Chapter 6: Establishing (Secure) Communication using TCP/IP Sockets

Objective

At the end of Chapter 6 you will understand how to use the WICED-SDK to send and receive data using TCP/IP sockets. You will also understand the fundamentals of symmetric and asymmetric encryption and how it is used to provide security to your IoT device.

Time: 2 ¼ Hours

Fundamentals

Sockets – Fundamentals of TCP Communication

For Applications, i.e. a web browser, to communicate via the TCP transport layer they need to open a **Socket**. A Socket, or more properly a TCP Socket, is simply a reliable, ordered pipe between two devices on the internet. To open a socket you need to specify the IP Address and Port Number (just an unsigned 16-bit integer) on the Server that you are trying to talk to. On the Server there is a program running that listens on that Port for bytes to come through. Sockets are uniquely identified by two tuples (source IP: source port) and (destination IP:destination port) e.g. 192.168.15.8:3287 + 184.27.235.114:80. This is one reason why there can be multiple open connections to a webserver running on port 80. The local (or ephemeral port) is allocated by the TCP stack and new ports are allocated on the initiator (client) for each connection to the receiver (server).

There are a bunch of standard ports (which you might recognize) for Applications including:

- HTTP 80
- SMTP 25
- DNS 53
- POP 110
- MQTT 1883

These are typically referred to as “Well Known Ports” and are managed by the IETF Internet Assigned Numbers Authority (IANA); IANA ensures that no two applications designed for the Internet use the same port (whether for UDP or TCP).

WICED easily supports TCP sockets (*wiced_tcp_create_socket()*) and you can create your own protocol to talk between your IoT device and a server or you can implement a custom protocol as defined by someone else.

To build a custom protocol, for instance, we can define the WICED Wi-Fi Example Protocol (WWEP) as an ASCII text based protocol. The client and the server both send a string of characters that are of the form:

- Command: 1 character representing the command (R=Read, W=Write, A=Accepted, X=Failed).
- Device ID: 4 characters representing the hex value of the device e.g. 1FAE or 002F. Each device will have its own unique register set on the server so you should use a unique ID (unless you want to read/write the same register set as another device).
- Register: 2 characters representing the register (each device has 256 registers) e.g. 0F or 1B.
- Value: 4 characters representing the hex value of a 16-bit unsigned integer. The value should be left out on "R" commands.

The client can send "R" and "W" commands. The server responds with "A" (and the data echo'd) or "X" (with nothing else). The server contains a database that will store values that are written to it (when a client uses the "W" command) and will send back requested values (when a client uses the "R" command). The server keeps track of a separate 256 register set for each device ID. For example, the register with address 0x0F for a device with ID 0x1234 is not the same as register with address 0x0F for a device with ID 0xABCD.

The open version of the protocol runs on port 27708 and the secure TLS version runs on port 40508. We will be using the open version of the protocol in this class.

Some examples:

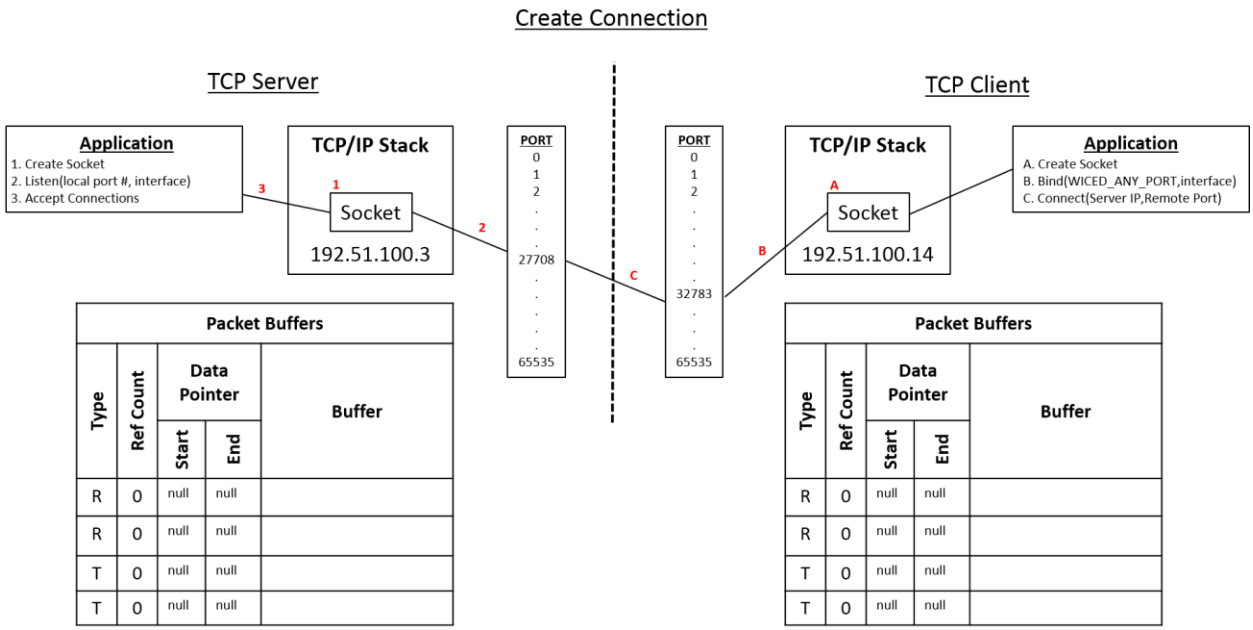
- "W0FAC0B1234" would write a value of 0x1234 to register 0x0B for device with an ID of 0x0FAC. The server would then respond with "A0FAC0B1234".
- "W01234" is an illegal packet and the server would respond with "X".
- "R0FAC0B" is a read of register 0x0B for a Device ID with an ID of 0x0FAC". In this case the server would respond with "A0FAC0B1234" (the value of 1234 was written in the first case).
- "R0BAC0B" is a legal read, but there has been no data written to that device so the server would respond with "X".

