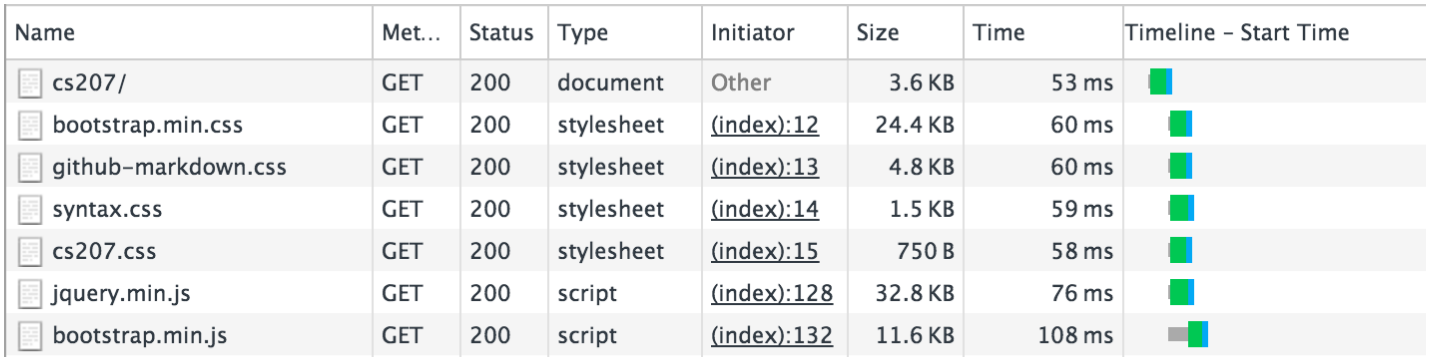
CS 207 | Systems Development for Computational Science

Introduction to C Lab

Q1: What to do while the data is away?

difficulty easy

The Chrome web browser has a diagnostic tool which records the network loading times for any page you visit. Below is a sample timeline for loading the CS207 homepage.



Name	Met...	Status	Type	Initiator	Size	Time	Timeline – Start Time
cs207/	GET	200	document	Other	3.6 KB	53 ms	
bootstrap.min.css	GET	200	stylesheet	(index):12	24.4 KB	60 ms	
github-markdown.css	GET	200	stylesheet	(index):13	4.8 KB	60 ms	
syntax.css	GET	200	stylesheet	(index):14	1.5 KB	59 ms	
cs207.css	GET	200	stylesheet	(index):15	750 B	58 ms	
jquery.min.js	GET	200	script	(index):128	32.8 KB	76 ms	
bootstrap.min.js	GET	200	script	(index):132	11.6 KB	108 ms	

It takes 108ms to retrieve all the page's data from the internet.

That's a *long* time for a computer to be waiting. Let's figure out exactly how long by estimating what we could do with that time.

AES (https://en.wikipedia.org/wiki/Advanced_Encryption_Standard), the Advanced Encryption Standard, is a common cryptographic algorithm for securing data. Let's say you've "acquired" a list of AES-encrypted passwords from a nefarious, trench-coat-wearing "friend". While we're waiting for our page to load, let's set to work breaking these passwords.

The most common methods for password cracking involve guessing random combinations, encrypting them, and checking against the encrypted list. To simplify things, let's assume that everything except the encryption is free. So we need to know how many passwords we can encrypt in 108ms.

Since 2010 (https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set), Intel CPUs have had special hardware built in which can compute AES in just a couple of instructions. Specifically, recent Haswell chips have an average decryption rate of 0.89 cycles per byte (http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/haswell-cryptographic-performance-paper.pdf) for 256-bit keys. If we assume that all your passwords are less than 11 bytes long, and that our processor runs at 2GHz, how many passwords can we try in the time it takes the CS207 webpage to load?

Q2: Fibonacci... again.

difficulty easy

Part 1: Do it.

Here is the recursive implementation of fibonacci from the previous lab:

```
def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-1) + fib(n-2)
```

Write it in C.

Part 2: Inspecting the stack

Compile your program with debugging symbols and load it into a debugger.

In most debuggers, breakpoints can be set not only on specific line numbers, but also on functions. This creates a breakpoint on the first non-declaration statement in the function. Set a breakpoint on your fibonacci function.

