Alt text: I looked at some of the data dumps from vulnerable sites, and it was ... bad. I saw emails, passwords, password hints. SSL keys and session cookies. Important servers brimming with visitor IPs. Attack ships on fire off the shoulder of Orion, c-beams glittering in the dark near the Tannhäuser Gate. I should probably patch OpenSSL.

http://xkcd.com/1353/
Administrivia

- Questions doc: [https://tinyurl.com/CSE351-7-27](https://tinyurl.com/CSE351-7-27)

- hw14 due Wednesday (7/29) – 10:30am
  - This one is especially long, please start early

- hw15 due Friday (7/31) – 10:30am

- Lab 3 due Friday (7/31) – 11:59pm
  - You get to write some buffer overflow exploits!

- Unit Summary 2 Due next Wednesday (8/5) – 11:59pm
Roadmap

C:

```c
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();
```

Assembly language:

```
get_mpg:
  pushq %rbp
  movq %rsp, %rbp
  ...
  popq %rbp
  ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
11000001111110100100011111
```

OS:

- Windows 10
- OS X Yosemite

Computer system:

- CPU
- RAM
- SSD

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Aside: Units and Prefixes

- Here focusing on large numbers (exponents > 0)
- Note that $10^3 \approx 2^{10}$
- SI prefixes are *ambiguous* if base 10 or 2
- IEC prefixes are *unambiguously* base 2

### SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

<table>
<thead>
<tr>
<th>SI Size</th>
<th>Prefix</th>
<th>Symbol</th>
<th>IEC Size</th>
<th>Prefix</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^3$</td>
<td>Kilo-</td>
<td>K</td>
<td>$2^{10}$</td>
<td>Kibi-</td>
<td>Ki</td>
</tr>
<tr>
<td>$10^6$</td>
<td>Mega-</td>
<td>M</td>
<td>$2^{20}$</td>
<td>Mebi-</td>
<td>Mi</td>
</tr>
<tr>
<td>$10^9$</td>
<td>Giga-</td>
<td>G</td>
<td>$2^{30}$</td>
<td>Gibi-</td>
<td>Gi</td>
</tr>
<tr>
<td>$10^{12}$</td>
<td>Tera-</td>
<td>T</td>
<td>$2^{40}$</td>
<td>Tebi-</td>
<td>Ti</td>
</tr>
<tr>
<td>$10^{15}$</td>
<td>Peta-</td>
<td>P</td>
<td>$2^{50}$</td>
<td>Pebi-</td>
<td>Pi</td>
</tr>
<tr>
<td>$10^{18}$</td>
<td>Exa-</td>
<td>E</td>
<td>$2^{60}$</td>
<td>Exbi-</td>
<td>Ei</td>
</tr>
<tr>
<td>$10^{21}$</td>
<td>Zetta-</td>
<td>Z</td>
<td>$2^{70}$</td>
<td>Zebi-</td>
<td>Zi</td>
</tr>
<tr>
<td>$10^{24}$</td>
<td>Yotta-</td>
<td>Y</td>
<td>$2^{80}$</td>
<td>Yobi-</td>
<td>Yi</td>
</tr>
</tbody>
</table>
How to Remember?

- Will be given to you on Final reference sheet

- Mnemonics
  - There unfortunately isn’t one well-accepted mnemonic
    - But that shouldn’t stop you from trying to come with one!
  - **Killer Mechanical Giraffe Teaches Pet, Extinct Zebra to Yodel**
  - **Kirby Missed Ganondorf Terribly, Potentially Exterminating Zelda and Yoshi**
  - xkcd: **Karl Marx Gave The Proletariat Eleven Zeppelins, Yo**
    - [https://xkcd.com/992/](https://xkcd.com/992/)
  - Post your best on Piazza!
How does execution time grow with SIZE?

```c
int array[SIZE];
int sum = 0;

for (int i = 0; i < 200000; i++) {
    for (int j = 0; j < SIZE; j++) {
        sum += array[j];
    }
}
```

Plot:

- Execution Time vs. SIZE
  - Expect linear relationship
  - Sum array (repeat 200,000 times)
Actual Data

![Graph showing the relationship between cache size and time with a linear trend line.](image)
Making memory accesses fast!

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
- Program optimizations that consider caches
Processor-Memory Gap

“Moore’s Law”

µProc
55%/year
(2X/1.5yr)

Processor-Memory
Performance Gap
(grows 50%/year)

1989 first Intel CPU with cache on chip
1998 Pentium III has two cache levels on chip
Problem: Processor-Memory Bottleneck

Processor performance doubled about every 18 months

Bus latency / bandwidth evolved much slower

Main Memory

Core 2 Duo:
Can process at least 256 Bytes/cycle

Core 2 Duo:
Bandwidth 2 Bytes/cycle
Latency 100-200 cycles (30-60ns)

Problem: lots of waiting on memory

cycle: single machine step (fixed-time)
Problem: Processor-Memory Bottleneck

Processor performance doubled about every 18 months

Bus latency / bandwidth evolved much slower

Core 2 Duo: Can process at least 256 Bytes/cycle

Core 2 Duo: Bandwidth 2 Bytes/cycle
Latency 100-200 cycles (30-60ns)

Solution: caches

cycle: single machine step (fixed-time)
Cache 💰

- **Pronunciation**: “cash”
  - We abbreviate this as “$”

- **English**: A hidden storage space for provisions, weapons, and/or treasures

- **Computer**: Memory with short access time used for the storage of frequently or recently used instructions (i-cache/I$) or data (d-cache/D$)
  - *More generally*: Used to optimize data transfers between any system elements with different characteristics (network interface cache, I/O cache, etc.)
General Cache Mechanics

- Smaller, faster, more expensive memory
- Caches a subset of the blocks

Cache

Data is copied in block-sized transfer units

Memory

- Larger, slower, cheaper memory.
- Viewed as partitioned into “blocks”
General Cache Concepts: Hit

1. Data in block b is needed

2. Data is returned to CPU

Block b is in cache: Hit!
**General Cache Concepts: Miss**

1. **Data in block b is needed**
2. **Block b is fetched from memory**
3. **Block b is stored in cache**
   - Placement policy: determines where b goes
   - Replacement policy: determines which block gets evicted (victim)
4. **Data is returned to CPU**
Why Caches Work

- **Locality**: Programs tend to use data and instructions with addresses near or equal to those they have used recently
Why Caches Work

- **Locality**: Programs tend to use data and instructions with addresses near or equal to those they have used recently.

- **Temporal locality**: Recently referenced items are *likely* to be referenced again in the near future.
Why Caches Work

- **Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently.

- **Temporal locality:**
  - Recently referenced items are *likely* to be referenced again in the near future.

- **Spatial locality:**
  - Items with nearby addresses *tend* to be referenced close together in time.

- How do caches take advantage of this?
Example: Any Locality?

```c
sum = 0;
for (i = 0; i < n; i++)
{
    sum += a[i];
}
return sum;
```

- **Data:**
  - **Temporal:** `sum` referenced in each iteration
  - **Spatial:** consecutive elements of array `a[]` accessed

- **Instructions:**
  - **Temporal:** cycle through loop repeatedly
  - **Spatial:** reference instructions in sequence
Locality Example #1

```c
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];

    return sum;
}
```
Locality Example #1

```c
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
M = 3, N=4
```

```
Access Pattern: stride = ?
```

```
Note: 76 is just one possible starting address of array a
```
Locality Example #2

```c
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];

    return sum;
}
```
Locality Example #2

```c
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];

    return sum;
}
```

M = 3, N=4

Access Pattern:
- stride = ?
  - stride-4
  - stride-N

Layout in Memory
Locality Example #3

What is wrong with this code?

How can it be fixed?

```c
int sum_array_3D(int a[M][N][L])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < L; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];

    return sum;
}
```
Locality Example #3

```c
int sum_array_3D(int a[M][N][L])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < L; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];

    return sum;
}
```

- What is wrong with this code?
- How can it be fixed?

Layout in Memory (M = ?, N = 3, L = 4)

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>k</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>76</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>92</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>108</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>124</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>140</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>156</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>172</td>
</tr>
</tbody>
</table>

Strides: N x L

```
Cache Performance Metrics

- Huge difference between a cache hit and a cache miss
  - Could be 100x speed difference between accessing cache and main memory (measured in *clock cycles*)

- Miss Rate (MR)
  - Fraction of memory references not found in cache (misses / accesses) = 1 - Hit Rate

- Hit Time (HT)
  - Time to deliver a block in the cache to the processor
    - Includes time to determine whether the block is in the cache

- Miss Penalty (MP)
  - Additional time required because of a miss
Cache Performance

- Two things hurt the performance of a cache:
  - Miss rate and miss penalty

- Average Memory Access Time (AMAT): average time to access memory considering both hits and misses
  \[
  \text{AMAT} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}
  \]
  (abbreviated AMAT = HT + MR × MP)

- 99% hit rate twice as good as 97% hit rate!
  - Assume HT of 1 clock cycle and MP of 100 clock cycles
  - 97%: AMAT = 1 + 0.03 × 100 = 4 clock cycles
  - 99%: AMAT = 1 + 0.01 × 100 = 2 clock cycles
Polling Question [Cache I]

- **Processor specs:** 200 ps clock, MP of 50 clock cycles, MR of 0.02 misses/instruction, and HT of 1 clock cycle

  \[
  \text{AMAT} = HT + \text{MR} \times \text{MP} = 1 + 0.02 \times 50 = 2 \text{ cycles}
  \]

  \[
  \text{Total time} = 2 \text{ cycles} \times 200 \text{ ps} = 400 \text{ ps}
  \]

- Which improvement would be best?
  - **A.** 190 ps clock (overclocking, faster CPU)
    \[
    2 \text{ cycles} \times 190 = 380 \text{ ps}
    \]
  - **B.** Miss penalty of 40 clock cycles (change memory)
    \[
    (1 + 0.02 \times 40) \times 200 = 1.8 \times 200 = 360 \text{ ps}
    \]
  - **C.** MR of 0.015 misses/instruction (write better code)
    \[
    (1 + 0.015 \times 50) \times 200 = 1.75 \times 200 = 350 \text{ ps}
    \]
Can we have more than one cache?

- Why would we want to do that?
  - Avoid going to memory!

- Typical performance numbers:
  - Miss Rate
    - L1 MR = 3-10%
    - L2 MR = Quite small (e.g., < 1%), depending on parameters, etc.
  - Hit Time
    - L1 HT = 4 clock cycles
    - L2 HT = 10 clock cycles
  - Miss Penalty
    - P = 50-200 cycles for missing in L2 & going to main memory
    - Trend: increasing!
Summary

Memory Hierarchy

- Successively higher levels contain “most used” data from lower levels
- Exploits temporal and spatial locality
- Caches are intermediate storage levels used to optimize data transfers between any system elements with different characteristics

Cache Performance

- Ideal case: found in cache (hit)
- Bad case: not found in cache (miss), search in next level
- Average Memory Access Time (AMAT) = HT + MR × MP
  - Hurt by Miss Rate and Miss Penalty
Notes Diagrams

Temporal Locality: blocks in cache

Spatial Locality: data accessed
Handout: Any Locality?

```c
sum = 0;
for (i = 0; i < n; i++)
{
    sum += a[i];
}
return sum;
```

- **Data:**
  - **Temporal:**
  - **Spatial:**

- **Instructions:**
  - **Temporal:**
  - **Spatial:**