COMS 4181: Security I

Programming Assignment 4
Due December 3rd, 11:59 pm

Late policy:
You have a total of two late days that you may choose to use in any of the assignments without any penalty. Any submission after the deadline will be considered as a late day and if it exceeds 24 hours after the deadline, then it will be considered as two late days. You may use the late days in separate assignments or together. Once you have used all of your late days, you will not receive any credit for late submissions.

Collaboration policy:
You are not allowed to discuss the solutions to homework problems/programming assignments with your fellow students.

Preliminaries:

You must use the LTS version of Ubuntu 18.04. Please download a ready to run image at OSBoxes (https://www.osboxes.org/ubuntu/). We recommend you only use VirtualBox. You must also have an x86 (Intel/AMD) family processor.

Installing Dependencies

Please run the following command to install all of the dependencies.

```
$ sudo apt-get install build-essential gdb gcc-multilib g++-multilib
```

NOTE: Depending on what language you choose to write the scripts for submission you may need to install additional dependencies (eg. python). Please document any additional packages you install in the REAME.md included in the homework.

Extract the homework4 project from `homework4.tar.xz` using a command like the following:

```
$ tar xvf homework4.tar.xz
```

A folder named `homework4` should have been extracted.

Part 0: Baby steps (0pts)

Compile the code

```
$ cd part0
$ make
```

Your goal here is to learn a bit about `gdb`, the GNU debugger. It'll prove immensely useful not only for this assignment, but on your future endeavours as a C/C++ developer.

The first thing you'll want to do is load a binary/executable.

```
$ gdb part0
```

Gdb will give you a prompt that looks like this: `(gdb).` From that prompt you can run your program, look at variables, etc., using the commands listed below (and others not listed).

help
Gdb provides online documentation. Just typing help will give you a list of topics. Then you can type help *topic* to get information about that topic (or it will give you more specific terms that you can ask for help about). Or you can just type help *command* and get information about any other command.

run
run will start the program running under gdb. (The program that starts will be the one that you have previously selected with the file command, or on the unix command line when you started gdb. You can give command line arguments to your program on the gdb command line the same way you would on the unix command line, except that you are saying run instead of the program name:
run 2048 24 4

You can even do input/output redirection: run > outfile.txt.

break
A ``breakpoint'' is a spot in your program where you would like to temporarily stop execution in order to check the values of variables, or to try to find out where the program is crashing, etc. To set a breakpoint you use the break command.

break *function* sets the breakpoint at the beginning of *function*. If your code is in multiple files, you might need to specify *filename:function*.
break *linenumber* or break *filename:linenumber* sets the breakpoint to the given line number in the source file. Execution will stop before that line has been executed.

delete

delete will delete all breakpoints that you have set.
delete *number* will delete breakpoint numbered *number*. You can find out what number each breakpoint is by doing info breakpoints. (The command info can also be used to find out a lot of other stuff. Do help info for more information.)

continue

continue will set the program running again, after you have stopped it at a breakpoint.

step

step will go ahead and execute the current source line, and then stop execution again before the next source line.

next

next will continue until the next source line in the current function (actually, the current innermost stack frame, to be precise). This is similar to step, except that if the line about to be executed is a function call, then that function call will be completely executed before execution stops again, whereas with step execution will stop at the first line of the function that is called.

list

list *linenumber* will print out some lines from the source code around *linenumber*. If you give it the argument *function* it will print out lines from the beginning of that function. Just list without any arguments will print out the lines just after the lines that you printed out with the previous list command.

print

print *expression* will print out the value of the expression, which could be just a variable name. To print out the first 25 (for example) values in an array called list, do
print list[0]@25

info

Info can give you a lot of useful information about a symbol. For this homework you'll probably want to do something like

info address *function*

To get the base address of a function.

disassemble
Disassemble will go ahead and show you the assembly for whatever you request.

There are a ton of resources available online as well so when in doubt check those as well. Here's a good one for example:
http://cseweb.ucsd.edu/classes/fa09/cse141/tutorial_gcc_gdb.html

You'll also need a very basic understanding of x86 assembly. A good primer can be found here: http://www.cs.virginia.edu/~evans/cs216/guides/x86.html Again there are a ton of good sources out there so feel free to use the one you understand best.

Part 1: Overflooooooooooooooooooow (20pts)

Compile the vulnerable code

```
$ cd part1
$ make
```

Objective:
Inspect (DO NOT MODIFY) the part1.c file for any potential vulnerabilities.
Your objective is to get the binary to print "`How did you even get here?!?!`" on the terminal.
Feed in some input via `stdin`.

Note:
Addresses will not change, you may hardcode them if necessary.
Use the exploit1.py or exploit1.sh files as templates for other parts.

To be submitted:

- A python or bash script named exploit (eg. exploit1.py or exploit1.sh) used to exploit the binary. Please save it in the same directory as part1.c

Part 2: Trusting A Stranger - Shellcode Style (40pts)

Compile the vulnerable code

```
$ cd part2
$ make
```

Objective:

Inspect (DO NOT MODIFY) the part2.c file for any potential vulnerabilities.
Your objective is to get the binary to print "`EasterEgg`" on the terminal by executing the `easteregg` function.
Feed in some input via `stdin`.

Note:
Addresses will not change, you may hardcode them if necessary.

Hint:
You might find some combination of the following x86 assembly instructions useful.

- `push` - opcode 0x68
- `ret` - opcode 0xc8
- `mov` - opcode 0xbb
- `call` - opcode 0xff

To be submitted:

- A python or bash script named exploit (eg. exploit2.py or exploit2.sh) used to exploit the binary. Please save it in the same directory as part2.c

Part 3: Mission Impossible - ROP Style (40pts)

Compile the vulnerable code

```
$ cd part3
$ make
```

Objective:
Inspect (DO NOT MODIFY) the `part3.c` file for any potential vulnerabilities.
Your mission should you choose to accept it is to get the binary to print "`This message will self destruct in 30 seconds...BOOM!`".
Feed in some input via `stdin`.

Note:
Addresses will not change, you may hardcode them if necessary.

To be submitted:

- A python or bash script named exploit (eg. exploit3.py or exploit3.sh) used to exploit the binary. Please save it in the same directory as part3.c