Low-Level Essentials for Understanding Security Problems

Aurélien Francillon
francill@eurecom.fr
• The modern computer architecture is based on “Von Neumann”
  – Two main parts: CPU (Central Processing Unit) and Memory
  – This architecture is used almost everywhere
  – This architecture is the most common one

• What is memory used for?
  – E.g., location of cursor, size of windows, shape of each letter being displayed, graphics of icons, text, values, etc…
  – Von Neuman also says that not only data, but programs should also be in main memory
Von Neumann Architecture

- Instructions, data from the same memory
- The most common CPU architecture
- Beware of the caches
  - Performance
  - Self modifying code
Harvard Architecture

- Program memory
- Program Addresses
- Instructions
- Instructions bus
- Core
- Data Addresses
- Data bus
- SRAM memory
- Data
- Stack
- SP

- Instructions, data from separate memory
- Common in embedded systems (AVR, PIC) and DSP
The Core (or CPU)

• Storing data by itself, of course, is not enough
  – The Core reads instructions from memory one by one
    • Executes them (*fetch-execute-cycle*)

• Following components make up the Core
  – Program Counter
  – Instruction Decoder
  – Data bus
  – General-purpose registers
  – Arithmetic and logic unit
Program Counter and Instruction Decoder

• Is used to tell the CPU where to fetch next instruction
  – There is no difference between memory and data
  – Program counter holds memory address of next instruction

• The instruction decoder then makes sense of the instruction
  – A typical instruction usually consists of memory locations as well
    • E.g., move this piece of data from memory address X to memory address Y
The Data Bus and Registers

• The data bus connects the memory and the CPU
  – i.e., it is the actual, physical wire

• In addition to the memory, the processor has high-speed, special memory location
  – Called registers
    • Special-purpose registers
    • General-purpose registers
    • Registers are used for computation
• Once instruction has been decoded, CPU passes data and decoded instruction to ALU
  – Now, the instruction is actually executed

  – Results of the computation are placed on the data bus and sent to memory locations given in the instruction

  – For example, we can tell ALU to add 1 to register A and place result in register B
Some basics (you know them ;))

• The number attached to each storage location
  – … is an address
  – A single storage location is called a byte
  – On x86 processors, a byte is between 0..255
  – Obviously, two bytes can be used to represent any number between 0..65536
  – Four bytes can be used to represent numbers between 0..4294967295., Luckily, we do not have to worry about this. The architecture helps us to do math with 4 byte numbers
  – (8 bytes/64 bits: 0..18,446,744,073,709,551,615 (i.e. 18*10^18))

• Addresses sizes are in words (a word is 4 bytes)
  – Note that this means that a number and an address are dealt with the same way
• Processors have a number of different ways of accessing data
  – Known as *addressing modes*
  – The simplest method is known as *immediate mode*
    • Data access is enabled in instruction itself
  – In *register addressing mode* the instruction contains a register to access rather than memory location
  – In *direct addressing mode*, the instruction contains memory address
  – In *indexed addressing mode*, the instruction contains memory address to access and an index register to offset
    • E.g., Address 1000, register=4, address=1004
Data Accessing Methods

- Processors have a number of different ways of accessing data
  - Indirect addressing mode
    - We take address that is stored in register
  - Base point addressing mode
    - You take address in register and add an offset
    - Used a lot
A Simple Program in Assembler

```assembly
# No INPUT
# Returns a status code, you can view it by typing echo $? 
# ebx holds the return code
.intel_syntax noprefix

.section .data
.section .text
.globl _start

_start:
    mov eax, 1  #This is the sys call for exiting program
    mov ebx, 0  #This value is returned as status
    int 0x80    #This interrupt calls the kernel, to execute sys call
```
So now we have the source code…

• So how do we create the application?
  – Well, we need to assemble and link the code
  – This can be done by using the assembler as:
    • `as exit.s -o exit.o`
    • `ld -o exit exit.o`
• .section breaks up the program into sections
  – .data, obviously, is used for variables and data you might need
  – .text is the code of the program that you write
  – .globl _start indicates that _start is a symbol, required by the linker
  – _start, in fact, indicates that the loader will load and start the program from this location
  – The next instruction, mov, has two operands: destination and source
    • mov, add, sub
A note on the syntax

