Computer Science 50 Week 5 Wednesday: October 22, 2008
Fall 2008 Andrew Sellergren, Anjuli Kannan
Scribe Notes

Contents

Computer Science 50
Fall 2008
Scribe Notes

Week 5 Wednesday: October 22, 2008
Andrew Sellergren, Anjuli Kannan

1 Announcements (0:00–5:00)

- Mark Wahlberg talks to animals.[1]
- CS 50 will do its best to remain apolitical during these heated times.[2]
- Quiz next Wednesday! Don't come to Sanders, though—check here to see what room you've been assigned to based on your last name.
- From the course website: "A course-wide review session for Quiz 0 will take place this Friday (10/24) from 1:00 P.M. until 2:30 P.M. in Northwest Science B101; it will be filmed and made available online by Sunday (10/26). Sections on Sunday (10/26), Monday (10/27), and Tuesday (10/28) will offer additional review. Note that last year's quizzes are available along with sample solutions.
- No lunch with David this week.[3]
- Use the Bulletin Board! Your question might already be answered, or, if not, it *will* be answered by a member of the staff or perhaps even a fellow student.

2 `ncurses` (5:00–13:00)

- What is it? It's an *application programming interface* (API) for C which allows us to add graphics to our programs. You'll be using this for Problem Set 4 so that you don't have to reinvent the wheel in order to display colors, move the cursor left and right, etc.
- To start using the `ncurses` library, you run the function `initscr()`. A call to `start_color()` enables your terminal to support color.
- To initialize the colors for your terminal window, call `init_pair()` and provide it with the pair number as well as a background color and a foreground color.
- `ncurses` gives you the ability to split up your window into "screens" of varying sizes. For Problem Set 4, we'll use just one: `stdscr`, which covers the whole window. We'll pass this to the macro `getmaxyx()`. Don't worry about the difference between a function and a macro just yet.
- The short program that David wrote intends to place a `char` in every single space on a 24x80 window. The trick is in the lines of code where `attroff()` and `attron()` are called depending on whether the boolean variable `on` is true or false.

[1] No, srsly this time.
[2] Go Perot!
[3] ☹

```
initscr();
start_color();
init_pair(1, COLOR_BLACK, COLOR_RED);

int y, x;
getmaxyx(stdscr, y, x);
bool on = false;
bool on = false;
for (int i = 0; i < x; i++)
{
    for (int j = 0; j < y; j++)
    {
        mvaddch(j, i, ' ');
        if (on)
            attroff(COLOR_PAIR(1));
        else
            attron(COLOR_PAIR(1));
        on = !on;
    }
}
refresh();
while(1);
```

- Note that none of the changes we make to the screen will actually show until we call the function `refresh()`;
- What happens when we try to compile though? We get a lot of `undefined reference` compiler errors. Well, we forgot to `#include` and link the `ncurses` library. Once we do that, the program compiles and runs, showing a pattern of red!

3 CS 50's Library (13:00–32:00)

- Recall that the secret to `GetChar()` and `GetInt()` was a call to `sscanf()` which allowed us to take user input and also perform some error checking.
- Why do we pass `&x` in `scanf1.c`? Because we want to put a value into `x`, not get one from it. We're giving `scanf()` the memory address of `x` so that it can alter what's stored there:

```
int
main(int argc, char *argv[])
{
    int x;
    printf("Number please: ");
    scanf("%d", &x);
    printf("Thanks for the %d!\n", x);
}
```

- So what's different (and more dangerous) about `scanf2.c`?

  ```
  int
  main(int argc, char *argv[])
  {
      char *buffer;
      printf("String please: ");
      scanf("%s", buffer);
      printf("Thanks for the \"%s\"!\n", buffer);
  }
  ```

  We're asking the Operating System for a 32-bit storage space for a memory address when we declare `char *buffer`. However we're trying to store a string of indeterminate length in that 32-bit storage space!

- If we take a look at this program during runtime using GDB, we can see that the contents of `buffer` are a memory address which can be written compactly as a hexadecimal number. What's being stored at that memory address? Just some garbage (or so it seems).

- Even if we try to overwrite the garbage, we are touching memory that we don't "own" and we'll most likely get a segmentation fault, which will cause a `core` file to be dumped.

- Using GDB, we can examine the `core` file that was dumped. Run `gdb scanf2 core`.

- Turns out the program failed while in the function `vfscanf()`. How can we debug this if we didn't write its code ourselves?

- If we type `backtrace`, we can get a look at the stack just before the program seg faulted. If we're lucky, this will point us to a line number that the program reached before quitting.

- We can also use the `frame` command to peel off the layers of the stack and poke around. For example, we can print out the values of key variables at the time the program crashed.

- How do we fix `scanf2.c`? Take a look at `scanf3.c`:

  ```
  int
  main(int argc, char *argv[])
  {
      char buffer[16];
      printf("String please: ");
  ```

```
        scanf("%s", buffer);
        printf("Thanks for the \"%s\"!\n", buffer);
    }
```

Well, this actually doesn't really fix the underlying problem. Now, we can store a few more characters, but the program will still crash if we overrun `buffer` by giving it a long string as input.

- Note that in this program, there's no obvious way to check the number of characters in a user's input. In `GetInt()`, we dealt with this by using the formatting placeholders `%d` and `%c` to ensure that we would read in 40 bits, maximally (32-bit `int` plus 8-bit `char`).
- On the other hand, `GetString()` reads input one `char` at a time. If we run out of space, we have to ask for more memory, copy the current string into a new, bigger amount of memory, and then free the old string. This is a pain!

4 Dangerous Functions (32:00–40:00)

- Although dynamically growing memory is a pain, it's at least safer that not growing it at all. Consider these dangerous functions:
  - `gets`
  - `scanf`
  - `strcpy`
  - `strcat`
  - `printf`
  - `fprintf`
- If we look at the `man` page for `gets()`, we see that it reads a line from `stdin` until a terminating newline or `EOF` is received. However, the page notes that "no check for buffer overrun is performed." This is a huge bug! In fact, `gets()` will keep reading in characters indefinitely if the caller isn't careful![4]
- What about `strcpy()`? It's biggest problem is that it doesn't check to make sure the destination string has enough memory to store the source string which it will copy into it. The good news is that there's another

[4]David made a mistake here regarding `fscanf()` so it won't be on the quiz! Woohoo! It went like this:

1. David misspoke during lecture.
2. ????
3. PROFIT!

function called `strncpy()` which *does* check lengths. You should always use `strncpy()` rather than `strcpy()`! `strncpy()` takes as a third argument a `size_t`, which is simply a generalized `typedef` (see Monday's lecture), referring to an `int`.

- So what's the big deal? David has a Linux server under his desk.[5] No, that's not the big deal—rather, it's the fact that on any given night, the server gets hit hundreds of times by scripts trying to guess passwords and log in. Threats abound!

5 Safe Code (40:00–61:00)

- Here's a good example:

```
int
main(int argc, char *argv[])
{
    int i;
    for (i = 1; i < argc; i++) {
        printf("%s "‘, argv[i]);
    }
    printf("\n");
}
```

What does it do? It prints out the command-line arguments it was given.

- Here's an unsafe version of the same program:

```
void echo_arg(const char s[])
{
    char buf[MAX_BUF_SIZE];
    strcpy(buf, s);
    printf("%s ", buf);
}

int main(int argc, char *argv[])
{
    int i;
    for (i = 1; i < argc; i++) {
        echo_arg(argv[i]);
    }
    printf("\n");
}
```

[5] I wonder what else is under there. Old Chinese food? A discarded computer?

Now, instead of printing out the array of command-line arguments, we pass it to another function as a `const char[]`. This means, remember, that the function can't change the array.

- What makes this unsafe? It only allocates enough memory for an array of size `MAX_BUF_SIZE`, but never checks user input to make sure it is smaller than this.

- Finally, take a look at another unsafe version put together by Mike Smith:

```
void gotcha()
{
    printf("\nGotcha!\n");
}

void echo_arg(const char s[])
{
    char buf[MAX_BUF_SIZE];
    strcpy(buf, s);
    printf("%s ", buf);
}

int main(int argc, char *argv[])
{
    int i;
    for (i = 1; i < argc; i++) {
        echo_arg(argv[i]);
    }
    printf("\n");
}
```

Notice that there is no call to `gotcha()` in the code itself. And yet, if we provide this program with certain malicious input, we can induce it to execute this function!