Note that "raw" sockets inherently don't have security. The TCP socket just sends whatever data it was given over the link. It is the responsibility of a layer above TCP such as SSL or TLS to encrypt/decrypt the data if security is being used (which we will cover later on).

Sockets are available in the WICED SDK and enable you to build your own custom protocol. However, in general developers are mostly using one of the standard Application Protocols (HTTP, MQTT etc.) which are discussed in Chapter 7.

WICED-SDK TCP Server & Client using Sockets

In the examples below I use the WWEP protocol as defined in the previous section to demonstrate the steps to create a connection between a WWEP Client (198.51.100.14) and a WWEP Server (198.51.100.3) using sockets.



The picture above describes the steps required to make a TCP connection between two devices, a TCP Server (on the left of the dotted line) and a TCP Client (on the right). These two devices are already connected to an IP network and have been assigned IP addresses (192.51.100.3 and 14). There are 4 parts of each system:

- Your firmware applications (the boxes labeled Application). This is the firmware that you write to control the system using the WICED-SDK. There is firmware for both the server and client.
- The TCP/IP stack which handles all of the communication with the network.
- The Port, which represents the 65536 TCP ports (numbered 0-65535).
- The Packet Buffer, which represents the 4x ~1500 bytes of RAM where the Transmit "T" and Receive "R" packets are held.

To setup the TCP server connection, the server firmware will:

1. Create the TCP socket by calling (the *socket* is a structure of type *wiced_tcp_socket_t*):

   wiced_tcp_create_socket(&socket, WICED_AP_INTERFACE);

2. Attach the socket to WWEP server TCP port 27708 by calling:

   wiced_tcp_listen(&socket, 27708); // 27708 is the port number for the WWEP protocol

3. Sleep the current thread and wait for a connection by calling:

   wiced_tcp_accept(&socket);

To setup the TCP client connection, the client firmware will:

A. Create the TCP socket by calling:

   wiced_tcp_create_socket(&socket, WICED_STA_INTERFACE);

B. "Bind" to some TCP port (it doesn't matter which one, so we specify WICED_ANY_PORT which lets the TCP/IP stack choose any available port) by calling:

   wiced_tcp_bind(&socket, WICED_ANY_PORT);

C. To create the actual connection to the server you need to do two things:
   a. Find the server address. This is passed as a WICED data structure of type *wiced_ip_address_t*. Let's assume you have defined a structure of that type called *serverAddress*.

      You can initialize the structure in one of two ways – either statically or using DNS.

      - To initialize it statically you can use the macros provided by the WICED SDK as follows:

        SET_IPV4_ADDRESS(serverAddress, MAKE_IPV4_ADDRESS(198, 51, 100, 3));

      - To initialize it by performing a DNS loop, do the following:

        wiced_hostname_lookup("wwep.ww101.cypress.com", &serverAddress, 10000, WICED_STA_INTERFACE);

   b. Now that you have the address of the server, you make the connection to port 27708 through the network by calling *wiced_tcp_connect()* and waiting a TIMEOUT number of milliseconds for a connection. In our local network the timeout can be small <1s but in a WAN situation the timeout may need to be extended to as long as a few seconds:

      wiced_tcp_connect(&socket, &serverAddress, 27708, TIMEOUT);

Transmitting and Receiving Data using Streams

Once the connection has been created, your application will want to transfer data between the client and server. The simplest way to transfer data over TCP is to use the stream functions from the WICED SDK. The stream functions allow you to send and receive arbitrary amounts of data without worrying about the details of packetizing data into uniform packets (see the next section for details about packets).

