Microsoft

Azure RTOS
NetX Secure DTLS
User Guide

Published: February 2020

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

Part Number: 000-1059

Revision 6.0

Contents

Chapter 1

Introduction to NetX Secure DTLS

NetX Secure DTLS is a high-performance real-time implementation of of the Datagram Transport Layer Security protocol designed exclusively for embedded ThreadX-based applications. This chapter contains an introduction to NetX Secure DTLS and a description of its applications and benefits.

NetX Secure Unique Features

Unlike most other TLS/DTLS implementations, NetX Secure was designed from the ground up to support a wide variety of embedded hardware platforms and scales easily from small micro-controller applications to the most powerful embedded processors available. The code is written with the limited resources of embedded systems in mind, and provides a number of configuration options to reduce the memory footprint needed to provide secure network communications over TLS or DTLS.

RFCs Supported by NetX Secure

NetX Secure supports the following protocols related to TLS and DTLS. The list is not necessarily comprehensive as there are numerous RFCs pertaining to TLS/DTLS and cryptography. NetX Secure follows all general recommendations and basic requirements within the constraints of a real-time operating system with small memory footprint and efficient execution.

RFC	Description	Page
RFC 6347	Datagram Transport Layer Security Version 1.2.	19
RFC 2246	The TLS Protocol Version 1.0	18
RFC 4346	The Transport Layer Security (TLS) Protocol Version 1.1	18
RFC 5246	The Transport Layer Security (TLS) Protocol Version 1.2	18
RFC 5280	X.509 PKI Certificates (v3)	**Error! Bookmark not defined.**
RFC 3268	Advanced Encryption Standard (AES) Ciphersuites for Transport	**Error! Bookmark not defined.**

	Layer Security (TLS)	
RFC 3447	Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1	**Error! Bookmark not defined.**
RFC 2104	HMAC: Keyed-Hashing for Message Authentication	**Error! Bookmark not defined.**
RFC 6234	US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)	**Error! Bookmark not defined.**
RFC 4279	Pre-Shared Key Ciphersuites for TLS	35

NetX Secure DTLS Requirements

In order to function properly, the NetX Secure run-time library requires that a NetX IP instance has already been created. In addition, and depending on the application, one or more DER-encoded X.509 Digital Certificates will be required, either to identify a TLS/DTLS instance or to verify certificates coming from a remote host. The NetX Secure package has no further requirements.

NetX Secure DTLS Constraints

The NetX Secure DTLS protocol implements the requirements of the RFC 6347 Standard(s) for DTLS 1.2. However, there are the following constraints:

1. Due to the nature of embedded devices, some applications may not have the resources to support the maximum TLS/DTLS record size of 16KB. NetX Secure can handle 16KB records on devices with sufficient resources.
2. Minimal certificate verification. NetX Secure will perform basic X.509 chain verification on a certificate to assure that the certificate is valid and signed by a trusted Certificate Authority, and can provide the certificate Common Name for the application to compare against the Top-Level Domain Name of the remote host. However, verification of certificate extensions and other data is the responsibility of the application implementer.

3. Software-based cryptography is processor-intensive. The NetX Secure software-based cryptographic routines have been optimized for performance, but depending on the power of the target processor, that performance may result in very long operations. When hardware-based cryptography is available, it should be used for optimal performance of NetX Secure DTLS.

Chapter 2

Installation and Use of NetX Secure DTLS

This chapter contains a description of various issues related to installation, setup, and usage of the NetX Secure DTLS component.

Product Distribution

NetX Secure is shipped on a single CD-ROM compatible disk. The package includes source files, include files, and a PDF file that contains this document, as follows:

nx_secure_dtls_api.h	Public API header file for NetX Secure DTLS
nx_secure_dtls_user.h	User defines header file for NetX Secure DTLS
nx_secure_ port.h	Platform-specific definitions for NetX Secure
nx_secure_dtls.h	Header file for NetX Secure DTLS
nx_secure_tls.h	Header file for NetX Secure TLS
nx_secure_dtls*.c/h	C/H Source files for NetX Secure DTLS
nx_secure_tls*.c/h	C/H Source files for NetX Secure TLS
nx_crypto*.c/h	C/H Source files for NetX Secure Cryptography
nx_secure_x509*.c/h	C/H Source files for X.509 digital certificates.
demo_netx_secure_dtls.c	C Source file for NetX Secure DTLS Demo

NetX_Secure_DTLS_User_Guide.pdf
PDF description of NetX Secure product

Note that the nx_crypto* files are provided for different hardware platforms in a subdirectory of the NetX Secure parent directory.

NetX Secure DTLS Installation

In order to use NetX Secure DTLS, the entire distribution mentioned previously should be copied to the same directory level where NetX is installed. For example, if NetX is installed in the directory "*threadx**arm7**NetX*" then the *nx_secure*.** directories should be copied into "*threadx**arm7**NetXSecure*".

Using NetX Secure DLTS

Using NetX Secure DTLS is straightforward. The application code must include *nx_secure_dtls_api.h* after it includes *tx_api.h* and *nx_api.h* (which are for ThreadX and NetX, respectively). Once

nx_secure_dtls_api.h is included, the application code is then able to make the NetX Secure DTLS function calls specified later in this guide. The application must also import the *nx_secure*.** files into a NetXSecure library, and the platform-specific *nx_crypto*.** files into a NetXCrypto library which are then linked into the final application binary.

Small Example System (DTLS Client)

An example of how easy it is to use NetX Secure DTLS is described in Figure 1.1, which appears below and demonstrates a simple DTLS Client, designed to work with an OpenSSL (or similar) DTLS server. Note that the DTLS client program structure is very similar to a NetX Secure TLS Client (see NetX Secure TLS documentation). This is because the DTLS protocol is essentially a version of TLS for use over unreliable transport network protocols such as UDP.

```
#include "tx_api.h"
#include "nx_api.h"
#include "nx_secure_dtls_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. */
#define     REMOTE_SERVER_PORT          4443

/* Define the size of the buffer used for incoming certificates. The
   Buffer will contain both the raw certificate data and an instance
   of the NX_SECURE_X509_CERT structure used for X.509 parsing. */
#define     REMOTE_CERT_BUFFER_SIZE     (sizeof(NX_SECURE_X509_CERT) + 2000)

/* Define the number of certificates we expect to receive from the server
   so we can allocate enough space for them. */
#define     REMOTE_CERT_NUMBER          2

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

NX_PACKET_POOL          pool_0;
NX_IP                   ip_0;
NX_UDP_SOCKET udp_socket;
NX_SECURE_DTLS_SESSION dtls_session;
NX_SECURE_X509_CERTIFICATE dtls_certificate;

/* Define space for remote certificate storage. The size of the buffer is determined
  by the expected number of certificates times the expected size of each certificate.
*/
UCHAR remote_certificate_buffer[REMOTE_CERT_BUFFER_SIZE * REMOTE_CERT_NUMBER];

/* Define some data to send to the DTLS server. */
UCHAR request_data[] = { … };

/* 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 DTLS Client thread.  */
ULONG             dtls_client_thread_stack[6 * 1024 / sizeof(ULONG)];
TX_THREAD         dtls_client_thread;
void              client_thread_entry(ULONG thread_input);

/* Define the DTLS packet reassembly buffer. */
UCHAR dtls_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 dtls_crypto_metadata[14000];

/* Pointer to the TLS/DTLS 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 DTLS library.
*/
extern const NX_SECURE_TLS_CRYPTO nx_crypto_tls_ciphers;

/* Binary data for the DTLS Client X.509 trusted root CA certificate: ASN.1 DER-
   encoded. A trusted certificate must be provided for TLS/DTLS Client applications
   (unless an alternate authentication mechanism is used, such as PSK) or DTLS will
   treat all certificates as untrusted and the handshake will fail.
*/
const UCHAR trusted_ca_data[] = { … }; /* DER-encoded binary certificate. */
const UINT trusted_ca_length[] = 0x574;

/* Define the application – initialize drivers and UDP 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 UDP traffic. Check status for errors. */
    status =  nx_udp_enable(&ip_0);

    status =  nx_ip_fragment_enable(&ip_0);

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

    /* Create the client thread to start handling incoming requests. */
    tx_thread_create(&dtls_client_thread, "DTLS Client thread", client_thread_entry,
                     0, dtls_client_thread_stack, sizeof(dtls_client_thread_stack),16,
                     16, 4, TX_AUTO_START);
}

 /* Thread to handle the DTLS 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);

    /* Check status for errors... */

    /* Create a UDP socket to use for our DTLS session.  */
    status =  nx_udp_socket_create(&ip_0, &udp_socket, "DTLS Client Socket",
                                   NX_IP_NORMAL, NX_FRAGMENT_OKAY, NX_IP_TIME_TO_LIVE,
                                   8192);

    /* Check status for errors... */

    /* Create a DTLS session for our socket. This sets up the DTLS session object for
       later use with encryption, packet buffer space for decryption, and buffer
       space for incoming server X.509 certificates. */
    status =  nx_secure_dtls_session_create(&dtls_session,
                                            &nx_crypto_tls_ciphers,
                                            tls_crypto_metadata,
                                            sizeof(tls_crypto_metadata),
                                            dtls_packet_buffer,
                                            sizeof(dtls_packet_buffer),
                                            REMOTE_CERT_NUMBER,
                                            remote_certificate_buffer,
                                            sizeof(remote_certificate_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_dtls_session_trusted_certificate_add(&dtls_session, &certificate);

    /* Setup this thread to open a connection on the UDP 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;

   /* Check for errors…  */

    /* Start the DTLS Session using the given UDP socket, remote server IP Address,
      and remote server port. */
    status = nx_secure_dtls_client_session_start(&dtls_session, &udp_socket,
                                                 &ip_address, REMOTE_SERVER_PORT,
                                                 NX_WAIT_FOREVER);

   /* Allocate a DTLS packet to send some encrypted data to the server. */
    status = nx_secure_dtls_packet_allocate(&dtls_session, &pool_0, &send_packet,
                                            NX_TLS_PACKET, NX_WAIT_FOREVER);

    /* Check for errors…  */

   /* Populate the packet with some data. */
    nx_packet_data_append(send_packet, request_data, sizeof(request_data), &pool_0,
                          NX_WAIT_FOREVER);

   /* Send the request over the DTLS Session, encrypting it before sending. */
    status = nx_secure_dtls_session_send(&dtls_session, send_packet, NX_WAIT_FOREVER);

    /* Check for errors…  */
    if (status)
    {
          /* Release the packet since we could not send it.  */
          nx_packet_release(send_packet);
    }
```

```
        /* Receive the response from the server. */
        status = nx_secure_dtls_session_receive(&dtls_session, &receive_packet,
                                                NX_WAIT_FOREVER);

       /* 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);

       /* End the DTLS session now that we have received our response. */
        status = nx_secure_dtls_session_end(&tls_session, NX_WAIT_FOREVER);

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

       /* Clean up the UDP socket. */
        status =  nx_udp_socket_delete(&udp_socket);

        /* Check for errors... */

    }
```

Figure 1.1 Example of NetX Secure use with NetX

Small Example System (DTLS Server)

An example of how easy it is to use NetX Secure is described in Figure 1.2, which appears below and demonstrates a simple DTLS Server. Note that the DTLS Server functionality is quite different from DTLS Client and TLS Client/Server since the DTLS Server needs to manage multiple incoming client requests on a single UDP port (stored in the DTLS Server instance).

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

#define     DEMO_STACK_SIZE         4096

/* Define the ThreadX and NetX object control blocks.
   NOTE: These must be initialized for the target platform. See the
   NetX documentation for details. */

NX_PACKET_POOL          pool_0;
NX_IP                   ip_0;

/* 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 DTLS Server thread.  */
ULONG             dtls_server_thread_stack[6 * 1024 / sizeof(ULONG)];
TX_THREAD         dtls_server_thread;
```

```
void              server_thread_entry(ULONG thread_input);

/* Define the DTLS packet reassembly buffer. */
UCHAR packet_buffer[4000];

/* Define the metadata area for TLS/DTLS cryptography. The actual size needed can be
   Ascertained by calling nx_secure_tls_metadata_size_calculate.
*/
UCHAR crypto_metadata_buffer[4000];

/* 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. The TLS structure is also
   used for DTLS. See the NetX Secure TLS User Guide for more information.
*/
extern const NX_SECURE_TLS_CRYPTO nx_crypto_tls_ciphers;

/* Define our server certificate structure. */
NX_SECURE_X509_CERTIFICATE certificate;

/* DER-encoded certificate data for the server identity X.509 certificate. */
UCHAR device_cert_der[] = { … };
UCHAR device_cert_der_length[] = { … };
UCHAR device_cert_key_der[] = { … };
UCHAR device_cert_key_der_length[] = { … };

/* Define the number of sessions we want to allocate to our DTLS Server. */
#define DTLS_SERVER_SESSIONS (3)

/* Our DTLS Server instance. */
NX_SECURE_DTLS_SERVER dtls_server;

/* Allocate space for DTLS sessions in the DTLS server. */
UCHAR dtls_server_session_buffer[sizeof(NX_SECURE_DTLS_SESSION) * DTLS_SERVER_SESSIONS];

/* Flag used to indicate that a DTLS Client has connected. */
UINT connect_flag = 0;

/* Flag used to indicate application data reception. */
UINT receive_flag = 0;

/* Pointer to newly-connected DTLS session.
   NOTE: In practice this should be an array or list in case a new connection is
         attempted while a previous session is being started. */
NX_SECURE_DTLS_SESSION *new_dtls_session;

/* Pointer to session for application data receive. NOTE: Should be an array or list as
   with new_dtls_session */
NX_SECURE_DTLS_SESSION *receive_dtls_session;

/* Connect notify callback routine. */
UINT dtls_server_connect_notify(NX_SECURE_DTLS_SESSION *dtls_session,
                                NXD_ADDRESS *ip_address, UINT port)
{
    /* NOTE: proper inter-thread communication procedures (e.g. mutex handling)
             Omitted for clarity. */

    /* Notify application thread that a connection request has been received. */
    connect_flag = 1;
    new_dtls_session = dtls_session;

    return(NX_SUCCESS);
}

/* Receive notify callback routine invoked when DTLS application data is received
   on an existing DTLS server session. */
UINT dtls_server_receive_notify(NX_SECURE_DTLS_SESSION *dtls_session)
{
    /* Receive and process DTLS record.
       NOTE: Mutex handling omitted for clarity. */
    receive_flag = 1;
```

```
    receive_dtls_session = dtls_session;

    return(NX_SUCCESS);
}

/* Define the application - initialize drivers and UDP 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 UDP traffic. Check status for errors. */
    status =  nx_udp_enable(&ip_0);

    status =  nx_ip_fragment_enable(&ip_0);

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

     /* Create the server thread to start handling incoming requests. */
    tx_thread_create(&dtls_server_thread, "DTLS Server thread", server_thread_entry,
                     0, dtls_server_thread_stack, sizeof(dtls_server_thread_stack),
                     16, 16, 4, TX_AUTO_START);
}


/* Primary application thread for handling DTLS server operations. */
void server_thread_entry(ULONG thread_input)
{
NX_PACKET *send_packet;
NX_PACKET *receive_packet;
UCHAR receive_buffer[100];
ULONG bytes;
UINT status;

    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);

    /* Setup DTLS Server instance. */
    status = nx_secure_dtls_server_create(&dtls_server, &ip_instance, LOCAL_SERVER_PORT,
                                          NX_IP_PERIODIC_RATE, dtls_server_session_buffer,
                                          sizeof(dtls_server_session_buffer),
                                          &tls_crypto_table, crypto_metadata_buffer,
                                          sizeof(crypto_metadata_buffer), packet_buffer,
                                          sizeof(packet_buffer),
                                          dtls_server_connect_notify,
                                          dtls_server_receive_notify);

    /* Initialize local server identity certificate with key and add to server. */
    status = nx_secure_x509_certificate_initialize(&certificate, device_cert_der,
                                device_cert_der_len, NX_NULL, 0,
                                device_cert_key_der, device_cert_key_der_len,
                                NX_SECURE_X509_KEY_TYPE_RSA_PKCS1_DER);
```

```
    /* Add local server identity certificate to DTLS server with ID of 1. */
    status = nx_secure_dtls_server_local_certificate_add(&dtls_server, &certificate, 1);


    /* Start server. */
    status = nx_secure_dtls_server_start(&dtls_server);

    /* Loop continuously to handle incoming data. */
    while(1)
    {
        /* Check for new connections. Muxtex handling omitted for clarity. */
        if(connect_flag)
        {
            /* We have a new connection attempt, start the DTLS session. */
            status = nx_secure_dtls_server_session_start(new_dtls_session,
                                    NX_IP_PERIODIC_RATE);

        }

        /* Check for received application data. */
        if(receive_flag)
        {
            /* We have received data over a previously-established DTLS session.
               Mutex handling omitted for clarity. */
            status = nx_secure_dtls_session_receive(receive_dtls_session, &receive_packet,
                                                    NX_IP_PERIODIC_RATE);


            /* Process received data… */
            status = nx_packet_data_extract_offset(receive_packet, 0, receive_buffer, 100,
                                                   &bytes);
            /* Display the Client request data. */
            receive_buffer[bytes] = 0;
            printf("Received data: %s\n", receive_buffer);


            /* Prepare and send response to client. */
            status = nx_secure_dtls_packet_allocate(receive_dtls_session, &packet_pool,
                                                    &send_packet, NX_IP_PERIODIC_RATE);


            /* Populate the packet with our response data. */
            nx_packet_data_append(send_packet, response_data, response_data_length,
                                  &pool_0, NX_WAIT_FOREVER);

            /* Send response to client. */
            status = nx_secure_dtls_server_session_send(receive_dtls_session,send_packet);

        }

        /* If not processing connections or received data, let the thread sleep. */
        if(!connect_flag && !receive_flag)
        {
            tx_thread_sleep(100);
        }
    }

    /* Server processing is done, stop the server instance from accepting requests. */
    status = nx_secure_dtls_server_stop(&dtls_server);

    /* If we exit the processing loop, clean up the server. */
    status = nx_secure_dtls_server_delete(&dtls_server);
}
```

Figure 1.2 Example of NetX Secure DTLS Server

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_DTLS	This macro must be defined to enable DTLS logic in NetX Secure.
NX_SECURE_DISABLE_ERROR_CHECKING	Defined, this option removes the basic NetX Secure error checking. It is typically used after the application has been debugged.
NX_SECURE_TLS_CLIENT_DISABLED	Defined, this option removes all TLS/DTLS stack code related to Client mode, reducing code and data usage.
NX_SECURE_TLS_SERVER_DISABLED	Defined, this option removes all TLS/DTLS stack code related to Server mode, reducing code and data usage.
NX_SECURE_ENABLE_PSK_CIPHERSUITES	Defined, this option enables Pre-Shared Key (PSK) functionality. It does not disable digital certificates.
NX_SECURE_X509_STRICT_NAME_COMPARE	Defined, this option enables strict distinguished name comparison for X.509 certificates for certificate searching and verification. The default is to only compare the Common Name fields the Distinguished Names.
NX_SECURE_X509_USE_EXTENDED_DISTINGUISHED_NAMES	Defined, this option enables the optional X.509 Distinguished Name fields, at the expense of extra memory use for X.509 certificates.

NX_CRYPTO_MAX_RSA_MODULUS_SIZE	Defined, this option gives the maximum RSA modulus expected, in bits. The default value is 4096 for a 4096-bit modulus. Other values can be 3072, 2048, or 1024 (not recommended).

Chapter 3

Functional Description of NetX Secure DTLS

Execution Overview

This chapter contains a functional description of NetX Secure DTLS. There are two primary types of program execution in a NetX Secure DTLS application: initialization and application interface calls.

NetX Secure assumes the existence of ThreadX and NetX/NetXDuo. From ThreadX, it requires thread execution, suspension, periodic timers, and mutual exclusion facilities. From NetX/NetXDuo it requires the UDP and IP networking facilities and drivers.

Datagram Transport Layer Security (DTLS) and Transport Layer Security (TLS)

NetX Secure DTLS implements the Datagram Transport Layer Security protocol version 1.2 defined in RFC 6347[1].

Also included are support routines for basic X.509 (RFC 5280).

NetX Secure supports DTLS version 1.2. DTLS 1.0 (RFC 4347) is **not** currently supported.

Secure Sockets Layer (SSL) was the original name of TLS before it became a standard in RFC 2246 and “SSL” is often used as a generic name for the TLS protocols. The last version of SSL was 3.0, and TLS 1.0 is sometimes referred to as SSL version 3.1. All versions of the official “SSL” protocol are considered obsolete and insecure and currently NetX Secure does not provide an SSL implementation.

TLS specifies a protocol to generate *session keys* which are created during the TLS *handshake* between a TLS client and server and those keys are used to encrypt data sent by the application during the TLS *session.*

[1] DTLS version 1.0 was defined in RFC 4347 and corresponded to TLS version 1.1. Due to DTLS being essentially an extension to TLS, it was decided that the next version would use the same version number as the corresponding TLS version. Thus, there is no DTLS version 1.1 as DTLS version 1.2 corresponds to TLS version 1.2.

DTLS is closely coupled with TLS, as the underlying security mechansims are shared between the protocols. However, TLS is designed to work over a transport layer protocol that provides guarantees about packet delivery and order (almost always TCP in practice) and will not function over an unreliable protocol like UDP. It is precisely because of UDP that DTLS was introduced: DTLS was designed handle the unreliable nature of UDP and similar protocols. It does this by including ordering and reliability logic (e.g. retransmission of dropped data) similar to reliable protocols like TCP.

A complete discussion of TLS is included in Chapter 3 of the NetX Secure TLS User's Guide, so this document will focus on the differences between TLS and DTLS.

DTLS Record header

Any valid DTLS record must have a DTLS header, as shown in Figure 1. The header is the same as TLS with the addition of two new fields: the 16-bit *epoch* and the 48-bit *sequence number*, described below.

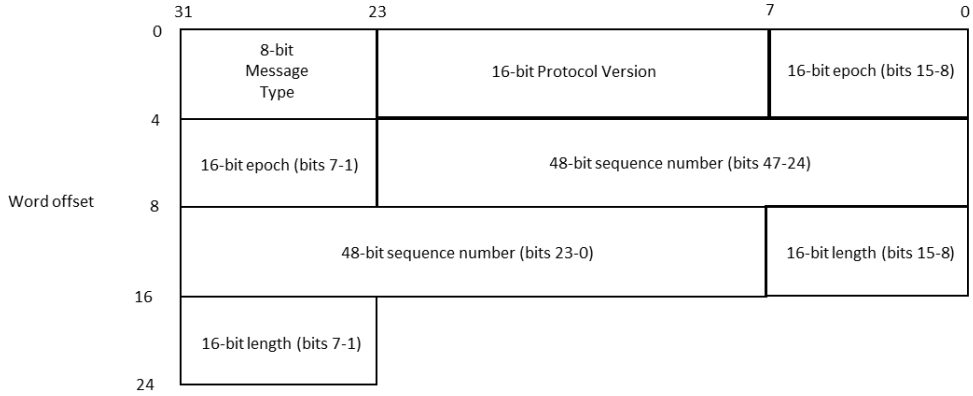

Figure 1 - DTLS record header

The fields of the TLS record header are defined as follows:

TLS Header Field	Purpose
8-bit Message Type	This field contains the type of TLS/DTLS record being sent. Valid types are as follows:

Type	Value
ChangeCipherSpec	0x14
Alert	0x15
Handshake	0x16
Application Data	0x17

16-bit Protocol Version

This field contains the DTLS protocol version. Valid values are as follows:

Protocol version	Value
DTLS 1.0[2]	0xFEFF
DTLS 1.1	0xFEFD

16-bit Epoch

This field contains the DTLS “epoch” which is a counter that is incremented each time the encryption state is changed (e.g. when generating new session keys).

48-bit Sequence Number

This field contains a sequence number which identifies this particular record. It is used by DTLS to maintain record ordering and check for retransmission need.

16-bit Length

This field contains the length of the data encapsulated in the DTLS record.

[2] DTLS version 1.0 is not currently supported in NetX Secure.

DTLS Handshake Record header

Any valid DTLS handshake record must have a DTLS Handshake header, as shown in Figure 2.

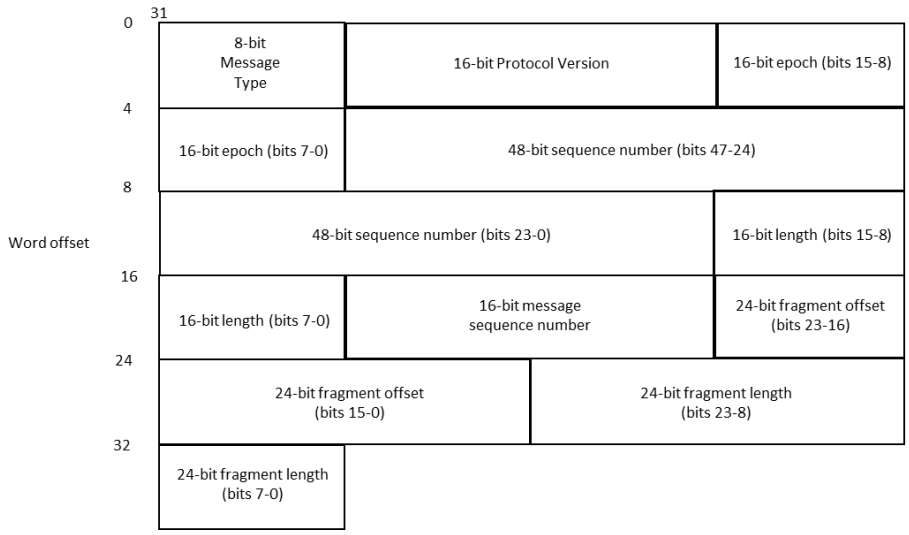

Figure 2 - DTLS Handshake record header

The fields of the DTLS Handshake record header are defined as follows:

TLS Header Field	Purpose
8-bit Message Type	This field contains the type of DTLS record being sent. Valid types are as follows: **Type** — **Value** ChangeCipherSpec — 0x14 Alert — 0x15 Handshake — 0x16 Application Data — 0x17
16-bit Epoch	This field contains the DTLS “epoch” which is a counter that is incremented each time the encryption state is changed (e.g. when generating new session keys).
48-bit Sequence Number	This field contains a sequence number which identifies this particular record. It is used by DTLS to maintain record ordering and check for retransmission need.

16-bit Protocol Version

This field contains the DTLS protocol version. Valid values are as follows:

Protocol version	Value
DTLS 1.0[3]	0xFEFF
DTLS 1.1	0xFEFD

16-bit Length

This field contains the length of the data encapsulated in the DTLS record.

8-bit Handshake Type

This field contains the handshake message type. Valid values are as follows:

Type	Value
HelloRequest	0x00
ClientHello	0x01
ServerHello	0x02
Certificate	0x0B
ServerKeyExchange	0x0C
CertificateRequest	0x0D
ServerHelloDone	0x0E
CertificateVerify	0x0F
ClientKeyExchange	0x10
Finished	0x14

24-bit Length

This field contains the length of the handshake message data.

16-bit Sequence Number

This field contains a sequence number.

[3] DTLS version 1.0 is not currently supported in NetX Secure.

The DTLS Handshake and DTLS Session

A typical DTLS handshake is shown in Figure 3. It is nearly identical to the typical TLS handshake with an important difference – when the ClientHello message is first sent, the server responds with a new DTLS-specific message, *HelloVerifyRequest* which contains a "cookie". The DTLS Client must respond with a second ClientHello message containing that cookie before the handshake can proceed. This mechanism was added to DTLS to prevent certain Denial of Service (DoS) attacks since UDP is a connectionless protocol (TCP requires a dedicated connection/port so TLS does not suffer from the same issue).

A DTLS handshake begins when the Client sends a *ClientHello* message to a DTLS server, indicating its desire to start a DTLS session. The message contains information about the encryption the client would like to use for the session, along with information used to generate the session keys later in the handshake. Until the session keys are generated, all messages in the DTLS handshake are not encrypted. As mentioned above, the DTLS Server may send a HelloVerifyRequest in response to the ClientHello, forcing the client to respond with a second updated ClientHello.

Upon receiving the second ClientHello message, the DTLS Server will verify the cookie and if correct will respond with a ServerHello message indicating a selection from the encryption options provided by the client. The ServerHello is followed by a Certificate message, in which the server provides a digital certificate to authenticate its identity to the client (if X.509 verification is used). Finally, the server sends a ServerHelloDone message to indicate it has no more messages to send. The server may optionally send other messages following the ServerHello and in some cases it may not send a Certificate message (such as when Pre-Shared Keys are used), hence the need for the ServerHelloDone message.

Once the client has received all the server's messages, it has enough information to generate the session keys. TLS/DTLS does this by creating a shared bit of random data called the *Pre-Master Secret*, which is a fixed-size and is used as a seed to generate all the keys needed once encryption is enabled. The Pre-Master Secret is encrypted using the public key algorithm (e.g. RSA) specified in the Hello messages (see below for information on public key algorithms) and the public key provided by the server in its certificate. An optional TLS/DTLS feature called Pre-Shared Keys (PSK) enables ciphersuites that do not use a certificate but instead use a secret value shared between the hosts (usually through physical transfer or other secured method). When PSK is enabled, the pre-shared secret key is used to generate the Pre-Master

Secret. See the section on Pre-Shared Keys in “Authentication Methods” below.

In a usual TLS/DTLS handshake, the encrypted Pre-Master Secret is sent to the server in the ClientKeyExchange message. The server, upon receiving the ClientKeyExchange message, decrypts the Pre-Master Secret using its private key and proceeds to generate the session keys in parallel with the TLS/DTLS client.

Once the session keys are generated, all further messages can be encrypted using the private-key algorithm (e.g. AES) selected in the Hello messages. One final un-encrypted message called ChangeCipherSpec is sent by both the client and server to indicate that all further messages will be encrypted.

The first encrypted message sent by both the client and server is also the final TLS handshake message, called Finished. This message contains a hash of all the handshake messages received and sent. This hash is used to verify that none of the messages in the handshake have been tampered with or corrupted (indicating a possible breach of security).

Once the Finished messages are received and the handshake hashes are verified, the TLS/DTLS session begins, and the application begins sending and receiving data. All data sent by either side during the TLS/DTLS session is first hashed using the hash algorithm chosen in the Hello messages (to provide message integrity) and encrypted using the chosen private-key algorithm with the generated session keys.

Finally, a TLS/DTLS session can only be successfully ended if either the Client or Server chooses to do so. A truncated session is considered a security breach (since an attacker may be attempting to prevent all the data being sent from being received) so a special notification is sent when either side wants to end the session, called a CloseNotify alert. Both the client and server must send and process a CloseNotify alert for a successful session shutdown.

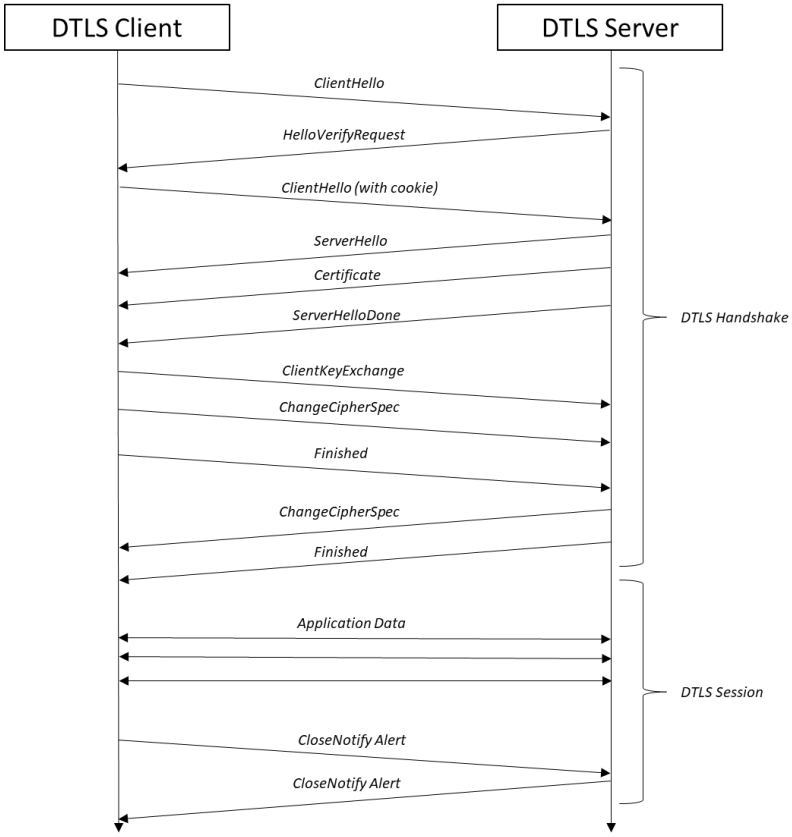

Figure 3- Typical DTLS handshake

Initialization

The NetX or NetXDuo stack must be initialized prior to using NetX Secure DTLS. Refer to the NetX or NetXDuo User Guide for information on how to properly initialize the TCP/IP stack for UDP operation.

Once NetX UDP has been initialized, DTLS can be enabled. Internally, all DTLS network traffic and processing is handled by the NetX/NetXDuo stack without requiring user intervention. However, DTLS has some specific requirements that must be handled separately from the underlying network stack. DTLS Client operation these parameters are assigned to the DTLS control block called ***NX_SECURE_DTLS_SESSION***. For DTLS Server operation, control block is called ***NX_SECURE_DTLS_SERVER*** and it contains the infrastructure needed to handle multiple DTLS sessions on a single UDP port – note that this is different from TLS where a each TLS session is bound to a single TCP port.

The two DTLS modes, Server and Client, may be enabled in an application (but only one mode per NetX socket), and each have their own specific requirements, detailed below.

Initialization – DTLS Server

NetX Secure DTLS Server mode differs from TLS Server mode due to the use of UDP for the underlying network transport protocol. With TCP, the port is bound to a single remote host for the duration of the TLS session. UDP has no notion of state with regard to the remote host so DTLS requests from different hosts will all be received on the same UDP interface. Therefore, DTLS must maintain session state rather than relying on the socket as with TLS and TCP. For this reason, the DTLS Server control block (NX_SECURE_DTLS_SERVER) maintains a mapping of remote host information (IP address and port) to DTLS sessions. All incoming data on the UDP socket assigned to a DTLS Server will be mapped to an existing or new DTLS session based on the remote host. For this reason, the DTLS server creation requires several additional parameters beyond what TLS and DTLS Client need.

In addition to the DTLS Server control block, TLS ciphersuites, and cipher scratchspace/metadata buffer, DTLS Servers require a buffer to maintain DTLS sessions and a packet reassembly buffer used to decrypt incoming DTLS records.

In addition to the session buffers, DTLS Servers require a *Digital Certificate*, which is a document used to identify the TLS server to the connecting TLS client, and the certificates corresponding *Private Key*, usually for the RSA encryption algorithm. The International Telecommunications Union X.509 standard specifies the certificate format used by TLS/DTLS and there are numerous utilities for creating X.509 digital certificates.

For NetX Secure DTLS, the X.509 certificate must be binary-encoded using the Distinguished Encoding Rules (DER) format of ASN.1. DER is the standard TLS over-the-wire binary format for certificates.

The private key associated with the provided certificate must be in DER-Encoded PKCS#1 format. The private key is only used on the device and will never be transmitted over the wire. Keep private keys safe as they provide the security for TLS/DTLS communications!

To initialize the DTLS Server certificate, the application must provide a pointer to a buffer containing the DER-encoded X.509 certificate and optional DER-encoded PKCS#1 RSA private key data using the ***nx_secure_x509_certificate_intialize*** service, which populates the **NX_SECURE_X509_CERT** structure with the appropriate certificate data for use by TLS.

Once the server certificate has been initialized, it must be added to the TLS control block using the ***nx_secure_dtls_server_local_certificate_add*** service.

Once the server's certificate has been added to the DTLS Server control block, the server can be used for secure DTLS communications (see example above).

Initialization – DTLS Client

NetX Secure DTLS Client mode is simple in operation compared to the DTLS server since there is only a single outgoing connection to the remote host over the UDP socket.

To setup a DTLS Client, it requires a *Trusted Certificate Store*, which is a collection of X.509-encoded digital certificates from trusted Certificate Authorities (CA's). These certificates are assumed by the DTLS protocol to be "trusted" and serve as the basis for authenticating certificates provided by DTLS server entities to the NetX Secure DTLS Client application.

A trusted CA certificate may either be *self-signed* or signed by another CA, in which case that certificate is called an *Intermediate CA* (ICA). In a typical TLS/DTLS application, the server provides the ICA certificates along with its server certificate, but the only requirement for successful authentication is that a chain of issuers (certificates used to sign other certificates) can be traced from the server certificate back to a trusted CA certificate in the Trusted Certificate Store. This chain is known as a *chain of trust* or *certificate chain*.

To initialize a trusted CA or ICA certificate, the application must provide a pointer to a buffer containing the DER-encoded X.509 certificate using the ***nx_secure_x509_certificate_intialize*** service, which populates the **NX_SECURE_X509_CERT** structure with the appropriate certificate data for use by TLS.

The DTLS Client also needs space for the incoming server certificate to be allocated (assuming a Pre-Shared Key mode is not being used) and a buffer for assembling packets into DTLS records to be decrypted. These buffers are passed in as parameters to the ***nx_secure_dtls_session_create*** service (see API reference for more information).

Trusted certificates that have been initialized are then added to the created DTLS session control block using the ***nx_secure_dtls_session_trusted_certificate_add*** service. Failure to

add a certificate will cause the DTLS Client session to fail as there will be no way for the DTLS protocol to authenticate remote server hosts.

Once the Trusted Certificate Store has been created the session may be used to establish a secure TLS Client connection.

Application Interface Calls

NetX Secure DTLS applications will typically make function calls from within application threads running under the ThreadX RTOS. Some initialization, particularly for the underlying network communications protocols (e.g. UDP and IP) may be called from ***tx_application_define*.** See the NetX/NetXDuo User Guide for more information on initializing network communications.

DTLS makes heavy use of encryption routines which are processor-intensive operations. Generally, these operations will be performed within the context of calling thread.

DTLS Session Start

DTLS requires an underlying transport-layer network protocol in order to function. The protocol typically used is TCP. In order to establish a NetX Secure TLS session an **NX_UDP_SOCKET** must be created and passed into the ***nx_secure_dtls_client_session_start*** service for DTLS Clients.

DTLS Servers operate differently. The UDP socket used for incoming DTLS Client requests is contained within the NX_SECURE_DTLS_SERVER control block and is initialized in the call to ***nx_secure_dtls_server_create***, which takes the local UDP port as a parameter. The service ***nx_secure_dtls_server_start*** is then used to start the DTLS Server to handle incoming requests. All incoming requests are handled in callback routines provided to *nx_secure_dtls_server_create*: one for connections and one for receive notifications. It is up to the application to handle starting the DTLS session when a connection notification is received (the connect notify callback is invoked by DTLS) by calling ***nx_secure_dtls_server_session_start***. The application also must handle incoming data when the receive notify callback is invoked (which follows a completed DTLS handshake) by calling ***nx_secure_dtls_session_receive***. The details of this are provided in the example above and in the API reference for each of the above mentioned services.

DTLS Packet Allocation

NetX Secure DTLS uses the same packet structure as NetX/NetXDuo TCP (***NX_PACKET***) except that instead of calling the

nx_packet_allocate service, the ***nx_secure_dtls_packet_allocate*** service must be called so that space for the DTLS header may be allocated properly.

DTLS Session Send

Once the TLS session has started, the application may send data using the ***nx_secure_dtls_session_send*** service. The send service is identical in use to the ***nx_udp_socket_send*** service, taking an ***NX_PACKET*** data structure containing the data being sent, a target IP address, and a target UDP port.

When sending data using nx_secure_dtls_session_send, it is important to use the same IP address and port that were used to establish the DTLS session, unless there is a mechanism in place to move the session to a new address and UDP port on-the-fly (this is not common).

Any data sent over DTLS will be encrypted by the NX Secure DTLS stack and the configured encryption routines before being sent.

DTLS Session Receive

Once the DTLS session has started, the application may begin receiving data using the ***nx_secure_Dtls_session_receive*** service. Like the DTLS Session send, this service is identical in use to ***nx_udp_socket_receive***, except that the incoming data is decrypted and verified by the DTLS stack before being returned in the packet structure.

TLS Session Close

Once a DTLS session is complete, both the DTLS client and server must send a CloseNotify alert to the other side to shut down the session. Both sides must receive and process the alert to ensure a successful session shutdown.

If the remote host sends a CloseNotify alert, any calls to the ***nx_secure_dtls_session_receive*** service will process the alert, send the corresponding alert back to the remote host, and return a value of ***NX_SECURE_TLS_SESSION_CLOSED***. Once the session is closed, any further attempts to send or receive data with that DTLS session will fail.

If the application wishes to close the TLS session, the ***nx_secure_dtls_session_end*** service must be called. The service will send the CloseNotify alert and process the response CloseNotify. If the response is not received, an error value of ***NX_SECURE_TLS_SESSION_CLOSE_FAIL*** will be returned, indicating

that the DTLS session was not cleanly shutdown, a possible security breach.

TLS/DTLS Alerts

TLS/DTLS is designed to provide maximum security, so any errant behavior in the protocol is considered a potential security breach. For this reason, any errors in message processing or encryption/decryption are considered fatal errors that terminate the handshake or session immediately.

While handling errors in a local application is relatively straightforward, the remote host needs to know that an error has occurred in order to properly handle the situation and prevent any further possible security breaches. For this reason, TLS/DTLS will send an *Alert* message to the remote host upon any error.

Alerts are treated in the same manner as any other TLS/DTLS messages and are encrypted during the session to prevent an attacker from gathering information from the type of alert provided. During the handshake, the alerts sent are limited in scope to limit the amount of information that could be obtained by a potential attacker.

The CloseNotify alert, used to close the TLS/DTLS session, is the only non-fatal alert. While it is considered an alert and sent as an alert message, a CloseNotify is unlike other alerts in that it does not indicate an error has occurred.

TLS/DTLS Session Renegotiation and Resumption

TLS supports the notion of “renegotiation” which is simply a renegotiation of the TLS session parameters within the context of an existing TLS session.

TLS session *resumption* should not be confused with session *renegotiation*, despite some similarities. Where session *renegotiation* involves starting a new handshake within an existing TLS session, session *resumption* is a purely optional feature that involves restarting a closed TLS session without performing a complete TLS handshake.

NX Secure DTLS handles incoming renegotiation requests from remote hosts. It does **not** support session resumption. A more complete discussion of these features can be found in Chapter 3 of the NetX Secure TLS User Guide.

Protocol Layering

The TLS protocol (and therefore DTLS as well) fits into the networking stack between the transport layer (e.g. TCP or UDP) and the application layer. TLS is sometimes considered a transport-layer protocol (hence *Transport Layer* Security) but because it acts as an application with regard to the underlying network protocols it is sometimes grouped into the application layer.

TLS requires a transport layer protocol that supports in-order and lossless delivery, such as TCP. Due to this requirement, TLS cannot run on top of UDP since UDP does not guarantee delivery of datagrams. *DTLS* is a modified version of TLS, is used for applications that need the security of TLS over a datagram protocol like UDP.

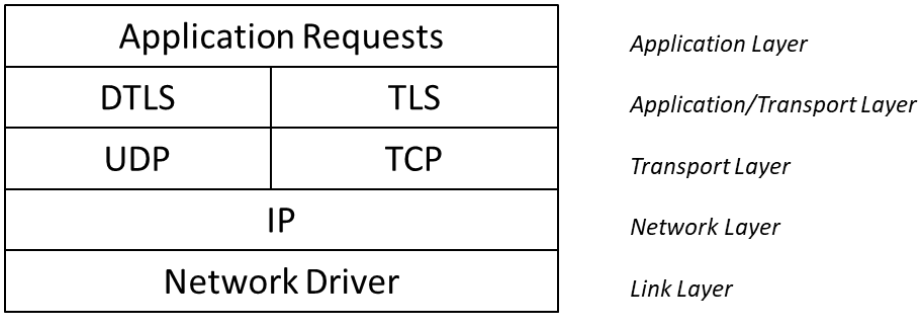

Figure 4- TCP/IP, UDP and TLS/DTLS protocol layers

Network Communications Security and Encryption

Securing communications over public networks and the Internet is a critically important topic and the subject of vast numbers of books, articles, and solutions. The topic is mind-bogglingly complex, but can be reduced to a simple idea: sending information over a network so that only the intended target can access or change that information. This breaks down into three important concepts: secrecy, integrity, and authentication. The TLS/DTLS protocol provides solutions for all three.

Encryption is used in different ways to provide secrecy, integrity, and authentication within the TLS and DTLS protocols. The encryption must be supplied to TLS or DTLS upon creation of a session or server instance as TLS provides a flexible framework for using encryption and not the encryption itself. NetX Secure DTLS provides the necessary encryption routines for most applications so you do not have to be concerned about finding appropriate encryption.

A more detailed description of these topics can be found in Chapter 3 of the NetX Secure TLS User Guide.

TLS and DTLS Extensions

TLS (and therefore DTLS) provides a number of extensions that provide additional functionality for certain applications. These extensions are typically sent as part of the ClientHello or ServerHello messages, indicating to a remote host the desire to use an extension or providing additional details for use in establishing the secure TLS session.

NetX Secure DTLS supports all of the extensions found in NetX Secure TLS, and a complete description of those can be found in the NetX Secure TLS User Guide, Chapter 3.

Authentication Methods

TLS and DTLS provide the framework for establishing a secure connection between two devices over an insecure network, but part of the problem is knowing the identity of the device on the other end of that connection. Without a mechanism for authenticating the identity of remote hosts, it becomes a trivial operation for an attacker to pose as a trusted device.

Initially, it may seem that using IP addresses, hardware MAC addresses, or DNS would provide a relatively high level of confidence for identifying hosts on a network, but given the nature of TCP/IP technology and the ease with which addresses can be spoofed and DNS entries corrupted (e.g. through DNS cache poisoning), it becomes clear that TLS needs an additional layer of protection against fraudulent identities.

There are various mechanisms that can provide this extra layer of authentication for TLS, but the most common is the *digital certificate.* Other mechanisms include Pre-Shared Keys (PSK) and password schemes.

Digital Cerificates

Digital certificates are the most common method for authenticating a remote host in TLS. Essentially, a digital certificate is a document with specific formatting that provides identity information for a device on a computer network.

TLS normally uses a format called X.509, a standard developed by the International Telecommunication Union, though other formats of certificates may be used if the TLS hosts can agree on the format being used. X.509 defines a specific format for certificates and various encodings that can be used to produce a digital document. Most X.509

certificates used with TLS are encoded using a variant of ASN.1, another telecommunications standard. Within ASN.1 there are various digital encodings, but the most common encoding for TLS certificates is the Distinguished Encoding Rules (DER) standard. DER is a simplified subset of the ASN.1 Basic Encoding Rules (BER) that is designed to be unambiguous, making parsing easier. Over the wire, TLS certificates are usually encoded in binary DER, and this is the format that NetX Secure expects for X.509 certificates.

Though DER-formatted binary certificates are used in the actual TLS protocol, they may be generated and stored in a number of different encodings, with file extensions such as .pem, .crt, and .p12. The different variants are used by different applications from different manufacturers, but generally all can be converted into DER using widely available tools.

The most common of the alternative certificate encodings is PEM. The PEM format (from Privacy-Enhanced Mail) is a base-64 encoded version of the DER encoding that is often used because the encoding results in printable text that can be easily sent using email or web-based protocols.

Generating a certificate for your NetX Secure application is generally outside the scope of this manual, but the OpenSSL command-line tool (www.openssl.org) is widely available and can convert between most formats.

Depending on your application, you may generate your own certificates, be provided certificates by a manufacturer or government organization, or purchase certificates from a commercial certificate authority.

To use a digital certificate in your NetX Secure application, you must first convert your certificate into a binary DER format and, optionally, convert the associated private key (the “private exponent” for RSA, for example) into a binary format, typically a PKCS#1-formatted, DER-encoded RSA key. Once the conversion is complete, it is up to you to load the certificate and private key onto the device. Possible options include using a flash-based file system or generating a C array from the data (using a tool such as “xxd” from Linux) and compiling the certificate and key into your application as constant data.

Once your certificate is loaded onto the device, the DTLS API can be used to associate your certificate with a DTLS session or server.

For details and examples on how to use X.509 certificates with NetX Secure DTLS, see the section “Importing X.509 certificates into NetX Secure” in the NetX Secure TLS User Guide.

Refer to the following DTLS services in the API reference for more information:
nx_secure_x509_certificate_initialize,
nx_secure_dtls_session_local_certificate_add,
nx_secure_dtls_server_local_certificate_add,
nx_secure_dtls_session_local_certificate_remove,
nx_secure_dtls_server_local_certificate_remove,
nx_secure_dtls_session_trusted_certificate_add,
nx_secure_dtls_server_trusted_certificate_add,
nx_secure_dtls_session_trusted_certificate_remove
nx_secure_dtls_server_trusted_certificate_remove

TLS Client Certificate Specifics

DTLS Client implementations generally do not require a “local” certificate[4] to be loaded onto the device. The exception to this is when Client Certificate Authentication is enabled, but this is less common.

A DTLS Client requires at least one “trusted” certificate[5] to be loaded (more may be loaded if required), and space for a “remote” certificate[6] to be allocated.

For more information on adding trusted certificates and allocating space for remote certificates, see the TLS API reference for the following services: nx_secure_dtls_session_create, nx_secure_dtls_session_trusted_certificate_add.

TLS/DTLS Server Certificate Specifics

DTLS Server implementations generally do not require “trusted” certificates to be loaded onto the device or remote certificates to be allocated. The exception to this being when Client Certificate Authentication is enabled.

A TLS Server requires a “local” (or “identity”) certificate to be loaded so the server can provide it to the remote client during the TLS handshake to authenticate the server to the client.

---

[4] A “local” certificate is a certificate that identifies the local device – that is, it provides identity information for the device upon which the TLS/DTLS application is loaded.
[5] A “trusted” certificate is a certificate that provides a basis for trust and authentication of the remote device, either directly or through a Public Key Infrastructure (PKI). The root of the chain of trust is usually called a “Certification Authority” or CA certificate.
[6] A “remote” certificate refers to the certificate sent by the remote host during the TLS handshake. It provides identity for that remote host and is authenticated by comparing it to a “trusted” certificate on the local device.

For more information about loading local certificates for use with NetX TLS server applications, see the API reference for the following services: nx_secure_dtls_server_local_certificate_add, nx_secure_dtls_server_local_certificate_remove.

Pre-Shared Keys (PSK)

An alternative mechanism for providing identification authentication in TLS is the notion of Pre-Shared Keys (PSK). Using a PSK ciphersuite removes the need to do the processor-intensive public-key encryption operations, a boon for resource-constrained embedded devices. The PSK replaces the certificate in the TLS/DTLS handshake and is used in place of the encrypted Pre-Master Secret for TLS/DTLS session key generation.

The PSK ciphersuites are limited in the sense that that a shared secret must be present on both devices before a TLS/DTLS session can be established. This means that the devices must have been loaded with that secret using some secure means other than a TLS PSK connection - PSKs may be updated over a TLS PSK connection, but the device must necessarily start with a PSK that is loaded through some other mechanism. For example, a sensor device and its gateway device could be loaded with PSKs in the factory before shipping, or a standard TLS connection (with a certificate) could be used to load the PSK.

PSK ciphersuites come in a couple of forms, described in RFC 4279. The first uses RSA or Diffie-Hellman keys which are used in the same manner as the public keys transmitted in the certificate in standard TLS handshakes. The second form, which is of more use in a resource-constrained environment, uses a PSK that is used to directly generate the session keys (for use by AES, for example), avoiding the use of the expensive RSA or Diffie-Hellman operations.

NetX Secure supports the second form of PSK ciphersuites, enabling applications to remove all public-key cryptography code and memory usage. The PSK itself is not an AES key, but can be considered as being more like a password from which the actual keys are generated. There are few restrictions on what the PSK value can be, though longer values will provide more security (same as with passwords).

To use PSK with your NetX Secure application, you must first define the global macro **NX_SECURE_ENABLE_PSK_CIPHERSUITES**. This is usually done through your compiler settings, but the definition can also be placed in the nx_secure_tls.h header file. With the macro defined, PSK ciphersuite support will be compiled into your NetX Secure DTLS application.

With PSK support enabled, you can then use the DTLS API to set up PSKs for your application. Each PSK will require a PSK value (the actual secret “key” – keep this value safe), an “identity” value used to identify the specific PSK, and an “identity hint” that is used by a TLS server to choose a particular PSK value.

The PSK itself can be any binary value as it is never sent over a network connection. The PSK can be any value up to 64 bytes in length.

The identity and hint must be printable character strings formatted using UTF-8. The identity and hint values may be any length up to 128 bytes.

The identity and PSK form a unique pair that is loaded onto every device in the network that need to communicate with one another.

The “hint” is primarily used for defining specific application profiles to group PSKs by function or service. These values must be agreed upon in advance and are application dependent. As an example, the OpenSSL command-line server application (with PSK enabled) uses the default string “Client_identity”, which must be provided by a TLS client in order to continue with the TLS handshake.

For more information on PSKs, see the NetX Secure API reference for the following services: nx_secure_dtls_psk_add, nx_secure_dtls_server_psk_add.

Importing X.509 certificates into NetX Secure

Digital certificates are required for most TLS connections on the Internet. Certificates provide a method for authenticating previously unknown hosts over the Internet through the use of trusted intermediaries, usually called *Certificate Authorities* or CAs. To connect your NetX Secure device with a commercial cloud service (such as Amazon Web Services), you will need to import certificates into your application by loading them onto your device.

Along with certificates, you will also sometimes need a *private key* that is associated with your certificate. In some applications (such as TLS Client when Client Certificate Authentication is not being used) the certificate alone will be sufficient, but if your certificate is being used to identify your device you will need a private key. Private keys are typically generated when you create your certificate and are stored in a separate file, often encrypted with a password.

For a detailed description of importing certificates into NetX Secure applications, please refer to Chapter 3 in the NetX Secure TLS User Guide.

Client Certificate Authentication in NetX Secure TLS

When using X.509 certificate authentication, the TLS/DTLS protocol requires that the DTLS Server instance provide a certificate for identification, but by default the DTLS Client instance does not need to provide a certificate for authentication, using another form of authentication instead (e.g. a username/password combination). This matches the most common use of TLS on the Internet for Web sites. For example, an online retail site must prove to a potential customer using a web browser that the server is legitimate, but the user will use a login/password to access a specific account.

However, the default case is not always desirable, so TLS/DTLS optionally allows for the DTLS Server instance to request a certificate from the remote Client. When this feature is enabled, the DTLS Server will send a CertificateRequest message to the DTLS Client during the handshake. The Client must respond with a certificate of its own and a CertificateVerify message which contains a cryptographic token proving that the Client owns the matching private key associated with that certificate. If the verification fails or the certificate is not connected to a trusted certificate on the Server, the TLS handshake fails.

There are two separate cases for Client Certificate Authentication in TLS – the following sections cover both cases.

Client Certificate Authentication for DTLS Clients

A DTLS Client may attempt a connection to a server that requests a certificate for client authentication. In this case the Client must provide a certificate to the server and verify that it owns the matching private key or the Server will terminate the DTLS handshake.

In NetX Secure DTLS, there is no special configuration to support this feature but the application will have to provide a local identification certificate for the TLS Client instance using the *nx_secure_tls_session_local_certificate_add* service. If no certificate is provided by the application but the remote server is using Client Certificate Authentication and requests a certificate, the DTLS handshake will fail. The certificate provided to the DTLS Session with *nx_secure_dtls_session_local_certificate_add* must be recognized by the remote server in order to complete the DTLS handshake.

Client Certificate Authentication for TLS Servers

The DTLS Server case for Client Certificate Authentication is slightly more complex than the DTLS Client case due to the feature being optional. In this case, the TLS Server needs to specifically request a certificate from the remote TLS Client, then process the CertificateVerify message to verify that the remote Client owns the matching private key, and then the Server must check that the certificate provided by the Client can be traced to a certificate in the local trusted certificate store.

In NetX Secure TLS Server instances, Client Certificate Authentication is controlled by the *nx_secure_dtls_server_x509_client_verify_configure* and *nx_secure_dtls_server_x509_client_verify_disable* services.

To enable Client Certificate Authentication, an application must call *nx_secure_dtls_server_x509_client_verify_configure* with the DTLS Server session instance before calling *nx_secure_dtls_server_start*. The verification requires space to be allocated for incoming client certificates which is provided as a parameter to *nx_secure_dtls_server_x509_client_verify_configure*. Note that the buffer must be large enough to hold the maximum-size certificate chain provided by a client *times the number of DTLS server sessions*. Each server session requires space which will be allocated from the single provided buffer. Make sure the buffer is large enough or an error will occur if the provided Client certificate chain is too large.

When Client Certificate Authentication is enabled, the DTLS Server will request a certificate from the remote DTLS Client during the DTLS handshake. In NetX Secure DTLS Server, the Client certificate is checked against the store of trusted certificates created with *nx_secure_dtls_server_trusted_certificate_add* by following the X.509 issuer chain. The remote Client must provide a chain that connects its identity certificate to a certificate in the trusted store or the DTLS handshake will fail. Additionally, if the CertificateVerify message processing fails, the DTLS handshake will also fail.

The signature methods used for the CertificateVerify method are fixed for TLS version 1.0 and TLS version 1.1, and are specified by the TLS Server in TLS version 1.2, upon which NetX Secure DTLS is based. For DTLS 1.2, the signature methods supported generally follow the relevant methods supplied in the cryptographic method table, but typically RSA with SHA-256 (see the section “Cryptography in NetX Secure TLS” for more information on initializing TLS with cryptographic methods).

Cryptography in NetX Secure TLS

TLS defines a protocol in which cryptography can be used to secure network communications. As such, it leaves the actual cryptography to be used fairly wide open for TLS users. The specification only requires a single ciphersuite to be implemented – in the case of TLS 1.2, that ciphersuite is TLS_RSA_WITH_AES_128_CBC_SHA, indicating the use of RSA for public-key operations, AES in CBC mode with 128-bit keys for session encryption, and SHA-1 for message authentication hashes.

Being TLS 1.2-compliant, NetX Secure enables the mandatory TLS_RSA_WITH_AES_128_CBC_SHA ciphersuite by default, but given the number of possible implementations for each of the cryptographic methods due to hardware capabilities and other considerations, NetX Secure provides a generic cryptographic API that allows a user to specify which cryptographic methods to use with TLS.

NOTE: The generic cryptographic API mechanism also allows users to implement their own ciphersuites, but this is recommended for advanced users who are familiar with the TLS ciphersuites and extensions. Please contact your Express Logic representative if you are interested in supporting your own ciphersuites.

Please see the NetX Secure TLS User Guide, Chapter 3 for a detailed discussion about how to configure cryptographic methods for DTLS. The same process applies to both TLS and DTLS.

Chapter 4

Description of NetX Secure DTLS Services

This chapter contains a description of all NetX Secure DTLS services (listed below) in alphabetic order.

In the “Return Values” section in the following API descriptions, values in **BOLD** are not affected by the **NX_SECURE_DISABLE_ERROR_CHECKING** macro that is used to disable API error checking, while non-bold values are completely disabled.

nx_secure_dtls_client_session_start

Start a NetX Secure DTLS Client Session

Prototype

