Computersysteme Zusammenfassung

01101 = 13
10011 = -13

Signed Bit

erstes Bit (von links) ist negativ alle anderen positiv

$$\overset{-8\ 0\ 0\ 1}{}$$

⇒ 1001 = -7

2's complement

→ complement of number is number negativ

→ invert every digit, then add 1

erstes Bit = 1 dann ist die Zahl negativ

erstes Bit = 0 dann ist die Zahl positiv

-16+8 +2+1

⇒ 11011 = -5

00101 = 5

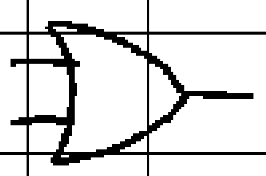
Numbersystems

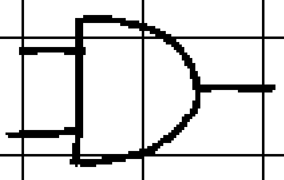
$12_{16} = 1 \cdot 16^1 + 2 \cdot 16^0 = 18$

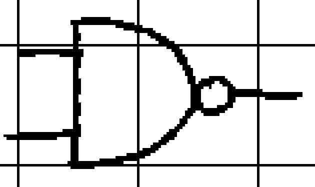
$\text{Zahl} = \text{Stelle } n \cdot \text{Basis}^n + \dots + \text{Stelle } 0 \cdot \text{Basis}^0$

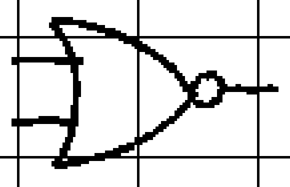
(convert on TR → 2nd → 9)

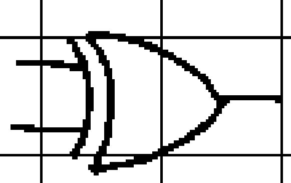
Logic Gates

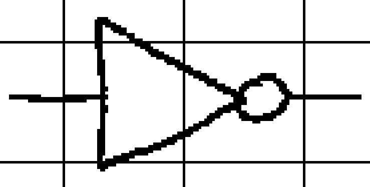
OR  – one or both TRUE = TRUE

AND  – both TRUE = TRUE

NAND  – both TRUE = False (¬AND)

NOR  – one or both TRUE = False (¬OR)

XOR  – one TRUE = TRUE

NOT  – OUTPUT = ¬ INPUT

Bit shift

ein Bit an eine gewünschte Postion schieben

0000 ; 1<<4 = 1000

Arithmetic /Logic Unit (ALU)

=) generell Purpose calculator

→ takes Binary (e.g. 3-parts)
- → first few digits = select function (z.B add, subtract)
- → second few digits = first number
- → third few digits = second number

CPU

Control Unit	- what is next to Do
ALU	- see above
Register	- fast onboard Memory (short term storage)

Bits & Bytes

8 Bit = 1 Byte · 1000 = kilobyte · 1000 = Megabyte . . .

Bit → BInary digiT

Assembly

Text machine code with instructions (machine-specific)

Language

Consists mostly of 3 Parts

$\underbrace{\text{mov}}_{\text{instruction}}\ \underbrace{\text{r3}}_{\text{register}},\ \underbrace{\#0}_{\text{number (decimal)}}$ | moves value 0 into $\underbrace{\text{register}}_{\text{contents in bits}}$ r3

movq (r3), r2 | moves the value that the address r3 points to into r2

(r3) = value at the address stored in r3

registers generally written as %rax

movT = move Top halfword → moves value to the top half of the register content and leaves the lower half unchanged

b = branch

jmp = Jump

Pointer

integer pointer written as int* x

A pointer stores an address → it "points" to the value at the address

x86-64 Registers

16 named Registers (8byte) (64-Bit) z.B.

