User's Manual

NetX™ Secure

User Guide

Renesas Synergy™ Platform
Synergy Software
Synergy Software (SSP) Component

All information contained in these materials, including products and product specifications, represents information on the product at the time of publication and is subject to change by Renesas Electronics Corp. without notice. Please review the latest information published by Renesas Electronics Corp. through various means, including the Renesas Electronics Corp. website (http://www.renesas.com).

Renesas Electronics
www.renesas.com

Mar 4, 2019

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The intended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.
   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user's manual or other Renesas Electronics document.
6. When using Renesas Electronics products, refer to the latest product information (data sheets, user's manuals, application notes, "General Notes for Handling and Using Semiconductor Devices" in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

(Rev.4.0-1 November 2017)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Trademarks

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit: www.renesas.com/contact/.

Renesas Synergy Specific Information

If you are using NetX Secure for the Renesas Synergy platform, please use the following information.

Supported MCUs

NetX Secure is supported on the following Renesas MCUs only:

- S7G2
- S5D9
- S5D5
- S5D3

It cannot run on other MCUs because they do not support hardware crypto which is required for TLS.

Unsupported Features

The following features have been integrated into Renesas SSP but not been tested:

- TLS 1.0
- TLS Server
- PSK Ciphersuites
  - TLS_PSK_WITH_AES_128_CBC_SHA
  - TLS_PSK_WITH_AES_256_CBC_SHA"
- Strict Name Compare
- Extended Distinguished names
- TLS session regnotiation
- TLS session resumption
- User defined Private Keys
- Ciphersuites
  - TLS_RSA_WITH_NULL_SHA
  - TLS_RSA_WITH_NULL_MD5

The following feature has not been integrated into SSP. Please ignore references to it.

- DTLS

Customer Support

For Renesas Synergy platform support, please contact Renesas directly:

Support: www.renesas.com/synergy/support

Installation and Use of NetX Secure

Page 9: If you are using Renesas Synergy SSP and the e^2 studio ISDE, NetX Secure will already be installed. You can ignore the installation section.

NetX Secure

User Guide

Express Logic, Inc.

858.613.6640
Toll Free 888.THREADX
FAX 858.521.4259

www.expresslogic.com

Express Logic, Inc. reserves the right to make changes to the specifications described herein at any time and without notice in order to improve design or reliability of NetX. The information in this document has been carefully checked for accuracy; however, Express Logic, Inc. makes no warranty pertaining to the correctness of this document.

Trademarks

Warranty Limitations

Express Logic, Inc. makes no warranty of any kind that the NetX products will meet the USER's requirements, or will operate in the manner specified by the USER, or that the operation of the NetX products will operate uninterrupted or error free, or that any defects that may exist in the NetX products will be corrected after the warranty period. Express Logic, Inc. makes no warranties of any kind, either expressed or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose, with respect to the NetX products. No oral or written information or advice given by Express Logic, Inc., its dealers, distributors, agents, or employees shall create any other warranty or in any way increase the scope of this warranty, and licensee may not rely on any such information or advice.

Part Number: 000-1059
Revision 5.11SP2

Contents

Chapter 1

Introduction to NetX Secure

NetX Secure is a high-performance real-time implementation of cryptographic network security standards including TLS/SSL designed exclusively for embedded ThreadX-based applications. This chapter contains an introduction to NetX Secure and a description of its applications and benefits.

NetX Secure Unique Features

Unlike most other TLS 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.

RFCs Supported by NetX Secure

NetX Secure supports the following protocols related to TLS. The list is not necessarily comprehensive as there are numerous RFCs pertaining to TLS 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 2246	The TLS Protocol Version 1.0	20
RFC 4346	The Transport Layer Security (TLS) Protocol Version 1.1	20
RFC 5246	The Transport Layer Security (TLS) Protocol Version 1.2	20
RFC 5280	X.509 PKI Certificates (v3)	42
RFC 3268	Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS)	32
RFC 3447	Public-Key Cryptography Standards (PKCS) #1: RSA	33

	Cryptography Specifications Version 2.1	
RFC 2104	HMAC: Keyed-Hashing for Message Authentication	34
RFC 6234	US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)	34
RFC 4279	Pre-Shared Key Ciphersuites for TLS	40

NetX Secure 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 instance or to verify certificates coming from a remote host. The NetX Secure package has no further requirements.

NetX Secure Constraints

The NetX Secure protocol implements the requirements of the RFC 5246 Standard(s) for TLS 1.2, as well as backwards-compatibility with RFCs 4346 (TLS 1.1) and 2246 (TLS 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 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 TLS.

Chapter 2

Installation and Use of NetX Secure

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

Product Version Number

User may verify the product version number by finding the following symbols in nx_secure_tls.h:

NETX_SECURE_MAJOR_VERSION
NETX_SECURE_MINOR_VERSION

Service Pack releases has the following symbol defined to indicate the service pack number:
NETX_SECURE_SERVICE_PACK_VERSION

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_tls_api.h	Public API header file for NetX Secure TLS
nx_secure_tls_user.h	User defines header file for NetX Secure TLS
nx_secure_tls_port.h	Platform-specific definitions for NetX Secure
nx_secure_tls.h	Header file for NetX Secure TLS
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_tls.c	C Source file for NetX Secure TLS Demo
NetX_Secure_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 Installation

In order to use NetX Secure, 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

Using NetX Secure TLS is straightforward. Basically, the application code must include *nx_secure_tls_api.h* after it includes *tx_api.h* and *nx_api.h*, in order to use ThreadX and NetX. Once *nx_secure_tls_api.h* is included, the application code is then able to make the NetX Secure 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.

Small Example System (TLS Client)

An example of how easy it is to use NetX Secure is described in Figure 1.1, which appears below and demonstrates a simple TLS Client, designed to work with the OpenSSL reverse-echo server (openssl s_server -rev).

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

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

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

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

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

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

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

/* Define an HTTP request to send to the HTTPS web server. */
```

```
UCHAR http_request[] = { … };

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

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

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

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

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

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

/* Define the metadata area for TLS cryptography. The actual size needed can be
   Ascertained by calling nx_secure_tls_metadata_size_calculate.
*/
UCHAR tls_crypto_metadata[14000];

/* Pointer to the TLS ciphersuite table that is included in the platform-specific
   cryptography subdirectory. The table maps the cryptographic routines for the
   platform to function pointers usable by the TLS library.
*/
extern const NX_SECURE_TLS_CRYPTO nx_crypto_tls_ciphers;

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

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

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

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

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

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

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

    status =  nx_ip_fragment_enable(&ip_0);

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

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

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

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

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

    /* Check status for errors... */

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

    /* Check status for errors... */

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

    /* Check status for errors...  */

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

    /* Check status for errors...  */


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

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

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

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

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

    /* Check for errors…  */

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

    /* Check for errors…  */

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

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

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

    /* Receive the HTTP response and any data from the server. */
    status = nx_secure_tls_session_receive(&tls_session, &receive_packet,
                                           NX_WAIT_FOREVER);

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

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

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

    /* Check for errors... */

}
```

Figure 1.1 Example of NetX Secure use with NetX

Small Example System (TLS Web Server)

An example of how easy it is to use NetX Secure is described in Figure 1.1, which appears below and demonstrates a simple TLS Web Server (HTTPS).

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

```
#define     DEMO_STACK_SIZE         4096

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

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

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

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

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

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


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

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

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

/* Binary data for the TLS Server X.509 certificate, ASN.1 DER-encoded. */
const UCHAR certificate_data[] = { … }; /* DER-encoded binary certificate. */
const UINT certificate_length[] = 0x574;

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

/* Define some HTML data (web page) with an HTTPS header to serve to connecting
clients. */
const UCHAR html_data[] = { … };

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


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

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

    /* Create an IP instance for the specific target. Check status for errors. */
    status = nx_ip_create(&ip_0, …);
    /* Enable ARP and supply ARP cache memory for IP Instance 0. Check status for
       errors. */
    status =  nx_arp_enable(&ip_0, (void *)arp_space_area, sizeof(arp_space_area));

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

    status =  nx_ip_fragment_enable(&ip_0);

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

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

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

    NX_PARAMETER_NOT_USED(thread_input);

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

    /* Check status for errors... */

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

    /* Check status for errors... */

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

    /* Check status for errors...  */

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

    /* Check status for errors...  */


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

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


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

    /* Check for errors…  */

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

        /* Check for errors… */

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

        /* Check for errors…  */

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

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

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

        /* Check for errors…  */

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

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

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

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

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

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

        /* Check for errors... */

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

        /* Check for errors... */

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

        /* Check for errors…  */
    }
}
```

Figure 1.2 Example of NetX Secure use with NetX

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_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_ALLOW_SELF_SIGNED_CERTIFICATES	Defined, this option allows TLS to accept self-signed certificates from a remote host. Otherwise the certificate must be issued by a certificate in the trusted certificate store (which may include self-signed certificates).
NX_SECURE_TLS_CLIENT_DISABLED	Defined, this option removes all TLS stack code related to TLS Client mode, reducing code and data usage.
NX_SECURE_TLS_SERVER_DISABLED	Defined, this option removes all TLS stack code related to TLS 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).

NX_SECURE_TLS_ENABLE_TLS_1_0

Defined, this option enables the legacy TLSv1.0 mode. TLSv1.0 is considered obsolete so it should only be enabled for backward-compatibility with older applications.

NX_SECURE_TLS_DISABLE_TLS_1_1

Defined, this option disables TLSv1.1 mode. TLSv1.1 is enabled by default, but can be disabled in favor of using only the more-secure TLSv1.2[1].

[1] Note that it is also possible to disable TLSv1.2 if using TLS 1.0 or TLS 1.1 only. However, this is not recommended and not directly supported. If you need to disable TLSv1.2 for any reason, please contact your Express Logic representative.

Chapter 3

Functional Description of NetX Secure

Execution Overview

This chapter contains a functional description of NetX Secure TLS. There are two primary types of program execution in a NetX Secure TLS 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 TCP/IP networking facilities and drivers.

Transport Layer Security (TLS) and Secure Sockets Layer (SSL)

The secure network protocol component of NetX secure is an implementation of the Transport Layer Security (TLS) protocol as described in RFCs 2246 (version 1.0), 4346 (version 1.1), and 5246 (version 1.2). Also includes are support routines for basic X.509 (RFC 5280)

NetX Secure TLS supports TLS versions 1.0, 1.1, and 1.2.

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.*

TLS data is divided into *records* which are equivalent in concept to a TCP packet. Every TLS record has a header, and TLS encrypted records also have a footer (checksum hash). TLS handshake records have an additional header encapsulated within the larger TLS record. The TLS record is encapsulated by the transport layer network protocol in the same manner that a TCP packet is encapsulated by an IP packet.

TLS Record header

Any valid TLS record must have a TLS header, as shown in Figure 1.

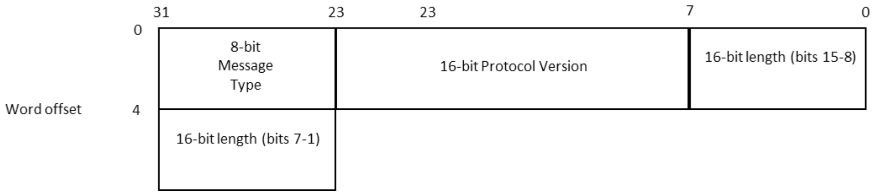

Figure 1 - TLS record header

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

<table>
<tr><th>TLS Header Field</th><th>Purpose</th></tr>
<tr><td><b>8-bit Message Type</b></td><td>This field contains the type of TLS record being sent. Valid types are as follows:
<table>
<tr><th>Type</th><th>Value</th></tr>
<tr><td>ChangeCipherSpec</td><td>0x14</td></tr>
<tr><td>Alert</td><td>0x15</td></tr>
<tr><td>Handshake</td><td>0x16</td></tr>
<tr><td>Application Data</td><td>0x17</td></tr>
</table></td></tr>
<tr><td><b>16-bit Protocol Version</b></td><td>This field contains the TLS protocol version. Valid values are as follows:
<table>
<tr><th>Protocol version</th><th>Value</th></tr>
<tr><td>SSL 3.0</td><td>0x0300</td></tr>
<tr><td>TLS 1.0</td><td>0x0301</td></tr>
<tr><td>TLS 1.1</td><td>0x0302</td></tr>
<tr><td>TLS 1.2</td><td>0x0303</td></tr>
</table></td></tr>
<tr><td><b>16-bit Length</b></td><td>This field contains the length of the data encapsulated in the TLS record.</td></tr>
</table>

TLS Handshake Record header

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

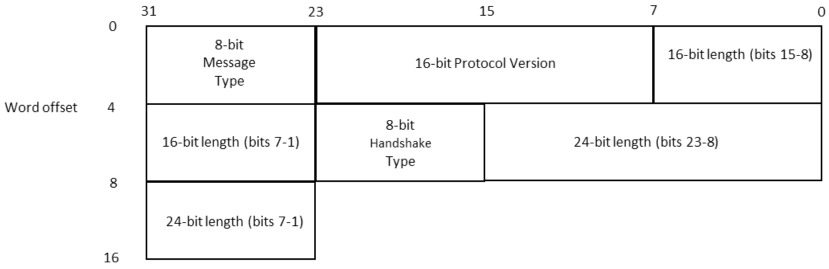

Figure 2 - TLS Handshake record header

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

TLS Header Field	Purpose
8-bit Message Type	This field contains the type of TLS 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 TLS protocol version. Valid values are as follows:

Protocol version	Value
SSL 3.0	0x0300
TLS 1.0	0x0301
TLS 1.1	0x0302
TLS 1.2	0x0303

16-bit Length	This field contains the length of the data encapsulated in the TLS 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.

The TLS Handshake and TLS Session

A typical TLS handshake is shown in Figure 3. A TLS handshake begins when the TLS Client sends a *ClientHello* message to a TLS server, indicating its desire to start a TLS 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 TLS handshake are not encrypted.

The TLS Server responds to the ClientHello 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. 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, 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 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 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). The shared secret is used to generate the Pre-Master Secret instead of using an encrypted message to send the Pre-Master Secret. See the section on Pre-Shared Keys below.

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 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 session begins, and the application begins sending and receiving data. All data sent by either side during the TLS 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 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.

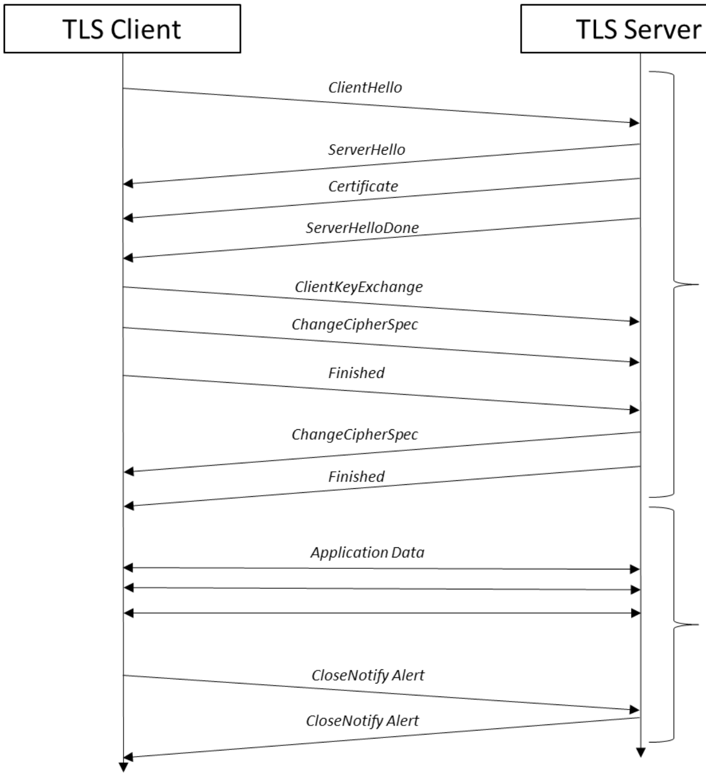

Figure 3- Typical TLS handshake

Initialization

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

Once the NetX TCP/IP stack has been initialized, TLS can be enabled. Internally, all TLS network traffic and processing is handled by the NetX/NetXDuo stack without requiring user intervention. However, TLS has some specific requirements that must be handled separately from the underlying network stack. These parameters are assigned to the TLS control block called ***NX_SECURE_TLS_SESSION*** using the ***nx_secure_tls_session_create*** service.

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

In either mode, NetX Secure TLS requires a TCP socket (***NX_TCP_SOCKET***) to be created and set up for TCP communications with the remote host. The TCP socket is assigned to a TLS session instance with the ***nx_secure_tls_session_start*** service, detailed below.

Initialization – TLS Server

In addition to a TCP socket, NetX Secure TLS Server mode requires 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 and there are numerous utilities for creating X.509 digital certificates.

For NetX Secure TLS, 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 communications!

To initialize the TLS 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_tls_local_certificate_add*** service.

Once the server's certificate has been added to the TLS control block, the socket may be used to establish a secure TLS Server connection.

Initialization – TLS Client

NetX Secure TLS Client mode 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 TLS protocol to be "trusted" and serve as the basis for authenticating certificates provided by TLS server entities to NetX Secure TLS Client.

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 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.

Trusted certificates that have been initialized are then added to the TLS control block using the ***nx_secure_tls_trusted_certificate_add*** service. Failure to add a certificate will cause the TLS Client session to fail as there will be no way for the TLS protocol to authenticate remote TLS server hosts.

The TLS Client also needs space for the incoming server certificate to be allocated (assuming a Pre-Shared Key mode is not being used). To allocate space for a server certificate, a certificate data structure (***NX_SECURE_X509_CERT***) must be initialized with enough space to hold the incoming certificate. This certificate is then associated with the TLS session using the ***nx_secure_tls_remote_certificate_allocate*** service.

Once the Trusted Certificate Store has been created and space for the server certificate has been allocated, the socket may be used to establish a secure TLS Client connection.

Application Interface Calls

NetX Secure TLS 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. TCP and IP) may be called from ***tx_application_define*****.** See the NetX/NetXDuo User Guide for more information on initializing network communications.

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

TLS Session Start

TLS 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, a TCP connection must be established using the NetX/NetXDuo TCP API. An **NX_TCP_SOCKET** must be created and a connection established using the ***nx_tcp_server_socket_listen*** and ***nx_tcp_server_socket_accept*** services (for TLS Server) or the ***nx_tcp_client_socket_connect*** service (for TLS Client).

Once a TCP connection has been established, the TCP socket is then passed to the ***nx_secure_tls_session_start*** service.

TLS Packet Allocation

NetX Secure TLS uses the same packet structure as NetX/NetXDuo TCP (***NX_PACKET***) except that instead of calling the ***nx_packet_allocate*** service, the ***nx_secure_tls_packet_allocate*** service must be called so that space for the TLS header may be allocated properly.

TLS Session Send

Once the TLS session has started, the application may send data using the ***nx_secure_tls_session_send*** service. The send service is identical in use to the ***nx_tcp_socket_send*** service, taking an ***NX_PACKET*** data structure containing the data being sent, only that data will be encrypted by the NX Secure TLS stack before being sent, and the packet must be allocated using ***nx_secure_tls_packet_allocate***.

TLS Session Receive

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

TLS Session Close

Once a TLS session is complete, both the TLS 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_tls_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 TLS session will fail.

If the application wishes to close the TLS session, the ***nx_secure_tls_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 TLS session was not cleanly shutdown, a possible security breach.

TLS Alerts

TLS 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 will send an *Alert* message to the remote host upon any error.

Alerts are treated in the same manner as any other TLS 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 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 Session Renegotiation

TLS supports the notion of "renegotiation" which is simply a renegotiation of the TLS session parameters within the context of an existing TLS session. What this means in practice is that the new handshake messages are encrypted and authenticated using the existing session. Renegotiation is used when a TLS host wants to generate new session parameters (e.g. generate new TLS session keys) without having to complete the existing session. For example, renegotiation may be desirable when security policies for an application dictate that session keys are only used for a limited time but a TLS session remains active beyond that time.

One issue with session renegotiation is that is makes TLS vulnerable to a specific Man-in-the-Middle attack where an attacker can convince a server to initiate a renegotiation with new parameters, thus allowing the attacker to hijack the TLS session. To mitigate this issue, the Secure Renegotiation Indication extension was introduced (see section Secure Renegotiation Indication section).

NetX Secure TLS completely supports session renegotiation and the Secure Renegotiation Indication extension.

When receiving data from a remote host, renegotations (and the extension) are handled automatically without application interaction. If notification about session renegotiations is desired, a renegotiation callback may be supplied with the *nx_secure_tls_session_renegotiate_callback_set* service. The callback will be invoked whenever a renegotiation is requested by the remote host, allowing the application to take action if desired.

To initiate a renegotiation from an active TLS session, simply invoke the *nx_secure_tls_session_renegotiate* service on the desired TLS session.

TLS Session Resumption

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. To achieve this, a TLS implementation may cache the session parameters and keys, associating them with a *session ID,* a unique identifier supplied in the original handshake. By supplying a session ID to a TLS server, a client indicates that a previous TLS session between the hosts existed and completed some time in the past, and that the client still possesses the state to re-establish the session with a reduced handshake. Since the session keys are theoretically still secret and only known by the two communicating host, the server can start a new TLS session and bypass most of the normal handshake.

Session resumption can be useful to avoid the potentially expensive public-key operations used to share the key generation master secret and verify certificate signatures, but it also requires that the session parameters, keys, and crypotgraphic state be maintained in memory for all possible sessions (at least for a configurable time window).

The current version of NetX Secure TLS does not support session resumption – the session ID is simply ignored by TLS servers and TLS clients always supply a NULL session ID which prompts the server to perform a complete handshake. The lack of session resumption should cause no inter-operability issues as it is a completely optional feature and all TLS implementations must default to a complete handshake should the session ID be NULL or unrecognized.

Protocol Layering

The TLS protocol fits into the networking stack between the transport layer (e.g. TCP) 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 (such as TCP) 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. A separate protocol called *DTLS,* which is a modified version of TLS, is used for applications that need the security of TLS over a datagram protocol like UDP. This version of NetX Secure does not support DTLS.

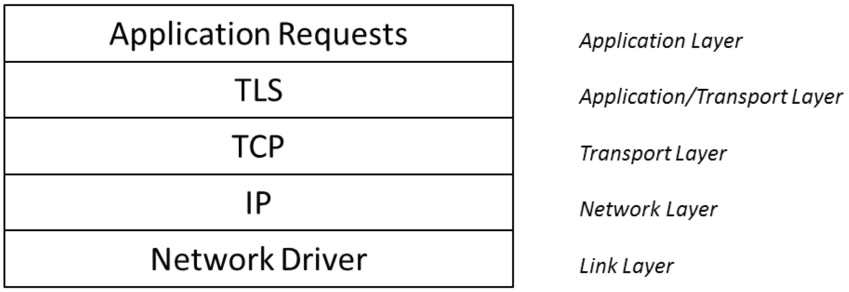

Figure 4- TCP/IP and TLS protocol layers

Network Communications Security

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 protocol provides solutions for all three.

Secrecy

When sending data over a network, it is often important that the data cannot be obtained by a malicious entity. If data is sent over a TCP/IP connection, anyone with access to the network will be able to read that data using easily-available networking tools. To prevent that data from being obtained, it must be encoded such that it cannot be read except by the intended target – this is *secrecy.* In TLS, encryption algorithms such as RSA and AES provide secrecy.

Integrity

Sometimes, secrecy is not enough to protect data travelling over a network. In some cases, it may be possible for a malicious entity to alter the contents of a TCP packet without needing to know what that packet contains. Encrypted data can be altered, rendering the decryption invalid or changing the parameters of the message leading to whatever result the attacker may be interested in achieving. On the network, we cannot prevent an attacker from changing data in transit, but we can provide a mechanism to know whether or not the data has been changed. When data is changed in transit, it will be known and the data can be rejected. This concept is *integrity*. In TLS, integrity is provided by a class of cryptographic routines known as *hash functions*. Some examples of hash functions are MD5 and SHA-1.

Authentication

The third important concept in network communications security is the idea that data should only be communicated to the intended target. An attacker may attempt to pose as a legitimate entity to receive data intended for another host. Even if the data is being sent with secrecy and integrity mechanisms in place, the attacker may still be able to achieve the desired result (a compromise of secure communications) through this deception. To prevent this, a mechanism is needed to prove the identity of a remote host before any sensitive data is sent. The process of proving the identity of a remote host is *authentication.* In TLS, authentication is provided using digital certificates, hash functions, and a mechanism called *digital signatures* which utilizes a property of public-key encryption (described below). A limited but useful form of authentication can also be provided with a *pre-shared key (*PSK).

TLS Encryption

The TLS protocol is a framework for providing secure network communications over the Internet utilizing encryption. Encryption is generally defined as the process of encoding data in such a way that obtaining the original data (or information about that data) is exceedingly difficult without a *key*. In computer systems encryption is based on complex mathematics such as finite fields and can be classified into two types: *private key (*or *symmetric encryption*) and *public key (*or *asymmetric encryption)*. Examples of private key encryption are AES (Advanced Encryption Standard) and RC4 (Rivest Cipher 4). Examples of public-key encryption are the RSA (Rivest, Shamir, Adleson) and Diffie-Hellman ciphers.

The TLS protocol makes use of both private key and public key encryption routines to provide a balance of performance, security, and flexibility.

Private-Key Encryption

Private-key encryption has been in use for thousands of years. Basic substitution ciphers (where a letter or word is replaced by another unrelated letter or word) are the earliest known examples of encryption, but with the advent of the information age private key encryption has significantly improved.

A private key cipher uses a “key” which is simply a value (which could be a word, phrase, or number in the general case) that is used to somehow encode some data so that only an entity that had access to that key could decode the data in a meaningful way. The key is used for both encryption and decryption of the data, hence the other name *symmetric encryption*.

Private key ciphers are generally fast and fairly simple to implement, even if the mathematics involved are exceedingly complex. For this reason, TLS uses private key ciphers for the bulk of secure communications.

However, private key encryption has a problem when we try to apply it to general computer network communications: the key must be shared between both machines trying to communicate. In the general case, it is impractical and often impossible to communicate a private key securely between two machines on the Internet, as it can be assumed that the network traffic can be obtained by any number of entities in the various hops that data takes when being routed through the Internet. If the key is obtained by a malicious entity, all data encrypted using that key is compromised. As most machines on the Internet have only a network connection and not another secure channel for communications, sending keys over the network is tantamount to sending the data unencrypted – it provides no security.

For this reason, private key encryption is not sufficient to implement a general-purpose network communications security protocol. This is where Public Key encryption can help.

NetX Secure TLS supports AES private-key encryption.

Public-Key Encryption

Unlike private key encryption, public key encryption is a fairly new concept, having been developed in the 1970's. Using a concept known as "trap-door functions" in mathematics, it was discovered that there was a way to share a key over a network without compromising the security of then encrypted data.

The way public key encryption works is that the key (in the private-key encryption sense described above) is split into two parts, a *private key* and a *public key*, from where public key encryption gets its name. The concept is that one of these keys (typically the public key) is used for encryption, while the other is used for decryption. This asymmetry of keys is the reason for the other name for public key encryption: *asymmetric encryption*.

The mathematics behind public key encryption are fairly complex, but the idea is that the public key can *only* be used for encryption, and obtaining that key does not allow encrypted data to be obtained. The private key, in turn, is the only way to decrypt data encrypted using the public key. Thus, by keeping the private key secret, anyone wishing to communicate securely with the owner of that private key need only encrypt their data with the corresponding public key with the knowledge that only someone in possession of that private key can obtain the secure data.

NetX Secure TLS supports RSA public-key encryption.

RSA is a very processor-intensive operation if the software RSA implementation is used. Larger key sizes increase the processing power required by a square factor – 4X slower for a 2X increase in key size.

Public-Key Authentication

An interesting side-effect of the public-key encryption concept is that it can be used to provide authentication as well as encryption by doing the operation in reverse: encrypting using the *private* key and decrypting using the *public* key. The actual mechanism for doing this depends on the public key algorithm being used, but the concept is the same.

To authenticate using public key authentication, the owner of a private key encrypts some piece of data (typically a cryptographic hash of the data to be authenticated) using that private key. Then, someone wishing to authenticate that the data came from the owner of the private key uses the associated public key to decrypt the data – if the decryption is successful, and assuming the user trusted the validity of that public key, then the user can be certain that the data came from the owner of the private key.

In TLS, public key authentication is used to verify the validity of a digital certificate provided by a TLS server (and optionally the TLS client) using public keys from the trusted certificate store. The certificate is checked against a public key in the store and the data in the certificate is used to check the identity of the server.

NetX Secure TLS supports RSA authentication.

Cryptographic Hashing

Encryption is not the only cryptographic operation used in TLS. In order to provide message integrity during a TLS session, a checksum is needed to ensure that the message contents have not been tampered with. However, a simple checksum (as is used in TCP) is insufficient to guarantee an acceptable level of integrity as it can be easily subverted by a knowledgeable attacker. The mechanism used by TLS to provide message integrity is known as a *cryptographic hash.*

Encryption is a 1:1 encoding – that is, the entirety of the original data can be obtained from the encrypted data. However, a hash maps an arbitrary amount of data into a fixed size value, just like a checksum. Unlike a simple checksum, a hash is specifically designed to reduce *collisions*, where different input data result in the same output. In a simple checksum, if a bit is flipped from 1 to 0 and another bit from 0 to 1, the checksum is the same. With a cryptographic hash, the output would differ significantly, making it difficult for an attacker to change the hashed data and have the hash operation on the changed data still result in the same value (and thus falsely verifying the integrity of that data).

TLS uses a number of different hash algorithms to provide integrity for messages, both application messages and TLS control messages. These include MD5, SHA-1 and SHA-256.

NetX Secure TLS supports MD5, SHA-1, and SHA-256 hashing.

TLS Extensions

TLS 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.

In general, extensions provide optional parameters to TLS at the beginning of the handshake that guide the proceeding operations. Some extensions require application input or decision making, while others are handled automatically.

The following table describes the TLS extensions currently supported by NetX Secure TLS:

Extension Name	Description
Secure Renegotiation Indication	This extension mitigates a Man-in-the-Middle attack vulnerability that could occur during a renegotiation handshake.
Server Name Indication	This extension allows a TLS Client to supply a specific DNS name to a TLS Server, allowing the server to select the correct credentials (assumes the server has multiple identity certificates and network entrypoints).
Signature Algorithms	This extension enables a TLS Client to provide a list of acceptable signature and hash algorithms to a TLS Server.

Overview of supported TLS Extensions

Secure Renegotiation Indication

TLS supports the notion of performing a handshake within an existing TLS session, thereby using the established session to encrypt the handshake messages. This process allows the cryptographic session keys to be re-established without ending the TLS session (see section "TLS Session Renegotiation").

Unfortunately, after TLS had been using renegotiation for some time, it was discovered that there was a vulnerability to a Man-in-the-Middle attack that exploited the renegotiation feature. To close the vulnerability, the Secure Renegotiation Indication extension was introduced. Essentially, the Secure Renegotiation extension uses the Finished message hash from the established connection to verify that the original hosts are participating in the renegotiation handshake – essentially the hash is used as a verification token under the assumption that an attacker would not be able to forge the hash (which would require access to the session keys).

NetX Secure TLS handles renegotiation automatically and uses the Secure Renegotiation Extension by default. No application interaction is required.

Server Name Indication

During the TLS handshake, a TLS Client expects a remote server to provide an identity certificate so the client can authenticate the server. However, there may be some cases where a server will provide multiple different services with different “virtual” servers each having unique identities. In the case of a single server with multiple identities, a TLS client can supply a specific DNS name that the server will use to select the proper credentials – the mechanism for supplying this name is the Server Name Indication (SNI) extension.

For an application using the SNI extension, some interaction is required. For TLS Clients, the application must supply a DNS name to be sent to the remote server. For TLS Servers, the application must read the DNS name from the extension and select an appropriate certificate to send back to the client.

The following sections provide more detail on how to use the SNI extension in NetX Secure TLS.

SNI Extension – TLS Client

A NetX Secure TLS Client wishing to use the SNI extension must provide a DNS name to TLS to be supplied during the handshake. This name must be initialized and supplied prior to starting a TLS session since the extension is sent in the ClientHello message which starts the handshake process.

The following code snippet illustrates the use of the extension. First, a NX_SECURE_X509_DNS_NAME object is initialized with the desired server name. Then, prior to starting the TLS session, the name is provided to TLS using the SNI extension API. Once the name is set, no further action is required. See the API reference in Chapter 4

Description of NetX Secure Services for more information on the individual functions.