To use a stream you must first declare a stream structure and then initialize that with the socket for your network connection:

```
wiced_tcp_stream_t stream;
wiced_tcp_stream_init(&stream, &socket);
```

Once this is done it is simple to write data using the *wiced_tcp_stream_write()* function. This function takes the stream and message as parameters. The message is just an array of characters to send. When

you are done writing to the stream you need to call the *wiced_tcp_stream_flush()* method. The following code demonstrates writing a single message:

```
char sendMessage[] = "TEST_MESSAGE";
wiced_tcp_stream_write(&stream, sendMessage, strlen(sendMessage));
wiced_tcp_stream_flush(&stream);
```

Reading data from the stream uses the *wiced_tcp_stream_read()* function. This method takes a stream and a message buffer as parameters. The function also requires you to specify the maximum number of bytes to read into the buffer and a timeout. The function returns a *wiced_result_t* value which can be used to ensure that reading the stream succeeded.

```
result = wiced_tcp_stream_read(&stream, rbuffer, 11, 500);
```

Behind the scenes, reading and writing via streams uses uniform sized packets. The stream functions in the WICED SDK hide the management of each of these packets from you so you can focus on the higher levels of your application. However, if you desire more control over the communication you can use the WICED SDK API to send and receive packets directly.

Given the above, the firmware to transmit date using streams might look something like this:

```
#define SERVER_PORT  (27708)
#define TIMEOUT (2000)
.
.
wiced_tcp_socket_t socket;
wiced_tcp_stream_t stream;
char sendMessage[]="WABCD051234";
.
.
wiced_tcp_create_socket(&socket, WICED_STA_INTERFACE);
wiced_tcp_bind(&socket, WICED_ANY_PORT );
wiced_tcp_connect(&socket, &serverAddress, SERVER_PORT, TIMEOUT);
wiced_tcp_stream_init(&stream, &socket);
wiced_tcp_stream_write(&stream, sendMessage, strlen(sendMessage);
wiced_tcp_stream_flush(&stream);
wiced_tcp_stream_deinit(&stream);
wiced_tcp_delete_socket (&socket);
```

(Advanced) Transmitting Data using Packets as a TCP Client using the WICED SDK

At the beginning of your application, when you run the *wiced_init()* function, on the console you will see the message "Creating Packet pools". The packet pools are just RAM buffers which store either incoming packets from the network (i.e. Receive packets) or will hold outgoing packets which have not yet been sent (i.e. Transmit packets). By default, there are two receive packets and two transmit packets but this can be configured in your firmware. If you run out of receive packets then TCP packets will be tossed. If you run out of transmit packets you will get an error when you try to create one.

Each packet in the buffer contains:

- An allocation reference count

Packet Buffer				
Type	Ref Count	Data Pointer		Buffer
		Start	End	
R	0	null	null	
R	0	null	null	
T	0	null	null	
T	0	null	null	

- The raw data
- A pointer to the start of the data
- A pointer to the end of the data
- The TCP packet overhead

A packet starts its life unallocated, and as such, the reference count is 0. When you want to send a message, you call *wiced_tcp_packet_create()* which has the prototype of:

```
wiced_result_t wiced_packet_create_tcp( wiced_tcp_socket_t* socket,
        uint16_t content_length,
        wiced_packet_t** packet,
        uint8_t** data,
        uint16_t* available_space );
```

This function will look for an unallocated packet (i.e. the reference count == 0) and assign it to you. The arguments are:

- *socket*: A pointer to the socket that was previously created by *wiced_tcp_connect()*.
- *content_length*: How many bytes of data you plan to put in the packet.
- *packet*: a pointer to a packet pointer. This enables the create function to give you a pointer to the packet structure in the RAM. To use it, you declare: *wiced_packet_t *myPacket;* Then when you call the *wiced_packet_create_tcp()* you pass a pointer to your pointer e.g. *&myPacket*. When the function returns, *myPacket* will then point to the allocated packet in the packet pool.
- *data*: a pointer to a uint8_t pointer. Just as above, this enables the create function to give you a pointer to the packet structure in the RAM. To use it, you declare: *uint8 *myData;* then when you call the *wiced_packet_create_tcp()* you pass a pointer to your pointer e.g. *&myData*. When the function returns, *myData* pointer will then point to the place inside of the packet buffer where you need to store your data.
- *available_space*: This is a pointer to an integer that will be set to the maximum amount of data that you are allowed to store inside of the packet. It works like the previous two in that the function changes the instance of your integer.

Once you have created the packet, you need to:

- Copy your data into the packet in the correct place i.e. using *memcpy()* to copy to the data location that was provided to you.
- Tell the packet where the end of your data is by calling *wiced_packet_set_data_end()*.
- Send the data by calling *wiced_tcp_send_packet()*. This function will increment the reference count (so it will be 2 after calling this function).
- Finally, you release control of the packet by calling *wiced_packet_delete()*. This function will decrement the reference count. Once the packet is actually sent by the TCP/IP stack, it will decrement the reference count again, which will make the packet buffer available for reuse.

After the call to wiced_tcp_packet_create_tcp:

- The pointer *myPacket* will point to the packet in the packet pool that is allocated to you.

- *availableDataSize* will be set to the maximum number of bytes that you can store in the packet (about 1500). You should make sure that you don't copy more into the packet than it can hold. In order to keep this example simple, I didn't perform this check in the above code.
- The pointer *data* will point to the place where you need to copy your message (which I do in the line with the *memcpy*).

Be very careful with the line that calls wiced_tcp_set_data_end as you are doing pointer arithmetic.

(Advanced) Receiving Packets as a TCP Server using the WICED SDK

As a TCP Server you will probably have a thread that will:

- Call the *wiced_tcp_accept(&socket)* function which will suspend your thread and wait for data to arrive. Once data arrives it will wakeup your thread and continue execution. The RTOS has an "accept timeout", which by default will wake your thread after about 3 seconds. If it times out, the return value from *wiced_tcp_accept* will be something other than WICED_SUCCESS. It is then your choice what to do.
- Once the data has arrived you can call *wiced_tcp_receive*. This function has the prototype:

  ```
  wiced_tcp_receive( wiced_tcp_socket_t* socket,
                     wiced_packet_t** packet,
                     uint32_t timeout )
  ```

  The *wiced_packet_t ** packet* means that you need to give it a pointer to a pointer of type *wiced_packet_t* so that the receive function can set your pointer to point to the TCP packet in the packet pool. This function will also increment the reference count of that packet so when you are done you need to delete the packet by calling *wiced_packet_delete*.
- Finally, you can get the actual TCP packet data by calling *wiced_packet_get_data* which has the following prototype:

  ```
  wiced_result_t wiced_packet_get_data(
                     wiced_packet_t* packet,
                     uint16_t offset,
                     uint8_t** data,
                     uint16_t* fragment_available_data_length,
                     uint16_t *total_available_data_length )
  ```

  This function is designed to let you grab pieces of the packet, hence the offset parameter. To get your data you need to pass a pointer to a uint8_t pointer. The function will update your pointer to point to the raw data in the buffer.

Given the above, the receive firmware might look something like this:

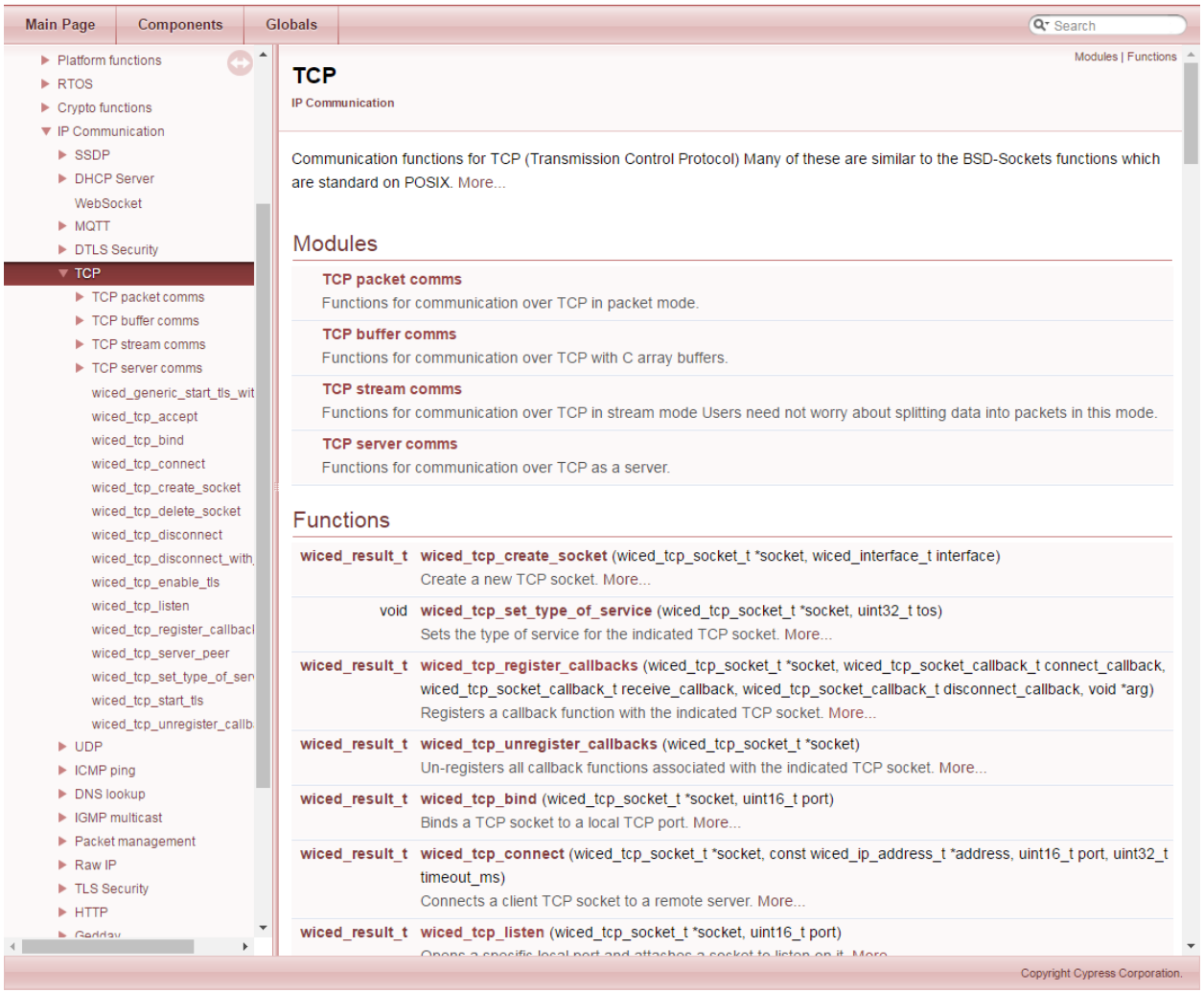
```
while(1)
{
        wiced_packet_t *myPacket;
        uint8_t *myData;
        uint16_t frag_len,avail_len;
        result = wiced_tcp_accept( &socket );   // The thread will suspend until a packet is
        received
        if (result != WICED_SUCCESS)            //  Probably a timeout occurred
                continue;                       //  Skip the rest of this iteration through the loop
        wiced_tcp_receive( &socket, &myPacket, WICED_WAIT_FOREVER );
        wiced_packet_get_data( myPacket, 0, &myData, &frag_len, &avail_len );
        myData[avail_len] =  0;          // add null termination so we can print it
        WPRINT_APP_INFO(("Packet=%s\n", myData));
        wiced_packet_delete( myPacket );
        wiced_tcp_disconnect(&socket);
}
```

The code fragment assumes that it is a short string that you are receiving and it fits in one packet. And obviously, there is no error checking.

Note that the server disconnects the socket once it has received a packet (it does not DELETE the socket, it just disconnects from it). This is commonly done in TCP servers so that socket connections are not maintained when not necessary. Once the client opens another connection, the *wiced_tcp_accept()* call allows the server to receive the next packet.

WICED Socket Documentation

The WICED-SDK provides you a library of functions to do Socket based communication. The WICED documentation on sockets resides in Components → IP Communication → TCP. There are sub-sections for APIs specific for packet communication, buffer communication, stream communication, and server communication. We will mainly deal with packet communications, but the advanced exercises will also cover stream and server APIs.



(Advanced) Symmetric and Asymmetric Encryption: A Foundation

Given that we have the problem that TCP/IP sockets are not encrypted, now what? When you see "HTTPS" in your browser window, the "S" stands for Secure. The reason it is called Secure is that it uses an encrypted channel for all communication. But how can that be? How do you get a secure channel going? And what does it mean to have a secure channel? What is secure? This is be a very complicated topic, as establishing a fundamental mathematical understanding of encryption requires competence in advanced mathematics that is far beyond almost everyone. It is also beyond what there is room to type in this manual. It is also far beyond what I have the ability to explain. But, don't despair. The practical aspects of getting this going are actually pretty simple.

All encryption does the same thing. It takes un-encrypted data, combines it with a key, and runs it through an encryption algorithm to produce encrypted data. The original data is called plain or clear text and the encrypted data is known as "cipher-text". You then transmit the cipher-text over the network. When the other side receives the data it decrypts the cipher-text by combining it with a key, and running the decrypt algorithm to produce clear-text - a.k.a. the original data.

