c = a; else c = b;
  d = red(c, buf);
  return d;
}
```

Don’t worry! We will walk through these one by one.

- Parameter area (caller)
  - return addr
  - caller’s ebp
  - callee-save
  - locals (buf, c, d ≥ 24 bytes if stored on stack)
- Caller-save
  - Buf
  - C
  - return addr
  - orange’s ebp
  - ...
When orange attains control,
1. return address has already been pushed onto stack by caller
When orange attains control,

1. return address has already been pushed onto stack by caller
2. own the frame pointer
   - push caller’s ebp
   - copy current esp into ebp
   - first argument is at ebp+8
When *orange* attains control,

1. return address has already been pushed onto stack by caller
2. own the frame pointer
   - push caller’s ebp
   - copy current esp into ebp
   - first argument is at ebp+8
3. save values of **other callee-save** registers *if used*
   - edi, esi, ebx: via push or mov
   - esp: can restore by arithmetic
When **orange** attains control,

1. **return address** has already been pushed onto stack by caller

2. **own the frame pointer**
   - push caller’s ebp
   - copy current esp into ebp
   - first argument is at ebp+8

3. save values of **other** callee-save registers *if used*
   - edi, esi, ebx: via push or mov
   - esp: can restore by arithmetic

4. **allocate** space for locals
   - subtracting from esp
   - “live” variables in registers, which on contention, can be “**spilled**” to stack space
For *caller* orange to call *callee* red,

...%ebp

b

a

return addr

caller’s ebp

callee-save

locals
(buf, c, d ≥ 24 bytes if stored on stack)

%esp
For **caller** orange to call **callee** red,

1. push any caller-save registers if their values are needed after red returns
   - eax, edx, ecx

```
<table>
<thead>
<tr>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>return addr</td>
</tr>
<tr>
<td>caller’s ebp</td>
</tr>
<tr>
<td>callee-save</td>
</tr>
<tr>
<td>locals</td>
</tr>
<tr>
<td>(buf, c, d ≥ 24 bytes if stored on stack)</td>
</tr>
<tr>
<td>caller-save</td>
</tr>
</tbody>
</table>
```
For **caller** orange to call **callee** red,

1. push any caller-save registers if their values are needed after **red** returns
   - eax, edx, ecx

2. push arguments to **red** from right to left (reversed)
   - from callee’s perspective, argument 1 is nearest in stack

<table>
<thead>
<tr>
<th>...</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>return addr</td>
<td></td>
</tr>
<tr>
<td>caller’s ebp</td>
<td></td>
</tr>
<tr>
<td>callee-save</td>
<td></td>
</tr>
<tr>
<td>locals</td>
<td></td>
</tr>
<tr>
<td>(buf, c, d ≥ 24 bytes if stored on stack)</td>
<td></td>
</tr>
<tr>
<td>caller-save</td>
<td></td>
</tr>
<tr>
<td>buf</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>
For *caller* orange to call *callee* red,

1. push any caller-save registers if their values are needed after red returns
   - eax, edx, ecx

2. push arguments to red from right to left (reversed)
   - from callee’s perspective, argument 1 is nearest in stack

3. push return address, i.e., the *next* instruction to execute in orange after red returns
For **caller** orange to call **callee** red,

1. **push any caller-save registers if their values are needed after red returns**
   - eax, edx, ecx

2. **push arguments to red from right to left (reversed)**
   - from callee’s perspective, argument 1 is nearest in stack

3. **push return address, i.e., the next instruction to execute in orange after red returns**

4. **transfer control to red**
   - usually happens together with step 3 using `call`
When red attains control,

1. return address has already been pushed onto stack by orange
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer
When **red** attains control,

1. return address has already been pushed onto stack by **orange**
2. own the frame pointer
3. ... (**red** is doing its stuff) ...
When red attains control,
1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. … (red is doing its stuff) …
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. ... (red is doing its stuff) ...
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
7. restore orange’s frame pointer
   - pop %ebp
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. ... (red is doing its stuff) ...
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
7. restore orange’s frame pointer
   - pop %ebp
8. return control to orange
   - ret
   - pops return address from stack and jumps there
When **orange** regains control,
When **orange** regains control,

1. **clean up arguments to red**
   - adding to esp
2. **restore any caller-save registers**
   - pops
3. ...

<table>
<thead>
<tr>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>return addr</td>
</tr>
<tr>
<td>caller’s ebp</td>
</tr>
<tr>
<td>callee-save</td>
</tr>
<tr>
<td>locals (buf, c, d ≥ 24 bytes if stored on stack)</td>
</tr>
</tbody>
</table>

%ebp  
%esp
### cdecl – One Slide

<table>
<thead>
<tr>
<th>Action</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>caller saves: eax, edx, ecx</td>
<td>push (old), or mov if esp already adjusted</td>
</tr>
<tr>
<td>arguments pushed right-to-left</td>
<td></td>
</tr>
<tr>
<td>linkage data starts new frame</td>
<td>call pushes return addr</td>
</tr>
<tr>
<td>callee saves: ebx, esi, edi, ebp, esp</td>
<td>ebp often used to deref args and local vars</td>
</tr>
<tr>
<td>return value</td>
<td>pass back using eax</td>
</tr>
<tr>
<td>argument cleanup</td>
<td>caller’s responsibility</td>
</tr>
</tbody>
</table>
Terminology

- **Function Prologue** – instructions to set up stack space and save callee saved registers
- **Function Epilogue** - instructions to clean up stack space and restore callee saved registers
- Why do we need calling conventions?
- Does the callee *always* have to save callee-saved registers?
Today’s Key Concepts

- Register to register moves
  - Register mnemonics
- Register/memory
  - mov and addressing modes for common codes
- Control flow
  - EFLAGS
- Program memory organization
  - Stack grows down
- Functions
  - Pass arguments, callee and caller saved, stack frame
Today’s Key Concepts

- Register to register moves
  - Register mnemonics

- Register/memory
  - mov and additional addressing modes for memory

- Control flow
  - EFLAGS

- Program memory
  - Stack grows down

- Functions
  - Pass arguments, callee and caller saved, stack frame

Next: Attacks!
For More Information

- Overall machine model: Computer Systems, a Programmer’s Perspective by Bryant and O’Hallaron
- Calling conventions: