CS107 Review Session

with Hannah and Ricardo :)

Mini-Disclaimer

- Topics presented here aren't exaustive
- Exam is cumulative, with a focus on post-midterm material
- We'll be focusing on the following this review session:
  - Generics
  - Assembly + Exploits
  - Heap Allocator

Binary Operands

- NOT, AND, OR, XOR, Shifts
- *How do shifts change behavior between signed / unsigned numbers?*
- *Know how to "multiply by -1 in binary!"*
- *Conceptual understanding of Bitwise operators*
  - `|` 1 - "Turn bit on"
  - `&` `0` - "Turn a bit off"
  - OR can be used to take the "union". AND for the "intersection"
  - XOR is useful for flipping certain bits
  - NOT (~) is useful for flipping *all* bits (useful in creating bitmasks)

Other Pre-Midterm Topics (1)

- Signed vs Unsigned
- 2's Complement
- Setting and Extracting Bits
- Strings
  - Null-Terminated
  - Data Segment vs Stack vs Heap
  - Important string.h functions
    - `strlen, str(n)cmp, strspn, strcspn, strdup, strcat`

Other Pre-Midterm Topics (2)

- Double / Triple Pointers
  - *"Why did scandir take a triple-pointer?"*
- What happens to arrays when they're passed into a function
- What goes on the stack vs heap?

Pre-Midterm Clarifications

- *Common Confusion Points*
  - strcmp: Compare two strings character by character
  - *strspn / strcspn*: Get the first occurrence of a character in / not-in a set of characters
  - *strstr:* Substring Search
  - When in doubt (before the final), use the manpage!
- *Pointers vs Arrays*
  - Arrays "Decay to Pointers"
  - *sizeof(pointer) = 8 bytes* (on 64-bit machines)
  - *sizeof(array)* = How many bytes does this array occupy?
  - &array -> Returns a pointer to the first element. Equivalent to just "array"

Generics

- Allows us to operate on arbitrary data types
- Motivating example: How write a sort function for *any* type?
- Implemented in C via operations on *void** and requires a *callback function* to go from *void** parameters to useful (concrete) types
- Many useful functions use generics:
  - *memcpy, memmove, qsort, bsearch*
- Why callback functions? To bypass limitations on *void*s*:
  - No Dereferencing / Can't index into arrays (No size info)
  - Callback Fn's need to know the level of indirection being used.

Assembly (1)

https://web.stanford.edu/class/archive/cs/cs107/cs107.1226/guide/x86-64.html

- Basic Operations:
  - *mov* (read from / write to memory)
  - *callq/retq* (call and return from functions)
- Arithmetic Operations
  - *add, mul, sub, div, xor, shr, shl, and, or*
  - *lea* (Compute an address and store in first operand. Useful for indexing!)
- Jumps
  - *jmp <target>* - Unconditionally jump (use break/continue/if statements)
  - Many other types of jumps (see reference sheet).

Assembly (2)

- Condition Flags:
  - CF = Carry Flag. Set to *ONE* when previous operand leads to carry
    - Used for *unsigned arithmetic*
  - OF = Overflow Flag: Set to ONE if previous operation overflowed
    - Used in *signed arithmetic*
  - ZF = Zero Flag. Set to ONE if previous result was equal to zero
  - SF = Sign Flag. Set to ONE if previous result's MSB is one

Assembly (3) If/Else Structure

```
Test
Jump to ElseBody if test passes
IfBody:
    <statements>
    Jump to EndIf
ElseBody:
    <statements>
EndIf:
    ...
```

Assembly (4): For-Loop Structure

```
Init (ie, int i = 0)
Test:
Check OPPOSITE of Test
Jump to LoopEnd if OPPOSITE passes
<statements>
<Update>
Back to Test
LoopEnd:
    <rest>
```

Assembly (5): Functions

- Know Caller vs Callee registers
- Return value is stored in *%rax*
- Know where arguments are stored
  - Special Case: What happens if more than six arguments are used?
- What happens on a *callq?*
  - Push next instruction's *%rip* onto the stack
  - Change *%rip* to the start of the function
- What happens on a *retq?*
  - Set *%rip* to what's stored on the stack
- How does the stack grow*?*
  - By modifying *%rsp* (via push or by subtracting from *%rsp*)

Heap Allocator

- What is Throughput? Utilization? Fragmentation?
- Difference between **External** and **Internal** fragmentation
  - **External**: When we have enough memory *overall* to satisfy a request, but split across the heap, so we can't give it to the user
  - **Internal:** Giving client more space than they need.

- Be able to create diagrams based on the state of the heap!