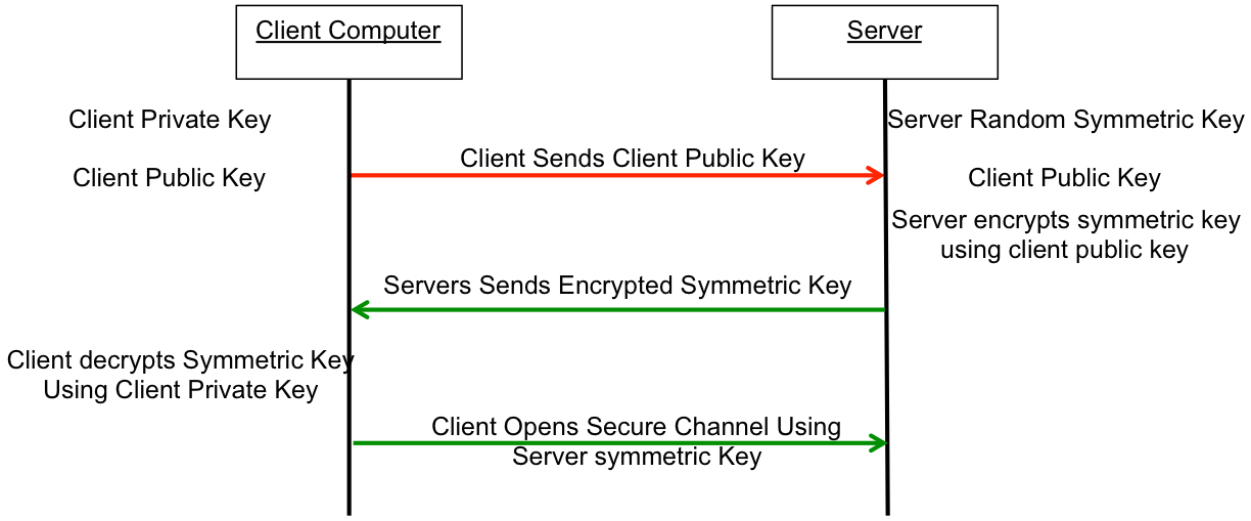
There are two types of encryption schemes, symmetric and asymmetric.

Symmetric means that both sides use the same key. That is, the key that you encrypt with is the same as the key you decrypt with. Examples of this type of encryption include AES and DES. Symmetric encryption is preferred because it is very fast and secure. Unfortunately, both sides need to know the key before you can use it, remember, the encrypt key is exactly the same as the decrypt key. The problem is, if you have never talked before how do you get both sides to know the key? The other problem with symmetric key cryptography is that once the key is lost or compromised, the entire system is compromised as well.

Asymmetric, often called Public Key, encryption techniques use two keys that are mathematically related. The keys are often referred to as the "public" and the "private" keys. The private key can be used to decrypt data that the public key encrypted and vice versa. This is super cool because you can give out your public key to everyone, they can encrypt data using your public key, then only your private key can be used to decrypt it. What is amazing about Asymmetric encryption is that even when you know the Public key you can't figure out the private key (one-way function). The problem with this encryption technique is that it is slow and requires large key storage on the device (usually in FLASH) to store the public key (e.g. 192 bytes for PGP).

What now? The most common technique to communicate is to use public key encryption to pass a private symmetric key which will then be used for the rest of the communication:

- You open an unencrypted connection to a server
- You give out your public key to the server
- The server then creates a random symmetric key
- The server then encrypts its newly created random symmetric key using your public key and sends it back to you
- You use your private key to decrypt the symmetric key
- You open a new channel using symmetric key encryption



This scheme is completely effective against eavesdropping. But, what happens if someone eavesdrops the original public key? That is OK because they won't have the "client private key" required to decrypt the symmetric key. So, what's the hitch? What this scheme doesn't work against is called man-in-the-middle (MIM). An MIM attack works by:

- You open an unencrypted connection to a server [but it really turns out that it is a MIM]
- You send your public key to the MIM
- The MIM opens a channel to the server
- The MIM sends its public key to the server
- The Server encrypts a symmetric key using the MIMs public key and sends it back to the MIM
- The MIM decrypts the symmetric key using its private key
- The MIM sends you the symmetric key encrypted with your public key
- You unencrypt the MIM symmetric key using your private key
- Then you open new channel to the MIM using the symmetric key
- The MIM opens up a channel to the server using the symmetric key

Once the MIM is in the middle it can read all of the traffic. You are only vulnerable to this attack if the MIM gets in the middle on the first transaction. After that, things are secure.

However, the MIM can easily happen if someone gets control of an intermediate connection point in the network - e.g. a Wi-Fi Access Point. There are only two ways to protect against MIM attacks:

- Pre Share the public key (so you are sure you have the right key)
- Use a Certificate Authority (CA)

A CA is a server on the internet that has a huge dictionary of keys. To use a CA, you embed the CA's verified public key in your system (so you can make a secure connection to the CA). Then when you get a key from someone you don't know, you open a secure connection to the CA and it verifies the key that you have matches the key you were sent.

