Binary Code Analysis and Secure Software Systems

02 – Assembly Overview

Sang Kil Cha
Did you read "Reflections on Trusting Trust"?
HW1 is out

• Where is the CTF server?

• Obey the law!
Our Environment

• Linux (Debian/Ubuntu) on x86 and x86-64
• GNU Binutils (objdump, readelf, strip, etc.)
• GNU Debugger (GDB)
• No Ida Pro
• Vagrant VM for exercise:
  – 32-bit: http://143.248.8.33/is561-deb32.box
  – 64-bit: http://143.248.8.33/is561-deb64.box
  – Install VirtualBox: https://www.virtualbox.org/
  – mkdir PATH_TO_BOX; cd PATH_TO_BOX
Our Environment

• Vagrant usage
  - `vagrant box add is561 is561.box`
  - `vagrant init is561`
  - `vagrant up`
  - `vagrant ssh`  Just use these two after installation is complete

• Vagrant boxes are created with
  - Virtualbox 5.1.26 r117224 (running on macOS)
  - Vagrant 1.9.8
  - Debian 9.1.0
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


The last human-readable format

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

01010101011011101100000010111110101000010101001010111010
Linking

Source Code 1

Binary Code 1
(Object File)

Source Code 2

Binary Code 2
(Object File)

... 

Source Code n

Binary Code n
(Object File)

Executable Binary Code
(= Executable)
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
  - ...

- Memory
  - VMA
  - Segment
  - VMA
  - Segment
Segmentation Fault
(= SegFault or Access Violation)

Happens when we reference an unmapped memory address.

[Diagram of Memory with VMA segments]
x86 (IA-32) Architecture
x86

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

Binary File

Library File

Virtual Memory

High

Stack

Heap

Low

Program Code and Data
Binary File

Library File

File System

Virtual Memory

High

Low

Stack

Heap

Read/Write

Stack Pointer

ESP

EBP

EIP

Program Counter
(= Instruction Pointer)
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

…

Word size was 16-bit

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

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

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

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>Register</th>
<th>32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>AH</td>
</tr>
<tr>
<td>EBX</td>
<td>BH</td>
</tr>
<tr>
<td>ECX</td>
<td>CH</td>
</tr>
<tr>
<td>EDX</td>
<td>DH</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 32</th>
<th>16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td>BX</td>
<td>CX</td>
</tr>
<tr>
<td></td>
<td>DX</td>
<td></td>
</tr>
</tbody>
</table>
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

`mov eax, ebx`  \(\rightarrow\)  `eax \leftarrow ebx`

`sub esp, 0x8`  \(\rightarrow\)  `esp \leftarrow esp - 0x8`

`inc eax`  \(\rightarrow\)  `eax \leftarrow eax + 1`
Operand Types

- **Register**
  - `mov eax, [ebx]`

- **Memory pointed by ebx**
  - `sub esp, 0x8`

- **Constant integer**
  - `mov cl, BYTE ptr [eax]`

- **Pointer directive**
Pointer Directive

```assembly
mov [esi], al    ; ok
mov [esi], 1     ; error (ambiguous)
mov DWORD PTR [esi], 1
  or
mov WORD PTR [esi], 1
  or
mov BYTE PTR [esi], 1
```

Pointer directive is required!
Moving Data Around (mov)

- `mov eax, ebx`  
  Register to Register
- `mov al, bl`  
  Register to Memory
- `mov [eax], ebx`  
  Register to Memory
- `mov eax, [ebx]`  
  Memory to Register
- `mov eax, [ebx + edx * 4]`  
  Memory to Register
- `mov al, BYTE PTR [esi]`  
  Constant to Register
- `mov eax, 42`  
  Constant to Register
- `mov [ebx], 42`  
  Constant to Memory
- `mov BYTE PTR [eax], 42`  
  Constant to Memory
Example: Storing a DWORD in Memory

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

VS.

