AWS IoT Device SDK for Embedded C

Step1: Install the AWS IoT Device SDK for Embedded C

Download the AWS IoT Device SDK for Embedded C to your device from GitHub

```
git clone https://github.com/aws/aws-iot-device-sdk-embedded-c.git --recurse-submodules
```

Install OpenSSL version 1.1.0 or later. The OpenSSL development libraries are usually called "libssl-dev" or "openssl-devel" when installed through a package manager.

```
sudo apt-get install libssl-dev
```

Step 2: Configure the sample app

You must configure the sample with your personal AWS IoT Core endpoint, private key, certificate, and root CA certificate. Navigate to the aws-iot-device-sdk-embedded-c/demos/mqtt/mqtt_demo_mutual_auth directory.

```
// Get from demo_config.h
// =================================================
#define AWS_IOT_ENDPOINT               "my-endpoint-ats.iot.us-east-1.amazonaws.com"
#define AWS_MQTT_PORT                  8883
#define CLIENT_IDENTIFIER              "testclient"
#define ROOT_CA_CERT_PATH              "certificates/AmazonRootCA1.crt"
#define CLIENT_CERT_PATH               "certificates/my-device-cert.pem.crt"
#define CLIENT_PRIVATE_KEY_PATH        "certificates/my-device-private-key.pem.key"
// =================================================
```

Step3: Build and run the sample application To run the AWS IoT Device SDK for Embedded C sample applications

Navigate to aws-iot-device-sdk-embedded-c and create a build directory.

```
mkdir build && cd build
```

Step4: Enter the following CMake command to generate the Makefiles needed to build.

```
cmake ..
```

Step5: Enter the following command to build the executable app file.

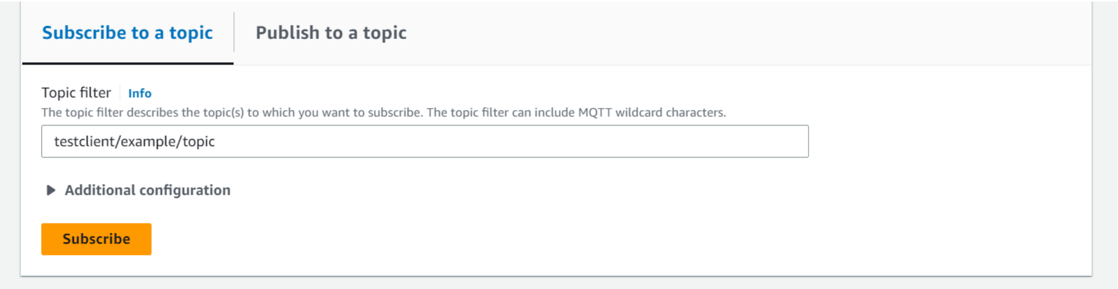
```
make
```

Step6: Run the mqtt_demo_mutual_auth app with this command.

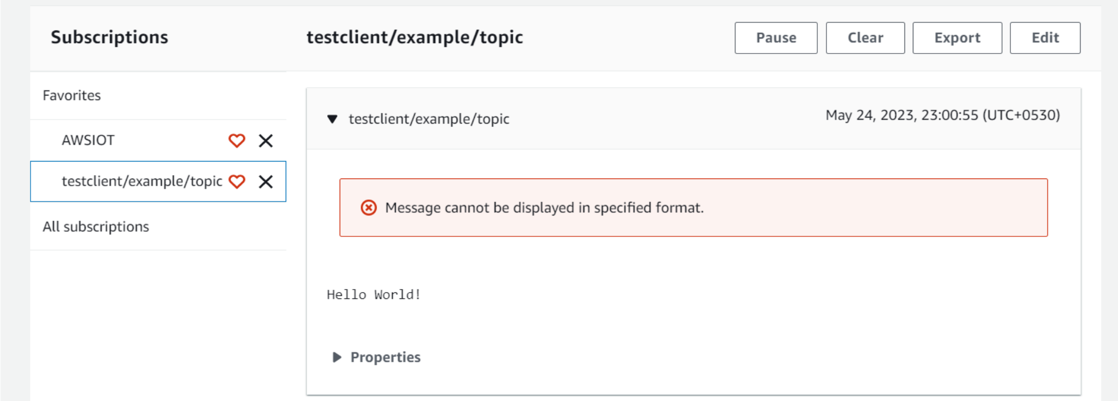
```
cd bin
./mqtt_demo_mutual_auth
```

View MQTT messages with the AWS IoT MQTT client

In the Subscribe to a topic tab, enter the topicName to subscribe to the topic on which your device publishes.

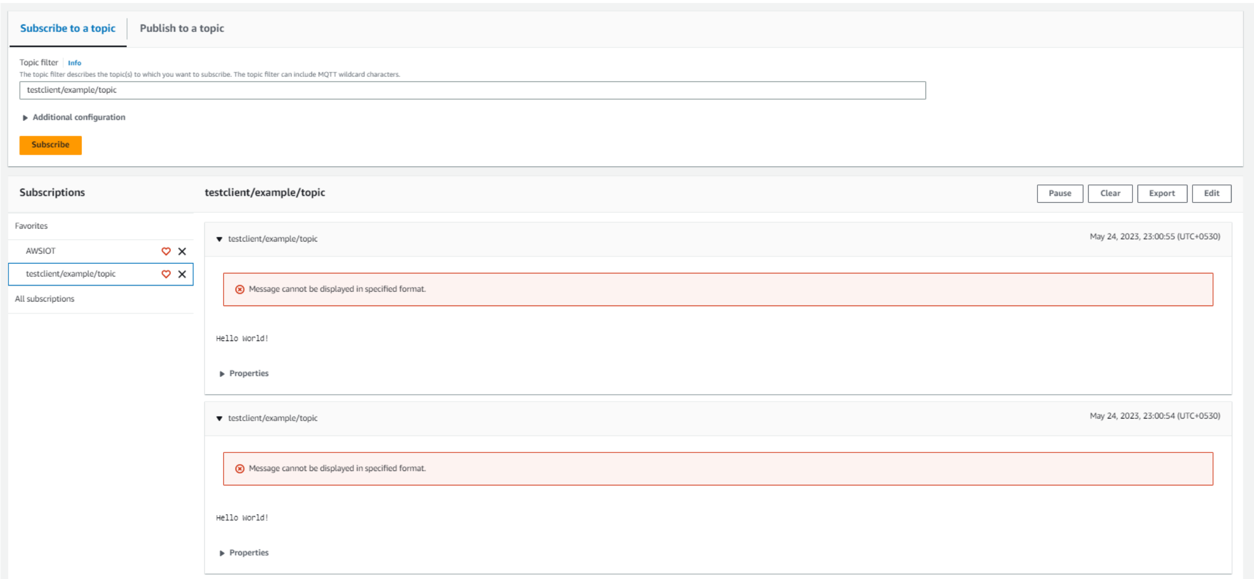


The topic message log page, opens and appears in the Subscriptions list. If the device that you configured in Configure your device is running the example program, you should see the messages it sends to AWS IoT in the message log. The message log entries will appear below the Publish section when messages with the subscribed topic are received by AWS IoT.

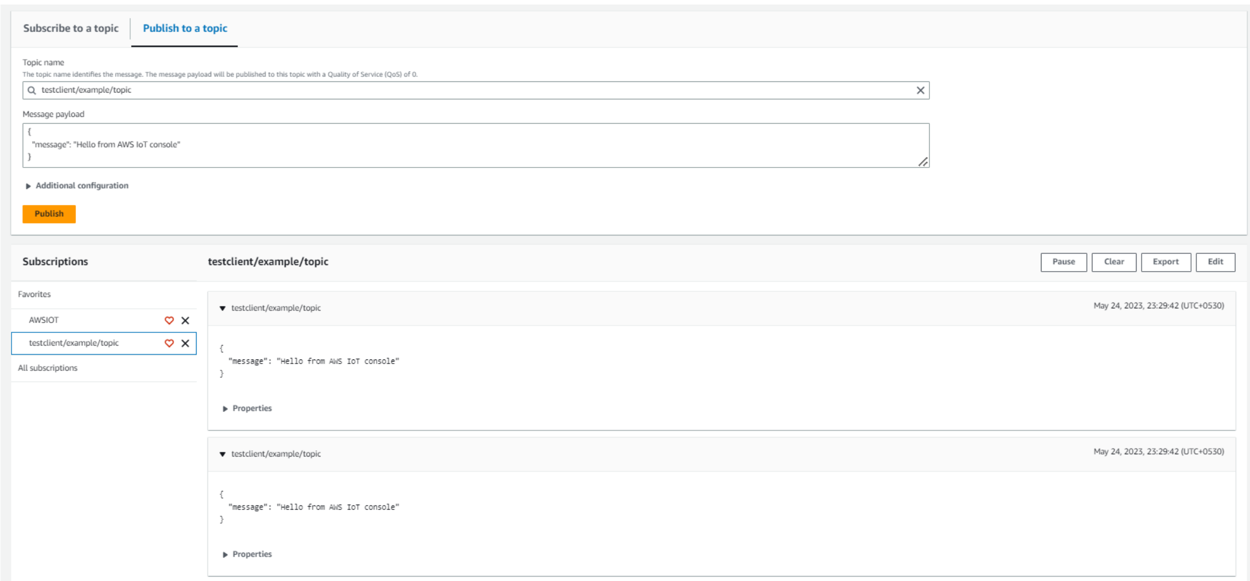


Choose one MQTT client, in the Publish to a topic tab, in the Topic name field, enter the topicName of your message. In this example, testclient/example/topic. Try publishing the message a few times. From the Subscriptions list of both MQTT clients, you should be able to see that the clients receive the message. In this example, we publish the same message "Hello from AWS IoT console".

Subscribe to a topic



Publish to a topic



You should see output similar to the following:

```
[INFO] [MQTT] [core_mqtt.c:1385] De-serialized incoming PUBLISH packet: DeserializerResult=MQTTSuccess.
[INFO] [MQTT] [core_mqtt.c:1413] State record updated. New state=MQTTPubAckSend.
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:881] Incoming QOS : 1.
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:889] Incoming Publish Topic Name: testclient/example/topic matches subscribed topic.
Incoming Publish message Packet Id is 1.
Incoming Publish Message : Hello World!.


[INFO] [DEMO] [mqtt_demo_mutual_auth.c:1478] Delay before continuing to next iteration.


[INFO] [DEMO] [mqtt_demo_mutual_auth.c:1488] Unsubscribing from the MQTT topic testclient/example/topic.
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:1278] UNSUBSCRIBE sent for topic testclient/example/topic to broker.


[INFO] [MQTT] [core_mqtt.c:1517] Ack packet deserialized with result: MQTTSuccess.
[INFO] [MQTT] [core_mqtt.c:1534] State record updated. New state=MQTTPublishDone.
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:1081] PUBACK received for packet id 21.


[INFO] [DEMO] [mqtt_demo_mutual_auth.c:791] Cleaned up outgoing publish packet with packet id 21.


[INFO] [MQTT] [core_mqtt.c:1385] De-serialized incoming PUBLISH packet: DeserializerResult=MQTTSuccess.
[INFO] [MQTT] [core_mqtt.c:1413] State record updated. New state=MQTTPubAckSend.
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:881] Incoming QOS : 1.
[INFO] [DEMO] [mqtt_demo_mutual_auth.c:889] Incoming Publish Topic Name: testclient/example/topic matches subscribed topic.
Incoming Publish message Packet Id is 1.
Incoming Publish Message : Hello World!.
```

AWS-IOT

Demo

In demo_config.h

Defining the endpoint, private key and the certificate.

```
/**
 * @brief Details of the MQTT broker to connect to.
 *
 * @note Your AWS IoT Core endpoint can be found in the AWS IoT console under
 * Settings/Custom Endpoint, or using the describe-endpoint API.
 */
 * #define AWS_IOT_ENDPOINT   "...insert here..."


/**
 * @brief AWS IoT MQTT broker port number.
 *
 * In general, port 8883 is for secured MQTT connections.
 *
 * @note Port 443 requires use of the ALPN TLS extension with the ALPN protocol
 * name. When using port 8883, ALPN is not required.
 */
#ifndef AWS_MQTT_PORT
    #define AWS_MQTT_PORT    ( 8883 )
#endif

/**
 * @brief Path of the file containing the server's root CA certificate.
 *
 * This certificate is used to identify the AWS IoT server and is publicly
 * available. Refer to the AWS documentation available in the link below
 * https://docs.aws.amazon.com/iot/latest/developerguide/server-authentication.html#server-authentication-certs
 *
 * Amazon's root CA certificate is automatically downloaded to the certificates
 * directory from @ref https://www.amazontrust.com/repository/AmazonRootCA1.pem
 * using the CMake build system.
 *
 * @note This certificate should be PEM-encoded.
 * @note This path is relative from the demo binary created. Update
 * ROOT_CA_CERT_PATH to the absolute path if this demo is executed from elsewhere.
 */
#ifndef ROOT_CA_CERT_PATH
    #define ROOT_CA_CERT_PATH    "certificates/AmazonRootCA1.crt"
#endif

/**
 * @brief Path of the file containing the client certificate.
 *
 * Refer to the AWS documentation below for details regarding client
 * authentication.
 * https://docs.aws.amazon.com/iot/latest/developerguide/client-authentication.html
 *
 * @note This certificate should be PEM-encoded.
 */
 * #define CLIENT_CERT_PATH   "...insert here..."


/**
 * @brief Path of the file containing the client's private key.
 *
 * Refer to the AWS documentation below for details regarding client
 * authentication.
 * https://docs.aws.amazon.com/iot/latest/developerguide/client-authentication.html
 *
 * @note This private key should be PEM-encoded.
 */
 * #define CLIENT_PRIVATE_KEY_PATH    "...insert here..."
```

```
/**
     * @brief Filepaths to certificates and private key that are used when
     * performing the TLS handshake.
     *
     * @note These strings must be NULL-terminated because the OpenSSL API requires them to be.
     */
    const char * pRootCaPath;     /**< @brief Filepath string to the trusted server root CA. */
    const char * pClientCertPath; /**< @brief Filepath string to the client certificate. */
    const char * pPrivateKeyPath; /**< @brief Filepath string to the client certificate's private key. */
} OpensslCredentials_t;
```

```
    /* Initialize credentials for establishing TLS session. */
```

Parameter	Description
`ROOT_CA_CERT_PATH`	opensslCredentials.pRootCaPath = ROOT_CA_CERT_PATH;
`CLIENT_CERT_PATH`	opensslCredentials.pClientCertPath = CLIENT_CERT_PATH;
`CLIENT_PRIVATE_KEY_PATH`	opensslCredentials.pPrivateKeyPath = CLIENT_PRIVATE_KEY_PATH;
`AWS_IOT_ENDPOINT`	opensslCredentials.sniHostName = AWS_IOT_ENDPOINT;

mqtt_demo_mutual_auth\mqtt_demo_mutual_auth.c

TLS authenticatION

```
/**
 * AWS IoT Core TLS ALPN definitions for MQTT authentication
 * These configuration settings are required to run the mutual auth demo.
 * Throw compilation error if the below configs are not defined.
 */
#ifndef AWS_IOT_ENDPOINT
    #error "Please define AWS IoT MQTT broker endpoint(AWS_IOT_ENDPOINT) in demo_config.h."
#endif
#ifndef ROOT_CA_CERT_PATH
    #error "Please define path to Root CA certificate of the MQTT broker(ROOT_CA_CERT_PATH) in demo_config.h."
#endif
#ifndef CLIENT_IDENTIFIER
    #error "Please define a unique client identifier, CLIENT_IDENTIFIER, in demo_config.h."
#endif

/* The AWS IoT message broker requires either a set of client certificate/private key
 * or username/password to authenticate the client. */
#ifndef CLIENT_USERNAME
    #ifndef CLIENT_CERT_PATH
        #error "Please define path to client certificate(CLIENT_CERT_PATH) in demo_config.h."
    #endif
    #ifndef CLIENT_PRIVATE_KEY_PATH
        #error "Please define path to client private key(CLIENT_PRIVATE_KEY_PATH) in demo_config.h."
    #endif
#else

/* If a username is defined, a client password also would need to be defined for
 * client authentication. */
    #ifndef CLIENT_PASSWORD
        #error "Please define client password(CLIENT_PASSWORD) in demo_config.h for client authentication based on username/password."
    #endif

/* AWS IoT MQTT broker port needs to be 443 for client authentication based on
 * username/password. */
    #if AWS_MQTT_PORT != 443
        #error "Broker port, AWS_MQTT_PORT, should be defined as 443 in demo_config.h for client authentication based on username/password.
    #endif
#endif /* ifndef CLIENT_USERNAME */
```