Run your program. It should freeze inside the fibonacci function. Now continue the program 10 times. In `gdb` , if you press "enter" at the gdb prompt, it will repeat the last command you ran. So here, to repeat 10 times, type `continue` and hit enter once, and then hit enter nine more times.

Print the call stack. How many stack frames are currently on the call stack? What is the current value of `n`? (hint: you can either do this with `print` or you can inspect the output of `backtrace`)

Q3: Crafting a buffer overflow attack

difficulty tricky

You are now the nefarious, trench-coat-wearing "friend" from question 1.

Part 1: Get familiar

You're given the code below. For all parts of this question, you aren't allowed to modify the source code at all.

```
// overflow.c
#include <stdlib.h>
#include <stdio.h>

void win()
{
  printf("Success!\n");
  exit(0);
}

int victim_function(int index)
{
  int64_t a[5] = {1,2,3,4,5};
  a[index] = (int64_t)(&win);
  return 0;
}

int main(int argc, char **argv)
{
  int magic_number = 0;

  if( argc>1 ) {
    magic_number = atoi(argv[1]);
  } else {
    printf("Please enter a magic number.\n");
    return -1;
  }

  victim_function( magic_number );

  return 0;
}
```

Copy the code into a file and compile it. The variables `argc` and `argv` are Unix's way of allowing C programs to interpret command-line arguments. `argc` is an integer containing the 1 + number of arguments given. `argv` is an array of strings. The first element of `argv` is the name of the program you compiled, and the rest of the arguments are strings corresponding to command-line arguments.

When you run your program like this:

```
$ ./overflow 0
```

`argc` is automatically set to 2, `argv[0]` is set to `"/your/path/to/overflow"`, and `argv[1]` is set to `"0"`. Additionally, the function `atoi` takes a string and converts it to an integer. It's a stupid function, so if you pass it a string that isn't an integer, like `"hello"`, it will just return 0. If you want, you can prove this to yourself by setting a breakpoint in gdb at the line with `atoi` and running `./overflow hello`. You can print the value of magic_number.

Part 2: Staking out the target

The general idea behind buffer overflows is that when a function adds a stack frame, it automatically stores the location it is supposed to return to on the stack. This is called a return address, and it corresponds to the place where the function was originally called. Because C arrays are just pointers, there is nothing (except good programming) to prevent you from continuing to index *beyond* the end of an array. In our code above, we specify the index for `a` on the command line, so there's nothing preventing us from passing a number larger than 5 or smaller than 0. A buffer overflow exploits this mechanism to *overwrite* the return address with a different value. When the function calls `return`, it jumps to this value, which could be anywhere!

In our case, the `victim_function` is already doing most of the work. It's writing the address of the function we want to call into the array. We just need to find a way to write that address over the return address on the stack.

Let's start by investigating what the program is already doing. Start your program, set a breakpoint at line 14 (the `a[index]=...` statement), and run it with `run 0`. Let's show the values of the array `a`:

```
(gdb) print/x a[0]@5
```

This command tells gdb to print out 5 values in hexadecimal format, starting at the address of `a[0]`. You should see the numbers 1 to 5. Now step forward one statement with the `next` command and print out the values again. What do you see? The first value of the array was overwritten with a funny value. This value is the address of the `win` function. Keep this in mind.

Part 3: Springing the trap

Now we just need to find the return address. We know two things:

- first, the return address must be located on the stack. The array `a` is also on the stack, so we can look before and after `a` to find the return address. This is equivalent to using large positive and small negative indices to `a`. For instance, this gdb command inspects `a` and the previous 5 and next 5 values on the stack:

```
(gdb) print/x a[-5]@15
```

- second, we know what the return address is! gdb gives us a complete listing of the call stack, including every return address as part of the `backtrace` command. The first entry on each stack entry is the return address. Run `backtrace` on your program now and write down the return address for the location in the `main` function where `victim_function` was originally called.

Now we know how to look around the stack, and we know the value of the return address we'd like to replace. Now it's your turn: use gdb to explore the values near to `a` on the stack. When you've found the return address, calculate its offset relative to the start of the array `a`. Use this value as a command line argument to your program to overflow the stack and win the game!

Once you've won, we'd like you to answer two quick questions:

In gdb, set a breakpoint on the function `win` and correctly overflow the stack. Run `backtrace`. What are the names of the other functions on the stack? Why do you think the call stack looks like this?