<table>
<thead>
<tr>
<th>0x1000</th>
<th>ef</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1003</td>
<td>be</td>
</tr>
<tr>
<td>0x1007</td>
<td>ad</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x1000</th>
<th>de</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1003</td>
<td>ad</td>
</tr>
<tr>
<td>0x1007</td>
<td>be</td>
</tr>
</tbody>
</table>
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

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

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

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

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

- Stack grows backward
- esp points to the top of the stack!
Stack Operations

*Push*

*Pop*

Stack grows backward

Stack

esp
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)

leave \equiv \text{mov esp, ebp}
\quad \text{pop ebp}
Call (call)

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

= push Nextret
  jmp foo
Return (ret)

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
Arithmetic

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++)
{ }
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;
...
go to 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
jne 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;
cmp x, 0 ; test if x is zero
jne 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

X  ID Flag (ID)
X  Virtual Interrupt Pending (VIP)
X  Virtual Interrupt Flag (VIF)
X  Alignment Check / Access Control (AC)
X  Virtual-8086 Mode (VM)
X  Resume Flag (RF)
X  Nested Task (NT)
X  I/O Privilege Level (IOPL)
S  Overflow Flag (OF)
C  Direction Flag (DF)
X  Interrupt Enable Flag (IF)
X  Trap Flag (TF)
S  Sign Flag (SF)
S  Zero Flag (ZF)
S  Auxiliary Carry Flag (AF)
S  Parity Flag (PF)
S  Carry Flag (CF)

S  Indicates a Status Flag
C  Indicates a Control Flag
X  Indicates a System Flag

| EFLAGS | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|        |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   1 |   1 |   D |   I |   V |   P |   F |   T |   R |   O |   N |   T |   N |   O |   P |   L |   D |   I |   F |   T |   S |   Z |   F |   A |   F |   P |   C |   F |

*Figure from Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1, Chapter 3.*
<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)
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!
Function: Adding an Abstraction Layer

```c
int Red(int a1)
{
    int r = 0;
    return r + Blue(a1 - 42);
}

int Blue(int a1)
{
    int b = 1;
    return b + Purple(a1, b);
}

int Purple(int a1, int a2)
{
    int p = 2;
    return p + a1 - a2;
}
```
Function: Adding an Abstraction Layer

```c
int Red(int a1) {
    int r = 0;
    return r + Blue(a1 - 42);
}

int Blue(int a1) {
    int b = 1;
    return b + Purple(a1, b);
}

int Purple(int a1, int a2) {
    int p = 2;
    return p + a1 - a2;
}
```

Q1. How to pass function parameters?

Q2. When a function returns, how to restore the register values of the caller function?

Q3. Where do we store local variables?
Stack

Higher Memory Address

Frame for function Red

Frame for function Blue

Frame for function Purple

Stack grows backward

“Top” of the stack (pointed by esp)
Calling Convention (cdecl)

Stack grows backward

Purple(a1, b);

1. Push arguments in reverse order
2. Return value is stored in eax

Frame for function Purple
Frame for function Blue
Frame for function Red

Higher Memory Address
Stack Frame

Higher Memory Address

- Frame for function Red
- Frame for function Blue
- Frame for function Purple

- Local variables for Blue
- Link to function Red
- Temporary space
- Function-call-related space
- Frame will be cleared when Blue returns
int Red(int a1)
{
    int r = 0;
    return r + Blue(a1 - 42);
}

int Blue(int a1)
{
    int b = 1;
    return b + Purple(a1, b);
}

int Purple(int a1, int a2)
{
    int p = 2;
    return p + a1 - a2;
}
<Red>:
0: push ebp
1: mov ebp,esp
3: sub esp,0x28
6: mov DWORD PTR [ebp-0xc],0x0
d: mov eax,DWORD PTR [ebp+0x8]
10: sub eax,0x2a
13: mov DWORD PTR [esp],eax
16: call Blue
1b: mov edx,DWORD PTR [ebp-0xc]
1e: add eax,edx
20: leave
21: ret

<Blue>:
22: push ebp
23: mov ebp,esp
25: sub esp,0x28
28: mov DWORD PTR [ebp-0xc],0x1
2f: mov eax,DWORD PTR [ebp-0xc]
32: mov DWORD PTR [esp+0x4],eax
36: mov eax,DWORD PTR [ebp+0x8]
39: mov DWORD PTR [esp],eax
3c: call Purple
41: mov edx,DWORD PTR [ebp-0xc]
44: add eax,edx
46: leave
47: ret