connectToServer

```
/**
 * @brief Path of the file containing the client's private key.
 *
 * Refer to the AWS documentation below for details regarding client
 * authentication.
 * https://docs.aws.amazon.com/iot/latest/developerguide/client-authentication.html
 *
 * @note This private key should be PEM-encoded.
 *
 * #define CLIENT_PRIVATE_KEY_PATH    "...insert here..."
 */

/* Initialize reconnect attempts and interval */
    BackoffAlgorithm_InitializeParams( &reconnectParams,
                                       CONNECTION_RETRY_BACKOFF_BASE_MS,
                                       CONNECTION_RETRY_MAX_BACKOFF_DELAY_MS,
                                       CONNECTION_RETRY_MAX_ATTEMPTS );

    /* Attempt to connect to HTTP server. If connection fails, retry after
     * a timeout. Timeout value will exponentially increase until maximum
     * attempts are reached. */
    do
    {
        returnStatus = connectFunction( pNetworkContext );

        if( returnStatus != EXIT_SUCCESS )
        {
            /* Generate a random number and get back-off value (in milliseconds) for the next connection retry. */
            backoffAlgStatus = BackoffAlgorithm_GetNextBackoff( &reconnectParams, generateRandomNumber(), &nextRetryBackOff );

            if( backoffAlgStatus == BackoffAlgorithmSuccess )
            {
                LogWarn( ( "Connection to the HTTP server failed. Retrying "
                           "connection after %hu ms backoff.",
                           ( unsigned short ) nextRetryBackOff ) );
                Clock_SleepMs( nextRetryBackOff );
            }
            else
            {
                LogError( ( "Connection to the HTTP server failed, all attempts exhausted." ) );
            }
        }
    } while( ( returnStatus == EXIT_FAILURE ) && ( backoffAlgStatus == BackoffAlgorithmSuccess ) );

    if( returnStatus == EXIT_FAILURE )
    {
        LogError( ( "Connection to the server failed, all attempts exhausted." ) );
    }

    return returnStatus;
}


/* Initialize information to connect to the MQTT broker. */

    serverInfo.pHostName = AWS_IOT_ENDPOINT;
    serverInfo.hostNameLength = AWS_IOT_ENDPOINT_LENGTH;
    serverInfo.port = AWS_MQTT_PORT;

    /* If #CLIENT_USERNAME is defined, username/password is used for authenticating
     * the client. */
    #ifndef CLIENT_USERNAME
        opensslCredentials.pClientCertPath = CLIENT_CERT_PATH;
        opensslCredentials.pPrivateKeyPath = CLIENT_PRIVATE_KEY_PATH;
    #endif
```

openssl

```
 /* OpenSSL sockets transport implementation. */

#include "openssl_posix.h"

/* Each compilation unit must define the NetworkContext struct. */
struct NetworkContext
{
    OpensslParams_t * pParams;
};
/*-----------------------------------------------------------*/

/**
 * @brief The network context used for Openssl operation.
 */
static NetworkContext_t networkContext = { 0 };

static bool connectToBrokerWithBackoffRetries( NetworkContext_t * pNetworkContext )
{
    bool returnStatus = false;
    BackoffAlgorithmStatus_t backoffAlgStatus = BackoffAlgorithmSuccess;
    OpensslStatus_t opensslStatus = OPENSSL_SUCCESS;
    BackoffAlgorithmContext_t reconnectParams;
    ServerInfo_t serverInfo;
    OpensslCredentials_t opensslCredentials;
    uint16_t nextRetryBackOff = 0U;
    struct timespec tp;

    /* Set the pParams member of the network context with desired transport. */
    pNetworkContext->pParams = &opensslParams;

    /* Initialize information to connect to the MQTT broker. */
    serverInfo.pHostName = AWS_IOT_ENDPOINT;
    serverInfo.hostNameLength = AWS_IOT_ENDPOINT_LENGTH;
    serverInfo.port = AWS_MQTT_PORT;

    /* Initialize credentials for establishing TLS session. */
    ( void ) memset( &opensslCredentials, 0, sizeof( OpensslCredentials_t ) );
    opensslCredentials.pRootCaPath = ROOT_CA_CERT_PATH;
    opensslCredentials.pClientCertPath = CLIENT_CERT_PATH;
    opensslCredentials.pPrivateKeyPath = CLIENT_PRIVATE_KEY_PATH;
    opensslCredentials.sniHostName = AWS_IOT_ENDPOINT;


        /* Establish a TLS session with the MQTT broker. This example connects
         * to the MQTT broker as specified in AWS_IOT_ENDPOINT and AWS_MQTT_PORT
         * at the demo config header. */
        LogDebug( ( "Establishing a TLS session to %.*s:%d.",
                    AWS_IOT_ENDPOINT_LENGTH,
                    AWS_IOT_ENDPOINT,
                    AWS_MQTT_PORT ) );
        opensslStatus = Openssl_Connect( pNetworkContext,
                                         &serverInfo,
                                         &opensslCredentials,
                                         TRANSPORT_SEND_RECV_TIMEOUT_MS,
                                         TRANSPORT_SEND_RECV_TIMEOUT_MS );

        if( opensslStatus == OPENSSL_SUCCESS )
        {
            /* Connection successful. */
            returnStatus = true;

        Greengrass creates a safe link to the cloud by enabling local device communications via the MQTT protocol.

        Greeengrass_auth.c(source file)


        /* Initialize credentials for establishing TLS session. */
    opensslCredentials.pRootCaPath = ROOT_CA_CERT_PATH;
    opensslCredentials.pClientCertPath = CLIENT_CERT_PATH;
```

```
    opensslCredentials.pClientCertPath = CLIENT_CERT_PATH;
    opensslCredentials.pPrivateKeyPath = CLIENT_PRIVATE_KEY_PATH;

    /* Initialize reconnect attempts and interval */
    BackoffAlgorithm_InitializeParams( &reconnectParams,
                                       500U,  /* Backoff base ms */
                                       5000U, /* Max backoff delay ms */
                                       5U );  /* Max attempts */

    /* Attempt to connect to the core's broker. If connection fails, retry after
     * backoff period. */
    do
    {
        LogInfo( ( "Establishing a TLS session to %.*s:%d.",
                   GREENGRASS_ADDRESS_LENGTH,
                   GREENGRASS_ADDRESS,
                   MQTT_PORT ) );

        opensslStatus = Openssl_Connect( pNetworkContext,
                                         &serverInfo,
                                         &opensslCredentials,
                                         TRANSPORT_SEND_RECV_TIMEOUT_MS,
                                         TRANSPORT_SEND_RECV_TIMEOUT_MS );

        if( opensslStatus == OPENSSL_SUCCESS )
        {
            /* Sends an MQTT Connect packet using the established TLS session,
             * then waits for connection acknowledgment (CONNACK) packet. */
            returnStatus = establishMqttSession( pMqttContext );

            if( returnStatus == EXIT_FAILURE )
            {
                /* End TLS session, then close TCP connection. */
                ( void ) Openssl_Disconnect( pNetworkContext );
            }
aws-iot-device-sdk-embedded-C-main\demos\greengrass\greengrass_demo_local_auth\openssl_posix.c
/**
 * @brief Passes TLS credentials to the OpenSSL library.
 *
 * Provides the root CA certificate, client certificate, and private key to the
 * OpenSSL library. If the client certificate or private key is not NULL, mutual
 * authentication is used when performing the TLS handshake.
 *
 * @param[out] pSslContext SSL context to which the credentials are to be
 * imported.
 * @param[in] pOpensslCredentials TLS credentials to be imported.
 *
 * @return 1 on success; -1, 0 on failure.
 */
static int32_t setCredentials( SSL_CTX * pSslContext,
                               const OpensslCredentials_t * pOpensslCredentials );


/**
 * @brief Establish TLS session by performing handshake with the server.
 *
 * @param[in] pServerInfo Server connection info.
 * @param[in] pOpensslParams Parameters to perform the TLS handshake.
 * @param[in] pOpensslCredentials TLS credentials containing configurations.
 *
 * @return #OPENSSL_SUCCESS, #OPENSSL_API_ERROR, and #OPENSSL_HANDSHAKE_FAILED.
 */
static OpensslStatus_t tlsHandshake( const ServerInfo_t * pServerInfo,
                                     OpensslParams_t * pOpensslParams,
                                     const OpensslCredentials_t * pOpensslCredentials );

 * @param[in] pNetworkContext The network context created using Openssl_Connect API.
 * @param[in] pBuffer Buffer containing the bytes to send over the network stack.
 * @param[in] bytesToSend Number of bytes to send over the network.
 *
```

```
 *
 * @return Number of bytes sent if successful; negative value on error.
 *
 * @note This function does not return zero value because it cannot be retried
 * on send operation failure.
 */
int32_t Openssl_Send( NetworkContext_t * pNetworkContext,
                      const void * pBuffer,
                      size_t bytesToSend );

   /*************************** Connect. *****************************/

            /* Establish a TLS connection on top of TCP connection using OpenSSL. */

            /* Attempt to connect to the HTTP server. If connection fails, retry
             * after a timeout. The timeout value will be exponentially
             * increased till the maximum attempts are reached or maximum
             * timeout value is reached. The function returns EXIT_FAILURE if
             * the TCP connection cannot be established to broker after
             * the configured number of attempts. */
            returnStatus = connectToServerWithBackoffRetries( connectToServer,
                                                              &networkContext );

            /* Define the transport interface. */
            if( returnStatus == EXIT_SUCCESS )
            {
                transportInterface.recv = Openssl_Recv;
                transportInterface.send = Openssl_Send;
                transportInterface.pNetworkContext = &networkContext;
            }
```

Generating a self-signed certificate using OpenSSL is a relatively simple process. The first step is to generate the key pair, which has a private key as well as a public key. This will be used to sign the certificate in Step 4. The second step is to extract the public key from the key pair. The third step is to generate a Certificate Signing Request (CSR). This will be used by the certificate authority (CA) to create the self-signed certificate.

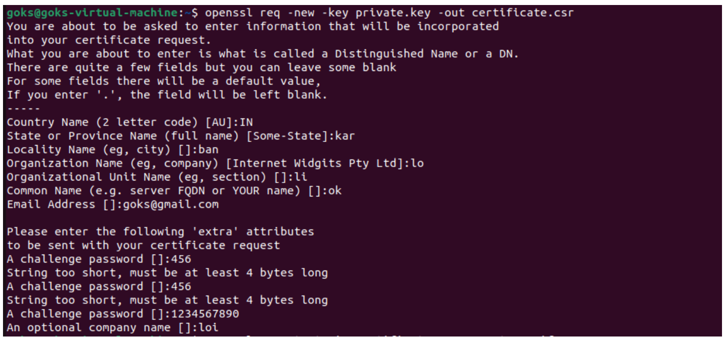
Generate key pair Extract public key Generate csr file generate self sign certificate

This command will generate an RSA key pair with a length of 2048.

```
openssl genrsa -out private.key 2048
```

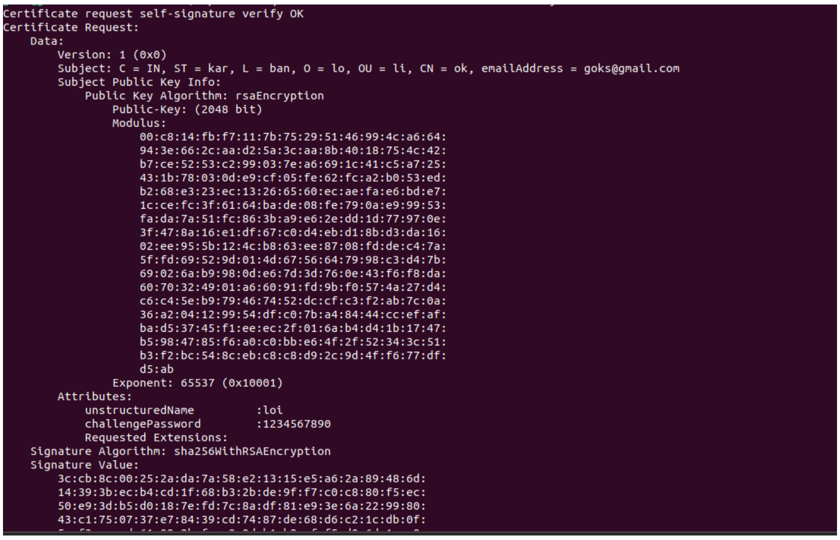
Extract the public key from the key pair Run this command to extract the public key from the key pair generated in step 1.

```
$ openssl rsa -in private.key -pubout -out public.key
```



Create a Certificate Signing Request (CSR) The next step is to generate a Certificate Signing Request (CSR). This will be used by the certificate authority (CA) to create the self-signed certificate. To generate the CSR, run this command in your terminal:

```
openssl req -new -key private.key -out certificate.csr
```



Create A Self-Signed Certificate Using Web-Based GUI OpenSSL Finally, generate the self-signed certificate using the private key and CSR. Run this command to generate the self-signed certificate on the terminal:

```
$ openssl x509 -in certificate.csr -out certificate.crt -req -signkey private.key -days 365
```

handshake

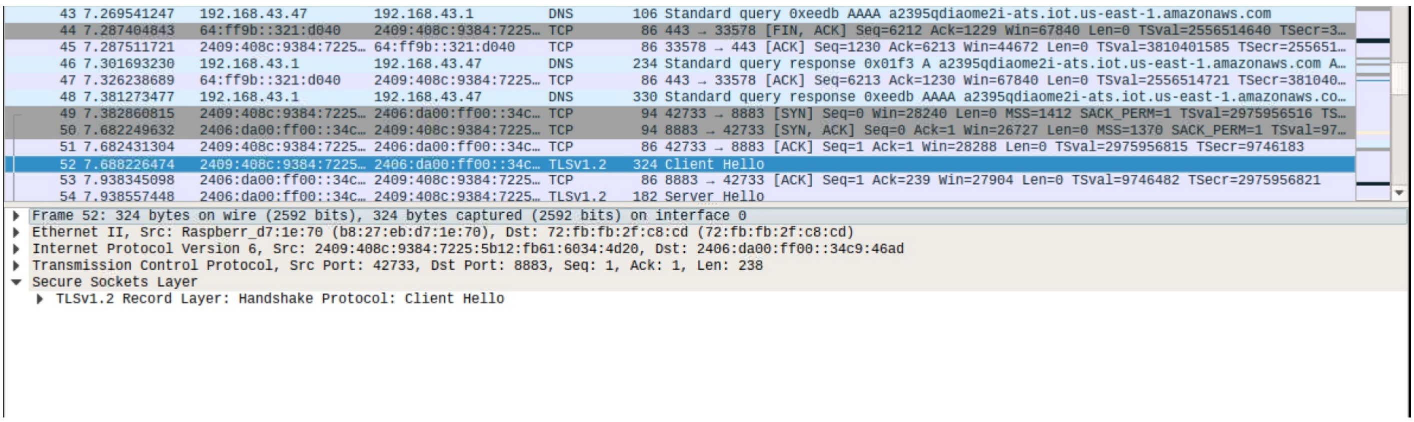
OpenSSL is an open-source command line tool that is commonly used to generate private keys, create CSRs, install your SSL/TLS certificate, and identify certificate information.

```
    /**
     * @brief Filepaths to certificates and private key that are used when
     * performing the TLS handshake.
     *
     * @note These strings must be NULL-terminated because the OpenSSL API requires them to be.
     */
    const char * pRootCaPath;     /**< @brief Filepath string to the trusted server root CA. */
    const char * pClientCertPath; /**< @brief Filepath string to the client certificate. */
    const char * pPrivateKeyPath; /**< @brief Filepath string to the client certificate's private key. */
} OpensslCredentials_t;

static OpensslStatus_t tlsHandshake( const ServerInfo_t * pServerInfo,
                                     OpensslParams_t * pOpensslParams,
                                     const OpensslCredentials_t * pOpensslCredentials )
{
    OpensslStatus_t returnStatus = OPENSSL_SUCCESS;
    int32_t sslStatus = -1, verifyPeerCertStatus = X509_V_OK;

    /* Validate the hostname against the server's certificate. */
    sslStatus = SSL_set1_host( pOpensslParams->pSsl, pServerInfo->pHostName );

    if( sslStatus != 1 )
    {
        LogError( ( "SSL_set1_host failed to set the hostname to validate." ) );
        returnStatus = OPENSSL_API_ERROR;
    }

    /* Enable SSL peer verification. */
    if( returnStatus == OPENSSL_SUCCESS )
    {
        SSL_set_verify( pOpensslParams->pSsl, SSL_VERIFY_PEER, NULL );

        /* Setup the socket to use for communication. */
        sslStatus =
            SSL_set_fd( pOpensslParams->pSsl, pOpensslParams->socketDescriptor );

        if( sslStatus != 1 )
        {
            LogError( ( "SSL_set_fd failed to set the socket fd to SSL context." ) );
            returnStatus = OPENSSL_API_ERROR;
```

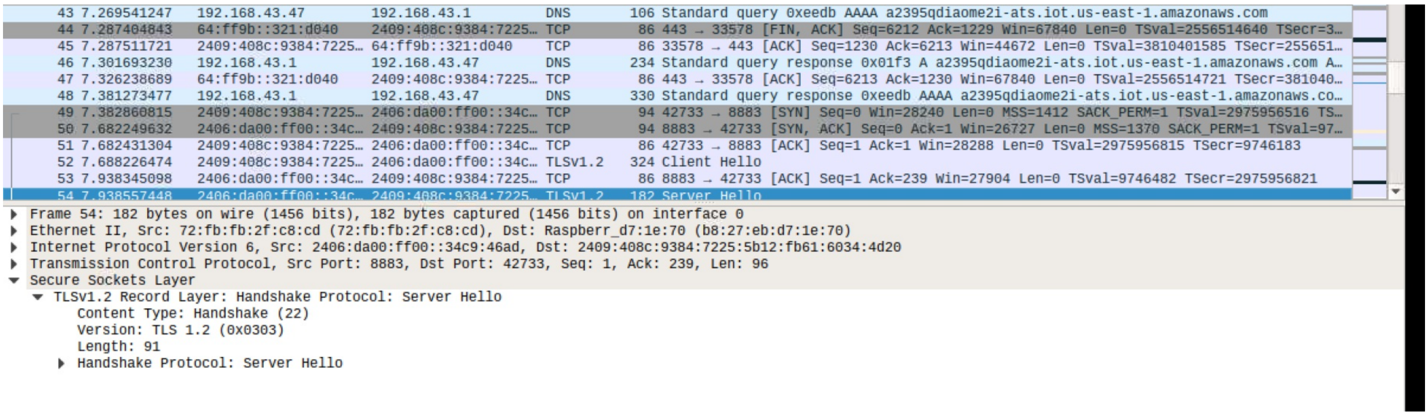
```
        }
    }

    /* Perform the TLS handshake. */
    if( returnStatus == OPENSSL_SUCCESS )
    {
        setOptionalConfigurations( pOpensslParams->pSsl, pOpensslCredentials );

        sslStatus = SSL_connect( pOpensslParams->pSsl );

        if( sslStatus != 1 )
        {
            LogError( ( "SSL_connect failed to perform TLS handshake." ) );
            returnStatus = OPENSSL_HANDSHAKE_FAILED;
        }
    }

    /* Verify X509 certificate from peer. */
    if( returnStatus == OPENSSL_SUCCESS )
    {
        verifyPeerCertStatus = ( int32_t ) SSL_get_verify_result( pOpensslParams->pSsl );

        if( verifyPeerCertStatus != X509_V_OK )
        {
            LogError( ( "SSL_get_verify_result failed to verify X509 "
                        "certificate from peer." ) );
            returnStatus = OPENSSL_HANDSHAKE_FAILED;
        }
    }

    return returnStatus;
}
```

Wireshark is a network protocol analyzer, or an application that captures packets from a network connection, such as from your computer to your home office or the internet.

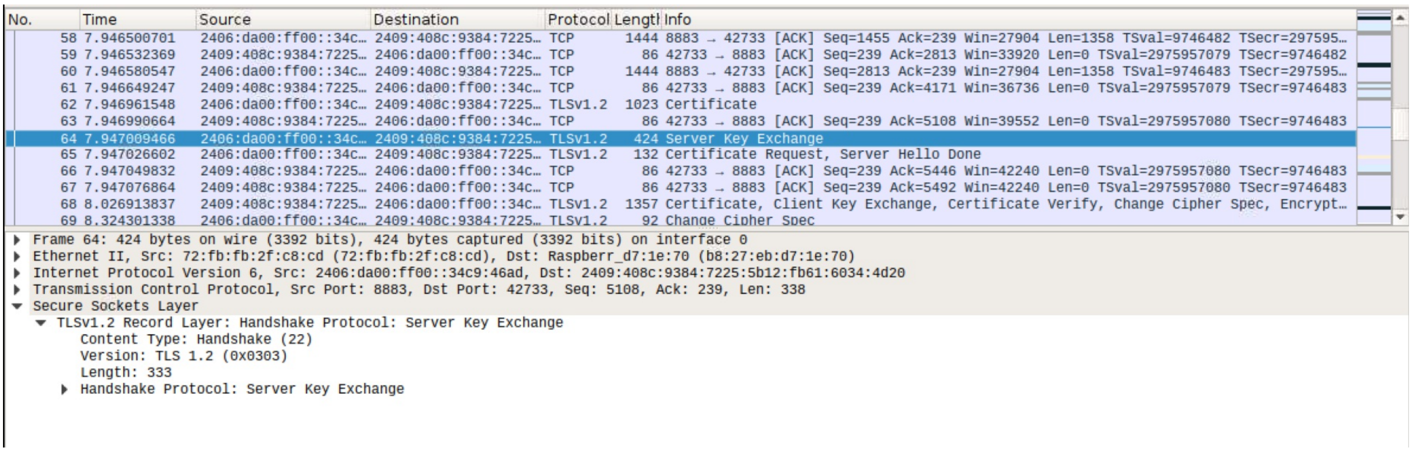
```
The 'client hello' message: The client initiates the handshake by sending a "hello" message to the server. The message will include which T
```



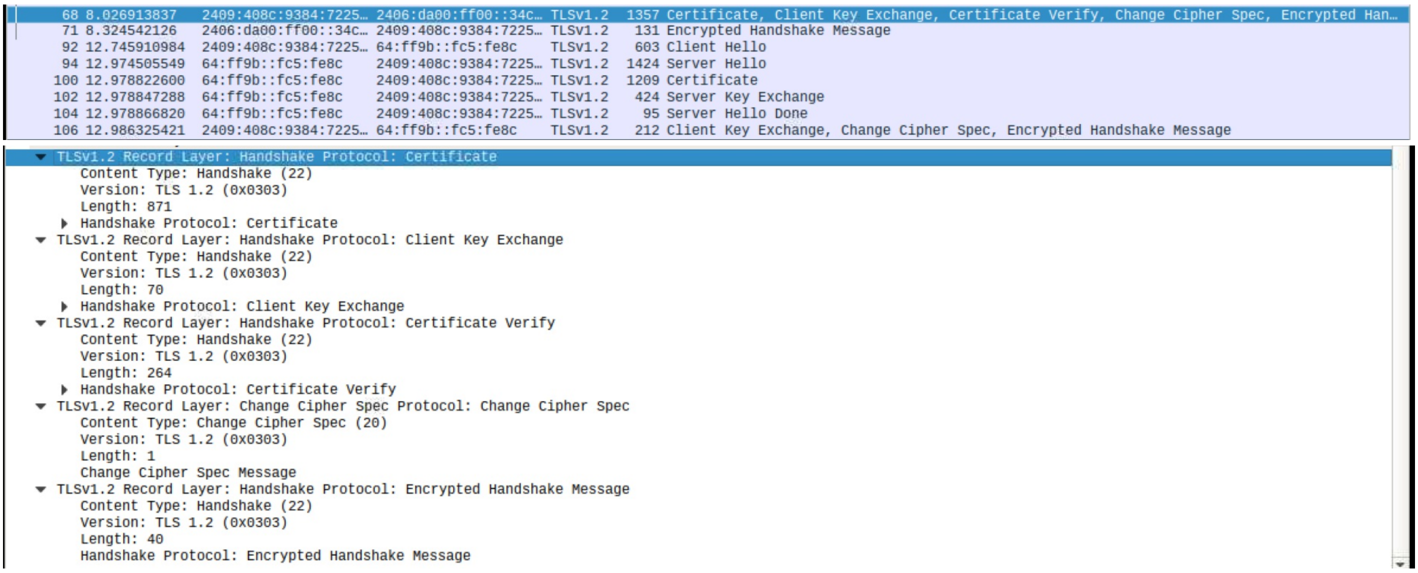

```
The 'server hello' message: In reply to the client hello message, the server sends a message containing the server's SSL certificate, the s
```



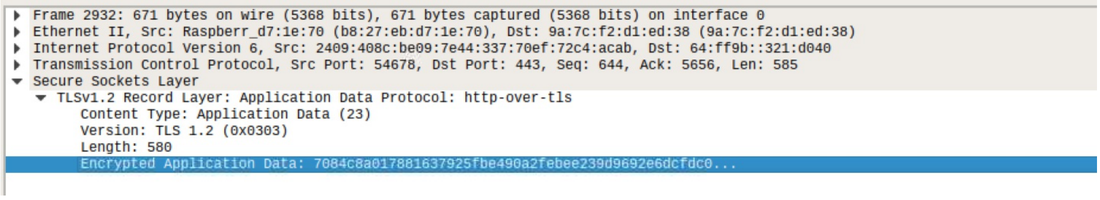
The client performs authentication by contacting the server's certificate authority (CA) to validate the web server's digital certificate. This confirms the authenticity of the web server, thus, establishing trust.

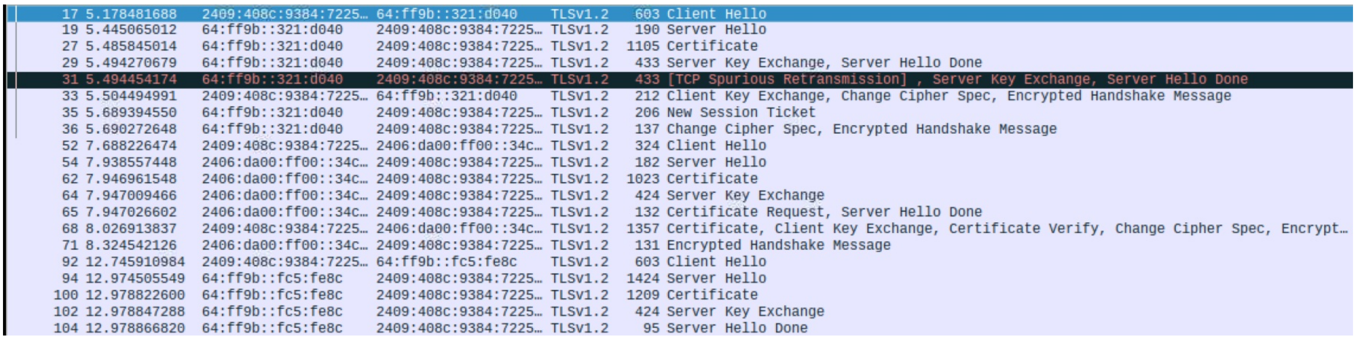



```
 During the ClientKeyExchange step, the client extracts the public key from the verified certificate and generates a new random sequence cal

Both the client and the server now use the premaster secret to configure a shared secret key. Next, the client sends an encrypted “finished
```




```
 Step 8: Finally, an encrypted “finished” message is sent back to the client from the server using the previously agreed shared secret key,
Step 9: Once the SSL/TLS handshake and negotiation is done, the server and the client communication continues, i.e., they begin to share fi
```





certificate signing request (CSR)

A certificate signing request (CSR) is an encoded file. This information is used by a Certificate Authority (CA) to create an SSL/TLS certificate for your website to encrypt traffic to your site.

Public-Key Cryptography Standards

The sender encrypts, or scrambles, the data before sending it. The receiver decrypts, or unscrambles, the data after receiving it.

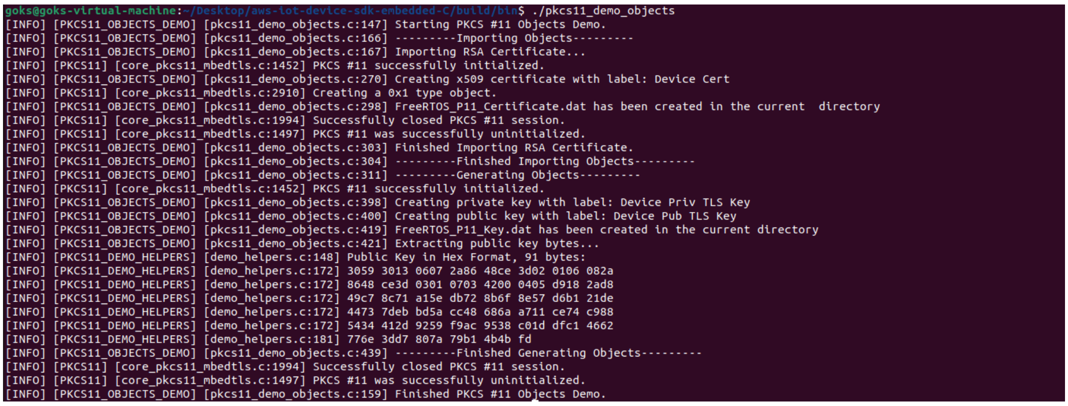
Elliptic Curve Digital Signature Algorithm

Elliptic Curve Cryptography (ECC) is a newer algorithm that offers shorter keys that achieve comparable strengths when compared with longer RSA keys. Very fast key generation. Smaller keys, cipher-texts, and signatures. Fast signatures. Signatures can be computed in two stages, allowing latency much lower. Moderately fast encryption and decryption. Than inverse throughput. RSA allows you to secure messages before you send them. And the technique also lets you certify your notes, so recipients know they haven't been adjusted or altered while in transit. The RSA algorithm is one of the most widely used encryption tools

```
/**
 * @brief Creates the request payload to be published to the
 * CreateCertificateFromCsr API in order to request a certificate from AWS IoT
 * for the included Certificate Signing Request (CSR).
 *
 * @param[in] pBuffer Buffer into which to write the publish request payload.
 * @param[in] bufferLength Length of #pBuffer.
 * @param[in] pCsr The CSR to include in the request payload.
 * @param[in] csrLength The length of #pCsr.
 * @param[out] pOutLengthWritten The length of the publish request payload.
 */
bool generateCsrRequest( uint8_t * pBuffer,
                         size_t bufferLength,
                         const char * pCsr,
                         size_t csrLength,
                         size_t * pOutLengthWritten );

  * This demo provisions a device certificate using the provisioning by claim
 * workflow with a Certificate Signing Request (CSR). The demo connects to AWS
 * IoT Core using provided claim credentials (whose certificate needs to be
 * registered with IoT Core before running this demo), subscribes to the
 * CreateCertificateFromCsr topics, and obtains a certificate. It then
 * subscribes to the RegisterThing topics and activates the certificate and
 * obtains a Thing using the provisioning template. Finally, it reconnects to
 * AWS IoT Core using the new credentials.
 */

      /**** Call the CreateCertificateFromCsr API ***************************/

        /* We use the CreateCertificatefromCsr API to obtain a client certificate
         * for a key on the device by means of sending a certificate signing
         * request (CSR). */
        if( status == true )
        {
```

```
        {
            /* Subscribe to the CreateCertificateFromCsr accepted and rejected
             * topics. In this demo we use CBOR encoding for the payloads,
             * so we use the CBOR variants of the topics. */
            status = subscribeToCsrResponseTopics();
        }

        if( status == true )
        {
            /* Create a new key and CSR. */
            status = generateKeyAndCsr( p11Session,
                                        pkcs11configLABEL_DEVICE_PRIVATE_KEY_FOR_TLS,
                                        pkcs11configLABEL_DEVICE_PUBLIC_KEY_FOR_TLS,
                                        csr,
                                        CSR_BUFFER_LENGTH,
                                        &csrLength );
        }

        if( status == true )
        {
            /* Create the request payload containing the CSR to publish to the
             * CreateCertificateFromCsr APIs. */
            status = generateCsrRequest( payloadBuffer,
                                         NETWORK_BUFFER_SIZE,
                                         csr,
                                         csrLength,
                                         &payloadLength );
        }

        if( status == true )
        {
            /* Publish the CSR to the CreateCertificatefromCsr API. */
            status = PublishToTopic( FP_CBOR_CREATE_CERT_PUBLISH_TOPIC,
                                     FP_CBOR_CREATE_CERT_PUBLISH_LENGTH,
                                     ( char * ) payloadBuffer,
                                     payloadLength );

            if( status == false )
            {
                LogError( ( "Failed to publish to fleet provisioning topic: %.*s.",
                            FP_CBOR_CREATE_CERT_PUBLISH_LENGTH,
                            FP_CBOR_CREATE_CERT_PUBLISH_TOPIC ) );
            }
        }

 A code signing certificate based on the SHA-256 ECDSA algorithm

/**
 * @brief MbedTLS callback for signing using the provisioned private key. Used for
 * signing the CSR.
 *
 * @param[in] pContext Unused.
 * @param[in] mdAlg Unused.
 * @param[in] pHash Data to sign.
 * @param[in] hashLen Length of #pHash.
 * @param[out] pSig The signature
 * @param[out] pSigLen The length of the signature.
 * @param[in] pRng Unused.
 * @param[in] pRngContext Unused.
 */
static int32_t privateKeySigningCallback( void * pContext,
                                          mbedtls_md_type_t mdAlg,
                                          const unsigned char * pHash,
                                          size_t hashLen,
                                          unsigned char * pSig,
                                          size_t * pSigLen,
                                          int ( * pRng )( void *, unsigned char *, size_t ),
                                          void * pRngContext );
```

```c
/**
 * @brief Generate a new public-private key pair in the PKCS #11 module, and
 * generate a certificate signing request (CSR) for them.
 *
 * This device-generated private key and CSR can be used with the
 * CreateCertificateFromCsr API of the the Fleet Provisioning feature of AWS IoT
 * Core in order to provision a unique client certificate.
 *
 * @param[in] p11Session The PKCS #11 session to use.
 * @param[in] pPrivKeyLabel PKCS #11 label for the private key.
 * @param[in] pPubKeyLabel PKCS #11 label for the public key.
 * @param[out] pCsrBuffer The buffer to write the CSR to.
 * @param[in] csrBufferLength Length of #pCsrBuffer.
 * @param[out] pOutCsrLength The length of the written CSR.
 *
 * @return True on success.
 */
bool generateKeyAndCsr( CK_SESSION_HANDLE p11Session,
                        const char * pPrivKeyLabel,
                        const char * pPubKeyLabel,
                        char * pCsrBuffer,
                        size_t csrBufferLength,
                        size_t * pOutCsrLength );
```



```
goks@goks-virtual-machine:~/Desktop/aws-iot-device-sdk-embedded-C/build/bin$ ./pkcs11_demo_objects
[INFO] [PKCS11_OBJECTS_DEMO] [pkcs11_demo_objects.c:147] Starting PKCS #11 Objects Demo.
[INFO] [PKCS11_OBJECTS_DEMO] [pkcs11_demo_objects.c:166] ---------Importing Objects---------
[INFO] [PKCS11_OBJECTS_DEMO] [pkcs11_demo_objects.c:167] Importing RSA Certificate...
[INFO] [PKCS11] [core_pkcs11_mbedtls.c:1452] PKCS #11 successfully initialized.
[INFO] [PKCS11_OBJECTS_DEMO] [pkcs11_demo_objects.c:270] Creating x509 certificate with label: Device Cert
[INFO] [PKCS11] [core_pkcs11_mbedtls.c:2910] Creating a 0x1 type object.
[INFO] [PKCS11_OBJECTS_DEMO] [pkcs11_demo_objects.c:298] FreeRTOS_P11_Certificate.dat has been created in the current  directory
[INFO] [PKCS11] [core_pkcs11_mbedtls.c:1994] Successfully closed PKCS #11 session.
[INFO] [PKCS11] [core_pkcs11_mbedtls.c:1497] PKCS #11 was successfully uninitialized.
[INFO] [PKCS11_OBJECTS_DEMO] [pkcs11_demo_objects.c:303] Finished Importing RSA Certificate.
[INFO] [PKCS11_OBJECTS_DEMO] [pkcs11_demo_objects.c:304] ---------Finished Importing Objects---------
[INFO] [PKCS11_OBJECTS_DEMO] [pkcs11_demo_objects.c:311] ---------Generating Objects---------
[INFO] [PKCS11] [core_pkcs11_mbedtls.c:1452] PKCS #11 successfully initialized.
[INFO] [PKCS11_OBJECTS_DEMO] [pkcs11_demo_objects.c:398] Creating private key with label: Device Priv TLS Key
[INFO] [PKCS11_OBJECTS_DEMO] [pkcs11_demo_objects.c:400] Creating public key with label: Device Pub TLS Key
[INFO] [PKCS11_OBJECTS_DEMO] [pkcs11_demo_objects.c:419] FreeRTOS_P11_Key.dat has been created in the current directory
[INFO] [PKCS11_OBJECTS_DEMO] [pkcs11_demo_objects.c:421] Extracting public key bytes...
[INFO] [PKCS11_DEMO_HELPERS] [demo_helpers.c:148] Public Key in Hex Format, 91 bytes:
[INFO] [PKCS11_DEMO_HELPERS] [demo_helpers.c:172] 3059 3013 0607 2a86 48ce 3d02 0106 082a
[INFO] [PKCS11_DEMO_HELPERS] [demo_helpers.c:172] 8648 ce3d 0301 0703 4200 0405 d918 2ad8
[INFO] [PKCS11_DEMO_HELPERS] [demo_helpers.c:172] 49c7 8c71 a15e db72 8b6f 8e57 d6b1 21de
[INFO] [PKCS11_DEMO_HELPERS] [demo_helpers.c:172] 4473 7deb bd5a cc48 686a a711 ce74 c988
[INFO] [PKCS11_DEMO_HELPERS] [demo_helpers.c:172] 5434 412d 9259 f9ac 9538 c01d dfc1 4662
[INFO] [PKCS11_DEMO_HELPERS] [demo_helpers.c:181] 776e 3dd7 807a 79b1 4b4b fd
[INFO] [PKCS11_OBJECTS_DEMO] [pkcs11_demo_objects.c:439] ---------Finished Generating Objects---------
[INFO] [PKCS11] [core_pkcs11_mbedtls.c:1994] Successfully closed PKCS #11 session.
[INFO] [PKCS11] [core_pkcs11_mbedtls.c:1497] PKCS #11 was successfully uninitialized.
[INFO] [PKCS11_OBJECTS_DEMO] [pkcs11_demo_objects.c:159] Finished PKCS #11 Objects Demo.
```

```
/**
 * @brief This function details how to use the PKCS #11 "Sign and Verify" functions to
 * create and interact with digital signatures.
 * The functions described are all defined in
 * https://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html
 * Please consult the standard for more information regarding these functions.
 *
 * The standard has grouped the functions presented in this demo as:
 * Object Management Functions
 * Signing and MACing Functions
 */
CK_RV PKCS11SignVerifyDemo( void )

 /* Signing variables. */
    /* The ECDSA mechanism will be used to sign the message digest. */
    CK_MECHANISM mechanism = { CKM_ECDSA, NULL, 0 };

/* Signing variables. */
    /* The ECDSA mechanism will be used to sign the message digest. */
    CK_MECHANISM mechanism = { CKM_ECDSA, NULL, 0 };
    /* Initializes the sign operation and sets what mechanism will be used
     * for signing the message digest. Specify what object handle to use for this
     * operation, in this case the private key object handle. */
    if( result == CKR_OK )
    {
        LogInfo( ( "Signing known message: %s",
                   ( char * ) knownMessage ) );

        result = functionList->C_SignInit( session,
                                           &mechanism,
                                           privateKeyHandle );
    }
```

```
goks@goks-virtual-machine:~/Desktop/aws-iot-device-sdk-embedded-C/build/bin$ ./pkcs11_demo_mechanisms_and_digests
[INFO] [PKCS11_MECH_AND_DIGEST_DEMO] [pkcs11_demo_mechanisms_and_digests.c:65] Starting PKCS #11 Mechanisms and Digest Demo.
[INFO] [PKCS11] [core_pkcs11_mbedtls.c:1452] PKCS #11 successfully initialized.
[INFO] [PKCS11_MECH_AND_DIGEST_DEMO] [pkcs11_demo_mechanisms_and_digests.c:130] This Cryptoki library supports signing messages with RSA private keys.
[INFO] [PKCS11_MECH_AND_DIGEST_DEMO] [pkcs11_demo_mechanisms_and_digests.c:155] This Cryptoki library supports verifying messages with RSA public keys.
[INFO] [PKCS11_MECH_AND_DIGEST_DEMO] [pkcs11_demo_mechanisms_and_digests.c:174] This Cryptoki library supports signing messages with ECDSA private keys.
[INFO] [PKCS11_MECH_AND_DIGEST_DEMO] [pkcs11_demo_mechanisms_and_digests.c:185] This Cryptoki library supports verifying messages with ECDSA public keys.
[INFO] [PKCS11_MECH_AND_DIGEST_DEMO] [pkcs11_demo_mechanisms_and_digests.c:204] The Cryptoki library supports the SHA-256 algorithm.
[INFO] [PKCS11_MECH_AND_DIGEST_DEMO] [pkcs11_demo_mechanisms_and_digests.c:252] Known message: Hello world!
[INFO] [PKCS11_MECH_AND_DIGEST_DEMO] [pkcs11_demo_mechanisms_and_digests.c:261] Hash of known message using SHA256: c0535e4be2b79ffd9329135436bf889314e4a3faec05ecffcbb7df31ad9e51a.
[INFO] [PKCS11_MECH_AND_DIGEST_DEMO] [pkcs11_demo_mechanisms_and_digests.c:262] Finished PKCS #11 Mechanisms and Digest Demo.
[INFO] [PKCS11] [core_pkcs11_mbedtls.c:1994] Successfully closed PKCS #11 session.
[INFO] [PKCS11] [core_pkcs11_mbedtls.c:1497] PKCS #11 was successfully uninitialized.
```

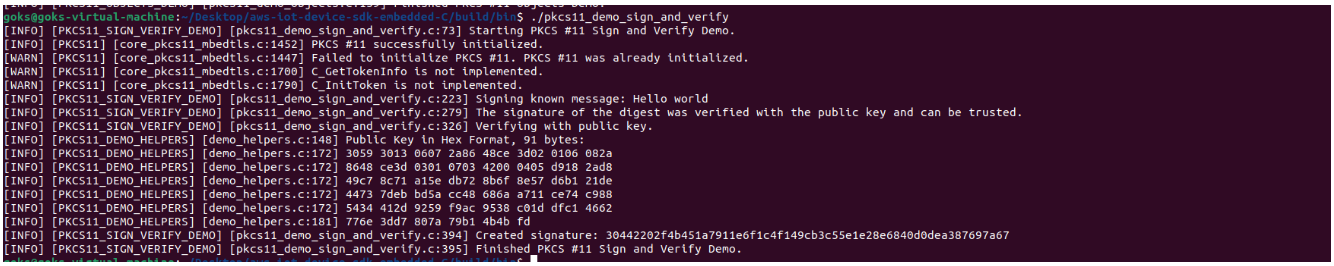
```
/******************************** Verify *********************************/

/* Verify the signature created by C_Sign. First we will verify that the
 * same Cryptoki library was able to trust itself.
 *
 * C_VerifyInit will begin the verify operation, by specifying what mechanism
 * to use (CKM_ECDSA, the same as the sign operation) and then specifying
 * which public key handle to use.
 */
if( result == CKR_OK )
{
    result = functionList->C_VerifyInit( session,
                                         &mechanism,
                                         publicKeyHandle );

************************** ECDSA Capabilities **************************/
if( result == CKR_OK )
{/
    result = functionList->C_GetMechanismInfo( slotId[ 0 ],
                                               CKM_ECDSA,
                                               &MechanismInfo );

    if( 0 != ( CKF_SIGN & MechanismInfo.flags ) )
    {
        LogInfo( ( "This Cryptoki library supports signing messages with"
                   " ECDSA private keys." ) );
    }
    else
    {
        LogInfo( ( "This Cryptoki library does not support signing messages"
                   " with ECDSA private keys." ) );
    }

    if( 0 != ( CKF_VERIFY & MechanismInfo.flags ) )
    {
        LogInfo( ( "This Cryptoki library supports verifying messages with"
                   " ECDSA public keys." ) );
    }
    else
    {
        LogInfo( ( "This Cryptoki library does not support verifying"
                   " messages with ECDSA public keys." ) );
    }
}
```




```
## Public Key Cryptography Standards Which specifies an API, called Cryptoki

