# Chapter 6B: Using TLS Sockets for Secure Communication

Time 1 Hour

At the end of Chapter 6B you will understand the fundamentals of symmetric and asymmetric encryption and how it is used to provide security to your IoT device by using TLS sockets.

**6B.1 SECURITY: SYMMETRIC AND ASYMMETRIC ENCRYPTION: A FOUNDATION ..... 2**

**6B.2 X.509 CERTIFICATES ..... 6**

6B.2.1 Basics ..... 6

6B.2.2 Downloading Certificates ..... 7

6B.2.3 Creating Your Own Certificates ..... 12

6B.2.4 Using Certificates in WICED ..... 12

**6B.3 TCP/IP SOCKETS WITH TRANSPORT LAYER SECURITY (TLS) ..... 15**

**6B.4 EXERCISE(S) ..... 19**

Exercise - 6B.1 Update the WWEP Client to use secure TLS connections ..... 19

Exercise - 6B.2 (Advanced) Implement Secure WWEP Server ..... 19

Exercise - 6B.3 (Advanced) Implement Dual Secure & Insecure WWEP Client ..... 19

Exercise - 6B.4 (Advanced) Implement Dual Secure & Insecure WWEP Server ..... 19

## 6B.1 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 decryption algorithm to produce clear-text - a.k.a. the original data.

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

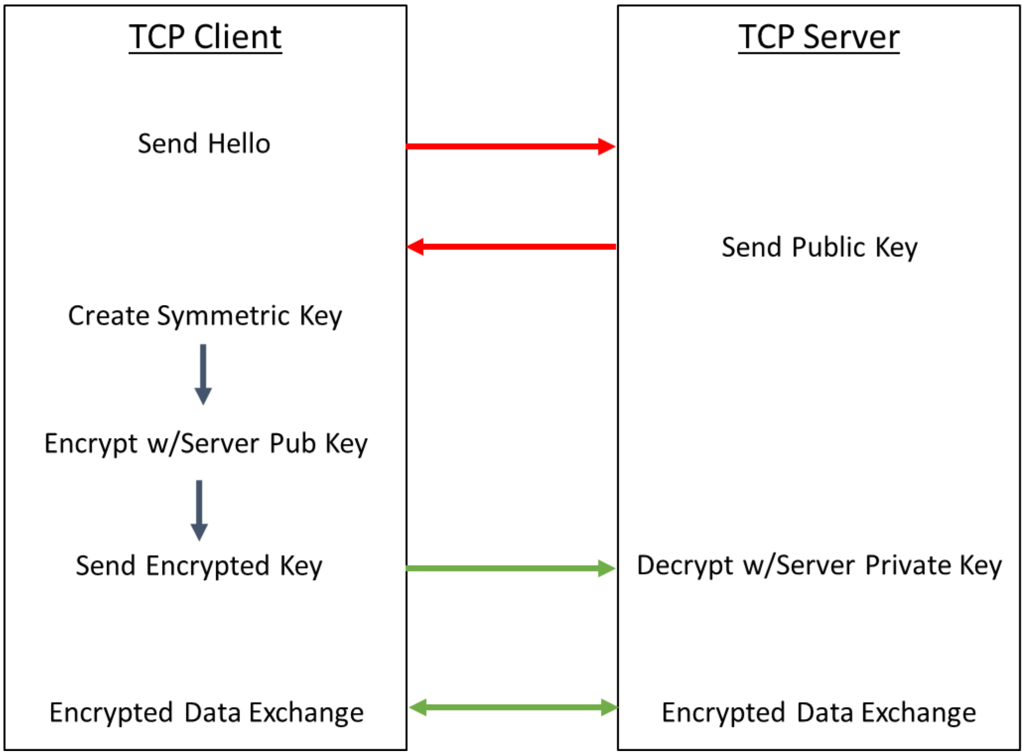
Symmetric means that both sides use the same key. That is, the key that you encrypt with is the same as the key you decrypt with. Examples of this type of encryption include AES and DES. Symmetric encryption is preferred because it is very fast and secure. Unfortunately, both sides need to know the key before you can use it - remember, the encryption key is exactly the same as the decryption 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:

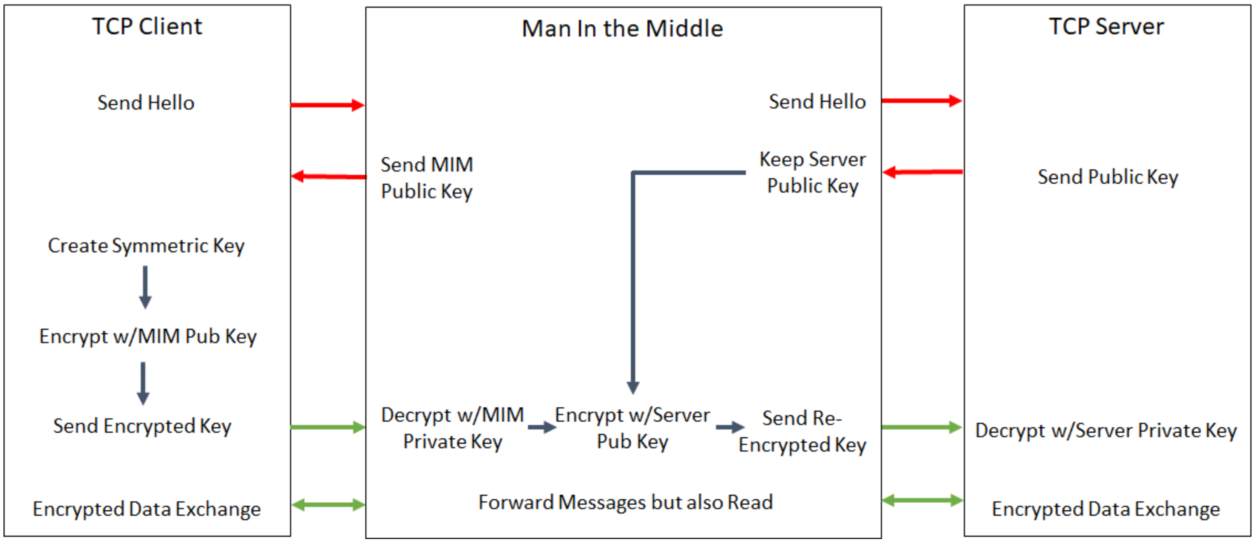
- You open an unencrypted connection to a server
- The server sends you its public key
- The client creates a random symmetric key
- The client encrypts its newly created random symmetric key using the server's public key and sends it back to server
- The Server decrypts the symmetric key using its private key

- You open a new channel using asymmetric 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's 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 as follows:

- You open an unencrypted connection to a Server [but it really turns out that it is an 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's Public Key)
- The Client sends it to the MIM (which it thinks is the server)
- The MIM unencrypts the symmetric key, then re-encrypts using the Server's Public Key
- The MIM opens a channel to the server using the re-encrypted 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. The certificates are built up in a "chain of trust" starting from the root certificate, to on or more intermediates and finally to the server. If you recognize either an 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.

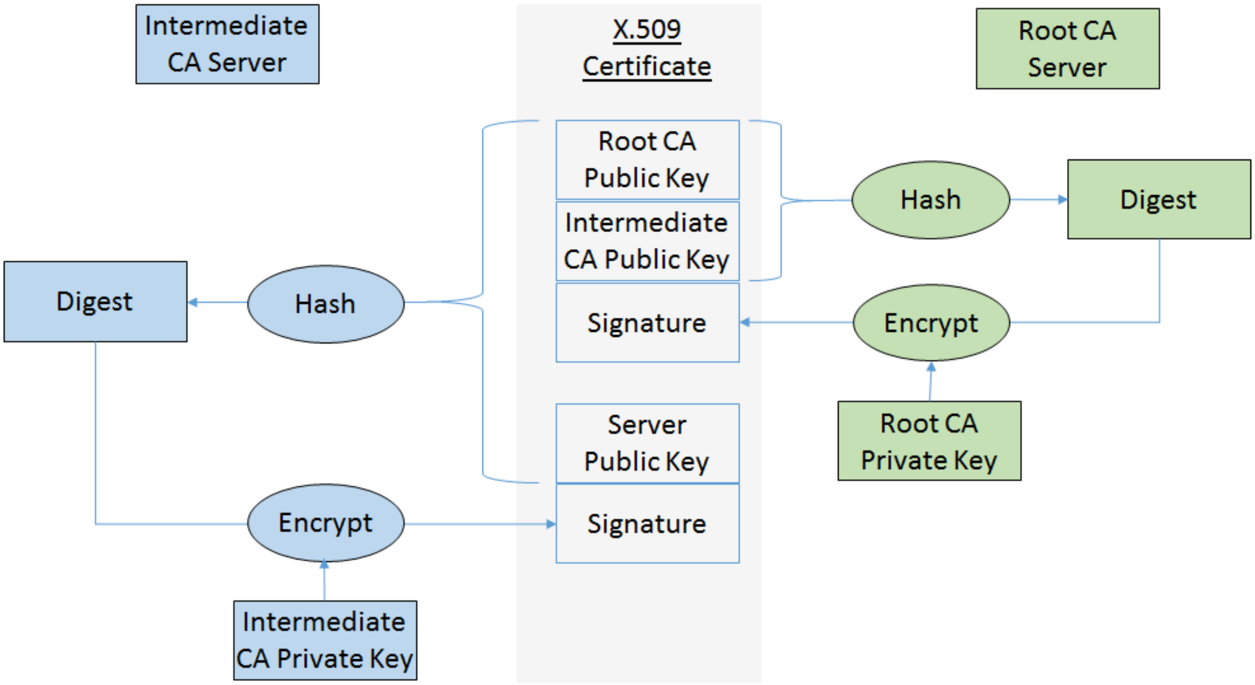
Note that this process can be done hierarchically to create a chain of signed certificates. That is, the server gets its identity validated by an intermediate authority who, in turn, gets its identity validated by a "higher" intermediate authority or by a root authority. Therefore, any certificate eventually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

- Take the public keys for the server and for the CA from the certificate and hash them– this reproduces the digest from the CA.
- Unencrypt the signature from the certificate 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.

The server's certificate is constructed as follows:



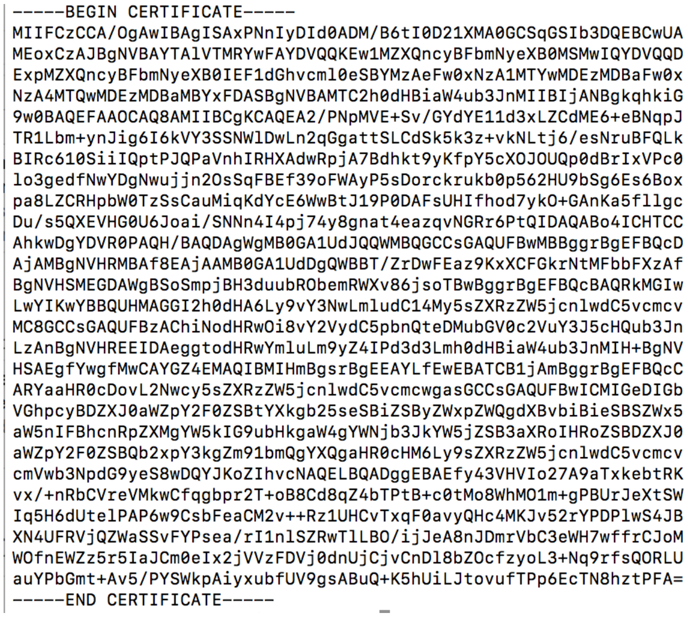
## 6B.2 X.509 Certificates

### 6B.2.1 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 – typically this will be a root certificate). 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 – if you don't verify the certificate, then you are susceptible to MIM attacks. You can register the server's CA's 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-in and can 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
- One or more 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

