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:

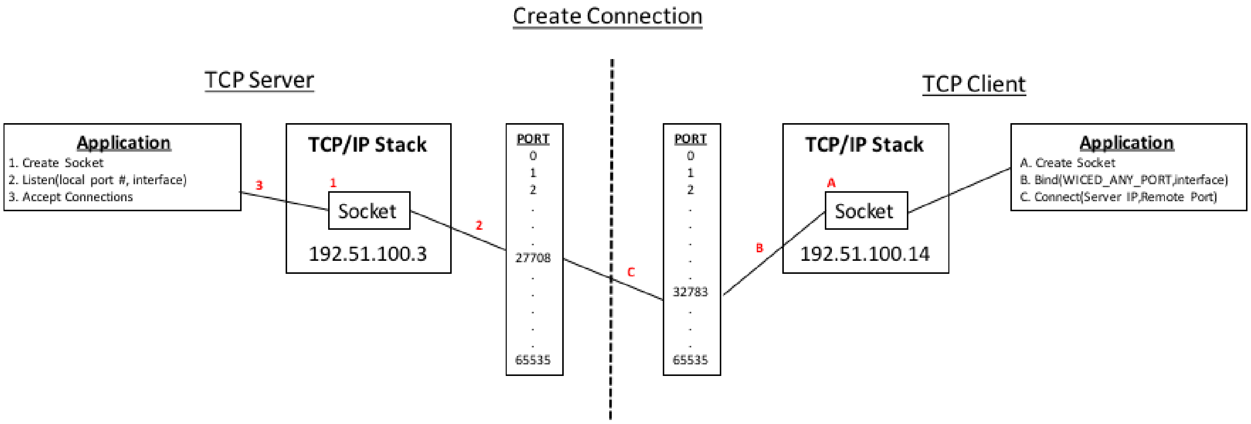
- “W0FAC0B1234” would write a value of 0x1234 to register 0x0B for device with an ID of 0x0FAC. The server would then respond with “A0FAC0B1234”.
- “W01234” is an illegal packet and the server would respond with “X”.
- “R0FAC0B” is a read of register 0x0B for a Device ID with an ID of 0x0FAC”. In this 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. Listen to the socket to WWEP server TCP port 27708 by calling:

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

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);
```

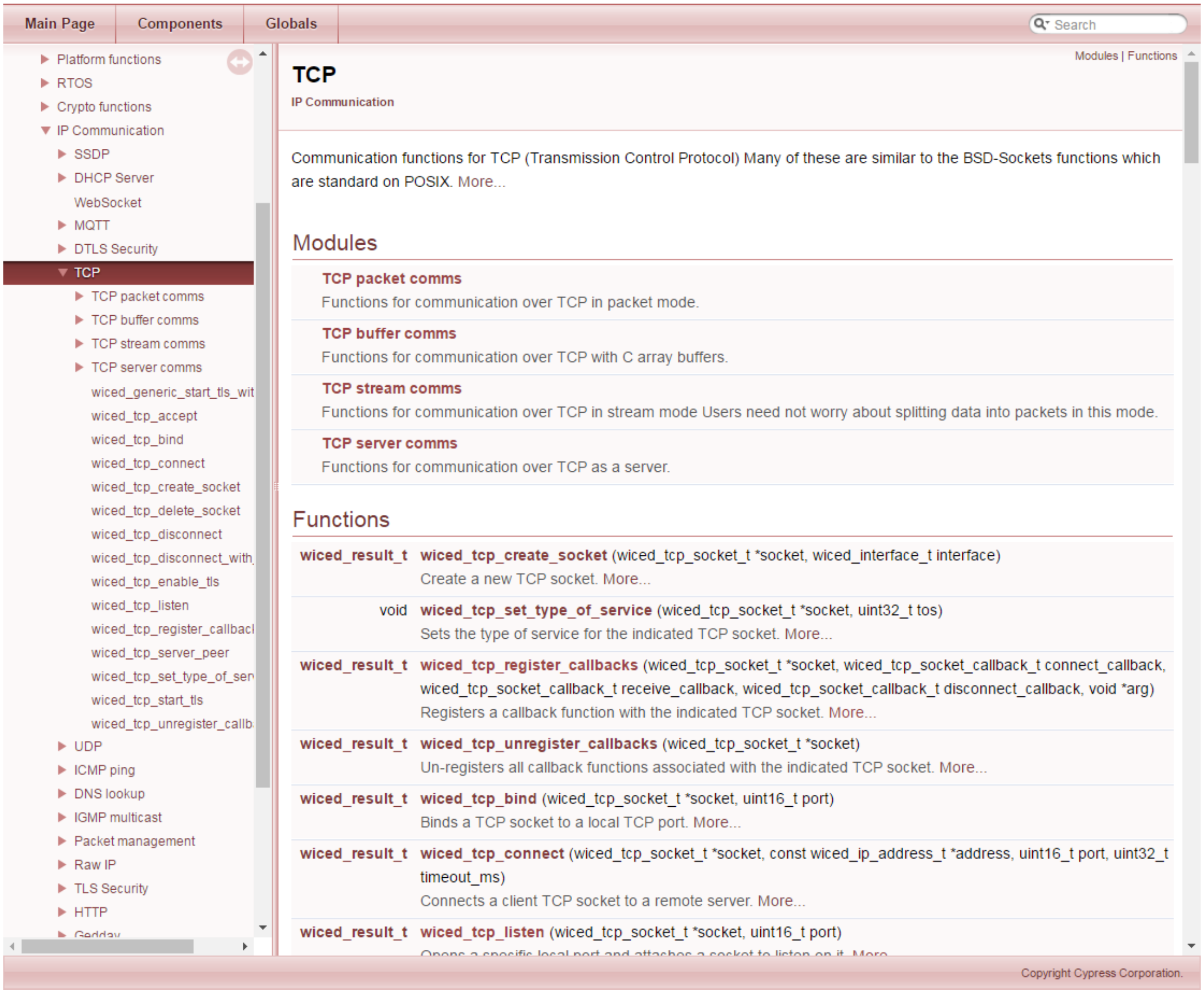
Behind the scenes, reading and writing via streams uses uniform sized packets. The stream functions in the WICED SDK hide the management of each of these packets from you so you can focus on the higher levels of your application. However, if you desire more control over the communication you can use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

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.



TCP

IP Communication

Communication functions for TCP (Transmission Control Protocol) Many of these are similar to the BSD-Sockets functions which are standard on POSIX. More...

Modules

TCP packet comms
Functions for communication over TCP in packet mode.

TCP buffer comms
Functions for communication over TCP with C array buffers.

TCP stream comms
Functions for communication over TCP in stream mode Users need not worry about splitting data into packets in this mode.

TCP server comms
Functions for communication over TCP as a server.

Functions

wiced_result_t **wiced_tcp_create_socket** (wiced_tcp_socket_t *socket, wiced_interface_t interface)
Create a new TCP socket. More...

void **wiced_tcp_set_type_of_service** (wiced_tcp_socket_t *socket, uint32_t tos)
Sets the type of service for the indicated TCP socket. More...

wiced_result_t **wiced_tcp_register_callbacks** (wiced_tcp_socket_t *socket, wiced_tcp_socket_callback_t connect_callback, wiced_tcp_socket_callback_t receive_callback, wiced_tcp_socket_callback_t disconnect_callback, void *arg)
Registers a callback function with the indicated TCP socket. More...

wiced_result_t **wiced_tcp_unregister_callbacks** (wiced_tcp_socket_t *socket)
Un-registers all callback functions associated with the indicated TCP socket. More...

wiced_result_t **wiced_tcp_bind** (wiced_tcp_socket_t *socket, uint16_t port)
Binds a TCP socket to a local TCP port. More...

wiced_result_t **wiced_tcp_connect** (wiced_tcp_socket_t *socket, const wiced_ip_address_t *address, uint16_t port, uint32_t timeout_ms)
Connects a client TCP socket to a remote server. More...

wiced_result_t **wiced_tcp_listen** (wiced_tcp_socket_t *socket, uint16_t port)
Opens a specific local port and attaches a socket to listen on it. More...

Copyright Cypress Corporation.

Security: 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 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.

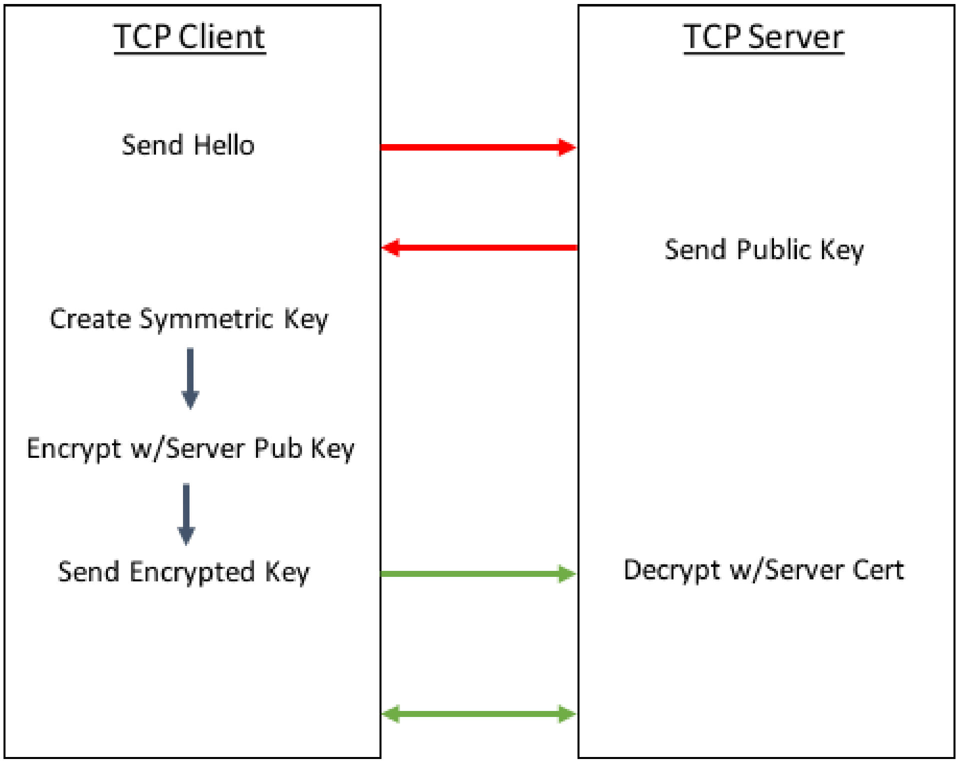
There are two types of encryption schemes, symmetric and asymmetric.

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, someone else 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 is then used for the rest of the communication:

- You open an unencrypted connection to a server
- The server sends you its public key
- The client creates a random symmetric key
- The client encrypts its newly created random symmetric key using the server public key and sends it back to server
- The Server decrypts the symmetric key using its private 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 "server 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]
- The MIM opens a channel to the Server
- The Server sends its public key to the MIM
- The MIM then sends its public key to the Client
- The Client creates a random symmetric key, encrypts it with the MIM Public key (which it thinks is really the Server Public Key)
- The Client sends it to the MIM (which it thinks is the server)
- The MIM unencrypts the symmetric key, then rencrypts using the Server Public Key
- The MIM opens up a channel to the server using the rencrypted symmetric key

Once the MIM is in the middle it can read all 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 is only one good way to protect against MIM attacks, specifically by using a Certificate Authority (CA). There are Root CAs as well as Intermediate CAs, which we will discuss in a minute. The process works as follows:

When you connect to an unknown server it will send a Certificate (in X.509 Format – more on that later) to you that contains public keys for the Certificate Authorities (Root CA and/or one or more Intermediate CA) and the server itself. If you recognize either the Intermediate CA Public Key or the Root CA Public Key then you have validated the connection. This morning when I looked at the certificates on my Mac there were 179 built in, valid Root certificates.

The last question is, "How do you know that the Certificate has not been tampered with?" The answer is that the CA provides you with a "signed" certificate. The process of signing uses an encrypted Cryptographic Hash which is essentially a fancy checksum. With a simple checksum you just add up all of the values in a file mod-256 so you will end up with a value between 0-255 (or mod-2^16 or mod-2^32). Even with big checksums (2^32) it is easy to come up with two input files that have the same checksum i.e. there is a collision. These collisions can lead to a checksum being falsified. To prevent collisions, there are several algorithms including Secure Hash Algorithm (SHA) and Message Digest (MD5) which for all practical purposes create a unique output for every known input. The output of a Cryptographic Hash is commonly called a Digest (just a short string of bytes). Once the Digest is encrypted, you then have the "Signature" for the certificate.

Let's say you need to get a signed certificate from a CA (for example to set up a secure web site). The process is as follows:

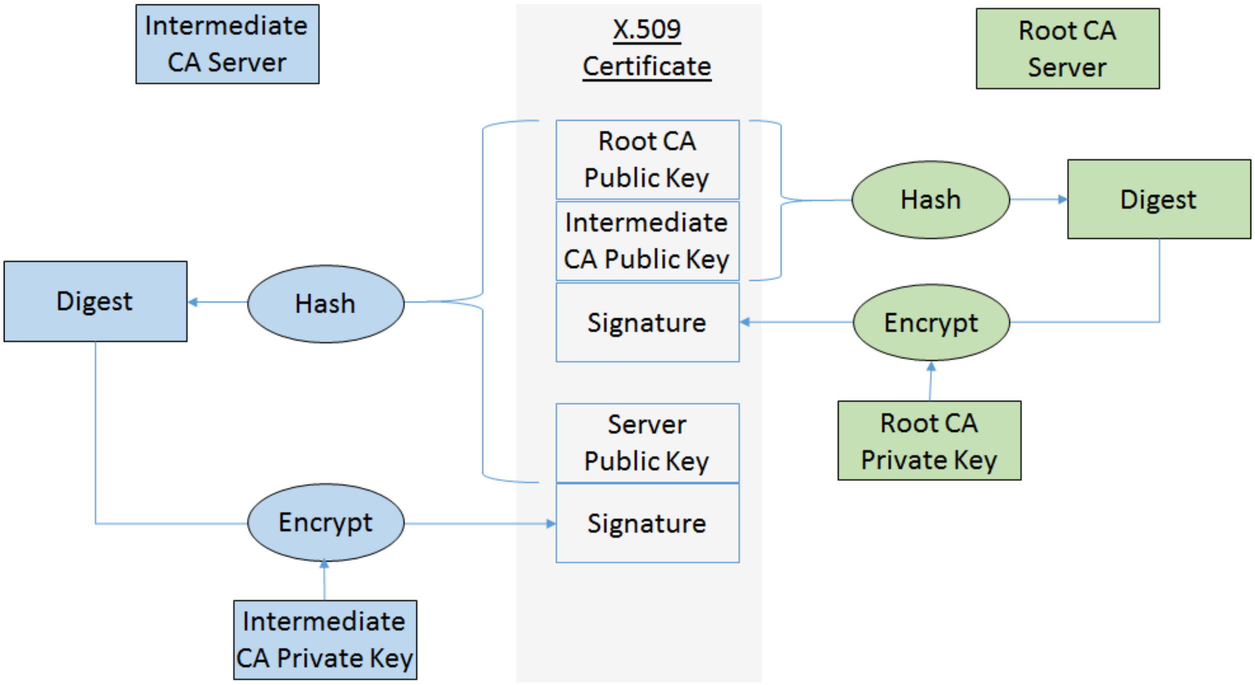
- Take your public key and send it to the CA that is providing the signed certificate.
- The CA will take its public key and your public key and will hash them to create a Digest
- The CA then encrypts the Digest with its private key. The result is the signature. Because the CA is using its private key, it is the only one capable of creating that particular encryption but anyone can decrypt it (using the CA's public key).
- The CA sends you the certificate which has the public keys and the signature embedded in it along with lots of other information.

Note that this process can be done hierarchically to create a chain of signed certificates. That is, the server gets its identity validated by an intermediate authority who, in turn, gets its identity validated by a "higher" intermediate authority or by a root authority. Therefore, any certificate eventually chains to a Root CA.

When a client opens a connection to an unknown server, the server sends its certificate.

When the client gets the certificate, it follows this process:

- Hash the public keys for the server and for the CA – this reproduces the digest from the CA.
- Unencrypt the signature from the CA using the CA's public key to recover the digest.
- Compare your calculated digest with the unencrypted digest. If they match then nothing has been changed (the certificate has not been tampered with).
- Compare the CA's public key (from the certificate) against your known list (built into your firmware). If you recognize the key then you assume that the CA has "signed" for the server you are talking to and that it can be trusted.

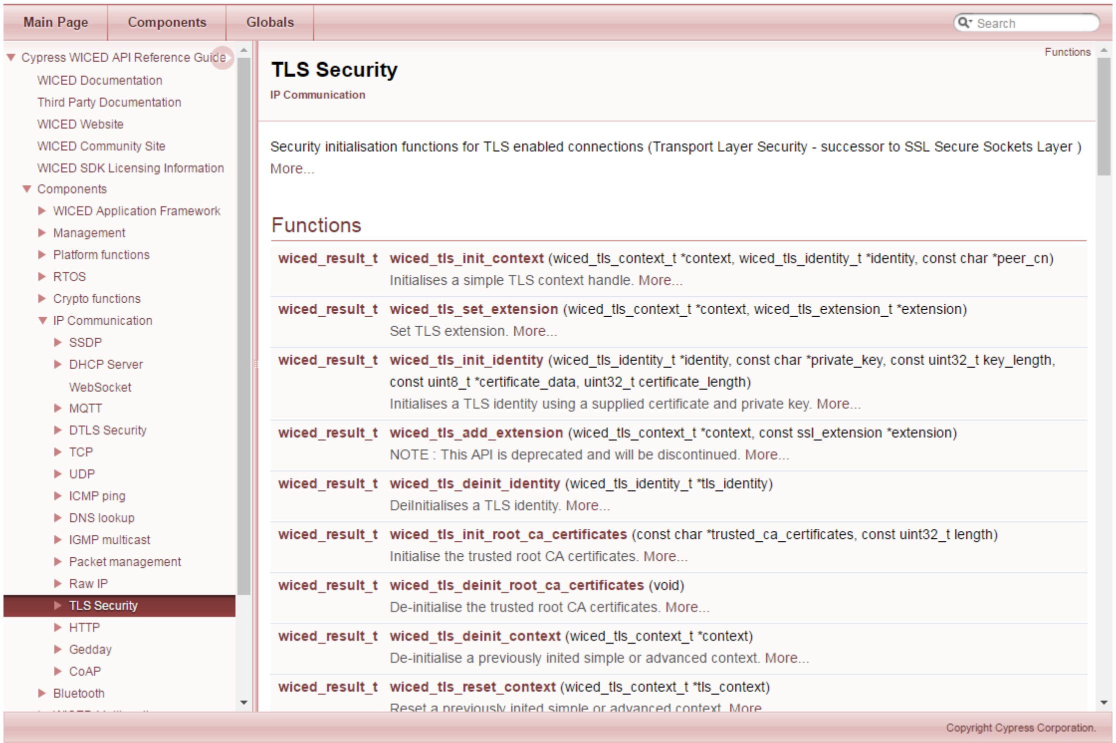


Secure Sockets Layer (SSL) / Transport Layer Security (TLS)

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 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, more secure, TLS (now on version 1.2).

Both 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 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→WEB→Router→Server Ethernet→Server Datalink→Server IP→Server TCP→TLS→HTTP Server

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



TLS Security

IP Communication

Security initialisation functions for TLS enabled connections (Transport Layer Security - successor to SSL Secure Sockets Layer) More...

Functions

wiced_result_t **wiced_tls_init_context** (wiced_tls_context_t *context, wiced_tls_identity_t *identity, const char *peer_cn)
Initialises a simple TLS context handle. More...

wiced_result_t **wiced_tls_set_extension** (wiced_tls_context_t *context, wiced_tls_extension_t *extension)
Set TLS extension. More...

wiced_result_t **wiced_tls_init_identity** (wiced_tls_identity_t *identity, const char *private_key, const uint32_t key_length, const uint8_t *certificate_data, uint32_t certificate_length)
Initialises a TLS identity using a supplied certificate and private key. More...

wiced_result_t **wiced_tls_add_extension** (wiced_tls_context_t *context, const ssl_extension *extension)
NOTE : This API is deprecated and will be discontinued. More...

wiced_result_t **wiced_tls_deinit_identity** (wiced_tls_identity_t *tls_identity)
DeInitialises a TLS identity. More...

wiced_result_t **wiced_tls_init_root_ca_certificates** (const char *trusted_ca_certificates, const uint32_t length)
Initialise the trusted root CA certificates. More...

wiced_result_t **wiced_tls_deinit_root_ca_certificates** (void)
De-initialise the trusted root CA certificates. More...

wiced_result_t **wiced_tls_deinit_context** (wiced_tls_context_t *context)
De-initialise a previously inited simple or advanced context. More...

wiced_result_t **wiced_tls_reset_context** (wiced_tls_context_t *tls_context)
Reset a previously inited simple or advanced context. More

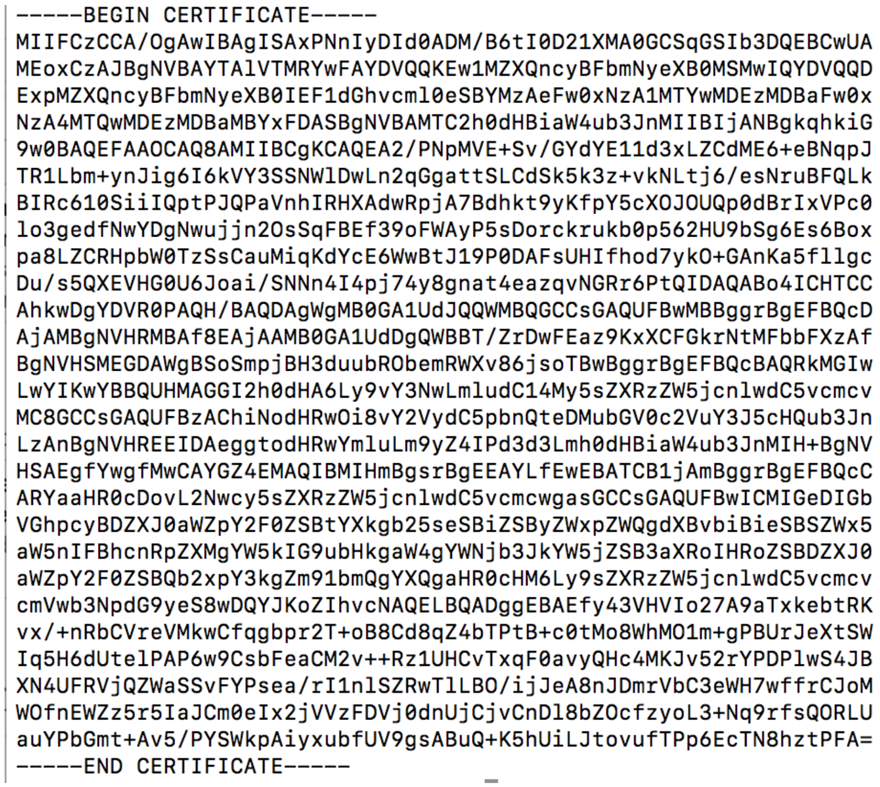
X.509 Certificates

Basics

To create a TLS connection, you need to have a root certificate to verify the identity of the server you are talking with. When a TLS connection is opened, the server will send its signed certificate which you will then verify against a known good certificate (that must be programmed in your firmware). The certificates will be in one of several X.509 formats. The two most common formats are “DER” which is a binary format, and “PEM” which is an ASCII format (and is the one that WICED uses).

In WICED it is optional for a client to verify the certificate. You can register the servers CAs by calling wiced_tls_init_root_ca_certificates. It is also possible for the server to verify the client using the same methodology. That is, the server can have the root certificate for the client built and ask that it be verified when opening a TLS connection.

The screenshot below is a PEM certificate:



```
-----BEGIN CERTIFICATE-----
MIIFCzCCA/OgAwIBAgISAxPNnIyDId0ADM/B6tI0D21XMA0GCSqGSIb3DQEBCwUA
MEoxCzAJBgNVBAYTAlVTMRYwFAYDVQQKEw1MZXQncyBFbmNyeXB0MSMwIQYDVQQD
ExpMZXQncyBFbmNyeXB0IEF1dGhvcml0eSBYMzAeFw0xNzA1MTYwMDEzMDBaFw0x
NzA4MTQwMDEzMDBaMBYxFDASBgNVBAMTC2h0dHBiaW4ub3JnMIIBIjANBgkqhkiG
9w0BAQEFAAOCAQ8AMIIBCgKCAQEA2/PNpMVE+Sv/GYdYE11d3xLZCdME6+eBNqpJ
TR1Lbm+ynJig6I6kVY3SSNWlDwLn2qGgattSLCdSk5k3z+vkNLtj6/esNruBFQLk
BIRc610SiiIQptPJQPaVnhIRHXAdwRpjA7Bdhkt9yKfpY5cXOJOUQp0dBrIxVPc0
lo3gedfNwYDgNwujjn2OsSqFBEf39oFWAyP5sDorckrukb0p562HU9bSg6Es6Box
pa8LZCRHpbW0TzSsCauMiqKdYcE6WwBtJ19P0DAFsUHIfhod7ykO+GAnKa5fllgc
Du/s5QXEVHG0U6Joai/SNNn4I4pj74y8gnat4eazqvNGRr6PtQIDAQABo4ICHTCC
AhkwDgYDVR0PAQH/BAQDAgWgMB0GA1UdJQQWMBQGCCsGAQUFBwMBBggrBgEFBQcD
AjAMBgNVHRMBAf8EAjAAMB0GA1UdDgQWBBT/ZrDwFEaz9KxXCFGkrNtMFbbFXzAf
BgNVHSMEGDAWgBSoSmpjBH3duubRObemRWXv86jsoTBwBggrBgEFBQcBAQRkMGIw
LwYIKwYBBQUHMAGGI2h0dHA6Ly9vY3NwLmludC14My5sZXRzZW5jcnlwdC5vcmcv
MC8GCCsGAQUFBzAChiNodHRwOi8vY2VydC5pbnQteDMubGV0c2VuY3J5cHQub3Jn
LzAnBgNVHREEIDAeggtodHRwYmluLm9yZ4IPd3d3Lmh0dHBiaW4ub3JnMIH+BgNV
HSAEgfYwgfMwCAYGZ4EMAQIBMIHmBgsrBgEEAYLfEwEBATCB1jAmBggrBgEFBQcC
ARYaaHR0cDovL2Nwcy5sZXRzZW5jcnlwdC5vcmcwgasGCCsGAQUFBwICMIGeDIGb
VGhpcyBDZXJ0aWZpY2F0ZSBtYXkgb25seSBiZSByZWxpZWQgdXBvbiBieSBSZWx5
aW5nIFBhcnRpZXMgYW5kIG9ubHkgaW4gYWNjb3JkYW5jZSB3aXRoIHRoZSBDZXJ0
aWZpY2F0ZSBQb2xpY3kgZm91bmQgYXQgaHR0cHM6Ly9sZXRzZW5jcnlwdC5vcmcv
cmVwb3NpdG9yeS8wDQYJKoZIhvcNAQELBQADggEBAEfy43VHVIo27A9aTxkebtRK
vx/+nRbCVreVMkwCfqgbpr2T+oB8Cd8qZ4bTPtB+c0tMo8WhMO1m+gPBUrJeXtSW
Iq5H6dUtelPAP6w9CsbFeaCM2v++Rz1UHCvTxqF0avyQHc4MKJv52rYPDPlwS4JB
XN4UFRVjQZWaSSvFYPsea/rI1nlSZRwTlLBO/ijJeA8nJDmrVbC3eWH7wffrCJoM
WOfnEWZz5r5IaJCm0eIx2jVVzFDVj0dnUjCjvCnDl8bZOcfzyoL3+Nq9rfsQORLU
auYPbGmt+Av5/PYSWkpAiyxubfUV9gsABuQ+K5hUiLJtovufTPp6EcTN8hztPFA=
-----END CERTIFICATE-----
```

X.509 certificates contain an insane amount of information and they have a bunch of options. However, they generally contain:

- The site's public key
- An intermediate authority's public key
- The root authority's public key
- The valid DNS domains for this certificate
- The expiration date of the certificate
- One or more secure signatures that let you verify the authenticity of the message

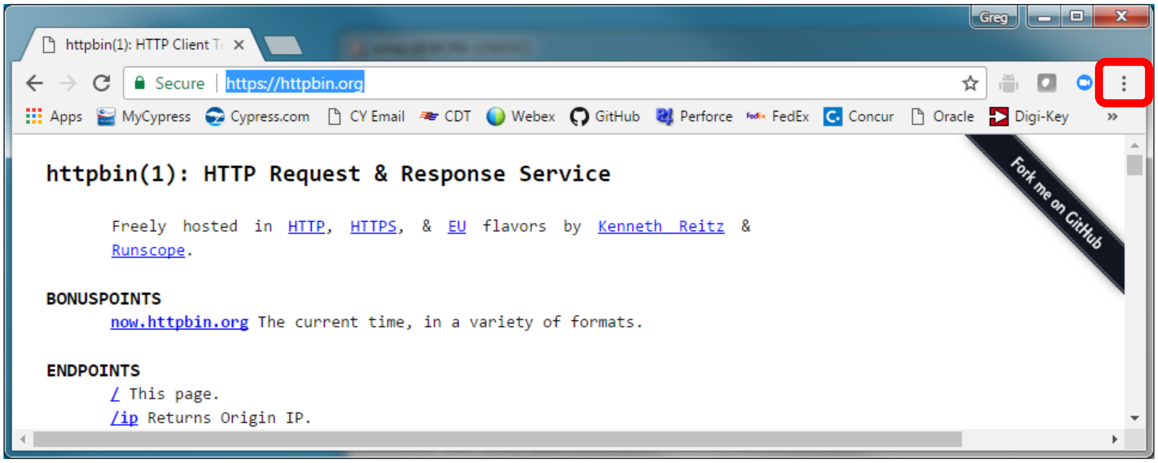
Downloading Certificates

You can get the root or intermediate certificate for a website from a browser. In the examples below, we will use https://httpbin.org as the site for which we want to retrieve the certificate.

Chrome

In Chrome, navigate to the site you are interested in (https://httpbin.org), and then follow these steps to download the certificate:

1. On the upper right corner, click the three dots and select "More Tools -> Developer Tools".



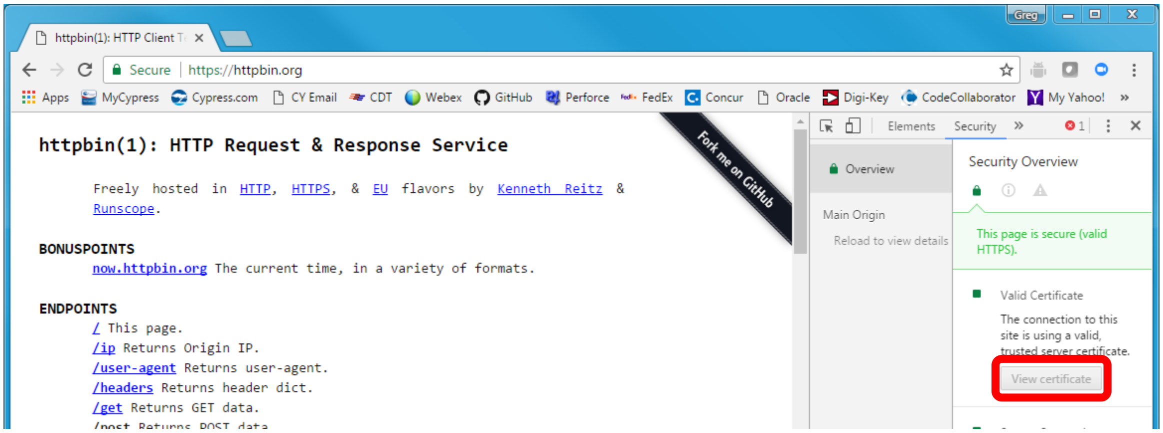
httpbin(1): HTTP Request & Response Service

Freely hosted in HTTP, HTTPS, & EU flavors by Kenneth Reitz & Runscope.

BONUSPOINTS
now.httpbin.org The current time, in a variety of formats.

ENDPOINTS
/ This page.
/ip Returns Origin IP.

2. Click on "View certificate".



