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 stratigic 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 *bot-nets* to mount *Distributed Denial of Service* attacks.
- 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 pace-maker or insulin pump.

Prudence suggests that all communications and control for IOT devices should be encrypted.

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 an Edison.
- A `Makefile` to build and test your application. The higher level targets should include:
  - default ... build both versions your program
  - **clean** ... delete all programs and output created by the Makefile
  - **dist** ... create the deliverable tarball

Note that this `Makefile` is intended to be executed on an Edison, but you may find it convenient to create a `Makefile` that can be run on either an Edison or a Linux server/desktop/notebook.

- a `README` file containing:
  - descriptions of each of the included files and any other information about your submission that you would like to bring to our attention (e.g. research, limitations, features, testing methodology).
  - any other comments on your submission (e.g. references consulted, slip days, etc.)

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 Edison. 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 Edison
- is based on the temperature sensor application you built previously
- accepts the following parameters:
  - `--id=`*9-digit-number*
  - `--host=`*name or address*
  - `--log=`*filename*
  - (required) *port number*

  Note: that there is no `--port=` in front of the port number. This is non-switch parameter.
- It accepts the same commands and generates the same reports as the previous Edison 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. send (and log) newline terminated temperature reports over the connection
  4. process (newline terminated) commands reveived over the connection
  5. the last command sent by the server will (as before) be `OFF`
- as before, assume that the temperature sensor has been connected to Analog input 0.

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

To facilitate development and testing I wrote my program to, if compiled with a special (`-DDUMMY`) define, include mock implementations for the `mraa_aio_` functionality, enabling me to do most of my testing on my desktop. I then modified my Makefile run the command "`uname -r`", check for the presence of the string "`edison`" 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 Edison
- 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 your student ID followed by a newline
    3. send temperature reports over the connection
    4. process commands reveived over the connection
    5. the last command sent by the server will be `OFF`

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.

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 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

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 Edison or (if with appropriately dummied sensor access) on your usual Linux development environment.

Note that the sanity checker works, in part, by checking the server logs for entries corresponding to the student ID you have given as a parameter. Thus, in order to pass the sanity check, you must have had recent successful sessions using your own student ID number (or at least the same number you have used to name your submission tarball).

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

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

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