```
/* The dns_name variable will contain our desired server name. */
UINT status;
NX_SECURE_X509_DNS_NAME dns_name;

/* Initialize the server DNS name. */
status = nx_secure_x509_dns_name_initialize(&dns_name, “www.example.com”,
                                             strlen(“www.example.com”));


/* Initialize SNI extension in previously-initialized TLS Session. */
```

```
status = nx_secure_tls_session_sni_extension_set(&client_tls_session, &dns_name);

/* Now start the TLS session, starting with establishing the TCP connection - if
   TLS is started before initializing the SNI extension, the extension will not be
   sent in the ClientHello message! */
status = nx_tcp_client_socket_connect(&client_socket, IP_ADDRESS(1, 2, 3, 4), 443,
                                       5 * NX_IP_PERIODIC_RATE);

status = nx_secure_tls_session_start(&client_tls_session, &client_socket,
                                      NX_WAIT_FOREVER);
```

SNI Extension – TLS Server

On the TLS Server side, the SNI extension may be processed by the application in order to select proper credentials (e.g. certificate) to provide to the remote client during the handshake. To do this, the application must supply a session callback which is invoked following the receipt of a ClientHello message.

The example code for the nx_secure_tls_session_server_callback_set API (see page 135) illustrates the parsing of an incoming SNI extension using a server callback. Essentially, the TLS Server receives a ClientHello and invokes the callback. Then the application uses the *nx_secure_tls_session_sni_extension_parse* API to parse the extension data provided to the callback to find the SNI extension and return the supplied DNS name (note that the extension only supports a single DNS name). Once the name is obtained, the application uses it to find and send the appropriate server identity certificate (and issuer chain if applicable).

Signature Algorithms Extension

This extension is specific to TLS 1.2 and allows a TLS Client to provide a list of acceptable signature and hash algorithm pairs that are acceptable for use in generating and verifying digital signatures. The list is generated automatically by NetX Secure TLS for TLS Clients using the cipher table supplied to *nx_secure_tls_session_create*. No application interaction is required.

Authentication Methods

TLS provides 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 TLS API can be used to associate your certificate with a TLS session.

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

Refer to the following TLS services in the API reference for more information:
nx_secure_x509_certificate_initialize, nx_secure_tls_local_certificate_add,
nx_secure_tls_local_certificate_remove,
nx_secure_tls_remote_certificate_allocate,
nx_secure_tls_trusted_certificate_add,
nx_secure_trusted_certificate_remove

TLS Client Certificate Specifics

TLS Client implementations generally do not require a “local” certificate[2] to be loaded onto the device. The exception to this is when Client Certificate Authentication is enabled, but this is far less common.

A TLS Client requires at least one “trusted” certificate[3] to be loaded (more may be loaded if required), and space for a “remote” certificate[4] 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_tls_remote_certificate_allocate, nx_secure_tls_trusted_certificate_add.

[2] A “local” certificate is a certificate that identifies the local device – that is, it provides identity information for the device upon which the TLS application is loaded.

[3] 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.

[4] 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.

TLS Server Certificate Specifics

TLS 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 (this is less common).

A TLS Server requires a “local” 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.

For more information about loading local certificates for use with NetX TLS server applications, see the API reference for the following services:
nx_secure_tls_local_certificate_add,
nx_secure_tls_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 handshake and is used in place of the encrypted Pre-Master Secret for TLS 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 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 TLS application.

With PSK support enabled, you can then use the TLS 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_tls_client_psk_set, nx_secure_tls_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.

Certificate Types

Digital certificates are generally used to identify entities on a network, but depending on what their application they will have slightly different properties.

Local Certificates

For the purposes of this documentation, we will refer to “local certificates” as those certificates which provide an identity for our local device (another possible name could be “device certificate”). These certificates will be provided to a remote host when the remote host desires to authenticate the local device.

Remote Certificates

In this documentation, “remote certificates” refers to those certificates provided by a remote host during the TLS handshake when applicable. Space for these certificates must be allocated or NetX Secure will not be able to parse them and complete the TLS handshake.

Signing Certificates

A “signing certificate” is used to digitally sign other certificates or data for the purpose of authentication. These certificates may be either intermediate or root certificates within a Public Key Infrastructure (PKI) and are generally not used to identify individual devices or hosts.

Root CA Certificates

“Root CA certificates” are signing certificates that provide the basis of a PKI and are self-signed, rather than being signed by another signing certificate. At least one Root CA certificate is typically required for a TLS Client to verify remote servers.

Certificate formats

Digital certificates are simply files containing structured data encoded using the ASN.1 syntax. However, there are various formats in which certificates may be stored and it is important to have the right format before loading a certificate into a NetX Secure application.

The most common formats for certificates are DER and PEM. DER (for *Distinguished Encoding Rules*, an ASN.1 format) is the binary format used by TLS when performing the initial handshake. PEM (from *Privacy Enhanced Mail*) is a base-64 encoded version of the DER format which is suitable for emailing or sending over HTTP on the web. Different vendors use different filename extensions for certificates, such as ".pem" or ".crt" for PEM certificates, and ".der" for DER certificates. If you have a certificate and it is not clear what format is used, opening the file in a text editor will allow you to determine the type since DER files are encoded binary, and PEM files are regular ASCII text that start with the header "-----BEGIN CERTIFICATE-----".

NetX Secure requires that your certificate be in binary DER format, so you will need to convert your certificate into DER format before importing. This can be done with readily available tools such as OpenSSL.

If you need a private key for your application, the key file will be encoded using PEM or DER in PKCS#1 format. The private key file will need to be converted into DER before being imported.

The following OpenSSL commands are given as an example for converting certificates and key files into the DER format required by NetX Secure.

```
openssl x509 -inform PEM -in <certificate> -outform DER -out cert.der
openssl x509 -inform PEM -in <root CA cert> -outform DER -out CA_cert.der
openssl rsa -inform PEM -in <private key> -outform DER -out private.key
```

Private Keys and Certificates

For certificates that identify a device, the associated private key must be loaded along with the certificate. The private key (which may be for one of the public-key algorithms such as RSA, Diffie-Hellman, or Elliptic-Curve Cryptography) is used by a TLS server to decrypt the incoming key material (the "pre-master secret") from a TLS client, thereby authenticating itself to the client. For a TLS Client, if an identity certificate (a certificate with its associated private key) is provided and a server requests a client certificate, the private key is used to authenticate the client – in the case of RSA the client encrypts a token using the private key which the server then decrypts using the client's public key, provided in the client certificate (Diffie-Hellman and ECC authentication happens in a similar fashion but the details are a bit different).

In NetX secure, the service *nx_secure_x509_certificate_initialize* is used to initialize an X.509 certificate (see section "Loading certificates onto your device" for more information) and optionally associate a private key with that certificate.

If a private key is supplied, the certificate is marked as being the "identity" certificate used to identify the device. The key is passed as a contiguous binary blob and a length, with an associated key type. The key type depends on the type of key (e.g. RSA, ECC, etc.) and the format (e.g. PKCS#1 DER). If no key is supplied, the value

NX_SECURE_X509_KEY_TYPE_NONE (value 0x0) can be passed to indicate no key is being supplied (a length of 0 and a NX_NULL pointer for the data parameter will achieve the same effect).

The following table shows the key types known to NetX Secure and the associated type identifier to be passed into *nx_secure_x509_certificate_initialize*. Additional key types will be added as more encryption algorithms are added to NetX Secure.

Identifier	Algorithm	Format	Encoding	Value
NX_SECURE_X509_KEY_TYPE_NONE	None	N/A	N/A	0x0
NX_SECURE_X509_KEY_TYPE_RSA_PKCS1_DER	RSA	PKCS#1	DER	0x1

User-defined private key types

The values of the key type identifiers for the *nx_secure_x509_certificate_initialize* service govern the actions taken when the private key is supplied. For known types, the values are in the range 0x0000 0000 – 0x0000 FFFF (bottom 16 bits of a 32-bit unsigned integer). For platforms with custom key types[5] (as is the case for some hardware-based encryption engines), a user-defined key type in the range 0x0000 1000-0xFFFF FFFF (top 16 bits non-zero) may be passed as the key type. If any of the top 16 bits of the key type are set, then the private key data is passed directly to the appropriate cryptographic routine (e.g. RSA) supplied in the TLS ciphersuite table. User-defined key types are not parsed or otherwise processed before being passed to the cryptographic routine. In addition, the user-defined key type will also be passed to the cryptographic routine so that any appropriate processing can be handled at that level.

Note that user-defined key types are generally used for specific hardware platforms that utilize custom (possibly encrypted) key data. Generally this implies that the key data is generated or encoded using a mechanism specific to that hardware vendor (or in the case of a standard like PKCS#11, a specific standard). Consult your hardware platform documentation for more information.

Loading certificates onto your device

Any method for loading a file onto your device will be sufficient to import your certificates.

The simplest method for loading a certificate is to convert the binary DER-encoded data into a C array and compile it into your application as a constant. This can be easily done with tools such as “xxd” in Linux (with the “-i” option).

[5] User-defined key types require a corresponding custom cryptographic routine to handle the custom key format. The cryptographic routine must have a matching algorithm (e.g. RSA) and be passed into TLS in the ciphersuite table.

Alternatively, you can load your certificate into a flash filesystem or other storage options as long as you can pass a pointer to the certificate data into the NetX Secure API.

Certificate files needed for NetX Secure

The certificate files you will need to import depends on your application. In general, TLS Servers require a certificate to identify the device, and TLS Clients require one or more *Trusted Certificates* to authenticate remote servers. The following table illustrates certificates needed for some different TLS applications.

TLS functionality/options	**Certificates/keys needed (minimum)**
TLS Client	Root CA certificate
TLS Server	Local certificate, private key for that certificate
TLS Server with Client Certificate Authentication	Local certificate, private key, Root CA
TLS Client with Client Certificate Authentication	Local certificate, private key, Root CA
TLS Client or Server with Pre-Shared Keys only	None (PSK used instead of certificates)

The relevant services for loading certificates are as follows:

API Name	**Purpose**
nx_secure_x509_certificate_initialize	Must be called for all certificates to populate the NX_SECURE_X509_CERT structure with your certificate data and private key.
nx_secure_tls_local_certificate_add	Add a local certificate to a TLS session to identify your device.
nx_secure_tls_local_certificate_remove	Remove a local certificate from a TLS session.
nx_secure_tls_remote_certificate_allocate	Allocate space for a remote certificate (called with an uninitialized NX_SECURE_X509_CERT).
nx_secure_tls_trusted_certificate_add	Add a certificate to a TLS Session as a Trusted Certificate for authenticating remote hosts.
nx_secure_tls_trusted_certificate_remove	Remove a trusted certificate from a TLS Session.

Working with AWS IoT Certificates

In the Amazon Web Services IoT interface, select “Security” from the sidebar menu and select “Certificates”. Create a new certificate and follow the instructions to download your new device certificate.

Once you have downloaded your certificates, you will need to convert them into DER format using OpenSSL or a similar utility.

NOTE: AWS will also provide a public key file. The public key is contained within the local device certificate so it does not need to be imported into your application.

As an example, here are the commands to convert the local device certificate and its private key into DER format for use with NetX Secure:

```
openssl x509 -inform PEM -in <certificate> -outform DER -out cert.der
openssl x509 -inform PEM -in <root CA cert> -outform DER -out CA_cert.der
openssl rsa -inform PEM -in <private key> -outform DER -out private.key
```

The converted files can be imported into your application following the above instructions.

X.509 Certificate Validation in NetX Secure

When using TLS with X.509 certificates for host identification and verification, it is important to understand how those certificates are actually validated. While the TLS specification does not provide detailed instructions on how to validate a certificate, it does refer to the X.509 specification (RFC 5280). In general, it is expected that TLS will perform at least basic validation on incoming certificates (those certificates supplied by the remote host during the TLS handshake), and NetX Secure TLS is no different.

Basic X.509 Validation

For any incoming certificate, NetX Secure TLS will perform basic X.509 path validation. The process involves checking each certificate's digital signature against its issuer certificate, which may be provided by the remote host or be located in the trusted certificate store (see the section "Importing X.509 certificates into NetX Secure" for more information on importing trusted certificates). The validation process is recursively repeated on the issuer certificates until a trusted certificate is reached or the chain ends (with a self-signed certificate or a missing issuer certificate). If a trusted certificate is reached, the certificate is verified, otherwise it is rejected. Additionally, at each stage in the verification process the expiration date of each certificate is checked against the time provided by the application timestamp function (see the service "nx_secure_tls_timestamp_function_set" for more information).

The X.509 specification also provides an algorithm for supporting "policies", which are identifiers that are present in an X.509 extension that can be checked during path validation. NetX Secure currently treats X.509 certificates as though the "anyPolicy" option is defined – that is, all policies are acceptable and the optional policy checking is not performed. The NetX Secure X.509 implementation may be augmented with this feature in a future release. For now, the policy extension may be obtained from a certificate using the *nx_secure_x509_extension_find* API.

Once the basic path validation is complete, TLS will invoke the certificate verification callback supplied by the application using the *nx_secure_tls_session_certificate_callback_set* API. If no callback is supplied, the certificate is considered to be trusted following successful path validation. If a callback is supplied, the callback will perform any additional validation of the certificate required by the application. The return value from the callback is used to determine whether to continue with the TLS handshake or to abort the handshake due to a validation failure.

The callback is invoked with a pointer to the relevant TLS session and an NX_SECURE_X509_CERT pointer to the certificate to be validated. Between the TLS session and the certificate, the application has all of the data it needs from TLS to perform additional verification checks.

To help with the additional validation, NetX Secure provides X.509 routines for some common validation operations, including DNS validation and Certificate Revocation List checking. All of these routines are suitable for use within the certificate verification callback but may also be used to perform off-line checking of X.509 certificates.

The following table summarizes the available helper functions for X.509 certificate processing. More detailed explanations for the operations can be found in the following sections and the API reference in Chapter 4

Description of NetX Secure Services provides additional details on the specific routines.

API Name	Description
nx_secure_x509_common_name_dns_check	Check the X.509 subject Common Name and SubjectAltName against an expected DNS name
nx_secure_x509_crl_revocation_check	Check for a revoked certificate in an X.509 Certificate Revocation List (CRL)
nx_secure_x509_extended_key_usage_extension_parse	Parse and find a specific extended key usage OID in a certificate
nx_secure_x509_key_usage_extension_parse	Parse and return the key usage bitfield in a certificate
nx_secure_x509_extension_find	Find and return the raw DER-encoded ASN.1 data for a specific extension.

X.509 helper functions for use in the certificate verification callback

X.509 Extensions

The X.509 specification describes a number of “extensions” that can be used to supply additional information that can be utilized in the verification of certificates. For the most part, these extensions are optional and are not required for secure validation of a digital certificate against a trusted root certificate. However, NetX Secure does support some basic extensions. Support for additional extensions may be added in future releases.

The currently supported extensions are listed in the following table:

Extension Name	Description	Relevant API

Key Usage	Provides acceptable uses for a certificate’s public key in a bitfield	nx_secure_x509_key_usage_extension_parse
Extended Key Usage	Provides additional acceptable uses for a certificate’s public key using OIDs	nx_secure_x509_extended_key_usage_extension_parse
Subject Alternative Name	Provides alternative DNS names that are also represented by the certificate	nx_secure_x509_common_name_dns_check

Unsupported X.509 Extensions

NetX Secure’s X.509 implemenation does provide a service to extract unsupported extensions as well: *nx_secure_x509_extension_find*. This API is intended for advanced users as it requires knowledge of DER-encoded ASN.1 in order to parse the data returned. It it used internally to extract supported extensions but is supplied for convenience in developing customized support for X.509 extensions.

To use nx_secure_x509_extension_find, a NX_SECURE_X509_EXTENSION is passed in, along with the certificate and an extension ID, which is an integer representation of the variable-length OID string for a known extension type. A complete list of supported OIDs for X.509 extensions is provided in the API reference for nx_secure_x509_extension_find on page 166.

The NX_SECURE_X509_EXTENSION structure is defined as follows:

```
typedef struct NX_SECURE_X509_EXTENSION_STRUCT
{
    /* Identifier (maps to OID) for this extension. */
    USHORT nx_secure_x509_extension_id;

    /* Critical flag - boolean value. */
    USHORT nx_secure_x509_extension_critical;

    /* Pointer to DER-encoded extension data. */
    const UCHAR *nx_secure_x509_extension_data;
    ULONG        nx_secure_x509_extension_data_length;
} NX_SECURE_X509_EXTENSION;
```

When the service returns successfully, the structure will be populated with the relevant data from the certificate. The nx_secure_x509_extension_id field is generally used for internal purposes but will be populated with the relevant OID integer representation. The nx_secure_x509_extension_critical field exposes the X.509 critical extension flag value (Boolean). The nx_secure_x509_extension_data and nx_secure_x509_extension_data_length fields contain a pointer to the DER-encoded ASN.1 data for the extension, and the length of that data, respectively.

Actual parsing of the extension ASN.1 data is beyond the scope of this document, but if you have access to the NetX Secure TLS source you can see how the parsing is done wherever nx_secure_x509_extension_find is called for supported extensions.

X.509 DNS Validation

A common certificate validation operation in TLS involves checking the Top-Level Domain (TLD) name of a remote host against the X.509 certificate provided by that host during the TLS handshake. This operation helps to ensure that the certificate does indeed match the host server that provided it, assuming the DNS lookup can be trusted.
In NetX Secure TLS, this functionality is provided by the service **nx_secure_x509_common_name_dns_check**, which takes the certificate and a string containing the TLD portion of the URL used to access the host. The TLD is compared to the certificate's Common Name field and if it matches, NX_SUCCESS is returned. If the Common Name does not match, the routine will also check for the existence of the X.509 certificate extension *subjectAltName*. If a subjectAltName is present, any DNSName entries in the extension are also checked against the provided TLD. Again, if any match, NX_SUCCESS is returned. If no match is found, an error suitable for returning from the certificate validation callback is returned.

X.509 Key Usage and Extended Key Usage Extensions

The X.509 Key Usage and Extended Key Usage extensions provide information on how a certificate's public key may be used when authenticating that certificate. The key usage is supplied by the certificate's issuer when the certificate is signed and issued. The key usage may be used by a TLS host to check that the certificate is authorized to be used to authenticate a remote TLS host and perform other operations.

The Key Usage extension consists of a simple bitfield where each of the bits represents a specific key usage. A complete list of these values is provided in the API reference for *nx_secure_x509_key_usage_extension_parse* on page 171. For a more complete description of the key usage bits and their meanings, refer to RFC 5280, section 4.2.1.3.

The Extended Key Usage extension, like the Key Usage extension, provides acceptable key use information. However, in order to support arbitrary usages, the Extended Key Usage extension utilizes OIDs instead of a bitfield. When parsing an Extended Key Usage extension in NetX Secure X.509, an integer representing the OID is supplied by the application – the *nx_secure_x509_extended_key_usage_extension_parse* service will then return whether that OID is present. A complete list of supported OIDs for Extended Key usage is provided in the API reference for *nx_secure_x509_extended_key_usage_extension_parse* on page 163. For a more complete description of the OIDs and their meanings, refer to RFC 5280, section 4.2.1.12.

X.509 CRL Revocation Status Checking

X.509 provides a mechanism called the *Certificate Revocation List* (CRL) that allows a digital certificate signing authority to revoke the validity of certificates it has signed. Any application that needs to verify certificates from a signing authority can obtain a CRL and compare any certificates signed by that authority (issuer) against the CRL to see if they have had their status revoked for some reason (such as compromised private key). In this way, the application can avoid using potentially dangerous certificates that pass other certificate validation checks.

Obtaining a CRL is done by an application by downloading the DER-encoded list from a pre-defined server or through some other means. The actual setup varies from issuer to issuer so NetX Secure does not provide a mechanism for obtaining CRLs, but it does provide a routine to check a certificate against a CRL, **nx_secure_x509_crl_revocation_check**.

The API takes a DER-encoded CRL, a certificate store (such as the one in a TLS session) to check against, and the certificate to be checked. The routine first validates the CRL itself against the trusted store (part of the certificate store provided by the application). This is important to protect against fraudulent CRLs being used for Denial-of-Service attacks and establishes that the CRL is actually from the proper issuer. Following the CRL validation, the issuer is checked – if the issuer of the CRL does not match the issuer of the certificate, then the CRL is not valid for that certificate and an error is returned. It is up to the application to determine whether the TLS handshake can continue at this point. If the issuers do match, then the CRL is searched for the serial number of the certificate being validated. If the serial number is present in the list, an error indicating that the certificate has been revoked is returned. If no match is found, NX_SUCCESS is returned.

Client Certificate Authentication in NetX Secure TLS

When using X.509 certificate authentication, the TLS protocol requires that the TLS Server instance provide a certificate for identification, but by default the TLS 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 optionally allows for the TLS Server instance to request a certificate from the remote Client. When this feature is enabled, the TLS Server will send a CertificateRequest message to the TLS 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 TLS Clients

A TLS 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 TLS handshake.

In NetX Secure TLS, 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_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 TLS handshake will fail. The certificate provided to the TLS Session with *nx_secure_tls_local_certificate_add* must be recognized by the remote server in order to complete the TLS handshake.

Client Certificate Authentication for TLS Servers

The TLS Server case for Client Certificate Authentication is slightly more complex than the TLS 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_tls_session_client_verify_enable* and *nx_secure_tls_session_client_verify_disable* services.

To enable Client Certificate Authentication, an application must call *nx_secure_tls_session_client_verify_enable* with the TLS Server session instance before calling *nx_secure_tls_session_start*. Note that calling this service on a TLS Session that is used for TLS Client connections will have no effect.

When Client Certificate Authentication is enabled, the TLS Server will request a certificate from the remote TLS Client during the TLS handshake. In NetX Secure TLS Server, the Client certificate is checked against the store of trusted certificates created with *nx_secure_tls_trusted_certificate_add* 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 TLS handshake will fail. Additionally, if the CertificateVerify message processing fails, the TLS 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. For

TLS 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.

Cryptographic Methods

NetX Secure TLS implements DES, 3DES, AES, MD5, HMAC-MD5, SHA-1, HMAC-SHA1, SHA-256, HMAC-SHA256, and RSA in software with hardware drivers for certain hardware platforms. An application may use the cryptographic routines provided with NetX Secure, or use custom routines provided by the end user or third parties.

The *NX_CRYPTO_METHOD* is a control block designed for an application to describe a particular implementation of a cryptographic algorithm to be used with NetX Secure TLS. With the *NX_CRYPTO_METHOD,* an application can easily integrate their own crypto implementation into NetX Secure. The *NX_CRYPTO_METHOD* structure is declared as:

```
typedef struct NX_CRYPTO_METHOD_STRUCT
{
    /* Symbolic name of the algorithm. */
    USHORT nx_crypto_algorithm;
```

```
/* Size of the key, in bits. */
USHORT nx_crypto_key_size_in_bits;

/* Size of the IV block, in bits, used for encryption. */
USHORT nx_crypto_IV_size_in_bits;

/* Size of the ICV block, in bits, used for authentication. */
USHORT nx_crypto_ICV_size_in_bits;

/* Size of the crypto block, in bytes. */
ULONG nx_crypto_block_size_in_bytes;

/* Size of the metadata area. */
ULONG nx_crypto_metadata_size;

/* nx_crypto_init function initializes the crypto method with the
    "secret key" or other state information. The initialization
    routine should return a handle to the caller.  This handle is
    used in subsequent crypto operations to identify the session.
    */

UINT (*nx_crypto_init) (NX_CRYPTO_METHOD      *method,
                        UCHAR               *key,
                        NX_CRYPTO_KEY_SIZE   key_size_in_bits,
                        VOID               **handler,
                        VOID                *crypto_metadata,
                        VOID                 crypto_metadata_size);

/* NetX Secure calls the nx_crypto_cleanup routine when a TLS
   session is to be deleted (or updated).  Resources allocated
   during the crypto operation should be released in this routine.
   */
UINT (*nx_crypto_cleanup) (VOID *handler);

/* nx_crypto_operation is the actual crypto or hash operation. Note
   that both input and output buffers are prepared by the caller.
   For encryption or decryption operations, the crypto operation
   routine uses the output buffer for encrypted or decrypted data.
   For authentication operations, the authentication routine shall
   use the output buffer for the digest. */
UINT (*nx_crypto_operation)(UINT  op,
            VOID              *handler,
            NX_CRYPTO_METHOD  *method,
            UCHAR             *key,
            NX_CRYPTO_KEY_SIZE key_size_in_bits,
            UCHAR             *input,
            ULONG              input_length_in_byte,
            UCHAR             *iv_ptr,
            UCHAR             *output,
            ULONG              output_length_in_byte,
            VOID              *crypto_metadata,
            VOID               crypto_metadata_size,
            NX_PACKET*         packet_ptr,
            VOID (*nx_crypto_hw_process_callback(NX_PACKET
                                                 *packet_ptr,
                                                 UINT status);
  } NX_CRYPTO_METHOD;
```

Below is the description of each element in the *NX_CRYPTO_METHOD* structure:

- `nx_crypto_algorithm`: This field identifies the algorithm described in the variable *method.* Valid values for NetX Secure TLS are:
  - `NX_CRYPTO_NONE`
  - `NX_CRYPTO_ENCRYPTION_NULL`
  - `NX_CRYPTO_ENCRYPTION_AES_CBC`
  - `NX_CRYPTO_AUTHENTICATION_NONE`
  - `TLS_HASH_SHA_1`
  - `TLS_HASH_SHA_256`
  - `TLS_HASH_MD5`
  - `TLS_CIPHER_RSA`
  - `TLS_CIPHER_NULL`

- `nx_crypto_key_size_in_bits`: this field specifies the size of the secret key used by the method.
- `nx_crypto_IV_size_in_bits`: this field specifies the size of the Initialization Vector (IV). Note that in most cases the IV block is only used for encryption/decryption algorithms. Authentication and verification algorithms rarely use this field.
- `nx_crypto_ICV_size_in_bits`: this field specifies the size of the Integrity Check Value (ICV) block. NOTE: This block is for IPsec usage and is unused in TLS. See NetX Duo IPsec for more information.
- `nx_crypto_block_size_in_bytes`: this field specifies the size of the cryptographic algorithm block for block-based ciphers, in bytes. In most cases this is used by encryption routines and rarely by authentication routines.
- `nx_crypto_metadata_area_size`: this field specifies the size of the metadata area this method requires. Each implementation may require certain memory to store its state information, or to store intermediate data (such as key transformation material), or to use as a scratch area. The amount of space required by an implementation is specified in this field. The application provides the memory space when creating a TLS session. The cryptographic function is responsible for managing this metadata area.
- `nx_crypto_init`: This is the initialization function for the cryptographic algorithm. For an implementation that does not need an initialization routine, this field may be set to `NX_NULL`. A typical use of an initialization function is to initialize the internal data structure for the algorithm. NetX Secure TLS will handle initialization of the cryptographic routine by calling this function internally.

  The prototype for the initialization function is:

```
UINT crypto_init_function(NX_CRYPTO_METHOD *method,
                          UCHAR *key,
                          UINT  key_size_in_bits,
                          VOID  **handle,
                          VOID  *crypto_metadata_area,
                          ULONG crypto_metadata_area_size)
```

  - `method` is a pointer to the crypto method control block.
  - `key` is the secret key string for processing the data packets.
  - `key_size_in_bits` defines the size of the secret key, in bits.
  - `handle` is an implementation-defined item that identifies a particular crypto session. The value is generated by the initialization routine, and is passed back to the caller. The subsequent crypto operation or clean up routine use this `handle` to identify the session.
  - `crypto_metadata` is a pointer to the metadata area required by the implementation of this algorithm. For algorithms that do not need a metadata area this field is set to `NX_NULL` and the initialization routine must not access the metadata area.
  - `crypto_metadata_size` specifies the size of the metadata area. For SAs created without metadata area, this field is set to zero, and the initialization routine must not access the metadata area.
  - This routine shall return *NX_SUCCESS* if the initialization process is successful. The caller treats any other return value as failure.

- `nx_crypto_cleanup`: This is the cleanup routine defined for the implementation of a crypto algorithm. It is invoked when a TLS session is deleted or restarted.

  The prototype for the cleanup function is:

  ```
  UINT crypto_cleanup_function(VOID *handle)
  ```

  - `handle` is passed to the cleanup function by the caller. The `handle` is initialized by the crypto initialization routine and used to identify cryptographic algorithm state.
  - This routine shall return *NX_SUCCESS* if the cleanup process is successful. The caller treats any other return value as failure.

- `nx_crypto_operation`: This is the routine that performs the actual encryption, decryption, and authentication services. The function prototype of the operation routine is:

  ```
  UINT crypto_operation_function(UINT   op,
            VOID  *handle,
            NX_CRYPTO_METHOD* method,
            UCHAR *key,
            UCHAR  key_size_in_bits,
            UCHAR* input,
            ULONG  input_length_in_byte,
            UCHAR* iv_ptr,
            UCHAR* output,
            ULONG  output_length_in_byte,
            VOID *crypto_metadata,
            ULONG crypto_metadata_size,
            NX_PACKET *packet_ptr,
            VOID (*nx_crypto_hw_process_callback)(NX_PACKET
                            *packet_ptr, UINT status));
  ```

- o op indicates the type of operation this routine is expected to carry out. Valid values are:
  - NX_CRYPTO_ENCRYPT
  - NX_CRYPTO_DECRYPT
  - NX_CRYPTO_AUTHENTICATE
  - NX_CRYPTO_VERIFY
- o handle is passed to the operation function by the caller. It is generated by the crypto initialization routine.
- o method points to the crypto method control block
- o key points to the secret key used for this operation
- o key_size_in_bits is the size of the secret key in bits
- o input is a pointer to the beginning of the message to be operated on.
- o input_length_in_byte is passed by the caller to indicate the size of the message to be operated on.
- o iv_ptr is setup by the caller to point to the beginning of an IV block. Note that the memory for the IV block is provided by the caller. For encryption, the operation function should write the IV information into this memory block; for decryption, the operation function should retrieve the IV information from this memory block. Algorithms for authentication and verification operation typically do not use the initialization vector.
- o output is setup by the caller to point to an output buffer. Note that the memory for the output buffer is provided by the caller. For encryption, the operation function should write the cipher text to the output buffer; for decryption, the operation should write the deciphered text (clear text) to the output buffer; for authentication, the hash value shall be written to the output buffer. For verification, the output buffer is used to store hash information.
- o output_length_in_byte indicates the size of the output buffer
- o crypto_metadata points to the metadata area to be used by this crypto operation. The crypto metadata area is typically initialized by crypto_init_function.
- o crypto_metadata_size indicates the size of the metadata area.
- o This routine shall return *NX_SUCCESS* if the operation process is successful. The caller treats any other return value as failure.
- o packet_ptr: The packet that contains the data being processed. NOTE: This parameter is unused by TLS and should be set to NX_NULL.
- o nx_crypto_hw_process_callback: A callback function provided by the encryption method. This is used if the crypto function is provided by hardware and requires a callback routine.

NetX Secure TLS provides the following encryption methods:

- *AES*
- *RSA*
- *NULL*

NetX Secure TLS provides the following authentication methods:

- *HMAC-MD5*
- *HMAC-SHA1*
- *HMAC-SHA256*

The following examples illustrate how to configure the *NX_CRYPTO_METHOD* structure to use the encryption and authentication methods provided by NetX Duo IPsec.

AES:

```
/* AES-CBC 128. */
NX_CRYPTO_METHOD crypto_method_aes_cbc_128 =
{
    /* AES crypto algorithm                              */
    NX_CRYPTO_ENCRYPTION_AES_CBC,

    /* Key size in bits. For AES-128 this value is 128  */
    NX_CRYPTO_AES_128_KEY_LEN_IN_BITS,

    /* IV size in bits.  For AES-128 this value is 128  */
    NX_CRYPTO_AES_IV_LEN_IN_BITS,

    /* ICV size in bits, not used.                       */
    0,

    /* Block size in bytes.  For AES this value is 16   */
    (NX_CRYPTO_AES_BLOCK_SIZE_IN_BITS >> 3),

    /* Metadata size in bytes, for AES this value is 262*/
    sizeof(NX_CRYPTO_AES),

    /* AES-CBC initialization routine.                   */
    _nx_secure_crypto_method_aes_init,

    /* AES-CBC cleanup routine, not used.                */
    NX_NULL,

    /* AES-CBC operation                                 */
    _nx_secure_crypto_method_aes_operation
};

/* RSA. */
NX_CRYPTO_METHOD crypto_method_rsa =
{
    /* RSA crypto algorithm                              */
    TLS_CIPHER_RSA,

    /* Key size. RSA key sizes vary, so set to 0.       */
    0,

    /* IV size in bits.  RSA does not use an IV.        */
    0,

    /* ICV size in bits, not used.                       */
    0,
    /* Block size in bytes.  RSA does not have a block size. */
    0,

    /* Metadata size in bytes, for RSA use the control block. */
    sizeof(NX_CRYPTO_RSA),

    /* RSA initialization routine.                  */
    _nx_secure_crypto_method_rsa_init,

    /* Cleanup routine, not used.                   */
    NX_NULL,

    /* RSA operation                                */
    _nx_secure_crypto_method_rsa_operation

};
```

NULL

```
/* NULL encryption method. */
NX_CRYPTO_METHOD crypto_method_null =
{
    NX_CRYPTO_ENCRYPTION_NULL,/* Name of the crypto algorithm  */
    0,                        /* Key size in bits, not used    */
    0,                        /* IV size in bits, not used     */
    0,                        /* ICV size in bits, not used    */
    4,                        /* Block size in bytes           */
    0,                        /* Metadata size in bytes        */
    NX_NULL,                  /* Initialization routine,unused */
    NX_NULL,                  /* Cleanup routine, not used     */
    _nx_secure_crypto_method_null_operation  /* NULL operation
*/
};
```

HMAC-SHA1

```
NX_CRYPTO_METHOD crypto_method_hmac_sha1 =
{
    /* HMAC SHA1 algorithm                               */
    TLS_HASH_SHA1,


    /* Key size in bits. For HMAC-SHA1 this value is 160 */
    NX_CRYPTO_HMAC_SHA1_KEY_LEN_IN_BITS,

    /* IV size in bits, not used                         */
    0,

    /* Transmitted ICV size in bits. Unused.              */
    0,
    /* Block size in bytes, not used                         */
    0,

    /* Metadata size in bytes                                */
    sizeof(NX_SHA1_HMAC),

    /* Initialization routine, not used                      */
    NX_NULL,

    /* Cleanup routine, not used                             */
    NX_NULL,

    /* HMAC SHA1 operation                                   */
    _nx_secure_crypto_method_hmac_sha1_operation
};
```

NONE

A special method **NX_CRYPTO_NONE** is used to signal the IPsec module that the encryption or the authentication service is not required. It is configured as follows:

```
/* NX_CRYPTO_NONE means encryption or authentication
   method is not needed.  */
NX_CRYPTO_METHOD crypto_method_none =
{
    NX_CRYPTO_NONE,       /* Name of the crypto algorithm */
    0,                    /* Key size in bits, not used   */
    0,                    /* IV size in bits, not used    */
    0,                    /* ICV size in bits, not used   */
    0,                    /* Block size in bytes          */
    0,                    /* Metadata size in bytes       */
    NX_NULL,              /* Initialization routine, not used */
    NX_NULL,              /* Cleanup routine, not used    */
    NX_NULL               /* NULL operation               */
};
```

Initializing TLS with Cryptographic Methods

Once you have created your cryptographic routines conforming to the cryptographic method signatures described in the previous section, you will need to pass them into TLS when you initialize an NX_SECURE_TLS_SESSION control block. This is done in the TLS service nx_secure_tls_session_create:

```
UINT  nx_secure_tls_session_create(
               NX_SECURE_TLS_SESSION*        session_ptr,
               const NX_SECURE_TLS_CRYPTO*   tls_cipher_table,
               VOID*                         encryption_metadata_area,
               ULONG                         encryption_metadata_size
);
```

- `session_pointer` is a pointer to your NX_SECURE_TLS_SESSION control block.

- `tls_cipher_table` is a pointer to an NX_SECURE_TLS_CRYPTO control block, described below.
- `encryption_metadata_area` points to space used by cryptographic routines in TLS.
- `encryption_metadata_size` is the size of the metadata area in bytes.

TLS Cryptographic Cipher Table

The NX_SECURE_TLS_CRYPTO structure is defined as:

```
typedef struct NX_SECURE_METHODS_STRUCT
{
    /* Table that maps ciphersuites to crypto methods. */
    NX_SECURE_TLS_CIPHERSUITE_INFO* nx_secure_tls_ciphersuite_lookup_table;
    USHORT nx_secure_tls_ciphersuite_lookup_table_size;

    /* Table that maps X.509 cipher identifiers to crypto methods. */
    NX_SECURE_X509_CRYPTO *nx_secure_tls_x509_cipher_table;
    USHORT nx_secure_tls_x509_cipher_table_size;

    /* Specific routines needed for specific TLS versions. */
#if (NX_SECURE_TLS_TLS_1_0_ENABLED || NX_SECURE_TLS_TLS_1_1_ENABLED)
    NX_CRYPTO_METHOD *nx_secure_tls_handshake_hash_md5_method;
    NX_CRYPTO_METHOD *nx_secure_tls_handshake_hash_sha1_method;
    NX_CRYPTO_METHOD *nx_secure_tls_prf_1_method;
#endif

#if (NX_SECURE_TLS_TLS_1_2_ENABLED)
    NX_CRYPTO_METHOD *nx_secure_tls_handshake_hash_sha256_method;
    NX_CRYPTO_METHOD *nx_secure_tls_prf_sha256_method;
#endif

} NX_SECURE_TLS_CRYPTO;
```

The table is created by filling in the entries for this structure in a static constant located within the NetX Secure TLS project, usually located with the cryptographic routines and modules.

As an example, the software-only (“generic”) cryptographic library provided with NetX Secure contains the following table definition:

```
/* Define the cipher table object we can pass into TLS. */
const NX_SECURE_TLS_CRYPTO nx_crypto_tls_ciphers =
{
    /* TLS Ciphersuite lookup table and size. */
    _nx_crypto_ciphersuite_lookup_table,
    sizeof(_nx_crypto_ciphersuite_lookup_table) /
    sizeof(NX_SECURE_TLS_CIPHERSUITE_INFO),

    /* X.509 certificate cipher table and size. */
    _nx_crypto_x509_cipher_lookup_table,
    sizeof(_nx_crypto_x509_cipher_lookup_table) / sizeof(NX_SECURE_X509_CRYPTO),

