Week 4

This is CS50. Harvard University. Fall 2014.

Cheng Gong

Table of Contents

Shellshock

- A bug has been recently discovered in a programming language called Bash, with many articles dramatizing its severity and bringing back memories of Heartbleed[1] from last spring.
- We watch Everything you need to know about the Shellshock Bash bug[2]. Even though most of the video footage is irrelevant, it does explain that Bash is not only a programming language but also a command-line shell that allows hackers to have a large amount of control over another system, deleting files or copying them.
- It's familiar to us, because the Terminal we've been using in the CS50 Appliance actually also runs Bash, Bourne Again SHell, which is the blinking prompt that waits for your input, as a command-line interface.
- We've learned to run commands like `cd`, but we can also define our own functions in Bash. This looks fairly similar to defining a function in C, and once we press enter we can run our `hello` function:

```
jharvard@appliance (~): hello() { echo "hello, world"; }
jharvard@appliance (~): hello
```

[1] http://en.wikipedia.org/wiki/Heartbleed
[2] http://youtu.be/hjmx7waVFXI

```
hello, world
jharvard@appliance (~):
```

  # The key here is that we wrote the name of the function and parentheses that signifies it's a function.

- We can do similar things in the Terminal program in Mac OS.

  # As an aside, David once played a prank on a former TF by doing the following:

```
air:~ jharvard$ alias ls="say beep"
```

    # Now every time he ran the `ls` command the Mac would actually say something through its speakers, rather than listing files.

- Turns out that the Shellshock bug has been around for over 20 years. If you have a Mac, you can type the following command into Terminal, and if it outputs `vulnerable`, then your computer is vulnerable:

```
env x='() { :; }; echo vulnerable' bash -c :
```

  # For once, Windows users don't have to worry about this threat.

- In particular, the part underlined is a function that does nothing:

```
env x='() { :; }; echo vulnerable' bash -c :
```

  # We have a variable called `x` that we assign a function, with a `:` that means do nothing and `;` that means stop doing nothing, and it could have anything else inside as well. But moving down the line, we see more:

```
env x='() { :; }; echo vulnerable' bash -c :
```

- So long story short this line of code allows someone to make a computer do something else, even though Bash was supposed to stop reading after the first part in red:

```
env x='() { :; }; echo vulnerable' bash -c :
```

- We used `echo vulnerable` in our example, but what if we did this?

```
env x='() { :; }; rm -rf *' bash -c :
```

  - The `rm` means remove, `-r` means recursive, or going into every subdirectory and deleting the files inside, `-f` means force, or don't prompt the user, and `*` means everything in the current directory.
  - It's the fastest way to clear everything in a directory, so there's some use, but also potential for serious damage.

- With this bug, we can trick a computer into running any other command as well, like emailing someone or creating or deleting files.
- How do we actually do that? Well, there are many Mac and Linux servers on the internet running Bash, and every time you request a page from them with your browser, it actually sends something like the following:

```
GET / HTTP/1.1
Location: www.example.com
User-Agent: () { :; }; rm -rf *' bash -c :
```

  - Notice the first two lines seem straightforward, but the third line, if sent by a bad guy to a server, will cause Bash to execute that command. Usually, `User-Agent` is used to describe the browser the user is running, like `User-Agent: Firefox` or `User-Agent: Chrome` or `User-Agent: Internet Explorer` (though the descriptions are usually more verbose). But by sending that command, bad guys can trick a vulnerable server on the internet into deleting all of its files.
    - In fact, you could start a Distributed Denial of Service[3] attack by making a bunch of servers send as much network traffic as possible to a particular website until it crashes.

- Even though Macs are vulnerable, you're actually safe unless you're running a web (or other type of) server, since your laptop won't be accepting any such messages.
  - Apple will soon release an update[4] that fixes this bug, and the world of Linux has already released a number of fixes as well. (The CS50 Appliance can be fixed with `update50`, but even the appliance is not publicly accessible on the internet by default, so there shouldn't be much to worry about there either.)

[3] http://en.wikipedia.org/wiki/Denial-of-service_attack
[4] http://support.apple.com/kb/dl1769

- Finally, it’s crazy to think that this bug has been out there for over 20 years, and that someone might have known about it. This is one of the fundamental challenges about security: just like in the real world, the good guy is at a disadvantage. We have to make sure every door is locked and every window and every access point is secure, but bad guys only have to find just one unlocked door or unlocked window to enter. In computer code, we might spend hundreds or thousands of hours to get it correct, but if you make just one mistake the entire system (and in this case internet) is put at risk.
  - More info can be found at http://wikipedia.org/wiki/Shellshock_(software_bug)
- "Reflections on trusting trust"[5] by Ken Thompson is a speech given some years ago, asking people if we can really trust the software we’re given. For example, we’ve been compiling programs with `clang`, and the programs we’ve written might not have any backdoors, or a way for bad guys to take over your computer, in its source code. But what if `clang` put some zeroes and ones into every program you compiled, that let whoever wrote it access your computer? We trust that `clang` is legit and that every program you run on your Mac or PC is trustworthy, but even trustworthy programs, like we’ve just seen, might have security issues from unintentional bugs.
- There’s no easy solution to this, but we can be more aware of the complexity of our computer systems, and how vulnerable we might be.

Breakout

- In Problem Set 3, we focus on Breakout, a graphical game where we move a paddle back and forth, bouncing a ball up and breaking bricks.
  - You can run the staff solution in the CS50 Appliance with:

```
jharvard@appliance (~): ~cs50/pset3/breakout
```

- So this brings us back to using a graphical user interface, as well as being given some amount of code. We’ll hand you code that you have to read and understand, with some lines that say `// TODO` for you to fill in.
  - We give you a framework and it might be overwhelming to think of all the functions that exist, but like Scratch you don’t need to know every puzzle piece in order to start using a few.

[5] http://cdn.cs50.net/2014/fall/lectures/4/m/p761-thompson.pdf

  - # We'll introduce you to documentation of new functions to use, but the programming constructs of loops and variables will be identical to the ones we used previously.

- Take advantage of CS50 Discuss[6] and office hours[7] and know that you won't have to write all that much code. The biggest challenge will be acclimating yourself to the code we've given you!

  - # As an aside, the distribution code is also somewhat object-oriented in that it is an approximation of object-oriented programming in a language like C, but it is still procedural and without methods in variables. We'll see more of this later with PHP and JavaScript.

Merge Sort

- Last time we ended on merge sort:

```
On input of n elements
    if n < 2
        return
    else
        sort left half of elements
        sort right half of elements
        merge sorted halves
```

  - # It was much faster and clean to write, with upper bound of #($n \log n$) with a better running time than bubble sort and insertion sort, which had #(n^2). ($\log n$ is smaller than n, so n times something smaller than n is going to be less than n^2.)

- But there was this tradeoff of needing space, where we stored items in a temporary array in order to merge them. Since memory and hard drives are relatively cheap, this isn't necessarily a bad thing.

- Let's look more closely at what we did. We started with this:

```
4    2    6    8    1    3    7    5
```

- And the first step was to look at the left half:

[6] http://cs50.harvard.edu/discuss
[7] http://cs50.harvard.edu/hours

$\underline{4\quad 2\quad 6\quad 8}\quad 1\quad 3\quad 7\quad 5$

- And then the left half of the left half:

$\underline{4\quad 2}\quad 6\quad 8\quad 1\quad 3\quad 7\quad 5$

- With a list of size two, we still need to divide it in half:

$\underline{4}\quad 2\quad 6\quad 8\quad 1\quad 3\quad 7\quad 5$

  # A list of size one is already sorted, so we’re done.

- Now we go back to the right half:

$4\quad \underline{2}\quad 6\quad 8\quad 1\quad 3\quad 7\quad 5$

  # And likewise it’s already sorted.

- Now we’re at the key step of merging the two lists, so we bring the 2 forward, and then the 4:

$4\quad 2\quad 6\quad 8\quad 1\quad 3\quad 7\quad 5$
$\underline{2}$

$4\quad \quad 6\quad 8\quad 1\quad 3\quad 7\quad 5$
$\underline{2\quad 4}$

- We mentally rewind to where we left off before entering the left half of the left half, and remember that we have to sort the right half of the left half:

$4\quad 2\quad 6\quad 8\quad 1\quad 3\quad 7\quad 5$
$2\quad 4\quad \underline{6}$

$4\quad 2\quad 6\quad 8\quad 1\quad 3\quad 7\quad 5$
$2\quad 4\quad 6\quad \underline{8}$

- Time to do the right half, similarly:

$4\quad 2\quad 6\quad 8\quad \underline{1\quad 3\quad 7\quad 5}$
$2\quad 4\quad 6\quad 8$

```
4     2     6     8     1     3     7     5     // left half
2     4     6     8

4     2     6     8     1     3     7     5     // left half of left half
2     4     6     8

4     2     6     8     1     3     7     5     // right half of left half
2     4     6     8

4     2     6     8     1     3     7     5     // merge left half
2     4     6     8     1     3

4     2     6     8     1     3     7     5     // right half
2     4     6     8

4     2     6     8     1     3     7     5     // left half of right half
2     4     6     8

4     2     6     8     1     3     7     5     // right half of right half
2     4     6     8

4     2     6     8     1     3     7     5     // merge right half
2     4     6     8     1     3     5     7
```

- Now we can merge the entire list:

```
4     2     6     8     1     3     7     5
2     4     6     8     1     3     5     7
1     2     3     4     5     6     7     8
```

- Notice that we have 8 elements and we divided by two in every row, and we have 3 rows, because log#8 gives us 3, which just means it takes 3 divisions by 2 to get 1. And for each of those rows, we looked at all *n* elements, even if it was one at a time, for a total of *n* log *n* steps.
- If we wanted to analyze it by looking at the code, we'd start with the beginning:

```
On input of n elements

<u>if n </u>
<
<u> 2</u>
```

```
<u>return</u>

    else
        sort left half of elements
        sort right half of elements
        merge sorted halves
```

- The step underlined has running time #(1), or $T(n)$ = #(1), if $n < 2$. meaning it will take constant time.

- What about this?

```
On input of $n$ elements
    if $n < 2$
        return
    else

<u>sort left half of elements</u>


<u>sort right half of elements</u>


<u>merge sorted halves</u>
```

- Well, sorting the left half would take $T(n/2)$ since we have half as many elements and we can be sort of recursive here in using some function T, and sorting the right half would also take $T(n/2)$. Merging the halves takes n steps, for #(n). So in general $T(n) = T(n/2) + T(n/2) +$ #(n) for $n \geq 2$, and we can again refer to the back of a high school math textbook to simply this recurrence to #($n \log n$).

- What's also interesting here is the cycling, where we defined $T(n)$ in terms of itself, both in the mathematical function and the pseudocode:

```
On input of $n$ elements
    if $n < 2$
        return
    else
        sort left half of elements

<u>sort right half of elements</u>
```

```
        merge sorted halves
```

# The code here is recursive, since it tells itself to solve a smaller version of the problem until we get to this base case.

Sigma

- Let’s take a look at `sigma-0.c`[8]:

[8] http://cdn.cs50.net/2014/fall/lectures/4/m/src4m/sigma-0.c

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

// prototype
int sigma(int);

int main(void)
{
    // ask user for a positive int
    int n;
    do
    {
        printf("Positive integer please: ");
        n = GetInt();
    }
    while (n < 1);

    // compute sum of 1 through n
    int answer = sigma(n);

    // report answer
    printf("%i\n", answer);
}

/**
 * Returns sum of 1 through m; returns 0 if m is not positive.
 */
int sigma(int m)
{
    // avoid risk of infinite loop
    if (m < 1)
    {
        return 0;
    }

    // return sum of 1 through m
    int sum = 0;
    for (int i = 1; i <= m; i++)
    {
        sum += i;
    }
    return sum;
}
```

- The program adds the numbers `1` through `n`, and first we notice that there is a prototype on line 5, and all that does is say that there will be a function later on in the program, named `sigma`, that takes an `int` in the parentheses, and returns an `int`.
- Now let's look at `main`, where we ask the user for an integer until we get a positive one. Then on line 19 we create a variable called `answer`, and store the return value of the `sigma` function to it, after we pass it the `n` from the user.
- Before moving on, let's run it:

```
jharvard@appliance (~/Dropbox/src4m): ./sigma-0
Positive integer please: 2
3
```

  # And 2 + 1 is indeed 3.

- What if we give it `3`? 3 + 2 + 1 = 6.

```
jharvard@appliance (~/Dropbox/src4m): ./sigma-0
Positive integer please: 3
6
```

- And bigger numbers should give us bigger sums:

```
jharvard@appliance (~/Dropbox/src4m): ./sigma-0
Positive integer please: 50
1275
```

- So how does the `sigma` function actually work?

```
int sigma(int m)
{
    // avoid risk of infinite loop
    if (m < 1)
    {
        return 0;
    }

    // return sum of 1 through m
    int sum = 0;
    for (int i = 1; i <= m; i++)
    {
        sum += i;
    }
    return sum;
}
```

  # We have some lines at the beginning that checks `m` is positive, and then a `for` loop that iterates through every number from `1` to `m`, adding them to `sum` as we go along. (Turns out the check is unnecessary, since passing in a negative number to `m` means that the `for` loop will notice that `i # m` isn't true, and not add anything to `sum`, resulting in `0` being returned anyways.)

- Remember that we declare `sum` outside the loop, so that we can access it outside of the `for` loop, and also so that we don't reset it to `0` every pass of the loop.
- Variables are generally scoped to the curly braces that encompass them, so we need to put them outside the curly braces of the `for` loop in order to `return` it after.
- Finally, in `main` we simply call the `sigma` function and print the value it returns. In this case, `sigma` is written with an **iterative** approach where it does the same thing, over and over again.
- But we can implement it differently:

```
int sigma(int m)
{
    if (m <= 0)
        return 0;
    else
        return (m + sigma(m - 1));
}
```

- Here we start by returning `0` if `m # 0`, which is the base case. The beauty is in the `else` condition: the sum of the numbers from `1` to `m` is the same as the sum of `m`, and the sum of the numbers from `1` to `m - 1`. So we can follow this logic, making the question smaller and smaller, from `sigma(n)` to `sigma(n - 1)` to `sigma(n - 2)` until we get to `sigma(0)`, which is added back up to all those other questions.
- Here we have the function call itself, which is dangerous if we delete the base case:

```
int sigma(int m)
{
    return (m + sigma(m - 1));
}
```

- Now we get this:

```
jharvard@appliance (~/Dropbox/src4m): ./sigma-1
Positive integer please: 50
Segmentation fault (core dumped)
```

  # The program didn't crash because the number got too big. The `int` might overflow and have an incorrect value, but is still 32 bits.

  # It actually crashes because the function never stops running, since none of the calls ever finish, but rather keep calling the function again and again.

- Let's look at this picture briefly:

```
-----------------------------
|                           |
|           text            |
|                           |
-----------------------------
|      initialized data     |
-----------------------------
|     uninitialized data    |
-----------------------------
|           heap            |
|             |             |
|             |             |
|             v             |
|                           |
|                           |
|                           |
|             ^             |
|             |             |
|             |             |
|           stack           |
----------------------------
|   environment variables   |
----------------------------
```

- The **stack** is just a chunk of memory used every time a function is called. The operating system takes some amount of bytes and lets you run your function with places for variables and other things you need. If you call another function, and another function, and another function, you get more pieces of memory.
  - Using trays from Annenberg, it's like putting one on a table (an empty stack), and when the tray calls itself, like `sigma` does, it's like putting another tray on top, since we need a little more memory. But the first tray is still there, since it called the function the second time. And if we keep doing this, eventually the stack will exceed the horizontal line above heap (which we'll come back to), and that is a bad thing that means we exceeded our segment of memory, and cause the segmentation fault and our program to crash.
- In general, we won't use recursion too much in CS50, but in CS51 and classes with data structures like trees, it becomes very useful.
  - If we go to Google[9], and search for `recusion` or `askew` or `do a barrel roll` we can encounter some little fun features.

Swap

- Let's now take a look at this function:

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

[9] http://www.google.com

- This function takes `a` and `b` as arguments but doesn't return a value (as implied by the `void`).
- Before we move on, a demonstration from a volunteer from the audience, Laura. We have some orange juice and milk, each in a glass, and to swap them, Laura needed a third cup, using it to store the orange juice. Then she poured the milk into the cup originally containing the orange juice, and finally the orange juice into the cup that originally had the milk.
- So a simple example, but we see more clearly why we need a temporary variable, `tmp`, in our code, to swap `a` and `b`.
- Let's open an example, `noswap.c`[10]:

[10] http://cdn.cs50.net/2014/fall/lectures/4/m/src4m/noswap.c

```
/**
 * noswap.c
 *
 * David J. Malan
 * malan@harvard.edu
 *
 * Should swap two variables' values, but doesn't!  How come?
 */

#include <stdio.h>

void swap(int a, int b);

int main(void)
{
    int x = 1;
    int y = 2;

    printf("x is %i\n", x);
    printf("y is %i\n", y);
    printf("Swapping...\n");
    swap(x, y);
    printf("Swapped!\n");
    printf("x is %i\n", x);
    printf("y is %i\n", y);
}

/**
 * Fails to swap arguments' values.
 */
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

- We call this `noswap` because it doesn't actually work. In `main`, we declare `x` and `y`, print out messages for us to see their values, call the `swap` function, and print their values again.
- But when we run it:

```
jharvard@appliance (~/Dropbox/src4m): ./noswap
x is 1
y is 2
Swapping...
Swapped!
x is 1
y is 2
```

- It didn’t work because each function, `main` and `swap`, has its own set of variables. Like the trays before, each gets a slice of memory, and when `main` calls `swap`, the operating system passes a copy of `x` and `y` to `swap`, storing it on its own "tray." The `swap` function works in swapping the numbers in its own memory, but has no impact on the variables in `main`.
- We can explain this with scope, since `x` and `y` are declared in the curly braces of `main`, and `swap` gets `a` and `b`, but those also only exist within `swap`.

GDB

- Let’s debug this, not with `printf` statements like you’ve probably done, but with a command called `gdb`, GNU Debugger. It has a bunch of features like `run`, `break`, `next`, `step`, `print`, etc, that allows us to understand and solve bugs in a program.
- We start by running:

```
jharvard@appliance (~/Dropbox/src4m): gdb ./noswap
```

  # which means that our program is not running in Bash, but inside this other program called `gdb`.

- Then we see a bunch of text, but most importantly the prompt:

```
(gdb)
```

- We can use `run` to run the program:

```
(gdb) run
Starting program: /home/jharvard/Dropbox/src4m/noswap
x is 1
y is 2
```

```
Swapping...
Swapped!
x is 1
y is 2
[Inferior 1 (process 16991) exited normally]
(gdb)
```

