Lec 6: Machine Code

CS492E: Introduction to Software Security

Sang Kil Cha
Our goal in software security is to find out whether a program is secure or not.

To do so, we need to see how the program binary (= executable code) executes on a machine.
Compilation

#include <stdio.h>
int someFunction(int a, int b)
{
    int s = a + b;
    printf("%d\n", s);
    return s;
}

int main(int argc, char* argv[])
{
    int x = 0;
    return someFunction(x, 42);
}
Compilation

Source Code

Intermediate Code

Assembly Code

The last human-readable format

Binary Code

0:  push ebp
1:  mov ebp, esp
3:  sub esp, 0x18

0101010101101110110000010111110101000010101001010111010000101001010101001011101011011101100000101111101010001010100101011101010101001010010101010100101110101101110110001010010100101001010010101010101010010111010110
Executable Binary
(= Executable, or Binary)

Show information about segments.

Each segment maps to one or more virtual memory areas (VMAs).

Header

Text section (.text)
Read-only (.rodata)
...

Data section (.data)
.bss
...

Segment

Segment
Executable Binary
(= Executable, or Binary)

Header
- Text section (.text)
- Read-only (.rodata)
- ...
- Data section (.data)
- .bss
- ...

Segment

Segment

Memory

VMA

VMA
Segmentation Fault
(= SegFault or Access Violation)

Happens when we reference an unmapped memory address.
x86 (IA-32) Architecture
x86

• Developed by Intel in 1985
• 32-bit address space
• One of the most common architecture
Binary File

Library File

File System

Virtual Memory

Stack

Heap

Read/Write

x86 CPU
File System

Virtual Memory

Binary File

Library File

Stack

Heap

Program Counter (= Instruction Pointer)

Read/Write

ESP

EBP

EIP

Stack Pointer

High

Low

File System

Virtual Memory
Registers in x86

- **General Purpose Registers**
  - EAX, EBX, ECX, EDX

- **Pointers**
  - ESI, EDI

- **Stack Pointers**
  - ESP: points to the top of the stack
  - EBP: points to the base of the current stack frame

- **Special Registers:**
  - EIP: instruction pointer
  - EFLAGS: holds the state of the processor

All of them have a size of a **double word** (= 32 bit)
Wait, Double Word?

- A word is the natural *unit* of data used by a processor.
- Typically, a word size is 32 bits on a 32-bit machine, and 64 bits on a 64-bit machine.

However, in x86, we say a word is 16 bits and a double word is 32 bits even though it is a 32-bit processor.
History of Intel/AMD Processors

1978: 8086
1982: 80286
1985: 80386
1989: 80486
...
2003: Opteron
2005: Prescott
2006: Core 2
...

16-bit processor, Registers (SP, BP, IP, …)

32-bit processor, Registers (ESP, EBP, EIP, …)

64-bit processor, Registers (RSP, RBP, RIP, …)

Word size was 16-bit

x86 or IA-32

x86-64 or AMD64
x86 Convention

• Word = 16 bits
• Double Word (DWORD) = 32 bits
• Quad Word (QWORD) = 64 bits

• Linear address space = 0 ~ $2^{32}$ bits
### x86 Register Accesses

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

- Bit 32 to 0
- 32 bits
- AX, BX, CX, DX
x86 Memory Access = Byte Addressing

We can access data from a byte, even though x86 is a 32-bit architecture
x86 Assembly
Basic Format of x86 Instructions

2 operands

1 operand

- Register names are not case sensitive in assembly code.
Basic Format of x86 Instructions

0 operand

ret

 Opcode
Opcode Decides Semantics

\[
\begin{align*}
\text{mov} & \quad \text{eax, ebx} & \Rightarrow & \quad \text{eax} \leftarrow \text{ebx} \\
\text{sub} & \quad \text{esp, 0x8} & \Rightarrow & \quad \text{esp} \leftarrow \text{esp} - 0x8 \\
\text{inc} & \quad \text{eax} & \Rightarrow & \quad \text{eax} \leftarrow \text{eax} + 1
\end{align*}
\]
Operand Types

- **Register**
  - `mov eax, [ebx]`
- **Memory pointed by ebx**
  - `sub esp, 0x8`
- **Constant integer**
  - `mov cl, BYTE ptr [eax]`
- **Pointer directive**
  - `mov eax, [ebx]`
**Pointer Directive**

\[
\begin{align*}
\text{mov} & \quad [esi], \text{al} \quad ; \text{ok} \\
\text{mov} & \quad [esi], 1 \quad ; \text{error (ambiguous)} \\
\text{mov} & \quad \text{DWORD PTR [esi]}, 1 \\
& \quad \text{or} \\
\text{mov} & \quad \text{WORD PTR [esi]}, 1 \\
& \quad \text{or} \\
\text{mov} & \quad \text{BYTE PTR [esi]}, 1
\end{align*}
\]
Moving Data Around (mov)

mov eax, ebx
mov al, bl
mov [eax], ebx
mov eax, [ebx]
mov eax, [ebx + edx * 4]
mov al, BYTE PTR [esi]
mov eax, 42
mov [ebx], 42
mov BYTE PTR [eax], 42

