Arrays

- Elements packed into contiguous region of memory
- Use index arithmetic to locate individual elements
- 2D array: address of
  \[ A[i][j] = A + i \times (C \times L) + j \times L \]

Two Methods to save arrays (and data in general)

- Fixed memory
- Stack memory - stack frame

- Spill to stack

- Floating point data in x86-64 - optional
Today’s Menu

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, move operation
  - Memory addressing modes
- Operation leaq and Arithmetic & logical operations
- Conditional Statement – Condition Code + cmovX
- Loops
- Function call – Stack – Recursion
  - Overview of Function Call
  - Memory Layout and Stack - x86-64 instructions and registers
  - Passing control
  - Passing data – Calling Conventions
  - Managing local data
  - Recursion
- Array
- Floating-point operations
- x86-64 - Buffer Overflow
C and stack ... so far

- C does not perform any bound checks on arrays
- Local variables
- **Callee** and **caller** saved registers stored on the stack
- Return addresses

- As we saw in Lab 2 and Lab 4, this may lead to trouble
The trouble -> buffer overflow

- If function does not perform “bound- check” when writing to a local array ...
  
  bound-check:
  while input size < array size
  write to array

  ... it may write more data that the space allocated to array can hold, hence overflowing the array -> buffer overflow

- Effect: the function may write over (corrupt) data kept on the stack:
  
  - Value of other local variables and registers
  - Return address

- Stack smashing
Demo the trouble -> buffer overflow

- See **Password Demo** files posted on our course web site on W March 4 - Lecture 21
Why is buffer overflow a problem

- Corrupted data
- Corrupted return value
  - Which may lead to segmentation fault
  - Which also makes a system vulnerable to attacks
- How?
An “attacker” could overflow the buffer:

- By inputting a string that contains byte representation of *malicious* executable code (exploit code) instead of legitimate characters
- String is written to array on stack and overwrites return address with a “return address” that points to exploit code
- When `func2` executes `ret` instruction, it pops this erroneous address onto PC and jumps to exploit code
- CPU starts executing the *malicious* code at this location
How to protection against such attack

1. Avoid overflow vulnerabilities by always checking bounds
   - For example, use library routines that limit string lengths
     - “Unsafe”: `gets()`, `strcpy()`, `strcat()`, `sprintf()`, ...
       - These functions can generate a byte sequence without being given any indication of the size of the destination buffer
     - “Safe”: `fgets()`
Suggestion from developer.apple.com

```c
char destination[5];
char * source = "LARGER";

strcpy(destination, source);

strncpy(destination, source, sizeof(destination));

strlcpy(destination, source, sizeof(destination));
```
2. Employ system-level protections

-> Randomized initial stack pointer

- At start of program, system allocates random amount of space on stack
- Effect: Shifts stack addresses for entire program
- Hence, makes it difficult for hackers to predict start of stack frames
How to protection against such attack

2. Employ system-level protections
   -> Non-executable code segments
     - In the old days of x86, memory segments marked as either “read-only” or “writeable”
     - Could execute anything readable
     - x86-64 has added an explicit “execute” permission
     - Stack segment now marked as non-executable

Any attempt to execute this code will fail
How to protection against such attack

3. Compiler (like gcc) uses a stack “canary” value
   - History: Starting early 1900’s, canaries used in the coal mines to detect gas leaks
   - Push a randomized canary value at end of an array on stack
   - Before executing a ret instruction, canary value is checked to see if it has been corrupted
     - If so, failure reported

```c
main:  # main.c from our Lab 4
    pushq %rbx
    ...
    subq $48, %rsp
    movq %fs:40, %rax
    movq %rax, 40(%rsp)
    ...
    leaq 16(%rsp), %rbx
    ...
    movq 40(%rsp), %rax
    xorq %fs:40, %rax
    jne .L5
    addq $48, %rsp
    popq %rbx
    ret

.L5:
    call __stack_chk_fail@PLT
```
Summary

- What is a buffer overflow
  - When function writes more data than the space allocated to array can hold
  - Effect: data kept on the stack (value of other local variables and registers, return address) may be corrupted
  - \(\rightarrow\) Stack smashing

- Why buffer overflow spells trouble -> it creates vulnerability
  - Hacker attacks

- How to protect system against such attacks
  - Avoid overflow vulnerabilities by always checking array bounds
    - Use “safe” library routines
  - Employ system-level protections
    - Randomized initial stack pointer
    - Non-executable code segments
  - Use compiler (like gcc) that makes use of stack “canary” value
Next Lecture

- Instruction Set Architecture (ISA)
  - Definition of ISA
  - ISA design
  - ISA evaluation
    - Strategies to improve our ISA
- Execution of machine instructions
  - Intro to logic design + Sequential logic circuit
  - Sequential execution of machine instructions
  - Pipelined execution of machine instructions
    - Hazards