    /* TLS version-specific methods. */
#if (NX_SECURE_TLS_TLS_1_0_ENABLED || NX_SECURE_TLS_TLS_1_1_ENABLED)
    &crypto_method_md5,
    &crypto_method_sha1,
    &crypto_method_tls_prf_1,
#endif
#if (NX_SECURE_TLS_TLS_1_2_ENABLED)
    &crypto_method_sha256,
    &crypto_method_tls_prf_sha_256
#endif

};
```

In the structure, the first entry is the TLS ciphersuite table. The NX_SECURE_TLS_CIPHERSUITE_INFO structure maps cryptographic routines (in the form of NX_CRYPTO_METHOD pointers) to specific ciphersuites as defined in the TLS specifications. The second value is the number of entries in the table pointed to by the first field.

The next field points to a table of routines used by X.509 when processing digital certificates and the structure NX_SECURE_X509_CRYPTO is similar in form to NX_SECURE_TLS_CIPHERSUITE_INFO. The following field is the number of entries in the table.

Following the lookup table are a number of routines needed for specific versions of TLS. For example, prior to TLS version 1.2, the key generation and handshake hashing routines were fixed to use a combination of SHA-1 and MD5 – the methods for these routines are called out specifically in the cipher structure since they are not tied to specific ciphersuites. In TLS version 1.2, the key generation and hashing routines are chosen by the ciphersuite, but for ciphersuites which do not specify the routines to use, the SHA-256 hash method is used, and the cipher structure calls out that routine specifically.

TLS Ciphersuite Lookup Table

To fill in the cipher table for TLS, you will also need to create a ciphersuite lookup table that maps cryptographic routines to specific ciphersuite identifiers. The identifiers are IANA-registered values that are universal. See the TLS RFCs for more information. The routines represent the 5 separate methods used in each ciphersuite (some ciphersuites may not use all 5): public cipher, public-key authentication, session cipher, session hash routine, and TLS Pseudo-Random Function (PRF). The following table explains each of the 5 methods:

Routine category	**Description**	**Example algorithms**
Public cipher	Used to exchange keys during the TLS handshake	RSA, Diffie-Hellman, ECC
Public-key authentication	Used to authenticate or sign data during the TLS handshake	RSA, DSS
Session cipher	Symmetric-key algorithm used to encrypt application data during the TLS session	AES, RC4
Session hash	Used to preserve the integrity of messages during the TLS session (assures that data has not changed)	SHA-1, SHA-256

TLS PRF	Used to generate key material and in the handshake hash in the TLS handshake	The PRF is based on hash routines – SHA-1 + MD5, SHA-256, SHA-512

The NX_SECURE_TLS_CIPHERSUITE_INFO structure is defined as follows:

```
typedef struct NX_SECURE_TLS_CIPHERSUITE_INFO_struct
{
    /* The IANA value of the ciphersuite as defined by the TLS spec.*/
    USHORT nx_secure_tls_ciphersuite;

    /* The Public Key operation in this suite - RSA or DH. */
    NX_CRYPTO_METHOD *nx_secure_tls_public_cipher;

    /* The Public Authentication method used for signing data. */
    NX_CRYPTO_METHOD *nx_secure_tls_public_auth;

    /* The session cipher being used - AES, RC4, etc. */
    NX_CRYPTO_METHOD *nx_secure_tls_session_cipher;

    /* The size of the initialization vectors for the session cipher (bytes).*/
    USHORT nx_secure_tls_iv_size;

    /* The key size for the session cipher (bytes). */
    UCHAR nx_secure_tls_session_key_size;

    /* The hash being used - MD5, SHA-1, SHA-256, etc. */
    NX_CRYPTO_METHOD *nx_secure_tls_hash;

    /* The size of the hash being used. SHA-1 is 20 bytes, MD5 is 16 bytes.*/
    USHORT nx_secure_tls_hash_size;

    /* The TLS PRF being used – this is only for TLSv1.2. */
    NX_CRYPTO_METHOD *nx_secure_tls_prf;

} NX_SECURE_TLS_CIPHERSUITE_INFO;
```

The nx_secure_tls_ciphersuite field contains the IANA ciphersuite value, and the NX_CRYPTO_METHOD pointers represent the 5 methods used by that ciphersuite. The scalar values (nx_secure_tls_iv_size, nx_secure_tls_key_size, and nx_secure_tls_hash_size) are informational, providing information that might not be available in the NX_CRYPTO_METHOD entries.

As an example, we will look at the default ciphersuite for TLS, TLS_RSA_WITH_AES_128_CBC_SHA, which specifies the use of RSA, AES-CBC with 128-bit keys, and SHA-1 for session hashing. No TLS PRF is specified for this ciphersuite, so in TLSv1.2 mode, it will use the default SHA-256 PRF. Note that all ciphersuites use the SHA-1+MD5 PRF in TLS 1.0 and 1.1, regardless of the PRF specified in the table.

The entry in the NX_SECURE_TLS_CIPHERSUITE_INFO table in the generic cryptographic library is defined as follows:

```
{
  TLS_RSA_WITH_AES_128_CBC_SHA,     /* Ciphersuite identifier */
  &crypto_method_rsa,               /* Public-key cipher (NX_CRYPTO_METHOD)*/
  &crypto_method_rsa,               /* Authentication method(NX_CRYPTO_METHOD)*/
  &crypto_method_aes_cbc_128,       /* Session cipher method(NX_CRYPTO_METHOD)*/
  16,                               /* Session cipher IV size in bytes */
  16,                               /* Session cipher key size in bytes */
  &crypto_method_hmac_sha1,         /* Session hash routine(NX_CRYPTO_METHOD) */
  20,                               /* Session hash output size in bytes */
  &crypto_method_tls_prf_sha_256    /* TLSv1.2 PRF */
},
```

Note that for the session cipher the key size is determined by the ciphersuite, but for the public-key methods the key size is not known until the TLS handshake is underway since the public keys are contained in the digital certificates exchanged during the handshake.

X.509 Cipher Lookup Table

Like the NX_SECURE_TLS_CIPHERSUITE_INFO table, the NX_SECURE_X509_CRYPTO structure maps cryptographic routines to known values. In the case of X.509, the identifiers are actually OIDs defined by X.509 and registered with the ISO and ITU standards bodies. OIDs are variable-length multi-byte values designed to uniquely identify various information in various telecommunication standards, including cryptographic routines used in digital certificates. Due to the fact that OIDs are variable length, NetX Secure TLS maps the official OID values to fixed-length constants that are used internally (see nx_secure_x509.h). These constants are used in the NX_SECURE_X509_CRYPTO structure, which is defined as follows:

```
/* Structure to hold X.509 cryptographic routine information. */
typedef struct NX_SECURE_X509_CRYPTO_struct
{
    /* Internal NetX Secure identifier for certificate "ciphersuite" which consists of
       a hash and a public key operation. These can be mapped to OIDs in X.509. */
    USHORT nx_secure_x509_crypto_identifier;

    /* Public-Key Cryptographic method used by certificates. */
    NX_CRYPTO_METHOD *nx_secure_x509_public_cipher_method;

    /* Hash method used by certificates. */
    NX_CRYPTO_METHOD *nx_secure_x509_hash_method;
} NX_SECURE_X509_CRYPTO;
```

The first field, *nx_secure_x509_crypto_identifier*, is the internal OID representation used by NetX Secure.

The second and third fields point to NX_CRYPTO_METHOD objects that represent the cryptographic methods identified by the OID, a public-key operation paired with a hash routine. Note that each digital certificate may have more than one OID for cryptographic routines.

The method table for X.509 is constructed in the same manner as the ciphersuite lookup table. As an example, we will look at the OID for RSA_SHA1. The actual OID for RSA_SHA1 is as follows:

```
{iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-1(1) sha1-with-rsa-
signature(5)}
```

The OID is represented in ASN.1 syntax and has a numeric value of 1.2.840.113549.1.1.5. This value is then encoded in binary format, creating the following bytes:

```
UCHAR RSA_SHA1_OID = { 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x01, 0x01, 0x05 };
```

The actual conversion from ASN.1 to the binary format is beyond the scope of this document. Search for ASN.1 encodings for OIDs for more information. The binary representation of the OIDs supported by NetX Secure can be found in the file *nx_secure_x509.c*.

Once we have a mapping of the actual OID to an internally-recognized constant, we can create an entry for RSA_SHA1 in the NX_SECURE_X509_CRYPTO table:

```
{
       NX_SECURE_TLS_X509_TYPE_RSA_SHA_1,    /* Internal OID constant. */
       &crypto_method_rsa,                   /* RSA method (NX_CRYPTO_METHOD). */
       &crypto_method_sha1                   /* SHA-1 method (NX_CRYPTO_METHOD). */
},
```

Default TLS Routines

As mentioned above, TLS requires some default routines for key generation and message verification during the handshake. The primary routine is the TLS Pseudo-Random Function, or PRF. The PRF is based on hash routines and can be used to generate an arbitrary amount of pseudo-random data[6] for key generation or other purposes.

In addition to the PRF, each version of TLS utilizes default hash routines that also need to be provided. For TLS versions 1.0 and 1.1, those hash routines are MD5 and SHA-1. TLS version 1.2 requires only SHA-256.

In the NX_SECURE_TLS_CRYPTO structure, there are NX_CRYPTO_METHOD pointers for MD5, SHA-1, SHA-256, the TLS version 1.0/1.1 PRF, and the default TLS 1.2 PRF.

Provided in the generic software cryptography library are software versions of the TLS PRF. For TLS 1.0/1.1, this function is called *nx_crypto_tls_prf_1*. For TLS 1.2, the function is called *nx_secure_tls_prf_sha256*. The suffix “1” represents the legacy TLS 1.0 PRF, and the “sha256” suffix refers to the fact that the TLS 1.2 default PRF is based on SHA-256. When support for other PRF routines is needed, the suffix for those routines will reflect the hash method used. Since the PRF routines are based on hash methods, the underlying hash routines may be hardware-accelerated independently on different target platforms.

In addition to the TLS ciphersuite and X.509 lookup tables, with the default PRF and hash routines filled in the NX_SECURE_TLS_CRYPTO structure can be populated and used to initialize a TLS session.

[6] “Pseudo-random” refers to the fact that the PRF is deterministic, meaning it will always produce the same output given the same input, but random in the fact that the output is not predictable. TLS uses this property of the PRF to generate the session keys from various public data combined with the master secret exchanged during the handshake using a public-key cipher like RSA.

Cryptographic Metadata

Before we can initialize the TLS session with the NX_SECURE_TLS_CRYPTO table, we need to allocate buffer space for the cryptographic routine metadata. The metadata is used to store all the state associated with a particular routine, represented by its control block. The *nx_crypto_metadata_area_size* field of each NX_CRYPTO_METHOD must be set to the size of the control structure associated with that routine or the TLS initialization will fail to properly account for the space needed, possibly causing buffer overrun issues.

Before the TLS session is created, the metadata buffer must be allocated. The buffer is automatically divided up by nx_secure_tls_session_create and space is reserved for each of the routines that are provided in the cryptographic method table. Note that since only one ciphersuite is active at a time in a TLS session, the number of supported ciphersuites does not affect the needed metadata space – space is reserved for each of the 5 ciphersuite routines using the maximum control block size for that category in the ciphersuite lookup table.

In order to make calculating the metadata buffer size easy, the service *nx_secure_metadata_size_calculate* performs the same calculations as nx_secure_tls_session_create but simply returns the total required metadata buffer size in bytes.

Initializing the TLS session

Once the NX_CRYPTO_METHOD and NX_SECURE_TLS_CRYPTO objects are created and the metadata area reserved, we can initialize a TLS session as follows (values taken from the above examples):

```
/* Pointer to the platform-specific cipher table. */
extern nx_crypto_tls_ciphers;

/* Cryptographic routine metadata buffer. Size is determined by calling
nx_secure_tls_metadata_size_calculate with the nx_crypto_tls_ciphers table referenced
above. */
UCHAR crypto_metadata[4500];

/* Initialize our TLS session using our cipher table and metadata area. Note that we can
use sizeof for the metadata array because the size parameter expects the size in bytes.*/

nx_secure_tls_session_create( &tls_session,            /* Pointer to TLS session.      */
                              &nx_crypto_tls_ciphers,  /* Pointer to cipher table.     */
                              crypto_metadata,         /* Cryptography metadata buffer.*/
                              sizeof(crypto_metadata), /* Size of metadata buffer.     */
                      );
```

Chapter 4

Description of NetX Secure Services

This chapter contains a description of all NetX Secure 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_crypto_table_self_test

Perform self-test on the crypto methods

Prototype

```
UINT nx_secure_crypto_table_self_test(
                        const NX_SECURE_TLS_CRYPTO *crypto_table,
                        VOID *metadata, UINT metadata_size);
```

Description

This service runs through the crypto method self tests to validate. The self test is available only if NetX Secure library is built with the symbol NX_SECURE_POWER_ON_SELF_TEST_MODULE_INTEGRITY_CHECK defined.

For each supported crypto method, the self test provides pre-defined input data, and verified the output matches the pre-defined expected value.

NetX Secure crypto self test supports the following algorithms and key sizes:

- DES: encryption and decryption
- Triple DES (3DES): encryption and decryption
- AES: 128-, 192-, 256-bit key size, encryption and decryption, in CBC mode and counter mode.
- HMAC-MD5: authentication and hash computation
- HMAC-SHA: SHA1-96, SHA1-160, SHA2-256, SHA2-384, SHA2-512, authentication and hash computation
- MD5: authentication
- Pseudo-random Function (PRF): PRF_HMAC_SHA1 and PRF_HMAC_SHA2-256
- RSA: 1024-, 2048-, 4096-bit RSA power-modulus operation
- SHA: SHA1 (96- and 160-bit), SHA2 (256bit, 384bit, 512bit) authentication

This function has the built-in vectors for the crypto algorithms listed above. However it only tests the ones listed in the *cipher_table* passed into this function. For example, for a TLS session uses only the ciphersuite TLS_RSA_WITH_AES_128_CBC_SHA, this function will perform self test on the RSA(1024-, 2048-, 4096-bit), AES-CBC (128-bit) and SHA1.

Parameters

crypto_table Pointer to the crypto table being used by the TLS session. This is the same crypto_table being passed into the ***nx_secure_tls_session_create()*** call.

metadata Pointer to space for cryptography metadata area. .

metadata_size Size of the metadata buffer.

Return Values

NX_SECURE_TLS_SUCCESS	(0x00)	Successfully tested the crypto methods provided.
NX_PTR_ERROR	(0x07)	Invalid crypto method structure
NX_NOT_SUCCESSFUL	(0x43)	Crypto self test fails.

Allowed From

Initialization**,** Threads

Example

```
/* crypto_tls_ciphers is the cipher table for the TLS session. */
/* metadata_buffer is the memory area used by the cipher functions. */

/* The following function verifies the supplied ciphersuties using the built-in
   self test. */


if (nx_secure_crypto_table_self_test(&crypto_tls_ciphers, metadata_buffer,
    sizeof(metadata_buffer)))
{
    printf("Crypto self test failed!\r\n");
    exit(0);
}
else
{
     printf("Crypto self test succeed!\r\n");
}

/* Now the ciphersuites are successfully test, it can be used in
   nx_secure_tls_session_create() */
```

See Also

nx_secure_tls_session_create

nx_secure_module_hash_compute

Computes hash value using user-supplied hash function

Prototype

```
UINT nx_secure_module_hash_compute(
                      NX_CRYPTO_METHOD *hamc_ptr,
                      UINT start_address, UINT end_address,
                      UCHAR *key, UINT key_length,
                      VOID *metadata, UINT metadata_size,
                      UCHAR *output_buffer, UINT output_buffer_size,
                      UINT *actual_size);
```

Description

This function computes the hash value of the data stream in the specified memory area, using supplied HMAC crypto method and the key string. The module hash compute function is available only if NetX Secure library is built with the following symbol being defined: `NX_SECURE_POWER_ON_SELF_TEST_MODULE_INTEGRITY_CHECK`

Parameters

hmac_ptr Pointer to the HMAC crypto method being used for the hash value computation.

start_address The starting address of the data buffer

end_address The ending address of the data buffer. Note that hash computation does not cover the data at this end_address.

key The key string being used in the HMAC computation.

key_length The size of the key string, in bytes

metadata Pointer to space used by the HMAC algorithm.

metadata_size Size of the metadata buffer.

output_buffer The memory location where the hash output is being stored at.

output_buffer_size The available space of the output buffer, in bytes

actual_size Returned by the function indicating the actual number of bytes being written into the output_buffer.

Return Values

0 Successfully computed the hash value.

1 The hash computation failed.

Allowed From

Initialization, Threads

Example

```
/* Define the HMAC SHA256 method */
NX_CRYPTO_METHOD hmac_sha256 =
{
    NX_CRYPTO_AUTHENTICATION_HMAC_SHA2_256,    /* HMAC SHA256 algorithm  */
    0,                                         /* Key size, not used     */
    0,                                         /* IV size, not used      */
    NX_CRYPTO_HMAC_SHA256_ICV_FULL_LEN_IN_BITS,/* Transmitted ICV size   */
    NX_CRYPTO_SHA2_BLOCK_SIZE_IN_BYTES,        /* Block size in bytes    */
    sizeof(NX_CRYPTO_SHA256_HMAC),             /* Metadata size in bytes */
    _nx_crypto_method_hmac_sha256_init,        /* HMAC SHA256 init       */
    _nx_crypto_method_hmac_sha256_cleanup,     /* HMAC SHA256 cleanup    */
    _nx_crypto_method_hmac_sha256_operation    /* HMAC SHA256 operation  */
};

/* Define the hash key being used. */
const CHAR hash_key = “my hash key”;

/* Define starting address. */
UINT starting_address = 0x10000;

/* Define the ending address.
   Note that the hash computation covers the memory location
   before the ending address. */
UINT ending_address = 0x11000;

/* Declare the output buffer. */
#define OUTPUT_BUFFER_SIZE
UCHAR output_buffer[OUTPUT_BUFFER_SIZE];

UINT actual_size;

/* Declare 1K bytes of metadata buffer area. */
UCHAR metadata_buffer[1024];

/* Compute the hash value of the data between 0x10000 – 0x10FFF */
nx_secure_module_hash_compute(&hmac_sha256,
                              starting_address, ending_address,
                              hash_key, strlen(hash_key),
                              metadata_buffer, sizeof(metadata_buffer),
                              output_buffer, OUTPUT_BUFFER_SIZE, &actual_size);

/* If this function returns “0”, the hash value has been computed and is stored
   in output_buffer. */
```

See Also

nx_secure_crypto_method_self_test

nx_secure_tls_active_certificate_set

Set the active identity certificate for a NetX Secure TLS Session

Prototype

```
UINT  nx_secure_tls_active_certificate_set(
                NX_SECURE_TLS_SESSION *tls_session,
                NX_SECURE_X509_CERT *certificate);
```

Description

This service is intended to be called from within a session callback (see nx_secure_tls_session_client_callback_set and nx_secure_tls_session_server_callback_set). When called with a previously-initialized NX_SECURE_X509_CERT structure, that certificate will be used instead of the default identity certificate. In most cases, the certificate must have been added to the local store (see nx_secure_tls_local_certificate_add) or the TLS handshake may fail.

This service is intended to allow TLS to support multiple identity certificates. This is useful for a TLS server that services multiple network addresses so the server can pick an appropriate certificate to provide to the remote client depending on the client’s entrypoint. For a TLS client, this routine may be used to change the certificate sent to a remote server at runtime after the server has identified itself in the TLS handshake (this is more rare than the TLS server use-case).

In the case where multiple certificates may share the same X.509 distinguished name, certificates will need to be added using nx_secure_tls_server_certificate_add, which introduces a numeric identifier separate from the certificate.

Parameters

session_ptr	Pointer to a TLS Session instance passed into the session callback.
certificate	Pointer to an initialized X.509 certificate that is to be used for the current session.

Return Values

NX_SUCCESS	(0x00)	Successful assignment of certificate to session.
NX_PTR_ERROR	(0x07)	Invalid TLS session or certificate pointer.

Allowed From

Threads

Example

```
#define TLS_SNI_SERVER_NAME "www.example.com"
#define TLS_DEFAULT_SERVER_CERT_ID 2
#define TLS_EXAMPLE_CERT_ID 3


/* Callback for ClientHello extensions processing. */
static ULONG tls_server_callback(NX_SECURE_TLS_SESSION *tls_session,
                                 NX_SECURE_TLS_HELLO_EXTENSION *extensions,
                                 UINT num_extensions)
{
NX_SECURE_X509_DNS_NAME dns_name;
INT compare_value;
UINT status;
NX_SECURE_X509_CERT *cert_ptr;

    /* Grab a pointer to our default certificate in case the SNI extension is not
       available or the specified server name is unknown. */
    nx_secure_tls_server_certificate_find(tls_session, &cert_ptr,
                                          TLS_DEFAULT_SERVER_CERT_ID);


    /* Process Server Name Indication extension. */
    status = _nx_secure_tls_session_sni_extension_parse(tls_session, extensions,
                                                        num_extensions, &dns_name);

    /* If no extension found, that is OK. Use default certificate. */
    if(status == NX_SUCCESS)
    {
        /* SNI extension found, select an active certificate based on the server
           name. */

        /* Make sure our SNI name matches. */
        compare_value = memcmp(dns_name.nx_secure_x509_dns_name,
                               TLS_SNI_SERVER_NAME, strlen(TLS_SNI_SERVER_NAME));

        if(compare_value == 0 && dns_name.nx_secure_x509_dns_name_length !=
           strlen(TLS_SNI_SERVER_NAME))
        {
            /* Found a match, find the certificate we want to use. */
            _nx_secure_tls_server_certificate_find(tls_session, &cert_ptr,
                                                   TLS_EXAMPLE_CERT_ID);
        }
        else
        {
            /* No matching server name found. The application may choose to simply
               provide the default certificate (and probably resulting in an error
               in the remote client) or returning an error here to end the
               handshake. A user-defined non-zero error code will be sufficient –
               the error code will abort the TLS handshake and be returned to the
               caller that started the server. */
               return(0x500);
        }
    }
    else
    {
    }

    /* Set the certificate we want to use. */
    nx_secure_tls_active_certificate_set(tls_session, cert_ptr);

    /* Return success to continue TLS handshake. */
    return(NX_SUCCESS);
}

/* Sockets, sessions, certificates defined in global static space to preserve
   application stack. */
NX_SECURE_TLS_SESSION server_tls_session;

/* Application where TLS session is started. */
void main()
{
    /* 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_tls_session_create(&server_tls_session,
                                           &nx_crypto_tls_ciphers,
                                           server_crypto_metadata,
                                           sizeof(server_crypto_metadata));

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

    /* Setup our packet reassembly buffer. See
        nx_secure_tls_session_packet_buffer_set for more information. */
    nx_secure_tls_session_packet_buffer_set(&server_tls_session,
                                      server_packet_buffer,
                                      sizeof(server_packet_buffer));


    /* Set callback for server TLS extension handling. */
    _nx_secure_tls_session_server_callback_set(&server_tls_session,
                                               tls_server_callback);

    /* Initialize our certificates – certificate data defined elsewhere. See
       section “Importing X.509 Certificates into NetX Secure” for more
       information. */
    nx_secure_x509_certificate_initialize(&default_certificate,
                                      default_cert_der,
                                      default _cert_der_len, NX_NULL, 0,
                                      default_cert_key_der,
                                      default_cert_key_der_len,
                                      NX_SECURE_X509_KEY_TYPE_RSA_PKCS1_DER);

    /* Add the certificate to the local store using a numeric ID. */
    nx_secure_tls_server_certificate_add(&server_tls_session,
                                         &default_certificate,
                                         TLS_DEFAULT_SERVER_CERT_ID);

    /* Alternative identity certificate, needs to have a private key! */
    nx_secure_x509_certificate_initialize(&example_certificate,
                                      example_cert_der,
                                      example cert_der_len, NX_NULL, 0,
                                      example_cert_key_der,
                                      example_cert_key_der_len,
                                      NX_SECURE_X509_KEY_TYPE_RSA_PKCS1_DER);

    /* Add the certificate to the local store using a numeric ID. */
    nx_secure_tls_server_certificate_add(&server_tls_session,
                                         &example_certificate,
                                         TLS_EXAMPLE_CERT_ID);

    /* Now we can start the TLS session as normal. */
        …
}
```

See Also

nx_secure_x509_certificate_initialize,
nx_secure_tls_local_certificate_add,
nx_secure_tls_session_client_callback_set,
nx_secure_tls_session_server_callback_set,
nx_secure_tls_server_certificate_add,
nx_secure_tls_server_certificate_find,
nx_secure_tls_server_certificate_remove

nx_secure_tls_client_psk_set

Set a Pre-Shared Key for a NetX Secure TLS Client Session

Prototype

```
UINT  nx_secure_tls_client_psk_set(NX_SECURE_TLS_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 TLS Session control block, and sets that PSK to be used in subsequent TLS Client connections. The PSK is used in place of a digital certificate when PSK ciphersuites are enabled and used.