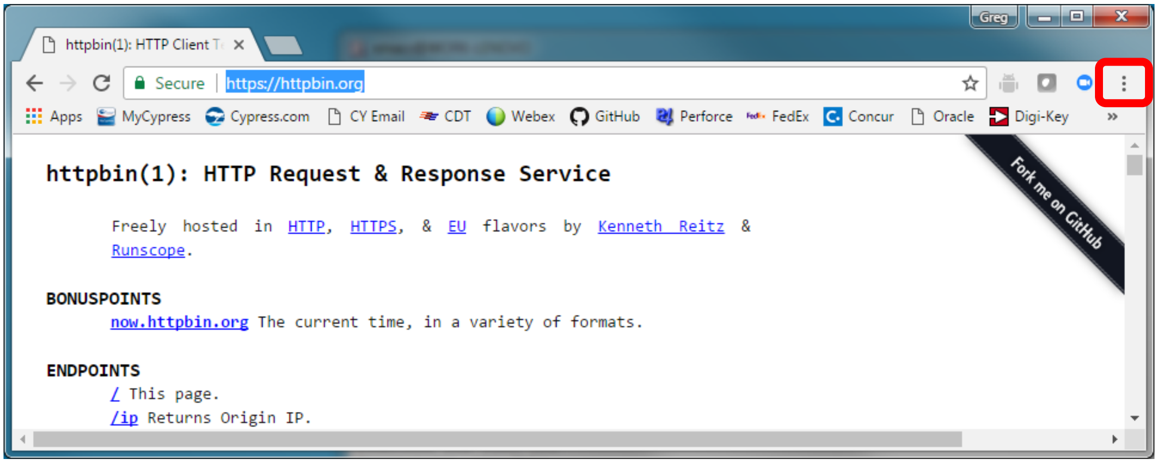
## 6B.2.2 Downloading Certificates

You can get the root or intermediate certificate for a secure 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".



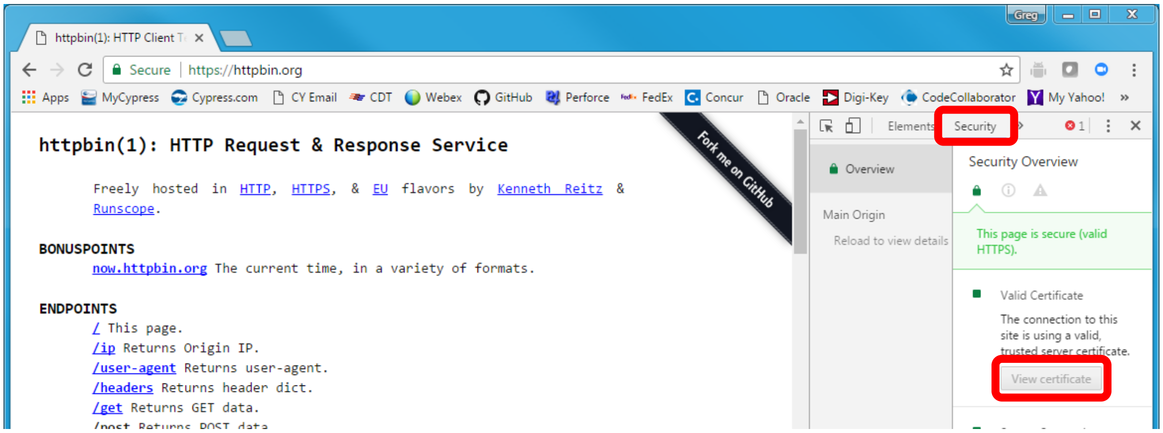
```
httpbin(1): HTTP Request & Response Service

      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 the "Security" tab and then 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.
```

Security Overview

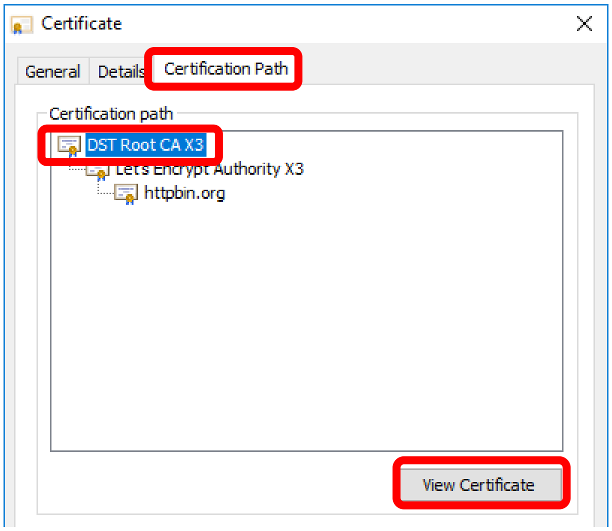
This page is secure (valid HTTPS).

Valid Certificate

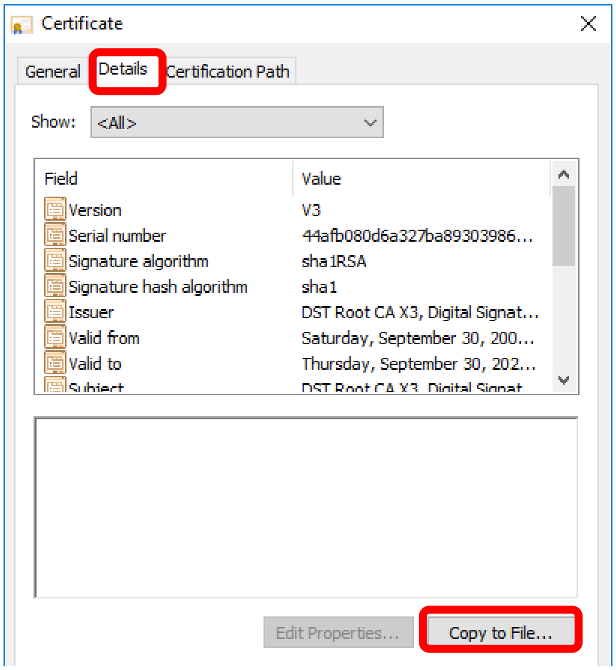
The connection to this site is using a valid, trusted server certificate.

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. It is recommended to get the root certificate rather than the

<u>intermediate since it will be valid for a longer period of time and it will validate more connections than the intermediate certificate</u>.

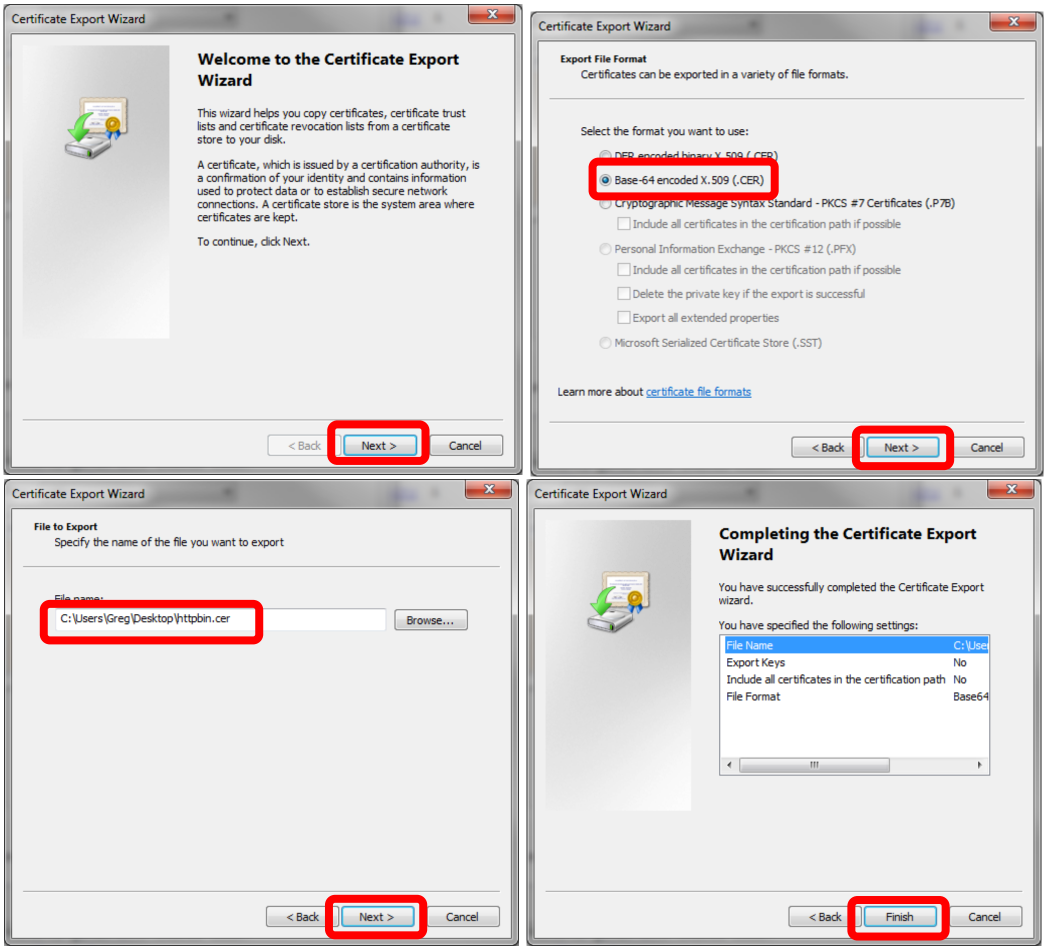


4. You will now have another window open showing information for the signed certificate. Click on the "Details" tab and then on "Copy to File..." to open the Certificate Export Wizard.



| Field | Value |
|---|---|
| Version | V3 |
| Serial number | 44afb080d6a327ba89303986... |
| Signature algorithm | sha1RSA |
| Signature hash algorithm | sha1 |
| Issuer | DST Root CA X3, Digital Signat... |
| Valid from | Saturday, September 30, 200... |
| Valid to | Thursday, September 30, 202... |
| Subject | DST Root CA X3, Digital Signat... |

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.