```
UINT nx_secure_dtls_client_session_start(
                        NX_SECURE_DTLS_SESSION *dtls_session,
                        NX_UDP_SOCKET *udp_socket,
                        NXD_ADDRESS *ip_address, UINT port,
                        UINT wait_option);
```

Description

This service starts a DTLS Client session, connecting to the server at the provided IP address and UDP port, using the provided UDP socket for network communications.

The DTLS session control block must be initialized prior to calling this service using nx_secure_dtls_session_create. Additionally, the DTLS Client requires that at least one trusted CA certificate has been added to the session using nx_secure_dtls_session_trusted_certificate_add or Pre-Shared Keys are enabled and configured.

Parameters

dtls_session	Pointer to a DTLS Session structure that was initialized previously.
udp_socket	Initialized UDP socket that will be used to establish network communications with the remote DTLS server.
ip_address	Pointer to IP address structure containing the address of the remote DTLS server.
port	Initialized UDP socket that will be used to establish network communications with the remote DTLS server.
wait_option	Suspension option for connection attempt.

Return Values

NX_SUCCESS	(0x00)	Successful assignment of certificate to session.
NX_NOT_CONNECTED	(0x38)	The server cannot be reached

at the given address and port.

NX_SECURE_TLS_UNRECOGNIZED_MESSAGE_TYPE
(0x102) A received TLS/DTLS message type is incorrect.

NX_SECURE_TLS_UNSUPPORTED_CIPHER
(0x106) A cipher provided by the remote host is not supported.

NX_SECURE_TLS_HANDSHAKE_FAILURE
(0x107) Message processing during the TLS handshake has failed.

NX_SECURE_TLS_HASH_MAC_VERIFY_FAILURE
(0x108) An incoming message failed a hash MAC check.

NX_SECURE_TLS_TCP_SEND_FAILED
(0x109) An underlying TCP socket send failed.

NX_SECURE_TLS_INCORRECT_MESSAGE_LENGTH
(0x10A) An incoming message had an invalid length field.

NX_SECURE_TLS_BAD_CIPHERSPEC
(0x10B) An incoming ChangeCipherSpec message was incorrect.

NX_SECURE_TLS_INVALID_SERVER_CERT
(0x10C) An incoming TLS certificate is unusable for identifying the remote DTLS server.

NX_SECURE_TLS_UNSUPPORTED_PUBLIC_CIPHER
(0x10D) The public-key cipher provided by the remote host is unsupported.

NX_SECURE_TLS_NO_SUPPORTED_CIPHERS
(0x10E) The remote host has indicated no ciphersuites that are supported by the NetX Secure DTLS stack.

NX_SECURE_TLS_UNKNOWN_TLS_VERSION
(0x10F) A received DTLS message had an unknown DTLS version in its header.

NX_SECURE_TLS_UNSUPPORTED_TLS_VERSION
(0x110) A received DTLS message had a known but unsupported DTLS version in its header.

NX_SECURE_TLS_ALLOCATE_PACKET_FAILED
(0x111) An internal TLS packet

		allocation failed.
NX_SECURE_TLS_INVALID_CERTIFICATE		
	(0x112)	The remote host provided an invalid certificate.
NX_SECURE_TLS_ALERT_RECEIVED		
	(0x114)	The remote host sent an alert indicating an error and ending the TLS session.
NX_SECURE_TLS_MISSING_CRYPTO_ROUTINE		
	(0x13B)	An entry in the ciphersuite table had a NULL function pointer.
NX_PTR_ERROR	(0x07)	Invalid session, socket, or address pointer.

Allowed From

Threads

Example

```
/* Sockets, sessions, certificates defined in global static space to preserve
   application stack. */
NX_SECURE_DTLS_SESSION client_dtls_session;

/* Trusted certificate structure and raw data. */
NX_SECURE_X509_CERT trusted_cert;
const UCHAR trusted_cert_der = { … };

/* Cryptography routines and crypto work buffers. */
extern const NX_SECURE_TLS_CRYPTO nx_crypto_tls_ciphers;
UCHAR crypto_metadata[9000];

/* Packet reassembly buffer for decryption. */
UCHAR packet_buffer[4000];

/* Remote certificate buffer for incoming certificates. */
#define REMOTE_CERT_SIZE (sizeof(NX_SECURE_X509_CERT) + 2000)
#define REMOTE_CERT_NUMBER  (3)
UCHAR remote_certs_buffer[REMOTE_CERT_SIZE * REMOTE_CERT_NUMBER];

/* Application thread where TLS session is started. */
void application_thread()
{
NXD_ADDRESS server_ip;

    /* Create a TLS session for our socket. Ciphers and metadata defined
       elsewhere. See nx_secure_tls_session_create reference for more
       information. */
    status =  nx_secure_dtls_session_create(&client_dtls_session,
                                        &nx_crypto_tls_ciphers,
                                        crypto_metadata,
                                        sizeof(crypto_metadata),
                                         packet_buffer,
                                         sizeof(packet_buffer),
                                         REMOTE_CERT_NUMBER,
                                         remote_certs_buffer,
                                         sizeof(remote_certs_buffer));

    /* Check for error.  */
    if(status)
```

```
    {
        printf("Error in function nx_secure_dtls_session_create: 0x%x\n", status);
    }

    /* Initialize our trusted certificate. See section "Importing X.509
       Certificates into NetX Secure" for more information. */
    nx_secure_x509_certificate_initialize(&trusted_certificate,
                                          trusted_cert_der,
                                          trusted_cert_der_len, NX_NULL, 0, NX_NULL,
                                          0,
                                          NX_SECURE_X509_KEY_TYPE_NONE);

    /* Add the certificate to the local store using a numeric ID. */
    nx_secure_dtls_session_trusted_certificate_add(&client_dtls_session,
                                                   &certificate, 1);

    /* Set up IP address of remote host. */
    server_ip.nxd_ip_version = NX_IP_VERSION_V4;
    server_ip.nxd_ip_address.v4 = IP_ADDRESS(192, 168, 1, 150);;


    /* Now we can start the DTLS session as normal. */
    status =  nx_secure_dtls_client_session_start(&client_dtls_session,
                                                  &udp_socket, &server_ip, 4443,
                                                  NX_IP_PERIODIC_RATE);

    if(status != NX_SUCCESS)
    {
        /* Error! */
        return(status);
    }

    /* Add data to send packet as usual for NX_PACKET and send to server.  */
    status = nx_secure_dtls_session_send(&client_dtls_session, &send_packet,
                                         &server_ip, 4443)

    /* Receive response from server. */
    status = nx_secure_dtls_session_receive(&client_dtls_session, &receive_packet,
                                            NX_IP_PERIODIC_RATE);

    /* Process response. */

    /* Shut down DTLS session. */
    status = nx_secure_dtls_session_end(&client_dtls_session,
                                        NX_IP_PERIODIC_RATE);

    /* Clean up. */
    status = nx_secure_dtls_session_delete(&client_dtls_session);

}
```

See Also

nx_secure_dtls_session_receive, nx_secure_dtls_session_send, nx_secure_dtls_session_create

nx_secure_dtls_packet_allocate

Allocate a packet for a NetX Secure DTLS Session

Prototype

```
UINT  nx_secure_dtls_packet_allocate(
                              NX_SECURE_DTLS_SESSION *session_ptr,
                NX_PACKET_POOL *pool_ptr,
                NX_PACKET **packet_ptr,
                              ULONG wait_option);
```

Description

This service allocates an NX_PACKET for the specified active DTLS session from the specified NX_PACKET_POOL. This service should be called by the application to allocate data packets to be sent over a DTLS connection. The DTLS session must be initialized before calling this service.

The allocated packet is properly initialized so that DTLS header and footer data may be added after the packet data is populated. The behavior is otherwise identical to *nx_packet_allocate*.

Parameters

session_ptr	Pointer to a DTLS Session instance.
pool_ptr	Pointer to an NX_PACKET_POOL from which to allocate the packet.
packet_ptr	Output pointer to the newly-allocated packet.
wait_option	Suspension option for packet allocation.

Return Values

NX_SUCCESS	(0x00)	Successful packet allocate.
NX_SECURE_TLS_ALLOCATE_PACKET_FAILED	(0x111)	Underlying packet allocation failed.
NX_SECURE_TLS_SESSION_UNINITIALIZED	(0x101)	The supplied DTLS session was not initialized.

Allowed From

Threads

Example

```
/* Allocate a new DTLS packet from the previously created packet pool and
previously initialized DTLS session.   */

status = nx_secure_dtls_packet_allocate(&dtls_session, &pool_0, &packet_ptr,
                                        NX_WAIT_FOREVER);

/* If status is NX_SUCCESS, the newly allocated packet pointer is found in the
variable packet_ptr.  */
```

See Also

nx_secure_x509_certificate_initialize, nx_secure_dtls_session_create, nx_secure_dtls_session_trusted_certificate_add, nx_secure_dtls_session_send, nx_secure_dtls_session_receive, nx_secure_dtls_session_end, nx_secure_dtls_session_delete

nx_secure_dtls_psk_add

Add a Pre-Shared Key to a NetX Secure DTLS Session

Prototype

```
UINT  nx_secure_dtls_psk_add(NX_SECURE_DTLS_SESSION *session_ptr,
                             UCHAR *pre_shared_key, UINT psk_length,
                             UCHAR *psk_identity, UINT
                             identity_length, UCHAR *hint, UINT
                             hint_length);
```

Description

This service adds a Pre-Shared Key (PSK), its identity string, and an identity hint to a DTLS Session control block. The PSK is used in place of a digital certificate when PSK ciphersuites are enabled and used.

Parameters

session_ptr	Pointer to a previously created DTLS Session instance.
pre_shared_key	The actual PSK value.
psk_length	The length of the PSK value.
psk_identity	A string used to identify this PSK value.
identity_length	The length of the PSK identity.
hint	A string used to indicate which group of PSKs to choose from on a TLS server.
hint_length	The length of the hint string.

Return Values

NX_SUCCESS	(0x00)	Successful addition of PSK.
NX_PTR_ERROR	(0x07)	Invalid DTLS session pointer.
NX_SECURE_TLS_NO_MORE_PSK_SPACE	(0x125)	Cannot add another PSK.

Allowed From

Threads

Example

```
/* PSK value.  */
UCHAR psk[] = { 0x1a, 0x2b, 0x3c, 0x4d };

/* Add PSK to DTLS session.  */
status =  nx_secure_dtls_psk_add(&dtls_session, psk, sizeof(psk), "psk_1", 4,
                                 "Client_identity", 15);


/* If status is NX_SUCCESS the PSK was successfully added.  */
```

See Also

nx_secure_dtls_server_psk_add, nx_secure_dtls_client_session_create

nx_secure_dtls_server_create

Create a NetX Secure DTLS Server

Prototype

```
UINT  nx_secure_dtls_server_create(
                NX_SECURE_DTLS_SERVER *server_ptr, NX_IP *ip_ptr,
                UINT port, ULONG timeout, VOID *session_buffer,
                UINT session_buffer_size,
                const NX_SECURE_TLS_CRYPTO *crypto_table,
                VOID *crypto_metadata_buffer, ULONG crypto_metadata_size,
                UCHAR *packet_reassembly_buffer,
                UINT packet_reassembly_buffer_size,
                UINT (*connect_notify)(
                             NX_SECURE_DTLS_SESSION *dtls_session,
                             NXD_ADDRESS *ip_address, UINT port),
                UINT (*receive_notify)(
                             NX_SECURE_DTLS_SESSION *dtls_session)));
```

Description

This service creates an instance of a DTLS server to handle incoming DTLS requests on a particular UDP port. Due to the fact that UDP is stateless, DTLS requests from multiple clients can come in on a single port while other DTLS sessions are active. Thus, the server is needed to maintain active sessions and properly route incoming messages to the proper handler.

The ip_ptr parameter points to an NX_IP instance to be used for the internal UDP socket associated with the DTLS Server (and stored in the NX_SECURE_DTLS_SERVER control block). The IP instance and port are used to define the UDP interface upon which the server is instantiated with the nx_secure_dtls_server_start service.

The session buffer parameter is used to hold the control blocks for all the possible simultaneous DTLS sessions for the DTLS server. It should be allocated with a size that is an even multiple of the size of the NX_SECURE_DTLS_SESSION control block structure.

To calculate the necessary metadata size, the API nx_secure_tls_metadata_size_calculate may be used.

The packet_reassembly_buffer parameter is used by DTLS to reassemble UDP datagrams into a complete DTLS record for the purposes of decryption and should be large enough to accommodate the largest expected DTLS record (16KB is the DTLS maximum record size but many applications don’t send that much data in a single record).

The connect_notify callback routine is invoked whenever a new DTLS client connects to the server. It is up to the application to then start the DTLS session using the service *nx_secure_dtls_server_session_start*. While the session may be started in the callback itself, it is recommended that the callback only be used to notify the application thread (or dedicated DTLS thread created by the application) of the connection as the callback is invoked by the IP thread which is used to process all lower-level network processing (e.g. UDP). This can be as simple as saving the DTLS session parameter (provided as a parameter to the callback) and invoking nx_secure_dtls_server_session_start in the other thread. The connect_notify callback should generally return NX_SUCCESS.

The receive_notify callback routine is invoked whenever a DTLS record is received that matches an existing established DTLS session (the remote host IP address and port are used to identify an existing session). This represents the “application data” that is encrypted and sent over DTLS. The application must call the service *nx_secure_dtls_session_receive* on the provided DTLS session to retrieve the received data. As with the connect_receive callback, it is recommended that the session be passed to another thread to handle the message processing as the callback is invoked from the IP thread. The receive_notify callback should generally return NX_SUCCESS.

Parameters

server_ptr	Pointer to a previously created DTLS Server instance.
ip_ptr	Pointer to an initialized NX_IP control block to use as the network interface for the DTLS server.
port	The local UDP port to which the DTLS server UDP socket is bound.
timeout	Timeout value to use for network operations.
session_buffer	Buffer space to contain control blocks for all instances of NX_SECURE_DTLS_SESSION assigned to this DTLS Server instance.
session_buffer_size	Size of the session buffer. This determines the number of DTLS sessions assigned to the DTLS Server.

crypto_table Pointer to a TLS/DTLS encryption table structure used for all cryptographic operations.

crypto_metadata_buffer
Buffer space for cryptographic operation calculations and state information.

crypto_metadata_size
Size of metadata buffer.

packet_reassembly_buffer
Buffer used by DTLS to reassemble UDP data into DTLS records for decryption.

packet_reassembly_buffer_size
Size of reassembly buffer. Generally should be greater than 16KB but may be smaller depending on application.

connect_notify Callback routine invoked whenever a remote DTLS Client attempts to connect to this DTLS server.

receive_notify Callback invoked whenever application data is received over an existing DTLS session.

Return Values

NX_SUCCESS	(0x00)	Successful creation of DTLS server.
NX_PTR_ERROR	(0x07)	Invalid pointer passed.
NX_INVALID_PARAMETERS	(0x4D)	Not enough buffer space for sessions, packet reassembly, or cryptography.

Allowed From

Threads

Example

```
#define DTLS_SERVER_SESSION (3)

/* Our DTLS Server instance. */
NX_SECURE_DTLS_SERVER dtls_server;

/* Allocate space for DTLS sessions in the DTLS server. */
UCHAR dtls_server_session_buffer[sizeof(NX_SECURE_DTLS_SESSION) * DTLS_SERVER_SESSIONS];

/* Flag used to indicate that a DTLS Client has connected. */
UINT connect_flag = 0;

/* Flag used to indicate application data reception. */
UINT receive_flag = 0;

/* Pointer to newly-connected DTLS session.
   NOTE: In practice this should be an array or list in case a new connection is
         attempted while a previous session is being started. */
NX_SECURE_DTLS_SESSION *new_dtls_session;

/* Pointer to session for application data receive. NOTE: Should be an array or list as
   with new_dtls_session */
NX_SECURE_DTLS_SESSION *receive_dtls_session;

/* Connect notify callback routine. */
UINT dtls_server_connect_notify(NX_SECURE_DTLS_SESSION *dtls_session, NXD_ADDRESS
*ip_address, UINT port)
{
    /* NOTE: proper inter-thread communication procedures (e.g. mutex handling)
             Omitted for clarity. */

    /* Notify application thread that a connection request has been received. */
    connect_flag = 1;
    new_dtls_session = dtls_session;

    return(NX_SUCCESS);
}

/* Receive notify callback routine invoked when DTLS application data is received
   on an existing DTLS server session. */
UINT dtls_server_receive_notify(NX_SECURE_DTLS_SESSION *dtls_session)
{
    /* Receive and process DTLS record.
       NOTE: Mutex handling omitted for clarity. */
    receive_flag = 1;
    receive_dtls_session = dtls_session;

    return(NX_SUCCESS);
}

/* Primary application thread for handling DTLS server operations. */
void dtls_server_thread(void)
{
NX_PACKET *send_packet;
NX_PACKET *receive_packet;
UINT status;

    /* Setup DTLS Server instance. */
    status = nx_secure_dtls_server_create(&dtls_server, &ip_instance, LOCAL_SERVER_PORT,
                          NX_IP_PERIODIC_RATE, dtls_server_session_buffer,
                          sizeof(dtls_server_session_buffer),
                          &tls_crypto_table, crypto_metadata_buffer,
                          sizeof(crypto_metadata_buffer), packet_buffer,
                          sizeof(packet_buffer),
                          dtls_server_connect_notify,
                          dtls_server_receive_notify);
```

