Microsoft

Azure RTOS NetX Secure ECC User Guide

Published: February 2020

For the latest information, please see azure.com/rtos

Part Number: 000-1059

Revision 6.0

Use Elliptic Curve Cryptography (ECC) in NetX Secure

Installation and Use of ECC

This chapter contains information related to installation, setup, and use of ECC crypto algorithms in NetX Secure TLS.

Product Distribution

In NetX Secure, the ECC-crypto is distributed as a separate package. To install ECC files, the user shall unzip the files into the NetX Secure source code directory.

The ECC-related files contain the keyword "ecc" in the file names. The following files are added to provide ECC functionality.

nx_secure_tls_ecc.h	Header file for NetX Secure TLS for ECC
nx_secure_tls_ecc_*.c	C Source files for NetX Secure TLS for ECC
nx_secure_x509_ecc_*.c	C Source files for X.509 digital certificates.

Supported ECC curves

NetX Secure implements parts of the curves as per http://www.secg.org/sec2-v2.pdf. The following curves are supported:

- secp192r1
- secp224r1
- secp256r1
- secp384r1
- secp521r1

If other ECC curves are used, the *`nx_secure_tls_session_start()`* routine returns `NX_SECURE_TLS_NO_SUPPORTED_CIPHERS` for unsupported curves.

Note that TLS certificate chain may be encrypted by ECC-algorithms as well. Even though the curves provided by the TLS Client are supported, it is possible that the ECC curve used in the certificate chain is not supported. In this case, `nx_secure_tls_session_start` routine returns NX_SECURE_TLS_UNSUPPORTED_PUBLIC_CIPHER.

Crypto Methods for ECC

Crypto methods for Elliptic Curve groups:

- `NX_CRYPTO_METHOD crypto_method_ec_secp192;`
- `NX_CRYPTO_METHOD crypto_method_ec_secp224;`
- `NX_CRYPTO_METHOD crypto_method_ec_secp256;`
- `NX_CRYPTO_METHOD crypto_method_ec_secp384;`
- `NX_CRYPTO_METHOD crypto_method_ec_secp521;`

The crypto methods for ECC curves are defined in nx_crypto_generic_ciphersuites.c

Crypto methods for ECDH and ECDHE:

- `NX_CRYPTO_METHOD crypto_method_ecdh;`
- `NX_CRYPTO_METHOD crypto_method_ecdhe;`

Crypto method for ECDSA:

- `NX_CRYPTO_METHOD crypto_method_ecdsa;`

ECDH, ECDSA and ECDHE crypto methods are defined in *nx_crypto_generic_ciphersuites.c. .*

Combined with other crypto methods such as RSA, SHA, AES, they can be used as building blocks for the ciphersuite lookup table.

Enable ECC in NetX Secure

By default ECC is enabled by default. To disable ECC support, the symbol *NX_SECURE_DISABLE_ECC_CIPHERSUITE* must be defined.

For the change to take effect, user shall rebuild NetX Secure Library, and all applications that use the library.

In the application code, the API n*x_secure_tls_ecc_initialize()* must be called after TLS session is created. This API notifies the TLS session of the type of curves used in the system. During the TLS handshake phase, if ECC algorithm is selected, the client and server exchange ECC curve-related parameters, so ECC can be used for the session.

The following code segment illustrates how to use the API. Note that the arguments (*nx_crypto_ecc_supported_groups, nx_crypto_ecc_supported_groups_size, and nx_crypto_ecc_curves)* are all defined in *nx_crypto_generic_ciphersuites.c*. Therefore these symbols can be used directly.

```
status = nx_secure_tls_ecc_initialize(&tls_session,
                    nx_crypto_ecc_supported_groups,
                    nx_crypto_ecc_supported_groups_size,
                    nx_crypto_ecc_curves);
```

The TLS Crypto table in nx_crypto_generic_ciphersuites_ecc.c contains ECC-class ciphersuites lookup table. Thereforfe TLS session wishing to use ECC shall use `nx_crypto_tls_ciphers.c` when creating TLS Sessions. The TLS_Crypto table defined in nx_crypto_generic_ciphersuites.c contains both ECC and non-ECC ciphersuites.

Known Limitations

The following known limitations are in the scope of ECC for TLS.

- SHA384 or SHA512 are not supported except for verification of certificate signature.
- In ServerKeyExchange of the ECDHE ciphersuites, only SHA1 and SHA256 are supported for the signature hash.
- TLS server does not support dynamic certificate selection when there are multiple certificates in the local store.
- X509 Certificate KeyUsage is not observed.
- ECDSA_fixed_ECDH, RSA_fixed_ECDH or ECDH_anon are not supported.

Configuration Options

There are several configuration options for building NetX Secure. Following is a list of all options, where each is described in detail:

Define	Meaning
NX_SECURE_ENABLE_ECC_CIPHERSUITE	Defined, this option enables the ECC support in TLS. By default ECC support is enabled but this symbol is not explicitly defined.
NX_SECURE_DISABLE_ECC_CIPHERSUITE	Defined, this option disables the ECC support in TLS. By default this symbol is not defined.

TLS Client Example

The following example uses ECC for a TLS Client application. This example should work with the OpenSSL reverse-echo server (openssl s_server -rev).

Note: This example is provided for demonstration purposes and will not compile as is. The HTTP message and certificate data are not supplied and instead represented by an ellipsis (…)for the sake of brevity.
In addition, when code is transferred to a Windows document, it cannot be compiled. Since much formatting is required to fit the code and comments into the margins of this document, small errors can slip in.

The status returns from the API calls below are not checked and the API is assumed to complete successfully, also for the sake of brevity. However, in your TLS applications, you should *always* check status returns on NetX Secure service calls.