- What we've been taking for granted up until now is that stack frames quickly and easily get lopped off once the functions they belong to reach their `return` statement. But how does this happen exactly? The memory address of the calling frame is actually written on the stack so that once a function finishes executing, it knows where to return. However, if we manage to unintentionally or intentionally overwrite this return address, the function might return to an entirely different location in memory. We can overwrite this return address by *overrunning a buffer*—that is, storing something too-large for it.

- A Perl script written by Jason Gao will automate the hacking process. Notice that we fill the buffer with the numbers "1234" and then overrun it with a series of hexadecimal numbers which, as it turns out, form the address in memory of the `gotcha()` function. We can see this address by running GDB and executing the command `print gotcha`.

- When we run `hack.pl eco` from the command line, we print "Gotcha!" to `stdout`.
- This is scary, no? Many web servers are written in languages like C and even today are vulnerable to attacks like this. Let it be clear that we don't in any way advocate (or tolerate) this kind of behavior!

6 Structs (61:00–72:00)

- Recall from last time the syntax for declaring a `struct`:

```
// structure representing a student
typedef struct
{
    int id;
    char *name;
    char *house;
}
student;
```

  Note that it's not entirely necessary for us to use `typedef` when we declare a `struct`, but this enables us to write `student` instead of `struct student` every time we want to declare.

- Take a look at `structs1.c`:

```
#include <cs50.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "structs.h"


// class size
#define STUDENTS 3


int
main(int argc, char *argv[])
{
    // declare class
    student class[STUDENTS];
```

```
        // populate class with user's input
        for (int i = 0; i < STUDENTS; i++)
        {
            printf("Student's ID: ");
            class[i].id = GetInt();

            printf("Student's name: ");
            class[i].name = GetString();

            printf("Student's house: ");
            class[i].house = GetString();
            printf("\n");
        }

        // now print anyone in Mather
        for (int i = 0; i < STUDENTS; i++)
            if (strcmp(class[i].house, "Mather") == 0)
                printf("%s is in Mather!\n\n", class[i].name);

        // free memory
        for (int i = 0; i < STUDENTS; i++)
        {
            free(class[i].name);
            free(class[i].house);
        }
    }
```

In this program, we declare an array of `struct`'s, ask the user for input to populate it, and loop over it, checking for any students in Mather so that we can call them out. Notice the syntax whereby we use a period to refer to the inner elements of a `struct`. Also notice that we are finally explicitly `free`'ing memory that we allocated!

- If we look at `structs2.c`, we see another first:

```
#include <cs50.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "structs.h"


// class size
#define STUDENTS 3
```

```
int
main(int argc, char *argv[])
{
    // declare class
    student class[STUDENTS];

    // populate class with user's input
    for (int i = 0; i < STUDENTS; i++)
    {
        printf("Student's ID: ");
        class[i].id = GetInt();

        printf("Student's name: ");
        class[i].name = GetString();

        printf("Student's house: ");
        class[i].house = GetString();
        printf("\n");
    }

    // now print anyone in Mather
    for (int i = 0; i < STUDENTS; i++)
        if (strcmp(class[i].house, "Mather") == 0)
            printf("%s is in Mather!\n\n", class[i].name);

    // let's save these students to disk
    FILE *fp = fopen("database", "w");
    if (fp != NULL)
    {
        for (int i = 0; i < STUDENTS; i++)
        {
            fprintf(fp, "%d\n", class[i].id);
            fprintf(fp, "%s\n", class[i].name);
            fprintf(fp, "%s\n", class[i].house);
        }
        fclose(fp);
    }

    // free memory
    for (int i = 0; i < STUDENTS; i++)
    {
        free(class[i].name);
        free(class[i].house);
    }
}
```

Finally, we're offering the ability to store data persistently (i.e. on disk). The function `fopen()` tries to open a file to write to. It takes as its first argument the filename and as its second argument, the mode to open it in, in this case "w" for write mode.

- If we compile and run `structs2.c`, we seemingly get the same output as with `structs1`; however, this time, we end up with a new file called `database` that stores our data.
- Implicit in this `database` file is its format. Basically, it's how you decided to store the data—the particular pattern that it's written in. Every file format is simply this: a specific output pattern for storing data which can be read by a matching input program.
- For Problem Set 5, you'll be taking advantage of this when we provide you with a formatted camera memory stick. Your job will be to recover the data which has been "erased" by interpreting the specific patterns which appear!