<Purple>:
48: push ebp
49: mov ebp,esp
4b: sub esp,0x10
4e: mov DWORD PTR [ebp-0x4],0x2
55: mov eax,DWORD PTR [ebp+0x8]
58: mov edx,DWORD PTR [ebp-0x4]
5b: add eax,edx
5d: sub eax,DWORD PTR [ebp+0xc]
<Red>:
0: push ebp
1: mov ebp,esp
3: sub esp,0x28
6: mov DWORD PTR [ebp-0xc],0x0
8: mov eax,DWORD PTR [ebp+0x8]
10: sub eax,0x2a
12: mov DWORD PTR [esp],eax
16: call Blue
1b: mov edx,DWORD PTR [ebp-0xc]
1e: add eax,edx
20: leave
21: ret

<Blue>:
22: push ebp
23: mov ebp,esp
25: sub esp,0x28
28: mov DWORD PTR [ebp-0xc],0x1
2a: mov eax,DWORD PTR [ebp-0xc]
2c: mov DWORD PTR [esp+0x4],eax
2e: mov eax,DWORD PTR [ebp+0x8]
30: mov DWORD PTR [esp],eax
34: call Purple
3c: call Purple
3e: mov edx,DWORD PTR [ebp-0xc]
40: add eax,edx
44: leave
46: ret

<Purple>:
48: push ebp
49: mov ebp,esp
4b: sub esp,0x10
4e: mov DWORD PTR [ebp-0x4],0x2
50: mov eax,DWORD PTR [ebp+0x8]
52: mov edx,DWORD PTR [ebp-0x4]
56: add eax,edx
5a: sub eax,DWORD PTR [ebp+0xc]
0:   push  ebp
1:   mov  ebp,esp
3:   sub  esp,0x28
6:   mov  DWORD PTR [ebp-0xc],0x0
d:   mov  eax,DWORD PTR [ebp+0x8]
10:  sub  eax,0x2a
13:  mov  DWORD PTR [esp],eax
14:  call  Blue
18:  mov  edx,DWORD PTR [ebp-0xc]
1e:   add  eax,edx
20:   leave
21:   ret

<Red>:

0xbfff0000

Execution Context

esp = 0xbfff0004
ebp = 0xbfff0020
eip = 0x1
<Red>:
0:    push  ebp
 1:    mov   ebp,esp
 3:    sub   esp,0x28
 6:    mov   DWORD PTR [ebp-0xc],0x0
d:    mov   eax,DWORD PTR [ebp+0x8]
10:   sub   eax,0x2a
13:   mov   DWORD PTR [esp],eax
16:   call   Blue
1b:   mov   edx,DWORD PTR [ebp-0xc]
1e:   add   eax,edx
20:   leave
21:   ret

<Blue>:
22:   push  ebp
23:   mov   ebp,esp
25:   sub   esp,0x28
28:   mov   DWORD PTR [ebp-0xc],0x1
2f:   mov   eax,DWORD PTR [ebp-0xc]
32:   mov   DWORD PTR [esp+0x4],eax
36:   mov   eax,DWORD PTR [ebp+0x8]
39:   mov   DWORD PTR [esp],eax
3c:   call   Purple
41:   mov   edx,DWORD PTR [ebp-0xc]
44:   add   eax,edx
46:   leave
47:   ret

<Purple>:
48:   push  ebp
49:   mov   ebp,esp
4b:   sub   esp,0x10
4e:   mov   DWORD PTR [ebp-0xc],0x2
55:   mov   eax,DWORD PTR [ebp+0x8]
58:   mov   edx,DWORD PTR [ebp-0xc]
5b:   add   eax,edx
5d:   sub   eax,DWORD PTR [ebp+0xc]

Execution Context
esp = 0xbffff0004
ebp = 0xbffff0004
eip = 0x3

Return address
0xbffff0000

0xbffff0020
<Red>:
0:  push  ebp
1:  mov  ebp,esp
3:  sub  esp,0x28
6:  mov  DWORD PTR [ebp-0xc],0x0
d:  mov  eax,DWORD PTR [ebp+0x8]
10: sub  eax,0x2a
13: mov  DWORD PTR [esp],eax
16: call  Blue
1b: mov  edx,DWORD PTR [ebp-0xc]
1e: add  eax,edx
20: leave
21: ret

<Blue>:
22:  push  ebp
23:  mov  ebp,esp
25:  sub  esp,0x28
28:  mov  DWORD PTR [ebp-0xc],0x1
2f:  mov  eax,DWORD PTR [ebp-0xc]
32:  mov  DWORD PTR [esp+0x4],eax
36:  mov  eax,DWORD PTR [ebp+0x8]
39:  mov  DWORD PTR [esp],eax
3c:  call  Purple
41:  mov  edx,DWORD PTR [ebp-0xc]
44:  add  eax,edx
46:  leave
47:  ret

<Purple>:
48:  push  ebp
49:  mov  ebp,esp
4b:  sub  esp,0x10
4e:  mov  DWORD PTR [ebp-0x4],0x2
55:  mov  eax,DWORD PTR [ebp+0x8]
58:  mov  edx,DWORD PTR [ebp-0x4]
5b:  add  eax,edx
5d:  sub  eax,DWORD PTR [ebp+0xc]
<Red>:
0:  push  ebp
1:  mov  ebp,esp
3:  sub  esp,0x28
6:  mov  DWORD PTR [ebp-0xc],0x0
d:  mov  eax,DWORD PTR [ebp+0x8]
10: sub  eax,0x2a
13: mov  DWORD PTR [esp],eax
16: call  Blue
1b: mov  edx,DWORD PTR [ebp-0xc]
1e: add  eax,edx
20: leave
21: ret

Blue>
22:  push  ebp
23:  mov  ebp,esp
25:  sub  esp,0x28
28:  mov  DWORD PTR [ebp-0xc],0x1
2f:  mov  eax,DWORD PTR [ebp-0x4]
32:  mov  DWORD PTR [esp+0x4],eax
36: mov  eax,DWORD PTR [ebp+0x8]
39: mov  DWORD PTR [esp],eax
3c: call  Purple
41: mov  edx,DWORD PTR [ebp-0xc]
44: add  eax,edx
46: leave
47: ret

Purple>
48:  push  ebp
49:  mov  ebp,esp
4b:  sub  esp,0x10
4e: mov  DWORD PTR [ebp-0x4],0x2
55: mov  eax,DWORD PTR [ebp+0x8]
58: mov  edx,DWORD PTR [ebp-0x4]
5b: add  eax,edx
5d: sub  eax,DWORD PTR [ebp+0xc]
<Red>:
0: push ebp
1: mov ebp,esp
3: sub esp,0x28
6: mov DWORD PTR [ebp-0xc],0x0
d: mov eax,DWORD PTR [ebp+0x8]
10: sub eax,0x2a
13: mov DWORD PTR [esp],eax
16: call Blue
1b: mov edx,DWORD PTR [ebp-0xc]
1e: add eax,edx
20: leave
21: ret

<Blue>:
22: push ebp
23: mov ebp,esp
25: sub esp,0x28
28: mov DWORD PTR [ebp-0xc],0x1
2f: mov eax,DWORD PTR [ebp-0xc]
32: mov DWORD PTR [esp+0x4],eax
36: mov eax,DWORD PTR [ebp+0x8]
39: mov DWORD PTR [esp],eax
3c: call Purple
41: mov edx,DWORD PTR [ebp-0xc]
44: add eax,edx
46: leave
47: ret

<Purple>:
48: push ebp
49: mov ebp,esp
4b: sub esp,0x10
4e: mov DWORD PTR [ebp-0xc],0x2
55: mov eax,DWORD PTR [ebp+0x8]
58: mov edx,DWORD PTR [ebp-0xc]
5b: add eax,edx
5d: sub eax,DWORD PTR [ebp+0xc]

Execution Context

esp = 0xbfffefffd
ebp = 0xbffff0004
eip = 0x10
eax = 0x100
<Red>:
0:   push ebp
1:   mov ebp,esp
3:   sub esp,0x28
6:   mov DWORD PTR [ebp-0xc],0x0
d:   mov eax,DWORD PTR [ebp+0x8]
10:  sub eax,0x2a
13:  mov DWORD PTR [esp],eax
16:  call Blue
1b:  mov edx,DWORD PTR [ebp-0xc]
1e:  add eax,edx
20:  leave
21:  ret

<Blue>:
22:  push ebp
23:  mov ebp,esp
25:  sub esp,0x28
28:  mov DWORD PTR [ebp-0xc],0x1
2f:  mov eax,DWORD PTR [ebp-0xc]
32:  mov DWORD PTR [esp+0x4],eax
36:  mov eax,DWORD PTR [ebp+0x8]
39:  mov DWORD PTR [esp],eax
3c:  call Purple
41:  mov edx,DWORD PTR [ebp-0xc]
44:  add eax,edx
46:  leave
47:  ret

<Purple>:
48:  push ebp
49:  mov ebp,esp
4b:  sub esp,0x10
4e:  mov DWORD PTR [ebp-0x4],0x2
55:  mov eax,DWORD PTR [ebp+0x8]
58:  mov edx,DWORD PTR [ebp-0xc]
5b:  add eax,edx
5d:  sub eax,DWORD PTR [ebp+0xc]

Execution Context

esp = 0xbfffeffdc
ebp = 0xbfff0004
eip = 0x13
eax = 0xd6
<Red>:
0:  push ebp
1:  mov ebp,esp
3:  sub esp,0x28
6:  mov DWORD PTR [ebp-0xc],0x0
d:  mov eax,DWORD PTR [ebp+0x8]
10: sub eax,0x2a
13: mov DWORD PTR [esp],eax
16: call Blue
1b: mov edx,DWORD PTR [ebp-0xc]
1e: add eax,edx
20: leave
21: ret

<Blue>:
22:  push ebp
23:  mov ebp,esp
25:  sub esp,0x28
28:  mov DWORD PTR [ebp-0xc],0x1
2f:  mov eax,DWORD PTR [ebp-0xc]
32:  mov DWORD PTR [esp+0x4],eax
36:  mov eax,DWORD PTR [ebp+0x8]
39:  mov DWORD PTR [esp],eax
3c:  call Purple
41:  mov edx,DWORD PTR [ebp-0xc]
44:  add eax,edx
46:  leave
47:  ret

<Purple>:
48:  push ebp
49:  mov ebp,esp
4b:  sub esp,0x10
4e:  mov DWORD PTR [ebp-0x4],0x2
55:  mov eax,DWORD PTR [ebp+0x8]
58:  mov edx,DWORD PTR [ebp-0xc]
5b:  add eax,edx
5d:  sub eax,DWORD PTR [ebp+0x4]

---

Execution Context

\[\text{esp} = 0xbffeffdc\]
\[\text{ebp} = 0xbfff0004\]
\[\text{eip} = 0x16\]
\[\text{eax} = 0xd6\]

---

Return to here
<Red>:
0:  push ebp
1:  mov ebp,esp
3:  sub esp,0x28
6:  mov DWORD PTR [ebp-0xc],0x0
d:  mov eax, DWORD PTR [ebp+0x8]
10: sub eax, 0x2a
13: mov DWORD PTR [esp], eax
16: call Blue
1b: mov edx, DWORD PTR [ebp-0xc]
1e: add eax, edx
20: leave
21: ret

<Blue>:
22: push ebp
23: mov ebp, esp
25: sub esp, 0x28
28: mov DWORD PTR [ebp-0xc], 0x1
2f: mov eax, DWORD PTR [ebp-0xc]
32: mov eax, DWORD PTR [esp+0x4], eax
36: mov eax, DWORD PTR [ebp+0x8]
39: mov DWORD PTR [esp], eax
3c: call Purple
41: mov edx, DWORD PTR [ebp-0xc]
44: add eax, edx
46: leave
47: ret

<Purple>:
48: push ebp
49: mov ebp, esp
4b: sub esp, 0x10
4e: mov DWORD PTR [ebp-0x4], 0x2
55: mov eax, DWORD PTR [ebp+0x8]
58: mov edx, DWORD PTR [ebp-0x4]
5b: add eax, edx
5d: sub eax, DWORD PTR [ebp+0xc]

Execution Context

esp = 0xbffeeffd8
ebp = 0xbffff0004
eip = 0x22
eax = 0xd6

0xbffff0020
0xbffff0040
0xbffff0000
0xbfff0020
0x100
0x0
0xd6
0xbff0000
return address
return address (0x1b)
Let’s fast forward to here

Execution Context

```
estp = 0xbffe0000
ebp = 0xbff00000
```

Return address (0x1b)

```
estp = 0xbffe0020
0x100
```

```
estp = 0xbffe0020
0x0
```

```
estp = 0xbffe0004
0xd6
```

Return address

```
estp = 0xbffe0004
0xbffe0000
```

```0xbffe0004```
<Red>:
0:   push   ebp
1:   mov   ebp,esp
3:   sub   esp,0x28
6:   mov   DWORD PTR [ebp-0xc],0x0
d:   mov   eax,DWORD PTR [ebp+0x8]
10:  sub   eax,0x2a
13:  mov   DWORD PTR [esp],eax
16:  call   Blue
1b:  mov   edx,DWORD PTR [ebp-0xc]
1e:  add   eax,edx
20:  leave
21:  ret

<Blue>:
22:  push   ebp
23:  mov   ebp,esp
25:  sub   esp,0x28
28:  mov   DWORD PTR [ebp-0xc],0x1
2f:  mov   eax,DWORD PTR [ebp-0xc]
32:  mov   DWORD PTR [esp+0x4],eax
36:  mov   eax,DWORD PTR [ebp+0x8]
39:  mov   DWORD PTR [esp],eax
3c:  call   Purple
41:  mov   edx,DWORD PTR [ebp-0xc]
44:  add   eax,edx
46:  leave
47:  ret

<Purple>:
48:  push   ebp
49:  mov   ebp,esp
4b:  sub   esp,0x10
4e:  mov   DWORD PTR [ebp-0x4],0x2
55:  mov   eax,DWORD PTR [ebp+0x8]
58:  mov   edx,DWORD PTR [ebp-0x4]
5b:  add   eax,edx
5d:  sub   eax,DWORD PTR [ebp+0xc]

Execution Context

esp = 0xbfffefffd8
ebp = 0xbffff0004
eip = 0x47

return address

0xbfff0020
0xbfff0000

0xbfff0020
0x100

0x0

0xd6

return address (0x1b)
Calling Convention

```c
int Blue(int a1)
{
    int b = 1;
    return b + Purple(a1, b);
}
```

Passing parameter values in a reverse order

Storing a return value in eax
Compilation

Source Code

Intermediate Code

Assembly Code

Binary Code

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

GNU AS (Assembler)

01010101011011101100000101111110101000010100011110101110100010101000101011101000101010001010111010001010100
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
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!*
Recap
Compilation

Source Code

Intermediate Code

Assembly Code

Binary Code

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

```
0101010101101110110
0000010111111010100
0010101001010111010
```

a.out
Our Goal: Understanding Binary

Source Code

Intermediate Code

Disassembly

Assembly Code

Binary Code

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

0101010101101110110
00000101111010100
001010010101110110
...

a.out
GNU objdump

$ objdump -M intel -d a.out

Intel syntax
**objdump Output**

<table>
<thead>
<tr>
<th>Address</th>
<th>Binary Code</th>
<th>Disassembled Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000 &lt;Red&gt;:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:</td>
<td>55</td>
<td>push ebp</td>
</tr>
<tr>
<td>1:</td>
<td>89 e5</td>
<td>mov ebp,esp</td>
</tr>
<tr>
<td>3:</td>
<td>83 ec 28</td>
<td>sub esp,0x28</td>
</tr>
<tr>
<td>6:</td>
<td>c7 45 f4 00 00 00 00</td>
<td>mov DWORD PTR [ebp-0xc],0x0</td>
</tr>
<tr>
<td>d:</td>
<td>8b 45 08</td>
<td>mov eax,DWORD PTR [ebp+0x8]</td>
</tr>
<tr>
<td>10:</td>
<td>83 e8 2a</td>
<td>sub eax,0x2a</td>
</tr>
<tr>
<td>13:</td>
<td>89 04 24</td>
<td>mov DWORD PTR [esp],eax</td>
</tr>
<tr>
<td>16:</td>
<td>e8 fc ff ff ff</td>
<td>call 17 &lt;Red+0x17&gt;</td>
</tr>
<tr>
<td>1b:</td>
<td>8b 55 f4</td>
<td>mov edx,DWORD PTR [ebp-0xc]</td>
</tr>
<tr>
<td>1e:</td>
<td>01 d0</td>
<td>add eax,edx</td>
</tr>
<tr>
<td>20:</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>21:</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>

*x86 uses variable-length encoding*
Question

Is perfect disassembly possible?
Key Concepts

- Compilation pipeline
- x86 architecture
- Assembly
- Disassembly
Question?