15-213 Recitation: C Review

TA's
1 Oct 2018
Agenda

- Logistics
- Attack Lab Conclusion
- C Assessment
- C Programming Style
- C Exercise
- Cache Lab Overview
- Appendix:
  - Valgrind
  - Clang / LLVM
  - Cache Structure
Logistics

- Attack Lab is due **tomorrow at midnight**!
  - Come to office hours for help
  - r-target phase 3 is only worth 5 points
    - 0.2% of your grade $\approx 0\%$ of your grade

- Cache Lab will be released shortly thereafter!
Attack Lab Conclusion

- Don’t use functions vulnerable to buffer overflow (like `gets`)
  - Use functions that allow you to specify buffer lengths:
    - `fgets` instead of `gets`
    - `strncpy` instead of `strcpy`
    - `strncat` instead of `strcat`
    - `snprintf` instead of `sprint`
  - Use `sscanf` and `fscanf` with input lengths (%213s)

- Stack protection makes buffer overflow very hard…
  - But very hard ≠ impossible!
C Assessment

- 3.5 Basic C Programming Questions

- Take some time to write down your answer for each question
C Assessment: Question 1

Which lines have a problem and how can you fix it?

```c
int main(int argc, char** argv) {
    int *a = (int*) malloc(213 * sizeof(int));
    for (int i=0; i<213; i++) {
        if (a[i] == 0) a[i] = i;
        else a[i] = -i;
    }
    return 0;
}
```
C Assessment: Question 1

- malloc can fail!

```c
int main(int argc, char** argv) {
    int *a = (int*) malloc(213 * sizeof(int));
    if (a == NULL) return 0;
    for (int i=0; i<213; i++) {
        if (a[i] == 0) a[i]=i;
        else a[i]=-i;
    }
    return 0;
}
```
C Assessment: Question 1

- Allocated memory is not initialized!

```c
int main(int argc, char** argv) {
    int *a = (int*)calloc(213, sizeof(int));
    if (a == NULL) return 0;
    for (int i=0; i<213; i++) {
        if (a[i] == 0) a[i]=i;
        else a[i]=-i;
    }
    return 0;
}
```
C Assessment: Question 1

- Declaring variables inside a for loop requires `-std=c99`

```c
1 int main(int argc, char** argv) {
2     int *a = (int*) calloc(213, sizeof(int));
3     if (a == NULL) return 0;
4     for (int i=0; i<213; i++) {
5         if (a[i] == 0) a[i]=i;
6         else a[i]=-i;
7     }
8     return 0;
9 }
```
C Assessment: Question 1

- All allocated memory must be freed!

```c
int main(int argc, char** argv) {
    int *a = (int*) calloc(213, sizeof(int));
    if (a == NULL) return 0;
    for (int i=0; i<213; i++) {
        if (a[i] == 0) a[i]=i;
        else a[i]=-i;
    }
    free(a);
    return 0;
}
```
What are the values of A and B?

```c
#define SUM(x, y) x + y

int sum(int x, int y) {
    return x + y;
}

int A = SUM(2, 1) * 3;
int B = sum(2, 1) * 3;
```
What is wrong with our macro `SUM`?

```c
#define SUM(x, y) x + y

int sum(int x, int y) {
    return x + y;
}
```

```c
int A = SUM(2, 1) * 3;       // A = 2 + 1 * 3 = 5!?  
int B = sum(2, 1) * 3;       // B = 6
```
C Assessment: Question 2

- Use parenthesis around result!

```c
#define SUM(x, y) (x + y)

int sum(int x, int y) {
    return x + y;
}

int A = SUM(2, 1) * 3;    // A = 6
int B = sum(2, 1) * 3;    // B = 6
```
C Assessment: Question 2 Part B

- What are the values of A and B?

```c
#define MULT(x, y) (x * y)

int mult(int x, int y) {
    return x * y;
}

int A = MULT(2, 0 + 1) * 3;
int B = mult(2, 0 + 1) * 3;
```
What is wrong with our macro MULT?

```c
#define MULT(x, y) (x * y)

int mult(int x, int y) {
    return x * y;
}

int A = MULT(2, 0 + 1) * 3;  // A = (2 * 0 + 1) * 3 = 3?!
int B = mult(2, 0 + 1) * 3;   // B = 6
```
C Assessment: Question 2 Part B

- Use parenthesis around macro arguments (and result)!

```
#define MULT(x, y) ((x) * (y))

int mult(int x, int y) {
    return x * y;
}

int A = MULT(2, 0 + 1) * 3; // A = ((2) * (0 + 1)) * 3 = 6
int B = mult(2, 0 + 1) * 3; // B = 6
```
C Assessment: Question 2

• Macros are good for compile-time decisions
  • Assert, requires, etc
  • dbg_print

• Macros are not functions and should not be used interchangeably
C Assessment: Question 3

- What lines make `safe_int_malloc` not so safe?

```c
1 int *safe_int_malloc(int *pointer) {
2    pointer = malloc(sizeof(int));
3    if (pointer == NULL) exit(-1);
4    return &pointer;
5 }
```
C Assessment: Question 3

- pointer is a local copy of the pointer!

```c
1 int *safe_int_malloc(int **pointer) {
2     *pointer = malloc(sizeof(int));
3     if (pointer == NULL) exit(-1);
4     return &pointer;
5 }
```
C Assessment: Question 3

- &pointer is a location on the stack in safe_int_malloc’s frame!

```c
int **safe_int_malloc(int **pointer) {
    *pointer = malloc(sizeof(int));
    if (pointer == NULL) exit(-1);
    return pointer;
}
```
C Assessment Conclusion

- Did you answer every question correctly? If not…
  - Refer the C Bootcamp slides

- Was the test so easy you were bored? If not…
  - Refer the C Bootcamp slides

- When in doubt…
  - Refer the C Bootcamp slides

- This will be very important for the rest of this class, so make sure you are comfortable with the material covered or come to the C Bootcamp!
C Programming Style

- Document your code with comments
- Check error and failure conditions
- Write modular code
- Use consistent formatting
- Avoid memory and file descriptor leaks

Warning: *Dr. Evil* has returned to grade style on Cache Lab! 😊
- Refer to full 213 Style Guide: [http://cs.cmu.edu/~213/codeStyle.html](http://cs.cmu.edu/~213/codeStyle.html)
C Exercise

- Learn to use getopt
  - Extremely useful for Cache Lab
  - Processes command line arguments

- Let’s write a Pythagorean Triples Solver!
  - Pair up!
  - Login to a shark machine
  - $ wget http://cs.cmu.edu/~213/recitations/rec6.tar
  - $ tar xvf rec6.tar
  - $ cd rec6

- But first, a simple getopt example...
  - $ vim getopt-example.c
C Exercise: \$ man 3 getopt

- `int getopt(int argc, char * const argv[], const char *optstring);`

- `getopt` returns -1 when done parsing

- `optstring` is string with command line arguments
  - Characters followed by colon require arguments
    - Find argument text in char *optarg
  - `getopt` can't find argument or finds illegal argument sets `optarg` to "?"
  - Example: “abc:d:”
    - a and b are boolean arguments (not followed by text)
    - c and d are followed by text (found in char *optarg)
C Exercise: C Hints and Math Reminders

- \( a^2 + b^2 = c^2 \)
  - \( a = \sqrt{c^2 - b^2} \)
  - \( b = \sqrt{c^2 - a^2} \)
  - \( c = \sqrt{a^2 + b^2} \)
  - \( 3^2 + 4^2 = 5^2 \)

- String to float in C:
  ```c
  #include <stdlib.h>
  float atof(const char *str);
  ```

- Square root in C:
  ```c
  #include <math.h>
  float sqrt(float x);
  ```
Cache Lab Overview

- Programs exhibiting locality run *a lot* faster!
  - Temporal Locality – same item referenced again
  - Spatial Locality – nearby items referenced again

- Cache Lab’s Goal:
  - Understand how L1, L2, … etc. caches work
  - Optimize memory dependent code to minimize cache misses and evictions
    - Noticeable increase in speed

- The use of git is required (affects your style grade)
  - Commit regularly with meaningful commit messages
If you get stuck…

- Reread the writeup
- Look at CS:APP Chapter 6
- Review lecture notes (http://cs.cmu.edu/~213)
- Come to Office Hours (Sunday to Thursday, 5-9pm WH-5207)
- Post private question on Piazza
- man malloc, man valgrind, man gdb
Cache Lab Tips!

- Review cache and memory lectures
  - Ask if you don’t understand something

- Start early, this can be a challenging lab!

- Don’t get discouraged!
  - If you try something that doesn't work, take a well deserved break, and then try again

- Finally, Good luck on Cache Lab!
Appendix

- Valgrind
- Clang / LLVM
- Cache Structure
Appendix: Valgrind

- Tool used for debugging memory use
  - Finds many potential memory leaks and double frees
  - Shows heap usage over time
  - Detects invalid memory reads and writes
  - To learn more… `man valgrind`

- Finding memory leaks
  - `$ valgrind -leak-resolution=high -leak-check=full -show-reachable=yes -track-fds=yes ./myProgram arg1 arg2`
Appendix: Clang / LLVM

- Clang is a (gcc equivalent) C compiler
  - Support for code analyses and transformation
  - Compiler will check you variable usage and declarations
  - Compiler will create code recording all memory accesses to a file
  - Useful for Cache Lab Part B (Matrix Transpose)
Appendix: Cache Structure

\[ S = 2^s \text{ sets} \]

\[ E = 2^e \text{ lines per set} \]

Address of word:
- \( t \) bits: tag
- \( s \) bits: set index
- \( b \) bits: block offset

\[ B = 2^b \text{ bytes per cache block (the data)} \]

data begins at this offset