Incident report on memory leak caused by Cloudflare parser bug

[S]ervers were running past the end of a buffer and returning memory that contained private information such as HTTP cookies, authentication tokens, HTTP POST bodies, and other sensitive data. And some of that data had been cached by search engines.

Many of Cloudflare’s services rely on parsing and modifying HTML pages as they pass through our edge servers. The C code uses, in the classic C manner, pointers to the HTML document being parsed. The root cause of the bug was that reaching the end of a buffer was checked using the equality operator and a pointer was able to step past the end of the buffer. This is known as a buffer overrun.

Administrivia

- Homework 4 due Tuesday (2/28)
- Lab 4 released today
  - Cache runtimes and parameter puzzles
Anatomy of a Cache Question

- Cache questions come in a few flavors:
  1) TIO Address Breakdown
  2) For fixed cache parameters, analyze the performance of the given code/sequence
  3) For fixed cache parameters, find best/worst case scenarios
  4) For given code/sequence, how does changing your cache parameters affect performance?
  5) Average Memory Access Time (AMAT)

\[ \text{AMAT} = HT + MR \times MP \]
The Cache

❖ What are the important cache parameters?
  ▪ Must figure these out from problem description
  ▪ Address size, cache size, block size, associativity, replacement policy
  ▪ Solve for TIO address breakdown, # of sets, management bits

❖ What starts in the cache?
  ▪ Not always specified (best/worst case)
Code: Arrays

- Elements stored contiguously in memory
  - Ideal for spatial locality – if used properly
  - Different arrays not necessarily next to each other

- Remember to account for data size!
  - char is 1 B, int/float is 4 B, long/double is 8 B

- Pay attention to access pattern
  - Touch *all* elements (e.g. shift, sum)
  - Touch *some* elements (e.g. histogram, stride)
  - How many times do we touch each element?
Code: Linked Lists/Structs

- Nodes stored separately in memory
  - Addresses of nodes may be very different
  - Method of linking and ordering of nodes are important

- Remember to account for size/ordering of struct elements

- Pay attention to access pattern
  - Generally must start from “head”
  - How many struct elements are touched?
Access Patterns

- How many hits within a single block once it is loaded into cache?
- Will block still be in cache when you revisit its elements?
- Are there special/edge cases to consider?
  - Usually edge of block boundary or edge of cache size boundary
Cache Example Problem (1/3)

- 1 MiB address space, 125 cycles to go to memory.

Fill in the following table:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Size ((C))</td>
<td>16 KiB = (2^{14}) B</td>
</tr>
<tr>
<td>Block Size ((K))</td>
<td>16 B = (2^4) B</td>
</tr>
<tr>
<td>Associativity ((E))</td>
<td>4-way = (2^2)</td>
</tr>
<tr>
<td>Hit Time ((HT))</td>
<td>3 cycles</td>
</tr>
<tr>
<td>Miss Rate ((MR))</td>
<td>20%</td>
</tr>
<tr>
<td>Write Policy</td>
<td>Write-through</td>
</tr>
<tr>
<td>Replacement Policy</td>
<td>LRU</td>
</tr>
<tr>
<td>Tag Bits</td>
<td>8</td>
</tr>
<tr>
<td>Index Bits</td>
<td>8</td>
</tr>
<tr>
<td>Offset Bits</td>
<td>4</td>
</tr>
<tr>
<td>AMAT</td>
<td>(3 + 0.2 \times 125 = 28)</td>
</tr>
</tbody>
</table>

**AMAT Calculation:**

- \(t = m - s - k\)
- \(s = \log_2(C/K/E)\)
- \(k = \log_2(K)\)

\(m = \log_2(2^{10}) = 20\) bits

\(m = \log_2(2^{10}) = 20\) bits
Cache Example Problem (2/3)

- Assuming the cache starts **cold** (all blocks invalid), calculate the hit rate for the following loop:

  - \( m = 20 \) bits, \( C = 16 \) KiB, \( K = 16 \) B, \( E = 4 \)

```c
#define AR_SIZE 8192  // \( 2^{13} \) int = \( 2^{15} \) B
int int_ar[AR_SIZE];  // &int_ar=0x80000
for (int i = 0; i < AR_SIZE/2; i++) {
    int_ar[i] *= int_ar[i + AR_SIZE/2];
}
```

1. Read \( i \)
2. Write \( i \)

\( i = 0 \):
\[ 0 \times 80 \rightarrow 0 \times 80 \]
\[ 0 \times 4000 \rightarrow 0 \times 4000 \]

Each block holds \( 16 \) B = \( 4 \) ints

\( \text{for each block: } 10 \text{ hits, 2 misses} \)

\[ HR = \frac{10}{10 + 2} = \frac{5}{6} \]
Cache Example Problem (3/3)

- For each of the proposed changes below, write U for “increase”, N for “no change”, or D for “decrease” to indicate the effect on the hit rate for the loop:

  - Direct-mapped cache: __D__
    - i and i+AR/SIZE/2 map into same set! → more conflict misses

  - Increase cache size: __N__
    - Index field(s) increased by 1 bit, but 0x80000 and 0x84000 still in same set

  - Double AR_SIZE: __N__
    - Now i+AR/SIZE/2 → 0x88000, but everything else stays the same

  - Double block size: __U__
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();

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

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Operating Systems
Leading Up to Processes

- System Control Flow
  - Control flow
  - Exceptional control flow
  - Asynchronous exceptions (interrupts)
  - Synchronous exceptions (traps & faults)
Control Flow

- **So far:** we’ve seen how the flow of control changes as a *single program* executes
- **Reality:** multiple programs running *concurrently*
  - How does control flow across the many components of the system?
  - In particular: More programs running than CPUs
- **Exceptional control flow** is basic mechanism used for:
  - Transferring control between *processes* and OS
  - Handling *I/O* and *virtual memory* within the OS
  - Implementing multi-process apps like shells and web servers
  - Implementing concurrency
Control Flow

- Processors do only one thing:
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU’s control flow (or flow of control)

Physical control flow

<startup>
instr$_1$
instr$_2$
instr$_3$
...
instr$_n$
<shutdown>
Altering the Control Flow

- Up to now, two ways to change control flow:
  - Jumps (conditional and unconditional)
  - Call and return
  - Both react to changes in *program state*

- Processor also needs to react to changes in *system state*
  - Unix/Linux user hits “Ctrl-C” at the keyboard
  - User clicks on a different application’s window on the screen
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - System timer expires

- Can jumps and procedure calls achieve this?
  - No – the system needs mechanisms for “*exceptional*” control flow!
Exceptional Control Flow

- Exists at all levels of a computer system

- Low level mechanisms
  - Exceptions
    - Change in processor’s control flow in response to a system event (i.e. change in system state, user-generated interrupt)
    - Implemented using a combination of hardware and OS software

- Higher level mechanisms
  - Process context switch
    - Implemented by OS software and hardware timer
  - Signals
    - Implemented by OS software
    - Covered briefly later
Exceptions

- An *exception* is transfer of control to the operating system (OS) kernel in response to some *event* (*i.e.* change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples: division by 0, page fault, I/O request completes, Ctrl-C

How does the system know where to jump to in the OS?
Exception Table

- A jump table for exceptions (also called *Interrupt Vector Table*)
  - Each type of event has a unique exception number \( k \)
  - \( k = \) index into exception table (a.k.a interrupt vector)
  - Handler \( k \) is called each time exception \( k \) occurs
Exception Table (Excerpt)

