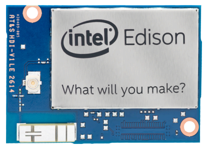
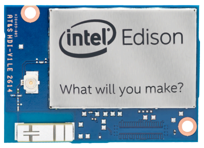
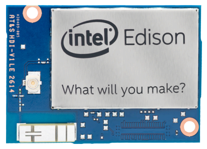


Internet of Things Security
Computer Science 111
Intel® Edison:



Contents

Revision history		
Version	**Date**	**Comment**
1.0	2/26/2016	First Draft



Introduction

In this tutorial, you will:

1. Learn about some common security vulnerabilities associated with the Internet of Things.
2. Learn about the security vulnerabilities associated with unprotected UDP network data transport.
3. Understand how the TLS protocol overcomes these vulnerabilities

The tutorial will be performed on an Intel Edison board. This platform is designed to support the kinds of embedded devices that will be increasingly common in the future. While it will require you to gain some familiarity with this platform, the underlying operating system software is essentially Linux, so your existing experience should prove helpful in getting started.

Things Needed

1. 2x Micro USB cable
2. 1x Intel Edison Board
3. 1x Server machine with tshark installed on it (This will be provided for you. Your TA will give you any necessary instructions on how to access it.)
4. All required code resources are available at:
   a. https://drive.google.com/folderview?id=0B4NGslzPqDhvbFU3d0prSmdiNjQ&usp=sharing

Background

Introduction

The new Internet of Things (IoT) thrust is one of the most important trends in technology today. IoT systems provide sensors and actuators in our global environment, on the scale of nations and cities, to individual buildings, homes, and vehicles. IoT systems also include the rapidly expanding world of wearable devices for the field of Wireless Health or Mobile Health (mHealth).

A critical requirement of Wireless Health is the support of patient monitoring by applying remote sensors. The sensors monitor such vital signs as heart rate, pulse, blood sugar levels, and many other critical health conditions. Generally, while the devices performing the sensing are located on or near the patient's body, the data ultimately must go to a health care facility for analysis. The first step in that process is using a wireless network to move the data gathered from the patient to some other nearby machine, which typically in turn passes it over the Internet to a doctor, hospital, or other health care facility. So the sensor must not only be able to observe the patient's condition, it must also operate as a wireless host computer. In many cases, the sensor is configurable, perhaps monitoring different vital signs, or monitoring them at different frequencies, or at different levels of sensitivity. In such cases, not only must the sensor's communications and computational capabilities move data off the system, but they must accept commands controlling their behavior coming in over the wireless network.

Health data is inherently private, yet wireless networking is inherently open. Anyone who puts up an antenna within range of a wireless device will hear its signal. So unless measures were taken to protect the wireless communications from sensor to base station, eavesdroppers could learn all the information they exchange. Such a vulnerability is extremely undesirable. Without going into the details, the fundamental way to address this kind of vulnerability is to encrypt the data, making it incomprehensible to anyone except the sender (the Edison, here) and the receiver (the base station that will pass the data to the health care facility).

These devices are now based on compact platforms including the Intel Edison IoT platform that you will be provided. This platform uses the Linux Operating System as its software base. Fortunately, the Linux system contains high quality built-in mechanisms to perform encryption and decryption. One only has to use them properly (which can be challenging, itself).

In this assignment, we are providing you with your own Intel Edison. (This will need to be returned after the completion of your assignment.)

This device will operate as a *virtual* heart rate monitor. It will apply a client-server architecture to communicate heart rate data to a remote server. The remote server also will include the capability to communicate *to* the remote Edison *virtual* heart rate monitor and command a change in heart rate monitoring period. Your Edison will come with an application to perform the virtual heart rate monitoring and basic wireless communication to a server already installed. However, this installed version does not provide any security via cryptography.

