Project 4C
Internet Of Things Security

INTRODUCTION:

The Internet Of Things is populated with an ever expanding range of sensors and appliances. Initially such devices were likely to be connected to monitoring and control devices over purely local/personal networks (e.g., infra-red, Bluetooth, ZigBee), but it is increasingly common to connect such devices (directly or via a bridge) to the Internet. This enables remote monitoring and control, but it also exposes them to a variety of remote attacks.

For some targets (e.g., a national power grid or uranium separation centrifuges) their strategic importance and need for protection should be clear. It might not be immediatly obvious how one might hijack simple devices (e.g., light switches or temperature/humidity sensors) for nefarious purposes, but:

- there have been numerous instances where web-cams have been hijacked to violate peoples' privacy.
- smart devices like routers, baby-monitors, washing machines, and even lightbulbs have been conscripted into *botnets* to mount *Distributed Denial of Service* attacks and to be used for other bad purposes.
- security researchers have been able to hijack the digital controls of recent cars.
- consider the havoc that could be wrought by someone who was able to seize control of a networked pacemaker or insulin pump.

Attackers have proven innovative and resourceful in making use of compromised devices of many kinds, so even if you do not see any obvious dangers, prudence suggests that greater care be taken with the security of IOT devices. In particular, all communications and control for IOT devices should be encrypted and authenticated.

In this project we will extend your embedded temperature sensor to accept commands from, and send reports back to, a network server. You will do this over both unencrypted and encrypted channels.

RELATION TO READING AND LECTURES:

This project applies the principles discussed in the reading and lectures on Cryptography, Distributed Systems Security, and Secure Socket Layer encryption.

PROJECT OBJECTIVES:

- Primary: Demonstrate the ability to design, build and debug an embedded application that interacts with a central control server with the aid of server-side logs.
- Primary: Demonstrate the ability to implement a secure channel using standard tools.
- Primary: Demonstrate the ability to research and exploit a complex API, and to debug an application involving encrypted communication.

DELIVERABLES:

A single compressed tarball (`.tar.gz`) containing:

- C source files for two embedded applications (`lab4c_tcp` and `lab4c_tls`) that build and run (with no errors or warnings) on a Beaglebone.
- A `Makefile` to build and test your application. The higher level targets should include:
    - default ... build both versions of your program (compiling with the **`-Wall`** and **`-Wextra`** options).

    - **clean** ... delete all programs and output created by the Makefile and restore the directory to its freshly untarred state.
    - **dist** ... create the deliverable tarball.

  Note that this `Makefile` is intended to be executed on a Beaglebone, but you may find it convenient to create a `Makefile` that can be run on either a Beaglebone or a Linux server/desktop/notebook.
- A `README` file containing:
    - descriptions of each of the included files.
    - any other comments on your submission that you would like to bring to our attention (e.g., research, limitations, features, testing methodology).

PREPARATION:

- Part 1
    - Obtain the host name, port # and server status URL for the TCP logging server.
- Part 2
    - Obtain the host name, port # and server status URL for the TLS logging server.
    - Review the documentation for the OpenSSL SSL/TLS library, which should already be installed on your Beaglebone. You will likely want to seek out additional tutorials on using OpenSSL to initiate connections and verify server certificates.

PROJECT DESCRIPTION:

Part 1: Communication with a Logging Server

Write a program (called `lab4c_tcp`) that:

- builds and runs on your Beaglebone.
- is based on the temperature sensor application you built previously (including the `--period=`, `--scale=` and `--log=` options).
- accepts the following (mandatory) new parameters:
    - `--id=`*9-digit-number*
    - `--host=`*name or address*
    - `--log=`*filename*
    - *port number*

  Note that there is no `--port=` in front of the port number. This is a non-switch parameter.
- accepts the same commands and generates the same reports as the previous Beaglebone project, but now the input and output are from/to a network connection to a server.
    1. open a TCP connection to the server at the specified address and port
    2. immediately send (and log) an ID terminated with a newline:
       `ID=`*ID-number*
       This new report enables the server to keep track of which devices it has received reports from.
    3. as before, send (and log) newline terminated temperature reports over the connection
    4. as before, process (and log) newline-terminated commands received over the connection
       If your temperature reports are mis-formatted, the server will return a **`LOG`** command with a description of the error.
       Having logged these commands will help you find and fix any problems with your reports.
    5. as before, the last command sent by the server will be an **`OFF`**.
       Unlike the previous project, the button will not be used as an alternative manual shutdown mechanism.

  Do not accept commands from standard input, or send received commands or generated reports to standard output.

- As before, assume that the temperature sensor has been connected to Analog input 0.

The ID number (passed with the initial **ID=** command) will appear in the TCP server log (follow the TCP server URL), and will permit you to find the reports and server-side logs for your sessions. It is vitally important that the **ID=** string you send to the server (in your first report) be identical to the value (passed to your client) in the `--id=` command line argument. If they are not identical, the sanity-check and grading scripts will be unable to find the reports from your session(s) and will assume that your program did not work.

If the server receives incorrect reports from you it will send back LOG messages describing the errors.

To protect your privacy, you do not have to use your student ID number, but merely a nine-digit number that you will recognize and that will be different from the numbers chosen by others.

From the server status page, you will also be able to see, for each client, a log of all commands sent to and reports received from that client in the most recent session. If the server does not like your reports, it may be due to garbage (typically null) characters. If the problem with your reports is not obvious, examine the server log in an editor that will display non-graphical characters.

As in Project 4B, to facilitate development and testing you might find it helpful to write your program to, if compiled with a special (`-DDUMMY`) define, include mock implementations for the `mraa_aio_` and `mraa_gpio_` functionality. Doing so will enable you to do most of your testing on your regular computer. When you are satisfied that it works there, modify your Makefile run the command "`uname -r`", check for the presence of the string "`beaglebone`" in that output, and if not found, build with a rule that passed the `-DDUMMY` flag to *gcc*.

Part 2: Authenticated TLS Session Encryption

Write a program (called `lab4c_tls`) that:

- builds and runs on your Beaglebone
- is based on the remote logging appliance build in part 1
- operates by:
    1. opening a TLS connection to the server at the specified address and port
    2. sending (and logging) your student ID followed by a newline
    3. sending (and logging) temperature reports over the connection
    4. processing (and logging) commands received over the connection

The ID number will appear in the TLS server log (follow the TLS server URL), and will permit you to find the reports for your sessions.

The SSL library is not bullet-proof against protocol errors, and buggy clients will occasionally crash the server. It should be restarted pretty promptly. If not, ask one of the TAs for help. If the server seems to crash when you are testing your client, you probably have a bug in your SSL connection establishment.

Note that you may choose to:

- write two versions of the program
- write a single program that can be compiled to produce two different executables
- write a single executable that implements both functionalities, and chooses which to use based on the name by which it was invoked. In this last case, your Makefile should produce two different links (with the required names) to that program.

SUMMARY OF EXIT CODES:

- 0: successful run

- 1: invalid command-line parameters (e.g., unrecognized parameter, no such host)
- 2: other run-time failures (e.g., server rejected or closed session)

SUBMISSION:

Your tarball should have a name of the form `lab4c-`*studentID*`.tar.gz`. You can sanity check your submission with this test script which should run on your Beaglebone or (if with appropriately dummied sensor access) on your usual Linux development environment. There will be no manual re-grading on this project. Submissions that do not pass the test script are likely to receive very low scores.

Your **README** file (and all source files) must include lines of the form:

> **NAME:** *your name*
> **EMAIL:** *your email*
> **ID:** *your student ID*

And, if slip days are allowed on this project, and you want to use some, this too must be included in the **README** file:

> **SLIPDAYS:** *#days*

If, for instance, you wanted to use two slip-days, you would add the following line:

> **SLIPDAYS: 2**

GRADING:

Points for this project will be awarded:

value	feature
	Packaging and build (10% total)
3%	un-tars expected contents
3%	clean build of correct program w/default action (no warnings)
2%	Makefile has working `clean`, `dist` targets
2%	reasonableness of `README` contents
	Unencrypted (50% total)
20%	establishes TCP session, and presents ID
10%	reports temperatures
10%	correct command processing
10%	command and data logging
	Encrypted Server Sessions (40% total)
20%	establishes TLS session, presents ID
10%	reports temperatures
10%	correct command processing