```
    /* Initialize local server identity certificate with key and add to server. */
    status = nx_secure_x509_certificate_initialize(&certificate, device_cert_der,
                            device_cert_der_len, NX_NULL, 0,
                            device_cert_key_der, device_cert_key_der_len,
                            NX_SECURE_X509_KEY_TYPE_RSA_PKCS1_DER);

    /* Add local server identity certificate to DTLS server with ID of 1. */
    status = nx_secure_dtls_server_local_certificate_add(&dtls_server, &certificate, 1);


    /* Start server. */
    status = nx_secure_dtls_server_start(&dtls_server);

    /* Loop continuously to handle incoming data. */
    while(1)
    {
        /* Check for new connections. Muxtex handling omitted for clarity. */
        if(connect_flag)
        {
            /* We have a new connection attempt, start the DTLS session. */
            status = nx_secure_dtls_server_session_start(new_dtls_session,
                                                         NX_IP_PERIODIC_RATE);

        }

        /* Check for received application data. */
        if(receive_flag)
        {
            /* We have received data over a previously-established DTLS session.
               Mutex handling omitted for clarity. */
            Status = nx_secure_dtls_session_receive(receive_dtls_session, &receive_packet,
                                                    NX_IP_PERIODIC_RATE);

            /* Process received data… */

            /* Prepare and send response to client. */
            status = nx_secure_dtls_packet_allocate(receive_dtls_session, &packet_pool,
                                                    &send_packet, NX_IP_PERIODIC_RATE);

            /* Check for errors and prepare response data (e.g. nx_packet_data_append). */

            /* Send response to client. */
            status = nx_secure_dtls_server_session_send(receive_dtls_session,send_packet);

        }

        /* If not processing connections or received data, let the thread sleep. */
        if(!connect_flag && !receive_flag)
        {
            tx_thread_sleep(100);
        }
    }

    /* Server processing is done, stop the server instance from accepting requests. */
    status = nx_secure_dtls_server_stop(&dtls_server);

    /* If we exit the processing loop, clean up the server. */
    status = nx_secure_dtls_server_delete(&dtls_server);
}
```

See Also

nx_secure_dtls_server_start, nx_secure_dtls_server_delete, nx_secure_dtls_session_receive, nx_secure_dtls_server_session_send, nx_secure_dtls_server_session_start, nx_secure_dtls_server_session_stop, nx_secure_dtls_server_local_certificate_add

nx_secure_dtls_server_delete

Free up resources used by a NetX Secure DTLS Server

Prototype

```
UINT  nx_secure_dtls_server_delete(NX_SECURE_DTLS_SERVER *server_ptr);
```

Description

This service frees up the resources allocated to a DTLS Server instance, including the internal UDP socket used by the server.

Parameters

server_ptr	Pointer to a previously created DTLS Server instance.

Return Values

NX_SUCCESS	(0x00)	Successful deletion of server.
NX_PTR_ERROR	(0x07)	Invalid pointer passed.
NX_STILL_BOUND	(0x42)	UDP socket is still bound.

Allowed From

Threads

Example

```
#define DTLS_SERVER_SESSION (3)

/* Our DTLS Server instance. */
NX_SECURE_DTLS_SERVER dtls_server;

/* Allocate space for DTLS sessions in the DTLS server. */
UCHAR dtls_server_session_buffer[sizeof(NX_SECURE_DTLS_SESSION) * DTLS_SERVER_SESSIONS];

/* Flag used to indicate that a DTLS Client has connected. */
UINT connect_flag = 0;

/* Flag used to indicate application data reception. */
UINT receive_flag = 0;

/* Pointer to newly-connected DTLS session.
   NOTE: In practice this should be an array or list in case a new connection is
         attempted while a previous session is being started. */
NX_SECURE_DTLS_SESSION *new_dtls_session;

/* Pointer to session for application data receive. NOTE: Should be an array or list as
   with new_dtls_session */
```

```
NX_SECURE_DTLS_SESSION *receive_dtls_session;

/* Connect notify callback routine. */
UINT dtls_server_connect_notify(NX_SECURE_DTLS_SESSION *dtls_session, NXD_ADDRESS
*ip_address, UINT port)
{
    /* NOTE: proper inter-thread communication procedures (e.g. mutex handling)
             Omitted for clarity. */

    /* Notify application thread that a connection request has been received. */
    connect_flag = 1;
    new_dtls_session = dtls_session;

    return(NX_SUCCESS);
}

/* Receive notify callback routine invoked when DTLS application data is received
   on an existing DTLS server session. */
UINT dtls_server_receive_notify(NX_SECURE_DTLS_SESSION *dtls_session)
{
    /* Receive and process DTLS record.
       NOTE: Mutex handling omitted for clarity. */
    receive_flag = 1;
    receive_dtls_session = dtls_session;

    return(NX_SUCCESS);
}

/* Primary application thread for handling DTLS server operations. */
void dtls_server_thread(void)
{
NX_PACKET *send_packet;
NX_PACKET *receive_packet;
UINT status;

    /* Setup DTLS Server instance. */
    status = nx_secure_dtls_server_create(&dtls_server, &ip_instance, LOCAL_SERVER_PORT,
                                          NX_IP_PERIODIC_RATE, dtls_server_session_buffer,
                                          sizeof(dtls_server_session_buffer),
                                          &tls_crypto_table, crypto_metadata_buffer,
                                          sizeof(crypto_metadata_buffer), packet_buffer,
                                          sizeof(packet_buffer),
                                          dtls_server_connect_notify,
                                          dtls_server_receive_notify);

    /* Check for errors. */

    /* Initialize local server identity certificate with key and add to server. */
    status = nx_secure_x509_certificate_initialize(&certificate, device_cert_der,
                                                   device_cert_der_len, NX_NULL, 0,
                                                   device_cert_key_der,
                                                   device_cert_key_der_len,
                                                   NX_SECURE_X509_KEY_TYPE_RSA_PKCS1_DER);

    /* Add local server identity certificate to DTLS server with ID of 1. */
    status = nx_secure_dtls_server_local_certificate_add(&dtls_server, &certificate, 1);


    /* Start server. */
    status = nx_secure_dtls_server_start(&dtls_server);

    /* Loop continuously to handle incoming data. */
    while(1)
    {
        /* Check for new connections. Muxtex handling omitted for clarity. */
        if(connect_flag)
        {
            /* We have a new connection attempt, start the DTLS session. */
            status = nx_secure_dtls_server_session_start(new_dtls_session,
                                                         NX_IP_PERIODIC_RATE);
        }
```

```
        /* Check for received application data. */
        if(receive_flag)
        {
            /* We have received data over a previously-established DTLS session.
               Mutex handling omitted for clarity. */
            Status = nx_secure_dtls_session_receive(receive_dtls_session, &receive_packet,
                                    NX_IP_PERIODIC_RATE);

            /* Process received data… */

            /* Prepare and send response to client. */
            status = nx_secure_dtls_packet_allocate(receive_dtls_session, &packet_pool,
                                                    &send_packet, NX_IP_PERIODIC_RATE);

            /* Send response to client. */
            status = nx_secure_dtls_server_session_send(receive_dtls_session,send_packet);

        }

        /* If not processing connections or received data, let the thread sleep. */
        if(!connect_flag && !receive_flag)
        {
            tx_thread_sleep(100);
        }
    }

    /* If we exit the processing loop, clean up the server. */
    status = nx_secure_dtls_server_delete(&dtls_server);
}
```

See Also

nx_secure_dtls_server_start, nx_secure_dtls_server_create, nx_secure_dtls_session_receive, nx_secure_dtls_server_session_send, nx_secure_dtls_server_session_start

nx_secure_dtls_server_local_certificate_add

Add a local server identity certificate to a NetX Secure DTLS Server

Prototype

```
UINT  nx_secure_dtls_server_local_certificate_add(
                                NX_SECURE_DTLS_SERVER *server_ptr,
                                NX_SECURE_X509_CERT *certificate,
                                UINT cert_id);
```

Description

This service adds a local server identity certificate to a DTLS Server instance. At least one identity certificate is required for clients to connect to a DTLS server unless an alternate authentication mechanism (e.g. Pre-Shared Keys) is used.

The cert_id parameter is a numeric, non-zero identifier for the certificate. This enables the certificate to be easily removed or found in the event there are multiple identity certificates with the same X.509 Common Name present in the DTLS server store. See the NetX Secure TLS User Guide for more information about X.509 server certificates.

Parameters

server_ptr	Pointer to a previously created DTLS Server instance.
certificate	Pointer to a previously initialized X.509 certificate structure.
cert_id	Numeric non-zero unique identifier for this certificate in this DTLS server.

Return Values

NX_SUCCESS	(0x00)	Successful addition of certificate to DTLS server.
NX_PTR_ERROR	(0x07)	Invalid pointer passed.
NX_INVALID_PARAMETERS	(0x4D)	A certificate ID of 0 was passed in.

Allowed From

Threads

Example

*See reference for *nx_secure_dtls_server_create* for a more complete example.

```
/* Our DTLS Server instance. */
NX_SECURE_DTLS_SERVER dtls_server;

/* Certificate control block and data. */
NX_SECURE_X509_CERT certificate;
UCHAR certificate_der_data[] = { … };
UCHAR certificate_key_der_data[] = { … };

/* Primary application thread for handling DTLS server operations. */
void dtls_server_thread(void)
{
UINT status;

    /* Setup DTLS Server instance. */
    status = nx_secure_dtls_server_create(&dtls_server, &ip_instance, LOCAL_SERVER_PORT,
                                          NX_IP_PERIODIC_RATE, dtls_server_session_buffer,
                                          sizeof(dtls_server_session_buffer),
                                          &tls_crypto_table, crypto_metadata_buffer,
                                          sizeof(crypto_metadata_buffer), packet_buffer,
                                          sizeof(packet_buffer),
                                          dtls_server_connect_notify,
                                          dtls_server_receive_notify);

    /* Check for errors. */

    /* Initialize local server identity certificate with key and add to server. */
    status = nx_secure_x509_certificate_initialize(&certificate, certificate_der_data,
                                                   sizeof(certificate_der_data), NX_NULL, 0,
                                                   certificate_key_der_data,
                                                   sizeof(certificate_key_der_data),
                                                   NX_SECURE_X509_KEY_TYPE_RSA_PKCS1_DER);

    /* Add local server identity certificate to DTLS server with ID of 1. */
    status = nx_secure_dtls_server_local_certificate_add(&dtls_server, &certificate, 1);

    /* Start server. */
    status = nx_secure_dtls_server_start(&dtls_server);

    /* Loop continuously to handle incoming data. */
    while(1)
    {
        /* Process incoming requests and data. */
    }

}
```

See Also

nx_secure_dtls_server_start, nx_secure_dtls_server_create, nx_secure_dtls_session_receive, nx_secure_dtls_server_session_send, nx_secure_dtls_server_session_start, nx_secure_dtls_server_local_certificate_remove, nx_secure_x509_certificate_initialize

nx_secure_dtls_server_local_certificate_remove

Remove a local server identity certificate from a NetX Secure DTLS Server

Prototype

```
UINT  nx_secure_dtls_server_local_certificate_remove(
                                NX_SECURE_DTLS_SERVER *server_ptr,
                                UCHAR *common_name,
                                UINT common_name_length,
                                UINT cert_id);
```

Description

This service removes a local server identity certificate from a DTLS Server instance. At least one identity certificate is required for clients to connect to a DTLS server unless an alternate authentication mechanism (e.g. Pre-Shared Keys) is used.

The certificate to be removed can be identified either by its X.509 Common Name or by the numeric cert_id that was assigned in the call to *nx_secure_dtls_server_local_certificate_add*. The cert_id is only used to identify the certificate and is maintained by the application. If the Common Name is used instead of the numeric certificate identifier, the cert_id parameter should be set to 0.

Note that removing a certificate while a DTLS handshake is being processed will result in unexpected behavior. The service *nx_secure_dtls_server_stop* should be called before removing certificates.

Parameters

server_ptr	Pointer to a previously created DTLS Server instance.
common_name	X.509 CommonName of the certificate to remove. If this is used, pass cert_id as zero.
common_name_length	Length of common_name string in bytes.
cert_id	Numeric non-zero unique identifier for this certificate in this DTLS server. If this is used, pass NX_NULL for the common_name parameter.

Return Values

NX_SUCCESS	(0x00)	Successful removal of certificate from DTLS server.
NX_PTR_ERROR	(0x07)	Invalid pointer passed.
NX_SECURE_TLS_CERTIFICATE_NOT_FOUND	(0x119)	No certificate matching the cert_id or common_name was found in the given DTLS server.

Allowed From

Threads

Example

```
/* Our DTLS Server instance. */
NX_SECURE_DTLS_SERVER dtls_server;

/* Certificate control block and data. */
NX_SECURE_X509_CERT certificate;
UCHAR certificate_der_data[] = { … };
UCHAR certificate_key_der_data[] = { … };

/* Primary application thread for handling DTLS server operations. */
void dtls_server_thread(void)
{
UINT status;

    /* Setup DTLS Server instance. */
    status = nx_secure_dtls_server_create(&dtls_server, &ip_instance, LOCAL_SERVER_PORT,
                                          NX_IP_PERIODIC_RATE, dtls_server_session_buffer,
                                          sizeof(dtls_server_session_buffer),
                                          &tls_crypto_table, crypto_metadata_buffer,
                                          sizeof(crypto_metadata_buffer), packet_buffer,
                                          sizeof(packet_buffer),
                                          dtls_server_connect_notify,
                                          dtls_server_receive_notify);

    /* Check for errors. */

    /* Initialize local server identity certificate with key and add to server. */
    status = nx_secure_x509_certificate_initialize(&certificate, certificate_der_data,
                                                   sizeof(certificate_der_data), NX_NULL, 0,
                                                   certificate_key_der_data,
                                                   sizeof(certificate_key_der_data),
                                                   NX_SECURE_X509_KEY_TYPE_RSA_PKCS1_DER);

    /* Add local server identity certificate to DTLS server with ID of 1. */
    status = nx_secure_dtls_server_local_certificate_add(&dtls_server, &certificate, 1);

    /* Start server. */
    status = nx_secure_dtls_server_start(&dtls_server);

    /* Process client requests, etc… */

    /* Stop the server before removing a certificate. */
    Status = nx_secure_dtls_server_stop(&dtls_server);

    /* At some point in the future we decide to remove the certificate we added. We can
```

```
       use the certificate identifier we passed into the call to
       nx_secure_dtls_local_certificate_add (value = 1); */
    status = nx_secure_dtls_server_local_certificate_remove(&dtls_server, NX_NULL, 0, 1);

}
```

See Also

nx_secure_dtls_server_start, nx_secure_dtls_server_create, nx_secure_dtls_server_session_start, nx_secure_dtls_server_session_stop, nx_secure_dtls_server_local_certificate_add, nx_secure_x509_certificate_initialize

nx_secure_dtls_server_notify_set

Assign optional notification callback routines to a NetX Secure DTLS Server

Prototype

```
UINT  nx_secure_dtls_server_notify_set(
                NX_SECURE_DTLS_SERVER *server_ptr,
                          UINT (*disconnect_notify)(
                                    NX_SECURE_DTLS_SESSION *dtls_session),
                          UINT (*error_notify)(
                                    NX_SECURE_DTLS_SESSION *dtls_session,
                                    UINT error_code));
```

Description

This service can be used to add optional notification callback routines to a DTLS server. Either callback parameter may be passed as NX_NULL if only one callback is desired.

The disconnect_notify callback is invoked when a remote client ends a DTLS session. The dtls_session parameter is the session instance that was closed. The callback should generally return NX_SUCCESS.

The error_notify callback is invoked whenever a DTLS error or timeout occurs. The dtls_session parameter is the session instance for which the error occurred, and error_code is the numeric status code for the error that caused the issue (see Appendix A

NetX Secure Return/Error Codes for a list of error codes that may be returned). The callback should generally return NX_SUCCESS.

Parameters

server_ptr	Pointer to a previously created DTLS Server instance.
disconnect_notify	Callback routine invoked whenever a remote client host closes a DTLS session.
error_notify	Callback routine invoked whenever DTLS encounters an error or timeout.

Return Values

NX_SUCCESS	(0x00)	Successful assignment of callback routines.
NX_PTR_ERROR	(0x07)	Invalid pointer passed.

Allowed From

Threads

Example

```
/* Our DTLS Server instance. */
NX_SECURE_DTLS_SERVER dtls_server;

UINT disconnect_notify(NX_SECURE_DTLS_SESSION *dtls_session)
{
NXD_ADDRESS client_ip_addr;
UINT client_port;
UINT local_port;

    /* We have received a disconnection notice (CloseNotify message) from a
       remote client. Application can use the dtls_session parameter for any
       desired processing. */

    /* See what client disconnected by extracting its IP address and port.
       NOTE: depending on how the session ended, the client information may
             no longer be available. */
    status  = nx_secure_dtls_session_client_info_get(dtls_session, &ip_addr, &client_port,
                                                     &local_port);

    return(NX_SUCCESS);
}

UINT error_notify(NX_SECURE_DTLS_SESSION *dtls_session, UINT error_code)
{
    /* The DTLS server has encountered an error or timeout condition. We
       can use the error_code parameter to determine the error and we
       can insect the dtls_session for more information. */

    return(NX_SUCCESS);
}

/* Primary application thread for handling DTLS server operations. */
void dtls_server_thread(void)
{
UINT status;

    /* Setup DTLS Server instance. */
    status = nx_secure_dtls_server_create(&dtls_server, &ip_instance, LOCAL_SERVER_PORT,
                                          NX_IP_PERIODIC_RATE, dtls_server_session_buffer,
                                          sizeof(dtls_server_session_buffer),
                                          &tls_crypto_table, crypto_metadata_buffer,
                                          sizeof(crypto_metadata_buffer), packet_buffer,
                                          sizeof(packet_buffer),
                                          dtls_server_connect_notify,
                                          dtls_server_receive_notify);

    /* Check for errors. */

    /* Other setup (e.g. certificates) goes here … */

    status = nx_secure_dtls_server_notify_set(&dtls_server, disconnect_notify,
                                              error_notify);

    /* Start server. */
    status = nx_secure_dtls_server_start(&dtls_server);

    /* Process client requests, etc… */

}
```

See Also

nx_secure_dtls_server_start, nx_secure_dtls_server_create, nx_secure_dtls_server_session_start, nx_secure_dtls_server_session_stop

nx_secure_dtls_server_psk_add

Add a Pre-Shared Key to a NetX Secure DTLS Server

Prototype

```
UINT  nx_secure_dtls_server_psk_add(
                            NX_SECURE_DTLS_SERVER *server_ptr,
                            UCHAR *pre_shared_key, UINT psk_length,
                            UCHAR *psk_identity,
                            UINT identity_length, UCHAR *hint,
                            UINT hint_length);
```

Description

This service adds a Pre-Shared Key (PSK), its identity string, and an identity hint to a DTLS Server control block. The PSK is used in place of a digital certificate when PSK ciphersuites are enabled and used.

The PSK that is added is replicated across all the DTLS sessions assigned to the DTLS Server (via the session buffer given in the call to nx_secure_dtls_server_create).

Parameters

server_ptr	Pointer to a previously created DTLS Server instance.
pre_shared_key	The actual PSK value.
psk_length	The length of the PSK value.
psk_identity	A string used to identify this PSK value.
identity_length	The length of the PSK identity.
hint	A string used to indicate which group of PSKs to choose from on a TLS server.
hint_length	The length of the hint string.

Return Values

NX_SUCCESS	(0x00)	Successful addition of PSK.
NX_PTR_ERROR	(0x07)	Invalid DTLS server pointer.
NX_SECURE_TLS_NO_MORE_PSK_SPACE	(0x125)	Cannot add another PSK.

Allowed From

Threads

Example

```
/* PSK value.  */
UCHAR psk[] = { 0x1a, 0x2b, 0x3c, 0x4d };

/* Add PSK to DTLS session.  */
status =  nx_secure_dtls_server_psk_add(&dtls_server, psk, sizeof(psk), "psk_1",
                                        4, "Client_identity", 15);


/* If status is NX_SUCCESS the PSK was successfully added.  */
```

See Also

nx_secure_dtls_psk_add, nx_secure_dtls_server_create

nx_secure_dtls_server_session_send

Send data over a DTLS session established with a NetX Secure DTLS Server

Prototype

```
UINT  nx_secure_dtls_server_session_send(
                            NX_SECURE_DTLS_SESSION *session_ptr,
                            NX_PACKET *packet_ptr);
```

Description

This service sends a packet of data over an established DTLS Server session to a remote DTLS Client host. The session used is obtained in the receive_notify callback routine provided to nx_secure_dtls_session_create.

The data provided in the packet, which must be allocated using *nx_secure_dtls_packet_allocate*, is encrypted using the DTLS session cryptographic parameters and routines and then sent to the remote host over the DTLS Server internal UDP port to the attached client's IP address and port (stored in the DTLS Session).

Parameters

session_ptr	Pointer to a DTLS session instance obtained from the receive_notify callback routine provided by the application.
packet_ptr	Pointer to an NX_PACKET instance allocated previously and populated with application data.

Return Values

NX_SUCCESS	(0x00)	Successful creation of DTLS server.
NX_PTR_ERROR	(0x07)	Invalid pointer passed.
NX_SECURE_TLS_TCP_SEND_FAILED	(0x109)	An error occurred in the underlying UDP send operation.

Allowed From

Threads

Example

```
#define DTLS_SERVER_SESSION (3)

/* Our DTLS Server instance. */
NX_SECURE_DTLS_SERVER dtls_server;

/* Flag used to indicate application data reception. */
UINT receive_flag = 0;

/* Pointer to session for application data receive. NOTE: Should be an array or list as
   with new_dtls_session */
NX_SECURE_DTLS_SESSION *receive_dtls_session;


/* Receive notify callback routine invoked when DTLS application data is received
   on an existing DTLS server session. */
UINT dtls_server_receive_notify(NX_SECURE_DTLS_SESSION *dtls_session)
{
    /* Receive and process DTLS record.
       NOTE: Mutex handling omitted for clarity. */
    receive_flag = 1;
    receive_dtls_session = dtls_session;

    return(NX_SUCCESS);
}

/* Primary application thread for handling DTLS server operations. */
void dtls_server_thread(void)
{
NX_PACKET *send_packet;
NX_PACKET *receive_packet;
UINT status;

    /* Setup DTLS Server instance. */
    status = nx_secure_dtls_server_create(&dtls_server, &ip_instance, LOCAL_SERVER_PORT,
                          NX_IP_PERIODIC_RATE, dtls_server_session_buffer,
                          sizeof(dtls_server_session_buffer),
                          &tls_crypto_table, crypto_metadata_buffer,
                          sizeof(crypto_metadata_buffer), packet_buffer,
                          sizeof(packet_buffer),
                          dtls_server_connect_notify,
                          dtls_server_receive_notify);

    /* Check for errors. */

    /* Initialize local server identity certificate with key and add to server. */
    status = nx_secure_x509_certificate_initialize(&certificate, device_cert_der,
                                                   device_cert_der_len, NX_NULL, 0,
                                                   device_cert_key_der,
                                                   device_cert_key_der_len,
                                                   NX_SECURE_X509_KEY_TYPE_RSA_PKCS1_DER);

    /* Add local server identity certificate to DTLS server with ID of 1. */
    status = nx_secure_dtls_server_local_certificate_add(&dtls_server, &certificate, 1);


    /* Start server. */
    status = nx_secure_dtls_server_start(&dtls_server);

    /* Loop continuously to handle incoming data. */
    while(1)
    {
        /* Check for new connections. Muxtex handling omitted for clarity. */
        if(connect_flag)
        {
            /* We have a new connection attempt, start the DTLS session. */
            status = nx_secure_dtls_server_session_start(new_dtls_session,
                                       NX_IP_PERIODIC_RATE);
```

```
            /* Check for errors. */
        }

        /* Check for received application data. */
        if(receive_flag)
        {
            /* We have received data over a previously-established DTLS session.
               Mutex handling omitted for clarity. */
            Status = nx_secure_dtls_session_receive(receive_dtls_session, &receive_packet,
                                    NX_IP_PERIODIC_RATE);

            /* Process received data… */

            /* Prepare and send response to client. */
            status = nx_secure_dtls_packet_allocate(receive_dtls_session, &packet_pool,
                                    &send_packet, NX_IP_PERIODIC_RATE);

            /* Check for errors and prepare response data (e.g. nx_packet_data_append). */

            /* Send response to client. */
            status = nx_secure_dtls_server_session_send(receive_dtls_session,send_packet);

        }

        /* If not processing connections or received data, let the thread sleep. */
        if(!connect_flag && !receive_flag)
        {
            tx_thread_sleep(100);
        }
    }

    /* Server processing is done, stop the server instance from accepting requests. */
    status = nx_secure_dtls_server_stop(&dtls_server);

    /* If we exit the processing loop, clean up the server. */
    status = nx_secure_dtls_server_delete(&dtls_server);
}
```

See Also

nx_secure_dtls_server_start, nx_secure_dtls_server_delete, nx_secure_dtls_session_receive, nx_secure_dtls_server_session_create, nx_secure_dtls_server_session_start, nx_secure_dtls_server_session_stop, nx_secure_dtls_server_local_certificate_add

nx_secure_dtls_server_session_start

Start a DTLS Session from a NetX Secure DTLS Server

Prototype