In this case, the PSK is associated with a specific remote TLS Server with which the TLS Client wishes to communicate. The PSK set through this API will be provided to the remote TLS Server host during the next TLS handshake.

Parameters

session_ptr	Pointer to a previously created TLS 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 TLS 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 TLS session.  */
status =  nx_secure_tls_client_psk_set(&tls_session, psk, sizeof(psk), “psk_1”, 4,
“Client_identity”, 15);


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

See Also

nx_secure_tls_psk_add, nx_secure_x509_certificate_initialize, nx_secure_tls_session_create, nx_secure_tls_remote_certificate_allocate, nx_secure_tls_local_certificate_add

nx_secure_tls_initialize

Initializes the NetX Secure TLS module

Prototype

```
VOID nx_secure_tls_initialize(VOID);
```

Description

This service initializes the NetX Secure TLS module. It must be called before other NetX Secure services can be accessed.

Parameters

None

Return Values

None

Allowed From

Initialization**,** Threads

Example

```
/* Initializes the TLS module. */
Nx_secure_tls_initialize();
```

See Also

nx_secure_tls_session_create

nx_secure_tls_local_certificate_add

Add a local certificate to NetX Secure TLS Session

Prototype

```
UINT  nx_secure_tls_local_certificate_add(
                    NX_SECURE_TLS_SESSION *session_ptr,
                    NX_SECURE_X509_CERT *certificate_ptr);
```

Description

This service adds an initialized NX_SECURE_X509_CERT structure instance to the local store of a TLS session. This certificate may used by the TLS stack to identify the device during the TLS handshake (if it was marked as an identity certificate during initialization of the certificate structure using nx_secure_x509_certificate_initialize), or as an issuer as part of a certificate chain provided to the remote host during the TLS handshake.

If multiple local certificates with the same Common Name are needed, certificates may be added using the *nx_secure_tls_server_certificate_add* service (see warning below).

A certificate is **required** for TLS Server mode.

A certificate is *optional* for TLS Client mode.

This API should not be used with the same TLS session when using nx_secure_tls_server_certificate_add. The server certificate API uses a unique numeric identifier for each certificate, and nx_secure_tls_local_certificate_add indexes based on the X.509 Common Name. The local certificate services provide a convenient alternative to the numeric identifier for applications that use only a single identity certificate – by using the Common Name, the application need not keep track of numeric identifiers.

Parameters

session_ptr Pointer to a previously created TLS Session instance.
certificate_ptr Pointer to an initialized TLS Certificate instance.

Return Values

NX_SUCCESS	(0x00)	Successful addition of certificate.
NX_INVALID_PARAMETERS	(0x4D)	Tried to add an invalid or duplicate certificate.
NX_PTR_ERROR	(0x07)	Invalid TLS session or certificate pointer.

Allowed From

Threads

Example

```
/* Initialize certificate structure. */
status =  nx_secure_x509_certificate_initialize(&certificate, certificate_data,
500, private_key, 64);

/* Add certificate to TLS session.  */
status =  nx_secure_tls_local_certificate_add(&tls_session, &certificate);


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

See Also

nx_secure_x509_certificate_initialize, nx_secure_tls_session_create,
nx_secure_tls_remote_certificate_allocate,
nx_secure_tls_local_certificate_remove,
nx_secure_tls_local_certificate_find

nx_secure_tls_local_certificate_find

Find a local certificate in a NetX Secure TLS Session by Common Name

Prototype

```
UINT  nx_secure_tls_local_certificate_find(NX_SECURE_TLS_SESSION
                        *session_ptr, NX_SECURE_X509_CERT
                        **certificate, UCHAR *common_name, UINT
                        name_length);
```

Description

This service finds a certificate in the local device certificate store of a TLS session and returns a pointer to the NX_SECURE_X509_CERT structure in the store. The common_name parameter and it's length (name_length) are used to identify the certificate in the store by matching the certificate's X.509 Subject Common Name field.

If more than one certificate with the same Common Name is present, only the first one will be returned – use *nx_secure_tls_server_certificate_find* instead.

This API should not be used with the same TLS session when using nx_secure_tls_server_certificate_add. The server certificate API uses a unique numeric identifier for each certificate, and nx_secure_tls_local_certificate_add indexes based on the X.509 Common Name. The local certificate services provide a convenient alternative to the numeric identifier for applications that use only a single identity certificate – by using the Common Name, the application need not keep track of numeric identifiers.

Parameters

session_ptr Pointer to a previously created TLS Session instance.
certificate Return Pointer to matched certificate.
common_name Common Name string to match (DNS name).
name_length Length of common_name string data.

Return Values

NX_SUCCESS	(0x00)	Certificate was found and pointer returned in “certificate” parameter.
NX_SECURE_TLS_CERTIFICATE_NOT_FOUND	(0x119)	No certificate with the supplied Common Name was found.
NX_PTR_ERROR	(0x07)	Invalid TLS session, certificate pointer, or common name string.

Allowed From

Threads

Example

```
NX_SECURE_X509_CERT *certificate_ptr;

/* Initialize certificate structure. */
status =  nx_secure_x509_certificate_initialize(&certificate, certificate_data,
500, private_key, 64);

/* Add certificate to TLS session.  */
status =  nx_secure_tls_local_certificate_add(&tls_session, &certificate);


/* If status is NX_SUCCESS the certificate was successfully added.  */

status = nx_secure_tls_local_certificate_find(&tls_session, &certificate_ptr,
                                    “common name”, strlen(“common name”));

/* If status is NX_SUCCESS the variable “certificate_ptr” points to a certificate
   structure containing a certificate with the Common Name “common name”. */
```

See Also

nx_secure_x509_certificate_initialize, nx_secure_tls_session_create, nx_secure_tls_remote_certificate_allocate, nx_secure_tls_local_certificate_remove, nx_secure_tls_local_certificate_add

nx_secure_tls_local_certificate_remove

Remove local certificate from NetX Secure TLS Session

Prototype

```
UINT  nx_secure_tls_local_certificate_remove(NX_SECURE_TLS_SESSION
                 *session_ptr, UCHAR *common_name, UINT
                 common_name_length);
```

Description

This service removes a local certificate instance from a TLS session, keyed on the Common Name field in the certificate.

This API should not be used with the same TLS session when using nx_secure_tls_server_certificate_add. The server certificate API uses a unique numeric identifier for each certificate, and nx_secure_tls_local_certificate_add indexes based on the X.509 Common Name. The local certificate services provide a convenient alternative to the numeric identifier for applications that use only a single identity certificate – by using the Common Name, the application need not keep track of numeric identifiers.

Parameters

session_ptr	Pointer to a previously created TLS Session instance.
common_name	The Common Name value of the certificate to be removed.
common_name_length	The length of the Common Name string.

Return Values

NX_SUCCESS	(0x00)	Successful addition of certificate.
NX_PTR_ERROR	(0x07)	Invalid TLS session pointer.
NX_SECURE_TLS_CERTIFICATE_NOT_FOUND	(0x119)	Certificate was not found.

Allowed From

Threads Example

```
/* Add certificate to TLS session.  */
status =  nx_secure_tls_local_certificate_remove(&tls_session,
                                            “www.example.com”, 15);


/* If status is NX_SUCCESS the certificate was successfully removed.  */
```

See Also

nx_secure_x509_certificate_initialize, nx_secure_tls_session_create, nx_secure_tls_remote_certificate_allocate, nx_secure_tls_local_certificate_add

nx_secure_tls_metadata_size_calculate

Calculate size of cryptographic metadata for a NetX Secure TLS Session

Prototype

```
UINT  nx_secure_tls_metadata_size_calculate(
                          const NX_SECURE_TLS_CRYPTO *crypto_table,
                          ULONG *metadata_size);
```

Description

This service calculates and returns the size of the cryptographic metadata needed for a particular TLS session and TLS cryptography table (see the section “Initializing TLS with Cryptographic Methods” for more information on the cryptographic cipher table).

This service should be called with the desired cryptographic table to calculate the size of the metadata buffer passed into nx_secure_tls_session_create.

Parameters

crypto_table	Pointer to a complete NetX Secure TLS cryptography table.
metadata_size	The output of the size calculation in bytes.

Return Values

NX_SUCCESS	(0x00)	Successful calculation of metadata size.
NX_PTR_ERROR	(0x07)	Invalid crypto table or return size pointer.

Allowed From

Threads

Example

```
/* Pointer to the platform-specific cipher table. */
extern nx_crypto_tls_ciphers;

/* Return variable for metadata size. */
ULONG crypto_metadata_size;

/* Calculate metadata size.  */
status =  nx_secure_tls_metadata_size_calculate(&nx_crypto_tls_ciphers,
                                                &crypto_metadata_size);

/* If status is NX_SUCCESS then crypto_metadata_size contains the size of the
   metadata buffer for the table nx_crypto_tls_ciphers.  */
```

See Also

nx_secure_tls_session_create

nx_secure_module_hash_compute

Compute the hash value of the NetX Secure library routines

Prototype

```
VOID nx_secure_module_hash_compute(VOID);
```

Description

This service initializes the NetX Secure TLS module. It must be called before other NetX Secure services can be accessed.

Parameters

None

Return Values

None

Allowed From

Initialization, Threads

Example

```
/* Initializes the TLS module. */
Nx_secure_tls_initialize();
```

See Also

nx_secure_tls_session_create

nx_secure_tls_packet_allocate

Allocate a packet for a NetX Secure TLS Session

Prototype

```
UINT  nx_secure_tls_packet_allocate(NX_SECURE_TLS_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 TLS session from the specified NX_PACKET_POOL. This service should be called by the application to allocate data packets to be sent over a TLS connection. The TLS session must be initialized before calling this service.

The allocated packet is properly initialized so that TLS 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 TLS 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 TLS session was not initialized.

Allowed From

Threads

Example

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

status = nx_secure_tls_packet_allocate(&tls_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_tcp_socket_receive, nx_secure_x509_certificate_initialize, nx_secure_tls_remote_certificate_allocate, nx_secure_tls_session_start, nx_secure_tls_session_delete, nx_secure_tls_session_receive, nx_secure_tls_session_send, nx_secure_tls_session_end, nx_secure_tls_session_create

nx_secure_tls_psk_add

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

Prototype

```
UINT  nx_secure_tls_psk_add(NX_SECURE_TLS_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 TLS 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 TLS 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 TLS 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 TLS session.  */
status =  nx_secure_tls_psk_add(&tls_session, psk, sizeof(psk), “psk_1”, 4,
“Client_identity”, 15);


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

See Also

nx_secure_tls_client_psk_set, nx_secure_x509_certificate_initialize, nx_secure_tls_session_create, nx_secure_tls_remote_certificate_allocate, nx_secure_tls_local_certificate_add

nx_secure_tls_remote_certificate_allocate

Allocate space for the certificate provided by a remote TLS host

Prototype

```
UINT  nx_secure_tls_remote_certificate_allocate(
                NX_SECURE_TLS_SESSION *session_ptr,
                NX_SECURE_X509_CERT *certificate_ptr,
                UCHAR *raw_certificate_buffer,
                UINT raw_buffer_size);
```

Description

This service adds an uninitialized NX_SECURE_X509_CERT structure instance to a TLS session for the purpose of allocating space for certificates provided by a remote host during a TLS session. The remote certificate data is parsed by NetX Secure TLS and that information is used to populate the certificate structure instance provided to this function. Certificates added in this manner are placed in a linked list.

If it is expected that the remote host will provide multiple certificates, this function should be called repeatedly to allocate space for all certificates. The additional certificates are added to the end of the certificate linked list.

Failure to allocate a remote certificate will cause TLS Client mode to fail during the TLS handshake unless a Pre-Shared Key (PSK) ciphersuite is in use.

The *raw_certificate_buffer* parameter points to space allocated to store the incoming remote certificate. Typical certificates with RSA keys of 2048 bits using SHA-256 for signatures are in the range of 1000-2000 bytes. The buffer should be large enough to at least hold that size, but depending on the remote host certificates may be significantly smaller or larger. Note that if the buffer is too small to hold the incoming certificate, the TLS handshake will end with an error.

For TLS Server mode, a remote certificate allocation is needed only if client certificate authentication is enabled.

Parameters

session_ptr	Pointer to a previously created TLS Session instance.
certificate_ptr	Pointer to an uninitialized X.509 Certificate instance.
raw_certificate_buffer	Pointer to a buffer to hold the un-parsed certificate received from the remote host.
raw_buffer_size	Size of the raw certificate buffer.

Return Values

NX_SUCCESS	(0x00)	Successful allocation of certificate.
NX_PTR_ERROR	(0x07)	Invalid TLS session pointer.
NX_SECURE_TLS_INSUFFICIENT_CERT_SPACE	(0x12D)	The supplied buffer was too small.
NX_INVALID_PARAMETERS	(0x4D)	Tried to add an invalid certificate.

Allowed From

Threads

Example

```
/* Certificate control block. */
NX_SECURE_X509_CERT certificate;

/* Buffer space to hold the incoming certificate. */
UCHAR certificate_buffer[2000];

/* Add certificate to TLS session.  */
status =  nx_secure_tls_remote_certificate_allocate(&tls_session, &certificate,
                                                    certificate_buffer,
                                                    sizeof(certificate_buffer));


/* If status is NX_SUCCESS the certificate was successfully allocated.  */
```

See Also

nx_secure_x509_certificate_initialize, nx_secure_tls_session_create

nx_secure_tls_remote_certificate_buffer_allocate

Allocate space for all certificates provided by a remote TLS host

Prototype

```
UINT  nx_secure_tls_remote_certificate_buffer_allocate(
                 NX_SECURE_TLS_SESSION *session_ptr,
                 UINT certs_number, VOID *certificate_buffer,
                 ULONG buffer_size);
```

Description

This service allocates space to process incoming certificate chains from remote server hosts in order to perform X.509 authentication and verification in a TLS Client instance. For TLS Server mode, remote certificate allocation is needed only if client X.509 certificate authentication is enabled – for TLS Server instances the service *nx_secure_tls_session_x509_client_verify_configure* should be used instead.

Failure to allocate remote certificates will cause TLS Client mode to fail during the TLS handshake unless a Pre-Shared Key (PSK) ciphersuite is in use.

The *certificate_buffer* parameter points to space allocated to store the incoming remote certificates and the control blocks needed for those certificates. The buffer will be divided by the number of certificates (*certs_number*) with an equal portion given to each certificate. The *buffer_size* parameter indicates the size of the buffer. The space needed can be found with the following formula:

```
buffer_size = (<expected max number of certificates in chain>) *
              (sizeof(NX_SECURE_X509_CERT) + <max cert size>)
```

Typical certificates with RSA keys of 2048 bits using SHA-256 for signatures are in the range of 1000-2000 bytes. The buffer should be large enough to at least hold that size for each certificate, but depending on the remote host certificates may be significantly smaller or larger. Note that if the buffer is too small to hold the incoming certificate, the TLS handshake will end with an error.

Parameters

session_ptr	Pointer to a previously created TLS Session instance.
certs_number	Number of certificates to allocate from the provided buffer.
certificate_buffer	Pointer to a buffer to hold certificates received from a remote host.
buffer_size	Size of the certificate buffer.

Return Values

NX_SUCCESS	(0x00)	Successful allocation of certificate.
NX_PTR_ERROR	(0x07)	Invalid TLS session or buffer pointer.
NX_SECURE_TLS_INSUFFICIENT_CERT_SPACE	(0x12D)	The supplied buffer was too small.
NX_INVALID_PARAMETERS	(0x4D)	The buffer was too small to hold the desired number of certificates.

Allowed From

Threads

Example

```
/* Buffer space to hold the incoming certificates. */
#define CERTIFICATE_NUMBER    (2)
#define CERTIFICATE_SIZE      (2000)
#define BUFFER_SIZE           (CERTIFICATE_NUMBER * \
                              (sizeof(NX_SECURE_X509_CERT) + CERTIFICATE_SIZE))
UCHAR certificate_buffer[BUFFER_SIZE];

/* Add certificate to TLS session.  */
status =  nx_secure_tls_remote_certificate_buffer_allocate(&tls_session,
                                                     CERTIFICATE_NUMBER,
                                                     certificate_buffer,
                                                     BUFFER_SIZE);


/* If status is NX_SUCCESS the buffer was successfully allocated.  */
```

See Also

nx_secure_x509_certificate_initialize, nx_secure_tls_session_create

nx_secure_tls_remote_certificate_free_all

Free space allocated for incoming certificates

Prototype

```
UINT  nx_secure_tls_remote_certificate_free_all(
                  NX_SECURE_TLS_SESSION *session_ptr);
```

Description

This service is used to free all of the certificate buffers allocated to a particular TLS Session by nx_secure_tls_remote_certificate_allocated by returning them to that session's free certificate space. This may be necessary if an application reuses a TLS session object without deleting it and recreating it with nx_secure_tls_session_delete and nx_secure_tls_session_create.

Note that the remote certificate space is recovered automatically when the TLS session is reset as happens in nx_secure_tls_session_end so most applications should not need to call this service.

Parameters

session_ptr	Pointer to a previously created TLS Session instance.

Return Values

NX_SUCCESS	(0x00)	Successful operation.
NX_PTR_ERROR	(0x07)	Invalid TLS session pointer.
NX_INVALID_PARAMETERS	(0x4D)	Internal error – certificate store likely corrupt.

Allowed From

Threads

Example

```
/* Certificate control block. */
NX_SECURE_X509_CERT certificate;

/* Buffer space to hold the incoming certificate. */
UCHAR certificate_buffer[2000];

/* Add certificate to TLS session.  */
status =  nx_secure_tls_remote_certificate_allocate(&tls_session, &certificate,
                                                    certificate_buffer,
                                                    sizeof(certificate_buffer));


/* If status is NX_SUCCESS the certificate was successfully allocated.  */

/* … TLS session setup, TCP connection, etc… */

/* End TLS session. */
nx_secure_tls_session_end(&tls_session, NX_WAIT_FOREVER);

/* Free up certificate buffers for next connection. */
nx_secure_tls_remote_certificate_free_all(&tls_session);
```

See Also

nx_secure_x509_certificate_initialize, nx_secure_tls_session_create,
nx_secure_tls_remote_certificate_allocate, nx_secure_tls_session_end

nx_secure_tls_server_certificate_add

Add a certificate specifically for TLS servers using a numeric identifier

Prototype

```
UINT  nx_secure_tls_server_certificate_add(
                  NX_SECURE_TLS_SESSION *session_ptr,
                  NX_SECURE_X509_CERT *certificate, UINT cert_id);
```

Description

This service is used to add a certificate to a TLS session's local store (see nx_secure_tls_local_certificate_add) using a numeric identifier instead of indexing the store using the X.509 Subject (Common Name) within the certificate. The numeric identifier is separate from the certificate and allows multiple certificates to be added as identity certificates to a TLS server, as well as allowing multiple certificates with the same Common Name to be added to the same TLS session local store. This same service can be used for client certificates, but it is rare for a TLS client to have multiple identity certificates.

The cert_id parameter is a non-zero positive integer assigned by the application. The actual value does not matter (other than zero) but it must be unique in the store for the provided TLS session. The cert_id value can be used to find and remove certificates from the local store using nx_secure_tls_server_certificate_find and nx_secure_tls_server_certificate_remove, respectively.

This API should not be used with the same TLS session when using nx_secure_tls_local_certificate_add. The nx_secure_tls_server_certificate_add API uses a unique numeric identifier for each certificate, and local certificate services index based on the X.509 Common Name. The server certificate services allow multiple certificates with shared data (especially Common Name) to exist in the same local store – this is useful for a server with multiple identities.

Parameters

session_ptr Pointer to a previously created TLS Session instance.

certificate Pointer to a previously initialized X.509 certificate instance.

cert_id Positive, non-zero, relatively unique certificate ID number.

Return Values

NX_SUCCESS	(0x00)	Successful operation.
NX_PTR_ERROR	(0x07)	Invalid TLS session or certificate pointer.
NX_SECURE_TLS_CERT_ID_INVALID	(0x138)	The provided certificate ID had An invalid value (likely 0).
NX_SECURE_TLS_CERT_ID_DUPLICATE	(0x139)	The provided certificate ID was already present in the local store.
NX_INVALID_PARAMETERS	(0x4D)	Internal error – certificate store likely corrupt.

Allowed From

Threads

Example

```
/* Certificate control block. */
NX_SECURE_X509_CERT certificate;


/* Add certificate to TLS session.  */
status =  nx_secure_tls_server_certificate_add(&tls_session, &certificate, 0x12);


/* If status is NX_SUCCESS the certificate was successfully added with the ID
0x12.  */

**See also the example for nx_secure_tls_active_certificate_set.
```

See Also

nx_secure_x509_certificate_initialize,
nx_secure_tls_local_certificate_add, nx_secure_tls_active_certificate_set,
nx_secure_tls_server_certificate_find,
nx_secure_tls_server_certificate_remove

nx_secure_tls_server_certificate_find

Find a certificate using a numeric identifier

Prototype

```
UINT  nx_secure_tls_server_certificate_find(
                NX_SECURE_TLS_SESSION *session_ptr,
                NX_SECURE_X509_CERT **certificate, UINT cert_id);
```

Description

This service is used to find a certificate in a TLS session’s local store (see nx_secure_tls_local_certificate_add) using a numeric identifier instead of indexing the store using the X.509 Subject (Common Name) within the certificate.

The cert_id parameter is a non-zero positive integer assigned by the application when the certificate is added to the TLS session local store using nx_secure_tls_server_certificate_add.

This API should not be used with the same TLS session when using nx_secure_tls_local_certificate_add. The nx_secure_tls_server_certificate_add API uses a unique numeric identifier for each certificate, and local certificate services index based on the X.509 Common Name. The server certificate services allow multiple certificates with shared data (especially Common Name) to exist in the same local store – this is useful for a server with multiple identities.

Parameters

session_ptr	Pointer to a previously created TLS Session instance.
certificate	Pointer to an X.509 certificate pointer to return a reference to the found certificate.
cert_id	Non-zero positive certificate ID value.

Return Values

NX_SUCCESS	(0x00)	Successful operation.
NX_PTR_ERROR	(0x07)	Invalid TLS session or certificate pointer.
NX_SECURE_TLS_CERTIFICATE_NOT_FOUND	(0x119)	The provided certificate ID did not match any in the local store of the provided TLS session.

Allowed From

Threads

Example

```
NX_SECURE_X509_CERT *certificate;


/* Find certificate in TLS session.  */
status =  nx_secure_tls_server_certificate_find(&tls_session, &certificate, 0x12);


/* If status is NX_SUCCESS the certificate was successfully found and a reference
returned in the certificate parameter.  */