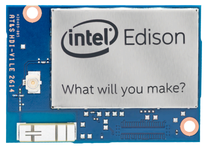
You will use this system and observe its operation through usage of a network monitoring tool, tshark. This tool will provide you with the capability to view network transported data. You will be able to compare transport systems that are not secure and contrast these with secure methods. Also, you will have the capability to *launch an attack* on this system and observe the results.

Finally, your objective will be to develop a secure client that resists attack and you again will be able to observe this.

Your Security Problem:

There are many different aspects to securing computers, and this exercise will only touch on a couple of them. In particular, we will deal with issues of confidentiality and authenticity in wireless communication. By confidentiality, we mean only allowing authorized parties to see the data. In this case, the authorized parties are the Edison device and the server it communicates to. This description of our confidentiality requirement instantly raises the question of authenticity: how can we be sure that a party we think is authorized is really who he claims to be? One way for an attacker to overcome our cryptography is to get us to regard him as the authorized party. Then he will be able to see all our confidential data.

You have been given a client side implementation of the virtual heart rate monitor that communicates with the server using the socket interface, a standard method of communicating between machines. If you are not familiar with using sockets in your programs, or if you'd like a refresher on them, here's a pointer to a useful tutorial:



http://www.tutorialspoint.com/unix_sockets/index.htm

Sockets do not inherently provide any confidentiality or authentication, so the client side implementation you've been given does not meet the security goals for this application. It does not ensure that the partner one communicates with via the socket is indeed the server, nor does it ensure that only the server will be able to see the data we are sending.

To achieve our authenticity goals, we must authenticate the server to the Edison device, and the Edison device to the server. (Think about why we need to authenticate both of them to each other.) How can we do that? In this exercise, we will rely on another form of cryptography to authenticate the parties. In essence, each party (the Edison and the server) will have a secret (called a private key) only that party knows. It uses that secret to encrypt the data in a special way. The other party doesn't know that secret, but it knows another piece of information (called a public key) that is associated with the authenticating party. This public key can be used to check the special encryption to determine if it was performed in the correct way. If it was, we believe that the other party is who it claims to be, since no one else could perform that encryption.

This authentication process is called public key cryptography. Whitfield Diffie and Martin Hellman won the Turing Award in 2016 for their invention of this idea in the 1970s. If you'd like to learn more about it, check out

https://en.wikipedia.org/wiki/Public-key_cryptography.

However, you shouldn't need to use that article to perform this exercise.

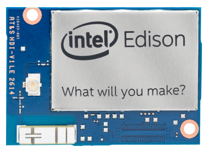
We can't use public key cryptography to provide confidentiality for a number of reasons. One reason is that public key cryptography typically assumes everyone knows a party's public key. So if we only encrypted with the private key, everyone in the world would be able to see the data. We wouldn't get much confidentiality out of that. So we use another form of cryptography to provide the confidentiality. This form is called symmetric cryptography, since both parties know the same secret key when we use it. Since only they know that key, only they can read the messages, and confidentiality is preserved. You can learn more about symmetric cryptography by reading

https://en.wikipedia.org/wiki/Symmetric-key_algorithm.

Again, you should be able to do this exercise without reading this article.

If you give the matter a little thought, you might wonder how we ensure that the sender and receiver share a particular symmetric key, when nobody else does. There are a number of ways of doing this, which we need not get into here. Should you care to learn more, check out

https://en.wikipedia.org/wiki/Key_exchange



And, yet again, you can do this exercise without knowing what's in that article, because fortunately the Linux operating system contains functions that help you exchange keys. In particular, using the TLS suite of functions allows relatively simple combined use of authentication through public key cryptography, key exchange of a key to perform symmetric cryptography, and encryption and decryption of the messages whose confidentiality we need to protect.

The TLS Security Technology:

The existing client side implementation uses the User Datagram Protocol (UDP). UDP does not encrypt traffic — the data is sent out over the wireless medium in plain text. If an individual is performing a banking transaction at an Internet café over UDP, their passwords can be stolen by someone executing a packet inspector tool like tshark. Furthermore, their data is also vulnerable to a "man in the middle attack" where a malicious party can inspect packets and send edited versions of them to the destination.