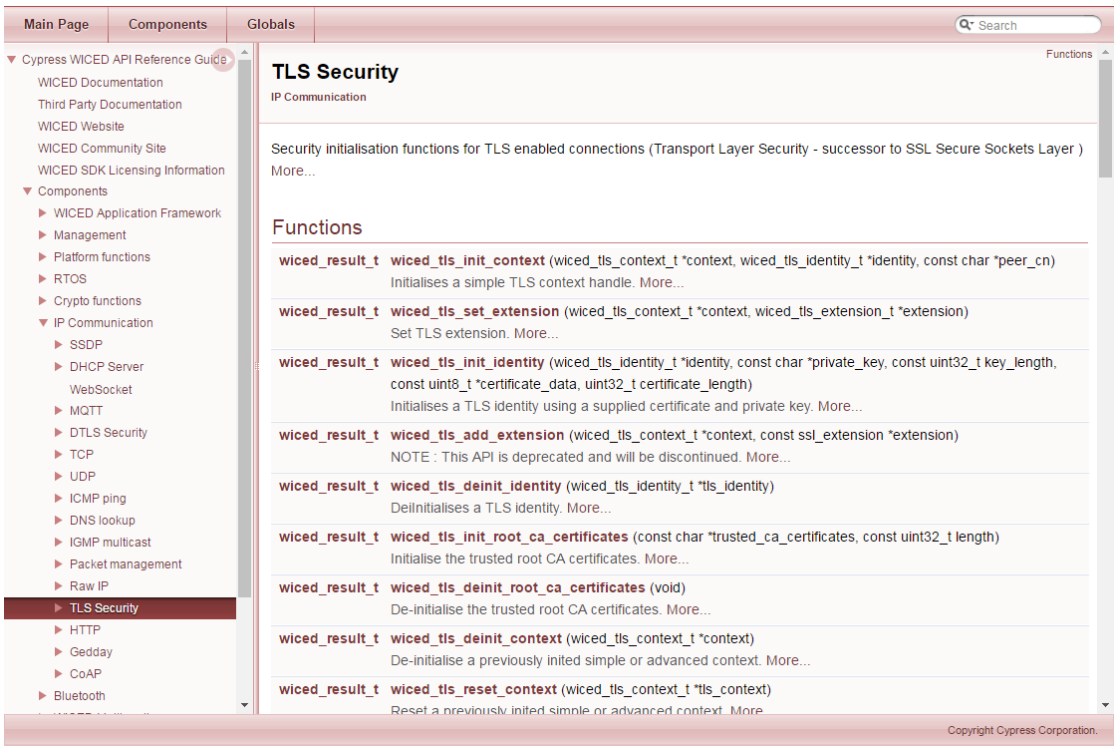
If the MIM sends you its public key then you check with the CA and find out that the MIM public key does not belong to the server that you are trying to connect to, then you know that you are being subjected to an MIM attack. How do you prevent an MIM when talking to a CA? This is done by building in known valid certificates into your program. This morning when I looked at the certificates on my Mac there were 179 built in, valid certificates.

Secure Sockets Layer (SSL) / Transport Layer Security (TLS)

For the key sharing to work, everyone must agree on a standard way to implement the key exchanges and resulting encryption. That method is SSL and and its successor TLS which are two Application Layer Protocols that handle the key exchange described in the previous section and present an encrypted data pipe to the layer above it - i.e. the Web Browser or the WICED device running MQTT. SSL is a fairly heavy (memory and CPU) protocol and has largely been displaced by the lighter weight and newer TLS.

Both of these protocols are generally ascribed to the Application layer but to me it has always felt like it really belongs between the Application and the Transport Layer. TLS is built into WICED and if you give it the keys (from the DCT) when you initialize a connection its operation appears transparent to the layer above it. Several of the application layer protocols that are discussed in the next chapter rest on a TLS connection - i.e. HTTP→TLS→TCP→IP→Wi-Fi Datalink → Wi-Fi → Router → Router→Server Ethernet→Server Datalink→Server IP→Server TCP→TLS→HTTP Server

The documentation for TLS resides in Components→IP Communication→TLS Security.



Exercise(s)

01 Create an IoT client to write data to a server running WWEP when a button is pressed on the client

We have implemented a server using the WICED-SDK running the unsecure version of the WWEP protocol as described above with the following:

- DNS name: wwep.ww101.cypress.com
- IP Address: 198.51.100.3
- Port: 27708

Your application will monitor button presses on the board and will toggle an LED in response to each button press. In addition, your application will connect to the WWEP server and will send the state of the LED each time the button is pressed. For the application:

- The LED characteristic number is 5. That is, the LED state is stored in address 0x05 in the 256 byte register space.
- The "value" of the LED is 0 for OFF and 1 for ON.
- For the device ID, use the 16-bit checksum of your device's MAC address.
    - Hint: See the exercise on printing network information from the "Connecting to Access Points" chapter for an example on getting the MAC address of your device.
    - Hint: to get the checksum, just take the six individual octets (bytes) of the MAC address and add them together.