**See also the example for nx_secure_tls_active_certificate_set.
```

See Also

nx_secure_x509_certificate_initialize,
nx_secure_tls_local_certificate_add, nx_secure_tls_active_certificate_set,
nx_secure_tls_server_certificate_add,
nx_secure_tls_server_certificate_remove

nx_secure_tls_server_certificate_remove

Remove a local server certificate using a numeric identifier

Prototype

```
UINT  nx_secure_tls_server_certificate_remove(
                NX_SECURE_TLS_SESSION *session_ptr, UINT cert_id);
```

Description

This service is used to remove a certificate from a TLS session's local store (see nx_secure_tls_local_certificate_add) using a numeric identifier instead of indexing the store using the X.509 Subject (Common Name) within the certificate.

The cert_id parameter is a non-zero positive integer assigned by the application when the certificate is added to the TLS session local store using nx_secure_tls_server_certificate_add.

Parameters

session_ptr	Pointer to a previously created TLS Session instance.
cert_id	Non-zero positive certificate ID value.

Return Values

NX_SUCCESS	(0x00)	Successful operation.
NX_PTR_ERROR	(0x07)	Invalid TLS session.
NX_SECURE_TLS_CERTIFICATE_NOT_FOUND	(0x119)	The provided certificate ID did not match any in the local store of the provided TLS session.

Allowed From

Threads

Example

```
/* Certificate control block. */
NX_SECURE_X509_CERT certificate;


/* Add certificate to TLS session with id 0x12.  */
status =  nx_secure_tls_server_certificate_add(&tls_session, &certificate, 0x12);
/* Use certificate in TLS session, etc… */

/* Remove certificate from TLS session with id 0x12.  */
status =  nx_secure_tls_server_certificate_remove(&tls_session, 0x12);


/* If status is NX_SUCCESS the certificate was successfully removed.  */
```

See Also

nx_secure_x509_certificate_initialize,
nx_secure_tls_local_certificate_add, nx_secure_tls_active_certificate_set,
nx_secure_tls_server_certificate_add,
nx_secure_tls_server_certificate_find

nx_secure_tls_session_certificate_callback_set

Set up a callback for TLS to use for additional certificate validation

Prototype

```
UINT  nx_secure_tls_ session_certificate_callback_set (
                NX_SECURE_TLS_SESSION *tls_session,
                ULONG (*func_ptr)(NX_SECURE_TLS_SESSION *session,
                            NX_SECURE_X509_CERT *certificate));
```

Description

This service assigns a function pointer to a TLS session that TLS will invoke when a certificate is received from a remote host, allowing the application to perform validation checks such as DNS validation, certificate revocation, and certificate policy enforcement.

NetX Secure TLS will perform basic X.509 path validation on the certificate before invoking the callback to assure that the certificate can be traced to a certificate in the TLS trusted certificate store, but all other validation will be handled by this callback.

The callback provides the TLS session pointer and a pointer to the remote host identity certificate (the leaf in the certificate chain). The callback should return NX_SUCCESS if all validation is successful, otherwise it should return an error code indicating the validation failure. Any value other than NX_SUCCESS will cause the TLS handshake to immediately abort.

Parameters

session_ptr	Pointer to a previously created TLS Session instance.
func_ptr	Pointer to the certificate validation callback function.

Return Values

NX_SUCCESS	(0x00)	Successful allocation of Function pointer.
NX_PTR_ERROR	(0x07)	Invalid TLS session pointer.

Allowed From

Threads

Example

```
/* Callback routine used to perform additional checks on a certificate. */
ULONG certificate_callback(NX_SECURE_TLS_SESSION *session, NX_SECURE_X509_CERT
*certificate)
{
    /* Certificate validation checking goes here. */
    return(NX_SUCCESS);
}

/* In TLS setup routine. */
{
       /* Add callback to TLS session.  */
       status =  nx_secure_tls_session_certificate_callback_set(&tls_session,
                                                            certificate_callback);

       /* If status is NX_SUCCESS the certificate callback was added.  */
}
```

See Also

nx_secure_tls_session_create,
nx_secure_x509_common_name_dns_check,
nx_secure_x509_crl_revocation_check

nx_secure_tls_session_client_callback_set

Set up a callback for TLS to invoke at the beginning of a TLS Client handshake

Prototype

```
UINT  nx_secure_tls_ session_client_callback_set (
                  NX_SECURE_TLS_SESSION *tls_session,
                  ULONG (*func_ptr)(NX_SECURE_TLS_SESSION *tls_session,
                                    NX_SECURE_TLS_HELLO_EXTENSION
                                    *extensions, UINT num_extensions));
```

Description

This service assigns a function pointer to a TLS session that TLS will invoke when a TLS Client handshake has received a ServerHelloDone message. The callback function allows an application to process any TLS extensions from the received ServerHello message that require input or decision making.

The callback is executed with the invoking TLS session control block and an array of NX_SECURE_TLS_HELLO_EXTENSION objects. The array of extension objects is intended to be passed into a helper function that will find and parse a specific extension. Currently, there are no specific extensions supported in NetX Secure that require TLS Client input, but the callback is available for application designers to handle custom or new extensions that may become available. For an example helper function that parses TLS extensions provided in hello messages, see *nx_secure_tls_session_sni_extension_parse*.

The client callback may also be used to select the active identity certificate using *nx_secure_tls_active_certificate_set* for the TLS Client in the event that the remote server has requested a certificate and provided information to allow the TLS Client to select a specific certificate. See the reference for nx_secure_tls_active_certificate_set for more information.

Parameters

session_ptr	Pointer to a previously created TLS Session instance.
func_ptr	Pointer to the TLS Client callback function.

Return Values

NX_SUCCESS	(0x00)	Successful allocation of function pointer.
NX_PTR_ERROR	(0x07)	Invalid TLS session pointer.

Allowed From

Threads

Example

```
/* Callback routine used to process ServerHello extensions and perform other
   runtime actions at the beginning of a TLS handshake (such as selecting an
   identify certificate to provide to the server). */

ULONG tls_client_callback(NX_SECURE_TLS_SESSION *session,
                          NX_SECURE_TLS_HELLO_EXTENSION *extensions, UINT
                          num_extensions)
{

    /* Extension parsing would go here. */

    /* Set an active identity certificate – the certificate should have been added
       to the local store at application start with
       nx_secure_tls_local_certificate_add. */
    nx_secure_tls_active_certificate_set(session, &global_identity_certificate);

    return(NX_SUCCESS);
}

/* TLS setup routine. */
UINT tls_setup(NX_SECURE_TLS_SESSION *tls_session)
{
       /* Add callback to TLS session.  */
       status =  nx_secure_tls_session_client_callback_set(tls_session,
                                                             client_callback);

       /* If status is NX_SUCCESS the callback was added and will be invoked once
          a ServerHelloDone message is received. */

       return(status);
}
```

See Also

nx_secure_tls_session_create,
nx_secure_tls_session_server_callback_set,
nx_secure_tls_active_certificate_set

nx_secure_tls_session_x509_client_verify_configure

Enable client X.509 verification and allocate space for client certificates

Prototype

```
UINT  nx_secure_tls_session_x509_client_verify_configure(
                  NX_SECURE_TLS_SESSION *session_ptr,
                  UINT certs_number, VOID *certificate_buffer,
                  ULONG buffer_size);
```

Description

This service enables the optional X.509 Client Authentication for a TLS Server instance. It also allocates the space needed to process incoming certificate chains from the remote client host. The certificates supplied by the remote client will be verified against the TLS server instance’s trusted certificates, assigned with the service *nx_secure_tls_trusted_certificate_add.*

For TLS Client mode, remote certificate allocation is required and the service *nx_secure_tls_remote_certificate_buffer_allocate* should be used instead. Enabling X.509 Client Authentication on a TLS Client instance will have no effect.

The *certificate_buffer* parameter points to space allocated to store the incoming remote certificates and the control blocks needed for those certificates. The buffer will be divided by the number of certificates (*certs_number*) with an equal portion given to each certificate. The *buffer_size* parameter indicates the size of the buffer. The space needed can be found with the following formula:

```
buffer_size = (<expected max number of certificates in chain>) *
              (sizeof(NX_SECURE_X509_CERT) + <max cert size>)
```

Typical certificates with RSA keys of 2048 bits using SHA-256 for signatures are in the range of 1000-2000 bytes. The buffer should be large enough to at least hold that size for each certificate, but depending on the remote host certificates may be significantly smaller or larger. Note that if the buffer is too small to hold the incoming certificate, the TLS handshake will end with an error.

Parameters

session_ptr	Pointer to a previously created TLS Session instance.
certs_number	Number of certificates to allocate from the provided buffer.
certificate_buffer	Pointer to a buffer to hold certificates received from a remote host.
buffer_size	Size of the certificate buffer.

Return Values

NX_SUCCESS	(0x00)	Successful allocation of certificate.
NX_PTR_ERROR	(0x07)	Invalid TLS session or buffer pointer.
NX_SECURE_TLS_INSUFFICIENT_CERT_SPACE	(0x12D)	The supplied buffer was too small.
NX_INVALID_PARAMETERS	(0x4D)	The buffer was too small to hold the desired number of certificates.

Allowed From

Threads

Example

```
/* Buffer space to hold the incoming certificates. */
#define CERTIFICATE_NUMBER    (2)
#define CERTIFICATE_SIZE      (2000)
#define BUFFER_SIZE           (CERTIFICATE_NUMBER * \
                              (sizeof(NX_SECURE_X509_CERT) + CERTIFICATE_SIZE))
UCHAR certificate_buffer[BUFFER_SIZE];

/* Enable X.509 Client verification and allocate certificate space in our TLS
   Server session.  */
status =  nx_secure_tls_session_x509_client_verify_configure(&tls_session,
                                                    CERTIFICATE_NUMBER,
                                                    certificate_buffer,
                                                    BUFFER_SIZE);


/* If status is NX_SUCCESS the buffer was successfully allocated.  */
```

See Also

nx_secure_x509_certificate_initialize, nx_secure_tls_session_create, nx_secure_tls_remote_certificate_buffer_allocate

nx_secure_tls_session_client_verify_disable

Disable Client Certificate Authentication for a NetX Secure TLS Session

Prototype

```
UINT  nx_secure_tls_session_client_verify_disable(
                              NX_SECURE_TLS_SESSION *session_ptr);
```

Description

This service disables Client Certificate Authentication for a specific TLS session. See nx_secure_tls_session_client_verify_enable for more information.

Parameters

session_ptr	Pointer to a TLS Session instance.

Return Values

NX_SUCCESS	(0x00)	Successful state change.
NX_PTR_ERROR	(0x07)	Invalid TLS session pointer.

Allowed From

Threads

Example

```
/* Disable client certificate authentication for this TLS session.   */
status = nx_secure_tls_session_client_verify_disable(&tls_session);

/* Any new TLS Server sessions using the tls_session control block will not
request certificates from the remote TLS client.  */
```

See Also

nx_secure_tls_session_client_verify_enable, nx_secure_tls_session_start, nx_secure_tls_session_create

nx_secure_tls_session_client_verify_enable

Enable Client Certificate Authentication for a NetX Secure TLS Session

Prototype

```
UINT  nx_secure_tls_session_client_verify_enable(
                              NX_SECURE_TLS_SESSION *session_ptr);
```

Description

This service enables Client Certificate Authentication for a specific TLS session. Enabling Client Certificate Authentication for a TLS Server instance will cause the TLS Server to request a certificate from any remote TLS Client during the initial TLS handshake. The certificate received from the remote TLS Client is accompanied by a CertificateVerify message, which the TLS Server uses to verify that the Client owns the certificate (has access to the private key associated with that certificate).

If the provided certificate can be verified and traced back to a certificate in the TLS Server trusted certificate store via an X.509 certificate chain, the remote TLS Client is authenticated and the handshake proceeds. In case of any errors in processing the certificate or CertificateVerify message, the TLS handshake ends with an error.

Note that the TLS Server must have at least one certificate in its trusted store added with *nx_secure_tls_trusted_certificate_add* or authentication will always fail.

Parameters

session_ptr	Pointer to a TLS Session instance.

Return Values

NX_SUCCESS	(0x00)	Successful state change.
NX_PTR_ERROR	(0x07)	Invalid TLS session pointer.

Allowed From

Threads

Example

```
/* Add a trusted certificate so the TLS Server has something to verify against. */
status = nx_secure_tls_trusted_certificate_add(&tls_session,
                                               &trusted_certificate);

/* Disable client certificate authentication for this TLS session.   */
status = nx_secure_tls_session_client_verify_enable(&tls_session);

/* Any new TLS Server sessions using the tls_session control block will now
request and verify certificates from the remote TLS client.  */
```

See Also

nx_secure_tls_session_client_verify_enable, nx_secure_tls_session_start, nx_secure_tls_session_create, nx_secure_tls_trusted_certificate_add

nx_secure_tls_session_create

Create a NetX Secure TLS Session for secure communications

Prototype

```
UINT  nx_secure_tls_session_create(NX_SECURE_TLS_SESSION *session_ptr
                                   NX_SECURE_TLS_CRYPTO *cipher_table,
                                   VOID *encryption_metadata_area,
                                   ULONG encryption_metadata_size);
```

Description

This service initializes an NX_SECURE_TLS_SESSION structure instance for use in establishing secure TLS communications over a network connection.

The method takes a NX_SECURE_TLS_CRYPTO object that is populated with the available cryptographic methods to be used for TLS. The *encryption_metadata_area* points to a buffer allocated for use by TLS for the “metadata” used by the cryptographic methods in the NX_SECURE_TLS_CRYPTO table for calculations. The size of the table can be determined by using the nx_secure_tls_metadata_size_calculate service. See the section “Cryptography in NetX Secure TLS” in Chapter 3 for more details.

Parameters

session_ptr	Pointer to a TLS Session instance.
cipher_table	Pointer to TLS cryptographic methods.
encryption_metadata_area	Pointer to space for cryptography metadata.
encryption_metadata_size	Size of the metadata buffer.

Return Values

NX_SUCCESS	(0x00)	Successful initialization of the TLS session.
NX_PTR_ERROR	(0x07)	Tried to use an invalid pointer.
NX_INVALID_PARAMETERS	(0x4D)	The metadata buffer was too small for the given methods.
NX_SECURE_TLS_UNSUPPORTED_CIPHER	(0x106)	A required cipher method for the enabled version of TLS was not supplied in the cipher table.

Allowed From

Threads

Example

```
/* Reference the platform-specific TLS cryptographic method table. */
extern const NX_SECURE_TLS_CRYPTO nx_crypto_tls_ciphers;

/* Declare a buffer for the cryptographic metadata. The size should be calculated
   using nx_secure_tls_metadata_size_calculate. */
UCHAR crypto_metadata[4500];

/* Initialize TLS session.  */
status =  nx_secure_tls_session_create(&tls_session,
                                        &nx_crypto_tls_ciphers,
                                        crypto_metadata,
                                        sizeof(crypto_metadata));


/* If status is NX_SUCCESS the TLS Session was successfully initialized.  */

**See also the section "Cryptography in NetX Secure TLS" in Chapter 3.
```

See Also

nx_secure_x509_certificate_initialize, nx_secure_tls_metadata_size_calculate, nx_secure_tls_remote_certificate_allocate, nx_secure_tls_session_start, nx_secure_tls_session_end, nx_secure_tls_session_send, nx_secure_tls_session_receive, nx_secure_tls_session_delete

nx_secure_tls_session_delete

Delete a NetX Secure TLS Session

Prototype

```
UINT  nx_secure_tls_session_delete(NX_SECURE_TLS_SESSION *session_ptr);
```

Description

This service deletes a TLS session represented by an NX_SECURE_TLS_SESSION structure instance and releases all system resources owned by that session instance.

Parameters

session_ptr	Pointer to a TLS Session instance.

Return Values

NX_SUCCESS	(0x00)	Successful initialization of the TLS session.
NX_PTR_ERROR	(0x07)	Tried to use an invalid pointer.

Allowed From

Threads

Example

```
/* Delete TLS session.  */
status =  nx_secure_tls_session_delete(&tls_session);


/* If status is NX_SUCCESS the TLS Session was successfully deleted.  */
```

See Also

nx_secure_x509_certificate_initialize, nx_secure_tls_remote_certificate_allocate, nx_secure_tls_session_start, nx_secure_tls_session_end, nx_secure_tls_session_send, nx_secure_tls_session_receive, nx_secure_tls_session_create

nx_secure_tls_session_end

End an active NetX Secure TLS Session

Prototype

```
UINT  nx_secure_tls_session_end(NX_SECURE_TLS_SESSION *session_ptr,
                                    ULONG wait_option);
```

Description

This service ends a TLS session represented by an NX_SECURE_TLS_SESSION structure instance by sending the TLS CloseNotify message to the remote host. The service then waits for the remote host to respond with its own CloseNotify message.

If the remote host does not send a CloseNotify message, TLS considers this an error and a possible security breach, so checking the return value is important for a secure connection. The **wait_option** parameter can be used to control how long the service should wait for the response before returning control to the calling thread.

Parameters

session_ptr	Pointer to a TLS Session instance.
wait_option	Indicates how long the service should wait for the response from the remote host.

Return Values

NX_SUCCESS	(0x00)	Successful initialization of the TLS session.
NX_SECURE_TLS_NO_CLOSE_RESPONSE	(0x113)	Did not receive a response from the remote host before timing out.
NX_SECURE_TLS_ALLOCATE_PACKET_FAILED	(0x111)	Could not allocate a packet to send the CloseNotify message.
NX_SECURE_TLS_TCP_SEND_FAILED	(0x109)	Could not send the CloseNotify message over the TCP socket.
NX_PTR_ERROR	(0x07)	Tried to use an invalid pointer.

Allowed From

Threads

Example

```
/* End TLS session.  */
status =  nx_secure_tls_session_end(&tls_session, NX_WAIT_FOREVER);


/* If status is NX_SUCCESS the TLS Session was successfully ended.  */
```

See Also

nx_secure_x509_certificate_initialize, nx_secure_tls_remote_certificate_allocate, nx_secure_tls_session_start, nx_secure_tls_session_delete, nx_secure_tls_session_send, nx_secure_tls_session_receive, nx_secure_tls_session_create

nx_secure_tls_session_packet_buffer_set

Set the packet reassembly buffer for a NetX Secure TLS Session

Prototype

```
UINT  nx_secure_tls_session_packet_buffer_set(
                                NX_SECURE_TLS_SESSION *session_ptr,
                                UCHAR *buffer_ptr,
                                ULONG buffer_size);
```

Description

This service associates a packet reassembly buffer to a TLS session. In order to decrypt and parse incoming TLS records, the data in each record must be assembled from the underlying TCP packets. TLS records can be up to 16KB in size (though typically are much smaller) so may not fit into a single TCP packet.

If you know the incoming message size will be smaller than the TLS record limit of 16KB, the buffer size can be made smaller, but in order to handle unknown incoming data the buffer should be made as large as possible. If an incoming record is larger than the supplied buffer, the TLS session will end with an error.

Parameters

session_ptr	Pointer to a TLS Session instance.
buffer_ptr	Pointer to buffer for TLS to use for packet reassembly.
buffer_size	Size of the supplied buffer in bytes.

Return Values

NX_SUCCESS	(0x00)	Successful initialization of the TLS session.
NX_INVALID_PARAMETERS	(0x4D)	Invalid buffer or TLS session pointer.
NX_PTR_ERROR	(0x07)	Tried to use an invalid pointer.

Allowed From

Threads

Example

```
/* Buffer for TLS packet reassembly. */
UCHAR tls_packet_buffer[16384];
/* Assign buffer to TLS session.  */
status =  nx_secure_tls_session_packet_buffer_set(&tls_session, tls_packet_buffer,
                                                  sizeof(tls_packet_buffer));


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

See Also

nx_secure_x509_certificate_initialize,
nx_secure_tls_remote_certificate_allocate, nx_secure_tls_session_start,
nx_secure_tls_session_delete, nx_secure_tls_session_send,
nx_secure_tls_session_receive, nx_secure_tls_session_create

nx_secure_tls_session_protocol_version_override

Override the default TLS protocol version for a NetX Secure TLS Session

Prototype

```
UINT  nx_secure_tls_session_protocol_version_override(
                              NX_SECURE_TLS_SESSION *session_ptr,
                              USHORT protocol_version);
```

Description

This service overrides the default (newest) TLS protocol version used by a particular session. This enables NetX Secure TLS to use an older version of TLS for a specific TLS Session without disabling newer TLS versions at compile time. This may be useful in applications that may need to communicate with an older host that does not support the newest version of TLS.

Valid values for the protocol_version parameter are the following macros:
NX_SECURE_TLS_VERSION_TLS_1_0,
NX_SECURE_TLS_VERSION_TLS_1_1, and
NX_SECURE_TLS_VERSION_TLS_1_2.

The macros NX_SECURE_TLS_DISABLE_TLS_1_1 and NX_SECURE_TLS_ENABLE_TLS_1_0 can be used to control the versions of TLS that are compiled into the application. TLS version 1.2 is always enabled.

Note that if the remote host does not support the supplied version, then a TLS Server instance will still attempt a version downgrade during the handshake if a lower version is available. For example, if TLS version 1.1 is specified and TLS 1.0 is enabled, the server will still attempt to connect using TLS 1.0 if TLS 1.1 is not accepted by the remote host.

Parameters

session_ptr	Pointer to a TLS Session instance.
protocol_version	TLS version macro for the specific TLS version to use.

Return Values

NX_SUCCESS	(0x00)	Successful state change.
NX_PTR_ERROR	(0x07)	Invalid TLS session pointer.
NX_SECURE_TLS_UNSUPPORTED_TLS_VERSION	(0x110)	Known but unsupported TLS version.
NX_SECURE_TLS_UNKNOWN_TLS_VERSION	(0x10F)	Invalid protocol version.

Allowed From

Threads

Example

```
/* Set the protocol version to be used to TLSv1.1. */
status = nx_secure_tls_session_protocol_version_override(&tls_session,
                                             NX_SECURE_TLS_VERSION_TLS_1_1);

/* This TLS Session will use TLSv1.1 for the handshake and TLS session. */
status = nx_secure_tls_session_start(&tls_session, &tcp_socket, NX_WAIT_FOREVER);
```

See Also

nx_secure_tls_session_start, nx_secure_tls_session_create

nx_secure_tls_session_receive

Receive data from a NetX Secure TLS Session

Prototype

```
UINT  nx_secure_tls_session_receive(NX_SECURE_TLS_SESSION *session_ptr,
                                     NX_PACKET **packet_ptr,
                                     ULONG wait_option);
```

Description

This service receives data from the specified active TLS session, handling the decryption of that data before providing it to the caller in the NX_PACKET parameter. If no data is queued in the specified session, the call suspends based on the supplied wait option.

If NX_SUCCESS is returned, the application is responsible for releasing the received packet when it is no longer needed.

Parameters

session_ptr	Pointer to a TLS Session instance.
packet_ptr	Pointer to an allocated TLS packet pointer.
wait_option	Indicates how long the service should wait for a packet from the remote host before returning.

Return Values

NX_SUCCESS	(0x00)	Successful initialization of the TLS session.
NX_NO_PACKET	(0x01)	No data received.
NX_NOT_CONNECTED	(0x38)	The underlying TCP socket is no longer connected.
NX_SECURE_TLS_HASH_MAC_VERIFY_FAILURE	(0x108)	A received message failed an authentication hash check.
NX_SECURE_TLS_UNKNOWN_TLS_VERSION	(0x10F)	A received message contained an unknown protocol version in its header.
NX_SECURE_TLS_ALERT_RECEIVED	(0x114)	Received a TLS alert from the remote host.

NX_PTR_ERROR	(0x07)	Tried to use an invalid pointer.
NX_SECURE_TLS_SESSION_UNINITIALIZED	(0x101)	The supplied TLS session was not initialized.