Please refer to the NetX secure demo files for a project you can build and run in your own environment.

```
#include "tx_api.h"
#include "nx_api.h"
#include "nx_secure_tls_api.h"

/* Define the size of our application stack. */
#define     DEMO_STACK_SIZE             4096

/* Define the remote server IP address using NetX IP_ADDRESS macro. */
#define     REMOTE_SERVER_IP_ADDRESS     IP_ADDRESS(192, 168, 1, 1)

/* Define the remote server port. 443 is the HTTPS default. */
#define     REMOTE_SERVER_PORT          443

/* Define the ThreadX and NetX object control blocks...  */

NX_PACKET_POOL          pool_0;
NX_IP                   ip_0;
NX_TCP_SOCKET tcp_socket;
NX_SECURE_TLS_SESSION tls_session;
NX_SECURE_X509_CERTIFICATE tls_certificate;

/* Define space for remote certificate storage. There must be one certificate
structure and its associated buffer for each expected certificate from the remote
host. If you expect 3 certificates, you will need 3 structures and 3 buffers. The
buffers must be large enough to hold the incoming certificate data (2KB is usually
sufficient but large RSA keys can push the size beyond that). */
NX_SECURE_X509_CERTIFICATE remote_certificate;
NX_SECURE_X509_CERTIFICATE remote_issuer_certificate;
UCHAR remote_certificate_buffer[2000];
UCHAR remote_issuer_buffer[2000];

/* Define an HTTP request to be sent to the HTTPS web server not defined here but
  represented by the ellipsis. */
UCHAR http_request[] = { … };

/* Define the IP thread's stack area.  */
ULONG ip_thread_stack[3 * 1024 / sizeof(ULONG)];
```

```
/* Define packet pool for the demonstration.  */
#define NX_PACKET_POOL_SIZE ((1536 + sizeof(NX_PACKET)) * 32)

ULONG packet_pool_area[NX_PACKET_POOL_SIZE/sizeof(ULONG) + 64 / sizeof(ULONG)];

/* Define the ARP cache area.  */
ULONG arp_space_area[512 / sizeof(ULONG)];

/* Define the TLS Client thread.  */
ULONG            tls_client_thread_stack[6 * 1024 / sizeof(ULONG)];
TX_THREAD        tls_client_thread;
void             client_thread_entry(ULONG thread_input);

/* Define the TLS packet reassembly buffer. */
UCHAR tls_packet_buffer[4000];

/* Define the metadata area for TLS cryptography. The actual size needed can be
   Ascertained by calling nx_secure_tls_metadata_size_calculate.
*/
UCHAR tls_crypto_metadata[18000];

/* Pointer to the TLS ciphersuite table that is included in the platform-specific
   cryptography subdirectory. The table maps the cryptographic routines for the
   platform to function pointers usable by the TLS library.
*/
extern const NX_SECURE_TLS_CRYPTO nx_crypto_tls_ciphers_ecc;
extern const USHORT nx_crypto_ecc_supported_groups[];
extern const NX_CRYPTO_METHOD *nx_crypto_ecc_curves[];
extern const UINT nx_crypto_ecc_supported_groups_size;

/* Binary data for the TLS Client X.509 trusted root CA certificate, ASN.1 DER-
   encoded. A trusted certificate must be provided for TLS Client applications
   (unless X.509 authentication is disabled) or TLS will treat all certificates as
   untrusted and the handshake will fail.
*/

/* DER-encoded binary certificate, not defined here but represented by the ellipsis,
   for the sake of brevity. */
const UCHAR trusted_ca_data[] = { … };
const UINT trusted_ca_length[] = 0x574;

/* Define the application - initialize drivers and TCP/IP setup.  */
void    tx_application_define(void *first_unused_memory)
{
UINT  status;

    /* Initialize the NetX system.  */
    nx_system_initialize();

    /* Create a packet pool. Check status for errors. */
    status =  nx_packet_pool_create(&pool_0, "NetX Main Packet Pool", 1536,
                                    (ULONG*)(((int)packet_pool_area + 64) & ~63) ,
                                    NX_PACKET_POOL_SIZE);

    /* Create an IP instance for the specific target. Check status for errors. This
       call is not completely defined. Please see other demo files for proper usage
       of the nx_ip_create call. */
    status = nx_ip_create(&ip_0, …);

    /* Enable ARP and supply ARP cache memory for IP Instance 0. Check status for
       errors. */
    status =  nx_arp_enable(&ip_0, (void *)arp_space_area, sizeof(arp_space_area));

    /* Enable TCP traffic. Check status for errors. */
    status =  nx_tcp_enable(&ip_0);

    status =  nx_ip_fragment_enable(&ip_0);

    /* Initialize the NetX Secure TLS system.  */
    nx_secure_tls_initialize();
    /* Create the TLS client thread to start handling incoming requests. */
    tx_thread_create(&tls_client_thread, "TLS Server thread", client_thread_entry, 0,
                     tls_client_thread_stack, sizeof(tls_client_thread_stack),
                     16, 16, 4, TX_AUTO_START);
    return;
}

/* Thread to handle the TLS Client instance. */
void client_thread_entry(ULONG thread_input)
{
UINT       status;
NX_PACKET *send_packet;
NX_PACKET *receive_packet;
UCHAR receive_buffer[100];
ULONG bytes;
ULONG server_ipv4_address;

    /* We are not using the thread input parameter so suppress compiler warning. */
    NX_PARAMETER_NOT_USED(thread_input);

    /* Ensure the IP instance has been initialized.  */
    status =  nx_ip_status_check(&ip_0, NX_IP_INITIALIZE_DONE, &actual_status,
                                 NX_IP_PERIODIC_RATE);

    /* Create a TCP socket to use for our TLS session.  */
    status =  nx_tcp_socket_create(&ip_0, &tcp_socket, "TLS Client Socket",
                                   NX_IP_NORMAL, NX_FRAGMENT_OKAY,
                                   NX_IP_TIME_TO_LIVE,
                                   8192, NX_NULL, NX_NULL);

    /* Create a TLS session for our socket. This sets up the TLS session object for
      later use */
    status =  nx_secure_tls_session_create(&tls_session,
                                           &nx_crypto_tls_ciphers,
                                           tls_crypto_metadata,
                                           sizeof(tls_crypto_metadata));

    /* Initialize ECC parameters for this session. */
    status = nx_secure_tls_ecc_initialize(&tls_session,
                                         nx_crypto_ecc_supported_groups,
                                         nx_crypto_ecc_supported_groups_size,
                                         nx_crypto_ecc_curves);

    /* Set the packet reassembly buffer for this TLS session. */
    status = nx_secure_tls_session_packet_buffer_set(&tls_session, tls_packet_buffer,
                                                     sizeof(tls_packet_buffer));

    /* Initialize an X.509 certificate with our CA root certificate data. */
    nx_secure_x509_certificate_initialize(&certificate, trusted_ca_data,
                                         trusted_ca_length, NX_NULL, 0, NX_NULL, 0,
                                         NX_SECURE_X509_KEY_TYPE_NONE);

    /* Add the initialized certificate as a trusted root certificate. */
    nx_secure_tls_trusted_certificate_add(&tls_session, &certificate);

    /* The remote server will be sending one or more certificates so we need to
       allocate space to receive and process them. Assume the server will provide at
       least an identity certificate and an intermediate CA issuer. */
    nx_secure_tls_remote_certificate_allocate(&tls_session, &remote_certificate,
      remote_certificate_buffer,
      sizeof(remote_certificate_buffer));
    nx_secure_tls_remote_certificate_allocate(&tls_session,
                                              &remote_issuer_certificate,
                                              remote_issuer_buffer,
                                              sizeof(remote_issuer_buffer));

    /* Setup this thread to open a connection on the TCP socket to a remote server.
       The IP address can be used directly or it can be obtained via DNS or other
       means.*/
    server_ipv4_address = REMOTE_SERVER_IP_ADDRESS;
    status = nx_tcp_client_socket_connect(&tcp_socket, server_ipv4_address,
                                          REMOTE_SERVER_PORT, NX_WAIT_FOREVER);

    /* Start the TLS Session using the connected TCP socket. This function will
       ascertain from the TCP socket state that this is a TLS Client session. */
    status = nx_secure_tls_session_start(&tls_session, &tcp_socket,
                                         NX_WAIT_FOREVER);

    /* Allocate a TLS packet to send an HTTP request over TLS (HTTPS). */
    status = nx_secure_tls_packet_allocate(&tls_session, &pool_0, &send_packet,
                                           NX_TLS_PACKET, NX_WAIT_FOREVER);

    /* Populate the packet with our HTTP request. */
    nx_packet_data_append(send_packet, http_request, strlen(http_request), &pool_0,
                          NX_WAIT_FOREVER);


    /* Send the HTTP request over the TLS Session, turning it into HTTPS. */
    status = nx_secure_tls_session_send(&tls_session, send_packet, NX_WAIT_FOREVER);

    /* If the send fails, you must release the packet.  */
    if (status != NX_SUCCESS)
    {
          /* Release the packet since the packet was not sent.  */
          nx_packet_release(send_packet);
    }

    /* Receive the HTTP response and any data from the server. */
    status = nx_secure_tls_session_receive(&tls_session, &receive_packet,
                                           NX_WAIT_FOREVER);
    if (status == NX_SUCCESS)
    {
         /* Extract the data we received from the remote server. */
           status = nx_packet_data_extract_offset(receive_packet, 0, receive_buffer,
                    100,  &bytes);
           /* Display the response data. */
           receive_buffer[bytes] = 0;
           printf("Received data: %s\n", receive_buffer);

           /* Release the packet when done with it. */
          nx_packet_release(receive_packet);
    }

    /* End the TLS session now that we have received our HTTPS/HTML response. */
    status = nx_secure_tls_session_end(&tls_session, NX_WAIT_FOREVER);

    /* Check for errors to make sure the session ended cleanly. */

    /* Disconnect the TCP socket. */
    status =  nx_tcp_socket_disconnect(&tcp_socket, NX_WAIT_FOREVER);

}
```

Figure 1.1 Example of using ECC for TLS Client Application

TLS Server Example (HTTPS Web Server)

The following example uses ECC for TLS server application. This example demonostrates a simple TLS Web Server (HTTPS). For simplicity, in this example API calls are assumed to be successful, and return values are not checked.

As with the TLS Client example, this is intended for demonstration purposes and should not be expected to compile and run as is in your environment. This example does not check status returns for the sake of brevity, it is strongly recommended that your application should do so.

```
#include "tx_api.h"
#include "nx_api.h"
#include "nx_secure_tls_api.h"

#define     DEMO_STACK_SIZE         4096

/* Define the ThreadX and NetX object control blocks...  */

NX_PACKET_POOL          pool_0;
NX_IP                   ip_0;
NX_TCP_SOCKET tcp_socket;
NX_SECURE_TLS_SESSION tls_session;
NX_SECURE_X509_CERTIFICATE tls_certificate;

/* Define the IP thread's stack area.  */
ULONG ip_thread_stack[3 * 1024 / sizeof(ULONG)];

/* Define packet pool for the demonstration.  */
#define NX_PACKET_POOL_SIZE ((1536 + sizeof(NX_PACKET)) * 32)

ULONG packet_pool_area[NX_PACKET_POOL_SIZE/sizeof(ULONG) + 64 / sizeof(ULONG)];

/* Define the ARP cache area.  */
ULONG arp_space_area[512 / sizeof(ULONG)];


/* Define the TLS Server thread.  */
ULONG             tls_server_thread_stack[6 * 1024 / sizeof(ULONG)];
TX_THREAD         tls_server_thread;
void              server_thread_entry(ULONG thread_input);

/* Define the TLS packet reassembly buffer. */
UCHAR tls_packet_buffer[4000];

/* Define the metadata area for TLS cryptography. The actual size needed can be
   Ascertained by calling nx_secure_tls_metadata_size_calculate.
*/
UCHAR tls_crypto_metadata[18000];

/* Pointer to the TLS ciphersuite table that is included in the platform-specific
   cryptography subdirectory. The table maps the cryptographic routines for the
   platform to function pointers usable by the TLS library.
*/
extern const NX_SECURE_TLS_CRYPTO nx_crypto_tls_ciphers_ecc;
extern const USHORT nx_crypto_ecc_supported_groups[];
extern const NX_CRYPTO_METHOD *nx_crypto_ecc_curves[];
extern const UINT nx_crypto_ecc_supported_groups_size;

/* Binary data for the TLS Server X.509 certificate, ASN.1 DER-encoded. Note that the
   certificate data and private key data is represented by an ellipsis for the sake
   of brevity.
*/
const UCHAR certificate_data[] = { … }; /* DER-encoded binary certificate. */
const UINT certificate_length[] = 0x574;

/* Binary data for the TLS Server RSA Private Key, from private key
   file generated at the time of the X.509 certificate creation. ASN.1 DER-encoded.
*/
const UCHAR private_key[] = { … }; /* DER-encoded RSA private key file (PKCS#1) */
const UINT private_key_length = 0x40;

/* Define some HTML data (web page) with an HTTPS header to serve to connecting
   clients. Here we use an ellipsis instead of actual web page data. */
const UCHAR html_data[] = { … };
```

```
/* Define the application - initialize drivers and TCP/IP setup.  */
void    tx_application_define(void *first_unused_memory)
{
UINT  status;


    /* Initialize the NetX system.  */
    nx_system_initialize();

    /* Create a packet pool. Check status for errors. */
    status =  nx_packet_pool_create(&pool_0, "NetX Main Packet Pool", 1536,
                                      (ULONG*)(((int)packet_pool_area + 64) & ~63) ,
                                      NX_PACKET_POOL_SIZE);

    /* Create an IP instance for the specific target. Check status for errors. */
    status = nx_ip_create(&ip_0, …);

    /* Enable ARP and supply ARP cache memory for IP Instance 0. Check status for
       errors. */
    status =  nx_arp_enable(&ip_0, (void *)arp_space_area, sizeof(arp_space_area));

    /* Enable TCP traffic. Check status for errors. */
    status =  nx_tcp_enable(&ip_0);

    status =  nx_ip_fragment_enable(&ip_0);

    /* Initialize the NetX Secure TLS system.  */
    nx_secure_tls_initialize();

    /* Create the TLS server thread to start handling incoming requests. */
    tx_thread_create(&tls_server_thread, "TLS Server thread", server_thread_entry, 0,
                     tls_server_thread_stack, sizeof(tls_server_thread_stack),
                     16, 16, 4, TX_AUTO_START);
    return;
}

 /* Thread to handle the TLS Server instance. */
void server_thread_entry(ULONG thread_input)
{
UINT       status;
NX_PACKET *send_packet;
NX_PACKET *receive_packet;
UCHAR receive_buffer[100];
ULONG bytes;

    NX_PARAMETER_NOT_USED(thread_input);