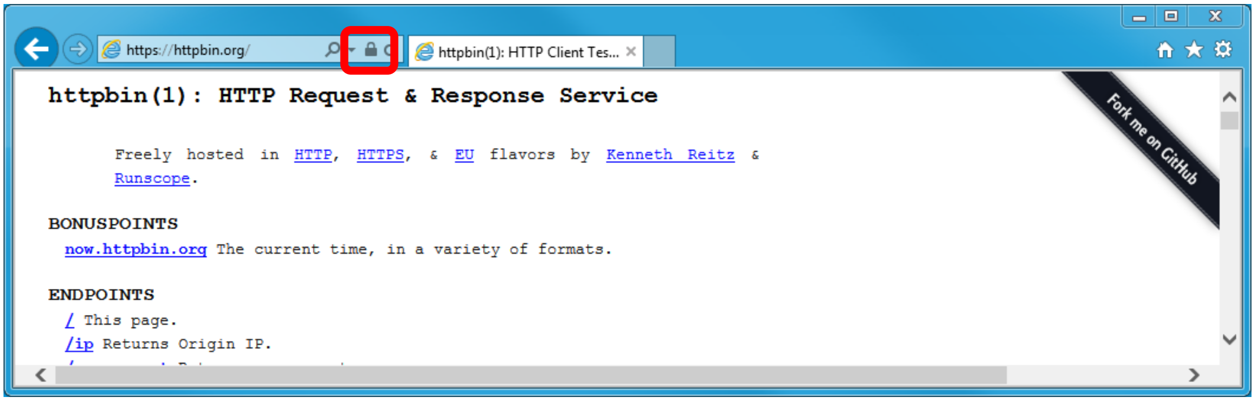
You have specified the following settings:

| | |
|---|---|
| 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 left 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

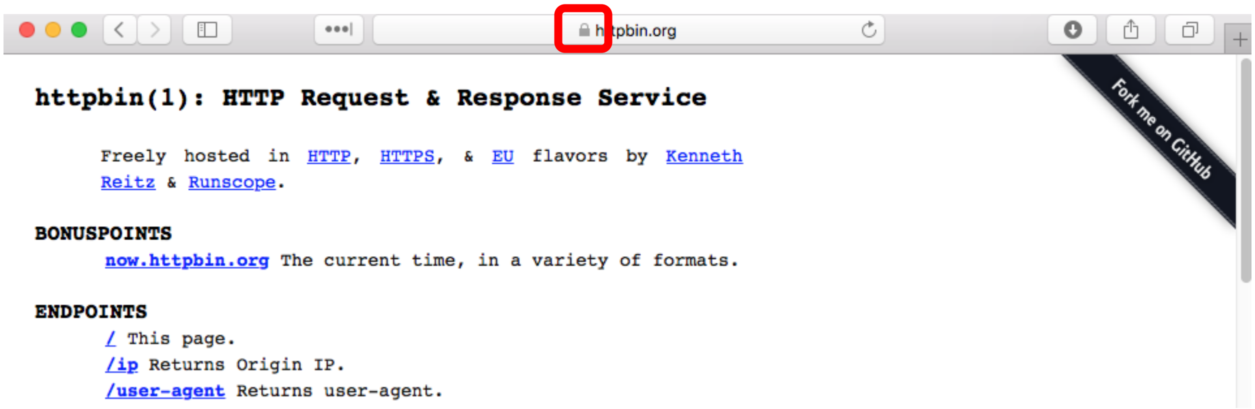
Freely hosted in HTTP, HTTPS, & EU flavors by Kenneth Reitz & Runscope.

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.



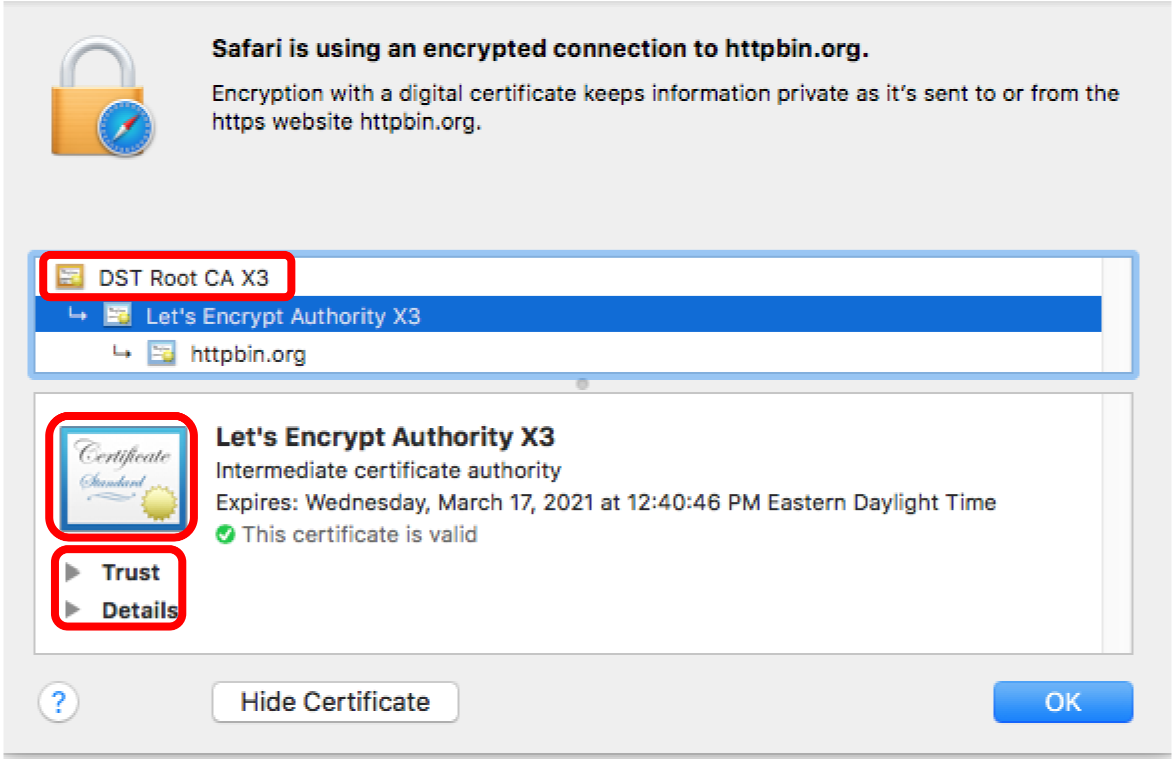
httpbin(1): HTTP Request & Response Service

