SimpleLink™ Wi-Fi® Certificates Handling

The SimpleLink™ Wi-Fi® device provides a rich set of multi-layer security features that allow customers to provide best-in-class security for Internet-of-Things (IOT)-connected devices. The generation and handling of the certificates is a key part of maintaining the security of the devices and the overall system. This document provides an overview on the certificates, and describes the different types of certificates used by the CC31xx/CC32xx device. It also details the main use cases of generating, installing, and using the certificates. The document goes through a general description of the certificate features, then describes the specific CC32xx device support.

CC32XX: Information specific to the CC32xx device family appears in a bordered text box, such as this one.

Contents

List of Figures

List of Tables

Trademarks

SimpleLink is a trademark of Texas Instruments.
Wi-Fi is a registered trademark of Wi-Fi Alliance.

1 Certificate Overview

1.1 Concept

Public key cryptography, or asymmetrical cryptography, refers to any cryptographic system that uses pairs of keys: public keys which may be disseminated widely, and private keys which are known only to the owner.

This accomplishes two functions: authentication, when the public key is used to verify that a holder of the paired private key sent the message, and encryption, where only the holder of the paired private key can decrypt the message encrypted with the public key.

NOTE: How does a digital signature work?

The signer uses a one-way hash function on the original data (the result is known as the digest). The digest is then encrypted using the private key to create the signature. The signer provides the verifier with the original data, the signature, and the public key. The verifier uses the same one-way hash function on the original data to calculate the digest. The verifier then uses the public key to decrypt the signature and compares the result to the digest.

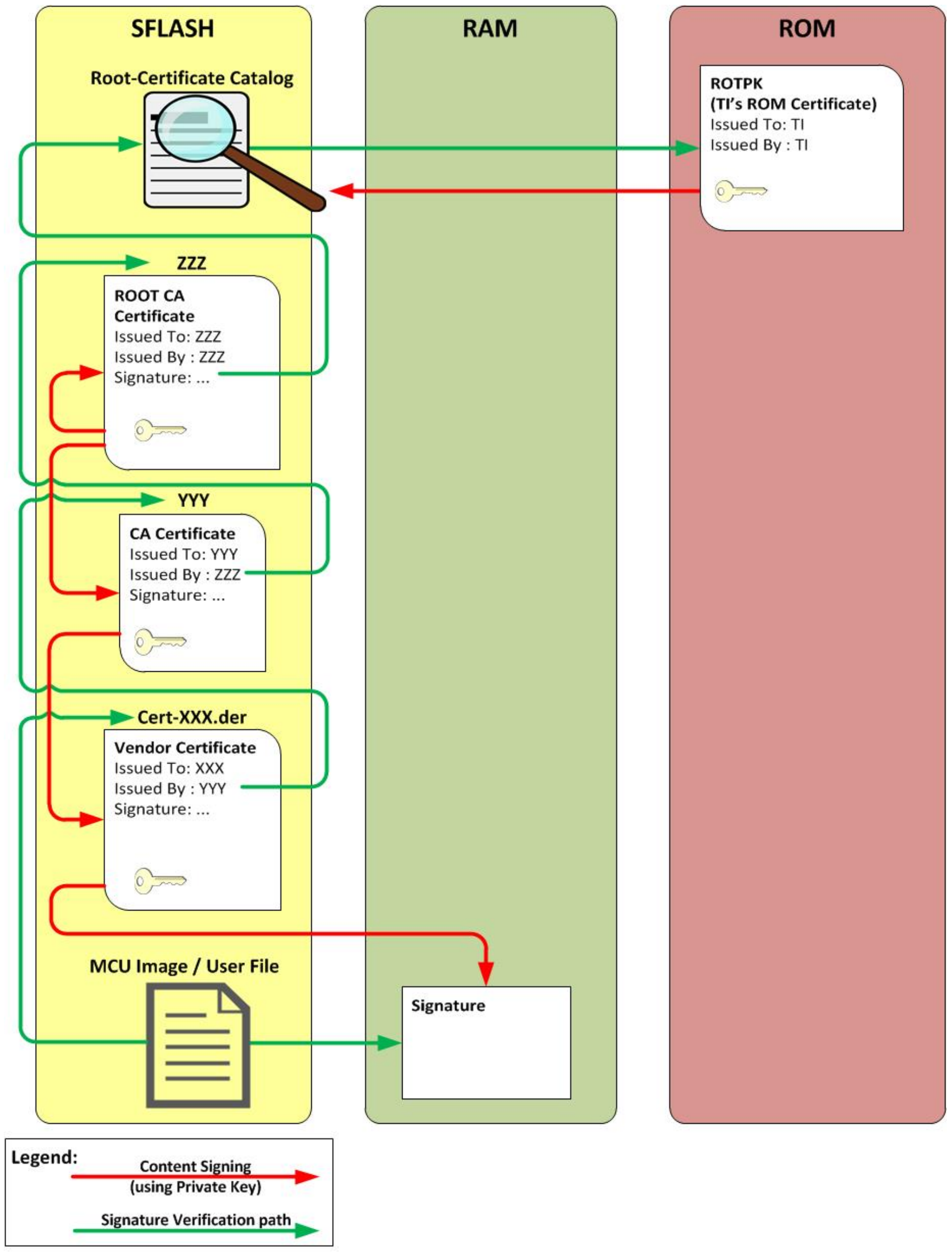
1.2 Certificate Chain

In computer security, digital certificates are verified using a chain of trust. The chain starts with the vendor certificate (also known as an identity or end-entity certificate). Each certificate in the chain is signed by the entity identified by the next (intermediate) certificate. When verifying a certificate, each signature along the chain is checked. The last element of the chain is the root CA (certificate authority) certificate which is signed by itself. The root CA is the trust anchor of the digital certificate. A certificate signed by a CA proves the ownership of a public key. A typical chain contains 3 certificates: the vendor (identity) certificate, an intermediate CA certificate (from which the customer obtained the root of trust), and the root CA certificate. This chain can also be extended with more intermediate certificates. The signatures of the root CA certificates are available within (and verified against) the certificate catalog.

NOTE: How is a certificate chain built?

Each certificate in a chain includes the owner’s info and public key. It is signed with the private key of the next certificate in the chain identified by the “Issued By” field (which can be an intermediate or root-CA). This issuer signature is included in the certificate. When authenticating a chain, each one of the issuer signatures is verified using the public key of the issuer. The root-CA is self-signed with its own key. A hash of the root CA's binary is stored in the certificate catalog.

CC32XX: The TI catalog (as well as the service pack) is signed using a TI private key and is verified by the public key inside the ROM code when the catalog is installed. The SimpleLink™ ROM code only verifies the authenticity of the certificate. The application must verify the end-entity information as it appears in the vendor certificate (such as company and domain info).



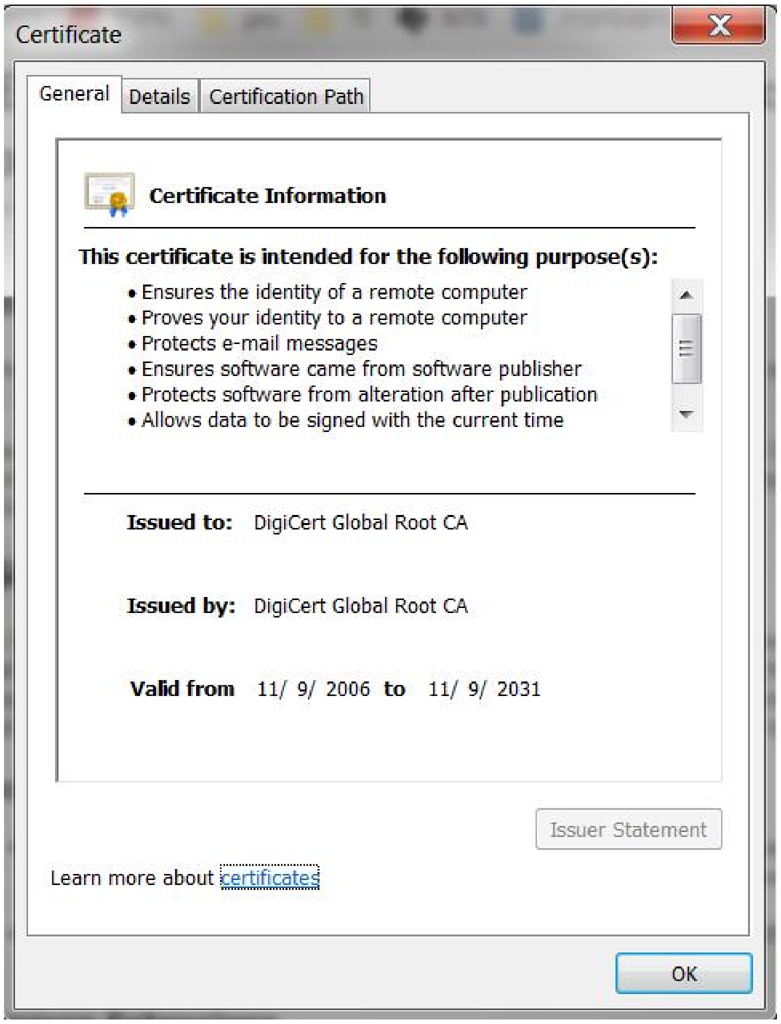
1.3 Certificate File Format

All certificates are based on the X.509 standard, but can be encoded in binary or ASCII (Base64) formats.

1.3.1 DER (*.der extensions) Format

DER format refers to a binary ASN1 DER encoded form compatible with the PKCS#10. Such certificate files can be opened in windows (if contains the ".der" extension) to view the certificate content, as shown in Figure 1.

Figure 1. DigiCert Global Root CA



Certificate Information

This certificate is intended for the following purpose(s):

- Ensures the identity of a remote computer
- Proves your identity to a remote computer
- Protects e-mail messages
- Ensures software came from software publisher
- Protects software from alteration after publication
- Allows data to be signed with the current time

Issued to: DigiCert Global Root CA

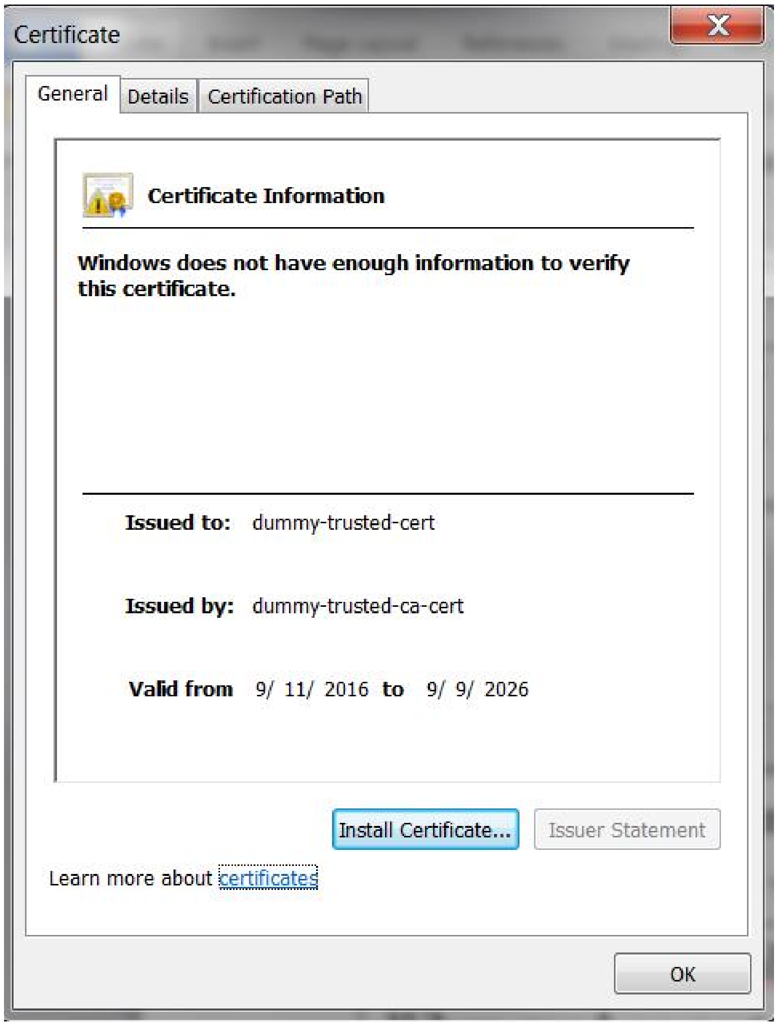
Issued by: DigiCert Global Root CA

Valid from 11/ 9/ 2006 to 11/ 9/ 2031

Learn more about certificates

The playground certificate files are given in DER format but without file extension (see the reason in the following NOTE). In order to view their content (in Windows), the ".der" suffix should be added.

CC32XX: The SimpleLink™ Wi-Fi® device supports the DER format for a specific certificate, but not as a full chain. When used as part of a chain, all the certificates (of the chain) must be added to the file system. The vendor (top level) certificate should be bound to the socket (with the sl_SetSockOpt command) when used in the TLS/SSL handshake, or to a specific file (as a parameter of sl_FsClose) with a signature that must be verified. The certificate file names of the rest of the chain must match exactly (without any extension) to the value of their “Issued To” field (below is an example of the “dummy-trusted-cert” from the playground environment). This value matches the “Issued By” of the next certificate of the chain – which enables the network processor to locate the entire chain’s files when starting from the vendor certificate. Unless the full path is stated by the “Issued To” field, the certificate files should be located under the root folder.



Certificate Information

Windows does not have enough information to verify this certificate.

Issued to: dummy-trusted-cert

Issued by: dummy-trusted-ca-cert

Valid from 9/ 11/ 2016 to 9/ 9/ 2026

Learn more about certificates

1.3.2 PEM (*.pem extension) Format

PEM format refers to an ASCII (Base64) encoded certificate, prefixed with "-----BEGIN CERTIFICATE-----" and suffixed with "-----END CERTIFICATE-----". It can be opened (but not parsed) by any text editor.

CC32XX: The SimpleLink™ Wi-Fi® device supports the PEM format for one certificate or for the entire chain. In the case of chained certificates, all the certificates are appended to each other (starting from the vendor certificate), as shown:

```
-----BEGIN CERTIFICATE-----
MIID8TCCAtmgAwIBAgIBATANBgkqhkiG9w0BAQsFADB/MQswCQYDVQQGEwJJTDEP
MA0GA1UECAwGU2hhcm9uMRAwDgYDVQQHDAdSYWFuYW5hMR8wHQYDVQQKDBZUZXhh
cyBJbnN0cnVtZW50cyBJbmMuMQwwCgYDVQQLDANSTkQxHjAcBgNVBAMMFWR1bW15
LXJldm9rZWQtY2EtY2VydDAeFw0xNjA5MTExNDE1MTFaFw0yNjA5MDkxNDE1MTFa
MHwxCzAJBgNVBAYTAklMMQ8wDQYDVQQIDAZTaGFyb24xEDAOBgNVBAcMB1JhYW5h
bmExHzAdBgNVBAoMFlRleGFzIEluc3RydW1lbnRzIEluYy4xDDAKBgNVBAsMA1JO
RDEbMBkGA1UEAwwSZHVtbXktcmV2b2tlZC1jZXJ0MIIBIjANBgkqhkiG9w0BAQEF
AAOCAQ8AMIIBCgKCAQEAmnUB5/TVmUR1nkisvaovBB0xZLvaLBJibcdyiGnOmPZH
MMe5j15dKiXefxteoIUE6TBV7Y9R86ca5WMVawWGSL0tkIwivUPlQbO2Yl5xD1zO
X6PO00zM+tkJcnXHVgGYLEo5uYakMZtphQzmcd8yW2GCF5lpDb9Z3PIuAbAgLH89
yvc2Jcw2MAapQ5N8NZeQd3hqthh4Vm2hTIEMck/AoN6sH4YjZLwiH1iAfTOS3swu
aWU2viXO+JWYIhqWzU9pnutfyaJ02gGi1g0i41X4iBUayrh+ib+bHd7L1NZAttM5
3Wce2UaV0vE/xuZldh7RToUZ3II01IDhooKuOzWi9wIDAQABo3sweTAJBgNVHRME
AjAAMCwGCWCGSAGG+EIBDQQfFh1PcGVuU1NMIEdlbmVyYXRlZCBDZXJ0aWZpY2F0
ZTAdBgNVHQ4EFgQUuvUnGR1fUnXRD+iPTjYJGyMojc8wHwYDVR0jBBgwFoAUNAVM
OO3F1jZ+QLIgFNU4+XbDNPcwDQYJKoZIhvcNAQELBQADggEBAHCymdOQ7idDvou2
OxcD6FlHxoo79miE5/6ZiaoBJMdA+rTRlHQFBh0u2GI+nfFu5DD/T97d4ow/kP1k
IdV6KGFnmvdEb3szkOerm64cFz2xvd8AIgOXze+iBmT40f/7sAM/3mXXGygUAcPE
jlENjbF1bXMf7eW76U3LmZv1Z+ZpqtzEMuNBw0wZb1PhKc30MbDgpqdCQ05x2zU8
Pb4pTHfpCZsyd4/aenxNzpN4yukEBG3jGhqTz+5DsL8IHFUW52dD3Ph8wp66NCdm
B9x3SJHe8thxlNOeumVEinmgfDExRYn0KVzMeWLzn8jM/nmwmXey4fvGwHsX+v8k
+MQBydg=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIDxjCCAq6gAwIBAgIBATANBgkqhkiG9w0BAQsFADB8MQswCQYDVQQGEwJJTDEP
MA0GA1UECAwGU2hhcm9uMRAwDgYDVQQHDAdSYWFuYW5hMR8wHQYDVQQKDBZUZXhh
cyBJbnN0cnVtZW50cyBJbmMuMQwwCgYDVQQLDANSTkQxGzAZBgNVBAMMEmR1bW15
LXJvb3QtY2EtY2VydDAeFw0xNjA5MTExMzUwNTJaFw0yNjA5MDkxMzUwNTJaMH8x
CzAJBgNVBAYTAklMMQ8wDQYDVQQIDAZTaGFyb24xEDAOBgNVBAcMB1JhYW5hbmEx
HzAdBgNVBAoMFlRleGFzIEluc3RydW1lbnRzIEluYy4xDDAKBgNVBAsMA1JORDEe
MBwGA1UEAwwVZHVtbXktcmV2b2tlZC1jYS1jZXJ0MIIBIjANBgkqhkiG9w0BAQEF
AAOCAQ8AMIIBCgKCAQEAufQB7A2FhBXjvTgHkJr/DHwyVtn0g0A/tq06+0YB7LR6
VgHj+/2a/JVhgXPTjY809W3SvhXPCLSBWgHHkj6hXbmUUHhDTE6B3QCNjoHxRBSt
pD1aOPcig0V7e4ofoQ4u4ORod+Jmgig9pN1kSa8kGhbGDo1+xrrt7C3CZiyHDtCB
VFaR/C34vw/fcCQrCmoAZuIx5dl9jdinOQJUQdODifBOOG6QXh4B8fl298aNt3l9
hupICrhjEAxSFsZo1EE5l6DALrYsVcN5VCCLGMsffVX3bDoIowEi8rL4PPsQLJWh
WmlbYxZs5YzMeuj7EZa9QBoxsqZYFmPpeV8PAVoHywIDAQABo1AwTjAdBgNVHQ4E
FgQUNAVMOO3F1jZ+QLIgFNU4+XbDNPcwHwYDVR0jBBgwFoAU/ctvc/HkJk5Wpz4n
oopsPF7GsTMwDAYDVR0TBAUwAwEB/zANBgkqhkiG9w0BAQsFAAOCAQEASkBLp7yk
CQAXBHXfFCHpYVJG9NKbWmT30IH4M82Gm6+rsukvDF4Kgr++Tp0QFj7a7uIMTnTp
1whFnckArWbe3JOe9L1PYlszKdLgwNps29jGKVinvkBJ7QPQdiq0qR/x7k3tFgT1
h/3x4CgLHA8wMMDseoQR7Qh/5tJ7VkfP0PDJyJdNfcWzgahVORE4w6ql00LusbZb
ffwpJm3135kSLgizs3+y5gDJkbMHU8Q34h2CQcygAyjMJFwwxQT9wRTrffUgUvZn
SuGwBvCdIqh1fWOqsVTOlvX6JZvdIz/yioVznuAEGruqrYG4JTHyThjK2EQJM7ur
LKKR5OdYhohKoQ==
-----END CERTIFICATE-----
```

1.3.3 Other Certificate File Extensions (*.crt, *.cer, *.key)

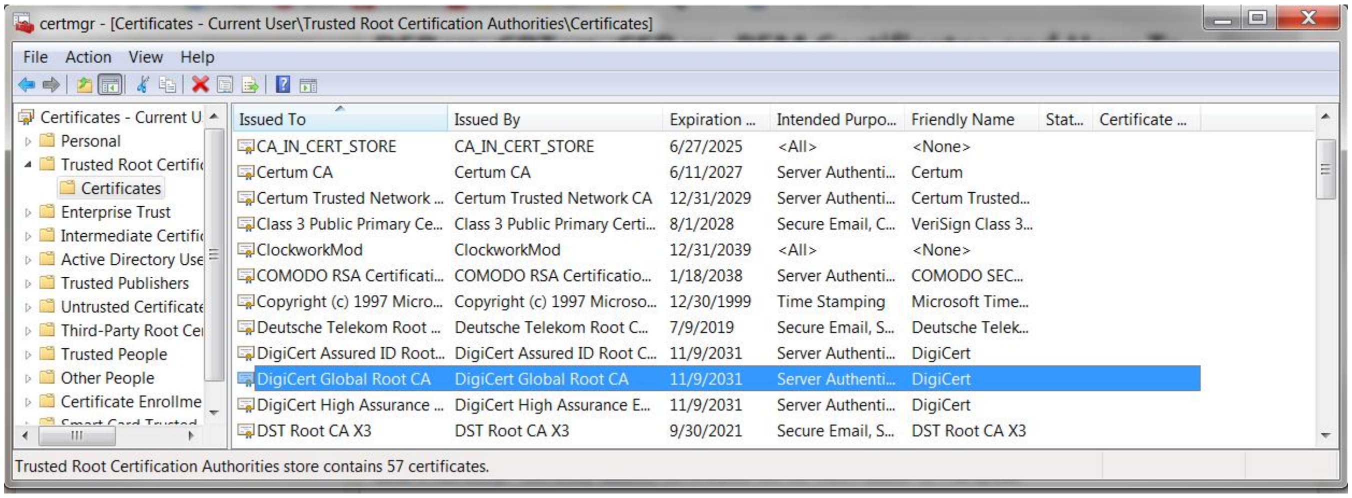
Certificate files that use other file extensions can be encoded as DER or PEM. OpenSSL can be used to convert one format to the other, see [4].

CC32XX: In most cases (SSL and code signing), both formats are supported by the SimpleLink™ Wi-Fi® device. The only exception is the SSL chained certificates, which must be programmed as a PEM file. In this case, the PEM file includes the chain of trust, starting from the vendor certificate but excluding the root CA certificate (the figure in the previous note is an example of the playground chain in PEM format). The file extension is not important, as the SimpleLink™ Wi-Fi® identifies the format by the content.

1.4 Certificate Catalog

A certificate catalog (or certificate store) is a list of approved root CA certificates used to authenticate a given certificate. A similar catalog is used by any OS or browser (for example in Windows, click Start and type “certmgr.msc” – then check the “Trusted Root Certification Authorities\Certificates\”, as shown in Figure 2).

Figure 2. Windows Certificates Manager


certmgr - [Certificates - Current User\Trusted Root Certification Authorities\Certificates]

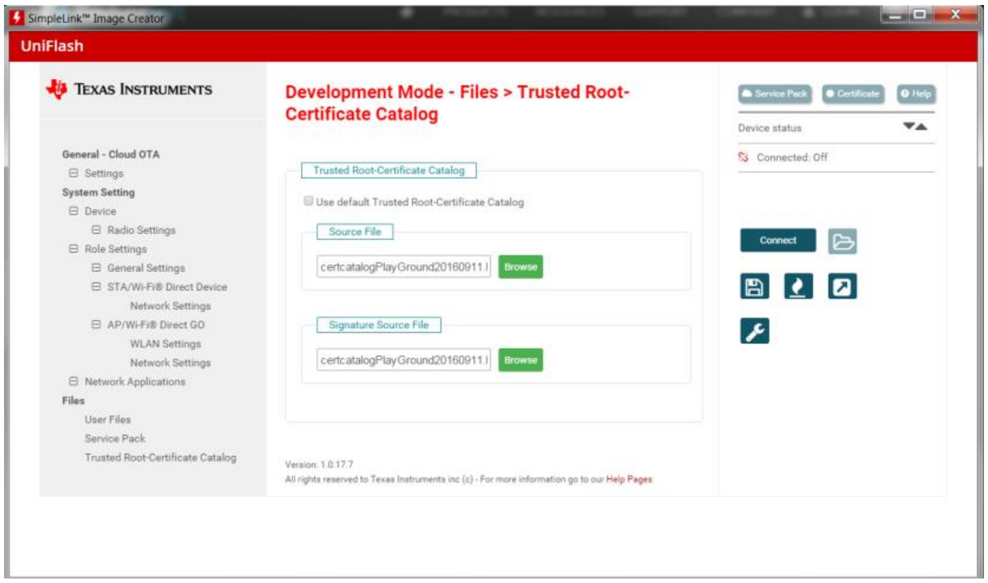
File Action View Help

Certificates - Current U
- Personal
- Trusted Root Certific
  - Certificates
- Enterprise Trust
- Intermediate Certific
- Active Directory Use
- Trusted Publishers
- Untrusted Certificate
- Third-Party Root Cer
- Trusted People
- Other People
- Certificate Enrollme

Issued To	Issued By	Expiration ...	Intended Purpo...	Friendly Name	Stat...	Certificate ...
CA_IN_CERT_STORE	CA_IN_CERT_STORE	6/27/2025	<All>	<None>		
Certum CA	Certum CA	6/11/2027	Server Authenti...	Certum		
Certum Trusted Network ...	Certum Trusted Network CA	12/31/2029	Server Authenti...	Certum Trusted...		
Class 3 Public Primary Ce...	Class 3 Public Primary Certi...	8/1/2028	Secure Email, C...	VeriSign Class 3...		
ClockworkMod	ClockworkMod	12/31/2039	<All>	<None>		
COMODO RSA Certificati...	COMODO RSA Certificatio...	1/18/2038	Server Authenti...	COMODO SEC...		
Copyright (c) 1997 Micro...	Copyright (c) 1997 Microso...	12/30/1999	Time Stamping	Microsoft Time...		
Deutsche Telekom Root ...	Deutsche Telekom Root C...	7/9/2019	Secure Email, S...	Deutsche Telek...		
DigiCert Assured ID Root...	DigiCert Assured ID Root C...	11/9/2031	Server Authenti...	DigiCert		
DigiCert Global Root CA	DigiCert Global Root CA	11/9/2031	Server Authenti...	DigiCert		
DigiCert High Assurance ...	DigiCert High Assurance E...	11/9/2031	Server Authenti...	DigiCert		
DST Root CA X3	DST Root CA X3	9/30/2021	Secure Email, S...	DST Root CA X3		

Trusted Root Certification Authorities store contains 57 certificates.

CC32XX: The SimpleLink™ Wi-Fi® catalog is provided as a data list file (“*.lst”) and comes with a signature (“*.lst.sign”) signed by TI’s private key. The certificate file is written as any other file, but the network processor (ROM-based code) identifies it by its reserved name (“certstore.lst”) and then verifies the signature using the (ROM-based) TI certificate. Only after the signature is verified, the file will be written securely to the file system. The Uniflash tool contains a specific tab for the “Trusted Root-Certificate Catalog” (see below), where the user can provide the catalog and signature file in their original names. The tool must write the file using the reserved name.



Development Mode - Files > Trusted Root-Certificate Catalog

Trusted Root-Certificate Catalog

☐ Use default Trusted Root-Certificate Catalog

Source File: certcatalogPlayGround20160911.l

Signature Source File: certcatalogPlayGround20160911.l

Version: 1.0.17.7
All rights reserved to Texas Instruments inc (c) - For more information go to our Help Pages

Due to resource limitations, the SimpleLink™ catalog does not contain the actual root CA certificates, but only hash values based on the binary format. The user must install the relevant root CA certificates before it is used. For example, to connect to a cloud server, the server certificate should be identified and the corresponding root CA certificate added to the file system and then linked to the client socket (using sl_SetSockOpt). Only then, the sl_Connect may be called. See more details on this process in Section 1.5.2.

1.4.1 The CC32xx Certificate Catalog

The certificate catalog is a supported in the secured SimpleLink™ Wi-Fi® devices (CC32xxS and CC32xxSF). The TI official certificate catalog is available within the SDK (<SDK-ROOT>\tools\cc32xx_tools\certificate-catalog\). The certificate catalog holds the entire trusted root certificate known to the SimpleLink™ CC3120/CC3220 device. The catalog may change over time and must be programmed to the device’s file system. The catalog holds only fingerprints of the original root CA certificate; the certificate itself must be programmed to the file system when used.

CC32XX: The full list of root certificates supported by the catalog can be found within the SDK in <SDK-ROOT-DIR>/ tools/cc32xx_tools/certificate-catalog/readme.html.

1.4.2 The Playground Certificate

The playground certificate is provided as a tool for practicing the certificate handling during development. The “certificate-playground” folder contains both the self-signed certificate chain (“dummy-trusted-cert”, “dummy-trusted-ca-cert”, and “dummy-root-ca-cert”) and the playground certificate catalog (which includes only the dummy root ca signature). The folder also contains the associated private keys – which makes it a critical security breach for a real product. Thus the playground certificates must NOT be used in production mode. Using the playground catalog and certificates enables customers to install an MCU image on a secured CC3220 device (with the playground as code-signing certificate) and to emulate secured connections (where both server and clients use the proprietary playground certificate as SSL/TLS certificates). It is also possible to connect to a cloud vendor (that uses a valid certificate) and disable or ignore the security warning (SL_ERROR_BSD_ESECUNKNOWNROOTCA), due to the usage of the dummy catalog. Though it is demonstrated with the playground in our examples, using the same certificates for both code-signing and SSL connection is not recommended (see Section 1.5 for more details on the different types of certificates).

1.4.3 Vendor-Specific Certificate Catalog

As part of the multi-layer security approach of the SimpleLink™ Wi-Fi® devices (CC32XXS/SF) maintain a secure boot sequence, file system security features, and secure sockets. These security features perform authentication to verify that a signature received from trusted chain. TI™ publishes a common trusted certificate list, which is used to authenticate the trusted chain. Vendors that are required to use their own trusted chain should use their own certificate catalog.

The Vendor Device Authentication With SimpleLink™ Wi-Fi® Devices User's Guide describes how to authenticate an application that is signed by vendor’s certificate using his own certificate catalog, how to store and use it, and how to update it over the air if needed.

1.5 Certificate Types

The secure operation of the SimpleLink™ Wi-Fi® requires the use of two types of certificates: code signing certificates and TLS/SSL (server and client) certificates. Both TLS/SSL certificates and code signing certificates are used to authenticate the identity of a digital data source. However, it is important to be aware of the major differences between them. This section provides a short overview on each certificate type, the specific use cases, and the related SimpleLink™ requirements.

1.5.1 Code Signing Certificate

A code signing certificate is used to sign a piece of digital data (such as code or user files). At a minimum, a company needs one valid certificate (that is, not per device) to sign any digital data it owns. The private key must be stored securely by the owning company. A code signing certificate does not have a limited validity period (there is no expiration date) and can be used as long as the private key is not compromised. The public certificate must be stored on each device (in the production line) before installing any other signed file, and may be encrypted. Similar to all other certificate files, it can be later updated in an over-the-air (OTA) process.

CC32XX: A valid code signing certificate is mandatory when using the Secure SimpleLink™ Wi-Fi® devices (CC3X20S/SF). Secured files are authenticated only when they are written to the file system. The sl_FsClose call (when writing a secured and signed file) should include the path to the certificate and the file signature as a binary ‘C’ array. After processing the sl_FsClose, the signature is verified. If the verification fails, the file is not flashed. The secure devices require that the MCU image is signed and authenticated. Other important files may also use the signature. The code signing certificate chain is verified against the installed catalog, which means that the root of trust (root CA certificate) of the chain must be included in the catalog (refer to the hash value of the root CA certificate). The user cannot override this verification (such as writing a secured file with a self-signed certificate). Unless used against the playground catalog during development, the code signing certificate must be purchased from a valid certificate authority (one that is based on a root CA certificate from the TI Catalog – see The CC32xx Certificate Catalog). Refer to Section 2.1 for more details on the procedure to obtain a certificate.

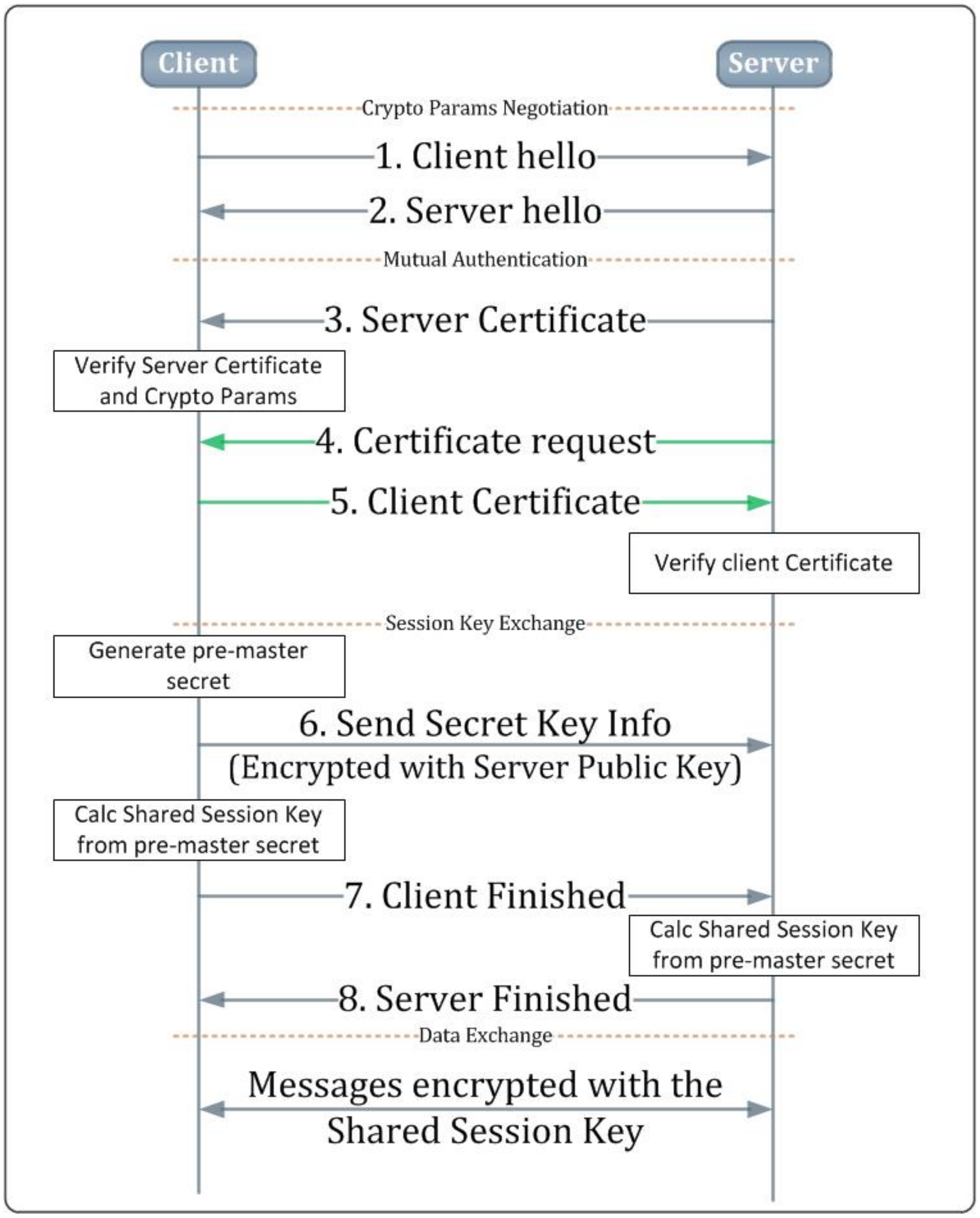
IMPORTANT: The production line programming (such as through Uniflash) works with the device boot (ROM) code that only supports the following signature types: PKCS#1, RSA 256 or 128 bytes, SHA_1 (the signature length is 256 or 128 bytes). Code signing certificates that use SHA-384 or SHA-512 (or other unsupported method) for the signature algorithm will cause the chain verification to fail with an "FS_SECURITY_ALERT_CERT_CHAIN_ERROR" alert. The newer SHA methods are supported by service packs, so such certificates can be used during OTA but not during the production line. This limitation is relevant for the user and intermediate certificates (the signature algorithm of the root CA certificate does not impact the chain verification, so any method will work).

TI uses its own private key for signing the service packs and certificate catalog files. The public key for TI content is stored in the ROM. This ensures that both the service pack and certificate catalog used on a SimpleLink™ Wi-Fi® device always originate from TI.

1.5.2 TLS/SSL Certificates

The TLS/SSL private key is typically unique for each device, and must be stored securely on the device. The public certificates are exposed during the TLS/SSL handshake (upon connection setup). The TLS/SSL certificates are used both to authenticate the identity of each of the parties (server or client) and to encrypt the session key. Each TLS/SSL certificate in a chain contains a validity period (start and end dates). Replace the certificates before they expire (such as by using OTA). This is critical for certificates that enable the OTA process itself. Figure 3 depicts the TLS/SSL setup sequence.

Figure 3. TLS/SSL Handshake Overview


Client
Server
Crypto Params Negotiation
1. Client hello
2. Server hello
Mutual Authentication
3. Server Certificate
Verify Server Certificate and Crypto Params
4. Certificate request
5. Client Certificate
Verify client Certificate
Session Key Exchange
Generate pre-master secret
6. Send Secret Key Info (Encrypted with Server Public Key)
Calc Shared Session Key from pre-master secret
7. Client Finished
Calc Shared Session Key from pre-master secret
8. Server Finished
Data Exchange
Messages encrypted with the Shared Session Key

1. The mutual authentication begins after the initial crypto protocols negotiation (“Hello” messages - 1, 2).
2. The server sends its owns “Public Certificate” (3) with domain info (this message is signed using the server’s private key).
3. The client authenticates the server certificate (this includes checking of the certificate expiration date) and signature.
4. Client authentication is optional:
   - The server may request the client authentication (4).
   - If required, the client replies with its own certificate (5) and signs it with its private key.
   - The server authenticates the client certificate and signature.
5. The client creates a pre-master secret and sends it to the server (6), encrypted with the server public key.

6. The server decrypts the pre-master secret with its private key.
7. Both parties send “Finished” messages (7, 8).
8. Both server and clients then generate the session keys from the pre-master secret. The symmetric session keys are used to encrypt the following transactions.

1.5.2.1 TLS/SSL Server Certificate

A TLS/SSL server must hold a server certificate and a server private key. The certificate may be unique per device (but can be shared for a product, such as to enable access from specific mobile application). The server certificate that is passed to the client during connection setup should include the entire chain, excluding the root CA certificate. A TLS/SSL client must hold the server’s root CA certificate. An SSL server certificate has two roles: first, to authenticate the server (see message 3 in Figure 3). Second, to encrypt the session (symmetric) keys (see message 6 in Figure 3). A server must also bind its private key to the socket (this is used to sign the authentication message).

CC32XX: A CC32xx TLS/SSL server should install its certificate chain in a chained PEM file format. See the example in the PEM format section. The server should install the private key in either PEM or DER format. A CC32xx TLS/SSL client should install the server’s root CA certificate in PEM or DER format. Instructions for binding the certificates (by client or server) to a socket can be found in the Enabling SSL Certificate chapter. The server certificate should be purchased from a valid certificate authority to be recognized by a secure client (such as a web browser). It should be updated before it is expired. Refer to Section 2.1 for more details on the certificate generation procedure.

1.5.2.2 TLS/SSL Client Certificates

A TLS/SSL client may be required to hold a client certificate (and private key) to connect to a cloud server that requires client authentication (such as AWS). The client certificate should be unique per device. The client certificate should include the entire chain of trust, excluding the root CA certificate. A TLS/SSL server that requires client authentication must hold the clients’ root CA certificates. A TLS/SSL client certificate is only used for authenticating the client (see message 5 in Figure 3). This is an optional TLS/SSL feature. If the client authentication is requested during a TLS/SSL negotiation, the client should provide the server with its entire certificate chain, except for the root CA certificate (which should be known to the server to complete the authentication). If the client authentication is requested, the client must also bind its private key to the socket (this is used to sign the authentication message).

CC32XX: A CC32xx TLS/SSL client should install its certificate chain (if needed) in a chained PEM file format. See the example in the PEM format section. The client should install the private key (if needed) in either PEM or DER format. A CC32xx TLS/SSL server should install its clients’ root CA certificates (if needed) in PEM or DER format. Instructions for binding the certificates (by client or server) to a socket can be found in the Enabling SSL Certificate chapter. The server must provide instructions for clients on how to generate their certificates. It typically does not involve an engagement with external certificate authority or any payment. In most cases, this is part of the first registration of the client by the server. Refer to Section 2.1 for more details on the certificate generation procedure.

1.5.3 Certificates Types Summary

Table 1. Certificate Types Summary

Certificate Type	Is It Mandatory?	Purchase Needed?	Number of Certificates Needed	Expiration Date
Code signing	Yes (for CC3220S/SF)	Yes	1 (company certificate)	No
TLS/SSL server	If implementing TLS/SSL server	Yes	Per device or per application	Yes
TLS/SSL client	If connecting to TLS/SSL server that requires client authentication	Typically not (provided by the server)	Per device	Yes

2 Certificate Procedures

2.1 Generating a Certificate

To generate a new certificate (of any type), it is required to create a private key and a Certificate Signing Request (CSR). The CSR is required by the certificate authority for creating the certificate chain. When the CSR is generated, it is signed by the private key. The private key must be stored securely and is not handed to the certificate authority. The open source OpenSSL project (see details in [4], provides a tool set for generating and handling keys and certificates. The following OpenSSL command is an example that creates the private key and the CSR:

```
> openssl req -newkey rsa:2048 -sha256 -nodes -keyout out.key -out out.csr
```

After running the command, answer the CSR information prompt to complete the process. The CSR information is customer-specific and includes company and domain info. The cryptographic parameters used here (such as “rsa:2048” for generating 2048 bit RSA key or “–sha256” for using SHA-2 hash) are given as an example. See [4] for details using of other options. The following is another example that generates a CSR from an available private key:

```
> openssl req -key in.key -new -out out.csr
```

The CSR contains the public key and the domain information (as entered above) encrypted with the generated private key. To view a parsed version of the CSR content, use the following command:

```
> openssl asn1parse -in my.csr -inform PEM -i
```

When generated, the CSR must be provided to a certificate authority to generate a complete chain of trust. When generating an SSL client certificate, the server typically provides a method for retrieving the client certificate (without purchasing one from an external certificate authority). This usually happens when the client registers with the server (this process can happen offline without involving the device). In most cases, this method still involves the generation of a private key and a CSR, as shown earlier. A major concern in this process is that the private key must be kept secured after it is generated. It should not be sent to the certificate authority or to any other party. For code signing purposes, one key may be used to sign all the company’s important files (at the minimum, a Secured SimpleLink™ Wi-Fi® device requires that the MCU image is signed and authenticated). The private key used for code signing must be stored securely by the company (not on the device) and used to sign new content before it is published. For TLS/SSL purposes, the private keys are unique per device, and should be installed on the device at some stage without compromising its privacy.

CC32XX: The SimpleLink™ Wi-Fi® offers a method that uses the unique device key to create a CSR on the device itself without compromising the privacy of the key (the key is only used for signing content internally and is never exposed). Section 3 introduces the method that can replace the OpenSSL method described above.

This method still requires that the generated CSR is sent to a certificate authority for obtaining a valid chain of trust (Section 3.2 suggests two methods to automate the acquiring of device certificates from a CA using the auto-generated CSR.

2.2 Installing the Certificates

Certificates are treated like any other user file. They may be stored encrypted. The certificate files may be installed in the production line (through the UART, for example using the Uniflash tool or through host programming) or in run-time (through the SimpleLink File System API, for example as part of an OTA update). When installing a DER-formatted certificate chain, the filename of the root and intermediate certificates (that is, all but the vendor certificate) should match exactly the value of the certificate’s “Issued To” field. The “Issued To” value makes a full path that is relative to the file system’s root folder (unless the field value contains subfolders, the certificates should be located in the root folder). The vendor (identity) certificate can have any path or name as long as the full path is referenced when it is used in the application code or in Uniflash.

NOTE: How to program the certificates to the file system?

The SimpleLink™ Wi-Fi® supports two methods for programming the flash:

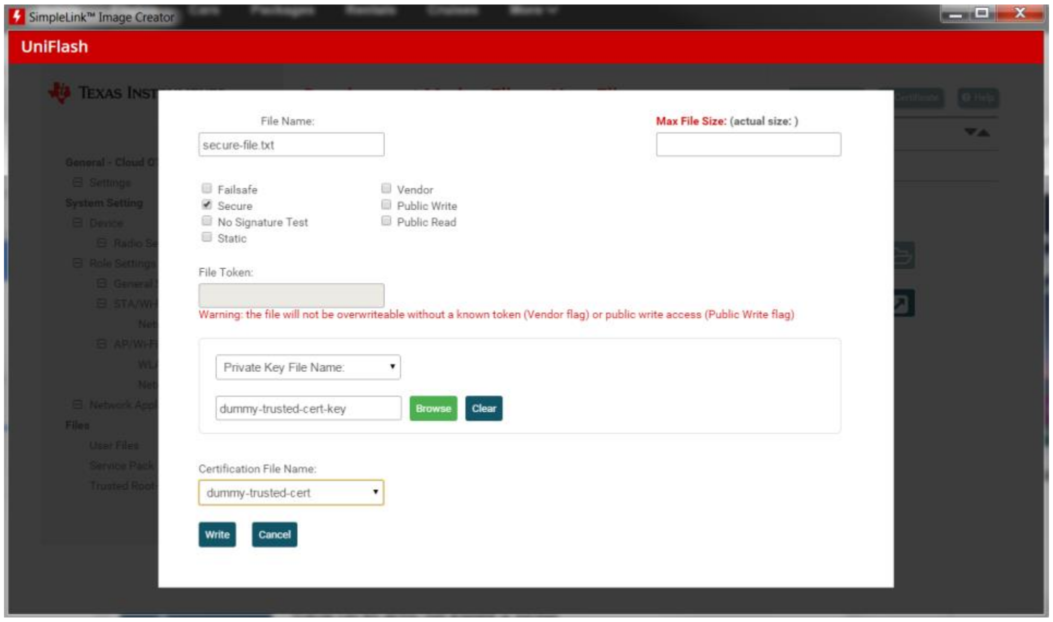
- As a full image using the sl_FsProgram command (“host Programming”) or through external programming interface (for details see Section 8.2 in [2]). This method is typically used in the production line. The image is created using the Uniflash and is installed as one binary to the flash. The image contains both the files and their metadata (such as the signature) in a proprietary format. The image is extracted to the file system by the boot ROM code.
- File-by-file using the standard file-system API (sl_FsOpen, sl_FsWrite, sl_FsClose). This is used, for example, in OTA updates. The SimpleLink™ OTA library supports a specific tar format (defined in the OTA application notes) that includes a list of files with their metadata. Such an OTA image can be created using the Uniflash.

2.3 Signing a File

2.3.1 Creating an Image With Uniflash

When using the Uniflash to create an image (either a programmable image or an OTA image), the signature can be generated instantly. If the MCU Image or secure user files are provided with the private key (see Figure 4) – the Uniflash will sign the file upon the image creation. The signature is provided when the file is written to the flash (the private key is only used to generate the signature and is not copied to flash).

Figure 4. Uniflash User File With Private Key



File Name: secure-file.txt

Max File Size: (actual size:)

[ ] Failsafe
[✓] Secure
[ ] No Signature Test
[ ] Static
[ ] Vendor
[ ] Public Write
[ ] Public Read

File Token:

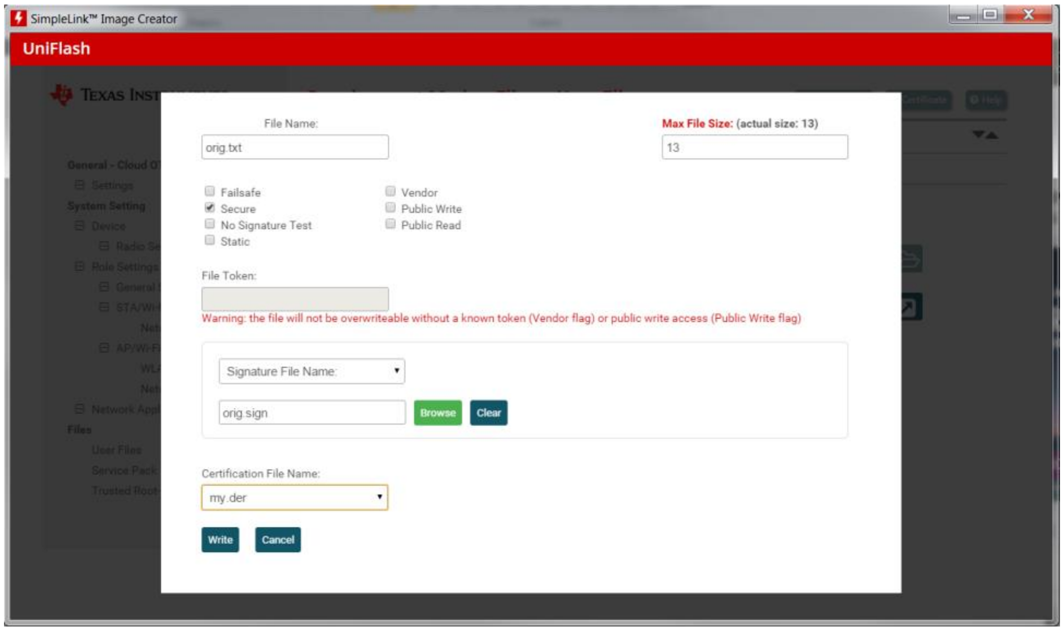
Warning: the file will not be overwriteable without a known token (Vendor flag) or public write access (Public Write flag)

Private Key File Name: dummy-trusted-cert-key

Certification File Name: dummy-trusted-cert

Alternatively, the user can generate the signature offline, as shown in Section 2.3.2, and provide the signature with the file (instead of the private key), as shown in Figure 5.

Figure 5. Uniflash User File With Signature



File Name: orig.txt

Max File Size: (actual size: 13) 13

- [ ] Failsafe
- [✓] Secure
- [ ] No Signature Test
- [ ] Static
- [ ] Vendor
- [ ] Public Write
- [ ] Public Read

File Token:

Warning: the file will not be overwriteable without a known token (Vendor flag) or public write access (Public Write flag)

Signature File Name: orig.sign

Certification File Name: my.der

When a file with a signature is used in an OTA, the signature should be written within the OTA image so that it can be used to authenticate the original file. In the SimpleLink™ OTA library implementation, the signature (in ASCII base64 format) is written among other file properties within the “ota.cmd”.

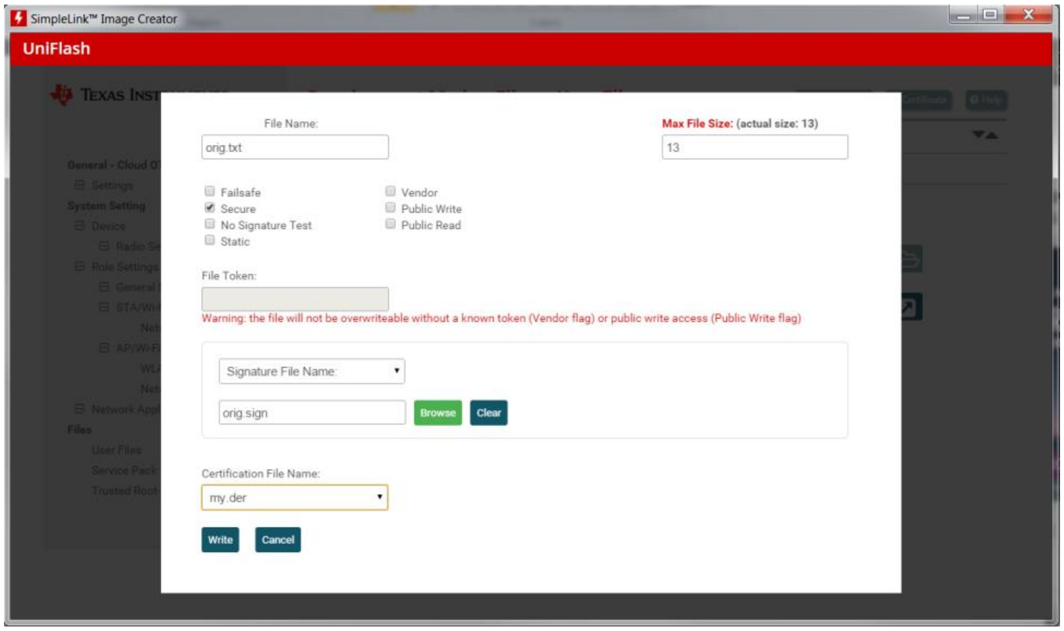
2.3.2 Offline Signature

The user is responsible for signing any important digital content with their private (code signing) key. The following OpenSSL command demonstrates a file signing:

```
> openssl dgst -hex -c -sha1 -sign my.key -out orig.sign orig.txt
```

This example uses the private key ("my.key") that was generated in Section 2.1, where orig.txt – is the input file, orig.sign is the resulting signature. Alternatively, the user can use the Uniflash Tools (see Figure 6) to sign a file by providing the source file and a private key (see Figure 7).

Figure 6. Uniflash Tools



File Name: orig.txt

Max File Size: (actual size: 13) 13

- [ ] Failsafe
- [✓] Secure
- [ ] No Signature Test
- [ ] Static
- [ ] Vendor
- [ ] Public Write
- [ ] Public Read

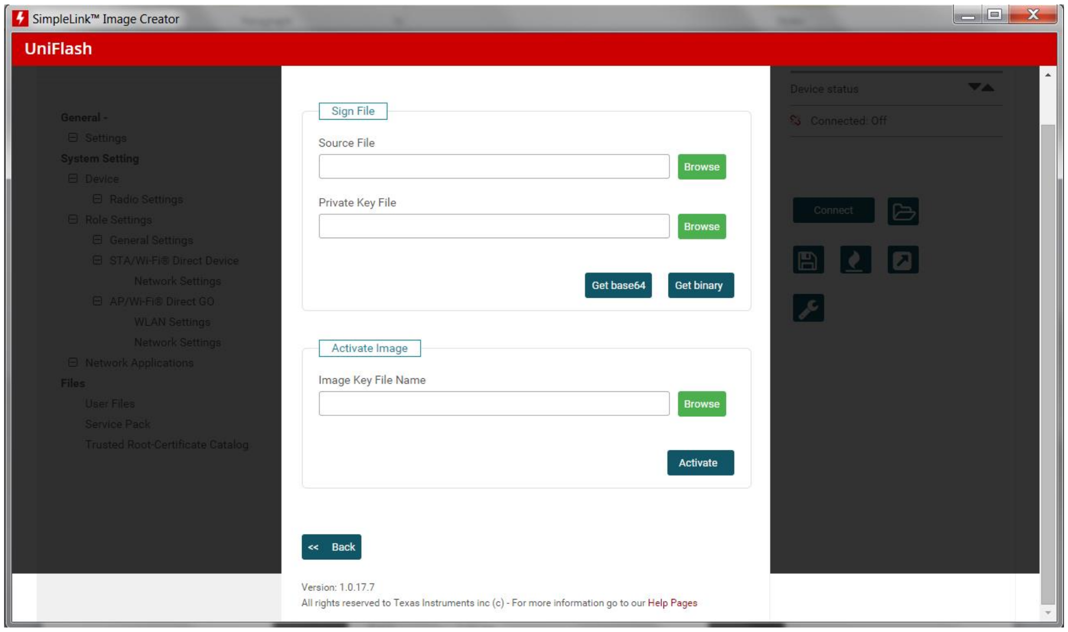
File Token:

Warning: the file will not be overwriteable without a known token (Vendor flag) or public write access (Public Write flag)

Signature File Name: orig.sign

Certification File Name: my.der

Figure 7. Uniflash Certificate Signing Tool



Version: 1.0.17.7

All rights reserved to Texas Instruments inc (c) - For more information go to our Help Pages

2.4 Verifying a File Signature

The signature is verified when a file is written to the file system. More specifically, when a secure file is closed (using sl_FsClose) after it was opened for writing. When writing the file to the file system, both the certificate and the signature must be provided with the sl_FsClose, as shown in the following code example:

```
#define CODE_SIGNING_CERT     “my-cert.der”
unsigned char Signature[] =  {
 0x28, 0xfa, 0x72, 0xea, 0x9e, 0xfe, 0xd2, 0x8c, 0x00,
    0x72, 0x04, 0xd4, 0x02, 0xfb, 0x35, 0x16, 0x31, 0xc9, 0x6d, 0xd8, 0x6d, 0xc5,
    0x26, 0x8e, 0x75, 0xe4, 0x36, 0xb2, 0xf3, 0xdf, 0xac, 0x1d, 0x5a, 0x4f, 0x80,
    0x46, 0xf4, 0x1e, 0x65, 0xda, 0xc0, 0x05, 0xbf, 0x29, 0xff, 0x79, 0x6d, 0x7d,
    0xa4, 0x29, 0x79, 0x54, 0xd5, 0x76, 0x58, 0x57, 0x5f, 0xf4, 0xf9, 0x53, 0x7b,
    0x48, 0x6b, 0x1c, 0x7c, 0xd9, 0xb1, 0x8e, 0xdf, 0x92, 0xc0, 0x4e, 0x86, 0x0e,
    0x0a, 0xa9, 0x1c, 0x76, 0xf0, 0x53, 0xf1, 0xef, 0xd2, 0x4a, 0xb9, 0xc6, 0xea,
    0xb1, 0xd8, 0x11, 0xdf, 0x33, 0x4d, 0x74, 0x1a, 0xbe, 0xb8, 0x68, 0xae, 0xee,
    0x2d, 0x5a, 0xb3, 0xa5, 0x12, 0x56, 0x96, 0x44, 0x53, 0xea, 0x3d, 0xe8, 0x5b,
    0xb8, 0x85, 0x75, 0x70, 0xb9, 0x34, 0x50, 0xb0, 0xa5, 0xaf, 0x6a, 0x8d, 0xde,
    0x4e, 0xe0, 0x15, 0x37, 0x2d, 0xe0, 0x39, 0xd4, 0xfa, 0x13, 0x73, 0x3a, 0x21,
    0x9d, 0x16, 0xaa, 0xf3, 0xb8, 0xf0, 0x9b, 0x76, 0xfc, 0xc3, 0x7d, 0xee, 0xd4,
    0x5e, 0x3c, 0x9c, 0x0e, 0xe7, 0xde, 0x88, 0x0d, 0x2f, 0x93, 0xe4, 0xb0, 0x42,
    0xe9, 0x94, 0x10, 0x69, 0x3b, 0x4b, 0xb6, 0x06, 0x0d, 0x87, 0xd1, 0x74, 0x6e,
    0x6c, 0xf4, 0x85, 0x6a, 0xf5, 0xa3, 0xac, 0x76, 0xe7, 0xb1, 0x86, 0xc3, 0xed,
    0x69, 0x7a, 0x11, 0x71, 0x10, 0x26, 0xb9, 0xbc, 0x22, 0x71, 0x1e, 0x25, 0xb1,
    0xc6, 0xec, 0xd4, 0xe0, 0x61, 0xa4, 0x7b, 0x26, 0x5a, 0xbf, 0x70, 0xbf, 0x62,
    0x5e, 0xbf, 0x6f, 0xdc, 0x34, 0xdf, 0x89, 0x6a, 0x87, 0xf3, 0x6f, 0x37, 0x75,
    0x2e, 0xa9, 0xa4, 0x67, 0xf6, 0x90, 0x97, 0xee, 0x36, 0xfb, 0x12, 0x5f, 0x23,
    0xe7, 0x98, 0xa1, 0xf9, 0x4a, 0xe2, 0x48, 0xdc, 0xa1, 0x94, 0x6d, 0x81, 0x59
};

RetVal = sl_FsClose(fd, CODE_SIGNING_CERT, Signature, sizeof(Signature));
```

When using Uniflash to program an image, the signature may be generated quickly, as explained in Section 2.3, and is included in the programmable image. The programmable image should also contain the associated code signing certificate as a user file (or chained files). The signature is then verified when the image is activated by the ROM based boot code. This includes authentication of the code signing certificate against the stored catalog. For all signed files that are updated through an OTA, the OTA image must include both the new file and the signature. The associated certificate may be updated within the OTA image as a user file or chained files. Otherwise, an already available certificate can be used. If a new certificate is provided within the OTA image, it must be written to the file system before any file that uses it. The signature is then verified when the file is written (closed) to the flash. The SimpleLink™ OTA library handles this internally. For more information and a detailed example of use, see [6].

2.5 Connecting to a Cloud (TLS/SSL) Server

Connecting to a TLS/SSL server typically does not require the client to purchase any certificate. The server’s root CA is typically available for free, and a client certificate, if needed, can be acquired from the server. A TLS/SSL connection requires that the client has the server’s root CA. Section 2.5.1 and Section 2.5.2 provide details on how to identify, obtain, and install the server root CA. A TLS/SSL server may require client authentication (see details in Section 2.5.3 and Section 2.5.4).

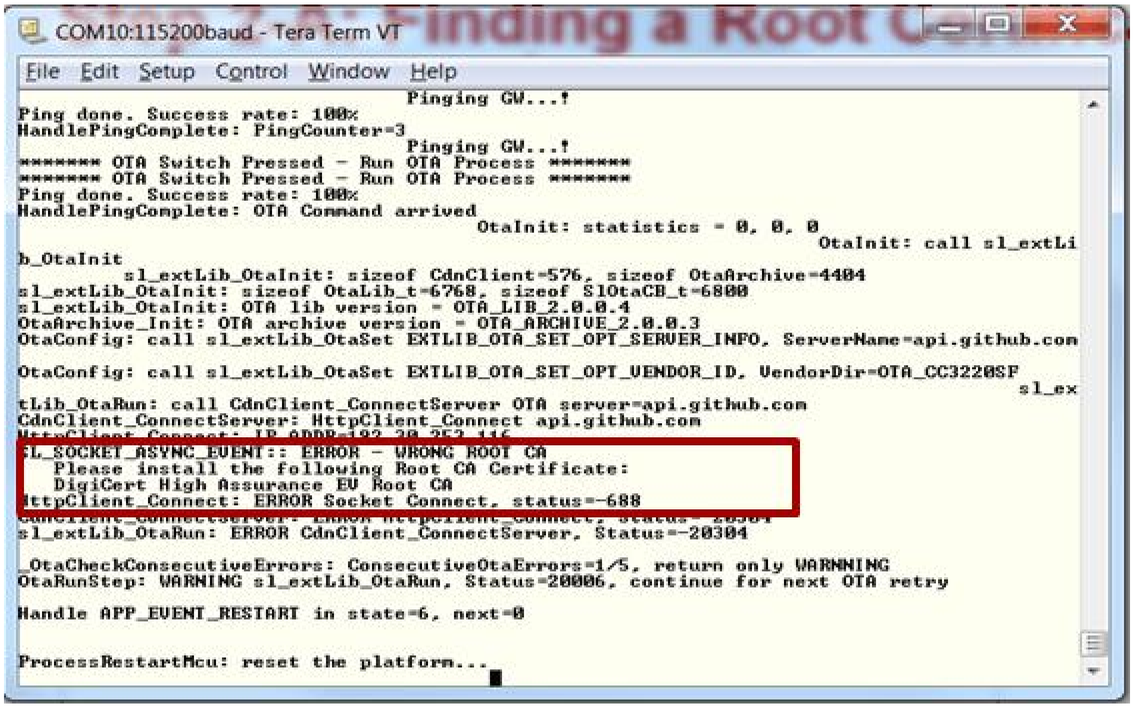
2.5.1 Finding a Server’s Root CA

The first step a user must take to connect to a cloud server over TLS/SSL is to find and download the root certificate required for the server authentication. There are several methods that can be used to retrieve the root certificate of a specific server, such as checking for instruction in the server’s site, using browser connection information, or using dedicated sites (such as https://www.sslchecker.com/). However, the previous methods are not assured to retrieve the correct certificates because different domains or ports can require different certificates. The following method uses the SimpleLink™ connection failure notification to retrieve the name of the certificate as requested by the server (that is, the name as it appeared in the TLS/SSL negotiation). Adding the SimpleLink™ socket event handler code prints the required root CA certificate name to the terminal.

```
void SimpleLinkSockEventHandler(SlSockEvent_t *pSock)
{
   ...
   if ( pSock->Event == SL_SOCKET_ASYNC_EVENT)
   {
       switch (pSock->SocketAsyncEvent.SockAsyncData.Type)
       {
           case SL_SSL_NOTIFICATION_WRONG_ROOT_CA:
              /* on socket error Restart OTA */
              UART_PRINT("SL_SOCKET_ASYNC_EVENT: ERROR - WRONG ROOT CA\n\r");
              UART_PRINT("Please install the following Root Certificate:\n\r");
              UART_PRINT("  %s\n\r", pSock->SocketAsyncEvent.SockAsyncData.pExtraInfo);
           break;
           ...
           default:
              /* on socket error Restart OTA */
              UART_PRINT("SL_SOCKET_ASYNC_EVENT socket event %d \n\r", pSock->Event);
        }
    }
   ...
}
```

Figure 8 shows an example of a failure printout (taken from the SDK's OTA example). The printed common name ("DigiCert High Assurance EV Root CA" in this example) is the public name of the certificate that must be located.

Figure 8. Requested Server's Root CA Certificate Printout



```
                                          Pinging GW...!
Ping done. Success rate: 100%
HandlePingComplete: PingCounter=3
                                          Pinging GW...!
******** OTA Switch Pressed - Run OTA Process ********
******** OTA Switch Pressed - Run OTA Process ********
Ping done. Success rate: 100%
HandlePingComplete: OTA Command arrived
                                                OtaInit: statistics = 0, 0, 0
                                                                       OtaInit: call sl_extLi
b_OtaInit
         sl_extLib_OtaInit: sizeof CdnClient=576, sizeof OtaArchive=4404
sl_extLib_OtaInit: sizeof OtaLib_t=6768, sizeof SlOtaCB_t=6800
sl_extLib_OtaInit: OTA lib version = OTA_LIB_2.0.0.4
OtaArchive_Init: OTA archive version = OTA_ARCHIVE_2.0.0.3
OtaConfig: call sl_extLib_OtaSet EXTLIB_OTA_SET_OPT_SERVER_INFO, ServerName=api.github.com

OtaConfig: call sl_extLib_OtaSet EXTLIB_OTA_SET_OPT_VENDOR_ID, VendorDir=OTA_CC3220SF
                                                                                     sl_ex
tLib_OtaRun: call CdnClient_ConnectServer OTA server=api.github.com
CdnClient_ConnectServer: HttpClient_Connect api.github.com
HttpClient_Connect: IP ADDR=192.30.253.116
SL_SOCKET_ASYNC_EVENT:: ERROR - WRONG ROOT CA
   Please install the following Root CA Certificate:
   DigiCert High Assurance EV Root CA
HttpClient_Connect: ERROR Socket Connect, status=-688
CdnClient_ConnectServer: ERROR HttpClient_Connect, Status=-20304
sl_extLib_OtaRun: ERROR CdnClient_ConnectServer, Status=-20304

_OtaCheckConsecutiveErrors: ConsecutiveOtaErrors=1/5, return only WARNNING
OtaRunStep: WARNING sl_extLib_OtaRun, Status=20006, continue for next OTA retry

Handle APP_EVENT_RESTART in state=6, next=0


ProcessRestartMcu: reset the platform...
```

The root CA certificates are free and available online, or can be retrieved from a local OS/browser certificate catalog. For example, in the Chrome browser, go to Settings→ Advanced→ Manage Certificates→ Trusted Root Certification Authorities. Search the list for the certificate name printed to the terminal earlier and export it in DER format. Alternatively, you may access the Windows' catalog, as shown certificate catalog.

2.5.2 Installing the Server's Root CA

The downloaded root CA certificate must be installed to the file system and referenced by the client application. A TLS/SSL client code example (configuring server's root-ca certificate and optionally the client's certificate and private key):

```
#define SSL_SERVER_ROOT_CA_CERT            "server-root-ca-cert"


sl_SetSockOpt(clt_sock, SL_SOL_SOCKET,
        SL_SO_SECURE_FILES_CA_FILE_NAME,
        SSL_SERVER_ROOT_CA_CERT,
        strlen(SSL_SERVER_ROOT_CA_CERT));
```

2.5.3 Creating the Client Certificate

If a server requires client-authentication (such as in an AWS case), signed certificates must be obtained for the client devices. In most cases, the server provides specific instructions for obtaining client certificates (see for example: https://docs.aws.amazon.com/apigateway/latest/developerguide/getting-started-client-side-ssl-authentication.html#generate-client-certificate). Usually this happens during the first registration of the client in the server's domain. First registration is performed offline (from any computer, and not necessarily using the SimpleLink™ device) by accessing the server's site. It might involve using the server's framework for generating client certificates. At the end of registration, the client should have its own token or certificate to be identified by the server during the future connections. The certificate chain must be stored in a PEM formatted file (see Section 1.3.2).

2.5.4 Installing the Client Certificate

Client-authentication requires the installation of both the certificate and private key. The example code below shows how to associate the certificate and private key with the client socket after they have been added to the file system:

```
#define SSL_CLIENT_KEY            "client-cert.key"
#define SSL_CLIENT_CERT             "client-cert.pem"

/* if Client authentication is required,             *
 * Set client's identity certificate and private key */
sl_SetSockOpt(clt_sock, SL_SOL_SOCKET,
        SL_SO_SECURE_FILES_CERTIFICATE_FILE_NAME,
        SSL_CLIENT_CERT,
        strlen(SSL_CLIENT_CERT));

sl_SetSockOpt(clt_sock, SL_SOL_SOCKET,
        SL_SO_SECURE_FILES_PRIVATE_KEY_FILE_NAME,
        SSL_CLIENT_KEY,
        strlen(SSL_CLIENT_KEY));
```

2.5.5 Connecting to a Server With a Self-Signed Certificate

If the TLS/SSL server uses a proprietary (self-signed) certificate, the signature verification will fail with a SL_ERROR_BSD_ESECUNKNOWNROOTCA warning. This warning does not break the connection and can be ignored. Alternatively, it is possible to avoid the certificate-catalog check for this socket by using a sl_SetSockOpt command, as shown below.

```
_u32 dummyVal;
sl_SetSockOpt(clt_sock, SL_SOL_SOCKET,
        SL_SO_SECURE_DISABLE_CERTIFICATE_STORE,
        &dummyVal, sizeof(dummyVal));
```

2.5.6 Complete TLS/SSL Client Code

The following is a complete example of how to set the certificates required by a TLS/SSL client. Refer to “socket_cmd.c” in the SDK’s “network_terminal” example to find the full secure client implementation.

```
#define SSL_SERVER_ROOT_CA_CERT            "server-root-ca-cert"
#ifdef CLIENT_AUTHENTICATION
#define SSL_CLIENT_KEY            "client-cert.key"
#define SSL_CLIENT_CERT             "client-cert.pem"
#endif // CLIENT_AUTHENTICATION

SlSockAddr_t        *sa;
#ifdef SELF_SIGNED_SERVER_CERTIFICATE
_u32 dummyVal;
#endif // SELF_SIGNED_SERVER_CERTIFICATE

...  (setting socket family)

clt_sock = sl_Socket(sa->sa_family, SL_SOCK_STREAM, TCP_PROTOCOL_FLAGS);

sl_SetSockOpt(clt_sock, SL_SOL_SOCKET,
        SL_SO_SECURE_FILES_CA_FILE_NAME,
        SSL_SERVER_ROOT_CA_CERT,
        strlen(SSL_SERVER_ROOT_CA_CERT));

#ifdef CLIENT_AUTHENTICATION
sl_SetSockOpt(clt_sock, SL_SOL_SOCKET,
      SL_SO_SECURE_FILES_CERTIFICATE_FILE_NAME,
         SSL_CLIENT_CERT,
         strlen(SSL_CLIENT_CERT));

sl_SetSockOpt(clt_sock, SL_SOL_SOCKET,
        SL_SO_SECURE_FILES_PRIVATE_KEY_FILE_NAME,
        SSL_CLIENT_KEY,
        strlen(SSL_CLIENT_KEY));
#endif // CLIENT_AUTHENTICATION

#ifdef SELF_SIGNED_SERVER_CERTIFICATE
sl_SetSockOpt(clt_sock, SL_SOL_SOCKET,
        SL_SO_SECURE_DISABLE_CERTIFICATE_STORE,
        &dummyVal, sizeof(dummyVal));
#endif // SELF_SIGNED_SERVER_CERTIFICATE

... (Setting other socket options and socket address)

status = sl_Connect(clt_sock, sa, addrSize);
```

2.6 Enabling a Secure (TLS/SSL) Server

A TLS/SSL server requires its own certificate. The certificate must be purchased and installed on the device. The server may require authentication of any clients that connect to it. In such case, the server must provide instructions on the required certificate. It may sign the client certificate by itself (using a valid “intermediate” certificate – which has known root CA) or let them purchase a certificate with a specific root CA. The following is a TLS/SSL server code example (configuring server’s certificate and private key and optionally the client root CA certificate):

```
#define SSL_SERVER_KEY             “server-cert.key”
#define SSL_SERVER_CERT             “server-cert.pem”
#ifdef CLIENT_AUTHENTICATION
#define SSL_CLIENT_ROOT_CA_CERT            “client-root-ca-cert”
#endif // CLIENT_AUTHENTICATION

/* Set server’s identity certificate */
sl_SetSockOpt(srv_sock, SL_SOL_SOCKET,
      SL_SO_SECURE_FILES_CERTIFICATE_FILE_NAME,
         SSL_SERVER_CERT,
         strlen(SSL_SERVER_CERT));

/* Set server’s private key */
sl_SetSockOpt(srv_sock, SL_SOL_SOCKET,
        SL_SO_SECURE_FILES_PRIVATE_KEY_FILE_NAME,
        SSL_SERVER_KEY,
        strlen(SSL_SERVER_KEY));1234ee

#ifdef CLIENT_AUTHENTICATION
/* if Client authentication is required,              *
 * Set client’s root-ca certificate                   */
sl_SetSockOpt(clt_sock, SL_SOL_SOCKET,
        SL_SO_SECURE_FILES_CA_FILE_NAME,
        SSL_SERVER_ROOT_CA_CERT,
        strlen(SSL_CLIENT_ROOT_CA_CERT));
#endif // CLIENT_AUTHENTICATION
```

2.7 Securing OTA Content

SimpleLink™ Wi-Fi® provides a method that protects an OTA process by authenticating the source of the updated image. This feature is currently supported through the SimpleLink™ OTA library. It enables the use of a code signing certificate to sign the entire content of an OTA image. Signing the entire OTA image enables the device that loads the image to ensure it is authentic before installing it. Section 4 details the concept and usage.

3 Registration to a Cloud Server

This section is focused on the provisioning of a TLS client certificate that will authenticate the device upon connecting to a cloud server.

The procedures presented in this section make use of the device unique key pair (with the Certificate Signing Request (CSR) feature). This eliminates the need of customers to generate the key pair and then compromise the privacy of the key when it is installed on the device during production-line.

The goal is to enable the SimpleLink™ Wi-Fi® device to securely register to the vendor's cloud server.

The challenge is to correlate between devices in the OEM production line and the accounts within the OEM cloud service without compromising the device identity (i.e. private keys).

The certificate provisioning procedure contains 3 steps:

1. Issuing of a client certificate.
2. Registration of the certificate at the cloud server.
3. Installation of the certificate on the SimpleLink™ Wi-Fi® device.

After a successful registration, the SimpleLink™ device will be able to connect by way of a secure connection (TLS/SSL) to the cloud server.

Section 3.1 describes the method of obtaining a certificate using the SimpleLink™ unique key pair and the CSR feature. It provides a code example for creating and reading the CSR that integrates user parameters with the device unique identity.

NOTE: One of the special security features of the SimpleLink™ Wi-Fi® is the Unique Key-Pair per device. This feature enables crypto utilities such as sign and verify without direct access to the private key of the device from the Host application.

The public key can be retrieved by the application (see following code example).

```
typedef struct {
    uint8_t bytes[256];
} PubKey_t;

static int getPublicKey(PubKey_t *pPubKey)
{
    uint8_t configOpt = SL_NETUTIL_CRYPTO_PUBLIC_KEY;
    uint32_t objId = 0;
    uint16_t len = sizeof(PubKey_t);
    return sl_NetUtilGet(configOpt, objId, pPubKey->bytes, &len);
}
```

This unique key pair can also be used for mutual authentication within a TLS handshake. This requires not only the unique key-pair itself but the device must have a certificate signed by an authority or chain of trust that is accepted by the server. To create such certificate, in most cases, access to the public key of the device is not enough. The common way to create and sign a certificate is to use Certificate Signing Request (CSR), which requires a signature of some data with the private key during the creation.

The SimpleLink™ device simplifies this process and provides the ability to create a CSR in PKCS #10 format internally by the device itself. The CSR is created as a file on the file system of the SimpleLink™ device and could be read like any other non-secure file on this file system.

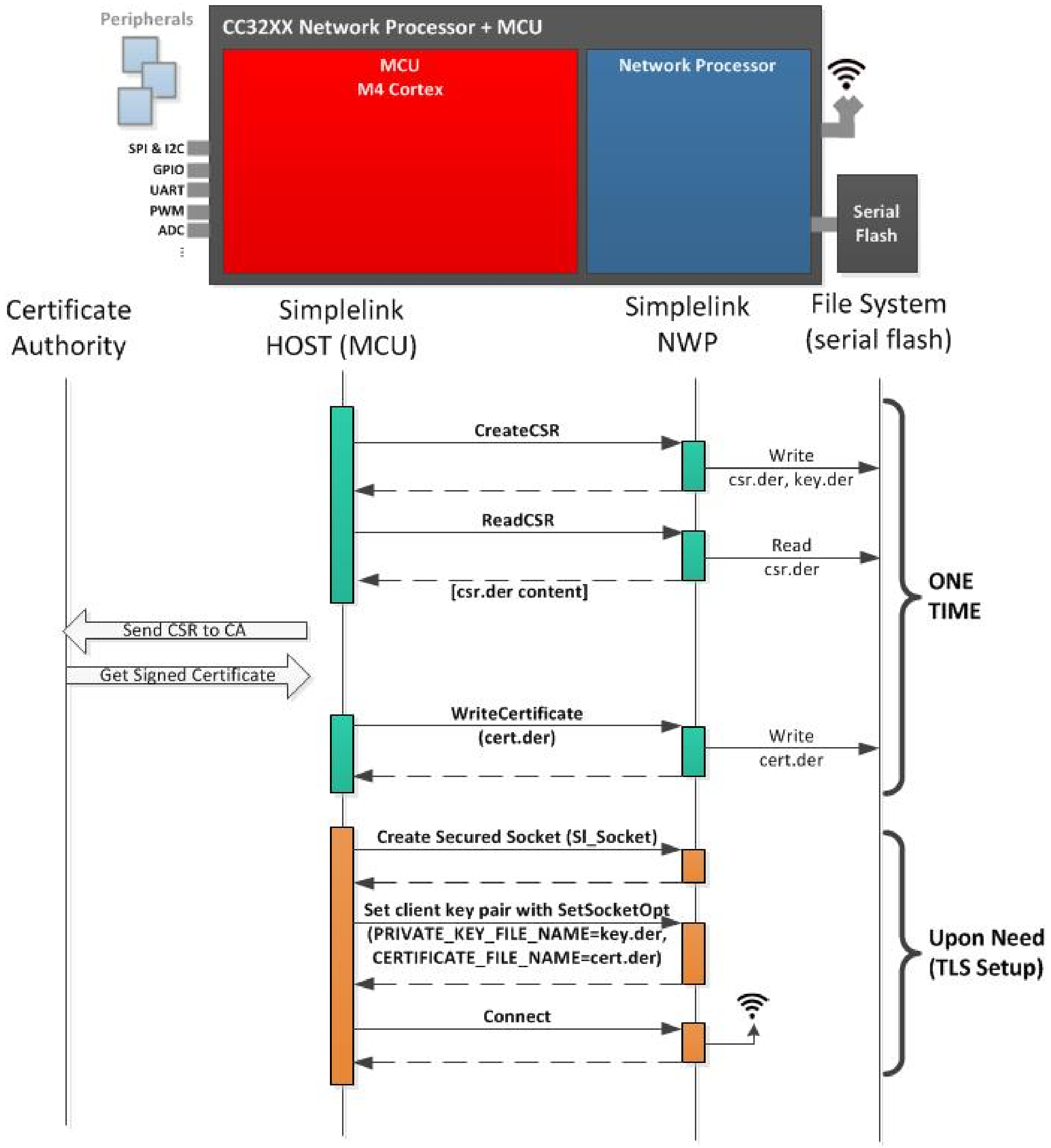
Once the CSR is created and read out of the file system, it will then be used by a certificate authority to create the client certificate for the device. This step requires an external interface to an authorized signer. Section 3.2 focuses on this procedure. It presents two methods (with and without the use of Hardware Security Modules (HSM)) that automate the certificate provisioning. Both methods take advantage of the SimpleLink™ Unique Key Pair. The methods are given as examples for an automatic and secure provisioning procedure.

NOTE: Hardware Security Modules (HSM) are secure devices that safely manage digital keys and is capable of provisioning cryptographic key. It can be used to store CA signer key and generate user certificates based on Certificate Signing Request.

3.1 Obtaining a Certificate Using the SimpleLink™ Key-Pair

Figure 9 shows the SimpleLink™ certificate signing request sequence.

Figure 9. SimpleLink™ Certificate Signing Request Sequence



3.1.1 Step 1 - Creating the CSR

The first step is to use the SimpleLink™ host driver API to generate a CSR based on the user settings and the unique key pair of the SimpleLink™. Once this is completed, the private key and the CSR will be stored on the serial flash. The private key will be stored as secure and encrypted file ("/sys/cert/iot/key.der") and disable external access (from the application MCU). The CSR ("/cert/iot/csr.der") will be kept in der format (plain text) with no access limitations.

CC32XX: The Unique Device Identifier (UDID) is an unmodifiable, unique 128-bit number that TI stores in the device during production. The identifier can be read by the application (see example below).

```
typedef struct {
    uint8_t bytes[16];
} UUID_t;

static int getUUID(UUID_t *pUUID)
{
    uint8_t configOpt;
    uint16_t len = sizeof(UUID_t);
    configOpt = SL_DEVICE_IOT_UDID;
    return sl_DeviceGet(SL_DEVICE_IOT,&configOpt,&len, pUUID->bytes);
}
```

The following code demonstrates how to create the CSR. The user specific parameters should be set through the CSRInfo_t structure. The "common name" field can be derived from the Unique Device ID can be used to correlate between a generated certificate and a specific device.

```
typedef struct
{
    uint32_t DaysValid;         // validity period dates defined from now until now + daysValid
    uint32_t IsCa;              // (0-No/1-Yes)
    uint8_t* SubjectCountry;
    uint8_t* SubjectState;
    uint8_t* SubjectLocality;
    uint8_t* SubjectSur;
    // The common name may be derived from the Unique Device ID
    uint8_t* SubjectCommonName;
    uint8_t* SubjectOrg;
    uint8_t* SubjectOrgUnit;
    uint8_t* SubjectEmail;
} CSRInfo_t;

static int generateCSR(CSRInfo_t *pCsrInfo)
{
    int rc = INVALID_VALUE;
    SlNetUtilCryptoCmdCreateCertAttrib_t certCmd;
    uint32_t certVersion;
    uint16_t outputLen = 0;
    uint32_t certSigType;

    /* Create the device certificate (identity) */

    /* initialize the creation process */
    certCmd.SubCmd = SL_NETUTIL_CRYPTO_CERT_INIT;
    certCmd.ObjId = SL_NETUTIL_CRYPTO_SERVICES_IOT_RESERVED_INDEX;
    certCmd.Flags = 0;
    rc = sl_NetUtilCmd(SL_NETUTIL_CRYPTO_CMD_CREATE_CERT,
                       (uint8_t*)&certCmd,sizeof(certCmd),
                       NULL,0,
                       NULL,&outputLen);
    if(0 == rc)
```

```
{
   /* Sets certificate version ( encoded certificate version: 0=v1, 1=v2, 2=v3)
    * SimpleLink device support only v3 */

   certCmd.SubCmd = SL_NETUTIL_CRYPTO_CERT_VER;
   certVersion = 2;
   rc = sl_NetUtilCmd(SL_NETUTIL_CRYPTO_CMD_CREATE_CERT,
                       (uint8_t*)&certCmd,
                       sizeof(SlNetUtilCryptoCmdCreateCertAttrib_t),
                       (uint8_t*)&certVersion, sizeof(certVersion),
                       NULL, &outputLen);
}
if(0 == rc)
{
   /* Sets certificate signature type
    * The type represent the algorithm identifier for the algorithm used
    * by the device to sign the certificate
    * SimpleLink device support only SL_UTILS_CRYPTO_SIG_SHAwECDSA for
      certificate generation */

   certCmd.SubCmd = SL_NETUTIL_CRYPTO_CERT_SIG_TYPE;
   certSigType = SL_NETUTIL_CRYPTO_SIG_SHAwECDSA;

   rc = sl_NetUtilCmd(SL_NETUTIL_CRYPTO_CMD_CREATE_CERT,
                       (uint8_t*)&certCmd,
                       sizeof(SlNetUtilCryptoCmdCreateCertAttrib_t),
                       (uint8_t*)&certSigType, sizeof(certSigType),
                       NULL,  &outputLen);
}
if(0 == rc)
{
   /* Get and set certificate validity period
    * validity period dates defined from now until now + daysValid */
    if(pCsrInfo->DaysValid > 0)
    {
        certCmd.SubCmd = SL_NETUTIL_CRYPTO_CERT_DAYS_VALID;
        rc = sl_NetUtilCmd(SL_NETUTIL_CRYPTO_CMD_CREATE_CERT,
                               (uint8_t *)&certCmd,
                               sizeof(SlNetUtilCryptoCmdCreateCertAttrib_t),
                               (uint8_t*)&pCsrInfo->DaysValid,
                               sizeof(pCsrInfo->DaysValid),
                               NULL,  &outputLen);
    }
    else
    {
        rc = INVALID_VALUE;
    }
}
/* define if certificate is a ca certificate */
if(rc == 0)
{
    certCmd.SubCmd = SL_NETUTIL_CRYPTO_CERT_IS_CA;

    if(pCsrInfo->IsCa <= 1)
    {
        rc = sl_NetUtilCmd(SL_NETUTIL_CRYPTO_CMD_CREATE_CERT,
                         (uint8_t*)&certCmd,
                         sizeof(SlNetUtilCryptoCmdCreateCertAttrib_t),
                         (uint8_t*)&pCsrInfo->IsCa, sizeof(pCsrInfo->IsCa),
                         NULL, &outputLen);
    }
    else
    {
        rc = INVALID_VALUE;
    }
}
```

```
    /* Set subject country */

    if(rc == 0)
    {
        if((strlen((char *)pCsrInfo->SubjectCountry) == 2) &&
           (pCsrInfo->SubjectCountry[0] >= 'A') &&
           (pCsrInfo->SubjectCountry[0] <= 'Z') &&
           (pCsrInfo->SubjectCountry[1] >= 'A') &&
           (pCsrInfo->SubjectCountry[1] <= 'Z'))
        {
            certCmd.SubCmd = SL_NETUTIL_CRYPTO_CERT_SUBJECT_COUNTRY;
            rc = sl_NetUtilCmd(SL_NETUTIL_CRYPTO_CMD_CREATE_CERT,
                               (uint8_t*)&certCmd,
                                sizeof(SlNetUtilCryptoCmdCreateCertAttrib_t),
                                pCsrInfo->SubjectCountry,
                               (strlen((char *)pCsrInfo->SubjectCountry) + 1),
                               NULL, &outputLen);
        }
        else
        {
            rc = INVALID_VALUE;
        }
    }

    /* Set certificate subject state */
    if(rc == 0)
    {
        if(IsValisString((char *)pCsrInfo->SubjectState, MAX_STRING_LENGTH))
        {
            certCmd.SubCmd = SL_NETUTIL_CRYPTO_CERT_SUBJECT_STATE;

            rc = sl_NetUtilCmd(SL_NETUTIL_CRYPTO_CMD_CREATE_CERT,
                                   (uint8_t*)&certCmd,
                                   sizeof(SlNetUtilCryptoCmdCreateCertAttrib_t),
                                   pCsrInfo->SubjectState,
                                   (strlen((char *)pCsrInfo->SubjectState) + 1),
                                   NULL, &outputLen);
        }
        else
        {
            rc = INVALID_VALUE;
        }
    }

    /* Set the subject locality */
    if(rc == 0)
    {
        if(IsValisString((char *)pCsrInfo->SubjectLocality, MAX_STRING_LENGTH))
        {
            certCmd.SubCmd = SL_NETUTIL_CRYPTO_CERT_SUBJECT_LOCALITY;

            rc = sl_NetUtilCmd(SL_NETUTIL_CRYPTO_CMD_CREATE_CERT,
                               (uint8_t*)&certCmd,
                               sizeof(SlNetUtilCryptoCmdCreateCertAttrib_t),
                               pCsrInfo->SubjectLocality,
                               (strlen((char *)pCsrInfo->SubjectLocality) + 1),
                               NULL, &outputLen);
        }
        else
        {
            rc = INVALID_VALUE;
        }
    }

    /* Set the subject surname */
```

```
        if(rc == 0)
        {
            if(IsValisString((char *)pCsrInfo->SubjectSur, MAX_STRING_LENGTH))
            {
                certCmd.SubCmd = SL_NETUTIL_CRYPTO_CERT_SUBJECT_SUR;
                rc = sl_NetUtilCmd(SL_NETUTIL_CRYPTO_CMD_CREATE_CERT,
                                       (uint8_t*)&certCmd,
                                       sizeof(SlNetUtilCryptoCmdCreateCertAttrib_t),
                                       pCsrInfo->SubjectSur,
                                       (strlen((char *)pCsrInfo->SubjectSur) + 1),
                                       NULL, &outputLen);
            }
            else
            {
                rc = INVALID_VALUE;
            }
        }

        /* Set the subject organization */
        if(rc == 0)
        {
            if(IsValisString((char *)pCsrInfo->SubjectOrg, MAX_STRING_LENGTH))
            {
                certCmd.SubCmd = SL_NETUTIL_CRYPTO_CERT_SUBJECT_ORG;
                rc = sl_NetUtilCmd(SL_NETUTIL_CRYPTO_CMD_CREATE_CERT,
                                       (uint8_t*)&certCmd,
                                       sizeof(SlNetUtilCryptoCmdCreateCertAttrib_t),
                                       pCsrInfo->SubjectOrg,
                                       (strlen((char *)pCsrInfo->SubjectOrg) + 1),
                                       NULL, &outputLen);
            }
            else
            {
                rc = INVALID_VALUE;
            }
        }

        /* Set the subject organization unit */
        if(rc == 0)
        {
            if(IsValisString((char *)pCsrInfo->SubjectOrgUnit, MAX_STRING_LENGTH))
            {
                certCmd.SubCmd = SL_NETUTIL_CRYPTO_CERT_SUBJECT_ORG_UNIT;
                rc = sl_NetUtilCmd(SL_NETUTIL_CRYPTO_CMD_CREATE_CERT,
                                    (uint8_t*)&certCmd,
                                    sizeof(SlNetUtilCryptoCmdCreateCertAttrib_t),
                                    pCsrInfo->SubjectOrgUnit,
                                    (strlen((char *)pCsrInfo->SubjectOrgUnit) + 1),
                                    NULL, &outputLen);
            }
            else
            {
                rc = INVALID_VALUE;
            }
        }

        /* Set the subject common name  */
        if(rc == 0)
        {
            if(strlen((char *)pCsrInfo->SubjectCommonName) <= MAX_STRING_LENGTH)
            {
                certCmd.SubCmd = SL_NETUTIL_CRYPTO_CERT_SUBJECT_COMMON_NAME;
                rc = sl_NetUtilCmd(SL_NETUTIL_CRYPTO_CMD_CREATE_CERT,
                                 (uint8_t*)&certCmd,
                                 sizeof(SlNetUtilCryptoCmdCreateCertAttrib_t),
                                 pCsrInfo->SubjectCommonName,
                                 (strlen((char *)pCsrInfo-
    >SubjectCommonName) + 1),                              NULL, &outputLen);
            }
            else
            {
                rc = INVALID_VALUE;
            }
        }

        /* Set the subject email */
        if(rc == 0)
        {
            if(strlen((char *)pCsrInfo->SubjectEmail) <= MAX_STRING_LENGTH)
            {
                certCmd.SubCmd = SL_NETUTIL_CRYPTO_CERT_SUBJECT_EMAIL;
                rc = sl_NetUtilCmd(SL_NETUTIL_CRYPTO_CMD_CREATE_CERT,
                                       (uint8_t*)&certCmd,
                                       sizeof(SlNetUtilCryptoCmdCreateCertAttrib_t),
                                       pCsrInfo->SubjectEmail,
                                       (strlen((char *)pCsrInfo->SubjectEmail) + 1),
                                       NULL, &outputLen);
            }
            else
            {
                rc = INVALID_VALUE;
            }
        }

        /* Close the process and create the certificate */
        if(0 == rc)
        {
            certCmd.SubCmd = SL_NETUTIL_CRYPTO_CSR_SIGN_AND_SAVE;
            rc = sl_NetUtilCmd(SL_NETUTIL_CRYPTO_CMD_CREATE_CERT,
                                   (uint8_t*)&certCmd,
                                   sizeof(SlNetUtilCryptoCmdCreateCertAttrib_t),
                                   NULL, 0,
                                   NULL, &outputLen);
        }
        /* (0 == rc) -> The CSR was created successfully
         * (INVALID_VALUE == rc) -> ERROR: Invalid Params
         * (0 > rc) -> ERROR: NetUtilCmd error
         */
        return(rc);
    }
```

3.1.2 Step 2 - Read CSR File

After Step 1 is complete, the CSR exists as a file on the file system. In this step, the vendor should read the file from the file system and send it to the CA for signing. At the end of the process, the file can be deleted from the file system.

The following code demonstrates how to read the CSR file from the NWP:

```
#define CSR_BUFF_SIZE     2000

int32_t readCSR()
{
    int32_t     fHdl;
    uint8_t     fileName[] = "/cert/iot/csr.der";
    uint8_t     data[CSR_BUFF_SIZE];
    int32_t     status;
    uint32_t    offset;
    SlFsFileInfo_t csrFileInf;

    status = sl_FsGetInfo(fileName,NULL,&csrFileInf);
    if(status < 0)
    {
        return -1;
    }

    fHdl = sl_FsOpen(fileName, SL_FS_READ, NULL);

    if(fHdl > 0)
    {
        status = sl_FsRead(fHdl, 0, data, csrFileInf.Len);

        sl_FsClose(fHdl,0,0,0);

        if(status < 0)
        {
            return -1;
        }

        /* here the data buffer contains the CSR file */
        /* ... */

        return 0;
    }
    else
    {
        return -1;
    }
}
```

3.1.3 Step 3 - Create and Sign a Certificate

Step 3 depends on the methods, tools and processes selected by the customer. Section 3.2 describes two procedures for automatic provisioning of a certificate in production line (using HSM as the signer) and in run time (using the Signing Service at the cloud).

3.1.4 Step 4 - Write Certificate to File-System

Once the signed certificate is obtained, it needs to be written to the file system of the device to allow the device use it for client authentication in the TLS handshake. The format of the file could be PEM or DER and it should include the entire chain-of-trust used to sign the certificate.

The following code demonstrates how to write the file back to the device:

```
#define CERT_WRITE_CHUNK_SIZE     1500

int32_t writeCert(uint8_t *data , uint32_t len)
{
    int32_t     fHdl;
    uint8_t     fileName[] = "/cert/iot/cert.der";
    int32_t     status;
    uint32_t    offset;
    uint32_t    writeLen;

    fHdl = sl_FsOpen(fileName, SL_FS_CREATE | SL_FS_OVERWRITE |
                     SL_FS_CREATE_FAILSAFE | SL_FS_CREATE_MAX_SIZE( len ), NULL);
    if( fHdl >= 0 )
    {
        offset = 0;
        do
        {
            if (len < CERT_WRITE_CHUNK_SIZE)
            {
                writeLen = len;
            }
            else
            {
                writeLen = CERT_WRITE_CHUNK_SIZE;
            }

            status = sl_FsWrite(fHdl, offset, &(data[offset]), writeLen);

            offset += writeLen;
        }while (offset < len);

        sl_FsClose(fHdl,0,0,0);

        return 0;
    }
    else
    {
        return -1;
    }
}
```

3.1.5 Step 5 - Use the Unique Key-Pair in TLS Connection

Once the device holds the signed certificate file in its file system, the TLS socket can use these files to establish a TLS connection that include client authentication based on this unique key-pair of the device.

The following code demonstrates how to create such TLS connection:

```
void tlsClientExample()
{
    SlSockAddrIn_t          addr;
    SlSockSecureMethod_t    method;
    int32_t                 sd,len,dummyVar;
    int16_t                 status;
    int16_t                 addrSize;
    int8_t                  buf[100];

    uint16_t                destinationPort = 443;
    uint32_t                destinationIP = SL_IPV4_VAL(10,5,9,7);
    const int8_t*           rootCaFileName = "/cert/rootSrvCA.der";
    const int8_t*           selfPrivateKeyFileName = "/sys/cert/iot/key.der";
    const int8_t*           selfCertFileName = "/cert/iot/cert.der";
    uint8_t  fnLen;


    addr.sin_family = SL_AF_INET;
    addr.sin_port = sl_Htons(destinationPort);
    addr.sin_addr.s_addr = sl_Htonl(destinationIP);
    addrSize = sizeof(SlSockAddrIn_t);

    /* Open TCP socket */
    sd = sl_Socket(SL_AF_INET,SL_SOCK_STREAM,SL_SEC_SOCKET);
    if(sd < 0)
    {
        /* error... */
    }

    /* set a CA filename to be used to verify the server
       certificate when the handshake will take place */
    status = sl_SetSockOpt(sd,SL_SOL_SOCKET,
                          SL_SO_SECURE_FILES_CA_FILE_NAME,
                          rootCaFileName,strlen(rootCaFileName));
    if(status < 0)
    {
        /* error... */
    }

    /* set a private key filename to be used in case the server requires client authentication */
    status = sl_SetSockOpt(sd,SL_SOL_SOCKET,
                          SL_SO_SECURE_FILES_PRIVATE_KEY_FILE_NAME,
                          selfPrivateKeyFileName,strlen(selfPrivateKeyFileName));
    if(status < 0)
    {
        /* error... */
    }

    /* set the certificate filename to be used in case the server requires client authentication
*/
    status = sl_SetSockOpt(sd,SL_SOL_SOCKET,
                          SL_SO_SECURE_FILES_CERTIFICATE_FILE_NAME,
                          selfCertFileName,strlen(selfCertFileName));
    if(status < 0)
    {
        /* error... */
    }

    status = sl_Connect(sd, ( SlSockAddr_t *)&addr, addrSize);
    if(status < 0)
```

```
        {
            /* error... */
        }

        /* connection is secured - run traffic */
        /* ... */


        status = sl_Close(sd);
        if(status < 0)
        {
            /* error... */
        }
    }
```

3.2 Cloud Provisioning Examples

This section presents two procedures for automatic provisioning of a certificate:

1. Using a Hardware Security Module (HSM) in the production line as the signer.
2. Provisioning at run-time by connecting to a Signing Service in the cloud.

3.2.1 Cloud Provisioing Using HSM

In the basic approach, the OEM vendor can use HSM to assign certificates to the devices during production line.

This will require a transfer of a CA and signer key from the OEM vendor to a Contract Manufacturer. The contract manufacturer is responsible of programming the HSM before the production process starts.

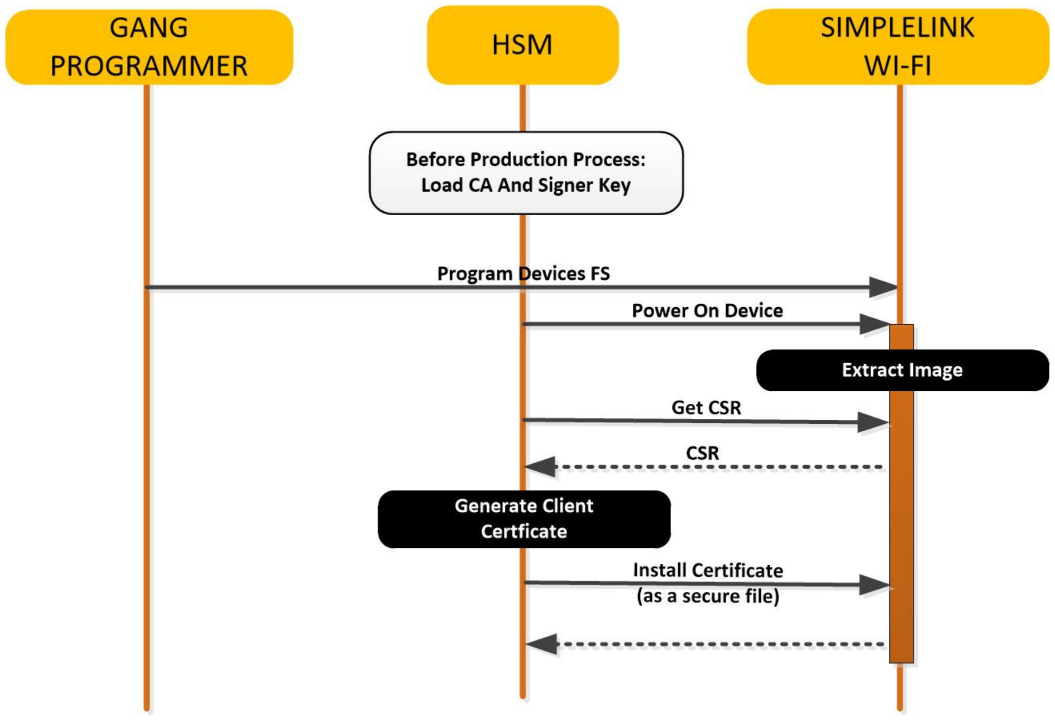
The signing CA (root or intermediate) key can be user defined or specifically selected by the cloud vendor. The important thing is to make sure the signer key can be verified by the cloud server. One way is that the customer will use a certificate signed by a CA (root or intermediate) that is supported by the cloud server. The alternative way is the customer will use any signer key and install the corresponding root (or intermediate) certificate at the cloud server (if the cloud server enables such operation).

The entire certificate chain of trusts would be stored as secured files on the SimpleLink™ Wi-Fi® file system.

The certificates then will be registered / activated in the cloud server upon the first connection. No artifact needs to be sent to cloud server following the production line and no specific operation is required from the device in operational mode. The certificate would be authenticated by the server based on its root of trust (for example see AWS JITR/JITP – Just-In-Time Registration/Provisioning).

This process is well-known in the industry.

Figure 10. Production Line Using HSM


GANG PROGRAMMER
HSM
SIMPLELINK WI-FI
Before Production Process: Load CA And Signer Key
Program Devices FS
Power On Device
Extract Image
Get CSR
CSR
Generate Client Certficate
Install Certificate (as a secure file)

3.2.2 Run-Time Cloud Provisioning

To reduce the HSM costs, the SimpleLink™ Wi-Fi® allows the use of an alternative method that avoids the use of private keys during the production. Thus, the HSM is no longer required.

In this method, a tester software will retrieve only the components of the device identity that are public information (that is, the UDID and the device public key). The tester will need to store the public information on a physical storage.

Once the production will complete, the stored information will be delivered to the cloud server which can use it to manage a vendor-specific database. The UDID will be the key to the database that stores public key. Note that the UDID is unique and long enough to serve as a JSON Web Token Claim.

The cloud server will use the database to verify the device upon its first access to the server.

Figure 11 depicts the sequence previously described. In this case, the Gang Programmer programs the flash before the device was activated. The tester should wait for the application to complete its initialization (following the "Power On" trigger). The image running in the production line should support the interface for retrieving the device UDID and Public Key over the external interface (for example, UART).

Figure 11. Production Line Without HSM

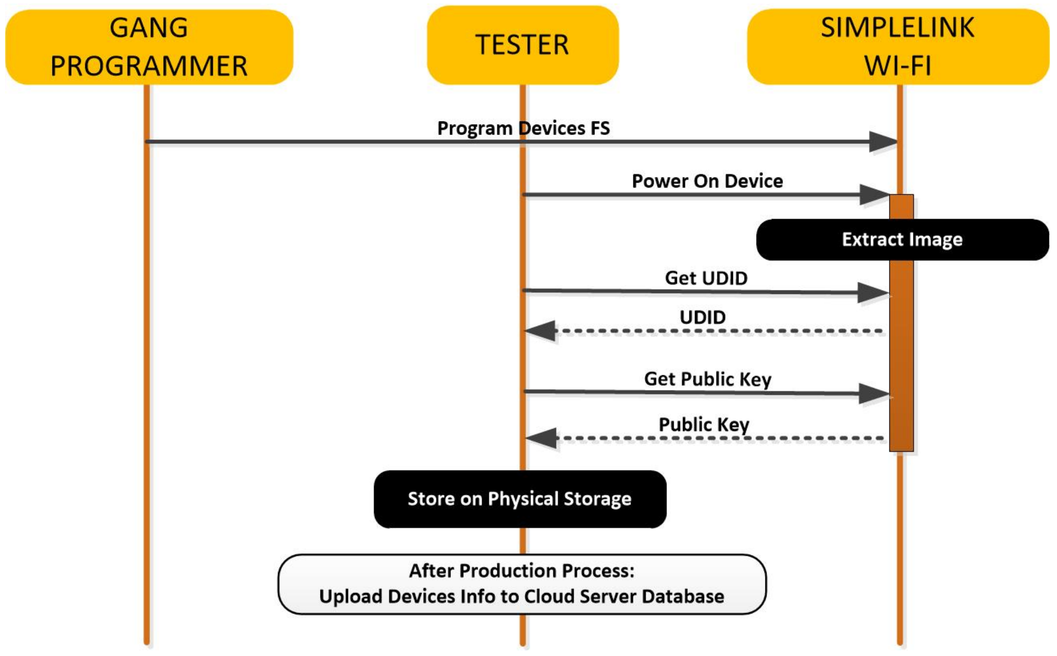


Figure 12 shows an example of an automatic client registration to a cloud server which can happen when the device is already operational (that is, not during manufacturing).

The device will generate and sign (using its private key) a CSR and provide it with the unique device ID (which can instead appear as the common name inside the CSR) to the Cloud Registration server.

The server can use the UDID as index to the vendor's database and retrieve the device's public key. The server will then use the public key to verify the CSR signature.

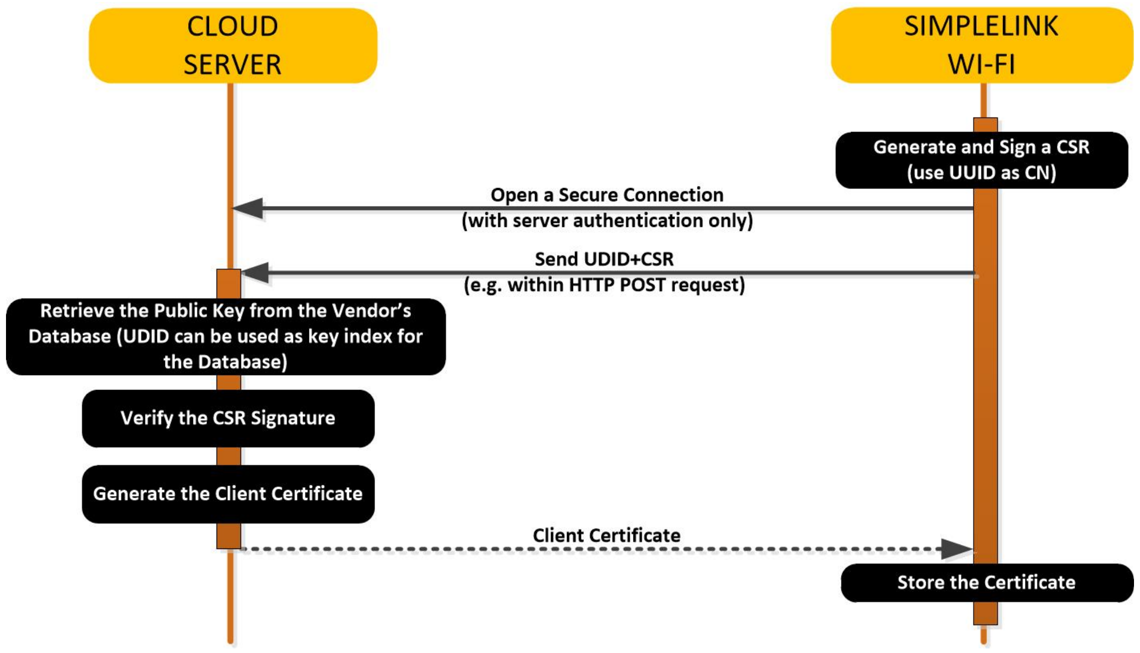
Once the signature is verified, the server will create a certificate for the client and send it back to the client.

The certificate root of trust can be based on Server's CA or on a CA installed by the vendor before (for example check the AWS Just-In-Time Propvisioning/Registeration – JITP/JITR).

The device will store the certificate and will use it within the next connections to the server (that is, in SSL/TLS with client authentication).

Since the CSR is only used in the secure context (SSL/TLS), it will make it harder for an aggressor to duplicate the CSR and perform a denial-of-service attack on the server. Note that even if the CSR is compromised, the certificate will not be usable with the private key that is stored safely on the device.

Figure 12. Cloud Server Registration


CLOUD SERVER
SIMPLELINK WI-FI
Generate and Sign a CSR (use UUID as CN)
Open a Secure Connection (with server authentication only)
Send UDID+CSR (e.g. within HTTP POST request)
Retrieve the Public Key from the Vendor's Database (UDID can be used as key index for the Database)
Verify the CSR Signature
Generate the Client Certificate
Client Certificate
Store the Certificate

4 SimpleLink™ Secure Over-The-Air (OTA) Delivery

The SimpleLink™ OTA library demonstrates a method for securing the OTA content. The method requires the use of a code signing certificate based on elliptic curve cryptography (ECDSA SECP256R1).

NOTE: The following is an example of openssl command for generating EC private key:

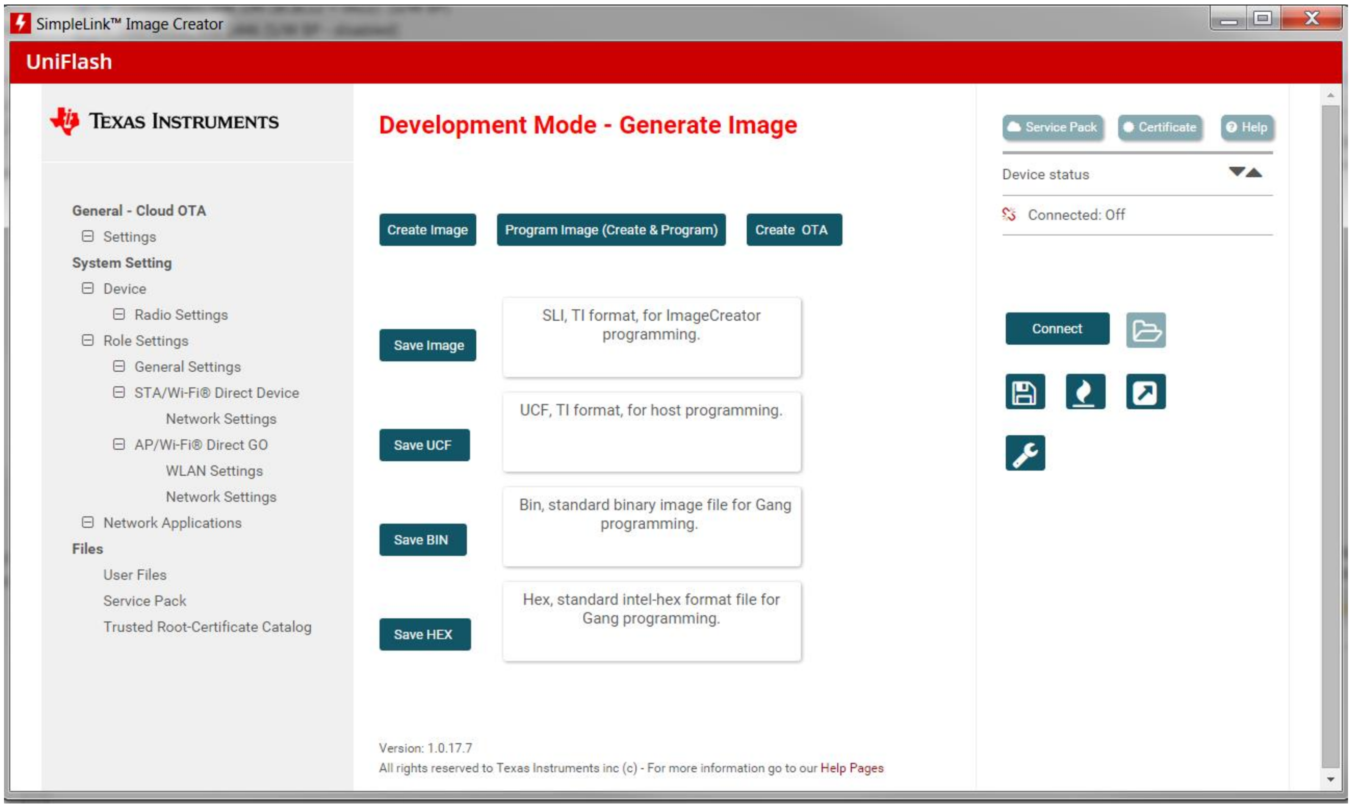
```
> openssl ecparam -name prime256v1 -genkey -out ota_vendor_key.pem
```

To create a public certificate from this key, use the following:

```
> openssl req -new -x509 -key ota_vendor_key.pem -out ota_vendor_cert.pem -
days 730
```

The Uniflash tool supports the creation of an OTA image (see Figure 13).

Figure 13. Uniflash Create OTA



SLI, TI format, for ImageCreator programming.

UCF, TI format, for host programming.

Bin, standard binary image file for Gang programming.

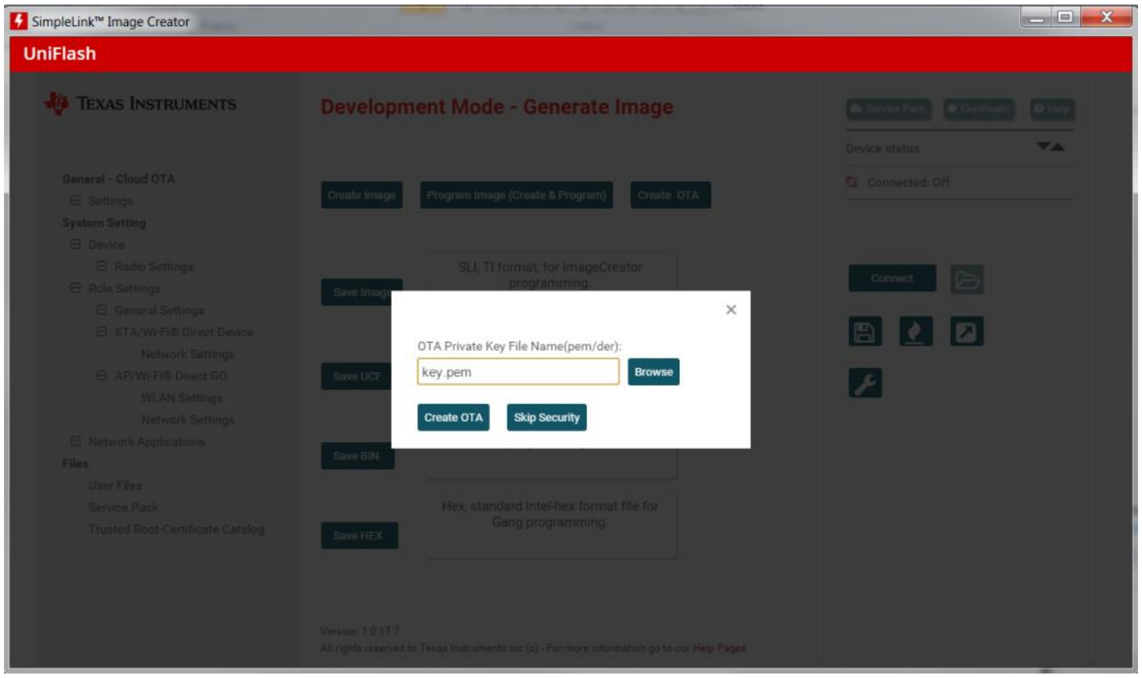
Hex, standard intel-hex format file for Gang programming.

Version: 1.0.17.7

All rights reserved to Texas Instruments inc (c) - For more information go to our Help Pages

When creating the OTA image, Uniflash prompts the user for the private key. This key is used to sign the image. The corresponding public certificate must be installed on the target device. It is used by the OTA library (see “ota_archive.c”) to verify the signature. The verification is done using the SimpleLink™ Wi-Fi® crypto engine (more details on the engine can be found in [2].

Figure 14. Uniflash OTA Private Key



OTA Private Key File Name(pem/der): key.pem

4.1 Signing the OTA

The OTA (tarball) image contains a command file (in “1/ota.cmd”) which lists the included files and their metadata. Uniflash uses a hash function to calculate a digest for each of the included files. The metadata of each file contains the digest value, as shown below:

```
{
   "digest":"d89cdff2f5c7784fa2551f265f0ebe16a7179f51fc6cd36e78ecee7285ec3075",
   "certificate":"",

"signature_base64":"WR+s7F86owFCEwm7wsczB21bsjBFVv0h6odJTkoP7+uCSO7o4Av1yiYncrxRyMhU4DCj6hWA5HfKsH
5oW1OArgN/a74JDVc7tYvwKH6iKI/IM28c0QF9GqQqeDFoXvlgTphJnDT242ieILk4J2J9j4Gx+hgg2+CQ34DaLogKHuueI+oP
UKBldCyMRdmb/gRxTGaT501u6qdyrFEqCv8z9y2lP+nQksm55m5KJh6GO6z7Se0bK46F/diJr1kozUh9f7fDC4r1ESzXY6nBtv
STLqzH8wWXbjCVwP+wElktKICbWIOKVRyUA8jJtzGNlBtovqJ5PZ7OuFNVhsJ0vYfB1w==",
   "secured":1,
   "bundle":1,
   "filename":"/sys/servicepack.ucf"
}
```

Finally, when all the digests are updated, the hash function is run on the “ota.cmd” file itself. The “ota.cmd” digest is then encrypted with the private key to create the signature (“1/ota.sign”).

4.2 Verifying the OTA Signature

The “ota_archive.c” file of the OTA library is responsible for verifying the signature. The “OTA_CERTIFICATE_NAME” definition must be updated to point to the installed OTA certificate. The code starts by verifying the digest of each of the files (by comparing them to the value in the “ota.cmd” metadata). It then calculates the “ota.cmd” digest and compares it to the signature (“ota.sign”) after the signature is decrypted with the public key.

5 Troubleshooting

This section provides methods for troubleshooting the most common certificate issues. It is sorted by the error codes retrieved.

Table 2. TLS/SSL Errors

Error Code	Error Name	Description and Fix	Comment
-456	SL_ERROR_BSD_ ESECBADCERTFILE	Either the certificate file was not found (was not found in file system or a wrong file name was used) or it is the wrong one for the connection (that is, the user must install a different certificate).	
-468	SL_ERROR_BSD_ ESECUNKNOWNROOTCA	The connection was created (the certificate set by SetSockOpt was accepted), but the server's certificate signature cannot be found in the installed catalog (that is, the root CA signature is not recognized). If this is a known issue (such as connecting to a server that uses self-signed certificate), the user can ignore the warning or disable it with sl_SetSockOpt (SL_SO_SECURE_DISABLE_CERTIFICATE_STORE - see Section 2.5.5).	A security warning (can be ignored or disabled)

Table 3. File System Errors

Error Code	Error Name	Description and Fix	Comment
-10289	SL_ERROR_FS_WRONG_ SIGNATURE_SECURITY_ALERT	Failure to verify the signature. Typically this is caused due to a wrong certificate that was provided (does not comply with the private key, such as "dummy-root-ca-cert" or "dummy-trusted-cert-key).	
-10341	SL_ERROR_FS_FILE_ NOT_EXISTS	Received when trying to close a signed file – the certificate file was not found (either the file is missing from the file system or the file path or name is wrong in the sl_FsClose command).	
-10292	FS_SECURITY_ALERT_ CERT_CHAIN_ERROR	Typically, received when trying to verify a certificate chain that uses a non-supported signature algorithm (such as SHA-384 or SHA-512).	Refer to Section 1.5.1 for more details on supported signature types.

6 Q & A

- Q: Do I have to purchase a certificate?

  A: When using a secured device, you must purchase a code-signing certificate to sign (at the minimum) your MCU image (see Section 1.5.1). During development you may use the playground catalog and certificates. However, the playground catalog and certificates setup is not secured and must not be used in production (as the playground private key is provided with the SDK). If you use a TLS or SSL connection as a server or client you must get SSL certificates to set up secure sessions (see Section 1.5.2).

- Q: Can I use the playground certificate store in my production release?
  A: No. The playground private key is exposed in TI's SDK, which means you cannot authenticate your code securely. In addition, the playground catalog does not contain the list of valid root CA certificates, which means that any access to a cloud server will not be fully authenticated. For more details, refer to Section 1.4.2.

- Q: How can I connect to a specific cloud server?
  A: See Section 2.5, then refer to the server's site for specific instructions.

- 4. Q: How can I connect to a server that uses a self-signed certificate?
  A: Connecting to such a server raises the SL_ERROR_BSD_ESECUNKNOWNROOTCA connection error. This is a security warning that does not break the connection. The user may decide to ignore the warning or to avoid the catalog check in advance (for this socket) – as demonstrated in Section 2.5.5).

7 References

1. CC3220 SimpleLink™ Wi-Fi® and Internet of Things Solution, a Single-Chip Wireless MCU Getting Started Guide
2. CC3x20, CC3x35 SimpleLink™ Wi-Fi® and Internet of Things Network Processor Programmers Guide
3. CC3x20, CC3x35 SimpleLink™ Wi-Fi® Internet-on-a chip™ Solution Built-in Security Features Application Report
4. OpenSSL
5. Over The Air SLA (SimpleLink Academy)
6. SimpleLink™ Wi-Fi® Secure File (SLA)
7. SimpleLink™ Wi-Fi® Secure Sockets (SLA)

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Original (July 2018) to A Revision **Page**

- Added Section 1.4.3 9
- Changed NOTE in Section 2.1 13
- Changed Section 3 22
- Added Section 3.1 23
- Changed Section 3.1.1 24
- Changed Section 3.1.2 29
- Changed Section 3.1.3 29
- Changed Section 3.1.4 30
- Added Section 3.2 33