Transport Layer Security (TLS) overcomes this vulnerability through a combination of authentication and encryption. It essentially uses sockets, but applies cryptography to them. Cryptography is notorious for being difficult to use properly, so using TLS, rather than trying to add it entirely on your own, is a much easier and safer option. So one task you will perform in this lab is to switch the existing client side implementation of the heart rate monitor to use TLS instead.

TLS is available on most operating systems you are likely to work with, so it's well worth your while to learn how to use it. Here is a tutorial on how to use TLS that should prove helpful in performing this part of the lab:

https://www.sans.org/reading-room/whitepapers/protocols/ssl-tls-beginners-guide-1029

Setup

Setup the Intel Edison:

If you are new to working with the Intel Edison boards, please see the tutorials available at the following link Intel Edison IoT Tutorials
https://drive.google.com/a/g.ucla.edu/folderview?id=0B7UDOP1BnKJLflRWUy1xNkF6bFpBYW9qYTN0dWxCVjRkNUl6NUFIMXR6QWpLMGNLeWdQSTg&usp=sharing&ts=5685baab

1. Plug in and set up your Intel Edison board as described in Intel Edison IoT Tutorials
2. Login to your Edison over the SSH protocol (this guide will use MobaXterm for SFTP and SSH)
3. Download the files available at https://drive.google.com/folderview?id=0B4NGslzPqDhvbFU3d0prSmdiNjQ&usp=sharing
    a. Within each subfolder, download the folder labelled "client"
4. Transfer the downloaded folders to your Edison
    a. TLS/client/… 7 Files total
    b. wireshark/client/… 3 Files total

        c. UDP/client/… 5 Files total
5. Navigate into the "../heartbeat/UDP/client" directory
6. Type "make"
7. Ask your lab instructor for the port information for:
    a. UDP server
    b. TLS server
    c. wireshark server
8. Update the configuration files labelled "config_file" in each directory under the subfolder "client" to match the server machines port and IP address given to you by your lab instructor.

Tasks

Investigating the Existing Implementation

This set of tasks involves examining the output from a packet sniffer installed at the server location. A packet sniffer is a program that listens to traffic on a shared channel (like a wireless network) and pulls out everything it hears, whether it is addressed to that program or not. In essence, it hears all the information sent across that shared channel. You will use this packet sniffer to understand why unencrypted communication channels pose a risk in today's world.

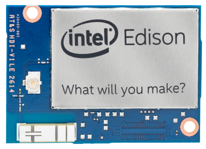
1. Navigate to the folder labelled "../heart_rate/UDP/client/"
2. Execute the UDP client by typing "./udp_client" into your terminal
    a. This must be running until you are finished with the whole UDP section
    b. Take a screenshot of the first 10 lines of the output. Include this screenshot in your lab submission.
3. Examine the output, see what is being sent to the server. Write a brief description of what you see and how this relates to the security goals of this system, and include it in your lab submission.
4. Open a new terminal window
    a. Navigate to the "wireshark" folder
    b. Execute ./get_tshark <port_udp_server_is_running_on>
    c. The output of this program is from the packet sniffer
    d. Explain the output of the packet sniffer and why this is important
    e. Take a screen shot of the output
    f. Close the terminal
    g. Include both the explanation and the screen shot in your lab submission
5. Open a new terminal window:
    a. Execute the command "./set_rate <0 – 5>"
    b. Comment on how your "./udp_client" reacts and take a screenshot of the output
    c. DO NOT CLOSE THIS TERMINAL WINDOW YET
    d. Repeat step 4
    e. Include the screenshot and your comments in your lab submission
6. In the terminal window opened in step 4:
    a. Execute the command "./start_attack"
    b. Comment on how your "./udp_client" reacts and take a screenshot of the output
    c. DO NOT CLOSE THIS TERMINAL WINDOW YET
    d. Repeat step 4

    e. Include the screenshot and your comments in the lab submission
7. Close the terminal window opened in step 5
8. Press cntrl+c in the terminal window that "./udp_client" is running in to kill the client

Building a More Secure Implementation

This set of tasks involves improving the UDP version of the client code to provide better security, using TLS. After making the required changes, you will run your new client code and examine the output from a packet analysis tool installed at the server location. You will use this output to understand why encrypted communication channels are crucial.

