CSCI 2270
Hacking lab

For this week's lab, you'll simulate a little buffer overrun technique used to hack computers. You won't need to change much of the code, but you will need to figure out how to trick the computer into doing this. And it should help you realize that weirder things than segmentation faults can happen if we get sloppy about writing outside the bounds of an array. This lab builds on the guest lecture from last Friday.

Begin by downloading the lab.cxx code from the Moodle, and putting this file in a subdirectory of your machine. Open the file in Geany. This code has 4 functions plus a main().

The first function, `void PrintOverrunStack(void* varAddress)`, is used to print out the data on your local memory stack. You will use this function to see what the code is doing to your stack as it runs.

The second function, `void Overrun(char* text, char* command)`, actually contains the line of code with the buffer overrun you'll exploit. It's the line:

```
strcpy(var1, text);
```

This line copies a C-type string called text into another C-type string called var1. (C-type strings are the old-fashioned strings terminated with NULL (the special character '\0')). Stop here a moment and think about what's likely to happen if var1 and text are different lengths.

The third function, `void NotCalledExplicitly(char* command)`, is not something you need to change; it will execute if you can trick the computer into performing the overrun. If this code prints SUCCESS!!, you have solved the problem. Look closely to see what else has changed now.

The fourth function, `void CauseOverrun(int argc, char *argv[])`, sets up Overrun to run the hack. This function takes an integer input argc, and an array of NULL-terminated string arguments. It gets these argv arguments from the main function.

To run the code, compile it in the terminal, using the command:

```
g++ -Wall -g -m32 -fno-stack-protector -o lab lab.cxx
```

If you prefer, set your Geany Build commands so the build (ladder) command reads:

```
g++ -Wall -g -m32 -fno-stack-protector -o "%e" "%f"
```

We need to add this extra flag (`-fno-stack-protector`) so the compiler will allow us to perform the hack. (Don't worry about warnings too much for this lab.)

Run the code at the terminal by typing:

```
./lab
```

You'll see the stack variables print before and after the call to the overrun.

Now, it's time to unpack that stack. Run the code again in the terminal with 2 input arguments by typing:

```
./lab ls 0x11111111
```

This calls the lab code with 4 bytes in the text string (in the strcpy) call above. Run that. You should see this 0x11111111 address print before and after the call to the overrun.

Add another 4 byte address (the digits do not matter so much as the length):

./lab ls 0x11111111 0x22222222

You'll see the stack variables print before and after the call to the overrun. Where would a third parameter 0x33333333 end up in your stack? Verify this by changing the above command.

Recall from Friday's lecture that you can overwrite stack slots with these hexadecimal addresses. Where do you want to stop? You know that you have the local variables first on the stack—you have just overwritten a couple of these. Next, you have the frame pointer (a single address that points back into the stack), and then right after that, you have the return address for the function, which is the stack slot that we want to overwrite. To identify this, we want to find an address that is near the one for NotCalledExplicitly; this address is different from most of the other addresses you are seeing on the stack. Given what you know from trying

./lab ls 0x11111111

and

./lab ls 0x11111111 0x22222222,

you can see that the first part of the solution is figuring out how many 0xnnnnnnnn inputs it takes to overwrite down to the slot with the return address. This is a matter of counting the stack slots you overwrite with different inputs.

The second part of the solution is figuring out what value to supply last, after the 0xnnnnnnnn inputs, to overwrite the return address of Overrun with the new destination, which is the address of the function you did not explicitly call.

When you get the right number of inputs, and the right input address in the right place, you should see that a command you never typed in is now running! Demo this to your TA. You'll need to get this task solved in lab, since we won't get your same memory address results if you send us your working example.

When you get this done, re-compile the lab.cxx file with no stack sneakiness:

```
g++ -Wall -g -m32 -o lab lab.cxx
```

What happens when you run this code now? Compare it with the output you saw earlier.