Computer Science 50
Fall 2009
Scribe Notes

Week 5 Monday: October 5, 2009
Andrew Sellergren

Contents

Computer Science 50
Fall 2009
Scribe Notes

Week 5 Monday: October 5, 2009
Andrew Sellergren

1 Geek Humor (0:00–10:00)

- For the Hacker Edition of Problem Set 2, students were tasked with deciphering the password hashes stored in `/etc/passwd/` on a Linux system. The `crypt()` function is generally used to create a series of random characters from a given password using a one-way algorithm. We say it is one-way because it's very hard to derive the plaintext from the encrypted string. This is why Linux administrators may be able to change your password if you have forgotten it, but they can't tell you what it is.

- For the curious, here are the decrypted passwords along with a little explanation of the attempted humor behind them:

  - `julius`: 13
  - `skroob`: 12345
  - `wbrandes`: voice
  - `baravelli`: swordfish
  - `blaise`: FOOBAR
  - `gcostanza`: Bosco
  - `malan`: ftw!!!111

  `julius` was Caesar's username with 13 corresponding to the ROT-13 algorithm. `skroob` was from *Spaceballs*, `wbrandes` from *Sneakers*, `baravelli` from an old Marx Brothers' film, `blaise` being Vigenère himself, `gcostanza` from *Seinfeld*. And `malan` FTW!

- Check out Pointer Fun for a fun (but also educational) tutorial on pointers.

2 Announcements (10:00–19:00)

- This is CS 50.

- Final projects aren't far away, so it would behoove you to being tossing around ideas in that old clunker of a brain of yours. Here's an interesting one brought to life by Phil, a teaching fellow from last year, using Flare, a visualization library for Adobe Flash. What it does is show the different pathways it is possible to take among computer science classes at Harvard.

- Our logo for the week is a Japanese character which has at least some relevance to Sudoku. If you're like me and were too daft to notice that we've had a different logo for each week of the course so far, check out the first page of each of the slide handouts.

- If you've already started Problem Set 4, you may have found it difficult to debug using GDB when the `ncurses` graphics library is in use. Our own Glenn Holloway has a solution for this:

   1. Connect to NICE in two separate terminal windows. Make sure you're connected to the same physical server on both. You can type `hostname` to find out which server you're on. If it's not the same on both, then logout on one window and login directly to that hostname, say `ice2`, by typing `ssh ice2.fas.harvard.edu`.
   2. On one terminal window, run `sudoku`, but add an `&` at the end of the command, which will background the process and spit out a `pid`, or process ID number, to identify the running program. You can see that it's running by typing `ps` to list all active processes on the machine. You should see `sudoku` along with `ps` and `tcsh`, which is the actual shell program itself.
   3. In your other terminal window, type `gdb sudoku <pid>`, where `pid` is the process ID number given to you upon running `sudoku` in the other terminal window. Now we can debug our Sudoku program in one terminal window while executing the program in another.

- Next Friday, 10/16 at 1:15 PM, will be the next Lunch with David et al. RSVP at cs50.net/rsvp.

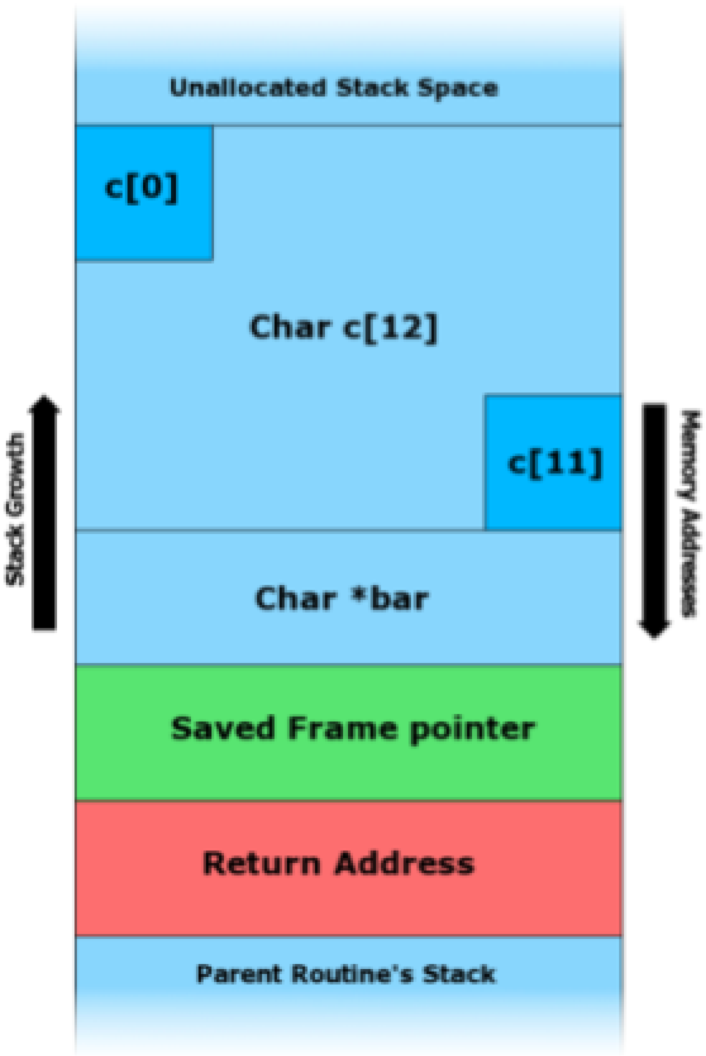
3 Buffer Overrun Attacks (19:00–30:00)

- Generally speaking, exploits are possible because humans aren't perfect, especially with regard to memory management. In languages like C and C++, memory mismanagement usually boils down to misuse of a pointer. One of the most common mistakes that a programmer can make in these languages is to fail to check the bounds of an array which leaves the program vulnerable to a *buffer overrun attack.* Take a look at the buggy code below:

```
#include <string.h>

void foo (char *bar)
{
   char  c[12];

   memcpy(c, bar, strlen(bar));  // no bounds checking...
}

int main (int argc, char **argv)
{
   foo(argv[1]);
}
```

  All this program does is to call a function named `foo`, which copies the first command-line argument into a buffer named `c`. To do so, it uses the `memcpy` function, which takes as its third argument the length of the string to be copied. So we're doing at least one sanity check to make sure we don't copy more characters than the first command-line argument has.

- However, we're not checking that the buffer `c` has enough space to store all of the characters of the first command-line argument. If that command-line argument is longer than 12 characters, then we'll be writing past the bounds of the `c` array and possibly overwriting something else on the stack. We can visualize this like so:[1]



[1] Source: Wikipedia.

As the red rectangle indicates, the stack is used to store not only function frames and parameters, but also a function's return address. Thus, when a function finishes executing, the program will know where in memory to return to in order to continue.

- As you can see from this diagram, if we write more than 12 characters to `c`, we're going to first overwrite the value of `bar` followed by something called the saved frame pointer, which allows the computer to remember where in RAM it currently is. Next, we'll overwrite or corrupt the return address. If an adversary knows where on the stack this return address is, he can overwrite it with a valid but different value which contains some malicious code he wants to execute. Then, instead of returning to the correct memory address, the program will jump to the malicious code instead.

- As a webmaster, if you ever scroll through the request logs, you might find some users are requesting what seems to be junk. Frankly, they're just trolling to see if they can crash your website which might reveal a vulnerability they can exploit.

- Although the diagram above would suggest that an adversary needs to know the exact memory location of his malicious code, the use of a NOP sled actually lowers the bar considerably. This instruction, which stands for "no operation" tells the program to skip over it and look for the next executable instruction. By inserting a whole slew of these followed by a jump to the top of the buffer, an adversary doesn't have to correctly guess the exact memory location of his malicious code, but only has to come close, as the exact location will be reached by the series of no-op instructions followed by the jump.

4 `structs` (30:00–51:00)

- So far we've only talked about primitive data structures, but what if we want to store a chunk of related information about a single entity. For example, a student has a name, a dorm, a phone number, etc. Can we collect all these pieces of information into a single data structure?

- Yes, we can![2] We can do so using the `typedef` command, which we previously used in CS 50's library to alias a string to a `char *`. That data structure is called a `struct`. Take a look at `structs.h` for an example of how to define them:

```
/****************************************************************************
 * structs.h
 *
```

[2] Who was it that used that as a slogan? Was it Bob the Builder? Yeah, I think it was Bob the Builder.

```
 * Computer Science 50
 * David J. Malan
 *
 * Defines a student for structs{1,2}.c.
 ***************************************************************************/


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

Even though this is a fairly straightforward definition, notice that we've abstracted it away into a separate header file so that it might be used by many different programs, including `structs1.c` and `structs2.c`. These will include this header file by writing:

```
#include "structs.h"
```

Here we're using quotation marks instead of angle brackets because the header file is local rather than on the server.

- Before we go over the syntax for defining a `struct`, let's take a look at `structs1.c` to see how to use one:

```
/****************************************************************************
 * structs1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Demonstrates use of structs.
 ***************************************************************************/

#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);
    }
}
```

So we've declared an array that contains three instances of the variable type `student`. For now, know that `student` is simply a special name for a `struct` that we've defined. The rest of the program asks the user for input to populate our `struct`'s, 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. We're doing this even though we didn't call `malloc` directly because we know that CS 50's library functions, e.g. `GetString()`, did call `malloc`. Whenever possible, we want to prevent memory leaks!

- structs1.c might not be all that interesting in terms of its output, but it is interesting in that it's the first example of a more sophisticated data structure which we've defined ourselves. To step it up a notch, let's see what we can do with the data once we've created our custom data structure. Take a look at structs2.c:

```
/****************************************************************************
 * structs.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Demonstrates use of structs.
 ***************************************************************************/

#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 introducing a way of storing data on disk, which will allow to persist beyond the program’s termination. The `fopen` function takes two arguments: the name of the file to be opened (and created if it doesn’t yet exist) and the mode in which that file is to be opened, `‘‘w’’` for write mode, in this case. Notice we’re checking its return value for `NULL`. When might it return `NULL`? If you don’t have privileges to open the file or if there isn’t enough disk space, for example.

- Once we’ve opened the file, how do we write to it? The function `fprintf` is like `printf` in that it takes format specifiers and fills them in with data that you provide, but unlike `printf`, it doesn’t print the output to stdout but rather to a file, designed by a file pointer given as the first argument.

- When we run `structs2`, now, we see that a file called `database` is created which contains our data. Anytime you’ve ever saved a file, no matter the extension, the file was simply created by laying out the data in a predetermined, often proprietary way much as we created it here. Some file formats are actually deceptively simple. JPEGs generally have a header which announce it as a JPEG, then a large chunk containing the actual photo data, and occasionally a footer that follows.

- This rigid structure of the JPEG file format is what will enable us in Problem Set 5 to recover photos which have been "deleted" from a disk. We'll simply search for the header signature which identifies a JPEG and continue reading until we find another header (so we know that another photo is there) or we reach the end of the file.

- One useful feature of GDB that we haven't yet investigated is the ability to examine the stack. If we go back to `bar.c` and add a breakpoint once the `bar` function is called, we can type `backtrace` to view the stack during the program's execution. It will show us something like this:

  ```
  (gdb) backtrace
  #0  bar (m=10) at bar.c:41
  #1 0x08048584 in foo (n=5) at bar.c:34
  #2 0x0804855c in main () at bar.c:24
  ```

  If you do this while executing the `sudoku` program, you might find the stack to have many more frames than just 2. Even if there are multiple functions that you don't recognize, chances are that they aren't to blame for your bugs, given that they belong to the `ncurses` library which has been around for quite some time. The numbers on the left are the frame numbers. If you type `frame 2`, for example, you can actually poke around in `main` while `bar` is in the middle of its execution. Then you can print out variables which are technically out of scope!