Heap Allocator - Design Considerations

- **Implicit**:
  - 8-Byte Header with *size and state* (in use / free)
- **Explicit**
  - 8 Byte Header with *size and state*
  - (In free blocks) Pointers to *previous/next* free blocks
  - Pointers stored in payload area (How does this affect fragmentation?)
- **Explicit**
  - What are benefits of first-fit vs best fit?
  - What are the benefits of address-order vs size order in free list?

Heap Allocator - Design Considerations (2)

- **Implicit** (Non-Coalescing) vs **Explicit** (Coalescing)
  - Which one do you *expect* to have better throughput
  - Which one has better utilization if there are *many small requests?*
  - How does *coalescing affect utilization?*

- Note: You can make a coalescing implicit allocator and a non-coalescing explicit allocator! We just required this design for the project.

Optimization

- Constant Folding
  - *int a = 10 * sizeof(char*) * 5000;* // can be computed by compiler!
- Common Subexpression Elimination
  - Expression calculated many times throughout code, but only computed once!
- Strength Reduction
  - *shift instead of multiplication*
- Dead Code Elimination
  - Remove code if it's never executed
- Code Motion
  - Rearrange code for better performance
- Loop Unrolling
  - Remove overhead of loop by copy+pasting loop body!

Ethics

- lecture 5 slide 24+: buffer overflows, vulnerabilities, and ACM code of ethics
- lecture 8 slide 6+: use-after-free, disclosure, and partiality
  - responsible disclosure: privately alert the software maker to the vulnerability to fix it in a reasonable amount of time before publicizing it
  - degrees of partiality: partiality, partial cosmopolitanism, universal care, impartial benevolence
- lecture 14 slide 23+: privacy and trust
  - privacy as control of information, autonomy, social good, trust

Ethics Example Questions

- Your colleague discovers a security vulnerability in a piece of software. They consider reporting it widely to news organizations to get the word out. Using your understanding of disclosure processes, give one reason why this might not be the best course of action, and explain one reason why your friend might think it is the best course of action.
- On assign2, you wrote a manual page for the scan_token function. Explain why good documentation is one technique to avoid vulnerabilities.
- In lecture, we learned about how the US Federal Government is one of the largest stockpilers and purchasers of 0-day vulnerabilities. Referencing the 4 degrees of partiality covered in lecture, explain one ethical argument in support of this behavior, and one ethical argument against this behavior.

Student Questions!

Jump Condition Codes / Test vs Cmp

- Jumps are mostly based off of the *Flags* that are set
- All jumps in the x86 reference sheet describe *when a jump is taken*
  - Example: *je* - Jump when ZF = 1, jle = Jump when *ZF = 1 OR SF != OF*
- The **cmp** instruction takes two arguments and subtracts them. Then, flags are updated
  - *`cmp a, b` = b - a, set flags*
- The **test** instruction is similar to cmp, but *ands* two arguments and updates flags:
  - *`test, a, b` = b & a, set flags*

Mov vs Lea

- The *mov* instruction copies bytes from *one place to another*, and has several *addressing modes*
  - *(%rax)* - Dereference the address stored in *rax*
  - *4(%rax)* - Dereference the address stored in *(rax + 4)*
  - *(%rax, %rdx)* - Dereference the address specified by "%rax + %rdx"
  - 8(%rax, %rcx, 2) - Derefrence the address: "8 + (2 * %rcx + %rax)
- General Operandform:
  - Imm(*rb, ri, s) = Imm + R[rb] + R[ri] * s*

Lea

- Unlike *mov*, which copies the data *at* the address *src* to *dst,* lea copies the *value of src* to the destination
- For example:
  - `mov (%rax, %rcx), (%rdx)` → `*%rdx = *(%rax + %rcx)`
  - `lea (%rax, %rcx), %rdx` → `%rdx = (%rax + %rcx)`
- Notice how lea doesn't dereference!

Reverse-Engineering!

```
int mystery(const char* input)
{
    int result = 0;
    /* rest of code */
}
```

Solution:

https://godbolt.org/z/b43GoahYW

```
1  mystery:
2    pushq %rbp
3    movq %rsp, %rbp
4    subq $32, %rsp
5    movq %rdi, -24(%rbp)
6    movl $0, -4(%rbp)
7    jmp .L2
8  .L3:
9    addq $1, -24(%rbp)
10   movq -24(%rbp), %rax
11   movl $47, %esi          # 47 = '/'
12   movq %rax, %rdi
13   call strchr
14   movq %rax, -24(%rbp)
15   addl $1, -4(%rbp)
16 .L2:
17   cmpq $0, -24(%rbp)
18   jne .L3
19   movl -4(%rbp), %eax
20   ret
21
```