<table>
<thead>
<tr>
<th>Exception Number</th>
<th>Description</th>
<th>Exception Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide error</td>
<td>Fault</td>
</tr>
<tr>
<td>13</td>
<td>General protection fault</td>
<td>Fault</td>
</tr>
<tr>
<td>14</td>
<td>Page fault</td>
<td>Fault</td>
</tr>
<tr>
<td>18</td>
<td>Machine check</td>
<td>Abort</td>
</tr>
<tr>
<td>32-255</td>
<td>OS-defined</td>
<td>Interrupt or trap</td>
</tr>
</tbody>
</table>
Leading Up to Processes

- System Control Flow
  - Control flow
  - Exceptional control flow
    - Asynchronous exceptions (interrupts)
    - Synchronous exceptions (traps & faults)
Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor’s interrupt pin(s) (wire into CPU)
  - After interrupt handler runs, the handler returns to “next” instruction

- **Examples:**
  - I/O interrupts
    - Hitting Ctrl-C on the keyboard
    - Clicking a mouse button or tapping a touchscreen
    - Arrival of a packet from a network
    - Arrival of data from a disk
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the OS kernel to take back control from user programs
Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - **Traps**
    - **Intentional**: transfer control to OS to perform some function
    - **Examples**: system calls, breakpoint traps, special instructions
    - Returns control to “next” instruction ("current" instr did what it was supposed to)
  - **Faults**
    - **Unintentional** but possibly recoverable
    - **Examples**: page faults, segment protection faults, integer divide-by-zero exceptions
    - Either re-executes faulting (“current”) instruction or aborts
  - **Aborts**
    - **Unintentional** and unrecoverable
    - **Examples**: parity error, machine check (hardware failure detected)
    - Aborts current program
System Calls

- Each system call has a unique ID number
- Examples for Linux on x86-64:

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read</td>
<td>Read file</td>
</tr>
<tr>
<td>1</td>
<td>write</td>
<td>Write file</td>
</tr>
<tr>
<td>2</td>
<td>open</td>
<td>Open file</td>
</tr>
<tr>
<td>3</td>
<td>close</td>
<td>Close file</td>
</tr>
<tr>
<td>4</td>
<td>stat</td>
<td>Get info about file</td>
</tr>
<tr>
<td>57</td>
<td>fork</td>
<td>Create process</td>
</tr>
<tr>
<td>59</td>
<td>execve</td>
<td>Execute a program</td>
</tr>
<tr>
<td>60</td>
<td>_exit</td>
<td>Terminate process</td>
</tr>
<tr>
<td>62</td>
<td>kill</td>
<td>Send signal to process</td>
</tr>
</tbody>
</table>
Traps Example: Opening File

- **User calls** `open(filename, options)`
- **Calls** `__open` **function**, which invokes system call instruction `syscall`

```
00000000000e5d70 <__open>:
...
e5d79:  b8 02 00 00 00          mov  $0x2,%eax  # open is syscall 2
e5d7e:  0f 05                   syscall  # return value in %rax
e5d80:  48 3d 01 f0 ff ff       cmp  $0xffffffffffffffff001,%rax
...
e5dfa:  c3                      retq
```

- `%rax` contains syscall number
- **Other arguments in** `%rdi, %rsi, %rdx, %r10, %r8, %r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`
Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user’s memory is currently on disk

Page fault handler must load page into physical memory

Returns to faulting instruction: \texttt{mov} is executed again!
- Successful on second try \checkmark

```c
int a[1000];
int main ()
{
    a[500] = 13;
}
```
Fault Example: Invalid Memory Reference

```c
int a[1000];
int main()
{
    a[5000] = 13;
}
```

```
80483b7: c7 05 60 e3 04 08 0d movl $0xd,0x804e360
```

- Page fault handler detects invalid address
- Sends SIGSEGV signal to user process
- User process exits with “segmentation fault”

Graph:

```
User Process                      OS

movl                        exception: page fault

signal process

handle_page_fault: detect invalid address
```
Summary

Exceptions

- Events that require non-standard control flow
- Generated externally (interrupts) or internally (traps and faults)
- After an exception is handled, one of three things may happen:
  - Re-execute the current instruction
  - Resume execution with the next instruction
  - Abort the process that caused the exception