- If we print the address of a variable in `main` and compare it to the address of a variable in a function called by `main`, we can see that the function variable has a similar, but lower memory address compared to that of the `main` variable. This coincides with our representation of the stack as growing from bottom to top, but with larger memory addresses at bottom.

5 More with Hexadecimal and File I/O (51:00–58:00)

- Incidentally, I/O simply stands for input/output. A few other functions that will come in handy for manipulating files in C:

  - `fopen/fclose`
  - `fscanf/fprintf`
  - `fread/fwrite`
  - `feof`

  For reading character data from file, use `fscanf`. For reading and writing binary data to and from files, use `fread` and `fwrite`. `feof` will tell you when you've reached the end of a file.

- Hexadecimal digits can conveniently represent 4 bits, e.g. 1111 in binary is equivalent to 0xF and 11111111 is equivalent to 0xFF, the value 255. If you have any background in HTML, you'll see hexadecimal codes to represent colors. Colors are represented as red-green-blue (RGB) triples. Pictures are composed of pixels, each of which has a color that can be created with some combination of red, green, and blue.[3] Red can be represented as 0xFF0000, green as 0x00FF00, blue as 0x0000FF, black as 0x000000, and white as 0xFFFFFF.

- Because every hexadecimal digit represents 4 bits, we can represent 32-bit memory addresses in 8 digits, e.g. 0xBF976C60. There's one complication: memory addresses can be either little-endian or big-endian, depending on whether they are read right-to-left or left-to-right.

6 Data and Forensics (58:00–70:00)

- During one of David's summers as a grad student[4], he interned as a forensic investigator at the Middlesex DA's office. There he spent time recovering data from confiscated hard drives in the hopes of compiling evidence.

- A short while ago, a story broke about the iPhone in which forensic data analysts warned that private data was vulnerable to being stolen because of the way in which Apple implemented the ability to press the home button and quickly exit an app. The way Apple did this was to take a screenshot and then quickly shrink the screenshot. This screenshot, of course, was stored somewhere in the iPhone's flash memory, making it accessible to anyone who knows what they're doing and has physical access to the phone.

- In both cases, data which has been "deleted," hasn't always been permanently removed. If you know what to look for, you can often recover this supposedly inaccessible data.

- Browsers have histories and caches. Computers also offer the ability to have the illusion of more RAM by reserving some hard disk space for virtual memory. If enough programs are running, then some of the data in RAM gets "paged to disk," meaning it gets temporarily written to the hard drive. Mac OS X gives the option under FileVault to encrypt the home directory and also to "Use secure virtual memory," which encrypts the virtual memory swap space itself.

- If you really like this kind of stuff, check out CS 120!

- Hard drives, if they're mechanical and not solid-state, are composed of circular platters which spin. The locations of files on those platters are stored

[3] Go reread your physics textbook, nerd.

[4] In, like, 1960, I think.

in a large table. When you delete a file, say by dragging it to the Recycle Bin and even emptying the Recycle Bin, you're not actually deleting the data itself, but only deleting the entry in the table which contains the file locations. This is actually for performance reasons, since historically the act of actually overwriting the data on disk might've taken quite a while. What would it mean to actually delete the data? Overwriting the data with all zeroes would seem to do the trick.

- Of course, over time, there's an increasing probability that a deleted file's data will simply be overwritten by virtue of having other files added to disk.

- As one of David's colleagues at MIT proved, it's remarkably easy to recover sensitive private data from discarded hard drives. He managed to find all kinds of social security numbers, credit card numbers, and good ol' pr0n simply by buying up a large number of hard drives on eBay. His purpose, of course, which we'll present to you in his paper, was to show that many programs which promise to erase or protect data fall way short of this goal. So be careful!