The steps the application must perform are:

1. Connect to Wi-Fi.
    a. Hint: Use one of your projects from the previous chapter as a starting point.
2. Figure out your device number by adding the MAC bytes together in a uint16_t (effectively a checksum).
3. Use DNS to get the IP address of the server wwep.ww101.cypress.com or hardcode the IP address using INITIALIZER_IPV4_ADDRESS and MAKE_IPV4_ADDRESS).
4. Initialize the LED to OFF.
5. Setup the GPIO to monitor the button.
6. If the button is pressed:
    a. Flip the LED state.
    b. Send data to the server
        i. Format the message you want to send (using *sprint()*)
            1. 'W<device number>05<state>'
            2. Hint: <device number> was calculated above
            3. Hint: <state> is '0000' for OFF and '0001' for ON
        ii. Open a socket to WWEP server (create, bind, connect).
        iii. Initialize a stream
        iv. Write your message to the stream
        v. Flush the stream

vi. Delete the TCP stream (Hint: *wiced_tcp_stream_deinit()*)
vii. Delete the socket

7. Go look at the console of the class WWEP server and make sure that your transactions happened.
8. Hint: Be sure to give any threads you create a large enough stack size (6200 should work).

02 Modify (01) to check the return code

Remember that in the WWEP protocol the server returns a packet with either "A" or and "X" as the first character. For this exercise, read the response back from the server and make sure that your original write occurred properly. Test with a legal and an illegal packet.

Hint: This can be done by calling "*wiced_tcp_stream_read()*"

(Advanced) 03 Modify (02) to use TCP packets (instead of streams)

For this exercise you will repeat exercise 02 to manually create your own packets instead of relying on the stream functions. Follow all of the steps from exercise 02 to set up the exercise, initialize your network connect, and create a socket. Then, for transferring data to the server, instead of creating a stream, use the following steps:

1. Create a transmit packet with 11 bytes:
    a. 'W' (the write command)
    b. 4-bytes of the hex encoded ASCII characters for your device ID
    c. '05' – the two ASCII characters representing the register number of the LED characteristic
    d. '0000' or '0001' – the 4 ASCII characters representing "OFF" and "ON"
    e. Hint: use the *sprint()* function to format the message
2. Send the packet to the socket
3. Delete the transmit packet
4. Read data back from the server
    a. Hint: use wiced_tcp_receive() and wiced_packet_get_data()
5. Print the received data to the terminal.
6. Delete the receive packet
7. Delete the Socket
8. Go look at the console of the class WWEP server and make sure that your transactions happened.

(Advanced) 04 Implement the server side of the unsecure WWEP protocol that can handle one connection at a time (using TCP packet reads)

Hint: Look at the documentation in the TCP server comms section of the API guide.

Hint: use a linked list for the database so that it will start out with no entries and will then grow as data is stored.

Hint: The WICED library has a linked list utility that can be found in the libraries/utilities directory. You can simply include it using #include "linked_list.h" which also provide the API documentation.

(Advanced) 05 Implement the server side of the unsecure WWEP protocol that can handle one connection at a time (using TCP callbacks)

Hint: Look at the function wiced_tcp_register_callbacks. You will need callback functions for a socket connection, for new data received from the client, and for a socket disconnection.

Hint: You may want to increase the transmit and receive buffer pool sizes to a value of 6 so that simultaneous connections do not result in an overflow of the buffers on the server. These are set in the file which can be found at "43xxx_Wi-Fi\ \platforms\BCM94343W_AVN\BCM94343W_AVN.mk".

The lines of interest are:

```
GLOBAL_DEFINES +=      TX_PACKET_POOL_SIZE=2 \
                       RX_PACKET_POOL_SIZE=2
```

Note that this file is used for every project in the workspace using this platform so it will affect all of your projects. In our case, we are not using a lot of SRAM but keep in mind that each packet is about 1500 bytes so increasing the packet pool is expensive in terms of SRAM use.

(Advanced) 06 Modify (05) to handle multiple connections at a time

(Advanced) 07 Modify (03) to use TLS security on port 40508

(Advanced) 08 Implement the server side of the secure WWEP protocol

Further Reading

[1] RFC1700 – “Assigned Numbers”; Internet Engineering Task Force (IETF) - https://www.ietf.org/rfc/rfc1700.txt

[3] IANA Service Name and Port Registry - http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml