Chapter 3b: Using TLS Sockets

At the end of this chapter 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.

Note: *If you are taking this course on your own, you will need two PSoC™ 6 kits to do the exercises in this chapter. If you are taking this course in a classroom setting, you will only need one kit.*

Table of contents

Document conventions

Convention	Usage	Example

`Courier New`	Displays code and text commands	`CY_ISR_PROTO(MyISR);`
		`make build`

Italics	Displays file names and paths	*sourcefile.hex*
[**bracketed, bold**]	Displays keyboard commands in procedures	[**Enter**] or [**Ctrl**] [**C**]
Menu > Selection	Represents menu paths	**File > New Project > Clone**
Bold	Displays GUI commands, menu paths and selections, and icon names in procedures	Click the **Debugger** icon, and then click **Next**.

3b.1 Security: Symmetric and asymmetric encryption

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-text" 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.

3b.1.1 Symmetric

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.

3b.1.2 Asymmetric

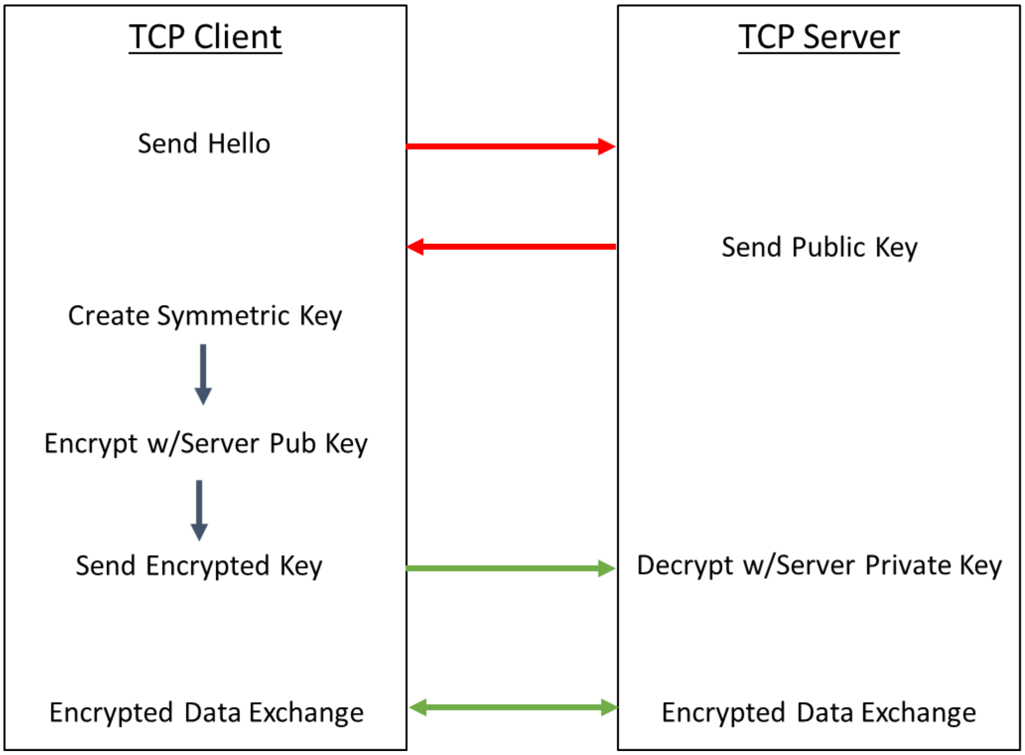
Asymmetric, often called Public Key, encryption techniques use a pair of keys that are mathematically related. The keys are often referred to as the "public" and the "private" keys. The private key can be used to decrypt data that the public key encrypted and vice versa. This is super cool because you 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 (it's a 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).

3b.1.3 Common communication technique

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 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 symmetric key encryption

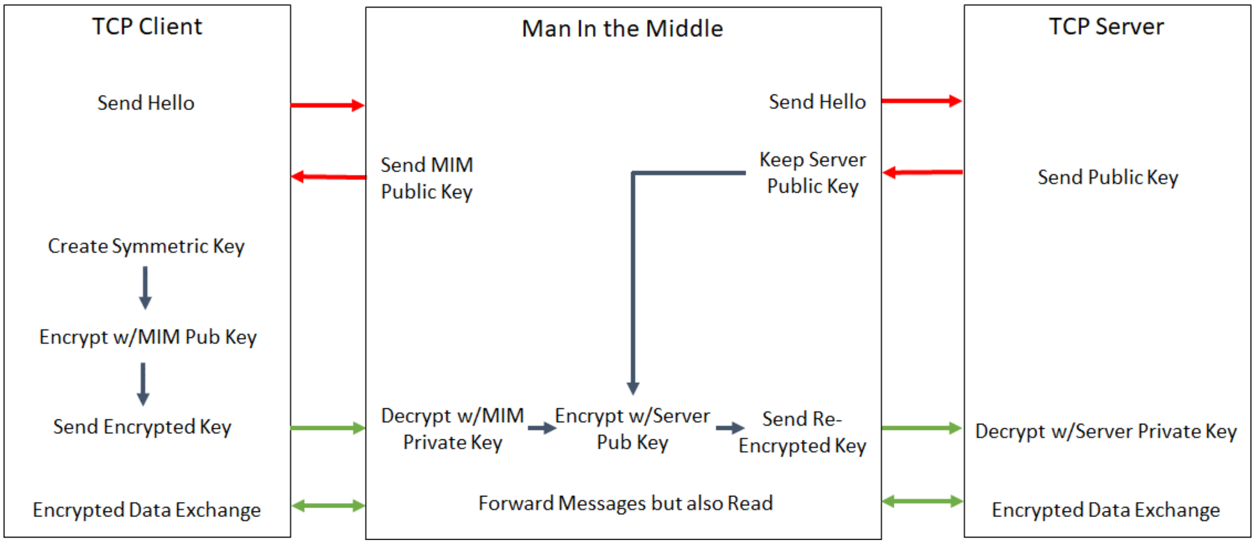


This scheme is completely effective against eavesdropping. But, what happens if someone eavesdrops the original public key? That is OK because they won't have the server's private key required to decrypt the symmetric key.

3b.1.4 Man-In-The-Middle (MITM) attacks

So, what's the hitch? What this scheme doesn't work against is called man-in-the-middle (MITM). An MITM attack works as follows:

- You open an unencrypted connection to a server [but it really turns out that it is an MITM]
- The MITM opens a channel to the server
- The server sends its public key to the MITM
- The MITM then sends its public key to the client
- The client creates a random symmetric key, encrypts it with the MITM Public key (which it thinks is really the server's Public Key)
- The client sends it to the MITM (which it thinks is the server)
- The MITM unencrypts the symmetric key, then re-encrypts using the server's Public Key
- The MITM opens a channel to the server and sends the re-encrypted symmetric key



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

However, the MITM can easily happen if someone gets control of an intermediate connection point in the network e.g. a Wi-Fi Access Point.

3b.1.5 Certificate Authority (CA)

There is only one good way to protect against MITM 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) 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 one or more intermediates and finally to the server. If you recognize either an Intermediate CA Public Key or the Root CA Public Key in the Certificate 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.

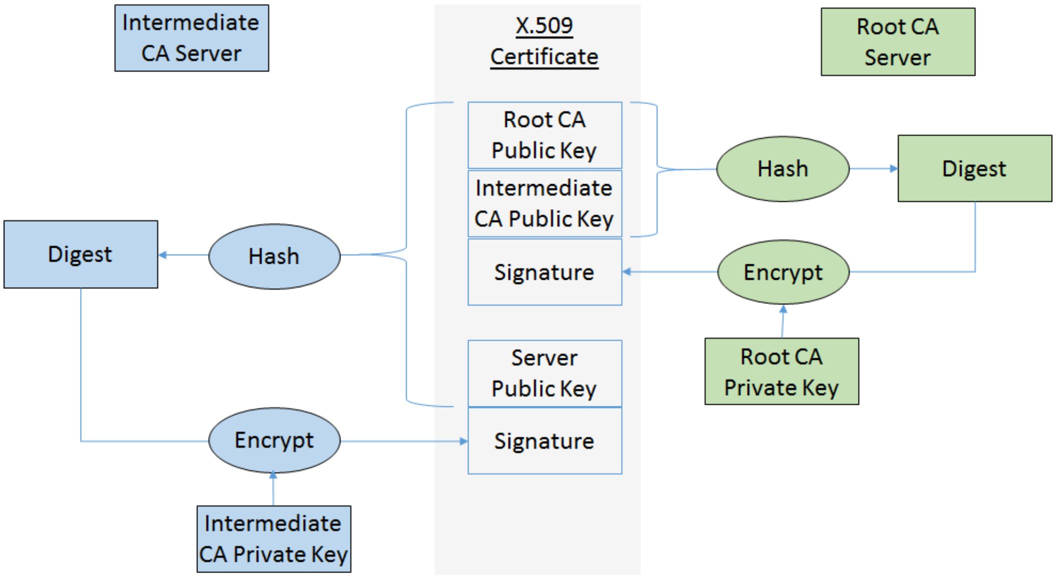
Note that this process can be done hierarchically to create a chain of signed certificates. That is, the server gets its identity validated by an intermediate authority who, in turn, gets its identity validated by a "higher" intermediate authority or by a root authority. Therefore, any certificate eventually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



3b.2 X.509 Certificates

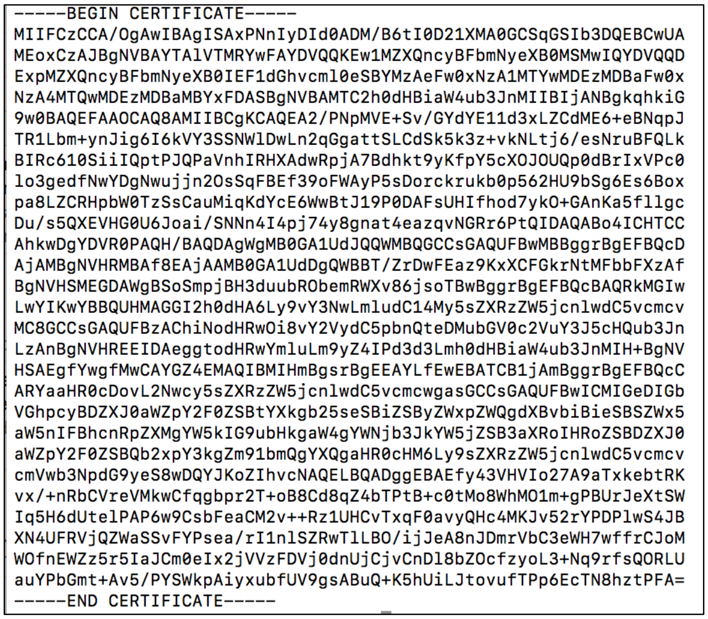
3b.2.1 Basics

To create a TLS connection, you need to have a root certificate available (i.e. built into your firmware) 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 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 the Secure Sockets Library uses).

In ModusToolbox™ for Connectivity it is optional for a client to verify the certificate from the server – if you don't verify the certificate from the server all traffic will still be encrypted, but you are susceptible to MITM attacks. You can register the server's CA's by calling `cy_tls_load_global_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. In this case it is required for the client to send its certificate to the server before the connection is allowed. This is common for cloud services so that they can verify an IoT device connecting to the cloud is not an imposter.

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 a large 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 the certificate
- The expiration date of the certificate
- One or more secure signatures that let you verify the authenticity of the certificate

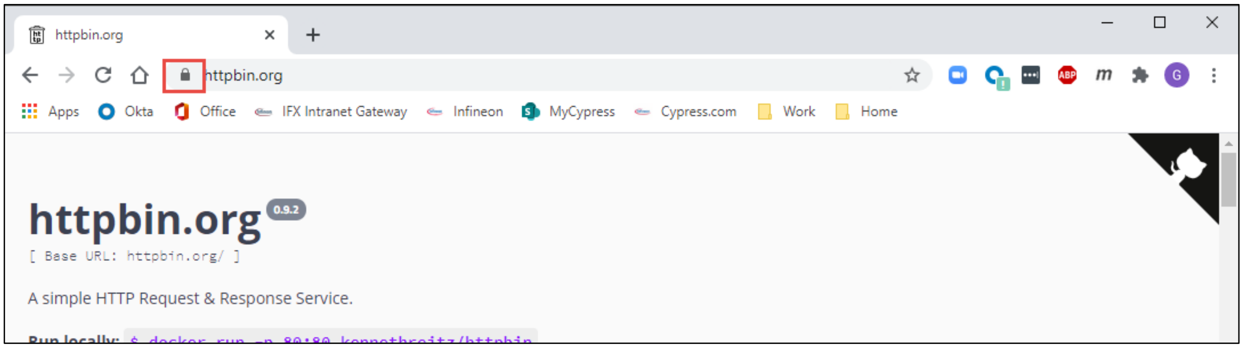
3b.2.2 Downloading certificates

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.

3b.2.2.1 Chrome or Edge

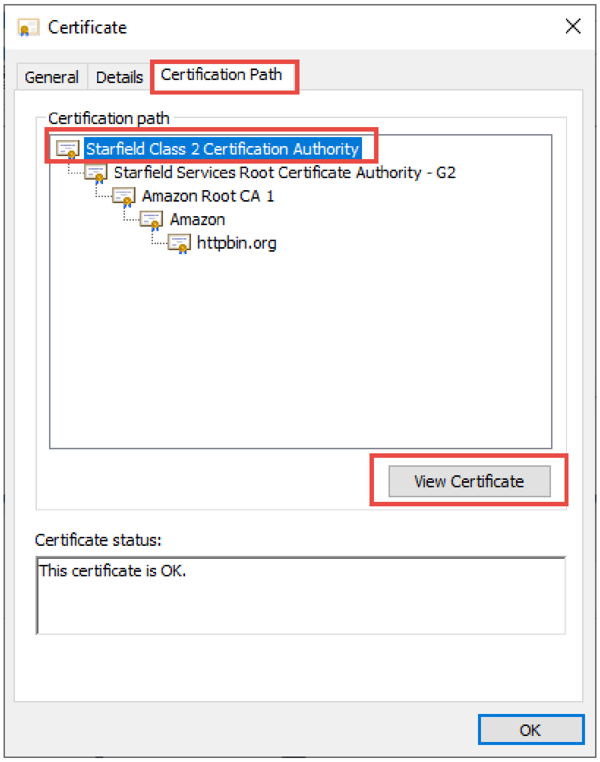
1. For both browsers, the first step is to navigate to the site you are interested in (https://httpbin.org), and then click the little lock icon in the address bar next to the website address.
    a. For Chrome, after clicking on the lock, select **Certificate**.
    b. For Edge, after clicking on the lock, select **Connection is Secure** and then click the **Show Certificate** button (it looks like a miniature certificate).

Note: You can view certificates using Internet Explorer but it will not allow you to save them.



A simple HTTP Request & Response Service.

2. Once the certificate opens in the certificate viewer, click on the **Certification Path** tab.

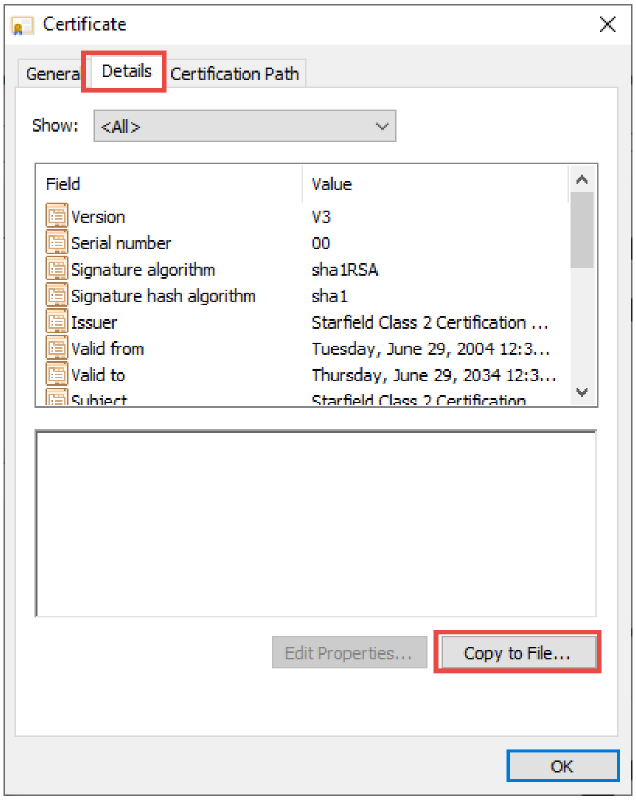


- Starfield Class 2 Certification Authority
  - Starfield Services Root Certificate Authority - G2
    - Amazon Root CA 1
      - Amazon
        - httpbin.org

This certificate is OK.

In this case, you can see that there is a chain of trust through several CAs. To make the TLS connection to https://httpbin.org you will need either a signed intermediate certificate or the root certificate, not the httpbin.org certificate. It is recommended to always get the highest certificate (the root of the chain of trust) since it will be able to validate more connections and will likely be valid for a longer period of time than an intermediate certificate.

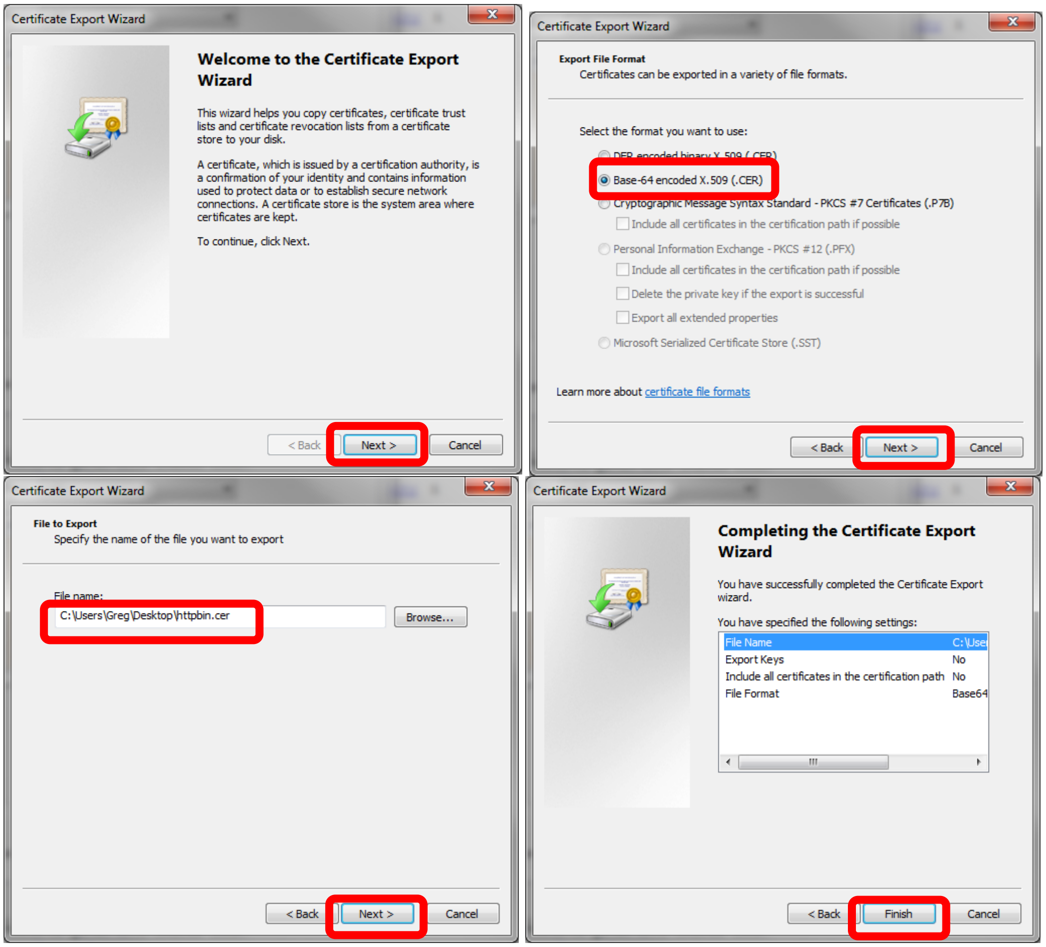
3. Click on the top certificate's name and then click on **View Certificate** to open a new window to look at (and save) the signed root certificate.
4. You will now have another window open showing information for the root certificate. Click on the **Details** tab and then on **Copy to File…** to open the Certificate Export Wizard.



Show: <All>

Field	Value
Version	V3
Serial number	00
Signature algorithm	sha1RSA
Signature hash algorithm	sha1
Issuer	Starfield Class 2 Certification ...
Valid from	Tuesday, June 29, 2004 12:3...
Valid to	Thursday, June 29, 2034 12:3...
Subject	Starfield Class 2 Certification

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.



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.

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

File to Export
Specify the name of the file you want to export

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

Completing the Certificate Export Wizard

You have successfully completed the Certificate Export wizard.

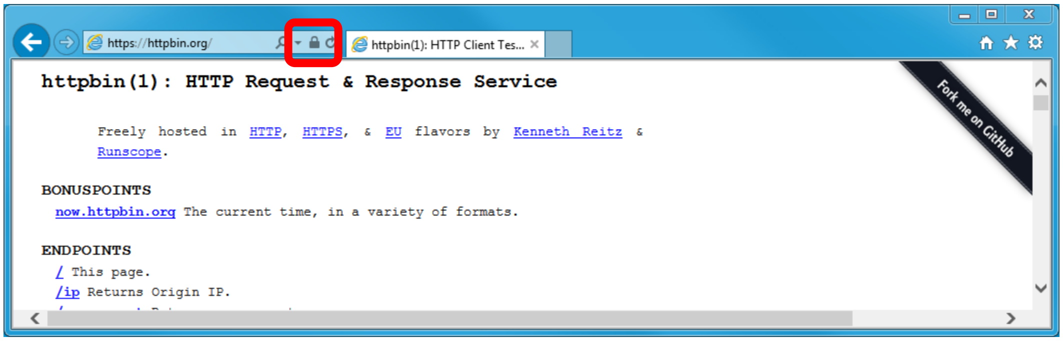
You have specified the following settings:

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.

3b.2.2.2 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".



httpbin(1): HTTP Request & Response Service

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

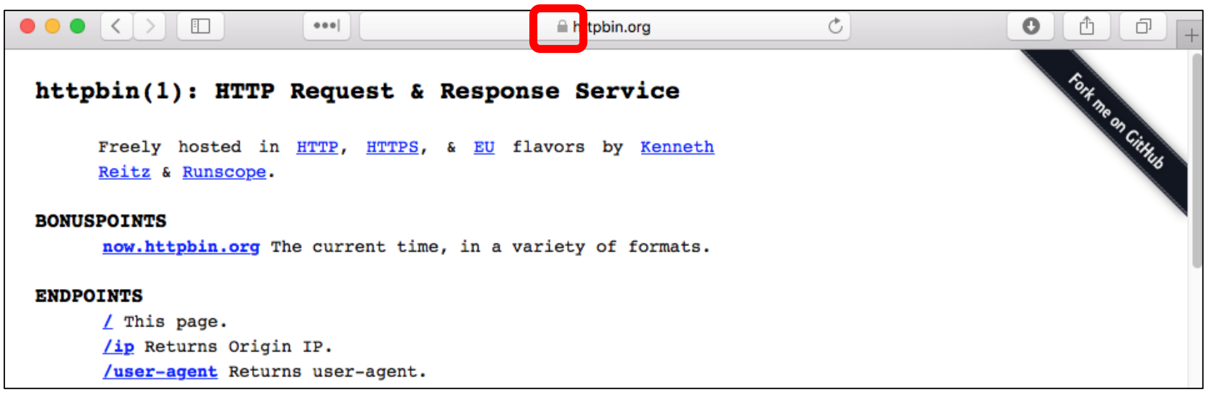
BONUSPOINTS
now.httpbin.org The current time, in a variety of formats.

ENDPOINTS
/ This page.
/ip Returns Origin IP.

Once you have the Certificate viewer open you can follow the same steps as for Chrome to save the certificate.

3b.2.2.3 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

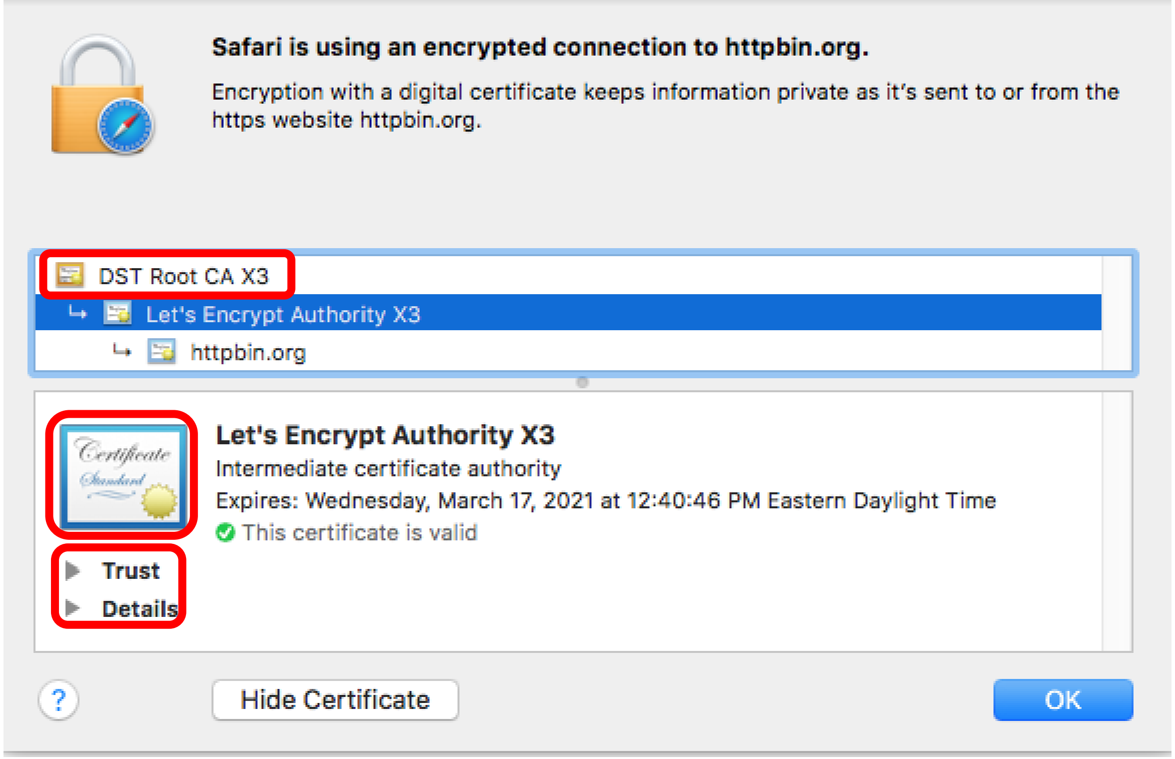
Freely hosted in HTTP, HTTPS, & EU flavors by Kenneth Reitz & Runscope.

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 root certificate 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.

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

To use the certificate in your application 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.pem” –text -noout

3b.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 is part of the ModusToolbox™ installation so you can run it from modus-shell.

First, you will create your own root certificate authority certificate using the following commands:

```
openssl ecparam -name prime256v1 -genkey -noout -out root_ca.key

openssl req -new -x509 -sha256 -key root_ca.key -out root_ca.crt -days 3650
```

Note: *You will be asked for some information while creating the certificates. You can simply use the defaults, or you can enter values if you prefer. For example, you can use "US" for the country name.*

Note: *The certificate will be valid for about 10 years (3,650 days). After that, it will no longer work.*

Next, you will create a public/private key pair for your device. In this case, we will assume it is the client, but the same would work for a server:

```
openssl ecparam -name prime256v1 -genkey -noout -out client.key

openssl req -new -sha256 -key client.key -out client.csr
```

Finally, you will use the root certificate you created in the first step to sign the public key, creating a signed certificate that your device can send out. The receiving device will need to have your root certificate to be able to validate your devices “self-signed” certificate.

```
openssl x509 -req -in client.csr -CA root_ca.crt -CAkey root_ca.key -
CAcreateserial -out client.crt -days 3650 -sha256
```

Note: *The command above should be entered on a single line. It is shown on 2 lines here due to space constraints. There should be no space between the dash and CAcreateserial.*

After completing the steps shown above, you will have *root_ca.crt* (root certificate), *client.crt* (signed client device certificate), and *client.key* (client private key).

3b.2.4 Using certificates in ModusToolbox™ for Connectivity

Once you have a certificate, you need to load it into your device. This is discussed in detail 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 need to include your own certificate (if the server requires it to validate your device) 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.

3b.2.4.1 Storing and using certificates

To install a certificate, you need to:

1. Convert the certificate to PEM format if it is not already in that format, then store it in your code via a macro.

   ```
   #define ROOTCA_PEM \
   "-----BEGIN CERTIFICATE-----\n"\
   "MIIDhTCCAm0CFHpvc/v8K67YWp7SkKZ+U8UgDBvRMA0GCSqGSIb3DQEBCwUAMH8x\n"\
   "CzAJBgNVBAYTAklOMRIwEAYDVQQIDAlLYXJuYXRha2ExGDAWBgNVBAcMD0Jlbmdh\n"\
   "bHVydRtbQxtbRDEQMA4GA1UECgwHQ3lwcmVzczEUMBIGA1UECwwLRW5naW5lZXJp\n"\
   "M9963Z24pQ6QRA7MrDiAFG7ajKrPwJpbJ5DgzD72+LSIPjejKeFiP5S3LgUvPXv8\n"\
   "neQ3UdXCY/8M44yULhKRlhKIk+B5vwBWfm8IxAfJCsktREomUM0jGoJScljVJmul\n"\
   "AgMBAAEwDQYJKoZIhvcNAQELBQADggEBAAqyZAIRhNJumQnLM0Kke8lobUpBS0Cq\n"\
   "kSaC1zQIQ8w6ruVAXVMohE9DFQbBk7OgX8HxUzDUQIL6h9i2knF0dNseKSFvul4w\n"\
   "4I6eYopg/jsTmEkWOgcw9bXb7ibJDjmxSOx9nPXy7ouyqw1/jMlEqAZ8l8hWMYA+\n"\
   "fsdkah64dvGTLfhyXpOtF/TUjhLG4A3y6VMTJcZhWbqmIBaY45u9c6nRksM/5ZX9\n"\
   "B6PWtpHE5Q4GfQJavgnlLhaOOTuznhssBKIMzTFivAA35RYL5btRsQkKu/2oALP4\n"\
   "yg+tikuvKL2cuAHvFmHbAlJcn5wsTMBLb5AU6pacdtS0uPvsD5QHEgM=\n"\
   "-----END CERTIFICATE-----\n"
   ```

Note: *There is a Python script in the Scripts directory called format_certificate.py that can be used to add the quote, backslash, and \n characters in the correct place for use in the firmware. The script prints the formatted certificate to the screen so that it can be copy/pasted into the file. The script does not include the `#define` line so you must still create that yourself. To use the script, open a command line terminal (modus-shell in Windows) and run:*

```
python format_certificate.py <name of certificate file>
```

2. Create a `char array` in your application to be able to reference the root certificate.

   ```
   /* Root CA certificate for TCP server identity verification. */
   static const char tcp_server_ca_cert[] = ROOTCA_PEM;
   ```

3. Load the root certificate into RAM using the function `cy_tls_load_global_root_ca_certificates`. For example:

   ```
   cy_tls_load_global_root_ca_certificates(tcp_server_ca_cert,
   strlen(tcp_server_ca_cert));
   ```

4. If the server will be verifying the client's identity, do the following steps as well:

   a. Define macros for your client's private key and certificate. These will look like the macro defined for the root certificate, but with different names such as `CLIENT_CERTIFICATE_PEM` and `CLIENT_PRIVATE_KEY_PEM`.

   b. Create char arrays for the client's private key and certificate. These will look like the array created for the root certificate but with different names such as `tcp_client_cert` and `client_private_key`.

   c. Load your public and private key into memory using the function `cy_tls_create_identity`. For example:

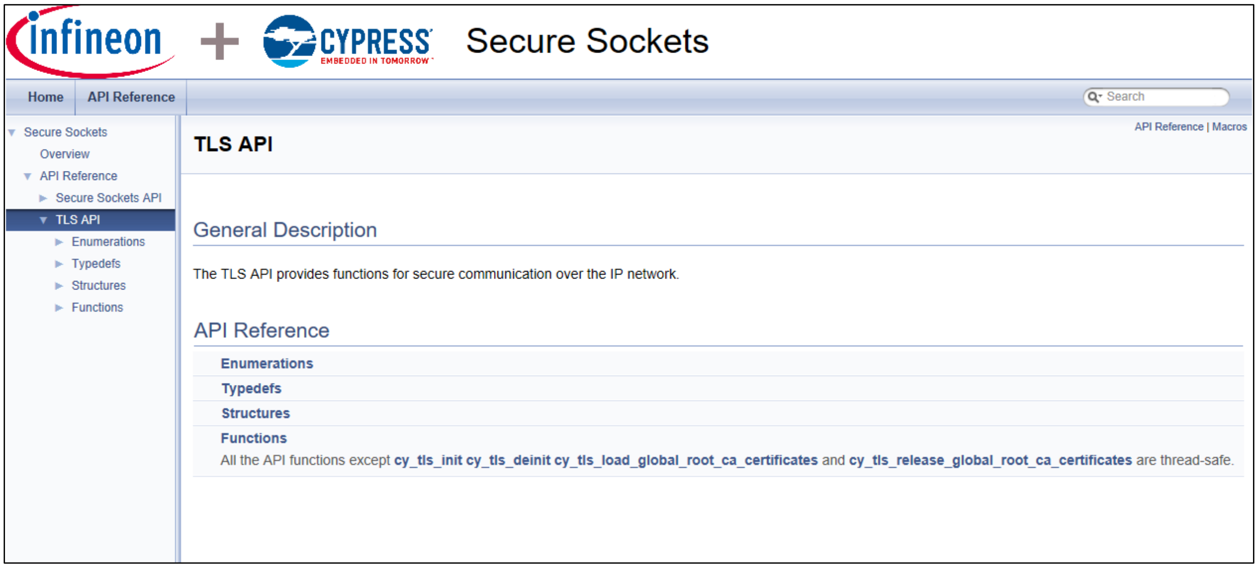
```
cy_tls_create_identity(tcp_client_cert,
strlen(tcp_client_cert),client_private_key, strlen(client_private_key),
&tls_identity);
```

3b.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 device running HTTP or 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 the Secure Sockets library and if you give it the certificates and 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 **Secure Sockets > API Reference > TLS API**.



TLS API

General Description

The TLS API provides functions for secure communication over the IP network.

API Reference

Enumerations

Typedefs

Structures

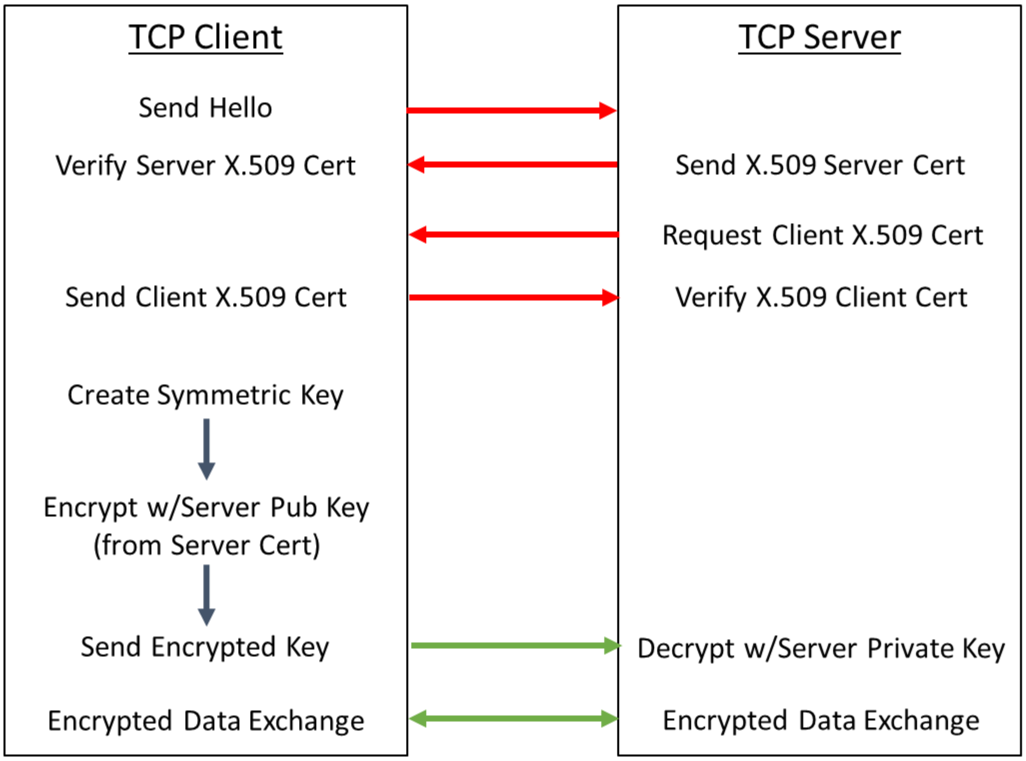
Functions

All the API functions except **cy_tls_init cy_tls_deinit cy_tls_load_global_root_ca_certificates** and **cy_tls_release_global_root_ca_certificates** are thread-safe.

In the TLS API, you need to create the a `tls_identity_t` structure, which is used to hold your Public Key (in PEM certificate format) and your Private Key (in PEM format). As shown above for 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 `cy_tls_create_identity`. You need to pass it the Certificate and Private key which can be read out of `#defines` as explained above.

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



Two steps in this picture are optional:

1. The server may optionally request the client's X.509 Certificate. If you are the client and the server requests your certificate, then you must have the `tls_identity_t` initialized or the server will get an error message. The server and the client choose whether to verify the connecting peer's certificate or not by setting the `CY_SOCKET_SO_TLS_AUTH_MODE` socket option.
2. The client is not required to verify the server's X.509 Certificate. In TLS, if you do not explicitly set the authentication mode then it will default to `CY_SOCKET_TLS_VERIFY_NONE`, and the firmware will assume that you don't want to verify the server certificate. In that case, it trusts the connection without verifying the certificate so the connection will be encrypted but will be susceptible to MITM attacks.

Note: *You must include cy_tls.h in the C source file to have access to the required TLS functions.*

To use TLS, 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 two extra function calls to enable TLS each time you want to send/receive data. Everything else is exactly the same as it was without TLS.

3b.3.1 For a TCP client:

One time (during initialization):

#	Step	Example
1	Initialize the secure sockets library	`cy_socket_init();`

#	Step	Example

2 (new)
(optional)
	Store the root certificate(s) via macros and create char arrays to reference them in your application. (Only needed if you are going to validate the certificate of the server – which you should always do to prevent MITM)	`#define ROOTCA_PEM "..."`
		
		`static const char tcp_server_ca_cert[] = ROOTCA_PEM;`

3 (new)	Store the client's certificate and private key via macros and create char arrays to reference them in your application.	`#define CLIENT_CERTIFICATE_PEM "..."`
		`#define CLIENT_PRIVATE_KEY_PEM "..."`
		
		`static const char tcp_client_cert[] = CLIENT_CERTIFICATE_PEM;`
		`static const char client_private_key[] = CLIENT_PRIVATE_KEY_PEM;`

4 (new)	Initialize a TLS Identity with the client's certificate and the private key.	`cy_tls_create_identity (tcp_client_cert, strlen(tcp_client_cert),client_private_key, strlen(client_private_key), &tls_identity);`
5 (new) (optional)	Initialize the Root Certificate of the TCP server. (Only needed if you are going to validate the root certificate of the server – which you should always do to prevent MITM)	`cy_tls_load_global_root_ca_certificates (tcp_server_ca_cert, strlen(tcp_server_ca_cert));`

Each time you want to send/receive data:

#	Step	Example
1 (changed)	Create a Socket, be sure the use `CY_SOCKET_IPPROTO_TLS` rather than `CY_SOCKET_IPPROTO_TCP`	`cy_socket_create (CY_SOCKET_DOMAIN_AF_INET, CY_SOCKET_TYPE_STREAM,CY_SOCKET_IPPROTO_TLS, &client_handle);`
2 (new)	Enable TLS on the Socket	`cy_socket_setsockopt (client_handle, CY_SOCKET_SOL_TLS, CY_SOCKET_SO_TLS_IDENTITY, tls_identity, sizeof(tls_identity));`

3 (new)	Set the TLS authentication mode – this can be one of the following depending on if you are validating the server's identity:
	`CY_SOCKET_TLS_VERIFY_NONE`
	`CY_SOCKET_TLS_VERIFY_OPTIONAL`
	`CY_SOCKET_TLS_VERIFY_REQUIRED`
		`cy_socket_tls_auth_mode_t tls_auth_mode = CY_SOCKET_TLS_VERIFY_REQUIRED;`
		`cy_socket_setsockopt(client_handle, CY_SOCKET_SOL_TLS, CY_SOCKET_SO_TLS_AUTH_MODE, &tls_auth_mode, sizeof(cy_socket_tls_auth_mode_t));`

4	Make the TCP connection	`cy_socket_connect (client_handle, &address, sizeof(cy_socket_sockaddr_t));`
5	Write data	`cy_socket_send (client_handle, message_buffer, strlen(message_buffer), CY_SOCKET_FLAGS_NONE, &bytes_sent);`
6	Read from the Stream (if you want to receive a response from the server)	`cy_socket_recv (client_handle, message_buffer, TCP_LED_CMD_LEN, CY_SOCKET_FLAGS_NONE, &bytes_received);`
7	Disconnect	`cy_socket_disconnect (client_handle, 0);`
8	Delete the Socket	`cy_socket_delete(client_handle);`

3b.3.2 For a TCP server:

One time (during initialization):

#	Step	Example
1	Initialize the secure sockets library	`cy_socket_init();`

2 (new) (optional)	Store the root certificate(s) via macros and create char arrays to reference them in your application. (Only needed if you are going to validate the certificate of the client)	`#define ROOTCA_PEM "..."`
		`static const char tcp_client_ca_cert[] = ROOTCA_PEM;`

3 (new)	Store the server's certificate and private key via macros and create char arrays to reference them in your application.	`#define SERVER_CERTIFICATE_PEM "..."`
		`#define SERVER_PRIVATE_KEY_PEM "..."`
		`static const char tcp_server_cert[] = SERVER_CERTIFICATE_PEM;`
		`static const char server_private_key[] = SERVER_PRIVATE_KEY_PEM;`

4 (new)	Initialize a TLS Identity with the server's certificate and the private key.	`cy_tls_create_identity(tcp_server_cert, tcp_server_cert_len, server_private_key, pkey_len, &tls_identity);`
5 (new) (optional)	Initialize the Root Certificate of the TCP client. (Only needed if you are going to validate the certificate of the client)	`cy_tls_load_global_root_ca_certificates (tcp_client_ca_cert, strlen(tcp_client_ca_cert));`
6 (changed)	Create a Socket	`cy_socket_create(CY_SOCKET_DOMAIN_AF_INET, CY_SOCKET_TYPE_STREAM, CY_SOCKET_IPPROTO_TLS, &server_handle);`
7 (new)	Set the socket to use the TLS identity	`cy_socket_setsockopt(server_handle, CY_SOCKET_SOL_TLS, CY_SOCKET_SO_TLS_IDENTITY, tls_identity, sizeof(tls_identity));`

8 (new)	Set the TLS authentication mode – this can be one of the following depending on if you are validating the client's identity:
	`CY_SOCKET_TLS_VERIFY_NONE`
	`CY_SOCKET_TLS_VERIFY_OPTIONAL`
	`CY_SOCKET_TLS_VERIFY_REQUIRED`
		`cy_socket_tls_auth_mode_t tls_auth_mode = CY_SOCKET_TLS_VERIFY_REQUIRED;`
		`cy_socket_setsockopt(server_handle, CY_SOCKET_SOL_TLS, CY_SOCKET_SO_TLS_AUTH_MODE, &tls_auth_mode, sizeof(cy_socket_tls_auth_mode_t));`

9	Bind the TCP socket created to your server IP address and TCP port	`cy_socket_bind(server_handle, &tcp_server_addr, sizeof(tcp_server_addr));`
10	Listen on the socket for incoming connection requests	`cy_socket_listen(server_handle, TCP_SERVER_MAX_PENDING_CONNECTIONS);`

Each time you want to setup and accept a connection:

#	Step	Example
1	Accept connection	`cy_socket_accept(server_handle, &peer_addr, &peer_addr_len, &client_handle);`
2	Read from the connection	`cy_socket_recv(server_handle, message_buffer, MAX_TCP_RECV_BUFFER_SIZE, CY_SOCKET_FLAGS_NONE, &bytes_received);`

#	Step	Example
3	Write a response	`cy_socket_send(server_handle, &message, strlen(message), CY_SOCKET_FLAGS_NONE, &bytes_sent);`
4	Disconnect the Socket	`cy_socket_disconnect(server_handle, timeOut);`

3b.4 Exercises

Exercise 1: Update the AWEP client to use secure TLS connections

The AWEP server has the hostname "awep". It is running the non-secure version of AWEP (on port 50007) as well as the secure version of the protocol (on port 50008). The connection is secured with the self-signed X.509 certificates in the directory *ClassCerts/AWEP.* The client will need its certificate and private key, as well as the root certificate used for signing the server's certificate.

Server Setup

If you are taking this course in a classroom setting, you can skip these steps, as the AWEP server has already been set up for you. However, if you are taking this class on your own, you will need to set the server up yourself. To do this, you will need a second PSoC™ 6 kit.

□ 1. Create a new application named **ch03_AWEP_server** using the **Browse** button to select the completed server application (***modustoolbox-level2-wifi/projects/ch03b_ex04_dual_server***) as a template.

□ 2. Open the file *tcp_server.h* and edit the macros `WIFI_SSID` and `WIFI_PASSWORD` so that the server can connect to your local network.

□ 3. Program the project to your second kit. Verify that the server connects to your network and begins listening for incoming connections.

Note: This server application runs both a non-secure (port 50007) and secure (port 50008) version of the AWEP protocol.

Client Setup

□ 1. Create a new application called **ch03b_ex01_client_secure** using the **Browse** button to select either your working client project from the last chapter or the solution project (either **ch03a_ex01_client** or **ch03a_ex02_client_response**) as a template.

□ 2. Copy the certificate/private key for the client and the root certificate used to validate the server's identity from the directory *ClassCerts/AWEP* into the application. Include the three files into the *tcp_client.h* file.

□ *Note: You will need to look in the files to find the #define names used for each certificate and key.*

□ 3. During initialization:

   a. Initialize the root certificate. This will allow the client to verify the server's identity.

   b. Initialize a TLS identity with the client's private key and certificate. This will allow the server to verify the client's identity.

□ 4. After creating the socket, add the call to enable TLS on the socket and set the authentication mode to `CY_SOCKET_TLS_VERIFY_REQUIRED`

Note: You must include cy_tls.h in the C source file to have access to the required TLS functions.

□ 5. Program and test the project to make sure your message is received by the server.

Exercise 2: Implement secure AWEP server

Implement the server side of the secure AWEP protocol.

Exercise 3: Implement dual secure & non-secure AWEP client and server

Implement a client for the AWEP protocol that can contact both non-secure and secure servers.

Implement a server for the AWEP protocol that will serve both non-secure and secure clients.

Note: *In order to call `cy_socket_listen` on two sockets at the same time, you will need to update the lwipopts.h config file. Copy lwipopts.h from mtb_shared/wifi-mw-core/<version>/configs to your projects root directory and change the variable `MEMP_NUM_TCP_PCB_LISTEN` from 1 to 2.*

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

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Published by

Infineon Technologies AG

81726 Munich, Germany

IMPORTANT NOTICE
The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics („Beschaffenheitsgarantie") .

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer's compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer's products and any use of the product of Infineon Technologies in customer's applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (**www.infineon.com**).

WARNINGS
Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.