httpbin(1): HTTP Request & Response Service

Freely hosted in HTTP, HTTPS, & EU flavors by Kenneth Reitz & Runscope.

BONUSPOINTS
now.httpbin.org The current time, in a variety of formats.

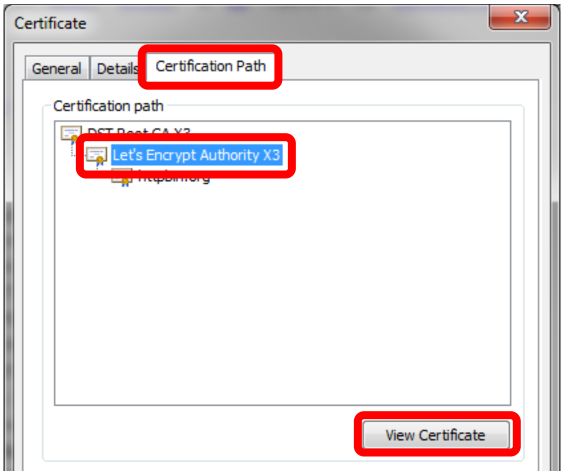
ENDPOINTS
/ This page.
/ip Returns Origin IP.
/user-agent Returns user-agent.
/headers Returns header dict.
/get Returns GET data.
/post Returns POST data.

This page is secure (valid HTTPS).

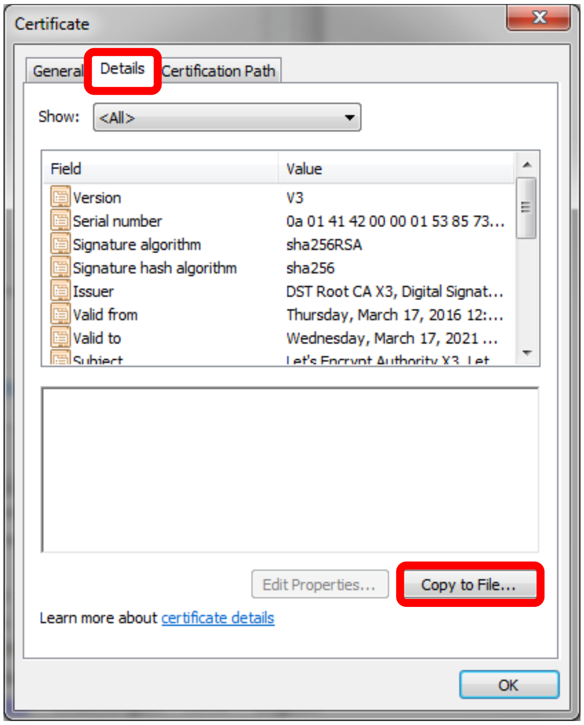
Valid Certificate

The connection to this site is using a valid, trusted server certificate.

3. Click on the “Certification Path” tab. In this case, you can see that the certificate is issued by (i.e. signed by) “Let’s Encrypt Authority X3” and that the root certificate is “DST ROOT CA X3”. To make the TLS connection to https://httpbin.org you will need either the signed intermediate certificate or the root certificate, not the httpbin.org certificate. Therefore, click on either the root or intermediate certificate and then click on “View Certificate” so that you are looking at (and saving) the signed certificate.



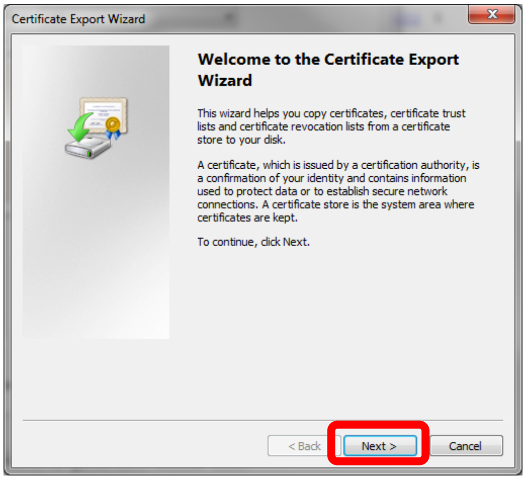
4. You will now have another window open showing information for the signed certificate. Click on the “Details” tab and then on “Copy to File…” to open the Certificate Export Wizard.



Show: <All>

Field	Value
Version	V3
Serial number	0a 01 41 42 00 00 01 53 85 73...
Signature algorithm	sha256RSA
Signature hash algorithm	sha256
Issuer	DST Root CA X3, Digital Signat...
Valid from	Thursday, March 17, 2016 12:...
Valid to	Wednesday, March 17, 2021 ...
Subject	Let's Encrypt Authority X3, Let

5. From the Certificate Export Wizard, select “Base-64 encoded X.509 (.CER)” to allow you to save the certificate in the ASCII PEM format.



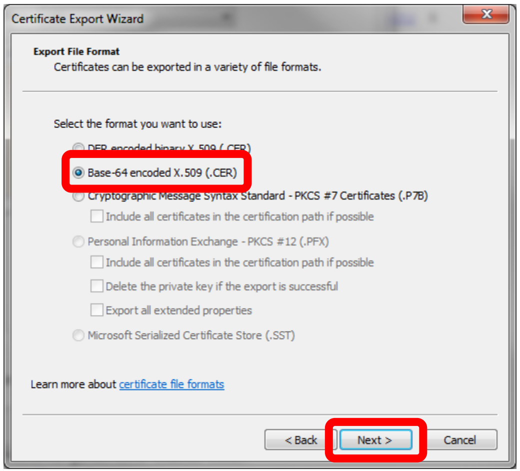
Certificate Export Wizard

Welcome to the Certificate Export Wizard

This wizard helps you copy certificates, certificate trust lists and certificate revocation lists from a certificate store to your disk.

A certificate, which is issued by a certification authority, is a confirmation of your identity and contains information used to protect data or to establish secure network connections. A certificate store is the system area where certificates are kept.

To continue, click Next.



Certificate Export Wizard

Export File Format

Certificates can be exported in a variety of file formats.

Select the format you want to use:

○ DER encoded binary X.509 (.CER)

● Base-64 encoded X.509 (.CER)

○ Cryptographic Message Syntax Standard - PKCS #7 Certificates (.P7B)

□ Include all certificates in the certification path if possible

○ Personal Information Exchange - PKCS #12 (.PFX)

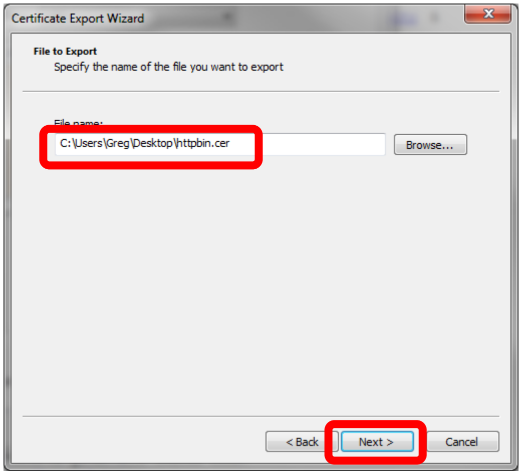
□ Include all certificates in the certification path if possible

□ Delete the private key if the export is successful

□ Export all extended properties

○ Microsoft Serialized Certificate Store (.SST)

Learn more about certificate file formats



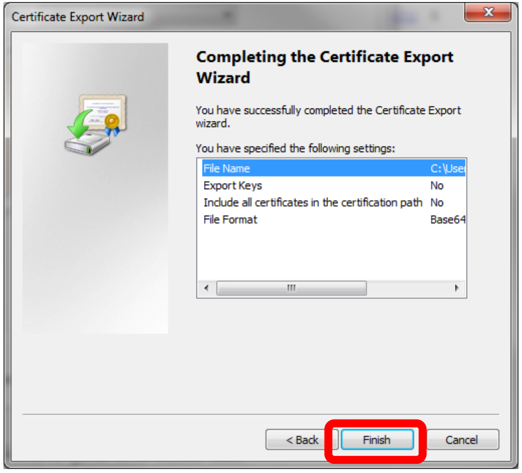