%rax	→ Return value (register a extended)
%rbp	→ base pointer
%rsp	→ stack pointer
%rdi	→ destination index

Lower registers

32 bit (4 byte) → prefix e z.b. edi

16 bit (2 byte) → no prefix or suffix w z.b ax / r8w

8 bit (1 byte) → suffix l or b z.b. al / r8b

86-64 assambly instructions

- Immediate Operands → prefix $
- Register Names → prefix %
- Memory Locations - usually Hex (zB 0x104)

Condition Codes

codes indicate how a comparison operation turned out. there are dedicated comparison opeations in assambly so that the operation doesn't require another register to store the result. The operation just returns the condition code:

CF - opeation was carried out

ZF - operation result was 0

SF - operation result was negativ

OF - operation resulted in an overflow

Comparison operations:

cmp - does sub | test - does ~~add~~ and

Stack

part of memory → keeps track of subroutines
↳ what is currenty executed
↳ where to next

Stack grows upside down → memory addresses get smaller

%rsp points always to the "top" of the stack (bottom memory address)

- every function has its own private section on the stack
  - ↳ %rsp points to the bottom memory address
  - ↳ %rbp points to the top memory address

Stack exploit

- Stack overflow (Buffer overflow) → overload the stack with data (z.B recursively generate big values)
  - → overflowing data is written outside the bounds of the function

Malicious Buffer Overflow

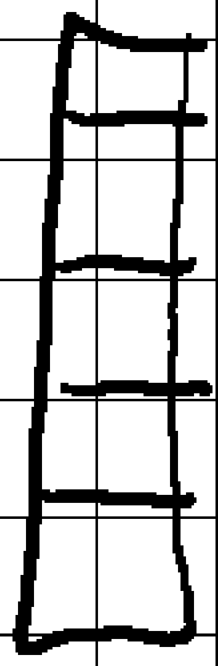
If you know the memory layout, you can write code, so that the overflowing data contains executable malicious code.
You could overwrite a pointer and let it point to an exploit payload.

Memory Layout

- Kernel Address Space - privileged processes from the operating system run here

- Empty Space - Empty

- User Address Space - software operates here



↳ Stack -
↳ Heap - maps memory in stack for usage
↳ BSS - uninitialized variables (optimization)
↳ Data - statically allocated e.g. global / static variables
↳ code - Executable machine instructions

Memory Hierarchy and caching

Memory is basically temporary fast storage

- there are multiple "levels" of memory (hierarchy)
  ↳ storing data on different 'levels' has major advantages

The fastes 'level' has the least amont of storage

L0: register → words from L1

L1: on-chip cache → holds cache lines of L2

L2: off-chip cache (SRAM) → holds cache lines from main Memory

L3: Main memory → disk blocks from disk

L4: Secondary storage → local disks hold files

L5: remote secondary storage → tapes, webservers etc.

Cache Mechanics

fastes cache is constantly replaced with values it thinks will be used

Cache Hit:

value gets replaced if it has already been used

Cache Miss:

value gets replaced if it hasn't been used

⇒ Running time increases if the required value is not in the cache
  ⇒ it has to check if in cache and then retrive it from memory

Memory access Patterns

Different languages access memory in a different way
→ resulting in faster/slower reads

⇒ Data saved as 2d array ⇒ you can read it row by row or column by column

⇒ - Row - major
  - Column - major

Sequential Processing

all instructions follow the same pattern: (rudimentary)

Fetch → Decode → Execute → Memory → write back → PC update

Processor Design

PC	- Program Counter (points at current instruction)
valP	- Predicted next PC (often next instruction)
new PC	- Determined Next PC
iCode	- instruction Code
iFun	- Function Code
rA and rB	- Register ID of valA & valB
valA and valB	- Register values
valC	- additional value in istruction (z.B. jump destination)
valE	- Result of ALU calculation
Cnd	- conditional code
dstE	- Register ID of valE
valM	- Value read from memory
dstM	- Register ID of valM

Pipelining

multiple ALUs can compute multiple things at once

unpipelined ⇒ $\underset{300c}{\text{logic}} \rightarrow \underset{20c}{\text{register}}$ (clocks = 320)

$\quad$ ↳ time

3-way-pipeline ⇒ $\underset{100c}{\text{logic}} \rightarrow \underset{20c}{\text{register}} \rightarrow \underset{100c}{\text{logic}} \rightarrow \underset{20c}{\text{register}} \rightarrow \underset{100c}{\text{logic}} \rightarrow \underset{20c}{\text{register}}$

(clocks per calc = 120)

⇒ clock speed increases

Speculative Executions

next execution is predicted with different methods

e.g. prediction based on previous cases

Exploit speculative Executions

Spectre

- programm trains CPU to predict one branch
- programm lets CPU calulate something else
  - ↳ CPU predicts wrong and calculates the wrong branch
    - ↳ CPU realices mistake and returns the other branch
- User can retrive wrong branch from memory

(E.g. acces to private data)