Allowed From

Threads

Example

```
/* Receive a packet from an active TLS session previously created and started with
nx_secure_tls_session_start. Wait until a packet is received, blocking otherwise.
*/
status =  nx_secure_tls_session_receive(&tls_session, &packet_ptr,
NX_WAIT_FOREVER);


/* If status is NX_SUCCESS the received packet is pointed to by  "packet_ptr". */
```

See Also

nx_tcp_socket_receive, nx_secure_x509_certificate_initialize, nx_secure_tls_remote_certificate_allocate, nx_secure_tls_session_start, nx_secure_tls_session_delete, nx_secure_tls_session_send, nx_secure_tls_session_end, nx_secure_tls_session_create

nx_secure_tls_session_renegotiate_callback_set

Assign a callback that will be invoked at the beginning of a session renegotiation

Prototype

```
UINT  nx_secure_tls_ session_renegotiate_callback_set (
                 NX_SECURE_TLS_SESSION *tls_session,
                 ULONG (*func_ptr)(struct NX_SECURE_TLS_SESSION_struct
                 *session));
```

Description

This service assigns a callback to a TLS session that will be invoked whenever the first message of a session renegotiation handshake is received from the remote host.

The callback function is intended as a notification to the application that a renegotiation handshake is beginning – the application may choose to terminate the TLS session by returning any non-zero value from the callback which will cause TLS to end the TLS session with an error. If the application wishes to proceed with the renegotiation, the callback should return NX_SUCCESS.

Note that due to the semantics of TLS renegotiation, the callback will be invoked for NetX Secure TLS Clients whenever a HelloRequest is received from the remote server, but not when the client initiates the renegotiation. On NetX Secure TLS Servers, the callback will be invoked whenever a renegotiation ClientHello is received (any ClientHello received in the context of an active TLS session). This means that the callback will be invoked whether the remote host or the local application has initiated the renegotiation because the TLS server will send a HelloRequest to which the remote client will respond.

NetX Secure TLS implements the Secure Renegotiation Inidication Extension from RFC 5746 to ensure that renegotiation handshakes are not subject to man-in-the-middle attacks.

Parameters

session_ptr	Pointer to TLS Session instance.
func_ptr	Pointer to callback function.

Return Values

NX_SUCCESS	(0x00)	Successful assignment of callback.
NX_PTR_ERROR	(0x07)	Tried to use an invalid pointer for the callback function or TLS session.

Allowed From

Threads

Example

```
/* Simple callback notifying the user that a renegotiation is starting. */
ULONG tls_renegotiation_callback(NX_SECURE_TLS_SESSION *session)
{
    printf("Renegotiation initiated by remote host\n");

    return(NX_SUCCESS);
}

/* Establish a TLS session with a remote host (TLS Client) */
status =  nx_secure_tls_session_create(&tls_session,
                                       &nx_crypto_tls_ciphers,
                                       crypto_metadata,
                                       sizeof(crypto_metadata));


/* Set callback for renegotiation notification. */
status = nx_secure_tls_session_renegotiate_callback_set(&tls_session,
                                               tls_renegotiation_callback);
```

See Also

nx_secure_tls_session_create, nx_secure_tls_session_start,
nx_secure_tls_session_receive, nx_secure_tls_session_send
nx_secure_tls_session_renegotiate

nx_secure_tls_session_renegotiate

Initiate a session renegotiation handshake with the remote host

Prototype

```
UINT  nx_secure_tls_ session_renegotiate (
                NX_SECURE_TLS_SESSION *tls_session,
                UINT wait_option);
```

Description

This service initiates a session *renegotiation* handshake with a connected remote TLS host. A renegotiation consists of a second TLS handshake within the context of a previously-established TLS session. Each of the new handshake messages is encrypted using the TLS session until new session keys are generated and ChangeCipherSpec messages are exchanged, at which time the new keys are used to encrypt all messages.

A renegotiation can be initiated at any time once a TLS session is established. If a renegotiation is attempted during a TLS handshake or before a TLS session is established no action will be taken.

Note that an entire TLS handshake will be performed when this service is invoked so the time to completion and returned status will vary depending on the current TLS settings and session parameters.

NetX Secure TLS implements the Secure Renegotiation Inidication Extension from RFC 5746 to ensure that renegotiation handshakes are not subject to man-in-the-middle attacks.

Parameters

session_ptr	Pointer to TLS Session instance.
wait_option	Indicates how long the service should wait for a packet from the remote host before returning. This is passed to all NetX services within TLS.

Return Values

NX_SUCCESS	(0x00)	Successful renegotiation.
NX_NO_PACKET	(0x01)	No data received.
NX_NOT_CONNECTED	(0x38)	The underlying TCP socket is no longer connected.
NX_SECURE_TLS_HASH_MAC_VERIFY_FAILURE	(0x108)	A received message failed an authentication hash check.
NX_SECURE_TLS_UNKNOWN_TLS_VERSION	(0x10F)	A received message contained an unknown protocol version in its header.
NX_SECURE_TLS_ALERT_RECEIVED	(0x114)	Received a TLS alert from the remote host.
NX_SECURE_TLS_RENEGOTIATION_SESSION_INACTIVE	(0x134)	The local or remote TLS session is inactive, making renegotiation impossible.
NX_SECURE_TLS_RENEGOTIATION_FAILURE	(0x13A)	The remote host did not provide either the SCSV or Secure Renegotiation Extension and thus renegotiation cannot be performed.
NX_PTR_ERROR	(0x07)	Tried to use an invalid pointer.
NX_SECURE_TLS_SESSION_UNINITIALIZED	(0x101)	The supplied TLS session was not initialized.
NX_SECURE_TLS_ALLOCATE_PACKET_FAILED	(0x111)	Underlying packet allocation failed.

Allowed From

Threads

Example

```
/* Establish a TLS session with a remote host (TLS Client) */
status =  nx_secure_tls_session_create(&tls_session,
                                       &nx_crypto_tls_ciphers,
                                       crypto_metadata,
                                       sizeof(crypto_metadata));


/* Setup a client socket connection.  */
status = nx_tcp_client_socket_connect(&tcp_socket, server_ipv4_address,
REMOTE_SERVER_PORT, NX_WAIT_FOREVER);
```

```
/* Start the TLS session. */
status = nx_secure_tls_session_start(&tls_session, &tcp_socket, NX_WAIT_FOREVER);

/* Send some data in the first TLS session. (Check “status” for errors…)*/
status = nx_secure_tls_packet_allocate(&tls_session, &pool_0, &send_packet,
                                       NX_WAIT_FOREVER);
status = nx_packet_data_append(send_packet, "Hello there!\r\n", 14, &pool_0,
                               NX_WAIT_FOREVER);
status = nx_secure_tls_session_send(&tls_session, send_packet,
                                    NX_IP_PERIODIC_RATE);

/* Now renegotiate the session. */
status  = nx_secure_tls_session_renegotiate(&tls_session, NX_WAIT_FOREVER);

/* If status == NX_SUCCESS, new TLS session keys have been generated. */

/* Send some data in the new TLS session. This will be encrypted using the new
   keys. */
status = nx_secure_tls_packet_allocate(&tls_session, &pool_0, &send_packet,
                                       NX_WAIT_FOREVER);
status = nx_packet_data_append(send_packet, "Another message…\r\n", 18, &pool_0,
                               NX_WAIT_FOREVER);
status = nx_secure_tls_session_send(&tls_session, send_packet,
                                    NX_IP_PERIODIC_RATE);
```

See Also

nx_secure_tls_session_create, nx_secure_tls_session_start,
nx_secure_tls_session_receive, nx_secure_tls_session_send
nx_secure_tls_session_renegotiation_callback_set

nx_secure_tls_session_reset

Clear out and reset a NetX Secure TLS Session

Prototype

```
UINT  nx_secure_tls_session_reset (NX_SECURE_TLS_SESSION *session_ptr);
```

Description

This service clears out a TLS session and resets the state as if the session had just been created so an existing TLS session object can be re-used for a new session.

Parameters

session_ptr	Pointer to a TLS Session instance.

Return Values

NX_SUCCESS	(0x00)	Successful initialization of the TLS session.
NX_INVALID_PARAMETERS	(0x4D)	Invalid TLS session pointer.
NX_PTR_ERROR	(0x07)	Tried to use an invalid pointer.

Allowed From

Threads

Example

```
/* Reset a TLS session.  */
status =  nx_secure_tls_session_reset(&tls_session);


/* If status is NX_SUCCESS the session was successfully reset and may be reused.*/
```

See Also

nx_secure_x509_certificate_initialize, nx_secure_tls_remote_certificate_allocate, nx_secure_tls_session_start, nx_secure_tls_session_delete, nx_secure_tls_session_send, nx_secure_tls_session_receive, nx_secure_tls_session_create

nx_secure_tls_session_send

Send data through a NetX Secure TLS Session

Prototype

```
UINT  nx_secure_tls_session_send(NX_SECURE_TLS_SESSION *session_ptr,
                                      NX_PACKET *packet_ptr,
                                      ULONG wait_option);
```

Description

This service sends data in the supplied NX_PACKET, using the specified active TLS session, and handling the encryption of that data before sending it to the remote host. If the receiver's last advertised window size is less than this request, the service optionally suspends based on the wait options specified.

Unless an error is returned, the application should not release the packet after this call. Doing so will cause unpredictable results because the network driver will release the packet after transmission.

Parameters

session_ptr	Pointer to a TLS Session instance.
packet_ptr	Pointer to a TLS packet containing data to be sent.
wait_option	Defines how the service behaves if the request is greater than the window size of the receiver.

Return Values

NX_SUCCESS	(0x00)	Successful initialization of the TLS session.
NX_NO_PACKET	(0x01)	No data received.
NX_NOT_CONNECTED	(0x38)	The underlying TCP socket is no longer connected.
NX_SECURE_TLS_TCP_SEND_FAILED	(0x109)	The underlying TCP socket send failed.
NX_PTR_ERROR	(0x07)	Tried to use an invalid pointer.
NX_SECURE_TLS_SESSION_UNINITIALIZED	(0x101)	The supplied TLS session was not initialized.

Allowed From

Threads

Example

```
/* Send a packet using an active TLS session previously created and started with
nx_secure_tls_session_start. Wait until a packet is sent, blocking otherwise.   */
status =  nx_secure_tls_session_send(&tls_session, &packet_ptr, NX_WAIT_FOREVER);


/* If status is NX_SUCCESS the packet has been sent. */
```

See Also

nx_tcp_socket_receive, nx_secure_x509_certificate_initialize, nx_secure_tls_remote_certificate_allocate, nx_secure_tls_packet_allocate, nx_secure_tls_session_start, nx_secure_tls_session_delete, nx_secure_tls_session_receive, nx_secure_tls_session_end, nx_secure_tls_session_create

nx_secure_tls_session_server_callback_set

Set up a callback for TLS to invoke at the beginning of a TLS Server handshake

Prototype

```
UINT  nx_secure_tls_ session_server_callback_set (
                  NX_SECURE_TLS_SESSION *tls_session,
                  ULONG (*func_ptr)(NX_SECURE_TLS_SESSION *tls_session,
                                    NX_SECURE_TLS_HELLO_EXTENSION
                                    *extensions, UINT num_extensions));
```

Description

This service assigns a function pointer to a TLS session that TLS will invoke when a TLS Server handshake has received a ClientHello message. The callback function allows an application to process any TLS extensions from the received ClientHello message that require input or decision making.

The callback is executed with the invoking TLS session control block and an array of NX_SECURE_TLS_HELLO_EXTENSION objects. The array of extension objects is intended to be passed into a helper function that will find and parse a specific extension. For an example helper function that parses TLS extensions provided in hello messages, see *nx_secure_tls_session_sni_extension_parse*.

The server callback may also be used to select the active identity certificate using *nx_secure_tls_active_certificate_set* for the TLS Server. This will most often occur in response to a Server Name Indication (SNI) extension which allows a TLS Client to indicate which server it is attempting to contact. See the references for *nx_secure_tls_session_sni_extension_parse* and *nx_secure_tls_active_certificate_set* for more information.

Parameters

session_ptr	Pointer to a previously created TLS Session instance.
func_ptr	Pointer to the TLS Server callback function.

Return Values

NX_SUCCESS	(0x00)	Successful allocation of Function pointer.
NX_PTR_ERROR	(0x07)	Invalid TLS session pointer.

Allowed From

Threads

Example

```
/* Application-supplied function to map server DNS name to a specific certificate
   ID. The certificate ID is supplied by the application when the certificate is
   added with nx_secure_tls_server_certificate_add. */
UINT application_find_id_by_dns_name(NX_SECURE_X509_DNS_NAME *dns_name)
{
    if(strncmp(dns_name->nx_secure_x509_dns_name, “server_name”,
               dns_name->nx_secure_x509_dns_name_length) == 0)
    {
       /* DNS name matches one we know, return it’s ID. */
       return(0x12);
    }

    /* Unknown DNS name, return 0 to indicate no matching ID found. */
    return(0);
}

/* Callback routine used to process ClientHello extensions and perform other
   runtime actions at the beginning of a TLS handshake (such as selecting an
   identify certificate to provide to the client). */
ULONG tls_server_callback(NX_SECURE_TLS_SESSION *session,
                         NX_SECURE_TLS_HELLO_EXTENSION *extensions, UINT
                         num_extensions)
{
UINT status;
NX_SECURE_X509_DNS_NAME dns_name;
UINT cert_id;
NX_SECURE_X509_CERT *certificate;


    /* Find and parse a Server Name Indication (SNI) extension. */
    status = nx_secure_tls_session_sni_extension_parse(session, extensions,
                                                      num_extensions, &dns_name);

    if(status != NX_SUCCESS && status != NX_SECURE_TLS_EXTENSION_NOT_FOUND)
    {
        /* Parsed an invalid extension, return the error. */
        return(status);
    }

    if(status == NX_SECURE_TLS_EXTENSION_NOT_FOUND)
    {
        /* SNI extension not found, just return success to use the default
          certificate. */
       return(NX_SUCCESS);
    }

    /* Find the application-supplied numeric identifier for the specified DNS
       name provided by the remote client. */
    cert_id = application_find_id_by_dns_name(&dns_name);

    /* If cert_id is 0, just use the default identity certificate added with
       nx_secure_tls_local_certificate_add. */
    if(cert_id != 0)
    {
        /* Application found a matching name, find the certificate in our
           store. */
        status = nx_secure_tls_server_certificate_find(tls_session, &certificate,
                                                      cert_id);

       if(status != NX_SUCCESS)
       {
           /* Didn’t find a valid certificate with the supplied ID. Return an
```

```
               error so TLS can shut down the handshake. */
            return(NX_SECURE_TLS_CERTIFICATE_NOT_FOUND);
        }

        /* Set the active identity certificate - the certificate should have been
           added to the local store at application start with
           nx_secure_tls_local_certificate_add. */
        nx_secure_tls_active_certificate_set(session, certificate);
    }

    return(NX_SUCCESS);
}

/* TLS setup routine. */
UINT tls_setup(NX_SECURE_TLS_SESSION *tls_session)
{
        /* Add callback to TLS session.  */
        status =  nx_secure_tls_session_server_callback_set(tls_session,
                                                            server_callback);

        /* If status is NX_SUCCESS the callback was added and will be invoked
           immediately after a ClientHello message is received. */

        return(status);
}

**See also the example for nx_secure_tls_active_certificate_set.
```

See Also

nx_secure_tls_session_create,
nx_secure_tls_session_client_callback_set,
nx_secure_tls_active_certificate_set,
nx_secure_tls_session_sni_extension_parse,
nx_secure_tls_server_certificate_add,
nx_secure_tls_server_certificate_find

nx_secure_tls_session_sni_extension_parse

Parse a Server Name Indication (SNI) extension received from a TLS Client

Prototype

```
UINT  nx_secure_tls_session_sni_extension_parse(
                              NX_SECURE_TLS_SESSION *session_ptr,
                              NX_SECURE_TLS_HELLO_EXTENSION
                              *extensions,
                              UINT num_extensions,
                              NX_SECURE_X509_DNS_NAME *dns_name);
```

Description

This service is intended to be called from within a TLS Server session callback, added to a TLS session using nx_secure_tls_session_server_callback_set. The callback is invoked following the reception of a ClientHello message from a remote TLS client and is supplied an array of available extensions (and the number of extensions in the array). That array and its length can be passed directly to this routine to determine if there is an SNI extension present – if not, NX_SECURE_TLS_EXTENSION_NOT_FOUND is returned indicating simply that the client opted not to provice the SNI extension (this is not an error).

If the SNI extension is found, the X.509 DNS name supplied by the TLS client is returned in the dns_name structure. Currently, the SNI extension only supplies a single DNS name entry, which may be used by the TLS server to determine which identity certificate to send to the remote client.

The NX_SECURE_X509_DNS_NAME structure simply contains the DNS name as a UCHAR string in the field *nx_secure_x509_dns_name* and the length of the name string in *nx_secure_x509_dns_name_length*. The macro NX_SECURE_X509_DNS_NAME_MAX controls the size of the nx_secure_x509_dns_name buffer.

Parameters

session_ptr	Pointer to a TLS Session instance.
extensions	Pointer to an array of TLS Hello extensions (from session callback).
num_extensions	Number of extensions in array (from session callback).
dns_name	Return DNS name supplied in the SNI extension.

Return Values

NX_SUCCESS	(0x00)	Successful parsing of the extension.
NX_PTR_ERROR	(0x07)	Invalid extensions array or TLS session pointer.
NX_SECURE_TLS_EXTENSION_NOT_FOUND	(0x136)	SNI extension not found.
NX_SECURE_TLS_SNI_EXTENSION_INVALID	(0x137)	SNI extension format was invalid.

Allowed From

Threads

Example

```
*See example for nx_secure_tls_server_callback_set.
**See also the example for nx_secure_tls_active_certificate_set.
```

See Also

nx_secure_tls_session_server_callback_set,
nx_secure_tls_session_client_callback_set

nx_secure_tls_session_sni_extension_set

Set a Server Name Indication (SNI) extension DNS name to send to a remote Server

Prototype

```
UINT  nx_secure_tls_session_sni_extension_set(
                                        NX_SECURE_TLS_SESSION *session_ptr,
                                        NX_SECURE_X509_DNS_NAME *dns_name);
```

Description

This service allows a TLS Client application to provide a preferred server DNS name to a remote TLS server using the Server Name Indication (SNI) TLS extension. The SNI extension allows the server to select the proper identity certificate and parameters based on the client's indicated server preference. The SNI extension currently only supports a single DNS name to be sent, hence the singular name parameter. The dns_name parameter must be initialized with *nx_secure_x509_dns_name_initialize* and will contain the client's preferred server. To unset the extension name, simply call this service with a "dns_name" parameter value of NX_NULL.

The NX_SECURE_X509_DNS_NAME structure simply contains the DNS name as a UCHAR string in the field *nx_secure_x509_dns_name* and the length of the name string in *nx_secure_x509_dns_name_length*. The macro NX_SECURE_X509_DNS_NAME_MAX controls the size of the nx_secure_x509_dns_name buffer.

NOTE: this routine must be called before nx_secure_tls_session_start is invoked or the ClientHello will not contain the SNI extension.

Parameters

session_ptr	Pointer to a TLS Session instance.
dns_name	DNS name supplied by the application.

Return Values

NX_SUCCESS	(0x00)	Successful addition of DNS server name.
NX_PTR_ERROR	(0x07)	Invalid DNS name or TLS session pointer.

Allowed From

Threads

Example

```
#define TLS_SNI_SERVER_NAME “www.example.com”

NX_SECURE_X509_DNS_NAME dns_name;
NX_SECURE_TLS_SESSION client_tls_session;

/* Application where TLS session is started. */
void main()
{
    /* 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_tls_session_create(&client_tls_session,
                                           &nx_crypto_tls_ciphers,
                                           server_crypto_metadata,
                                           sizeof(server_crypto_metadata));


    /* Initialize the DNS server name we want to send in the SNI extension. */
    nx_secure_x509_dns_name_initialize(&dns_name, TLS_SNI_SERVER_NAME,
                                   strlen(TLS_SNI_SERVER_NAME));

    /* The SNI server name needs to be set prior to starting the TLS session. */
    nx_secure_tls_session_sni_extension_set(&tls_session, &dns_name);

   /* Start TLS session as normal. */
}
```

See Also

nx_secure_tls_session_start, nx_secure_x509_dns_name_initialize

nx_secure_tls_session_start

Start a NetX Secure TLS Session

Prototype

```
UINT  nx_secure_tls_session_start(NX_SECURE_TLS_SESSION *session_ptr,
                                  NX_TCP_SOCKET *tcp_socket_ptr,
                                  ULONG wait_option);
```

Description

This service starts a TLS session using the supplied TLS session control block and a connected TCP socket. The TCP connection must already be complete following a successful call to either nx_tcp_client_socket_connect or nx_tcp_server_socket_accept.

This service will determine the type of TLS session (Client or Server) from the TCP socket.

The wait option defines how the service behaves while the TLS handshake is in progress.

Parameters

session_ptr	Pointer to a TLS Session instance.
tcp_socket_ptr	Pointer to a connected TCP socket.
wait_option	Defines how the service behaves while the TLS handshake is in progress.

Return Values

NX_SUCCESS	(0x00)	Successful initialization of the TLS session.
NX_NOT_CONNECTED	(0x38)	The underlying TCP socket is no longer connected.
NX_SECURE_TLS_UNRECOGNIZED_MESSAGE_TYPE	(0x102)	A received TLS 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 TLS 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 TLS stack.

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

NX_SECURE_TLS_UNSUPPORTED_TLS_VERSION
(0x110) A received TLS message had a known but unsupported TLS 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) Tried to use an invalid pointer.