```
UINT  nx_secure_dtls_server_session_start(
                NX_SECURE_DTLS_SESSION *session_ptr, UINT wait_option);
```

Description

This service starts a DTLS Server session by performing the server-side DTLS handshake when a remote DTLS Client has connected to the server and requested a DTLS connection.

The DTLS Session is obtained in the connect_notify callback routine provided to nx_secure_dtls_server_create.

Parameters

session_ptr	Pointer to a DTLS Session instance obtained from a DTLS Server connect_notify callback.
wait_option	ThreadX wait value to use for network operations.

Return Values

NX_SUCCESS	(0x00)	Successful creation of DTLS server.
NX_PTR_ERROR	(0x07)	Invalid pointer passed.
NX_SECURE_TLS_ALLOCATE_PACKET_FAILED	(0x111)	Could not allocate a DTLS handshake packet (packet pool empty).
NX_SECURE_TLS_INVALID_PACKET	(0x104)	Recevied data that was not a valid DTLS record.
NX_SECURE_TLS_HASH_MAC_VERIFY_FAILURE	(0x108)	A DTLS record failed to be properly hashed (encryption error).
NX_SECURE_TLS_PADDING_CHECK_FAILED	(0x12A)	Encryption padding check failure.
NX_SECURE_TLS_ALERT_RECEIVED	(0x114)	Recevied an alert from the remote host during the DTLS

handshake.

NX_SECURE_TLS_UNRECOGNIZED_MESSAGE_TYPE
(0x102) Received an unrecognized message during the DTLS handshake.

NX_SECURE_TLS_INCORRECT_MESSAGE_LENGTH
(0x10A) Recevied a DTLS record with an invalid length.

NX_SECURE_TLS_NO_SUPPORTED_CIPHERS
(0x10E) Received a ClientHello with no supported DTLS ciphersuites.

NX_SECURE_TLS_BAD_COMPRESSION_METHOD
(0x118) Recevied a ClientHello with a unknown compression method.

NX_SECURE_TLS_HANDSHAKE_FAILURE
(0x107) Generic (unspecified) handshake failure, usually due to problems with extension processing.

NX_SECURE_TLS_UNSUPPORTED_FEATURE
(0x130) A feature that is not yet supported was invoked during the DTLS handshake.

NX_SECURE_TLS_UNKNOWN_CIPHERSUITE
(0x105) An unknown ciphersuite was encountered (indicated internal cryptography error).

NX_SECURE_TLS_PROTOCOL_VERSION_CHANGED
(0x12E) Recevied a DTLS record with a mismatched DTLS version.

NX_SECURE_TLS_FINISHED_HASH_FAILURE
(0x115) Failed to validate the DTLS handshake hash, session is invalid.

NX_SECURE_TLS_TCP_SEND_FAILED
(0x109) Internal UDP send failed.

Allowed From

Threads

Example

```
#define DTLS_SERVER_SESSION (3)

/* Our DTLS Server instance. */
NX_SECURE_DTLS_SERVER dtls_server;

/* Allocate space for DTLS sessions in the DTLS server. */
UCHAR dtls_server_session_buffer[sizeof(NX_SECURE_DTLS_SESSION) * DTLS_SERVER_SESSIONS];

/* Flag used to indicate that a DTLS Client has connected. */
UINT connect_flag = 0;

/* Flag used to indicate application data reception. */
UINT receive_flag = 0;

/* Pointer to newly-connected DTLS session.
   NOTE: In practice this should be an array or list in case a new connection is
         attempted while a previous session is being started. */
NX_SECURE_DTLS_SESSION *new_dtls_session;

/* Pointer to session for application data receive. NOTE: Should be an array or list as
   with new_dtls_session */
NX_SECURE_DTLS_SESSION *receive_dtls_session;

/* Connect notify callback routine. */
UINT dtls_server_connect_notify(NX_SECURE_DTLS_SESSION *dtls_session, NXD_ADDRESS
*ip_address, UINT port)
{
    /* NOTE: proper inter-thread communication procedures (e.g. mutex handling)
             Omitted for clarity. */

    /* Notify application thread that a connection request has been received. */
    connect_flag = 1;
    new_dtls_session = dtls_session;

    return(NX_SUCCESS);
}

/* Receive notify callback routine invoked when DTLS application data is received
   on an existing DTLS server session. */
UINT dtls_server_receive_notify(NX_SECURE_DTLS_SESSION *dtls_session)
{
    /* Receive and process DTLS record.
       NOTE: Mutex handling omitted for clarity. */
    receive_flag = 1;
    receive_dtls_session = dtls_session;

    return(NX_SUCCESS);
}

/* Primary application thread for handling DTLS server operations. */
void dtls_server_thread(void)
{
NX_PACKET *send_packet;
NX_PACKET *receive_packet;
UINT status;

    /* Setup DTLS Server instance. */
    status = nx_secure_dtls_server_create(&dtls_server, &ip_instance, LOCAL_SERVER_PORT,
                                          NX_IP_PERIODIC_RATE, dtls_server_session_buffer,
                                          sizeof(dtls_server_session_buffer),
                                          &tls_crypto_table, crypto_metadata_buffer,
                                          sizeof(crypto_metadata_buffer), packet_buffer,
                                          sizeof(packet_buffer),
                                          dtls_server_connect_notify,
                                          dtls_server_receive_notify);
```

```
    /* Initialize local server identity certificate with key and add to server. */
    status = nx_secure_x509_certificate_initialize(&certificate, device_cert_der,
                                                   device_cert_der_len, NX_NULL, 0,
                                                   device_cert_key_der,
                                                   device_cert_key_der_len,
                                                   NX_SECURE_X509_KEY_TYPE_RSA_PKCS1_DER);

    /* Add local server identity certificate to DTLS server with ID of 1. */
    status = nx_secure_dtls_server_local_certificate_add(&dtls_server, &certificate, 1);


    /* Start server. */
    status = nx_secure_dtls_server_start(&dtls_server);

    /* Loop continuously to handle incoming data. */
    while(1)
    {
        /* Check for new connections. Muxtex handling omitted for clarity. */
        if(connect_flag)
        {
            /* We have a new connection attempt, start the DTLS session. */
            status = nx_secure_dtls_server_session_start(new_dtls_session,
                                                         NX_IP_PERIODIC_RATE);
        }

        /* Check for received application data. */
        if(receive_flag)
        {
            /* We have received data over a previously-established DTLS session.
               Mutex handling omitted for clarity. */
            Status = nx_secure_dtls_session_receive(receive_dtls_session, &receive_packet,
                                                    NX_IP_PERIODIC_RATE);

            /* Process received data… */

            /* Prepare and send response to client. */
            status = nx_secure_dtls_packet_allocate(receive_dtls_session, &packet_pool,
                                                    &send_packet, NX_IP_PERIODIC_RATE);

            /* Check for errors and prepare response data (e.g. nx_packet_data_append). */

            /* Send response to client. */
            status = nx_secure_dtls_server_session_send(receive_dtls_session,send_packet);

        }

        /* If not processing connections or received data, let the thread sleep. */
        if(!connect_flag && !receive_flag)
        {
            tx_thread_sleep(100);
        }
    }

    /* Server processing is done, stop the server instance from accepting requests. */
    status = nx_secure_dtls_server_stop(&dtls_server);

    /* If we exit the processing loop, clean up the server. */
    status = nx_secure_dtls_server_delete(&dtls_server);
}
```

See Also

nx_secure_dtls_server_start, nx_secure_dtls_server_delete, nx_secure_dtls_session_receive, nx_secure_dtls_server_session_send, nx_secure_dtls_server_session_create, nx_secure_dtls_server_session_stop, nx_secure_dtls_server_local_certificate_add

nx_secure_dtls_server_start

Start a NetX Secure DTLS Server instance listening on the configured UDP port

Prototype

```
UINT  nx_secure_dtls_server_start(
                NX_SECURE_DTLS_SERVER *server_ptr);
```

Description

This service starts a DTLS Server. After the call returns, the server is active and will begin processing incoming requests from DTLS clients. The server instance must have been configured with the service *nx_secure_dtls_server_create*.

Note that this service binds the internal DTLS Server UDP port to the configured local port so most issues encountered will have to do with UDP communications and network configuration.

Parameters

server_ptr	Pointer to a previously created DTLS Server instance.

Return Values

NX_SUCCESS	(0x00)	Successful start of server.
NX_PTR_ERROR	(0x07)	Invalid pointer passed.
NX_NOT_ENABLED	(0x14)	UDP not enabled.
NX_NO_FREE_PORTS	(0x45)	No available UDP ports.
NX_INVALID_PORT	(0x46)	Invalid UDP port.
NX_ALREADY_BOUND	(0x22)	UDP port already bound.
NX_PORT_UNAVAILABLE	(0x23)	UDP port not available for use.

Allowed From

Threads

Example

```
#define DTLS_SERVER_SESSION (3)

/* Our DTLS Server instance. */
NX_SECURE_DTLS_SERVER dtls_server;

/* Allocate space for DTLS sessions in the DTLS server. */
UCHAR dtls_server_session_buffer[sizeof(NX_SECURE_DTLS_SESSION) * DTLS_SERVER_SESSIONS];

/* Flag used to indicate that a DTLS Client has connected. */
UINT connect_flag = 0;

/* Flag used to indicate application data reception. */
UINT receive_flag = 0;

/* Pointer to newly-connected DTLS session.
   NOTE: In practice this should be an array or list in case a new connection is
         attempted while a previous session is being started. */
NX_SECURE_DTLS_SESSION *new_dtls_session;

/* Pointer to session for application data receive. NOTE: Should be an array or list as
   with new_dtls_session */
NX_SECURE_DTLS_SESSION *receive_dtls_session;

/* Connect notify callback routine. */
UINT dtls_server_connect_notify(NX_SECURE_DTLS_SESSION *dtls_session, NXD_ADDRESS
*ip_address, UINT port)
{
    /* NOTE: proper inter-thread communication procedures (e.g. mutex handling)
             Omitted for clarity. */

    /* Notify application thread that a connection request has been received. */
    connect_flag = 1;
    new_dtls_session = dtls_session;

    return(NX_SUCCESS);
}

/* Receive notify callback routine invoked when DTLS application data is received
   on an existing DTLS server session. */
UINT dtls_server_receive_notify(NX_SECURE_DTLS_SESSION *dtls_session)
{
    /* Receive and process DTLS record.
       NOTE: Mutex handling omitted for clarity. */
    receive_flag = 1;
    receive_dtls_session = dtls_session;

    return(NX_SUCCESS);
}

/* Primary application thread for handling DTLS server operations. */
void dtls_server_thread(void)
{
NX_PACKET *send_packet;
NX_PACKET *receive_packet;
UINT status;

    /* Setup DTLS Server instance. */
    status = nx_secure_dtls_server_create(&dtls_server, &ip_instance, LOCAL_SERVER_PORT,
                          NX_IP_PERIODIC_RATE, dtls_server_session_buffer,
                          sizeof(dtls_server_session_buffer),
                          &tls_crypto_table, crypto_metadata_buffer,
                          sizeof(crypto_metadata_buffer), packet_buffer,
                          sizeof(packet_buffer),
                          dtls_server_connect_notify,
                          dtls_server_receive_notify);
```

```
    /* Initialize local server identity certificate with key and add to server. */
    status = nx_secure_x509_certificate_initialize(&certificate, device_cert_der,
                            device_cert_der_len, NX_NULL, 0,
                            device_cert_key_der, device_cert_key_der_len,
                            NX_SECURE_X509_KEY_TYPE_RSA_PKCS1_DER);

    /* Add local server identity certificate to DTLS server with ID of 1. */
    status = nx_secure_dtls_server_local_certificate_add(&dtls_server, &certificate, 1);


    /* Start server. */
    status = nx_secure_dtls_server_start(&dtls_server);

    /* Loop continuously to handle incoming data. */
    while(1)
    {
        /* Check for new connections. Muxtex handling omitted for clarity. */
        if(connect_flag)
        {
            /* We have a new connection attempt, start the DTLS session. */
            status = nx_secure_dtls_server_session_start(new_dtls_session,
                                        NX_IP_PERIODIC_RATE);

            /* Check for errors. */
        }

        /* Check for received application data. */
        if(receive_flag)
        {
            /* We have received data over a previously-established DTLS session.
               Mutex handling omitted for clarity. */
            Status = nx_secure_dtls_session_receive(receive_dtls_session, &receive_packet,
                                                    NX_IP_PERIODIC_RATE);

            /* Process received data… */

            /* Prepare and send response to client. */
            status = nx_secure_dtls_packet_allocate(receive_dtls_session, &packet_pool,
                                                    &send_packet, NX_IP_PERIODIC_RATE);

            /* Check for errors and prepare response data (e.g. nx_packet_data_append). */

            /* Send response to client. */
            status = nx_secure_dtls_server_session_send(receive_dtls_session,send_packet);

        }

        /* If not processing connections or received data, let the thread sleep. */
        if(!connect_flag && !receive_flag)
        {
            tx_thread_sleep(100);
        }
    }

    /* If we exit the processing loop, clean up the server. */
    status = nx_secure_dtls_server_delete(&dtls_server);
}
```

See Also

nx_secure_dtls_server_stop, nx_secure_dtls_server_create, nx_secure_dtls_server_delete, nx_secure_dtls_session_receive, nx_secure_dtls_server_session_send, nx_secure_dtls_server_session_start

nx_secure_dtls_server_stop

Stop an active NetX Secure DTLS Server instance

Prototype

```
UINT  nx_secure_dtls_server_stop(NX_SECURE_DTLS_SERVER *server_ptr);
```

Description

This service stops a DTLS Server from listening on the configure UDP port and resets all the associated DTLS sessions, halting any in-progress DTLS communications.

Parameters

server_ptr	Pointer to an active DTLS Server instance.

Return Values

NX_SUCCESS	(0x00)	Successful stop of server.
NX_PTR_ERROR	(0x07)	Invalid pointer passed.

Allowed From

Threads

Example

```
#define DTLS_SERVER_SESSION (3)

/* Our DTLS Server instance. */
NX_SECURE_DTLS_SERVER dtls_server;

/* Allocate space for DTLS sessions in the DTLS server. */
UCHAR dtls_server_session_buffer[sizeof(NX_SECURE_DTLS_SESSION) * DTLS_SERVER_SESSIONS];

/* Flag used to indicate that a DTLS Client has connected. */
UINT connect_flag = 0;

/* Flag used to indicate application data reception. */
UINT receive_flag = 0;

/* Pointer to newly-connected DTLS session.
   NOTE: In practice this should be an array or list in case a new connection is
         attempted while a previous session is being started. */
NX_SECURE_DTLS_SESSION *new_dtls_session;

/* Pointer to session for application data receive. NOTE: Should be an array or list as
   with new_dtls_session */
NX_SECURE_DTLS_SESSION *receive_dtls_session;
```

```
/* Connect notify callback routine. */
UINT dtls_server_connect_notify(NX_SECURE_DTLS_SESSION *dtls_session, NXD_ADDRESS
*ip_address, UINT port)
{
    /* NOTE: proper inter-thread communication procedures (e.g. mutex handling)
             Omitted for clarity. */

    /* Notify application thread that a connection request has been received. */
    connect_flag = 1;
    new_dtls_session = dtls_session;

    return(NX_SUCCESS);
}

/* Receive notify callback routine invoked when DTLS application data is received
   on an existing DTLS server session. */
UINT dtls_server_receive_notify(NX_SECURE_DTLS_SESSION *dtls_session)
{
    /* Receive and process DTLS record.
       NOTE: Mutex handling omitted for clarity. */
    receive_flag = 1;
    receive_dtls_session = dtls_session;

    return(NX_SUCCESS);
}

/* Primary application thread for handling DTLS server operations. */
void dtls_server_thread(void)
{
NX_PACKET *send_packet;
NX_PACKET *receive_packet;
UINT status;

    /* Setup DTLS Server instance. */
    status = nx_secure_dtls_server_create(&dtls_server, &ip_instance, LOCAL_SERVER_PORT,
                          NX_IP_PERIODIC_RATE, dtls_server_session_buffer,
                          sizeof(dtls_server_session_buffer),
                          &tls_crypto_table, crypto_metadata_buffer,
                          sizeof(crypto_metadata_buffer), packet_buffer,
                          sizeof(packet_buffer),
                          dtls_server_connect_notify,
                          dtls_server_receive_notify);

    /* Check for errors. */

    /* Initialize local server identity certificate with key and add to server. */
    status = nx_secure_x509_certificate_initialize(&certificate, device_cert_der,
                             device_cert_der_len, NX_NULL, 0,
                             device_cert_key_der, device_cert_key_der_len,
                             NX_SECURE_X509_KEY_TYPE_RSA_PKCS1_DER);

    /* Add local server identity certificate to DTLS server with ID of 1. */
    status = nx_secure_dtls_server_local_certificate_add(&dtls_server, &certificate, 1);


    /* Start server. */
    status = nx_secure_dtls_server_start(&dtls_server);

    /* Loop continuously to handle incoming data. */
    while(1)
    {
        /* Check for new connections. Muxtex handling omitted for clarity. */
        if(connect_flag)
        {
            /* We have a new connection attempt, start the DTLS session. */
            status = nx_secure_dtls_server_session_start(new_dtls_session,
                                    NX_IP_PERIODIC_RATE);

            /* Check for errors. */
        }
```

```
        /* Check for received application data. */
        if(receive_flag)
        {
            /* We have received data over a previously-established DTLS session.
               Mutex handling omitted for clarity. */
            Status = nx_secure_dtls_session_receive(receive_dtls_session, &receive_packet,
                                    NX_IP_PERIODIC_RATE);

            /* Process received data… */

            /* Prepare and send response to client. */
            status = nx_secure_dtls_packet_allocate(receive_dtls_session, &packet_pool,
                                    &send_packet, NX_IP_PERIODIC_RATE);

            /* Check for errors and prepare response data (e.g. nx_packet_data_append). */

            /* Send response to client. */
            status = nx_secure_dtls_server_session_send(receive_dtls_session,send_packet);

        }

        /* If not processing connections or received data, let the thread sleep. */
        if(!connect_flag && !receive_flag)
        {
            tx_thread_sleep(100);
        }
    }

    /* We have exited the processing loop, stop the server. */
    status = nx_secure_dtls_server_stop(&dtls_server);

    /* If we exit the processing loop, clean up the server. */
    status = nx_secure_dtls_server_delete(&dtls_server);
}
```

See Also

nx_secure_dtls_server_start, nx_secure_dtls_server_create, nx_secure_dtls_server_delete, nx_secure_dtls_session_receive, nx_secure_dtls_server_session_send, nx_secure_dtls_server_session_start

nx_secure_dtls_server_trusted_certificate_add

Add a trusted CA certificate to a NetX Secure DTLS Server

Prototype

```
UINT  nx_secure_dtls_server_trusted_certificate_add(
                                NX_SECURE_DTLS_SERVER *server_ptr,
                                NX_SECURE_X509_CERT *certificate,
                                UINT cert_id);
```

Description

This service adds a trusted CA or intermediate CA certificate to a DTLS Server instance and assigned to all the internal DTLS server sessions. This is only necessary if X.509 Client certificate authentication is enabled using *nx_secure_dtls_server_x509_client_verify_configure*. The added certificate will be used to verify incoming Client X.509 certificates.

The cert_id parameter is a numeric, non-zero identifier for the certificate. This enables the certificate to be easily removed or found in the event there are multiple identity certificates with the same X.509 Common Name present in the DTLS server store. See the NetX Secure TLS User Guide for more information about X.509 server certificates.

Parameters

server_ptr	Pointer to a previously created DTLS Server instance.
certificate	Pointer to a previously initialized X.509 certificate structure.
cert_id	Numeric non-zero unique identifier for this certificate in this DTLS server.

Return Values

NX_SUCCESS	(0x00)	Successful addition of certificate to DTLS server.
NX_PTR_ERROR	(0x07)	Invalid pointer passed.
NX_INVALID_PARAMETERS	(0x4D)	A certificate ID of 0 was passed in.

Allowed From

Threads

Example

*See reference for *nx_secure_dtls_server_create* for a more complete example.

```
/* Our DTLS Server instance. */
NX_SECURE_DTLS_SERVER dtls_server;

/* Certificate control block and data. */
NX_SECURE_X509_CERT trusted_ca_certificate;
UCHAR certificate_der_data[] = { … };

/* Primary application thread for handling DTLS server operations. */
void dtls_server_thread(void)
{
UINT status;

    /* Setup DTLS Server instance. */
    status = nx_secure_dtls_server_create(&dtls_server, &ip_instance, LOCAL_SERVER_PORT,
                          NX_IP_PERIODIC_RATE, dtls_server_session_buffer,
                          sizeof(dtls_server_session_buffer),
                          &tls_crypto_table, crypto_metadata_buffer,
                          sizeof(crypto_metadata_buffer), packet_buffer,
                          sizeof(packet_buffer),
                          dtls_server_connect_notify,
                          dtls_server_receive_notify);

    /* Check for errors. */

    /* Initialize trusted certificate with key and add to server. */
    status = nx_secure_x509_certificate_initialize(&trusted_ca_certificate,
                                                   certificate_der_data,
                                                   sizeof(certificate_der_data),
                                                   NX_NULL, 0, NX_NULL, 0,
                                                   NX_SECURE_X509_KEY_TYPE_NONE);

    /* Add trusted CA certificate to DTLS server with ID of 1. */
    status = nx_secure_dtls_server_trusted_certificate_add(&dtls_server,
                                                  &trusted_ca_certificate, 1);

    /* Start server. */
    status = nx_secure_dtls_server_start(&dtls_server);

    /* Loop continuously to handle incoming data. */
    while(1)
    {
        /* Process incoming requests and data. */
    }

}
```

See Also

nx_secure_dtls_server_start, nx_secure_dtls_server_create, nx_secure_dtls_session_receive, nx_secure_dtls_server_session_send, nx_secure_dtls_server_session_start, nx_secure_dtls_server_local_certificate_add, nx_secure_dtls_server_trusted_certificate_remove, nx_secure_x509_certificate_initialize

nx_secure_dtls_server_trusted_certificate_remove

Remove a trusted CA certificate from a NetX Secure DTLS Server

Prototype

```
UINT  nx_secure_dtls_server_trusted_certificate_remove(
                                    NX_SECURE_DTLS_SERVER *server_ptr,
                                    UCHAR *common_name,
                                    UINT common_name_length,
                                    UINT cert_id);
```

Description

This service removes a trusted CA certificate from a DTLS Server instance. Trusted CA certificates are only necessary for a DTLS Server for which X.509 Client certificate verification has been enabled by calling *nx_secure_dtls_server_x509_client_verify_configure*.

The certificate to be removed can be identified either by its X.509 Common Name or by the numeric cert_id that was assigned in the call to *nx_secure_dtls_server_trusted_certificate_add*. The cert_id is only used to identify the certificate and is maintained by the application. If the Common Name is used instead of the numeric certificate identifier, the cert_id parameter should be set to 0.

Note that removing a certificate while a DTLS handshake is being processed may result in unexpected behavior. The service *nx_secure_dtls_server_stop* should be called before removing certificates.

Parameters

server_ptr Pointer to a previously created DTLS Server instance.

common_name X.509 CommonName of the certificate to remove. If this is used, pass cert_id as zero.

common_name_length
Length of common_name string in bytes.

cert_id Numeric non-zero unique identifier for this certificate in this DTLS server. If this is used, pass NX_NULL for the common_name parameter.

Return Values

NX_SUCCESS	(0x00)	Successful removal of certificate from DTLS server.
NX_PTR_ERROR	(0x07)	Invalid pointer passed.
NX_SECURE_TLS_CERTIFICATE_NOT_FOUND	(0x119)	No certificate matching the cert_id or common_name was found in the given DTLS server.

Allowed From

Threads

Example

```
/* Our DTLS Server instance. */
NX_SECURE_DTLS_SERVER dtls_server;

/* Certificate control block and data. */
NX_SECURE_X509_CERT trusted_ca_certificate;
UCHAR certificate_der_data[] = { … };

/* Primary application thread for handling DTLS server operations. */
void dtls_server_thread(void)
{
UINT status;

    /* Setup DTLS Server instance. */
    status = nx_secure_dtls_server_create(&dtls_server, &ip_instance, LOCAL_SERVER_PORT,
                                          NX_IP_PERIODIC_RATE, dtls_server_session_buffer,
                                          sizeof(dtls_server_session_buffer),
                                          &tls_crypto_table, crypto_metadata_buffer,
                                          sizeof(crypto_metadata_buffer), packet_buffer,
                                          sizeof(packet_buffer),
                                          dtls_server_connect_notify,
                                          dtls_server_receive_notify);

    /* Check for errors. */

    /* Initialize trusted certificate with key and add to server. */
    status = nx_secure_x509_certificate_initialize(&trusted_ca_certificate,
                                                   certificate_der_data,
                                                   sizeof(certificate_der_data), NX_NULL,
                                                   0, NX_NULL, 0,
                                                   NX_SECURE_X509_KEY_TYPE_NONE);

    /* Add local server identity certificate to DTLS server with ID of 1. */
    status = nx_secure_dtls_server_trusted_certificate_add(&dtls_server,
                                                           &trusted_ca_certificate, 1);

    /* Start server. */
    status = nx_secure_dtls_server_start(&dtls_server);

    /* Process client requests, etc… */

    /* Stop the server before removing a certificate. */
    Status = nx_secure_dtls_server_stop(&dtls_server);
```

```
/* At some point in the future we decide to remove the certificate we added. We can
       use the certificate identifier we passed into the call to
       nx_secure_dtls_trusted_certificate_add (value = 1); */
    status = nx_secure_dtls_server_trusted_certificate_remove(&dtls_server,
                                                              NX_NULL, 0, 1);

}
```

See Also

nx_secure_dtls_server_start, nx_secure_dtls_server_create, nx_secure_dtls_server_session_start, nx_secure_dtls_server_session_stop, nx_secure_dtls_server_trusted_certificate_add, nx_secure_x509_certificate_initialize

nx_secure_dtls_server_x509_client_verify_configure

Configure a NetX Secure DTLS Server to request and verify client certificates

Prototype