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 (but it is recommended to always get the root 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

### 6B.2.3 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
```

### 6B.2.4 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. If you are going to validate the server's certificate, then either the root or intermediate certificate must be included in the firmware. The firmware uses the public key, expiration date, and domain from the root or intermediate certificate to validate the certificate that was sent from the server. Optionally, you may also have your own certificate (if the server requires it) included in the firmware.

Note that a certificate file may contain more than one certificate. This is useful if you need to connect to multiple servers that have different root certificates. To have multiple certificates in a file, just open the file with a text editor and place each certificate one after the other – as long as you have the BEGIN CERTIFICATE and END CERTIFICATE lines for each one, they will be treated independently. Any text outside the begin/end lines is ignored so you can add comments if you wish.

#### Method 1: Storing and using certificates from the DCT

It is possible to have the WICED make system install certificates into the DCT automatically. Note that the DCT only has space for one set of certificate/key files so if you need multiple certificate/key files, then some of them need to be stored using one of the other two methods. Note that the DCT is a fixed size whether you include a certificate/key or not, so it is more space efficient to store one set using this method.

To install a certificate in the DCT you need to:

1. Convert the certificate to PEM format if it is not already in that format, then store it in the SDK directory *resources.* You can have any additional hierarchy below resources to separate certificates for different projects.

2. Assuming you have a certificate file called *wwep_cert.pem* stored in the *resources/certificates/wwep* folder, you would add to your Makefile the line:

   ```
   CERTIFICATE := $(SOURCE_ROOT)resources/certificates/wwep/wwep_cert.pem
   ```

3. Likewise, if you have a private key called wwep_privkey.pem stored in the resources/certificates folder, you would add this to your Makefile:

   ```
   PRIVATE_KEY := $(SOURCE_ROOT)resources/certificates/wwep/wwep_privkey.pem
   ```

   Note that a private key is not required for a client unless it is going to be providing its certificate (which contains the public key) for validation by the server.

4. Once you have the certificate specified in the Makefile, you can load the security section of the DCT into RAM and use it to initialize the root certificate for the TLS connection. The structure of type platform_dct_sectурity_t includes both a certificate and private key (if there is one).

   ```
   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;
   }

   wiced_dct_read_unlock(dct_security, WICED_FALSE);
   ```

## 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 SDK directory *resources*.
2. Add the path to the certificate as a RESOURCES tag in your Makefile. For example, assuming you have a certificate file called *wwep_cert.pem* stored in the *resources/certificates/wwep* folder:

   ```
   $(NAME)_RESOURCES  := certificates/wwep/wwep_cert.pem
   ```

   Note that in this case, you don't specify "resources" in the path since it is part of the keyword.

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 "/" after the first sub-folder underneath *resources* and the first "/" is specified with "_".
   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 */
   char * wwep_root_ca_certificate;
   uint32_t size_out;
   resource_get_readonly_buffer( &resources_certificates_DIR_wwep_DIR_wwep_cert_pem, 0, 2048, &size_out,
(const void **) &wwep_root_ca_certificate );
   result = wiced_tls_init_root_ca_certificates( wwep_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.  The certificate will look like this in the source file:

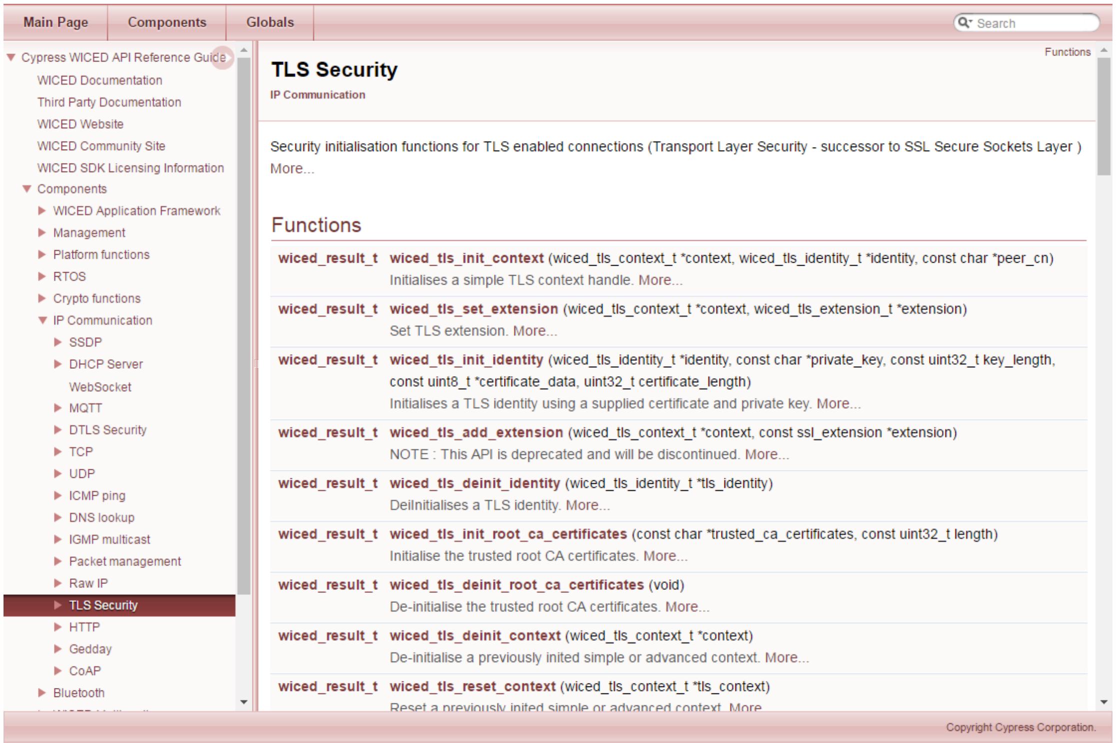
```
static const char wwep_root_ca_certificate[] =
        "-----BEGIN CERTIFICATE-----\r\n"                                   \
        "MIIDSjCCAjKgAwIBAgIQRK+wgNajJ7qJMDmGLvhAazANBgkqhkiG9w0BAQUFADA/\r\n"   \
        "MSQwIgYDVQQKExtEaWdpdGFsIFNpZ25hdHVyZSBUcnVzdCBDby4xFzAVBgNVBAMT\r\n"   \
        "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";
```

## 6B.3 TCP/IP Sockets with Transport Layer Security (TLS)

For 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...

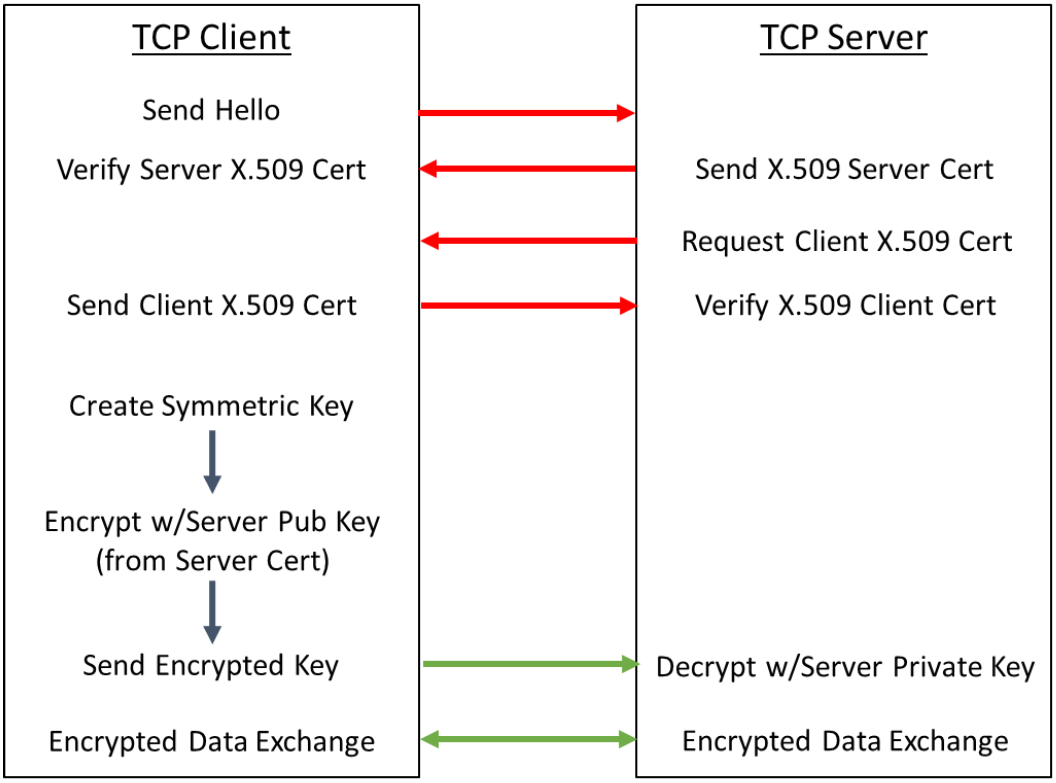
Functions

| | |
|---|---|
| wiced_result_t | **wiced_tls_init_context** (wiced_tls_context_t *context, wiced_tls_identity_t *identity, const char *peer_cn)<br>Initialises a simple TLS context handle. More... |
| wiced_result_t | **wiced_tls_set_extension** (wiced_tls_context_t *context, wiced_tls_extension_t *extension)<br>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)<br>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)<br>NOTE : This API is deprecated and will be discontinued. More... |
| wiced_result_t | **wiced_tls_deinit_identity** (wiced_tls_identity_t *tls_identity)<br>DeiInitialises a TLS identity. More... |
| wiced_result_t | **wiced_tls_init_root_ca_certificates** (const char *trusted_ca_certificates, const uint32_t length)<br>Initialise the trusted root CA certificates. More... |
| wiced_result_t | **wiced_tls_deinit_root_ca_certificates** (void)<br>De-initialise the trusted root CA certificates. More... |
| wiced_result_t | **wiced_tls_deinit_context** (wiced_tls_context_t *context)<br>De-initialise a previously inited simple or advanced context. More... |
| wiced_result_t | **wiced_tls_reset_context** (wiced_tls_context_t *tls_context)<br>Reset a previously inited simple or advanced context. More |

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

> **wiced_tls_identity_t** – 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.
>
> **wiced_tls_context_t** – 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 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 so the connection would be encrypted but would be susceptible to MIM attacks.

**You must include wiced_tls.h in the C source file to have access to the required WICED TLS functions.**

To use TLS with streams, there are just a few additions to the code used in the previous chapter. As you can see, there is some additional initialization but there are only three extra function calls to enable TLS each time you want to send/receive stream data. Everything else is exactly the same as it was without TLS.

For a TCP Client:

One time (during initialization):

| # | Step | Example |
|---|---|---|
| 1 (new) | Read the root certificate from the security section of the DCT into RAM (e.g. using method 1) | wiced_dct_read_lock( (void**) &dct_security, WICED_FALSE, DCT_SECURITY_SECTION, 0, sizeof( *dct_security ) ); |
| 2 (new)<br>(optional) | Read the client certificate from the resources filesystem (e.g. using method 2) (only if the server requires client certificate validation) | resource_get_readonly_buffer( &resources_apps_DIR_clientcer_pem, 0, 2048, &size_out, (const void **) &security ); |
| 3 (new)<br>(optional) | Initialize a TLS Identity with the client's certificate and the private key (only if the server requires client certificate validation) | wiced_tls_init_identity( &tls_identity, (char*) security->key, security->key_len , (const uint8_t*) security->cert, security->cert_len ); |
| 4 (new) | Initialize the Root Certificate of the TCP Server (only if you are going to validate the root certificate of the server – which you should always do to prevent MIM) | wiced_tls_init_root_ca_certificates( dct_security->certificate, strlen( dct_security->certificate ) ); |
| 5 (new) | Unlock the DCT | wiced_dct_read_unlock(dct_security, WICED_FALSE); |

Each time you want to send/receive data:

| # | Step | Example |
|---|---|---|
| 1 | Create a Socket | wiced_tcp_create_socket(&socket, WICED_STA_INTERFACE); |
| 2 | Bind the Socket to a port | wiced_tcp_bind(&socket, WICED_ANY_PORT); |
| 3 (new) | Initialize the TLS Context | wiced_tls_init_context( &tls_context, NULL, NULL );<br>or, if the Server is verifying client certificate:<br>wiced_tls_init_context( &tls_context, &tls_identity, NULL ); |
| 4 (new) | Enable TLS on the Socket | wiced_tcp_enable_tls( &socket, &tls_context ); |
| | Make the TCP connection | wiced_tcp_connect(&socket, &serverAddress, SERVER_PORT,2000); |
| 5 | Initialize a Stream | wiced_tcp_stream_init(&stream, &socket); |
| 6 | Write to the Stream | wiced_tcp_stream_write(&stream, sendMessage, strlen(sendMessage)); |
| 7 | Flush the Stream | wiced_tcp_stream_flush(&stream); |
| 8 | Read from the Stream (if you want to receive a response from the server) | result = wiced_tcp_stream_read(&stream, rbuffer, MAX_CHAR, TIMEOUT); |
| 9 (new) | Deinit the TLS Context | wiced_tls_deinit_context(&tls_context); |
| 10 | Deinit the Stream | wiced_tcp_stream_deinit(&stream); |
| 11 | Delete the Socket | wiced_tcp_delete_socket(&socket); |

For a TCP Server:

One time (during initialization):

| # | Step | Example |
|---|---|---|
| 1 (new) | Read the server's certificate and private key from the security section of the DCT into RAM (e.g. method 1) | wiced_dct_read_lock( (void**) &dct_security, WICED_FALSE, DCT_SECURITY_SECTION, 0, sizeof( *dct_security ) ); |
| 2 (new) | Read the certificate for the authority that signed for the client from the resources filesystem (e.g. method 2) (only if you are going to verify the client's certificate) | resource_get_readonly_buffer( &resources_apps_DIR_clientrootcer_pem, 0, 2048, &size_out, (const void **) &security ); |
| 3 (new) | Initialize a TLS Identity with the server's 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 ) ); |
| 4 (new)<br>(optional) | Initialize the client's root certificate (only if you are going to verify the client's certificate) | wiced_tls_init_root_ca_certificates(security->cert, security->cert_len); |
| 5 (new) | Unlock the DCT | wiced_dct_read_unlock(dct_security, WICED_FALSE); |
| 6 | Create a Socket | wiced_tcp_create_socket(&socket, INTERFACE); |
| 7 (modified) | Attach the Socket to the Secure Port | wiced_tcp_listen( &socket, TCP_SERVER_SECURE_LISTEN_PORT ); |

Each time you want to setup and accept a connection:

| # | Step | Example |
|---|---|---|
| 1 (new) | Initialize a TLS Context | wiced_tls_init_context( &tls_context, &tls_identity, NULL ); |
| 2 (new) | Enable TLS on the socket | wiced_tcp_enable_tls(&socket, &tls_context); |
| 3 | Initialize a stream socket | wiced_tcp_stream_init(&stream, &socket); |
| 4 | Accept connection | wiced_tcp_accept( &socket ); |
| 5 | Read from the Stream | wiced_tcp_stream_read_with_count(&stream, &rbuffer, MAX_LEGAL_MSG, 100, &dataReadCount); |
| 6 | Write a response to the Stream | wiced_tcp_stream_write(&stream, returnMessage, strlen(returnMessage)); |
| 7 | Flush the Stream | wiced_tcp_stream_flush(&stream); |
| 8 | Disconnect the Socket | wiced_tcp_disconnect(&socket); |
| 9 (new) | Deinit the TLS context | iced_tls_deinit_context(&tls_context); |
| 10 | Deinit the Stream | wiced_tcp_stream_deinit(&stream); |

## 6B.4 Exercise(s)

### Exercise - 6B.1 Update the WWEP Client 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 working client project from the last chapter or the solution project (either 06a/01_client or 06a/02_client_response) to 06b/01_secure_client.
2. Copy the certificate/private key from the directory “ClassCerts/WWEP” into the WICED resources/certificates directory.
3. Update the makefile to load the WWEP server root certificate into the resources
4. During initialization:
    a. Read the root certificate into RAM (use method 1 or 2)
    b. Initialize the root certificate
    c. Initialize the TLS context
5. After binding to the socket, add the call to enable TLS on the socket

Hint: Remember that you must include wiced_tls.h in the C source file to have access to the required WICED TLS functions.

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.

### Exercise - 6B.2 (Advanced) Implement Secure WWEP Server

Implement the server side of the secure WWEP protocol.

### Exercise - 6B.3 (Advanced) Implement Dual Secure & Insecure WWEP Client

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.

### Exercise - 6B.4 (Advanced) Implement Dual Secure & Insecure WWEP Server

Implement a server for the WWEP protocol that will serve both non-secure and secure connections.