```http
API
```

Parameter	Description
`Cryptoki`	API to sign messages;
'PKCS #11`	Signing And Verifying A Signature;
```

````
```javascript
/* Enable the following cipher modes. */
#define MBEDTLS_CIPHER_MODE_CBC
#define MBEDTLS_CIPHER_MODE_CFB
#define MBEDTLS_CIPHER_MODE_CTR

/* Enable the following cipher padding modes. */
#define MBEDTLS_CIPHER_PADDING_PKCS7
#define MBEDTLS_CIPHER_PADDING_ONE_AND_ZEROS
#define MBEDTLS_CIPHER_PADDING_ZEROS_AND_LEN
#define MBEDTLS_CIPHER_PADDING_ZEROS

/* Cipher suite configuration. */
#define MBEDTLS_REMOVE_ARC4_CIPHERSUITES
#define MBEDTLS_ECP_DP_SECP256R1_ENABLED
#define MBEDTLS_ECP_NIST_OPTIM
#define MBEDTLS_KEY_EXCHANGE_ECDHE_RSA_ENABLED
#define MBEDTLS_KEY_EXCHANGE_ECDHE_ECDSA_ENABLED
````

```
 ## Encryption and Decryption
SHA256 Mechanism

/**
 * @brief Length in bytes of hex encoded hash digest.
 */
#define HEX_ENCODED_SHA256_HASH_DIGEST_LENGTH         ( ( ( uint16_t ) 64 ) )

/**
 * @brief Length in bytes of SHA256 hash digest.
 */
#define SHA256_HASH_DIGEST_LENGTH                     ( HEX_ENCODED_SHA256_HASH_DIGEST_LENGTH / 2 )

/**
 * @brief Maximum of all the block sizes of hashing algorithms used in the demo for the
 * calculation of hash digest.
 *
 * @note SHA256 hashing Algorithm is used in the demo for calculating the
 * hash digest and maximum block size for this is 64U.
 */
#define SIGV4_HASH_MAX_BLOCK_LENGTH       64U
/**
 * @brief CryptoInterface provided to SigV4 library for generating the hash digest.
 */
static SigV4CryptoInterface_t cryptoInterface =
{
    .hashInit      = sha256Init,
    .hashUpdate    = sha256Update,
    .hashFinal     = sha256Final,
    .pHashContext  = &hashContext,
    .hashBlockLen  = HEX_ENCODED_SHA256_HASH_DIGEST_LENGTH,
    .hashDigestLen = SHA256_HASH_DIGEST_LENGTH,
};

    /********************************* Verify **********************************/

    /* Verify the signature created by C_Sign. First we will verify that the
     * same Cryptoki library was able to trust itself.
     *
     * C_VerifyInit will begin the verify operation, by specifying what mechanism
     * to use (CKM_ECDSA, the same as the sign operation) and then specifying
     * which public key handle to use.
     */
    if( result == CKR_OK )
    {
        result = functionList->C_VerifyInit( session,
                                             &mechanism,
                                             publicKeyHandle );
    }
```
```

```
}

/* Given the signature and it's length, the Cryptoki will use the public key
 * to verify that the signature was created by the corresponding private key.
 * If C_Verify returns CKR_OK, it means that the sender of the message has
 * the same private key as the private key that was used to generate the
 * public key, and we can trust that the message we received was from that
 * sender.
 *
 * Note that we are not using the actual message, but the digest that we
 * created earlier of the message, for the verification.
 */
if( result == CKR_OK )
{
    result = functionList->C_Verify( session,
                                     digestResult,
                                     pkcs11SHA256_DIGEST_LENGTH,
                                     signature,
                                     signatureLength );

    if( result == CKR_OK )
    {
        LogInfo( ( "The signature of the digest was verified with the"
                   " public key and can be trusted." ) );
    }
    else
    {
        LogInfo( ( "Unable to verify the signature with the given public"
                   " key, the message cannot be trusted." ) );
    }
}

/* Export public key as hex bytes and print the hex representation of the
 * public key.
 *
 * We need to export the public key so that it can be used by a different
 * device to verify messages signed by the private key of the device that
 * generated the key pair.
 *
 * To do this, we will output the hex representation of the public key.
 * Then create an empty text file called "DevicePublicKeyAsciiHex.txt".
 *
 * Copy and paste the hex value of the public key into this text file.
 *
 * Then we will need to convert the text file to binary using the xxd tool.
 *
 * xxd will take a text file that contains hex data and output a binary of
 * the hex in the file. See "$ man xxd" for more information about xxd.
 *
 * Copy the below command into the terminal.
 * "$ xxd -r -ps DevicePublicKeyAsciiHex.txt DevicePublicKeyDer.bin"
 *
 * Now that we have the binary encoding of the public key, we will convert
 * it to PEM using OpenSSL.
 *
 * The following command will create a PEM file of the public key called
 * "public_key.pem"
 *
 * "$ openssl ec -inform der -in DevicePublicKeyDer.bin -pubin -pubout -outform pem -out public_key.pem"
 *
 * Now we can use the extracted public key to verify the signature of the
 * device's private key.
 *
 * WARNING: Running the object generation demo will create a new key pair,
 * and make it necessary to redo these steps!
 *
 */
if( result == CKR_OK )
{
    LogInfo( ( "Verifying with public key." ) );
```

```
        LogInfo( ( "Verifying with public key:" ) );
        exportPublicKey( session,
                         publicKeyHandle,
                         &derPublicKey,
                         &derPublicKeyLength );
        writeHexBytesToConsole( "Public Key in Hex Format",
                                derPublicKey,
                                derPublicKeyLength );

        /* exportPublicKey allocates memory which needs to be freed. */
        if( derPublicKey != NULL )
        {
            free( derPublicKey );
        }
    }

    /* Set TLS MFLN if requested. */
    if( pOpensslCredentials->maxFragmentLength > 0U )
    {
        LogDebug( ( "Setting max send fragment length %u.",
                    pOpensslCredentials->maxFragmentLength ) );

        /* Set the maximum send fragment length. */

        /* MISRA Directive 4.6 flags the following line for using basic
         * numerical type long. This directive is suppressed because openssl
         * function #SSL_set_max_send_fragment expects a length argument
         * type of long. */
        /* coverity[misra_c_2012_directive_4_6_violation] */
        sslStatus = ( int32_t ) SSL_set_max_send_fragment(
            pSsl, ( long ) pOpensslCredentials->maxFragmentLength );

        if( sslStatus != 1 )
        {
            LogError( ( "Failed to set max send fragment length %u.",
                        pOpensslCredentials->maxFragmentLength ) );
        }
        else
        {
            readBufferLength = ( int16_t ) pOpensslCredentials->maxFragmentLength +
                               SSL3_RT_MAX_ENCRYPTED_OVERHEAD;

            /* Change the size of the read buffer to match the
             * maximum fragment length + some extra bytes for overhead. */
            SSL_set_default_read_buffer_len( pSsl, ( size_t ) readBufferLength );
        }
    }
```

```
/**
 * @brief Path of the file containing the server's root CA certificate.
 *
 * This certificate is used to identify the AWS IoT server and is publicly
 * available. Refer to the AWS documentation available in the link below
 * https://docs.aws.amazon.com/iot/latest/developerguide/server-authentication.html#server-authentication-certs
 *
 * Amazon's root CA certificate is automatically downloaded to the certificates
 * directory from @ref https://www.amazontrust.com/repository/AmazonRootCA1.pem
 * using the CMake build system.
 *
 * @note This certificate should be PEM-encoded.
 * @note This path is relative from the demo binary created. Update
 * ROOT_CA_CERT_PATH to the absolute path if this demo is executed from elsewhere.
 */
#ifndef ROOT_CA_CERT_PATH
    #define ROOT_CA_CERT_PATH    "certificates/AmazonRootCA1.crt"
#endif
```