```
UINT nx_secure_dtls_server_x509_client_verify_configure(
                            NX_SECURE_DTLS_SERVER *server_ptr,
                            UINT certs_per_session,
                            UCHAR *certs_buffer, ULONG buffer_size);
```

Description

This service configures a DTLS Server to request and verify DTLS Client certificates. This optional feature is used when X.509 certificates are desired for client authentication in place of other mechanisms (e.g. a Pre-Shared Key).

When a DTLS Server is configured to verify client certificates using this service, at least one trusted CA certificate must be added to the server using nx_secure_dtls_server_trusted_certificate_add or the server will reject all incoming client connections because it will be unable to verify client certificates against the trusted store.

Upon calling this service, the DTLS Server instance will (once started) request client certificates as part of the DTLS handshake. Assuming the client is properly configured with an identity certificate (and associated certificate chain when applicable), the DTLS Server requires memory to be allocated to process the client certificate data. This memory is passed in as the *certs_buffer* parameter.

The certs_buffer must be sized to accommodate the largest expected certificate chain from a DTLS client, *times the number of DTLS server sessions*. The buffer is divided amongst the available sessions using the *certs_per_session* parameter which represents the maximum expected number of certificates in a Client certificate chain. The buffer also needs to provide space for the NX_SECURE_X509_CERT data structure which is used to parse the certificate data.

Calculating the proper buffer size can be done with the following formula:

```
buffer_size = (# of DTLS sessions in server) *
               (certs_per_session) *
               (    maximum expected certificate size +
                    sizeof(NX_SECURE_X509_CERT)  )
```

- The number of DTLS sessions is determined by the size of the session buffer passed into nx_secure_dtls_server_create.

- certs_per_session should be set to the maximum expected number of certificates in any client certificate chain.
- The maximum expected certificate size is dependent on the application, key sizes, and other factors but 2KB is generally sufficient.

Parameters

server_ptr	Pointer to a previously created DTLS Server instance.
certs_per_session	Number of certificates to allocate to each DTLS server session.
certs_buffer	Buffer space for incoming certificate data.
buffer_size	Size of the certificate buffer.

Return Values

NX_SUCCESS	(0x00)	Successful configuration of X.509 Client verification.
NX_PTR_ERROR	(0x07)	Invalid pointer passed.
NX_INVALID_PARAMETERS	(0x4D)	Invalid certificate store (DTLS server instance not initalized?).

Allowed From

Threads

Example

```
/* Configure number of sessions per server. */
#define DTLS_SERVER_SESSIONS 3

/* Define parameters for X.509 client verification. */
#define MAX_CERT_SIZE     (2000)     /* 2KB expected maximum certificate size. */
#define CERTS_PER_SESSION (3)        /* Assume maximum chain length of 3 certificates. */
#define CERT_BUFFER_SIZE  (DTLS_SERVER_SESSIONS * CERTS_PER_SESSION * \
                           (MAX_CERT_SIZE + sizeof(NX_SECURE_X509_CERT))

/* Define our incoming certificate buffer. */
UCHAR client_certs_buffer[CERT_BUFFER_SIZE];

/* Our DTLS Server instance. */
NX_SECURE_DTLS_SERVER dtls_server;

/* Certificate control block and data. */
NX_SECURE_X509_CERT trusted_ca_certificate;
UCHAR certificate_der_data[] = { … };

/* Primary application thread for handling DTLS server operations. */
```

```
void dtls_server_thread(void)
{
UINT status;

    /* Setup DTLS Server instance. */
    status = nx_secure_dtls_server_create(&dtls_server, &ip_instance, LOCAL_SERVER_PORT,
                          NX_IP_PERIODIC_RATE, dtls_server_session_buffer,
                          sizeof(dtls_server_session_buffer),
                          &tls_crypto_table, crypto_metadata_buffer,
                          sizeof(crypto_metadata_buffer), packet_buffer,
                          sizeof(packet_buffer),
                          dtls_server_connect_notify,
                          dtls_server_receive_notify);

    /* Check for errors. */

    /* Initialize trusted certificate with key and add to server. */
    status = nx_secure_x509_certificate_initialize(&trusted_ca_certificate,
                             certificate_der_data,
                             sizeof(certificate_der_data), NX_NULL, 0,
                             NX_NULL, 0, NX_SECURE_X509_KEY_TYPE_NONE);

    /* Add local server identity certificate to DTLS server with ID of 1. */
    status = nx_secure_dtls_server_trusted_certificate_add(&dtls_server,
                                                           &trusted_ca_certificate, 1);

    /* Configure client X.509 authentication and verification. */
    status = nx_secure_dtls_server_x509_client_verify_configure(&dtls_server,
                                                    CERTS_PER_SESSION,
                                                    client_certs_buffer,
                                                    sizeof(client_certs_buffer));

    /* Start server. */
    status = nx_secure_dtls_server_start(&dtls_server);

    /* Process client requests, etc… */
}
```

See Also

nx_secure_dtls_server_x509_client_verify_disable, nx_secure_dtls_server_start, nx_secure_dtls_server_create, nx_secure_dtls_server_session_start, nx_secure_dtls_server_session_stop, nx_secure_dtls_server_trusted_certificate_add, nx_secure_x509_certificate_initialize

nx_secure_dtls_server_x509_client_verify_disable

Disables client X.509 certificate verification for a NetX Secure DTLS Server

Prototype

```
UINT nx_secure_dtls_server_x509_client_verify_disable(
                            NX_SECURE_DTLS_SERVER *server_ptr);
```

Description

This service disables X.509 Client certificate verification on a DTLS Server. The service has no effect if X.509 Client certificate verification is not enabled.

Note that disabling client authentication on an active DTLS Server instance may result in unpredictable behavior. The nx_secure_dtls_server_stop service should be called before changing server state.

Parameters

server_ptr	Pointer to a previously created DTLS Server instance.

Return Values

NX_SUCCESS	(0x00)	Successful disabling of X.509 client authentication.
NX_PTR_ERROR	(0x07)	Invalid pointer passed.

Allowed From

Threads

Example

```
/* Configure number of sessions per server. */
#define DTLS_SERVER_SESSIONS 3

/* Define parameters for X.509 client verification. */
#define MAX_CERT_SIZE     (2000)      /* 2KB expected maximum certificate size. */
#define CERTS_PER_SESSION (3)         /* Assume maximum chain length of 3 certificates. */
#define CERT_BUFFER_SIZE  (DTLS_SERVER_SESSIONS * CERTS_PER_SESSION * \
                           (MAX_CERT_SIZE + sizeof(NX_SECURE_X509_CERT))

/* Define our incoming certificate buffer. */
UCHAR client_certs_buffer[CERT_BUFFER_SIZE];
```

```
/* Our DTLS Server instance. */
NX_SECURE_DTLS_SERVER dtls_server;

/* Certificate control block and data. */
NX_SECURE_X509_CERT trusted_ca_certificate;
UCHAR certificate_der_data[] = { … };

/* Primary application thread for handling DTLS server operations. */
void dtls_server_thread(void)
{
UINT status;

    /* Setup DTLS Server instance. */
    status = nx_secure_dtls_server_create(&dtls_server, &ip_instance, LOCAL_SERVER_PORT,
                          NX_IP_PERIODIC_RATE, dtls_server_session_buffer,
                          sizeof(dtls_server_session_buffer),
                          &tls_crypto_table, crypto_metadata_buffer,
                          sizeof(crypto_metadata_buffer), packet_buffer,
                          sizeof(packet_buffer),
                          dtls_server_connect_notify,
                          dtls_server_receive_notify);

    /* Check for errors. */

    /* Initialize trusted certificate with key and add to server. */
    status = nx_secure_x509_certificate_initialize(&trusted_ca_certificate,
                             certificate_der_data,
                             sizeof(certificate_der_data), NX_NULL, 0,
                             NX_NULL, 0, NX_SECURE_X509_KEY_TYPE_NONE);

    /* Add local server identity certificate to DTLS server with ID of 1. */
    status = nx_secure_dtls_server_trusted_certificate_add(&dtls_server,
                                                           &trusted_ca_certificate, 1);

    /* Configure client X.509 authentication and verification. */
    status = nx_secure_dtls_server_x509_client_verify_configure(&dtls_server,
                                                    CERTS_PER_SESSION,
                                                    client_certs_buffer,
                                                    sizeof(client_certs_buffer));

    /* Start server. */
    status = nx_secure_dtls_server_start(&dtls_server);

    /* Process client requests, etc… */

    /* Stop the server before changing state. */
    status = nx_secure_dtls_server_stop(&dtls_server);

    /* Disable X.509 authentication and verification. */
    status = nx_secure_dtls_server_x509_client_verify_disable(&dtls_server);

}
```

See Also

nx_secure_dtls_server_x509_client_verify_configure, nx_secure_dtls_server_start, nx_secure_dtls_server_create, nx_secure_dtls_server_session_start, nx_secure_dtls_server_session_stop, nx_secure_dtls_server_trusted_certificate_add, nx_secure_x509_certificate_initialize

nx_secure_dtls_session_client_info_get

Get remote client information from a DTLS Server session

Prototype

```
UINT  nx_secure_dtls_session_client_info_get(
                                NX_SECURE_DTLS_SESSION *session_ptr,
                                NXD_ADDRESS *client_ip_address,
                                UINT *client_port, UINT *local_port);
```

Description

This service returns the network information about a DTLS Client that is connected to a particular DTLS Server session. The information returned consists of the remote client's IP address and UDP port, as well as the local server port to which the client is connected.

In general, the DTLS session instance will be the one obtained in the invocation of one of the DTLS notification callback routines (e.g. the connect_notify or receive_notify callbacks passed into nx_secure_dtls_server_create).

Parameters

session_ptr	Pointer to an active DTLS server session instance.

Return Values

NX_SUCCESS	(0x00)	Successful deletion of server.
NX_PTR_ERROR	(0x07)	Invalid pointer passed.
NX_INVALID_SOCKET	(0x13)	The associated UDP socket is not valid (session not initialized?).
NX_NOT_CONNECTED	(0x38)	UDP socket is not connected – client connection dropped or not yet established.

Allowed From

Threads

Example

```
#define DTLS_SERVER_SESSION (3)

/* Our DTLS Server instance. */
```

```
NX_SECURE_DTLS_SERVER dtls_server;

/* Allocate space for DTLS sessions in the DTLS server. */
UCHAR dtls_server_session_buffer[sizeof(NX_SECURE_DTLS_SESSION) * DTLS_SERVER_SESSIONS];

/* Flag used to indicate that a DTLS Client has connected. */
UINT connect_flag = 0;

/* Flag used to indicate application data reception. */
UINT receive_flag = 0;

/* Pointer to newly-connected DTLS session.
   NOTE: In practice this should be an array or list in case a new connection is
         attempted while a previous session is being started. */
NX_SECURE_DTLS_SESSION *new_dtls_session;

/* Pointer to session for application data receive. NOTE: Should be an array or list as
   with new_dtls_session */
NX_SECURE_DTLS_SESSION *receive_dtls_session;

/* Connect notify callback routine. */
UINT dtls_server_connect_notify(NX_SECURE_DTLS_SESSION *dtls_session,
                                NXD_ADDRESS *ip_address, UINT port)
{
    /* NOTE: proper inter-thread communication procedures (e.g. mutex handling)
             Omitted for clarity. */

    /* Notify application thread that a connection request has been received. */
    connect_flag = 1;
    new_dtls_session = dtls_session;

    return(NX_SUCCESS);
}

/* Receive notify callback routine invoked when DTLS application data is received
   on an existing DTLS server session. */
UINT dtls_server_receive_notify(NX_SECURE_DTLS_SESSION *dtls_session)
{
NXD_ADDRESS client_ip;
UINT client_port, server_port;

    /* Get DTLS client information which can be used in filtering or associating
       the DTLS session with application data (e.g. a session pointer table). */
    status = nx_secure_dtls_session_client_info_get(dtls_session, &client_ip,
                                                    &client_port, &server_port);

    /* Receive and process DTLS record.
       NOTE: Mutex handling omitted for clarity. */
    receive_flag = 1;
    receive_dtls_session = dtls_session;

    return(NX_SUCCESS);
}

/* Primary application thread for handling DTLS server operations. */
void dtls_server_thread(void)
{
NX_PACKET *send_packet;
NX_PACKET *receive_packet;
UINT status;

    /* Setup DTLS Server instance. */
    status = nx_secure_dtls_server_create(&dtls_server, &ip_instance, LOCAL_SERVER_PORT,
                                          NX_IP_PERIODIC_RATE, dtls_server_session_buffer,
                                          sizeof(dtls_server_session_buffer),
                                          &tls_crypto_table, crypto_metadata_buffer,
                                          sizeof(crypto_metadata_buffer), packet_buffer,
                                          sizeof(packet_buffer),
                                          dtls_server_connect_notify,
                                          dtls_server_receive_notify);
```

```
    /* Check for errors. */

    /* Initialize local server identity certificate with key and add to server. */
    status = nx_secure_x509_certificate_initialize(&certificate, device_cert_der,
                              device_cert_der_len, NX_NULL, 0,
                              device_cert_key_der, device_cert_key_der_len,
                              NX_SECURE_X509_KEY_TYPE_RSA_PKCS1_DER);

    /* Add local server identity certificate to DTLS server with ID of 1. */
    status = nx_secure_dtls_server_local_certificate_add(&dtls_server, &certificate, 1);


    /* Start server. */
    status = nx_secure_dtls_server_start(&dtls_server);

    /* Loop continuously to handle incoming data. */
    while(1)
    {
        /* Check for new connections. Muxtex handling omitted for clarity. */
        if(connect_flag)
        {
            /* We have a new connection attempt, start the DTLS session. */
            status = nx_secure_dtls_server_session_start(new_dtls_session,
                                    NX_IP_PERIODIC_RATE);

            /* Check for errors. */
        }

        /* Check for received application data. */
        if(receive_flag)
        {
            /* We have received data over a previously-established DTLS session.
               Mutex handling omitted for clarity. */
            Status = nx_secure_dtls_session_receive(receive_dtls_session, &receive_packet,
                                    NX_IP_PERIODIC_RATE);

            /* Process received data… */

            /* Prepare and send response to client. */
            status = nx_secure_dtls_packet_allocate(receive_dtls_session, &packet_pool,
                                    &send_packet, NX_IP_PERIODIC_RATE);

            /* Check for errors and prepare response data (e.g. nx_packet_data_append). */

            /* Send response to client. */
            status = nx_secure_dtls_server_session_send(receive_dtls_session,send_packet);

        }

        /* If not processing connections or received data, let the thread sleep. */
        if(!connect_flag && !receive_flag)
        {
            tx_thread_sleep(100);
        }
    }

    /* If we exit the processing loop, clean up the server. */
    status = nx_secure_dtls_server_delete(&dtls_server);
}
```

See Also

nx_secure_dtls_server_start, nx_secure_dtls_server_stop, nx_secure_dtls_server_create, nx_secure_dtls_server_delete, nx_secure_dtls_session_receive, nx_secure_dtls_server_session_send, nx_secure_dtls_server_session_start

nx_secure_dtls_session_create

Create and configure a NetX Secure DTLS Session

Prototype

```
UINT nx_secure_dtls_session_create(
                        NX_SECURE_DTLS_SESSION *dtls_session,
                        const NX_SECURE_TLS_CRYPTO *crypto_table,
                        VOID *metadata_buffer, ULONG metadata_size,
                        UCHAR *packet_reassembly_buffer,
                        UINT packet_reassembly_buffer_size,
                        UINT certs_number,
                        UCHAR *remote_certificate_buffer,
                        ULONG remote_certificate_buffer_size));
```

Description

This service creates and configures a DTLS session. Generally, this will be used to create DTLS Client sessions as DTLS Server sessions are managed with the DTLS Server mechanism (see *nx_secure_dtls_server_create*), but there may be instances where an application needs to create a single stand-alone DTLS Server session instance in which case this service may be used[7].

The parameters configure the information and memory allocation needed to instantiate a DTLS session. The crypto_table parameter is a TLS table containing all of the cryptographic routines needed for TLS/DTLS encryption and authentication. The metadata_buffer is used for encryption caclulations (see nx_secure_tls_metadata_size_calculate in the NetX Secure TLS User Guide), and the packet_reassembly_buffer is used to reassemble UDP datagrams into a complete DTLS record for decryption.

The certs_number and remote_certificate_buffer are required for DTLS Clients which need space to store and process the incoming DTLS Server certificate chain. The buffer must be able to accommodate the maximum expected size of the certificate chain for any server to which it will connect. The buffer is divided up by the number of expected certificates (certs_number parameter) and must also be large enough to hold one NX_SECURE_X509_CERT structure per certificate. The buffer size can be determined using the following formula:

```
remote_certificate_buffer_size = (certs_number) *
                (maximum cert size + sizeof(NX_SECURE_X509_CERT))
```

[7] Creating DTLS Server sessions with this routine is not recommended and comes with some limitations. The primary issue is that the session will not handle additional client connections gracefully – since UDP is connectionless a second client can legally send data to the server's UDP port when a previous DTLS session is still active which would cause the server session to end with an error.

- certs_number is the expected maximum number of certificates in the server's certificate chain
- Maximum certificate size is dependent on the size of keys being used and the information in the certificate, but 2KB is generally sufficient.

Parameters

dtls_session Pointer to an uninitialized DTLS Session structure.

crypto_table Pointer to a TLS/DTLS encryption table structure used for all cryptographic operations.

crypto_metadata_buffer
Buffer space for cryptographic operation calculations and state information.

crypto_metadata_size
Size of metadata buffer.

packet_reassembly_buffer
Buffer used by DTLS to reassemble UDP data into DTLS records for decryption.

packet_reassembly_buffer_size
Size of reassembly buffer. Generally should be greater than 16KB but may be smaller depending on application.

certs_number Maximum expected number of certificates in the remote server's certificate chain.

remote_certificate_buffer
Buffer space for incoming certificate data.

remote_certificate_buffer_size
Size of the certificate buffer.

Return Values

NX_SUCCESS	(0x00)	Successful creation of session.
NX_PTR_ERROR	(0x07)	Invalid session or buffer pointer.
NX_INVALID_PARAMETERS	(0x4D)	Not enough buffer space for packet reassembly, certificates, or cryptography.

Allowed From

Threads

Example

```
/* Sockets, sessions, certificates defined in global static space to preserve
   application stack. */
NX_SECURE_DTLS_SESSION client_dtls_session;

/* Trusted certificate structure and raw data. */
NX_SECURE_X509_CERT trusted_cert;
const UCHAR trusted_cert_der = { … };

/* Cryptography routines and crypto work buffers. */
extern const NX_SECURE_TLS_CRYPTO nx_crypto_tls_ciphers;
UCHAR crypto_metadata[9000];

/* Packet reassembly buffer for decryption. */
UCHAR packet_buffer[4000];

/* Remote certificate buffer for incoming certificates. */
#define REMOTE_CERT_SIZE (sizeof(NX_SECURE_X509_CERT) + 2000)
#define REMOTE_CERT_NUMBER  (3)
UCHAR remote_certs_buffer[REMOTE_CERT_SIZE * REMOTE_CERT_NUMBER];

/* Application thread where TLS session is started. */
void application_thread()
{
NXD_ADDRESS server_ip;

    /* Create a TLS session for our socket. Ciphers and metadata defined
       elsewhere. See nx_secure_tls_session_create reference for more
       information. */
    status =  nx_secure_dtls_session_create(&client_dtls_session,
                                            &nx_crypto_tls_ciphers,
                                            crypto_metadata,
                                            sizeof(crypto_metadata),
                                             packet_buffer,
                                             sizeof(packet_buffer),
                                             REMOTE_CERT_NUMBER,
                                             remote_certs_buffer,
                                             sizeof(remote_certs_buffer));

    /* Check for error.  */
    if(status)
    {
        printf("Error in function nx_secure_dtls_session_create: 0x%x\n", status);
    }

    /* Initialize our trusted certificate. See section “Importing X.509
       Certificates into NetX Secure” for more information. */
    nx_secure_x509_certificate_initialize(&trusted_certificate,
                                          trusted_cert_der,
```

```
                                        trusted_cert_der_len, NX_NULL, 0, NX_NULL,
                                        0,
                                        NX_SECURE_X509_KEY_TYPE_NONE);

    /* Add the certificate to the local store using a numeric ID. */
    nx_secure_dtls_session_trusted_certificate_add(&client_dtls_session,
                                                   &certificate, 1);

    /* Set up IP address of remote host. */
    server_ip.nxd_ip_version = NX_IP_VERSION_V4;
    server_ip.nxd_ip_address.v4 = IP_ADDRESS(192, 168, 1, 150);;


    /* Now we can start the DTLS session as normal. */
    status =  nx_secure_dtls_client_session_start(&client_dtls_session,
                                                  &udp_socket, &server_ip, 4443,
                                                  NX_IP_PERIODIC_RATE);

 if(status != NX_SUCCESS)
    {
 /* Error! */
 return(status);
 }

/* Add data to send packet as usual for NX_PACKET and send to server.  */
status = nx_secure_dtls_session_send(&client_dtls_session, &send_packet,
                                     &server_ip, 4443)

   /* Receive response from server. */
   status = nx_secure_dtls_session_receive(&client_dtls_session, &receive_packet,
                                           NX_IP_PERIODIC_RATE);

/* Process response. */

/* Shut down DTLS session. */
    status = nx_secure_dtls_session_end(&client_dtls_session, NX_IP_PERIODIC_RATE);

/* Clean up. */
    status = nx_secure_dtls_session_delete(&client_dtls_session);

 }
```

See Also

nx_secure_dtls_client_session_start, nx_secure_dtls_session_receive, nx_secure_dtls_session_send

nx_secure_dtls_session_delete

Free up resources used by a NetX Secure DTLS Session

Prototype

```
UINT nx_secure_dtls_session_delete(
                        NX_SECURE_DTLS_SESSION *dtls_session);
```

Description

This service deletes a DTLS session, freeing up any resources that were allocated when it was created.

Parameters

dtls_session	Pointer to a DTLS Session structure that was initialized previously.

Return Values

NX_SUCCESS	(0x00)	Successful deletion of session.
NX_PTR_ERROR	(0x07)	Invalid session or buffer pointer.

Allowed From

Threads

Example

```
/* Sockets, sessions, certificates defined in global static space to preserve
   application stack. */
NX_SECURE_DTLS_SESSION client_dtls_session;

/* Trusted certificate structure and raw data. */
NX_SECURE_X509_CERT trusted_cert;
const UCHAR trusted_cert_der = { … };

/* Cryptography routines and crypto work buffers. */
extern const NX_SECURE_TLS_CRYPTO nx_crypto_tls_ciphers;
UCHAR crypto_metadata[9000];

/* Packet reassembly buffer for decryption. */
UCHAR packet_buffer[4000];

/* Remote certificate buffer for incoming certificates. */
#define REMOTE_CERT_SIZE (sizeof(NX_SECURE_X509_CERT) + 2000)
#define REMOTE_CERT_NUMBER  (3)
UCHAR remote_certs_buffer[REMOTE_CERT_SIZE * REMOTE_CERT_NUMBER];

/* Application thread where TLS session is started. */
void application_thread()
{
NXD_ADDRESS server_ip;
```

```
        /* Create a TLS session for our socket. Ciphers and metadata defined
           elsewhere. See nx_secure_tls_session_create reference for more
           information. */
        status =  nx_secure_dtls_session_create(&client_dtls_session,
                                                &nx_crypto_tls_ciphers,
                                                crypto_metadata,
                                                sizeof(crypto_metadata),
                                                packet_buffer,
                                                sizeof(packet_buffer),
                                                REMOTE_CERT_NUMBER,
                                                remote_certs_buffer,
                                                sizeof(remote_certs_buffer));

        /* Check for error.  */
        if(status)
        {
            printf("Error in function nx_secure_dtls_session_create: 0x%x\n", status);
        }

        /* Initialize our trusted certificate. See section "Importing X.509
           Certificates into NetX Secure" for more information. */
        nx_secure_x509_certificate_initialize(&trusted_certificate,
                                              trusted_cert_der,
                                              trusted_cert_der_len, NX_NULL, 0, NX_NULL,
                                              0,
                                              NX_SECURE_X509_KEY_TYPE_NONE);

        /* Add the certificate to the local store using a numeric ID. */
        nx_secure_dtls_session_trusted_certificate_add(&client_dtls_session,
                                                       &certificate, 1);

        /* Set up IP address of remote host. */
        server_ip.nxd_ip_version = NX_IP_VERSION_V4;
        server_ip.nxd_ip_address.v4 = IP_ADDRESS(192, 168, 1, 150);;


        /* Now we can start the DTLS session as normal. */
        status =  nx_secure_dtls_client_session_start(&client_dtls_session,
                                                      &udp_socket, &server_ip, 4443,
                                                      NX_IP_PERIODIC_RATE);

    if(status != NX_SUCCESS)
       {
    /* Error! */
    return(status);
    }

   /* Add data to send packet as usual for NX_PACKET and send to server.  */
   status = nx_secure_dtls_session_send(&client_dtls_session, &send_packet,
                                        &server_ip, 4443)

       /* Receive response from server. */
       status = nx_secure_dtls_session_receive(&client_dtls_session, &receive_packet,
                                               NX_IP_PERIODIC_RATE);

  /* Process response. */

  /* Shut down DTLS session. */
       status = nx_secure_dtls_session_end(&client_dtls_session, NX_IP_PERIODIC_RATE);

  /* Clean up. */
       status = nx_secure_dtls_session_delete(&client_dtls_session);

    }
```

See Also

nx_secure_dtls_client_session_start, nx_secure_dtls_session_receive, nx_secure_dtls_session_send, nx_secure_dtls_session_delete

nx_secure_dtls_session_end

Shut down an active NetX Secure DTLS Session

Prototype