    /* Ensure the IP instance has been initialized.  */
    status =  nx_ip_status_check(&ip_0, NX_IP_INITIALIZE_DONE, &actual_status,
                                 NX_IP_PERIODIC_RATE);

    /* Create a TCP socket to use for our TLS session.  */
    status =  nx_tcp_socket_create(&ip_0, &tcp_socket, "TLS Server Socket",
                                   NX_IP_NORMAL, NX_FRAGMENT_OKAY,
                                   NX_IP_TIME_TO_LIVE,
                                   8192, NX_NULL, NX_NULL);

    /* Create a TLS session for our socket.  */
    status =  nx_secure_tls_session_create(&tls_session,
                                           &nx_crypto_tls_ciphers,
                                           tls_crypto_metadata,
                                           sizeof(tls_crypto_metadata));

   status = nx_secure_tls_ecc_initialize(&tls_session,
                                         nx_crypto_ecc_supported_groups,
                                         nx_crypto_ecc_supported_groups_size,
                                         nx_crypto_ecc_curves);

    /* Set the packet reassembly buffer for this TLS session. */
    status = nx_secure_tls_session_packet_buffer_set(&tls_session, tls_packet_buffer,
                                                     sizeof(tls_packet_buffer));

    /* Initialize an X.509 certificate and private RSA key for our TLS Session. */
    nx_secure_x509_certificate_initialize(&certificate, certificate_data, NX_NULL, 0,
                                         certificate_length, private_key,
                                         private_key_length);

    /* Add the initialized certificate as a local identity certificate. */
    nx_secure_tls_add_local_certificate(&tls_session, &certificate);


    /* Setup this thread to listen on the TCP socket.
       Port 443 is standard for HTTPS. */
    status =  nx_tcp_server_socket_listen(&ip_0, 443, &tcp_socket, 5, NX_NULL);

    while(1)
    {
        /* Accept a client TCP socket connection.  */
        status =  nx_tcp_server_socket_accept(&tcp_socket, NX_WAIT_FOREVER);

        /* Start the TLS Session using the connected TCP socket. */
        status = nx_secure_tls_session_start(&tls_session, &tcp_socket,
                                              NX_WAIT_FOREVER);

        /* Receive the HTTPS request. */
        status = nx_secure_tls_session_receive(&tls_session, &receive_packet,
                                                NX_WAIT_FOREVER);


         if (status == NX_SUCCESS)
         {
                /* Extract the HTTP request information from the HTTPS request. */
            status = nx_packet_data_extract_offset(receive_packet, 0, receive_buffer,
                                                    100, &bytes);
            /* Display the HTTP request data. */
             receive_buffer[bytes] = 0;
             printf("Received data: %s\n", receive_buffer);

             /* Release the packet when done with it */
             nx_packet_release(receive_packet);
        }

        /* Allocate a TLS packet to send HTML data back to client. */
        status = nx_secure_tls_packet_allocate(&tls_session, &pool_0, &send_packet,
                                                NX_TLS_PACKET, NX_WAIT_FOREVER);

        /* Populate the packet with our HTTP response and HTML web page data. */
        nx_packet_data_append(send_packet, html_data, strlen(html_data), &pool_0,
                              NX_WAIT_FOREVER);

        /* Send the HTTP response over the TLS Session, turning it into HTTPS. */
        status = nx_secure_tls_session_send(&tls_session, send_packet,
                                             NX_WAIT_FOREVER);

        /* If the send fails, you must release the packet.  */
        if (status != NX_SUCCESS)
        {
              /* Release the packet since it was not sent.  */
              nx_packet_release(send_packet);
        }

        /* End the TLS session now that we have sent our HTTPS/HTML response. */
        status = nx_secure_tls_session_end(&tls_session, NX_WAIT_FOREVER);

        /* Check for errors to make sure the session ended cleanly! */

        /* Disconnect the TCP socket so we can be ready for the next request. */
        status =  nx_tcp_socket_disconnect(&tcp_socket, NX_WAIT_FOREVER);

        /* Unaccept the server socket.  */
        status =  nx_tcp_server_socket_unaccept(&tcp_socket);
```

```
        /* Setup server socket for listening again.  */
        status =  nx_tcp_server_socket_relisten(&ip_0, 443, &tcp_socket);
    }
}
```

Figure 1.2 Example of NetX Secure use with NetX