- We can ignore the other messages and notice it ran correctly. But let's do this:

```
(gdb) break main
Breakpoint 1 at 0x8048ac: file noswap.c, line 16.
(gdb) run
Starting program: /home/jharvard/Dropbox/src4m/noswap

Breakpoint 1, main () at noswap.c:16
16          int x = 1;
(gdb)
```

  # And now we see that `gdb` has paused our program, so we can walk through it step by step to see what's going on.

- If we went back to `gedit`, we'd see that line 16 is indeed `int x = 1;` in our `noswap.c` file.
- And `gdb` is telling us that line 16 hasn't been executed yet, but is about to be. If we say `print x` we get a value of `0`, but if we say `next`, it gives us the next line, line 17, and we see a value of `1` if we `print x` again:

```
(gdb) break main
Breakpoint 1 at 0x8048ac: file noswap.c, line 16.
(gdb) run
Starting program: /home/jharvard/Dropbox/src4m/noswap

Breakpoint 1, main () at noswap.c:16
16          int x = 1;
(gdb) print x
$1 = 0
(gdb) next
17          int y = 2;
(gdb) print x
$2 = 1
(gdb)
```

And `$1` and `$2` are just identifiers via which we can print those values again later.

- Now if we type `next` again and `print y`, we see `2`:

```
(gdb) next
19          printf("x is %i\n", x);
(gdb) print y
$3 = 2
(gdb)
```

- We can also type `list` to see where we are:

```
(gdb) list
14      int main(void)
15      {
16          int x = 1;
17          int y = 2;
18
19          printf("x is %i\n", x);
20          printf("y is %i\n", y);
21          printf("Swapping...\n");
22          swap(x, y);
23          printf("Swapped!\n");
(gdb)
```

- We continue, and see that our program prints to the screen again, and if we glance back to our code we can figure out which lines are from our program and which are from `gdb`:

```
(gdb) next
x is 1
20          printf("y is %i\n", y);
(gdb) next
y is 2
21          printf("Swapping...\n");
(gdb)
```

- Let's continue:

```
(gdb) next
Swapping...
22          swap(x, y);
```

```
(gdb) next
23          printf("Swapped!\n");
(gdb) print x
$5 = 1
(gdb) print y
$6 = 2
(gdb)
```

- So let's redo this, saying `next` until we get to this step:

```
(gdb) break main
Breakpoint 1 at 0x8048ac: file noswap.c, line 16.
(gdb) run
Starting program: /home/jharvard/Dropbox/src4m/noswap

Breakpoint 1, main () at noswap.c:16
16          int x = 1;
(gdb) next
17          int y = 2;
(gdb) next
19         printf("x is %i\n", x);
(gdb) next
x is 1
20          printf("y is %i\n", y);
(gdb) next
y is 2
21          printf("Swapping...\n");
(gdb) next
Swapping...
22          swap(x, y);
(gdb)
```

- Now we want to say `step`:

```
(gdb) step
swap (a=1, b=2) at noswap.c:33
33          int tmp = a;
(gdb)
```

  # Notice that we've jumped to line 33, which is the first line of code in `swap`, which can check in gedit.

- Let's really poke around now:

```
(gdb) print tmp
$7 = -1209908752
(gdb)
```

# There's a crazy value in `tmp` because it hasn't been initialized, and the bits there are left over from whatever code that used that piece of memory last.

- If we say `next` and then `print tmp`, we see `1`, which is the value of `a`, as we set it to:

```
(gdb) next
34          a = b;
(gdb) print tmp
$8 = 1
(gdb)
```

- In **Problem Set 3** is a tutorial on GDB, and the beginnings of a tool that will help us solve problems much more effectively.
- On Wednesday we'll look deeper and remove some layers from strings and look at the `char*` that's underneath.
- A preview of pointers[11], for Wednesday.

[11] http://youtu.be/9WsyLL6KVBY?t=50m11s