```
UINT nx_secure_dtls_session_end(NX_SECURE_DTLS_SESSION *dtls_session,
                                UINT wait_option);
```

Description

This service ends an active DTLS session by sending a TLS/DTLS CloseNotify alert to the remote host. The IP address and port used are those used in the previous call to nx_secure_dtls_session_send.

Parameters

dtls_session	Pointer to a DTLS Session structure that was initialized previously.
wait_option	ThreadX wait value to use for network operations.

Return Values

NX_SUCCESS	(0x00)	Successful deletion of session.
NX_PTR_ERROR	(0x07)	Invalid session or buffer pointer.
NX_SECURE_TLS_ALLOCATE_PACKET_FAILED	(0x111)	Could not allocate packet(s) for CloseNotify alert.
NX_SECURE_TLS_UNKNOWN_CIPHERSUITE	(0x105)	Likely internal error – cryptographic routine not recognized.
NX_SECURE_TLS_TCP_SEND_FAILED	(0x109)	Underlying UDP send failed.
NX_IP_ADDRESS_ERROR	(0x21)	Error with remote host IP address.
NX_NOT_BOUND	(0x24)	Underlying UDP socket not bound to port.
NX_INVALID_PORT	(0x46)	Invalid UDP port.
NX_PORT_UNAVAILABLE	(0x23)	UDP port not available for use.

Allowed From

Threads

Example

```
/* Sockets, sessions, certificates defined in global static space to preserve
   application stack. */
NX_SECURE_DTLS_SESSION client_dtls_session;

/* Trusted certificate structure and raw data. */
NX_SECURE_X509_CERT trusted_cert;
const UCHAR trusted_cert_der = { … };

/* Cryptography routines and crypto work buffers. */
extern const NX_SECURE_TLS_CRYPTO nx_crypto_tls_ciphers;
UCHAR crypto_metadata[9000];

/* Packet reassembly buffer for decryption. */
UCHAR packet_buffer[4000];

/* Remote certificate buffer for incoming certificates. */
#define REMOTE_CERT_SIZE (sizeof(NX_SECURE_X509_CERT) + 2000)
#define REMOTE_CERT_NUMBER  (3)
UCHAR remote_certs_buffer[REMOTE_CERT_SIZE * REMOTE_CERT_NUMBER];

/* Application thread where TLS session is started. */
void application_thread()
{
NXD_ADDRESS server_ip;

    /* Create a TLS session for our socket. Ciphers and metadata defined
        elsewhere. See nx_secure_tls_session_create reference for more
        information. */
    status =  nx_secure_dtls_session_create(&client_dtls_session,
                                            &nx_crypto_tls_ciphers,
                                            crypto_metadata,
                                            sizeof(crypto_metadata),
                                             packet_buffer,
                                             sizeof(packet_buffer),
                                             REMOTE_CERT_NUMBER,
                                             remote_certs_buffer,
                                             sizeof(remote_certs_buffer));

    /* Check for error.  */
    if(status)
    {
        printf("Error in function nx_secure_dtls_session_create: 0x%x\n", status);
    }

    /* Initialize our trusted certificate. See section "Importing X.509
        Certificates into NetX Secure" for more information. */
    nx_secure_x509_certificate_initialize(&trusted_certificate,
                                          trusted_cert_der,
                                          trusted_cert_der_len, NX_NULL, 0, NX_NULL, 0,
                                          NX_SECURE_X509_KEY_TYPE_NONE);

    /* Add the certificate to the local store using a numeric ID. */
    nx_secure_dtls_session_trusted_certificate_add(&client_dtls_session,
                                                   &certificate, 1);

    /* Set up IP address of remote host. */
    server_ip.nxd_ip_version = NX_IP_VERSION_V4;
    server_ip.nxd_ip_address.v4 = IP_ADDRESS(192, 168, 1, 150);;


    /* Now we can start the DTLS session as normal. */
    status =  nx_secure_dtls_client_session_start(&client_dtls_session,
                                                  &udp_socket, &server_ip, 4443,
                                                  NX_IP_PERIODIC_RATE);

    if(status != NX_SUCCESS)
    {
        /* Error! */
```

```
        return(status);
     }

   /* Add data to send packet as usual for NX_PACKET and send to server.  */
   status = nx_secure_dtls_session_send(&client_dtls_session, &send_packet,
                                        &server_ip, 4443)

    /* Receive response from server. */
      status = nx_secure_dtls_session_receive(&client_dtls_session, &receive_packet,
                                              NX_IP_PERIODIC_RATE);

    /* Process response. */

    /* Shut down DTLS session. */
    status = nx_secure_dtls_session_end(&client_dtls_session, NX_IP_PERIODIC_RATE);

    /* Clean up. */
    status = nx_secure_dtls_session_delete(&client_dtls_session);

}
```

See Also

nx_secure_dtls_client_session_start, nx_secure_dtls_session_receive, nx_secure_dtls_session_send, nx_secure_dtls_session_delete

nx_secure_dtls_session_local_certificate_add

Add a local identity certificate to a NetX Secure DTLS Session

Prototype

```
UINT  nx_secure_dtls_session_local_certificate_add(
                                NX_SECURE_DTLS_SESSION *session_ptr,
                                NX_SECURE_X509_CERT *certificate,
                                UINT cert_id);
```

Description

This service adds a local identity certificate to a DTLS Session instance. In general, this service will be used when a DTLS Client session needs to provide an identity certificate to a remote server host. This is an optional configuration for DTLS so a certificate is not generally required for DTLS Clients. An identity certificate requires an associated private key.

The cert_id parameter is a numeric, non-zero identifier for the certificate. This enables the certificate to be easily removed or found in the event there are multiple identity certificates with the same X.509 Common Name present in the DTLS certificate store. See the NetX Secure TLS User Guide for more information about X.509 certificates.

Parameters

session_ptr	Pointer to a previously created DTLS Session instance.
certificate	Pointer to a previously initialized X.509 certificate structure.
cert_id	Numeric non-zero unique identifier for this certificate in this DTLS server.

Return Values

NX_SUCCESS	(0x00)	Successful addition of certificate to DTLS session.
NX_PTR_ERROR	(0x07)	Invalid pointer passed.
NX_INVALID_PARAMETERS	(0x4D)	A certificate ID of 0 was passed in.

Allowed From

Threads

Example

*See reference for *nx_secure_dtls_session_create* for a more complete example.

```
/* Our DTLS Server instance. */
NX_SECURE_DTLS_SESSION dtls_client;

/* Certificate control block and data. Identity certificates require a private key. */
NX_SECURE_X509_CERT certificate;
UCHAR certificate_der_data[] = { … };
UCHAR certificate_key_der_data[] = { … };


/* Application thread where TLS session is started. */
void application_thread()
{
NXD_ADDRESS server_ip;

    /* Create a TLS session for our socket. Ciphers and metadata defined
       elsewhere. See nx_secure_tls_session_create reference for more
       information. */
    status =  nx_secure_dtls_session_create(&client_dtls_session,
                                        &nx_crypto_tls_ciphers,
                                        crypto_metadata,
                                        sizeof(crypto_metadata),
                                         packet_buffer,
                                         sizeof(packet_buffer),
                                         REMOTE_CERT_NUMBER,
                                         remote_certs_buffer,
                                         sizeof(remote_certs_buffer));

    /* Check for error.  */
    if(status)
    {
        printf("Error in function nx_secure_dtls_session_create: 0x%x\n", status);
    }

    /* Initialize our trusted certificate. See section “Importing X.509
       Certificates into NetX Secure” for more information. */
    nx_secure_x509_certificate_initialize(&trusted_certificate,
                                    trusted_cert_der,
                                    trusted_cert_der_len, NX_NULL, 0, NX_NULL, 0,
                                    NX_SECURE_X509_KEY_TYPE_NONE);

    /* Add the certificate to the local store using a numeric ID. */
    nx_secure_dtls_session_trusted_certificate_add(&client_dtls_session,
                                                   &certificate, 1);

    /* Initialize local server identity certificate with key and add to server. */
    status = nx_secure_x509_certificate_initialize(&certificate, certificate_der_data,
                            sizeof(certificate_der_data), NX_NULL, 0,
                            certificate_key_der_data,
                            sizeof(certificate_key_der_data),
                            NX_SECURE_X509_KEY_TYPE_RSA_PKCS1_DER);

    /* Add local server identity certificate to DTLS server with ID of 1. */
    status = nx_secure_dtls_session_local_certificate_add(&dtls_client, &certificate, 1);

    /* Set up IP address of remote host. */
    server_ip.nxd_ip_version = NX_IP_VERSION_V4;
    server_ip.nxd_ip_address.v4 = IP_ADDRESS(192, 168, 1, 150);;
```

```
    /* Now we can start the DTLS session as normal. */
    status =  nx_secure_dtls_client_session_start(&client_dtls_session,
                                                  &udp_socket, &server_ip, 4443,
                                                  NX_IP_PERIODIC_RATE);


    /* Process responses, etc…*/
}
```

See Also

nx_secure_dtls_session_create, nx_secure_dtls_session_receive, nx_secure_dtls_session_send, nx_secure_dtls_session_start, nx_secure_dtls_session_local_certificate_remove, nx_secure_x509_certificate_initialize

nx_secure_dtls_session_local_certificate_remove

Remove a local identity certificate from a NetX Secure DTLS Session

Prototype

```
UINT  nx_secure_dtls_session_local_certificate_remove(
                                    NX_SECURE_DTLS_SESSION *session_ptr,
                                    UCHAR *common_name,
                                    UINT common_name_length, UINT cert_id);
```

Description

This service removes a local identity certificate from a DTLS Session instance using either a certificate ID number (assigned when the certificate was added with nx_secure_dtls_session_local_certificate_add) or the X.509 CommonName field.

If the common_name is used to match the certificate, the cert_id parameter should be set to 0. If cert_id is used, common_name should be passed a value of NX_NULL.

The cert_id parameter is a numeric, non-zero identifier for the certificate. This enables the certificate to be easily removed or found in the event there are multiple identity certificates with the same X.509 Common Name present in the DTLS certificate store. See the NetX Secure TLS User Guide for more information about X.509 certificates.

Parameters

session_ptr	Pointer to a previously created DTLS Session instance.
common_name	Pointer to the CommonName string to match.
common_name_length	Length of the common_name string.
cert_id	Numeric non-zero unique identifier for this certificate in this DTLS server.

Return Values

NX_SUCCESS	(0x00)	Successful removal of certificate from DTLS session.
NX_PTR_ERROR	(0x07)	Invalid pointer passed.
NX_SECURE_TLS_CERTIFICATE_NOT_FOUND	(0x119)	No certificate matching the cert_id or common_name was found in the given DTLS session.

Allowed From

Threads

Example

*See reference for *nx_secure_dtls_session_create* for a more complete example.

```
/* Our DTLS Server instance. */
NX_SECURE_DTLS_SESSION dtls_client;

/* Certificate control block and data. Identity certificates require a private key. */
NX_SECURE_X509_CERT certificate;
UCHAR certificate_der_data[] = { … };
UCHAR certificate_key_der_data[] = { … };


/* Application thread where TLS session is started. */
void application_thread()
{
NXD_ADDRESS server_ip;

    /* Create a TLS session for our socket. Ciphers and metadata defined
       elsewhere. See nx_secure_tls_session_create reference for more
       information. */
    status =  nx_secure_dtls_session_create(&client_dtls_session,
                                        &nx_crypto_tls_ciphers,
                                        crypto_metadata,
                                        sizeof(crypto_metadata),
                                        packet_buffer,
                                        sizeof(packet_buffer),
                                        REMOTE_CERT_NUMBER,
                                        remote_certs_buffer,
                                        sizeof(remote_certs_buffer));

    /* Check for error.  */
    if(status)
    {
        printf("Error in function nx_secure_dtls_session_create: 0x%x\n", status);
    }

    /* Initialize our trusted certificate. See section "Importing X.509
       Certificates into NetX Secure" for more information. */
    nx_secure_x509_certificate_initialize(&trusted_certificate,
                                    trusted_cert_der,
                                    trusted_cert_der_len, NX_NULL, 0, NX_NULL, 0,
                                    NX_SECURE_X509_KEY_TYPE_NONE);
```

```
    /* Add the certificate to the local store using a numeric ID. */
    nx_secure_dtls_session_trusted_certificate_add(&client_dtls_session,
                                                   &certificate, 1);

    /* Initialize local server identity certificate with key and add to server. */
    status = nx_secure_x509_certificate_initialize(&certificate, certificate_der_data,
                              sizeof(certificate_der_data), NX_NULL, 0,
                              certificate_key_der_data,
                              sizeof(certificate_key_der_data),
                              NX_SECURE_X509_KEY_TYPE_RSA_PKCS1_DER);

    /* Add local server identity certificate to DTLS server with ID of 1. */
    status = nx_secure_dtls_session_local_certificate_add(&dtls_client, &certificate, 1);

    /* Set up IP address of remote host. */
    server_ip.nxd_ip_version = NX_IP_VERSION_V4;
    server_ip.nxd_ip_address.v4 = IP_ADDRESS(192, 168, 1, 150);;


    /* Now we can start the DTLS session as normal. */
    status =  nx_secure_dtls_client_session_start(&client_dtls_session,
                                                  &udp_socket, &server_ip, 4443,
                                                  NX_IP_PERIODIC_RATE);

    /* Process responses, etc…*/

    /* At some point in the future, we decide to remove the certificate using the ID of 1
       when it was added to the session. */
    status = nx_secure_dtls_session_local_certificate_remove(&client_dtls_session,
                                                             NX_NULL, 0, 1);
}
```

See Also

nx_secure_dtls_session_create, nx_secure_dtls_session_receive, nx_secure_dtls_session_send, nx_secure_dtls_session_start, nx_secure_dtls_session_local_certificate_add, nx_secure_x509_certificate_initialize

nx_secure_dtls_session_receive

Receive application data over an established NetX Secure DTLS Session

Prototype

```
UINT nx_secure_dtls_session_receive(
                        NX_SECURE_DTLS_SESSION *dtls_session,
                        NX_PACKET **packet_ptr_ptr,
                        UINT wait_option);
```

Description

This service returns application data received by an active DTLS Session. The DTLS Session may be either a DTLS Server session (managed by a DTLS Server instance) or a DTLS Client session. The returned packet may be processed using any of the NX_PACKET API services (see the NetX documentation for more information).

Parameters

dtls_session	Pointer to a DTLS Session structure that was initialized previously.
packet_ptr_ptr	Pointer to an NX_PACKET pointer for the return packet.
wait_option	ThreadX wait value to use for network operations.

Return Values

NX_SUCCESS	(0x00)	Successful receipt of application data packet.
NX_PTR_ERROR	(0x07)	Invalid session or packet pointer.
NX_NOT_ENABLED	(0x14)	UDP is not enabled.
NX_NOT_BOUND	(0x24)	UDP socket not bound to port.
NX_SECURE_TLS_INVALID_PACKET	(0x104)	Recevied data that was not a valid DTLS record.
NX_SECURE_TLS_HASH_MAC_VERIFY_FAILURE	(0x108)	A DTLS record failed to be properly hashed (encryption error).
NX_SECURE_TLS_PADDING_CHECK_FAILED	(0x12A)	Encryption padding check failure.

NX_SECURE_TLS_ALERT_RECEIVED
(0x114) Recevied an alert from the remote host.
NX_SECURE_TLS_UNRECOGNIZED_MESSAGE_TYPE
(0x102) Received an unrecognized message.
NX_SECURE_TLS_INCORRECT_MESSAGE_LENGTH
(0x10A) Recevied a DTLS record with an invalid length.
NX_SECURE_TLS_UNKNOWN_CIPHERSUITE
(0x105) An unknown ciphersuite was encountered (indicates internal cryptography error).
NX_SECURE_TLS_PROTOCOL_VERSION_CHANGED
(0x12E) Recevied a DTLS record with a mismatched DTLS version.

Allowed From

Threads

Example

```
/* Sockets, sessions, certificates defined in global static space to preserve
   application stack. */
NX_SECURE_DTLS_SESSION client_dtls_session;

/* Trusted certificate structure and raw data. */
NX_SECURE_X509_CERT trusted_cert;
const UCHAR trusted_cert_der = { … };

/* Cryptography routines and crypto work buffers. */
extern const NX_SECURE_TLS_CRYPTO nx_crypto_tls_ciphers;
UCHAR crypto_metadata[9000];

/* Packet reassembly buffer for decryption. */
UCHAR packet_buffer[4000];

/* Remote certificate buffer for incoming certificates. */
#define REMOTE_CERT_SIZE (sizeof(NX_SECURE_X509_CERT) + 2000)
#define REMOTE_CERT_NUMBER  (3)
UCHAR remote_certs_buffer[REMOTE_CERT_SIZE * REMOTE_CERT_NUMBER];

/* Application thread where TLS session is started. */
void application_thread()
{
NXD_ADDRESS server_ip;

    /* Create a TLS session for our socket. Ciphers and metadata defined
        elsewhere. See nx_secure_tls_session_create reference for more
        information. */
    status =  nx_secure_dtls_session_create(&client_dtls_session,
                                            &nx_crypto_tls_ciphers,
                                            crypto_metadata,
                                            sizeof(crypto_metadata),
                                             packet_buffer,
                                             sizeof(packet_buffer),
                                             REMOTE_CERT_NUMBER,
                                             remote_certs_buffer,
```

```
                                                sizeof(remote_certs_buffer));

    /* Check for error.  */
    if(status)
    {
        printf("Error in function nx_secure_dtls_session_create: 0x%x\n", status);
    }

    /* Initialize our trusted certificate. See section "Importing X.509
        Certificates into NetX Secure" for more information. */
    nx_secure_x509_certificate_initialize(&trusted_certificate,
                                          trusted_cert_der,
                                          trusted_cert_der_len, NX_NULL, 0, NX_NULL, 0,
                                          NX_SECURE_X509_KEY_TYPE_NONE);

    /* Add the certificate to the local store using a numeric ID. */
    nx_secure_dtls_session_trusted_certificate_add(&client_dtls_session,
                                                   &certificate, 1);

    /* Set up IP address of remote host. */
    server_ip.nxd_ip_version = NX_IP_VERSION_V4;
    server_ip.nxd_ip_address.v4 = IP_ADDRESS(192, 168, 1, 150);;


    /* Now we can start the DTLS session as normal. */
    status =  nx_secure_dtls_client_session_start(&client_dtls_session,
                                                  &udp_socket, &server_ip, 4443,
                                                  NX_IP_PERIODIC_RATE);

    if(status != NX_SUCCESS)
    {
        /* Error! */
        return(status);
     }

   /* Add data to send packet as usual for NX_PACKET and send to server.  */
   status = nx_secure_dtls_session_send(&client_dtls_session, &send_packet,
                                                 &server_ip, 4443)

    /* Receive response from server. */
    status = nx_secure_dtls_session_receive(&client_dtls_session, &receive_packet,
                                                  NX_IP_PERIODIC_RATE);

    /* Process response. */

    /* Shut down DTLS session. */
    status = nx_secure_dtls_session_end(&client_dtls_session, NX_IP_PERIODIC_RATE);

    /* Clean up. */
    status = nx_secure_dtls_session_delete(&client_dtls_session);

}
```

See Also

nx_secure_dtls_client_session_start, nx_secure_dtls_session_end, nx_secure_dtls_session_send, nx_secure_dtls_session_delete

nx_secure_dtls_session_reset

Clear data in an NetX Secure DTLS Session instance

Prototype

```
UINT nx_secure_dtls_session_reset(NX_SECURE_DTLS_SESSION *dtls_session);
```

Description

This service resets a DTLS session, clearing all ephemeral cryptographic data and allowing the structure to be re-used for a new session. Persistent data (e.g. certificate stores) are maintained so that nx_secure_dtls_session_create need not be called repeatedly.

Parameters

dtls_session	Pointer to a DTLS Session structure that was initialized previously.

Return Values

NX_SUCCESS	(0x00)	Successful reset of session.
NX_PTR_ERROR	(0x07)	Invalid session or buffer pointer.

Allowed From

Threads

Example

```
/* Sockets, sessions, certificates defined in global static space to preserve
   application stack. */
NX_SECURE_DTLS_SESSION client_dtls_session;

/* Trusted certificate structure and raw data. */
NX_SECURE_X509_CERT trusted_cert;
const UCHAR trusted_cert_der = { … };

/* Cryptography routines and crypto work buffers. */
extern const NX_SECURE_TLS_CRYPTO nx_crypto_tls_ciphers;
UCHAR crypto_metadata[9000];

/* Packet reassembly buffer for decryption. */
UCHAR packet_buffer[4000];

/* Remote certificate buffer for incoming certificates. */
#define REMOTE_CERT_SIZE (sizeof(NX_SECURE_X509_CERT) + 2000)
#define REMOTE_CERT_NUMBER  (3)
UCHAR remote_certs_buffer[REMOTE_CERT_SIZE * REMOTE_CERT_NUMBER];

/* Application thread where TLS session is started. */
void application_thread()
{
NXD_ADDRESS server_ip;
```

```
    /* Create a TLS session for our socket. Ciphers and metadata defined
        elsewhere. See nx_secure_tls_session_create reference for more
        information. */
    status =  nx_secure_dtls_session_create(&client_dtls_session,
                                            &nx_crypto_tls_ciphers,
                                            crypto_metadata,
                                            sizeof(crypto_metadata),
                                            packet_buffer,
                                            sizeof(packet_buffer),
                                            REMOTE_CERT_NUMBER,
                                            remote_certs_buffer,
                                            sizeof(remote_certs_buffer));

    /* Check for error.  */
    if(status)
    {
        printf("Error in function nx_secure_dtls_session_create: 0x%x\n", status);
    }

    /* Initialize our trusted certificate. See section "Importing X.509
        Certificates into NetX Secure" for more information. */
    nx_secure_x509_certificate_initialize(&trusted_certificate,
                                          trusted_cert_der,
                                          trusted_cert_der_len, NX_NULL, 0, NX_NULL, 0,
                                          NX_SECURE_X509_KEY_TYPE_NONE);

    /* Add the certificate to the local store using a numeric ID. */
    nx_secure_dtls_session_trusted_certificate_add(&client_dtls_session,
                                                   &certificate, 1);

    /* Set up IP address of remote host. */
    server_ip.nxd_ip_version = NX_IP_VERSION_V4;
    server_ip.nxd_ip_address.v4 = IP_ADDRESS(192, 168, 1, 150);;


    /* Now we can start the DTLS session as normal. */
    status =  nx_secure_dtls_client_session_start(&client_dtls_session,
                                                  &udp_socket, &server_ip, 4443,
                                                  NX_IP_PERIODIC_RATE);

    if(status != NX_SUCCESS)
    {
        /* Error! */
        return(status);
     }

   /* Add data to send packet as usual for NX_PACKET and send to server.  */
   status = nx_secure_dtls_session_send(&client_dtls_session, &send_packet,
                                                 &server_ip, 4443)

    /* Receive response from server. */
      status = nx_secure_dtls_session_receive(&client_dtls_session, &receive_packet,
                                                  NX_IP_PERIODIC_RATE);

    /* Process response. */

    /* Shut down DTLS session. */
    status = nx_secure_dtls_session_reset(&client_dtls_session);

    /* A new session can now be started using the same structure. */
    status =  nx_secure_dtls_client_session_start(&client_dtls_session,
                                                  &udp_socket, &server_ip, 4443,
                                                  NX_IP_PERIODIC_RATE);


    /* Clean up. */
    status = nx_secure_dtls_session_delete(&client_dtls_session);

}
```

See Also

nx_secure_dtls_client_session_start, nx_secure_dtls_session_receive, nx_secure_dtls_session_send, nx_secure_dtls_session_delete

nx_secure_dtls_ session_send

Send data over a DTLS session

Prototype

```
UINT  nx_secure_dtls_session_send(NX_SECURE_DTLS_SESSION *session_ptr,
                                  NX_PACKET *packet_ptr,
                                  NXD_ADDRESS *ip_address, UINT port);
```

Description

This service sends a packet of data over an established DTLS Session to a remote DTLS host at the given IP address and port. The session used is an active DTLS Client session. Note that the IP address and port are provided due to the stateless nature of UDP but should generally match the address and port used to start the session in nx_secure_dtls_session_start.

The data provided in the packet, which must be allocated using *nx_secure_dtls_packet_allocate*, is encrypted using the DTLS session cryptographic parameters and routines and then sent to the remote host over the DTLS Session's UDP socket.

Parameters

session_ptr	Pointer to an active DTLS client session instance.
packet_ptr	Pointer to an NX_PACKET instance allocated previously and populated with application data.
ip_address	Pointer to an NXD_ADDRESS structure containing the IP address of the remote host.
port	UDP port on the remote host.

Return Values

NX_SUCCESS	(0x00)	Successful send of packet.
NX_PTR_ERROR	(0x07)	Invalid pointer passed.
NX_SECURE_TLS_TCP_SEND_FAILED	(0x109)	An error occurred in the underlying UDP send operation.

Allowed From

Threads

Example

```
/* Sockets, sessions, certificates defined in global static space to preserve
   application stack. */
NX_SECURE_DTLS_SESSION client_dtls_session;

/* Trusted certificate structure and raw data. */
NX_SECURE_X509_CERT trusted_cert;
const UCHAR trusted_cert_der = { … };

/* Cryptography routines and crypto work buffers. */
extern const NX_SECURE_TLS_CRYPTO nx_crypto_tls_ciphers;
UCHAR crypto_metadata[9000];

/* Packet reassembly buffer for decryption. */
UCHAR packet_buffer[4000];

/* Remote certificate buffer for incoming certificates. */
#define REMOTE_CERT_SIZE (sizeof(NX_SECURE_X509_CERT) + 2000)
#define REMOTE_CERT_NUMBER  (3)
UCHAR remote_certs_buffer[REMOTE_CERT_SIZE * REMOTE_CERT_NUMBER];