Certificate Export Wizard

File to Export

Specify the name of the file you want to export

File name:

C:\Users\Greg\Desktop\httpbin.cer



Certificate Export Wizard

Completing the Certificate Export Wizard

You have successfully completed the Certificate Export wizard.

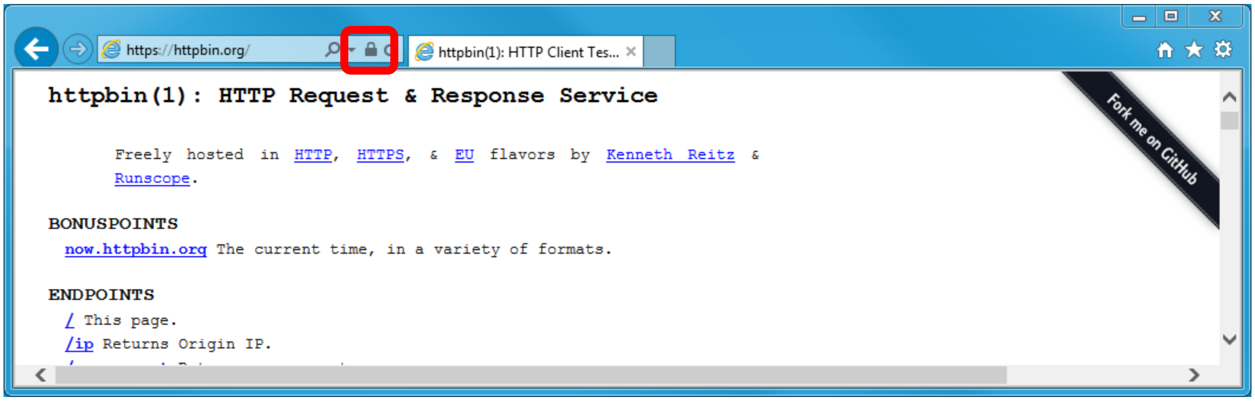
You have specified the following settings:

File Name	C:\Use
Export Keys	No
Include all certificates in the certification path	No
File Format	Base64

6. Once you have saved the certificate you can double-click on it to see the certificate information again, or you can open it with a text viewer to see the actual ASCII code of the certificate.

Internet Explorer

In Internet Explorer, navigate to the site you are interested in (https://httpbin.org), click on the little padlock to the right of the URL and select View Certificates. Once you have the Certificate viewer open you can follow the same steps as for Chrome to save the certificate.



httpbin(1): HTTP Request & Response Service

Freely hosted in HTTP, HTTPS, & EU flavors by Kenneth Reitz & Runscope.

BONUSPOINTS

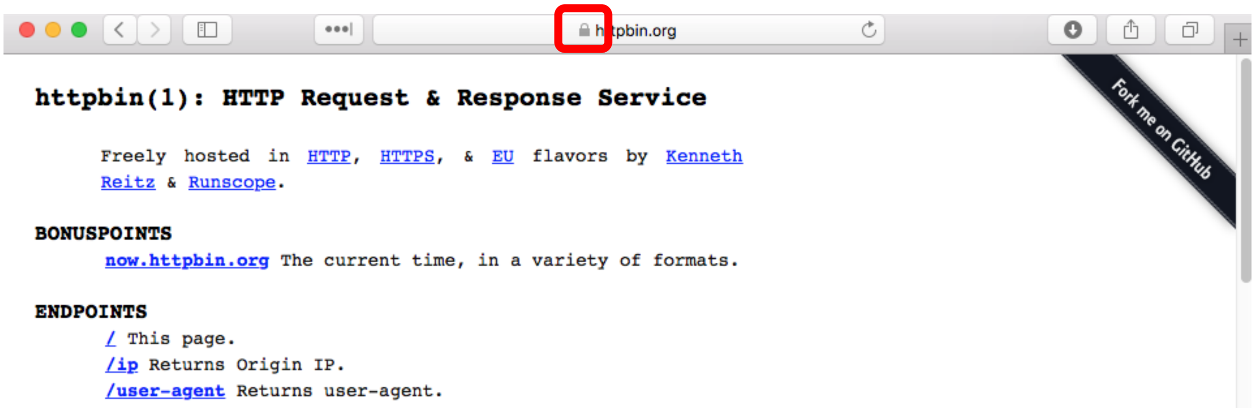
now.httpbin.org The current time, in a variety of formats.

ENDPOINTS

/ This page.
/ip Returns Origin IP.

Safari

In Safari navigate to the site you are interested in (https://httpbin.org), and click on the little padlock right next to URL. This will bring up the certificate browser.



httpbin(1): HTTP Request & Response Service

Freely hosted in HTTP, HTTPS, & EU flavors by Kenneth Reitz & Runscope.

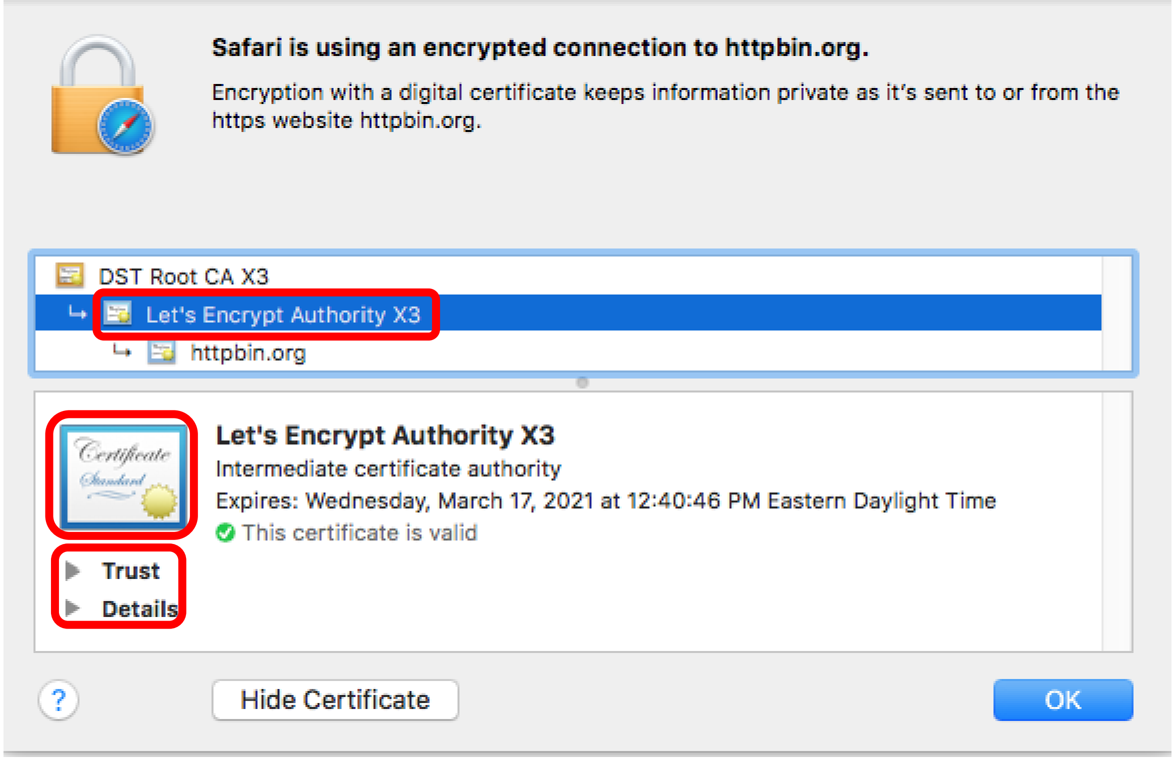
BONUSPOINTS

now.httpbin.org The current time, in a variety of formats.

ENDPOINTS

/ This page.
/ip Returns Origin IP.
/user-agent Returns user-agent.

Once you are in the certificate browser you can examine the certificate by clicking the little down arrows next to “trust” and “details”. In this case, you can see that the certificate is issued by (i.e. signed by) “Let’s Encrypt Authority X3” and that the root certificate is “DST ROOT CA X3”. To make the TLS connection to https://httpbin.org you will need either the intermediate certificate or the root certificate, not the httpbin.org certificate. Therefore, click on the certificate that you want to download and then click-drag-drop it to your desktop.



Safari is using an encrypted connection to httpbin.org.

Encryption with a digital certificate keeps information private as it’s sent to or from the https website httpbin.org.

DST Root CA X3
↳ Let's Encrypt Authority X3
↳ httpbin.org

Let's Encrypt Authority X3
Intermediate certificate authority
Expires: Wednesday, March 17, 2021 at 12:40:46 PM Eastern Daylight Time
✓ This certificate is valid

Certificates downloaded from Safari will be in the binary format called “DER” which Apple gives the extension of “.cer”. You can now examine the content of the certificate from the command line using “openssl” which is built into the Mac OS. For example, you can look at the “Let’s Encrypt Authority X3” by running:

```
openssl x509 -in Let’s Encrypt Authority X3.cer -inform der -text -noout
```

You can also examine the certificate by pasting it to https://www.sslshopper.com/certificate-decoder.html

To use the certificate in WICED you will need to transform it into the ASCII PEM format which can be done by running:

```
openssl x509 -inform der -in Let’s Encrypt Authority X3.cer -out Let’s Encrypt Authority X3.pem
```

You can view the PEM formatted certificate by running:

```
openssl x509 -in Let’s Encrypt Authority X3.cer –text -noout
```

You can also decode a certificate at https://www.sslshopper.com/certificate-decoder.html

Creating Your Own Certificates

You can create your own “self-signed” certificates by running openssl. This is built into MacOS and Linux. For Windows, it can be downloaded and can run in Cygwin. For example, running the command below will create:

- A new public/private key pair. The private key will be saved in key.pem and the public key will reside in the certificate.
- A new signed certificate called “certificate.pem”. The root authority will be the server (i.e. it is signed by itself). This will cause web browser to complain as the certificate will not be present in your browser meaning that it will be untrusted.

```
openssl req -newkey rsa:2048 -nodes -keyout key.pem -x509 -days 365 -out certificate.pem
```

Using Certificates in WICED

Once you have a certificate, there are three ways that you can access it from your device. Each of these is discussed separately below.

Method 1: Storing and using certificates from the DCT

It is possible to have the WICED make system install your certificate into the DCT automatically. To do this you need to:

1. Convert your certificate to PEM format if it is not already in that format, then store it in the directory *resources/apps/yourapp/*
2. Assuming yourapp is called *httpbin_org* and your certificate file is called *ca.pem*, you would add to your Makefile the line:

```
CERTIFICATE := $(SOURCE_ROOT)resources/apps/httpbin_org/ca.pem
```

3. Then you can load the security section of the DCT into RAM and use it to initialize the root certificate for the TLS connection.

```
platform_dct_security_t *dct_security;

WPRINT_APP_INFO(( "Read the certificate Key from DCT\n" ));
result = wiced_dct_read_lock( (void**) &dct_security, WICED_FALSE, DCT_SECURITY_SECTION, 0, sizeof(
*dct_security ) );

if ( result != WICED_SUCCESS )
{
   WPRINT_APP_INFO(("Unable to lock DCT to read certificate\n"));
   return;
 }

WPRINT_APP_INFO(("Certificate Length = %d\n",strlen(dct_security->certificate)));
WPRINT_APP_INFO(("Certificate =%s",dct_security->certificate));

result = wiced_tls_init_root_ca_certificates(dct_security->certificate,strlen(dct_security->certificate) );
```

```
if ( result != WICED_SUCCESS )
{
   WPRINT_APP_INFO( ( "Error: Root CA certificate failed to initialize: %u\n", result) );
   return;
}
```

Method 2: Storing and using certificates in the Resources filesystem

WICED can load files into a flash filesystem that resides after the DCT. You can then access those files from your firmware. You can store and use certificates in that filesystem by doing the following:

1. Store the file to the resources directory *resources/apps/yourapp/*.
2. Again, assuming an app name of *httpbin_org* and certificate file name of *ca.pem*, add the path to the certificate as a RESOURCES tag in your Makefile. For example:

   ```
   $(NAME)_RESOURCES  := apps/httpbin_org/ca.pem
   ```

3. In the project's .c file, add:

   ```
   #include "resources.h"
   ```

4. Load the file into RAM using the API resource_get_readonly_buffer.
   a. Note how paths are specified with "_DIR_" instead of "/".
   b. Note how the file extension is separated using "_" instead of "."
5. Initialize the certificate using the API wiced_tls_init_root_ca_certificates.

   For example:

```
   /* Initialize the root CA certificate */
   uint32_t size_out;
   resource_get_readonly_buffer( &resources_apps_DIR_httpbin_org_DIR_ca_pem, 0, 2048, &size_out, (const void
**) &httpbin_root_ca_certificate );
   result = wiced_tls_init_root_ca_certificates( httpbin_root_ca_certificate, size_out );
   if ( result != WICED_SUCCESS )
   {
      WPRINT_APP_INFO( ( "Error: Root CA certificate failed to initialize: %u\n", result) );
      return;
   }
```

Method 3: Storing and using certificates from “char arrays”

You can embed the certificate into a static const char array in the source code by editing the PEM file to have “\r\n” at the end of the lines. You certificate will look like this in the source file:

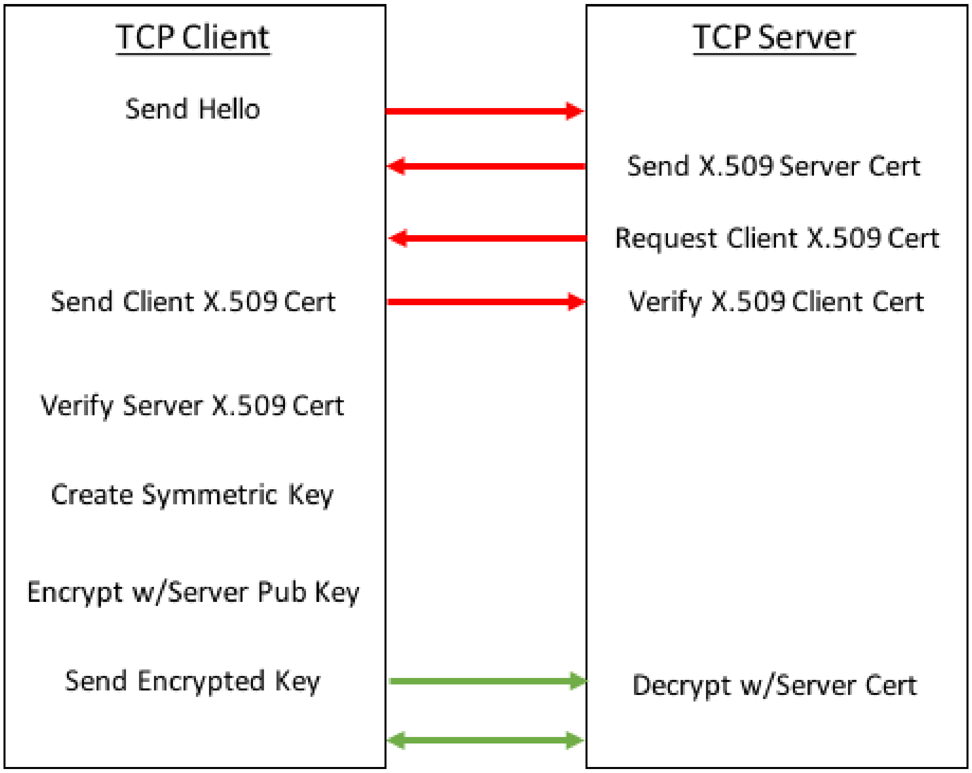
```
static const char httpbin_root_ca_certificate[] =
        "-----BEGIN CERTIFICATE-----\r\n"                                       \
        "MIIDSjCCAjKgAwIBAgIQRK+wgNajJ7qJMDmGLvhAazANBgkqhkiG9w0BAQUFADA/\r\n"   \
        "MSQwIgYDVQQKExtEaWdpdGFsIFNpZ25hdHVyZSBUcnVzdCBDby4xFzAVBgNVBAMT\r\n"   \
        "DkRTVCBSb290IENBIFgzMB4XDTAwMDkzMDIxMTIxOVoXDTIxMDkzMDE0MDExNVow\r\n"   \
        "PzEkMCIGA1UEChMbRGlnaXRhbCBTaWduYXR1cmUgVHJ1c3QgQ28uMRcwFQYDVQQD\r\n"   \
        "Ew5EU1QgUm9vdCBDQSBYMzCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEB\r\n"   \
        "AN+v6ZdQCINXtMxiZfaQguzH0yxrMMpb7NnDfcdAwRgUi+DoM3ZJKuM/IUmTrE4O\r\n"   \
        "rz5Iy2Xu/NMhD2XSKtkyj4zl93ewEnu1lcCJo6m67XMuegwGMoOifooUMM0RoOEq\r\n"   \
        "OLl5CjH9UL2AZd+3UWODyOKIYepLYYHsUmu5ouJLGiifSKOeDNoJjj4XLh7dIN9b\r\n"   \
        "xiqKqy69cK3FCxolkHRyxXtqqzTWMIn/5WgTe1QLyNau7Fqckh49ZLOMxt+/yUFw\r\n"   \
        "7BZy1SbsOFU5Q9D8/RhcQPGX69Wam40dutolucbY38EVAjqr2m7xPi71XAicPNaD\r\n"   \
        "aeQQmxkqtilX4+U9m5/wAl0CAwEAAaNCMEAwDwYDVR0TAQH/BAUwAwEB/zAOBgNV\r\n"   \
        "HQ8BAf8EBAMCAQYwHQYDVR0OBBYEFMSnsaR7LHH62+FLkHX/xBVghYkQMA0GCSqG\r\n"   \
        "SIb3DQEBBQUAA4IBAQCjGiybFwBcqR7uKGY3Or+Dxz9LwwmglSBd49lZRNI+DT69\r\n"   \
        "ikugdB/OEIKcdBodfpga3csTS7MgROSR6cz8faXbauX+5v3gTt23ADq1cEmv8uXr\r\n"   \
        "AvHRAosZy5Q6XkjEGB5YGV8eAlrwDPGxrancWYaLbumR9YbK+rlmM6pZW87ipxZz\r\n"   \
        "R8srzJmwN0jP41ZL9c8PDHIyh8bwRLtTcm1D9SZImlJnt1ir/md2cXjbDaJWFBM5\r\n"   \
        "JDGFoqgCWjBH4d1QB7wCCZAA62RjYJsWvIjJEubSfZGL+T0yjWW06XyxV3bqxbYo\r\n"   \
        "Ob8VZRzI9neWagqNdwvYkQsEjgfbKbYK7p2CNTUQ\r\n"                           \
        "-----END CERTIFICATE-----\n";
```

TCP/IP Sockets with TLS

In the WICED TLS API there are two structures which you need:

> **<u>wiced_tls_identity_t</u>** – this structure is used to hold your Public Key (in PEM certificate format) and your Private Key (in PEM format). On the client side, this is only used if the protocol requires the server to verify the client's identity (e.g. MQTT). On the server side, this is the server's certificate (which contains its public key) that is sent to initialize the connection. You initialize this structure with a call to *wiced_tls_init_identity*. You need to pass it the Certificate and Private key which can be read out of the DCT, Resources, or from #defines as explained above.
>
> **<u>wiced_tls_context_t</u>** – this structure is used to hold the TLS state machine and security information for the connection. Before launching TLS you need to initialize this structure with a *wiced_tls_init_context* call.

A TLS encrypted TCP Socket is almost exactly the same as an unencrypted socket.



Two steps in this picture are optional:

1. The server may optionally request the Client X.509 Certificate. If you are the Client and the server requests your certificate then you must have the *wiced_tls_identity_t* initialized or the server will get an error message.
2. The Client is not required to verify the Server X.509 Certificate. In the WICED TLS if you do not call *wiced_tls_init_root_ca_certificates*, then the firmware assumes that you don't want to verify the server certificate. In that case it trusts the connection without verifying the certificate.

The TCP Client TLS firmware flow is:

#	Step	Example
1	Create the socket	wiced_tcp_create_socket(&socket, WICED_STA_INTERFACE);
2	Bind the socket to a port	wiced_tcp_bind(&socket,WICED_ANY_PORT);
3	Read the security section of the DCT into RAM	wiced_dct_read_lock((void**) &dct_security, WICED_FALSE, DCT_SECURITY_SECTION, 0, sizeof(*dct_security));
4	Initialize a TLS Identity with the certificate and the private key (if protocol requires client certificate validation)	wiced_tls_init_identity(&tls_identity, dct_security->private_key, strlen(dct_security->private_key), (uint8_t*) dct_security->certificate, strlen(dct_security->certificate));
5	Initialize the Root Certificate of the TCP Server (only if you are going to validate the root certificate of the server)	wiced_tls_init_root_ca_certificates(dct_security->certificate, strlen(dct_security->certificate));

6	Initialize a TLS Context	wiced_tls_init_context(&tls_context, NULL, NULL);
		or if Server is verifying client certificate
		wiced_tls_init_context(&tls_context, &tls_identity, NULL);

7	Enable TLS on that Socket	wiced_tcp_enable_tls(&socket, &tls_context);
8	Make the TCP connection	wiced_tcp_connect(&socket,&serverAddress,SERVER_PORT,2000);

The TCP Server TLS firmware flow is:

#	Step	Example
1	Create the Socket	wiced_tcp_create_socket(&socket, INTERFACE);
2	Attach the Socket to Port	wiced_tcp_listen(&socket, TCP_SERVER_INSECURE_LISTEN_PORT);
3	Read the security section of the DCT into RAM	wiced_dct_read_lock((void**) &dct_security, WICED_FALSE, DCT_SECURITY_SECTION, 0, sizeof(*dct_security));
4	Initialize a TLS Identity with the certificate and the private key	wiced_tls_init_identity(&tls_identity, dct_security->private_key, strlen(dct_security->private_key), (uint8_t*) dct_security->certificate, strlen(dct_security->certificate));
5	Initialize a TLS Context	wiced_tls_init_context(&tls_context, &tls_identity, NULL);
6	**OPTIONAL** Initialize the Client Certificate	wiced_tls_init_root_ca_certificates(dct_security->certificate, strlen(dct_security->certificate));
7	Enable TLS on that socket	wiced_tcp_enable_tls(&socket,&tls_context);
8	Initialize a stream socket (same as non-TLS)	wiced_tcp_stream_init(&stream,&socket);
9	Accept connection (same as non-TLS)	wiced_tcp_accept(&socket);

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 non-secure 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 secure TLS connections

The WICED device attached to your network with name “wwep.ww101.cypress.com” and ip address 198.51.100.3 is running the non-secure version of WWEP (on port 27708) as well as the secure version of the protocol (on port 40508). The connection is secured with the self signed X.509 certificates in the directory “ClassCerts/WWEP/wwep_cert.pem”

1. Copy your (02) project to (03)
2. Copy the certificate/private key into the WICED resources directory
3. Update the makefile to load the WWEP server root certificate into the resources
4. After binding to the socket, add calls to:
   a. Load the resources into the RAM
   b. Initialize the root certificate
   c. Intialize the wiced tls context
   d. Enable TLS on the socket
5. After closing the connection don’t forget to deinit the tls context after the connection is done.

Hint: Run a “clean” before building or else your project may not see the new certificate and key. You will find clean at the top of the list of Make Targets. Just double-click on it to run it.

(Advanced) 04 Implement the server side of the non-secure WWEP protocol that can handle one connection at a time

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. 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 secure WWEP protocol

(Advanced) 06 Implement a client that can send both non-secure and secure TLS messages

Use button 0 to send non-secure messages and button 1 to send secure messages.

(Advanced) 07 Implement a server for the WWEP protocol that will serve both non-secure and secure connections

Further Reading

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

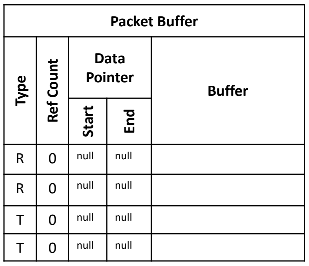
[3] IANA Service Name and Port Registry - http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

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



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	

A packet starts its life unallocated, and as such, the reference count is 0. When you want to send a message, you call *wicked_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 ); // Suspend until 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.