**BASIC DEFINITIONS**

**Assembler**: transforms source code into instructions for the machine

**Arithmetic Logic Unit**: takes decoded instruction and actually executes the instruction

**Computer Architecture**: modern computer architecture is based off of the Von Neumann architecture, where the computer is divided into three main parts: CPU, I/O, and memory (RAM)

**Data Bus**: connects the CPU and the memory; fetches the memory locations to be used in the calculation

* **Instruction Decoder**: figures out what an instruction means

**Kernel**: computer program that manages I/O requests from software and translates them into data processing instructions for the CPU; part of an operating system that keeps track of everything; ie, Linux is kernel

**Linker**: program that puts the object files together and adds information to it so that the kernel knows how to load and run it

**Pointer**: addresses which are stored in memory

**Program Counter**: tell the computer where to fetch the next instruction from

**Register**: memory location on the CPU itself; much faster than other memory

**Source Code**: human-readable form of a program

**Word**: four-byte words (32 bits)

**ADDRESSING MODES**

**Base-Pointer Addressing Mode**: similar to indirect addressing, but you also include an offset to add to the register’s value before using it for lookup

**Direct Addressing Mode**: instruction contains the address to load the data from

**Immediate Mode**: data to access is embedded in the instruction itself

**Indexed Addressing Mode**: instruction contains an address to load the data from, and also specifies an index register to offset that address

**Indirect Addressing Mode**: instruction contains a register that contains a pointer to where the data would be loaded from

* **Tools**: x86 Processors and GNU/Linux OS
* CISC: complex instruction set computer
* RISC: reduced instruction set computer
* Register Conventions:
  + EAX: store function return values
  + EBX: base pointer to the data section
  + ECX: counter for string and loop operations
  + EDX: I/O pointer
  + ESI: source pointer for string operations
  + EDI: destination pointer for string operations
  + ESP: stack pointer
  + EBP: stack frame base pointer
  + EIP: pointer to next instruction to execute (“instruction pointer”)
* Caller-save registers: EAX, EDX, ECX
  + Caller is in charge of saving the value before a call to a subroutine and restoring the value after the call returns
  + The callee can (and is likely to) modify values in caller-save registers
* Callee-save registers: EBP, EBX, ESI, EDI
  + Callee is responsible for making sure the values are stored/restored before using more registers than there are saved by the caller
  + Callee must not modify registers which the caller didn’t save unless the callee itself saves and restores the existing values
* EFLAGS: register that holds many single bit flags
  + Zero flag (ZF): set if the result of some instruction is zero; clear otherwise
  + Sign flag (SF): set equal to the most-significant bit of the result, which is the sign bit of a signed integer (0 = positive, 1 = negative)
* THE STACK
* Conceptual area of main memory (RAM) which is designated by the OS when a program is started
* LAST IN FIRST OUT structure
* Stack grows toward lower memory addresses
* Adding something to the stack means the top of the stack is now at a lower memory address
* Ie) FFFF to 0000 (old 🡪 new)
* INSTRUCTIONS
  + NOP: no operation; just there to pad/align bytes or to delay time
  + PUSH: value to push can either be an immediate or a value in a register; automatically decrements the stack pointer (ESP) by 4 (as in, there are more things in the stack now)
  + POP: take a word off of the stack and put it in a register; increments the stack pointer (ESP) by 4; interesting to note that the value that used to be in the stack is technically still there, but it is labeled as undefined…as in, we shouldn’t really access it
  + CALL: transfer control to a different function in a way that control can later be resumed where it left off; it first pushes the address of the next instruction onto the stack (for use by RET when the procedure is done); then it changes EIP to the address given in the instruction
  + RET: return from procedure
    - Two forms
      * Pop the top of the stack into EIP (typically used by cdecl functions)
      * Pop the top of the stack into EIP and add a constant number of bytes to ESP (used by stdcall functions)
  + MOV: move
    - Register to register
    - Memory to register or register to memory
    - Immediate to register or immediate to memory
    - Never memory to memory
* CALLING CONVENTIONS
  + Cdecl
    - “C Declaration”
    - Most common convention
    - Function parameters pushed onto stack right to left
    - Caller is responsible for cleaning up the stack
  + Stdcall
    - “Standard Call”
    - Same as above for the most part
    - Callee is responsible for cleaning up any stack parameter it takes
* Exploitation (pg. 115)
* Program exploitation = staple of hacking
  + Security holes are flaws/oversights in a program’s design or the environment the program is running in