/* Application thread where TLS session is started. */
void application_thread()
{
NXD_ADDRESS server_ip;

    /* Create a TLS session for our socket. Ciphers and metadata defined
       elsewhere. See nx_secure_tls_session_create reference for more
       information. */
    status =  nx_secure_dtls_session_create(&client_dtls_session,
                                        &nx_crypto_tls_ciphers,
                                        crypto_metadata,
                                        sizeof(crypto_metadata),
                                        packet_buffer,
                                        sizeof(packet_buffer),
                                        REMOTE_CERT_NUMBER,
                                        remote_certs_buffer,
                                        sizeof(remote_certs_buffer));

    /* Check for error.  */
    if(status)
    {
        printf("Error in function nx_secure_dtls_session_create: 0x%x\n", status);
    }

    /* Initialize our trusted certificate. See section "Importing X.509
       Certificates into NetX Secure" for more information. */
    nx_secure_x509_certificate_initialize(&trusted_certificate,
                                    trusted_cert_der,
                                    trusted_cert_der_len, NX_NULL, 0, NX_NULL,
                                    0,
                                    NX_SECURE_X509_KEY_TYPE_NONE);

    /* Add the certificate to the local store using a numeric ID. */
    nx_secure_dtls_session_trusted_certificate_add(&client_dtls_session,
                                                   &certificate, 1);

    /* Set up IP address of remote host. */
    server_ip.nxd_ip_version = NX_IP_VERSION_V4;
    server_ip.nxd_ip_address.v4 = IP_ADDRESS(192, 168, 1, 150);;


    /* Now we can start the DTLS session as normal. */
    status =  nx_secure_dtls_client_session_start(&client_dtls_session,
                                                  &udp_socket, &server_ip, 4443,
                                                  NX_IP_PERIODIC_RATE);

if(status != NX_SUCCESS)
   {
```

```
    /* Error! */
    return(status);
    }

   /* Add data to send packet as usual for NX_PACKET and send to server.  */
   status = nx_secure_dtls_session_send(&client_dtls_session, &send_packet,
                                        &server_ip, 4443)

      /* Receive response from server. */
      status = nx_secure_dtls_session_receive(&client_dtls_session, &receive_packet,
                                              NX_IP_PERIODIC_RATE);

/* Process response. */

/* Shut down DTLS session. */
     status = nx_secure_dtls_session_end(&client_dtls_session, NX_IP_PERIODIC_RATE);

/* Clean up. */
     status = nx_secure_dtls_session_delete(&client_dtls_session);

    }
```

See Also

nx_secure_dtls_client_session_start, nx_secure_dtls_session_receive, nx_secure_dtls_session_create

nx_secure_dtls_session_trusted_certificate_add

Add a trusted CA certificate to a NetX Secure DTLS Session

Prototype

```
UINT  nx_secure_dtls_session_trusted_certificate_add(
                                    NX_SECURE_DTLS_SESSION *session_ptr,
                                    NX_SECURE_X509_CERT *certificate,
                                    UINT cert_id);
```

Description

This service adds a trusted CA or intermediate CA X.509 certificate to a DTLS Session instance. A DTLS Client requires at least one trusted certificate in order to validate remote server certificates unless an alternative authentication mechanism is used (e.g. Pre-Shared Keys). A trusted certificate does not usually have a private key.

The cert_id parameter is a numeric, non-zero identifier for the certificate. This enables the certificate to be easily removed or found in the event there are multiple identity certificates with the same X.509 Common Name present in the DTLS certificate store. See the NetX Secure TLS User Guide for more information about X.509 certificates.

Parameters

session_ptr	Pointer to a previously created DTLS Session instance.
certificate	Pointer to a previously initialized X.509 certificate structure.
cert_id	Numeric non-zero unique identifier for this certificate in this DTLS server.

Return Values

NX_SUCCESS	(0x00)	Successful addition of certificate to DTLS session.
NX_PTR_ERROR	(0x07)	Invalid pointer passed.
NX_INVALID_PARAMETERS	(0x4D)	A certificate ID of 0 was passed in.

Allowed From

Threads

Example

*See reference for *nx_secure_dtls_session_create* for a more complete example.

```
/* Our DTLS Server instance. */
NX_SECURE_DTLS_SESSION dtls_client;

/* Certificate control block and data. Identity certificates require a private key. */
NX_SECURE_X509_CERT certificate;
UCHAR certificate_der_data[] = { … };
UCHAR certificate_key_der_data[] = { … };


/* Application thread where TLS session is started. */
void application_thread()
{
NXD_ADDRESS server_ip;

    /* Create a TLS session for our socket. Ciphers and metadata defined
       elsewhere. See nx_secure_tls_session_create reference for more
       information. */
    status =  nx_secure_dtls_session_create(&client_dtls_session,
                                        &nx_crypto_tls_ciphers,
                                        crypto_metadata,
                                        sizeof(crypto_metadata),
                                        packet_buffer,
                                        sizeof(packet_buffer),
                                        REMOTE_CERT_NUMBER,
                                        remote_certs_buffer,
                                        sizeof(remote_certs_buffer));

    /* Check for error.  */
    if(status)
    {
        printf("Error in function nx_secure_dtls_session_create: 0x%x\n", status);
    }

    /* Initialize our trusted certificate. See section "Importing X.509
       Certificates into NetX Secure" for more information. */
    nx_secure_x509_certificate_initialize(&trusted_certificate,
                                    trusted_cert_der,
                                    trusted_cert_der_len, NX_NULL, 0, NX_NULL, 0,
                                    NX_SECURE_X509_KEY_TYPE_NONE);

    /* Add the certificate to the trusted store using a numeric ID. */
    nx_secure_dtls_session_trusted_certificate_add(&client_dtls_session,
                                                   &certificate, 1);

    /* Initialize local server identity certificate with key and add to server. */
    status = nx_secure_x509_certificate_initialize(&certificate, certificate_der_data,
                            sizeof(certificate_der_data), NX_NULL, 0,
                            certificate_key_der_data,
                            sizeof(certificate_key_der_data),
                            NX_SECURE_X509_KEY_TYPE_RSA_PKCS1_DER);

    /* Add local server identity certificate to DTLS server with ID of 1. */
    status = nx_secure_dtls_session_local_certificate_add(&dtls_client, &certificate, 1);

    /* Set up IP address of remote host. */
    server_ip.nxd_ip_version = NX_IP_VERSION_V4;
    server_ip.nxd_ip_address.v4 = IP_ADDRESS(192, 168, 1, 150);;


    /* Now we can start the DTLS session as normal. */
    status =  nx_secure_dtls_client_session_start(&client_dtls_session,
                                                  &udp_socket, &server_ip, 4443,
                                                  NX_IP_PERIODIC_RATE);
```

```
    /* Process responses, etc…*/
}
```

See Also

nx_secure_dtls_session_create, nx_secure_dtls_session_receive, nx_secure_dtls_session_send, nx_secure_dtls_session_start, nx_secure_dtls_session_trusted_certificate_remove, nx_secure_x509_certificate_initialize

nx_secure_dtls_session_trusted_certificate_remove

Remove a trusted CA certificate from a NetX Secure DTLS Session

Prototype

```
UINT  nx_secure_dtls_session_trusted_certificate_remove(
                                    NX_SECURE_DTLS_SESSION *session_ptr,
                                    UCHAR *common_name,
                                    UINT common_name_length, UINT cert_id);
```

Description

This service removes a trusted CA certificate from a DTLS Session instance using either a certificate ID number (assigned when the certificate was added with nx_secure_dtls_session_trusted_certificate_add) or the X.509 CommonName field.

If the common_name is used to match the certificate, the cert_id parameter should be set to 0. If cert_id is used, common_name should be passed a value of NX_NULL.

The cert_id parameter is a numeric, non-zero identifier for the certificate. This enables the certificate to be easily removed or found in the event there are multiple identity certificates with the same X.509 Common Name present in the DTLS certificate store. See the NetX Secure TLS User Guide for more information about X.509 certificates.

Parameters

session_ptr	Pointer to a previously created DTLS Session instance.
common_name	Pointer to the CommonName string to match.
common_name_length	Length of the common_name string.
cert_id	Numeric non-zero unique identifier for this certificate in this DTLS server.

Return Values

NX_SUCCESS	(0x00)	Successful removal of certificate from DTLS session.
NX_PTR_ERROR	(0x07)	Invalid pointer passed.
NX_SECURE_TLS_CERTIFICATE_NOT_FOUND	(0x119)	No certificate matching the cert_id or common_name was found in the given DTLS session.

Allowed From

Threads

Example

*See reference for *nx_secure_dtls_session_create* for a more complete example.

```
/* Our DTLS Server instance. */
NX_SECURE_DTLS_SESSION dtls_client;

/* Certificate control block and data. Identity certificates require a private key. */
NX_SECURE_X509_CERT certificate;
UCHAR certificate_der_data[] = { … };
UCHAR certificate_key_der_data[] = { … };


/* Application thread where TLS session is started. */
void application_thread()
{
NXD_ADDRESS server_ip;

    /* Create a TLS session for our socket. Ciphers and metadata defined
       elsewhere. See nx_secure_tls_session_create reference for more
       information. */
    status =  nx_secure_dtls_session_create(&client_dtls_session,
                                        &nx_crypto_tls_ciphers,
                                        crypto_metadata,
                                        sizeof(crypto_metadata),
                                         packet_buffer,
                                         sizeof(packet_buffer),
                                         REMOTE_CERT_NUMBER,
                                         remote_certs_buffer,
                                         sizeof(remote_certs_buffer));

    /* Check for error.  */
    if(status)
    {
        printf("Error in function nx_secure_dtls_session_create: 0x%x\n", status);
    }

    /* Initialize our trusted certificate. See section “Importing X.509
       Certificates into NetX Secure” for more information. */
    nx_secure_x509_certificate_initialize(&trusted_certificate,
                                    trusted_cert_der,
                                    trusted_cert_der_len, NX_NULL, 0, NX_NULL, 0,
                                    NX_SECURE_X509_KEY_TYPE_NONE);
```

```
    /* Add the certificate to the local store using a numeric ID. */
    nx_secure_dtls_session_trusted_certificate_add(&client_dtls_session,
                                                   &certificate, 1);

    /* Initialize local server identity certificate with key and add to server. */
    status = nx_secure_x509_certificate_initialize(&certificate, certificate_der_data,
                               sizeof(certificate_der_data), NX_NULL, 0,
                               certificate_key_der_data,
                               sizeof(certificate_key_der_data),
                               NX_SECURE_X509_KEY_TYPE_RSA_PKCS1_DER);

    /* Add local server identity certificate to DTLS server with ID of 1. */
    status = nx_secure_dtls_session_local_certificate_add(&dtls_client, &certificate, 1);

    /* Set up IP address of remote host. */
    server_ip.nxd_ip_version = NX_IP_VERSION_V4;
    server_ip.nxd_ip_address.v4 = IP_ADDRESS(192, 168, 1, 150);;


    /* Now we can start the DTLS session as normal. */
    status =  nx_secure_dtls_client_session_start(&client_dtls_session,
                                                  &udp_socket, &server_ip, 4443,
                                                  NX_IP_PERIODIC_RATE);

    /* Process responses, etc…*/

    /* At some point in the future, we decide to remove the certificate using the ID of 1
       when it was added to the session. */
    status = nx_secure_dtls_session_trusted_certificate_remove(&client_dtls_session,
                                                               NX_NULL, 0, 1);
}
```

See Also

nx_secure_dtls_session_create, nx_secure_dtls_session_receive, nx_secure_dtls_session_send, nx_secure_dtls_session_start, nx_secure_dtls_session_trusted_certificate_add, nx_secure_x509_certificate_initialize

Appendix A

NetX Secure Return/Error Codes

NetX Secure TLS/DTLS Return Codes

Table 1 below lists the possible error codes that may be returned by NetX Secure TLS services. Note that the services may also return UDP or IP error codes – TLS values begin at 0x101 and TCP/IP/UDP values are below 0x100. X.509 return values start at 0x181. Refer to the NetX TCP/IP/UDP documentation for information on IP and UDP return values and see below for X.509 values.

Error Name	Value	Description
NX_SECURE_TLS_SUCCESS	0x00	Function returned successfully. (Same as NX_SUCCESS).
NX_SECURE_TLS_SESSION_UNINITIALIZED	0x101	TLS main loop called with uninitialized socket.
NX_SECURE_TLS_UNRECOGNIZED_MESSAGE_TYPE	0x102	TLS record layer received an unrecognized message type.
NX_SECURE_TLS_INVALID_STATE	0x103	Internal error - state not recognized.
NX_SECURE_TLS_INVALID_PACKET	0x104	Internal error - received packet did not contain TLS data.
NX_SECURE_TLS_UNKNOWN_CIPHERSUITE	0x105	The chosen ciphersuite is not supported - internal error for server, for client it means the remote host sent a bad ciphersuite (error or attack).
NX_SECURE_TLS_UNSUPPORTED_CIPHER	0x106	In doing an encryption or decryption, the chosen cipher is disabled or unavailable.
NX_SECURE_TLS_HANDSHAKE_FAILURE	0x107	Something in message processing during the handshake has failed.
NX_SECURE_TLS_HASH_MAC_VERIFY_FAILURE	0x108	An incoming record had a MAC that did not match the one we generated.
NX_SECURE_TLS_TCP_SEND_FAILED	0x109	The outgoing TCP send of a record failed for some reason.
NX_SECURE_TLS_INCORRECT_MESSAGE_LENGTH	0x10A	An incoming message had a length that was incorrect (usually a length other than one in the header, as in certificate messages)
NX_SECURE_TLS_BAD_CIPHERSPEC	0x10B	An incoming ChangeCipherSpec message was incorrect.
NX_SECURE_TLS_INVALID_SERVER_CERT	0x10C	An incoming server certificate did not parse correctly.
NX_SECURE_TLS_UNSUPPORTED_PUBLIC_CIPHER	0x10D	A certificate provided by a server specified a public-key operation we do not support.
NX_SECURE_TLS_NO_SUPPORTED_CIPHERS	0x10E	Received a ClientHello with no supported ciphersuites.
NX_SECURE_TLS_UNKNOWN_TLS_VERSION	0x10F	An incoming record had a TLS version that isn't recognized.
NX_SECURE_TLS_UNSUPPORTED_TLS_VERSION	0x110	An incoming record had a valid TLS version, but one that isn't supported.
NX_SECURE_TLS_ALLOCATE_PACKET_FAILED	0x111	An internal packet allocation for a TLS message failed.

Error Name	Value	Description
NX_SECURE_TLS_INVALID_CERTIFICATE	0x112	An X509 certificate did not parse correctly.
NX_SECURE_TLS_NO_CLOSE_RESPONSE	0x113	During a TLS session close, did not receive a CloseNotify from the remote host.
NX_SECURE_TLS_ALERT_RECEIVED	0x114	The remote host sent an alert, indicating an error and closing the connection.
NX_SECURE_TLS_FINISHED_HASH_FAILURE	0x115	The Finish message hash received does not match the local generated hash - handshake corruption.
NX_SECURE_TLS_UNKNOWN_CERT_SIG_ALGORITHM	0x116	A certificate during verification had an unsupported signature algorithm.
NX_SECURE_TLS_CERTIFICATE_SIG_CHECK_FAILED	0x117	A certificate signature verification check failed - certificate data did not match signature.
NX_SECURE_TLS_BAD_COMPRESSION_METHOD	0x118	Received a Hello message with an unsupported compression method.
NX_SECURE_TLS_CERTIFICATE_NOT_FOUND	0x119	In an operation on a certificate list, no matching certificate was found.
NX_SECURE_TLS_INVALID_SELF_SIGNED_CERT	0x11A	The remote host sent a self-signed certificate and NX_SECURE_ALLOW_SELF_SIGNED_ CERTIFICATES is not defined.
NX_SECURE_TLS_ISSUER_CERTIFICATE_NOT_FOUND	0x11B	A remote certificate was received with an issuer not in the local trusted store.
NX_SECURE_TLS_OUT_OF_ORDER_MESSAGE	0x11C	A DTLS message was received in the wrong order - a dropped datagram is the likely culprit.
NX_SECURE_TLS_INVALID_REMOTE_HOST	0x11D	A packet was received from a remote host that we do not recognize.
NX_SECURE_TLS_INVALID_EPOCH	0x11E	A DTLS message was received and matched to a DTLS session but it had the wrong epoch and should be ignored.
NX_SECURE_TLS_REPEAT_MESSAGE_RECEIVED	0x11F	A DTLS message was received with a sequence number we have already seen, ignore it.
NX_SECURE_TLS_NEED_DTLS_SESSION	0x120	A TLS session was used in a DTLS API that was not initialized for DTLS.
NX_SECURE_TLS_NEED_TLS_SESSION	0x121	A TLS session was used in a TLS API that was initialized for DTLS and not TLS.
NX_SECURE_TLS_SEND_ADDRESS_MISMATCH	0x122	Caller attempted to send data over a DTLS session with an IP address or port that did not match the session.
NX_SECURE_TLS_NO_FREE_DTLS_SESSIONS	0x123	A new connection tried to get a DTLS session from the cache, but there were none free.
NX_SECURE_DTLS_SESSION_NOT_FOUND	0x124	The caller searched for a DTLS session, but the given IP address and port did not match any entries in the cache.
NX_SECURE_TLS_NO_MORE_PSK_SPACE	0x125	The caller attempted to add a PSK to a TLS session but there was no more space in the given session.
NX_SECURE_TLS_NO_MATCHING_PSK	0x126	A remote host provided a PSK identity hint that did not match any in our local store.
NX_SECURE_TLS_CLOSE_NOTIFY_RECEIVED	0x127	A TLS session received a CloseNotify alert from the remote host indicating the session is complete.
NX_SECURE_TLS_NO_AVAILABLE_SESSIONS	0x128	No TLS sessions in a TLS object are available to handle a connection.
NX_SECURE_TLS_NO_CERT_SPACE_ALLOCATED	0x129	No certificate space was allocated for incoming remote certificates.

Error Name	Value	Description
NX_SECURE_TLS_PADDING_CHECK_FAILED	0x12A	Encryption padding in an incoming message was not correct.
NX_SECURE_TLS_UNSUPPORTED_CERT_SIGN_TYPE	0x12B	In processing a CertificateVerifyRequest, no supported certificate type was provided by the remote server.
NX_SECURE_TLS_UNSUPPORTED_CERT_SIGN_ALG	0x12C	In processing a CertificateVerifyRequest, no supported signature algorithm was provided by the remote server.
NX_SECURE_TLS_INSUFFICIENT_CERT_SPACE	0x12D	Not enough certificate buffer space allocated for a certificate.
NX_SECURE_TLS_PROTOCOL_VERSION_CHANGED	0x12E	The protocol version in an incoming TLS record did not match the version of the established session.
NX_SECURE_TLS_NO_RENEGOTIATION_ERROR	0x12F	A HelloRequest message was received, but we are not re-negotiating.
NX_SECURE_TLS_UNSUPPORTED_FEATURE	0x130	A feature that was disabled was encountered during a TLS session or handshake.
NX_SECURE_TLS_CERTIFICATE_VERIFY_FAILURE	0x131	A CertificateVerify message from a remote Client failed to verify the Client certificate.
NX_SECURE_TLS_EMPTY_REMOTE_CERTIFICATE_RECEIVED	0x132	The remote host sent an empty certificate message.
NX_SECURE_TLS_RENEGOTIATION_EXTENSION_ERROR	0x133	An error occurred in processing an or sending a Secure Renegotation Indication extension.
NX_SECURE_TLS_RENEGOTIATION_SESSION_INACTIVE	0x134	A session renegotiation was attempting with a TLS session that was not active.
NX_SECURE_TLS_PACKET_BUFFER_TOO_SMALL	0x135	TLS received a record that was too large for the assigned packet buffer. The record could not be processed.
NX_SECURE_TLS_EXTENSION_NOT_FOUND	0x136	A specified extension was not received from the remote host during the TLS handshake.
NX_SECURE_TLS_SNI_EXTENSION_INVALID	0x137	TLS received an invalid Server Name Indication extension.
NX_SECURE_TLS_CERT_ID_INVALID	0x138	Application tried to add a server certificate with an invalid certificate ID value (likely 0).
NX_SECURE_TLS_CERT_ID_DUPLICATE	0x139	Application tried to add a server certificate with a certificate ID already present in the local store.
NX_SECURE_TLS_RENEGOTIATION_FAILURE	0x13A	The remote host did not provide the Secure Renegotiation Indication Extension or the SCSV pseudo-ciphersuite so secure renegotiation cannot be performed.
NX_SECURE_TLS_MISSING_CRYPTO_ROUTINE	0x13B	In attempting to perform a cryptographic operation, one of the entries in the ciphersuite table (or one of its function pointers) was improperly set to NULL.

Table 1 – NetX Secure TLS error return codes

NetX Secure X.509 Return Codes

Table 2 below lists the possible error codes that may be returned by NetX Secure X.509 services. Note that the services may also return other error codes. X.509 return values start at 0x181, TLS values begin at 0x101, and TCP/IP values are below 0x100. Refer to

the NetX TCP/IP documentation for information on TCP/IP return values and above for TLS return values.

Error Name	Value	Description
NX_SECURE_X509_SUCCESS	0x00	Successful return status. (Same as NX_SUCCESS)
NX_SECURE_X509_MULTIBYTE_TAG_UNSUPPORTED	0x181	We encountered a multi-byte ASN.1 tag - not currently supported.
NX_SECURE_X509_ASN1_LENGTH_TOO_LONG	0x182	Encountered a length value longer than we can handle.
NX_SECURE_X509_FOUND_NON_ZERO_PADDING	0x183	Expected a padding value of 0 - got something different.
NX_SECURE_X509_MISSING_PUBLIC_KEY	0x184	X509 expected a public key but didn't find one.
NX_SECURE_X509_INVALID_PUBLIC_KEY	0x185	Found a public key, but it is invalid or has an incorrect format.
NX_SECURE_X509_INVALID_CERTIFICATE_SEQUENCE	0x186	The top-level ASN.1 block is not a sequence - invalid X509 certificate.
NX_SECURE_X509_MISSING_SIGNATURE_ALGORITHM	0x187	Expecting a signature algorithm identifier, did not find it.
NX_SECURE_X509_INVALID_CERTIFICATE_DATA	0x188	Certificate identity data is in an invalid format.
NX_SECURE_X509_UNEXPECTED_ASN1_TAG	0x189	We were expecting a specific ASN.1 tag for X509 format but we got something else.
NX_SECURE_PKCS1_INVALID_PRIVATE_KEY	0x18A	A PKCS#1 private key file was passed in, but the formatting was incorrect.
NX_SECURE_X509_CHAIN_TOO_SHORT	0x18B	An X509 certificate chain was too short to hold the entire chain during chain building.
NX_SECURE_X509_CHAIN_VERIFY_FAILURE	0x18C	An X509 certificate chain was unable to be verified (catch-all error).
NX_SECURE_X509_PKCS7_PARSING_FAILED	0x18D	Parsing an X.509 PKCS#7-encoded signature failed.
NX_SECURE_X509_CERTIFICATE_NOT_FOUND	0x18E	In looking up a certificate, no matching entry was found.
NX_SECURE_X509_INVALID_VERSION	0x18F	A certificate included a field that isn't compatible with the given version.
NX_SECURE_X509_INVALID_TAG_CLASS	0x190	A certificate included an ASN.1 tag with an invalid tag class value.
NX_SECURE_X509_INVALID_EXTENSIONS	0x191	A certificate included an extensions TLV but that did not contain a sequence.
NX_SECURE_X509_INVALID_EXTENSION_SEQUENCE	0x192	A certificate included an extension sequence that was invalid X.509.
NX_SECURE_X509_CERTIFICATE_EXPIRED	0x193	A certificate had a "not after" field that was less than the current time.
NX_SECURE_X509_CERTIFICATE_NOT_YET_VALID	0x194	A certificate had a "not before" field that was greater than the current time.
NX_SECURE_X509_CERTIFICATE_DNS_MISMATCH	0x195	A certificate Common Name or Subject Alt Name did not match a given DNS TLD.

Error Name	Value	Description
NX_SECURE_X509_INVALID_DATE_FORMAT	0x196	A certificate contained a date field that is not in a recognixed format.
NX_SECURE_X509_CRL_ISSUER_MISMATCH	0x197	A provided CRL and certificate were not issued by the same Certificate Authority.
NX_SECURE_X509_CRL_SIGNATURE_CHECK_FAILED	0x198	A CRL signature check failed against its issuer.
NX_SECURE_X509_CRL_CERTIFICATE_REVOKED	0x199	A certificate was found in a valid CRL and has therefore been revoked.
NX_SECURE_X509_WRONG_SIGNATURE_METHOD	0x19A	In attempting to validate a signature the signature method did not match the expected method.
NX_SECURE_X509_EXTENSION_NOT_FOUND	0x19B	In looking for an extension, no extension with a matching ID was found.
NX_SECURE_X509_ALT_NAME_NOT_FOUND	0x19C	A name was searched for in a subjectAltName extension but was not found.
NX_SECURE_X509_INVALID_PRIVATE_KEY_TYPE	0x19D	Private key type given was unknown or invalid.
NX_SECURE_X509_NAME_STRING_TOO_LONG	0x19E	Passed a name string that was too long for an internal buffer (DNS name, etc…).
NX_SECURE_X509_EXT_KEY_USAGE_NOT_FOUND	0x19F	In searching an Extended Key Usage extension, the specified key usage OID was not found.
NX_SECURE_X509_KEY_USAGE_ERROR	0x1A0	To be returned by the application callback if there is a failure in key usage during a certificate verification check.

Table 2 – NetX Secure X.509 error return codes