Register to Register
Register to Memory
Memory to Register
Constant to Register
Constant to Memory
Example: Storing a DWORD in Memory

mov [eax], 0xdeadbeef ; eax = 0x1000

VS.
Endianness

The order in which a sequence of bytes are stored in memory

• Big Endian = The MSB goes to the lowest address
• Little Endian = The LSB goes to the lowest address

x86 uses Little Endian
Addressing Modes

• Addressing mode specifies how an instruction can access a memory location.
• There are many ways to represent a memory address other than just: [register]
• For example
  - [register + register]
  - [register + register * num]
  - [register + register * num + num]

  e.g., mov eax, [edx + ebx * 4 + 8]
Addressing Modes

\[
\begin{align*}
&\{\text{eax, ebx, ecx, edx, esp, ebp, esi, edi}\} \\
&\quad + \{\text{eax, ebx, ecx, edx, ebp, esi, edi}\} \times \{1, 2, 4, 8\} \\
&\quad + \text{displacement}
\end{align*}
\]
Loading Address (lea)

lea eax, [ebx]
lea eax, [ebp-0x8]
What’s the Difference?

\[
\begin{align*}
\text{mov } eax, [ebp + 0x10] \\
\text{vs.} \\
\text{lea } eax, [ebp + 0x10]
\end{align*}
\]

\[
\begin{align*}
eax & \leftarrow *(ebp + 0x10) \\
\text{vs.} \\
eax & \leftarrow (ebp + 0x10)
\end{align*}
\]
Stack Operations

Stack grows backward

esp points to the top of the stack!
Stack Operations

*Push*

*Pop*

Stack grows backward
Stack Operations (push)

push eax ➔ Push register on the stack
push 0x42 ➔ Push constant on the stack
push [eax] ➔ Push a value at the memory address on the stack

push x = sub esp, 4
         mov [esp], x
Stack Operations (pop)

- `pop eax` → Pop the top element of the stack into register
- `pop [eax]` → Pop the top element of the stack into the memory address

```
pop x = mov x, [esp]
       add esp, 4
```
Stack Operations (leave)

\[
\text{leave} \quad = \quad \text{mov esp, ebp} \\
\text{pop ebp}
\]
Call (call)

call foo ; call function foo
Nextret: ; next label after returning
; from foo

= push Nextret
  jmp foo
Return (ret)

```assembly
call foo ; call function foo
Nextret: ; next label after returning
    ; from foo

= push Nextret
    jmp foo

---------------------------------------------

ret ; return to the caller

= pop eip
```

Stack grows backward

---

Nextret

Stack
Arithmetic

```assembly
add  eax, [ebx]  
sub  esp, 0x40  
inc  ecx  
dec  edx  
and  [eax + ecx], ebx  
xor  edx, ebx  
shl  eax, 1  
...```

Control Flow

if ( x ) A();
else B();

while ( x ) { }

for ( i = 0; i < n; i++ )
{ }

How to represent in assembly?
Use Only “IF”s and “GOTO”s

if ( x ) A();
else B();

while ( x ) { }

for ( i = 0; i < n; i++ )
{ }

if ( !x ) goto F;
A(); goto E;
F:
B();
E: // next …

WHILE:
if ( !x ) goto DONE;
...
goto WHILE;
DONE: // next …

i = 0;
LOOP:
if ( i >= n ) goto DONE;
...
i++; goto LOOP;
Use Only “IF”s and “GOTO”s

This is roughly how assembly looks like

```assembly
if ( !x ) goto F;
A(); goto E;
F:
B();
E: // next …

WHILE:
if ( !x ) goto DONE;
...
goto WHILE;
DONE: // next …

i = 0;
LOOP:
if ( i >= n ) goto DONE;
...
i++; goto LOOP;
```
Jump and Branch

cmp x, 0 ; test if x is zero
je F ; if x = zero then goto F

cmp i, n ; test if i >= n
jge DONE ; if x = zero then goto F

if ( !x ) goto F;
A(); goto E;
F:
B();
E: // next ...

WHILE:
if ( !x ) goto DONE;
...
goto WHILE;
DONE: // next ...

i = 0;
LOOP:
if ( i >= n ) goto DONE;
...
i++; goto LOOP;

jmp LOOP
cmp x, 0 ; test if x is zero
je F ; if x != zero then goto F

Where do we store the result of the comparison?

Implicitly fetch the stored result to perform conditional branch
EFLAGS: Storing the Processor State

Figure from Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1, Chapter 3
# Branch Instructions

<table>
<thead>
<tr>
<th>Branch Instruction</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ja</td>
<td>CF = 0 and ZF = 0</td>
<td>Jump if above</td>
</tr>
<tr>
<td>jb</td>
<td>CF = 1</td>
<td>Jump if below</td>
</tr>
<tr>
<td>je</td>
<td>ZF = 1</td>
<td>Jump if equal</td>
</tr>
<tr>
<td>jl</td>
<td>SF ≠ OF</td>
<td>Jump if less</td>
</tr>
<tr>
<td>jle</td>
<td>ZF = 1 or SF ≠ OF</td>
<td>Jump if less or equal</td>
</tr>
<tr>
<td>jna</td>
<td>CF = 1 or ZF = 1</td>
<td>Jump if not above</td>
</tr>
<tr>
<td>jnb</td>
<td>CF = 0</td>
<td>Jump if not below</td>
</tr>
<tr>
<td>jz</td>
<td>ZF = 1</td>
<td>Jump if zero</td>
</tr>
</tbody>
</table>

… and many more

(For more information, see Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 2, Chapter 3)
CMP Internals

cmp x, y
je F ; if x = y then goto F

T := x − y
ZF := if T = 0 then 1 else 0
CF := if x < 0 then 1 else 0
...

If ZF = 1 then goto F
Else fall-through
So Far

• We learned how to move around data
  – How to load/store data from/to memory and registers
  – How to compute a pointer (address) for a memory
  – How to use stack (push/pop)

• We learned how to perform arithmetic operations

• We also learned how to control program’s flow
  – Compare values and conditionally jump based on the comparison
  – Directly jump to a certain location

Already Turing Complete!
How to Know Instruction Semantics?

• Read the manual

• Or use \textbf{B2R2}
Running B2R2

• Install .NET 6 (or above) SDK
  − https://dotnet.microsoft.com/en-us/download/dotnet/6.0
  − Choose the right OS (any OS should work)
  − After the installation, you should be able to run `dotnet` command from your terminal

• Type the following command in the terminal:

  dotnet tool install --global B2R2.RearEnd.Launcher --version 0.6.0-alpha

• Now run B2R2 by typing “b2r2” in your terminal
Example: CMP EAX, 0

```
$ b2r2 dump -i x86 -s 83f80074de --lift

[83 F8 00]
(3) {
    T_1:I32 := EAX
    T_2:I32 := 0x0:I32
    T_3:I32 := (T_1:I32 - T_2:I32)
    CF := (T_1:I32 < T_2:I32)
    OF := (((T_2:I32 ^ T_1:I32) & (T_3:I32 ^ T_1:I32))[31:31])
    AF := (((T_2:I32 ^ (T_1:I32 ^ T_3:I32)) & 0x10:I32) = 0x10:I32)
    SF := (T_3:I32[31:31])
    ZF := (T_3:I32 = 0x0:I32)
    T_4:I32 := ((T_3:I32 >> 0x4:I32) ^ T_3:I32)
    T_5:I32 := ((T_4:I32 >> 0x2:I32) ^ ((T_3:I32 >> 0x4:I32) ^ T_3:I32))
    PF := (~ (((T_5:I32 >> 0x1:I32) ^ ((T_4:I32 >> 0x2:I32) ^ ((T_3:I32 >> 0x4:I32) ^ T_3:I32))))[0:0])
} // 3
```
Example: je  -0x20

$ b2r2 dump -i x86 -s 74de --lift
[74 DE]
(2) {
if ZF then ijmp (EIP + 0xfffffffffe0:I32) else ijmp (EIP + 0x2:I32)
} // 2
Compilation

Source Code

GNU AS (Assembler)

Intermediate Code

Assembly Code

Binary Code

0: push ebp
1: mov ebp, esp
3: sub esp, 0x18

0101010110111011
0000010111110100
0010101001011101
GNU AS (Assembler)

$ as file.s
$ ls a.out

.intel_syntax noprefix
mov eax, ebx
...

When testing on a 64-bit machine, use --32 option:
$ as --32 file.s
Intel Syntax?

There are two ways to represent x86 assembly code.

• At&t: `mov %eax, %ebx ; src, dst in reverse`
• Intel: `mov ebx, eax`

*We will only use Intel syntax!*
Questions?