1. Navigate to the folder "../heart_beat/TLS/client/"
2. Type "make"
3. Execute the TLS client by typing "./tls_client" into your terminal
    a. This must be running until you are finished with the whole TLS section
    b. Take a screenshot of the first 10 lines of the output
    c. Note that your client will react by displaying messages returned from the server that match messages sent to the server by the client.
    d. Include this screenshot in your lab submission
4. Examine the output, see what is being sent to the server. Include an analysis of this output in your lab submission.
5. Open a new terminal window
    a. Navigate to the "wireshark" folder
    b. Execute ./get_tshark <port_tls_server_is_running_on>
    c. The output of this program is from the packet sniffer
    d. Explain the output of the packet sniffer and why this is important
    e. Take a screen shot of the output
    f. Close the terminal
    g. Include the screenshot and the explanation in your lab submission
6. Open a new terminal window:
    a. Execute the command "./set_rate <0 – 5>"
    b. Comment on how your "./tls_client" reacts and take a screenshot of the output that appears both before and after the execution, "./set_rate 1"
    c. DO NOT CLOSE THIS TERMINAL WINDOW YET
    d. Repeat step 6
    e. Note that your client will react by displaying messages returned from the server that may not match messages sent to the server by the client.
    f. Include the screenshot and your comments in your lab submission
7. In the terminal window opened in step 7:
    a. Execute the command "./start_attack"
    b. Comment on how your "./tls_client" reacts and take a screenshot of the output
    c. DO NOT CLOSE THIS TERMINAL WINDOW YET
    d. Repeat step 6
    e. Include the screenshot and your comments in your lab submission
8. Close the terminal window opened in step 7
9. Press cntrl+c in the terminal window that "./tls_client" is running in to terminate the client execution



Building a More Secure Implementation with Capability for Resolving Error Behavior

This set of tasks involves developing a new TLS client that resolves the condition leading to the behavior associated with unmatched message response noted in step 6 above. You may use the tls_client.c program to create a new program.

First, it is important to note that tls_client sends a message to the server via the SSL_write() function. It expects a matching reply via SSL_read().

However, if an unexpected reply message is sent by the server, then the return of SSL_read() will not match that of the message sent by SSL_write(). The behavior appearing in Step 7 will result. This unexpected behavior could result from multiple forms of system error or system compromise.

Your task is to resolve this behavior through two steps:

1) Extend the tls_client.c program to include a second thread of execution. This thread will include SSL_read() actions.
2) Extend the tls_client.c program to also include logging of messages written to a log file.

After making the required changes, you will

1) Execute your new client code and examine the output of the client.
2) During execution of the client, execute the command "./set_rate 1"
3) Comment on how your "./tls_client" reacts and take a screenshot of the output that appears both before and after the execution, "./set_rate 1"
4) Comment on how your dual thread tls_client avoids errors that are discussed above.
5) Include a screenshot in your lab submission.
6) Examine the log file and take a screenshot of the log file contents that appears both before and after the execution, "./set_rate 1"
7) Include a screenshot in your lab submission

Final Submission

Please zip all files you have edited/created. This should include:

1. tls_client.c
2. 4 screenshots for UDP_client
3. 4 screenshots for the single thread TLS_client responses
4. 1 screenshot for the dual thread TLS_client responses
5. The TLS_client log file
6. Your explanations for each part including a description of the log file format and description of the received messages

Email the resulting zip file to your lab instructor.