• `.intel_syntax noprefix`
  – Forces that the assembly is parsed in Intel-syntax
  – `instruction destination, source`

• But:
  – Majority of GNU/Linux tools default to AT&T Syntax
  – `instruction source, destination`
  – More explicit due to `prefixes` and `suffixes`

• Used syntax is a matter of taste
  – Be aware of the differences!
The same program (AT&T syntax)

# No INPUT

# Returns a status code, you can view it by typing echo $?
# %ebx holds the return code

.section .data
.section .text
.globl _start

_start:

mov $1, %eax # This is the sys call for exiting program
movl $0, %ebx # This value is returned as status
int $0x80 # This interrupt calls the kernel, to execute sys call
Registers on the x86 Architecture

- General purpose registers:
  - eax, ebx, ecx, edx, edi, esi
- Special purpose registers:
  - ebp, esp, eip, eflags
- Some registers (e.g., eip, eflags) can only be accessed through special instructions
Let Us Write a More Complicated Program

• Task: Find the maximum of a list of numbers
  – Questions to ask:
    • Where will the numbers be stored?
    • How do we find the maximum number?
    • How much storage do we need?
    • Will registers be enough or is memory needed?
  – Let us designate registers for the task at hand:
    • edi holds position in list
    • ebx will hold current highest
    • eax will hold current element examined
"Algorithm" we use

- Check if eax is zero (i.e., termination sign)
  - If yes, exit
  - If not, increase current position edi
  - Load next value in the list to eax
    - We need to think about what addressing mode to use here
  - Compare eax (the current value) with highest value so far ebx
  - If the current value is higher, replace ebx
  - Repeat until termination
Let’s get down to the code

.intel_syntax noprefix

.section .data
    data_items:
        .long 3,67,34,222,45,75,54,34,44,33,22,11,66,0

.section .text
.globl _start
_start:
    mov edi, 0 # Reset index
    mov eax, [edi*4+data_items]
    mov ebx, eax #First item is the biggest so far
Let’s get down to the code

```
start_loop:
cmp eax, 0
je loop_exit
inc edi  # Increment edi
mov eax, [edi*4+data_items]  # load the next value
cmp eax, ebx  # Compare ebx with eax
jle start_loop  # if it is less, then just jump to the beginning
mov ebx, eax  # Otherwise, store the new largest number
jmp start_loop

loop_exit:
mov eax, 1  # Remember the exit sys call? It is 1
int 0x80
```
Important Instructions

• The compare instruction
  – `cmp eax, 0`
  – `je end_loop`
  – Other jump instructions `jg, jge, jl, jle, jmp`

• `mov` instruction
  – Used often. One of the most important and common instructions that you are going to see
    • ... and use, for example, when writing shell code
Different Addressing Modes

- **direct addressing mode**
  - `mov eax, [ADDRESS]`

- **indirect addressing mode**
  - `mov ebx, [eax]`

- **indexed addressing mode**
  - `mov eax, [ecx*1+data_start]`
    - uses the ADDRESS and the INDEX portion

- **Immediate mode (already seen)**
What if we do not want to move a word?

- Suppose you only need to move a byte at a time, not a word
  - You can use the `mov byte ptr` instruction
  - In eax, the least significant half is addressed by ax (i.e., 2 bytes)
  - ax is divided into ah and al (1 byte in size)
Memory Layout

- **Stack segment**
  - local variables
  - procedure activation records
- **Data segment**
  - global uninitialized variables (.bss)
  - global initialized variables (.data)
  - dynamic variables (heap)
- **Code (Text) segment**
  - program instructions
  - usually read-only
- In linux, under the proc filesystem
  $ cat /proc/<pid>/maps

In linux, under the proc filesystem
$ cat /proc/<pid>/maps
Functions

- A function is composed of several different pieces
  - function name
    - Symbol that represents where the function starts
  - function parameters
    - Data items passed to function for processing
  - Local variables
    - Temporary storage areas used in the function
    - Thrown away when the processing finishes
  - Static variables
    - Storage area that is reused over invocations
  - Global variables
    - Storage areas outside the function
The Return Address

• The return address is a parameter which tells the function where to resume execution after the function is completed
  – It is “invisible” – the programmer does not necessarily know the address
  – When a function is invoked, the calling point is saved
  – When the function completes, it returns to the initial calling point
  – In machine code, functions are called with call and they return when they execute ret

• Return value
  – Usually, a single value is returned to caller
The Stack

• In most architectures (Intel, Sparc, Motorola, MIPS), the stack grows toward the bottom of the memory

• The ESP register (stack pointer) always points to the current top of the stack

• Composed of frames:
  – Upon function call, a new frame is pushed on the stack
  – Upon function return, the frame is discarded

• The EBP register (base pointer) points at the current frame

• Each frame contains:
  – Return address (where to jump at the end of the function)
  – Address of the previous frame
  – Parameters passed to the function
  – Local variables of the function
The Calling Convention

• The way that the variables are stored and the parameters and return values are transferred by the computer varies
  – Depends on language, compiler, architecture …
  – The Application Binary Interface (ABI)
  – This variance is known as the *calling convention*

• The assembly language can use any calling convention
  – You can make one up yourself… however…
  – If you want to inter-operate with other languages, use libraries, you need to follow their calling conventions
    • E.g., suppose you want your code to be callable from a C program…
The Stack

• Each computer program that runs uses a region of memory called the stack to enable functions to work properly
  – You generally keep the things that you are working on toward the top, and you take things off as you are finished working with them

• The computer’s stack lives at the very top addresses of memory
  – You can push values onto stack using push
  – You can retrieve items from the stack using pop
• Where is the “top” of the stack
  – Because of architectural considerations, the computer stack grows from higher addresses to lower addresses
  – I.e., it grows downwards
  – How do we know where the “top” of the stack is?
    • The esp register stores a pointer to stack location
  – If something is pushed onto stack, the stack decreases by esp – 4 if push is used
  – Yes, but if I only want to read? How can I do this without popping?
    • mov eax, [esp]
    • mov eax, [esp+4] (for addressing a higher value)
The C calling convention

- The stack is the key element for implementing a function’s…
  - local variables, parameters, and return address
  - Before executing a function, a program pushes all of the parameters for the function onto the stack in the reverse order that they are documented
  - Program issues *call* instruction
  - *call* does two things
    - pushes address of next instruction (i.e., return)
    - Modifies eip to point to function start
The C calling convention

- Parameter #N
  ...
  Parameter 2
  Parameter 1
  Return Address <--- (esp)

- Now, function has to do some thing
  - It saves the current base pointer register ebp
    - Needed for local variables and parameters
  - push ebp
  - mov ebp, esp
• Copying the stack pointer into the base pointer at the beginning of a function allows you to always know where your parameters are

Parameter \#N \leftarrow N \times 4 + [ebp+4]

\ldots

Parameter 2 \leftarrow [ebp+12]
Parameter 1 \leftarrow [ebp+8]
Return Address \leftarrow [ebp+4]
Saved ebp \leftarrow [esp] and [ebp]
• *stack frame* consists of all of the stack variables used within a function

• Next, the function reserves space on the stack for any local variables it needs
  – This is done by simply moving the stack pointer out of the way
  – E.g., if we need two words: *sub esp, 8*
  – Variables are local because when function returns, the stack frame is reset and variables disappear
The C calling convention

• Our stack now looks like this:

  Parameter #N <--- N*4+[ebp+4]
  ...
  Parameter 2 <--- [ebp+12]
  Parameter 1 <--- [ebp+8]
  Return Address <--- [ebp+4]
  Saved ebp <--- [ebp]
  Local Variable 1 <--- [ebp-4]
  Local Variable 2 <--- [ebp-8] and [esp]
The C calling convention

• Now we can use the ebp for accessing all variables
  – ebp was specifically designed for this purpose
• When a function is done…
  – It stores its return value in eax
  – It resets the stack to what it was when it was called
    • It gets rid of the current stack frame and puts the stack frame of
      the calling code back into effect
  – It returns by invoking the *ret* command
  – mov esp, ebp
    pop ebp
    ret

SysSec
The general form of memory address references is this:

- ADDRESS_OR_OFFSET(%BASE_OR_OFFSET, %INDEX, MULTIPLIER)

To calculate the address, simply perform the following calculation:

- FINAL ADDRESS = ADDRESS_OR_OFFSET + %BASE_OR_OFFSET + MULTIPLIER * %INDEX
- ADDRESS_OR_OFFSET and MULTIPLIER must both be constants, while the other two must be registers.