Allowed From

Threads

Example

```
NX_TCP_SOCKET tcp_socket;
NX_SECURE_TLS_SESSION tls_session;
NX_SECURE_X509_CERT certificate;

/* Initialize the TLS session structure. */
nx_secure_tls_session_create(&tls_session,
                              &nx_crypto_tls_ciphers,
                              crypto_metadata,
                              sizeof(crypto_metadata));

/* Setup the TLS packet reassembly buffer. */
status = nx_secure_tls_session_packet_buffer_set(&tls_session, tls_packet_buffer,
                                                  sizeof(tls_packet_buffer));

/* Initialize a certificate for the TLS server. */
status =  nx_secure_x509_certificate_initialize(&certificate, certificate_data, 500,
NX_NULL, 0, private_key, 64);

/* If status is NX_SUCCESS, certificate is initialized. */

/* Add the certificate a local certificate to identify this TLS server. */
status = nx_secure_tls_add_local_certificate(&tls_session, &certificate);

/* If status is NX_SUCCESS, certificate was added successfully. */

/* Create and start a TCP socket as a server. */
/* NOTE: This assumes an IP instance called “ip_0” has already been created. */

/* Create a TCP server socket on the previously created IP instance, with normal
   delivery, IP fragmentation enabled, default time to live, a 8192-byte receive
   window, no urgent callback routine, and the "client_disconnect" routine to
   handle disconnection initiated from the other end of the connection.  */
status =  nx_tcp_socket_create(&ip_0, &tcp_socket, "TLS Server Socket", NX_IP_NORMAL,
                               NX_FRAGMENT_OKAY, NX_IP_TIME_TO_LIVE, 8192, NX_NULL,
                               NX_NULL);

/* If status is NX_SUCCESS, the TCP socket was created successfully. */

/* Start up a TCP server socket by listening on port 443 with a listen queue of
   size 5. */
status =  nx_tcp_server_socket_listen(&ip_0, 443, &tcp_socket, 5, NX_NULL);

/* Application main loop. */
while(1)
{
        /* Accept incoming requests on the TCP socket. */
        status =  nx_tcp_server_socket_accept(&tcp_socket, NX_WAIT_FOREVER);

        /* At this point, the TCP socket should be established (but not the TLS
           session yet). */

        /* Start the TLS session on our active TCP socket. */
        status = nx_secure_tls_session_start(&tls_session, &tcp_socket,
                                              NX_WAIT_FOREVER);

        /* At this point, if status is NX_SUCCESS, the TLS session has been
           established.  Application may now send/receive data through this
           channel. */

        /* Send and receive data using the TLS session. */
        /* Allocate TLS packets using nx_secure_tls_packet_allocate, write data with
           nx_packet_data_append, read data with nx_packet_data_extract_offset. */
        /* Send TLS data. */
        nx_secure_tls_session_send(&tls_session, &send_packet, NX_WAIT_FOREVER);

        nx_secure_tls_session_receive(&tls_session, &receive_packet_ptr,
        NX_WAIT_FOREVER);


        /* Once all application data is sent/received, end the TLS session. */
        status = nx_secure_tls_session_end(&tls_session);

        /* If status is NX_SUCCESS, the TLS session has been closed properly. */

        /* Now disconnect the TCP server socket from the client.  */
        nx_tcp_socket_disconnect(&tcp_socket, 200);

        /* Unaccept the TCP server socket.  Note that unaccept is called even if
           disconnect or accept fails.  */
        nx_tcp_server_socket_unaccept(&tcp_socket);

        /* Setup server socket for listening with this socket again.  */
        nx_tcp_server_socket_relisten(&ip_0, 443, &tcp_socket);
}

/* When the server application is shut down, clean up the TLS session. */
nx_secure_tls_session_delete(&tls_session);

/* Finally, clean up the TCP socket as well. */
nx_tcp_socket_delete(&tcp_socket);
```

See Also

nx_tcp_socket_receive, nx_secure_x509_certificate_initialize, nx_secure_tls_remote_certificate_allocate, nx_secure_tls_session_send, nx_secure_tls_session_delete, nx_secure_tls_session_receive, nx_secure_tls_packet_allocate, nx_secure_tls_session_end, nx_secure_tls_session_create,
nx_tcp_socket_accept, nx_tcp_socket_listen, nx_tcp_socket_disconnect, nx_tcp_socket_unaccept, nx_tcp_socket_relisten, nx_tcp_socket_delete, nx_packet_allocate, nx_packet_data_append, nx_packet_data_extract_offset

nx_secure_tls_timestamp_function_set

Assign a timestamp function to a NetX Secure TLS Session

Prototype

```
UINT  nx_secure_tls_timestamp_function_set(
                               NX_SECURE_TLS_SESSION *session_ptr,
                               ULONG (*time_func_ptr)(void));
```

Description

This function sets up a function pointer that TLS will invoke when it needs to get the current time, which is used in various TLS handshake messages and for verification of certificates.

The function is expected to return the current GMT in UNIX 32-bit format (seconds since the midnight starting Jan 1, 1970, UTC, ignoring leap seconds), as per the ClientHello requirements in the TLS RFC 5246.

If no timestamp function is assigned, a value of 0 for the timestamp in the TLS handshake will be used and certificate expiration checking will not work.

Parameters

session_ptr	Pointer to a TLS Session instance.
time_func_ptr	Pointer to a function that returns the current time (GMT) in UNIX 32-bit format.

Return Values

NX_SUCCESS	(0x00)	Successful initialization of the TLS session.
NX_INVALID_PARAMETERS	(0x4D)	Invalid TLS session pointer.
NX_PTR_ERROR	(0x07)	Tried to use an invalid pointer.

Allowed From

Threads

Example

```
/* This function returns a 32-bit UNIX-style representation of the current GMT
   time. */
ULONG get_gmt_time(void)
{
ULONG time_value;

   /* Platform-specific time calculation goes here… */

   return(time_value);
}

/* Reset a TLS session.  */
status =  nx_secure_tls_timestamp_function_set(&tls_session, get_gmt_time);


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

See Also

nx_secure_x509_certificate_initialize, nx_secure_tls_remote_certificate_allocate, nx_secure_tls_session_start, nx_secure_tls_session_delete, nx_secure_tls_session_send,nx_secure_tls_session_receive, nx_secure_tls_session_create

nx_secure_tls_trusted_certificate_add

Add trusted certificate to NetX Secure TLS Session

Prototype

```
UINT  nx_secure_tls_trusted_certificate_add(NX_SECURE_TLS_SESSION
*session_ptr, NX_SECURE_X509_CERT *certificate_ptr);
```

Description

This service adds an initialized NX_SECURE_X509_CERT structure instance to a TLS session. This certificate is used by the TLS stack to verify certificates supplied by the remote host during the TLS handshake.

Trusted certificates are required for TLS Client mode.

Trusted certificates are only required for TLS Server mode if client certificate authentication is enabled.

Parameters

session_ptr	Pointer to a previously created TLS Session instance.
certificate_ptr	Pointer to an initialized TLS Certificate instance.

Return Values

NX_SUCCESS	(0x00)	Successful addition of certificate.
NX_INVALID_PARAMETERS	(0x4D)	Tried to add an invalid certificate.
NX_PTR_ERROR	(0x07)	Invalid TLS session pointer.

Allowed From

Threads

Example

```
/* Initialize certificate structure. */
status = nx_secure_x509_certificate_initialize(&certificate, certificate_data,
                                               certificate_buffer,
                                               sizeof(certificate_buffer), 500,
                                               private_key, 64);

/* Add certificate to TLS session.  */
status =  nx_secure_tls_trusted_certificate_add(&tls_session, &certificate);


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

See Also

nx_secure_x509_certificate_initialize, nx_secure_tls_session_create,
nx_secure_tls_remote_certificate_allocate,
nx_secure_tls_trusted_certificate_remove

nx_secure_tls_trusted_certificate_remove

Remove trusted certificate from NetX Secure TLS Session

Prototype

```
UINT  nx_secure_tls_trusted_certificate_remove(
                                    NX_SECURE_TLS_SESSION *session_ptr,
                                    UCHAR *common_name,
                                    UINT common_name_length);
```

Description

This service removes a trusted certificate from a TLS session, keyed on the Common Name field in the certificate.

Parameters

session_ptr	Pointer to a previously created TLS Session instance.
common_name	The Common Name value of the certificate to be removed.
common_name_length	The length of the Common Name string.

Return Values

NX_SUCCESS	(0x00)	Successful addition of certificate.
NX_PTR_ERROR	(0x07)	Invalid TLS session pointer.
NX_SECURE_TLS_CERTIFICATE_NOT_FOUND	(0x119)	Certificate was not found.

Allowed From

Threads

Example

```
/* Remove certificate from TLS session.  */
status =  nx_secure_tls_trusted_certificate_remove(&tls_session,
                                                 “www.example.com”, 15);


/* If status is NX_SUCCESS the certificate was successfully removed.  */
```

See Also

nx_secure_x509_certificate_initialize, nx_secure_tls_session_create, nx_secure_tls_remote_certificate_allocate, nx_secure_tls_trusted_certificate_add

nx_secure_x509_certificate_initialize

Initialize X.509 Certificate for NetX Secure TLS

Prototype

```
UINT  nx_secure_x509_certificate_initialize(
                  NX_SECURE_X509_CERT *certificate_ptr,
                  const UCHAR *certificate_data,
                  USHORT certificate_data_length,
                  UCHAR *raw_data_buffer,
                  USHORT buffer_size,
                  const UCHAR *private_key_data,
                  USHORT private_key_data_length,
                  UINT private_key_type);
```

Description

This service initializes an NX_SECURE_X509_CERT structure from a binary-encoded X.509 digital certificate for use in a TLS session.

The certificate data **must** be a valid X.509 digital certificate in DER-encoded binary format. The data can some from any source (e.g. file system, compiled constant buffer, etc.) as long as a UCHAR pointer to that data is provided.

The *raw_data_buffer* parameter and its size are optional parameters that specify a dedicated buffer into which the certificate data is copied before parsing. If raw_data_buffer is passed as NX_NULL then the NX_SECURE_X509_CERT structure will point directly into the certificate_data array (buffer_size is ignored in this case). If raw_data_buffer is passed as NX_NULL, ***do not*** modify the data pointed to by the certificate_data pointer or certificate processing will likely fail!

The private key parameter is for local identity certificates – the private key is used by servers to decrypt the incoming key data from a client (encrypted using the server’s public key) and by clients to verify their identity to a server when the server requests a client certificate. Adding a private key with this API will automatically mark the associated certificate as being an identity certificate for a TLS application. When initializing certificates for other purposes (e.g. the trusted store), the *private_key_data* parameter should be passed as NULL, the *private_key_data_length* as 0, and the *private_key_type* should be passed as NX_SECURE_X509_KEY_TYPE_NONE.

The *private_key_type* parameter indicates the formatting of the private key. For example, if the private key is a DER-encoded PKCS#1-format RSA private key, the private_key_type should be passed as

NX_SECURE_X509_KEY_TYPE_RSA_PKCS1_DER, a type known to NetX Secure which will be parsed immediately and saved for later use.

The private_key_type also supports user-defined key types[7] for platforms and applications that have specific key formats or other needs. For example, a hardware-based encryption engine may use a specific format not understood by the NetX Secure software, or a private key may be encrypted or represented by a cryptographic token as might be the case with a Trusted Platform Module (TPM) or PKCS#11 cryptographic hardware. When a user-defined key type is used, the key data is passed verbatim to the appropriate cryptographic routine - for example, an RSA private key would be passed, without any parsing or processing, directly to the RSA cryptographic routine provided to TLS in the ciphersuite table. The user-defined key type is also passed to the cryptographic routine (in the case of RSA, this is the "op" parameter).

The range of user-defined keys covers the top half of a 32-bit unsigned integer, from 0x0001 0000-0xFFFF FFFF. Values less than 0x0001 0000 are reserved for NetX Secure use.

Parameters

certificate_ptr	Pointer to an uninitialized X.509 Certificate instance.
certificate_data	Pointer to DER-encoded X.509 binary data.
raw_data_buffer	Pointer to optional dedicated certificate data buffer.
buffer_size	Size of optional dedicated certificate data buffer.
certificate_data_length	Length of certificate binary data in bytes.
private_key_data	Pointer to optional private key data.
private_key_data_length	Length of private key data.
private_key_type	Key type identifier.

[7] User-defined key types are an advanced feature that requires custom cryptographic routines to handle the raw private key data. Please contact your Express Logic representative if you have need of this feature.

Return Values

NX_SUCCESS	(0x00)	Successful addition of certificate.
NX_PTR_ERROR	(0x07)	Tried to use an invalid pointer.
NX_SECURE_TLS_INVALID_CERTIFICATE	(0x112)	Certificate data did not contain a DER-encoded X.509 certificate.
NX_SECURE_TLS_UNSUPPORTED_PUBLIC_CIPHER	(0x10D)	Certificate did not have a public-key cipher that is supported by NetX Secure.
NX_SECURE_X509_INVALID_CERTIFICATE_SEQUENCE	(0x186)	Private key or certificate did not contain a valid ASN.1 sequence.
NX_SECURE_PKCS1_INVALID_PRIVATE_KEY	(0x18A)	The provided private key was not a valid PKCS#1 RSA key.
NX_SECURE_X509_INVALID_PRIVATE_KEY_TYPE	(0x19D)	The private key type provided was not user-defined and did not match any known type.

Allowed From

Threads

Example

```
/* Initialize certificate structure. The certificate structure will point directly
   into the certificate_data array since we are passing the raw_data_buffer as
   NX_NULL. This certificate has a private key so it will be used to identify this
   device. */
status =  nx_secure_x509_certificate_initialize(&certificate, certificate_data,
500, NX_NULL, 0, private_key, 64, NX_SECURE_X509_KEY_TYPE_RSA_PKCS1_DER);


/* If status is NX_SUCCESS the certificate was successfully initialized.  */

** See also the section "Importing X.509 Certificates into NetX Secure" in Chapter
   3.
```

See Also

nx_secure_local_certificate_add, nx_secure_tls_session_create, nx_secure_tls_remote_certificate_allocate

nx_secure_x509_common_name_dns_check

Check DNS name against X.509 Certificate

Prototype

```
UINT  nx_secure_x509_common_name_dns_check(
                        NX_SECURE_X509_CERT *certificate,
                        const UCHAR *dns_tld, UINT dns_tld_length);
```

Description

This service checks a certificate's Common Name against a Top-Level Domain name (TLD) provided by the caller for the purposes of DNS validation of a remote host. This utility function is intended to be called from within a certificate validation callback routine provided by the application. The TLD name should be the top part of the URL used to access the remote host (the "."-separated string before the first slash).
If the Common Name contains a wildcard (such as *.example.com), the wildcard will match any with the same suffix. Note that only the first wildcard ("*") encountered (reading right-to-left) will be considered for wildcard matching – for example, abc.*.example.com will match *any* name ending in ".example.com".

If the Common Name does not match the provided string, the "subjectAltName" extension is parsed (if it exists in the certificate) and any DNSName entries are also compared. If none of those entries match, an error is returned.

It is important to understand the format of the common name (and subjectAltName entries) in expected certificates. For example, some certificates may use a raw IP address or a wild card. The DNS TLD string must be formatted such that it will match the expected values in received certificates.

Parameters

certificate_ptr	Pointer to an X.509 Certificate instance.
dns_tld	Top-Level Domain name to compare against.
dns_tld_length	Length of TLD string.

Return Values

NX_SUCCESS	(0x00)	Successful comparison with Common Name or subjectAltName.
NX_SECURE_X509_CERTIFICATE_DNS_MISMATCH	(0x195)	No matching name found.
NX_PTR_ERROR	(0x07)	Tried to use an invalid pointer.

Allowed From

Threads

Example

```
/* Callback routine used to perform additional checks on a certificate. */
ULONG certificate_callback(NX_SECURE_TLS_SESSION *session, NX_SECURE_X509_CERT
*certificate)
{
ULONG status;
UCHAR *tld = "www.example.com";

    /* Check our DNS TLD against the certificate provided by the
       remote TLS host. */
    status = nx_secure_x509_common_name_dns_check(certificate, tld, strlen(tld));

    if(status != NX_SUCCESS)
    {
        /* TLD did not match any names in the certificate. */
       return(status);
    }

    /* DNS validation and any other checks were successful. */
    return(NX_SUCCESS);
}

…

/* Add callback to TLS session in TLS setup routine.  */
status =  nx_secure_tls_session_certificate_callback_set(&tls_session,
                                                        certificate_callback);
```

See Also

nx_secure_tls_session_create,
nx_secure_tls_session_certificate_callback_set,
nx_secure_x509_crl_revocation_check

nx_secure_x509_crl_revocation_check

Check X.509 Certificate against a supplied Certificate Revocation List (CRL)

Prototype

```
UINT  nx_secure_x509_crl_revocation_check(const UCHAR *crl_data,
                                UINT crl_length,
                                NX_SECURE_X509_CERTIFICATE_STORE *store,
                                NX_SECURE_X509_CERT *certificate);
```

Description

This service takes a DER-encoded Certificate Revocation List and searches for a specific certificate in that list. The issuer of the CRL is validated against a supplied certificate store, the CRL issuer is validated to be the same as the one for the certificate being checked, and the serial number of the certificate in question is used to search the revoked certificates list. If the issuers match, the signature checks out, and the certificate is **not** present in the list, the call is successful. All other cases cause an error to be returned.

Parameters

crl_data	Pointer to a DER-encoded CRL.
crl_length	Length in bytes of CRL data.
store	Pointer to an X.509 Certificate store.
certificate_ptr	Pointer to an X.509 Certificate instance.

Return Values

NX_SUCCESS	(0x00)	Successful validation that the certificate was not revoked.
NX_SECURE_TLS_CERTIFICATE_NOT_FOUND	(0x119)	CRL issuer certificate not found.
NX_SECURE_TLS_ISSUER_CERTIFICATE_NOT_FOUND	(0x11B)	Certificate issuer certificate not found.
NX_SECURE_X509_ASN1_LENGTH_TOO_LONG	(0x182)	The CRL ASN.1 contained an invalid length field.
NX_SECURE_X509_UNEXPECTED_ASN1_TAG	(0x189)	The CRL contained invalid ASN.1.
NX_SECURE_X509_CHAIN_VERIFY_FAILURE	(0x18C)	A certificate chain verification failed.

NX_SECURE_X509_CRL_ISSUER_MISMATCH	(0x197)	CRL and certificate issuers did not match.
NX_SECURE_X509_CRL_SIGNATURE_CHECK_FAILED	(0x198)	The CRL signature was invalid.
NX_SECURE_X509_CRL_CERTIFICATE_REVOKED	(0x199)	The certificate being checked was found in the CRL and is therefore revoked.
NX_PTR_ERROR	(0x07)	Tried to use an invalid pointer.

Allowed From

Threads

Example

```
/* CRL obtained for the expected certificate issuer through some means (downloaded
   from server manually, obtained from CRL endpoint, etc…) */
const UCHAR *crl_data;
UINT crl_length = 300;

/* Callback routine used to perform additional checks on a certificate. */
ULONG certificate_callback(NX_SECURE_TLS_SESSION *session, NX_SECURE_X509_CERT
*certificate)
{
ULONG status;
NX_SECURE_X509_CERTIFICATE_STORE *store;

    /* Obtain a certificate store to check against. In the certificate callback,
       it usually makes the most sense to use the store associated with the TLS
       session. */
    store = &session -> nx_secure_tls_credentials.nx_secure_tls_certificate_store;

    /* Check our certificate against the CRL and TLS certificate store. */
    status = nx_secure_x509_crl_revocation_check(crl, crl_length, store,
                                                  certificate);

    if(status != NX_SUCCESS)
    {
        if(status == NX_SECURE_X509_CRL_CERTIFICATE_REVOKED)
        {
             /* Certificate was revoked. */
            return(status);
        }
        else
        {
            /* CRL was invalid or some other issue. In this case the certificate
               may still be valid since the CRL itself was a problem. At this
               point it is up to the application to decide whether to continue
               with the TLS handshake. For this example, assume certificate is
               valid (faulty CRL is a possible Denial-of-Service attack).*/
            status = NX_SUCCESS;
    }

    /* Other certificate checking can go here. */

    /* Return status of certificate checks. */
    return(status);
}
```

```
…

/* Add callback to TLS session in TLS setup routine.  */
status =  nx_secure_tls_session_certificate_callback_set(&tls_session,
                                                         certificate_callback);
```

See Also

nx_secure_tls_session_create,
nx_secure_tls_session_certificate_callback_set,
nx_secure_x509_common_name_dns_check

nx_secure_x509_dns_name_initialize

Initialize an X.509 DNS name structure

Prototype

```
UINT  nx_secure_x509_dns_name_initialize(
                        NX_SECURE_X509_DNS_NAME *dns_name,
                        const UCHAR *name_string, UINT length);
```

Description

This service initializes an X.509 DNS name for use with certain API services requiring a specific name format. For example, the *nx_secure_tls_sni_extension_parse* service expects an NX_SECURE_X509_DNS_NAME object in order to match the name provided by a remote host in the Server Name Indication extension during the TLS handshake. A DNS name is simply a charater string with a length – the maximum allowed length of a DNS name (and the size of the internal buffer in NX_SECURE_X509_DNS_NAME) is controlled by the macro NX_SECURE_X509_DNS_NAME_MAX (default 100 bytes).

Parameters

dns_name	DNS name structure to initialize.
name_string	DNS name string data.
length	Length of name string.

Return Values

NX_SUCCESS	(0x00)	Successful initialization.
NX_SECURE_X509_NAME_STRING_TOO_LONG	(0x19E)	The given name string exceeded NX_SECURE_X509_DNS_NAME_MAX.
NX_PTR_ERROR	(0x07)	Tried to use an invalid pointer.

Allowed From

Threads

Example

```
NX_SECURE_X509_DNS_NAME dns_name;

/* Initialize DNS name. */
status = nx_secure_x509_dns_name_initialize(&dns_name, “www.example.com”,
                                            strlen(“www.example.com”);

/* Use initialized DNS name to send the Server Name Indication extension to the
   remote TLS server. */
status = nx_secure_tls_session_sni_extension_set(&tls_session, &dns_name);
** See also the example for nx_secure_tls_session_sni_extension_set.
```

See Also

nx_secure_tls_session_create,
nx_secure_tls_session_sni_extension_parse,
nx_secure_tls_session_sni_extension_set

nx_secure_x509_extended_key_usage_extension_parse

Find and parse an X.509 extended key usage extension in an X.509 certificate

Prototype

```
UINT  nx_secure_x509_extended_key_usage_extension_parse(
                        NX_SECURE_X509_CERT *certificate,
                        UINT key_usage);
```

Description

This service is intended to be called from within a certificate verification callback (see *nx_secure_tls_session_certificate_callback_set)*. It will search for a specific extended key usage OID within an X.509 certificate and return whether the OID is present. The key_usage parameter is an integer mapping of the OIDs which is used internally by NetX Secure X.509 and TLS to avoid passing the variable-length OID strings as parameters.

The relevant OIDs for the extended key usage extension are given in the table below. A typical TLS Client implementation wishing to check extended key usage in a received TLS server certificate would check for the existence of the OID NX_SECURE_TLS_X509_TYPE_PKIX_KP_SERVER_AUTH – if the extension is present but that OID is not, then the certificate would be considered invalid for identifiying the host as a TLS server and the certificate verification callback should return an error. If the extension itself is missing, then it is up to the application whether or not to proceed with the TLS handshake.

In the certificate verification callback, the error return code NX_SECURE_X509_KEY_USAGE_ERROR is reserved for application use. If there is an error in checking key usage, this value may be returned from the callback to indicate the reason for failure.

NetX Secure Identifier	**OID Value**	Description
NX_SECURE_TLS_X509_TYPE_PKIX_KP_SERVER_AUTH	1.3.6.1.5.5.7.3.1	Certificate can be used to identify a TLS server
NX_SECURE_TLS_X509_TYPE_PKIX_KP_CLIENT_AUTH	1.3.6.1.5.5.7.3.2	Certificate can be used to identify a TLS client
NX_SECURE_TLS_X509_TYPE_PKIX_KP_CODE_SIGNING	1.3.6.1.5.5.7.3.3	Certificate can be used to sign code
NX_SECURE_TLS_X509_TYPE_PKIX_KP_EMAIL_PROTECT	1.3.6.1.5.5.7.3.4	Certificate can be used to sign emails

NX_SECURE_TLS_X509_TYPE_PKIX_KP_TIME_STAMPING	1.3.6.1.5.5.7.3.8	Certificate can be used to sign timestamps
NX_SECURE_TLS_X509_TYPE_PKIX_KP_OCSP_SIGNING	1.3.6.1.5.5.7.3.9	Certificate can be used to sign OCSP responses

OIDs and mappings for X.509 Extended Key Usage Extension

Parameters

certificate Pointer to certificate being verified.
key_usage OID integer mapping from table above.

Return Values

NX_SUCCESS	(0x00)	Specified key usage OID found.
NX_SECURE_X509_MULTIBYTE_TAG_UNSUPPORTED	(0x181)	ASN.1 multi-byte tag encountered (unsupported certificate).
NX_SECURE_X509_ASN1_LENGTH_TOO_LONG	(0x182)	Invaild ASN.1 field encountered (invalid certificate).
NX_SECURE_X509_INVALID_TAG_CLASS	(0x190)	Invalid ASN.1 tag class encountered (invalid certificate).
NX_SECURE_X509_INVALID_EXTENSION_SEQUENCE	(0x192)	Invalid extension encountered (invalid certificate).
NX_SECURE_X509_EXTENSION_NOT_FOUND	(0x19B)	The Extended Key Usage extension was not found in the provided certificate.
NX_PTR_ERROR	(0x07)	Invalid certificate pointer.

Allowed From

Threads

Example

```
ULONG certificate_verification_callback(NX_SECURE_TLS_SESSION *session,
NX_SECURE_X509_CERT* certificate)
{
UINT status;

    /* Extended key usage - look for specific OIDs. */
    status = nx_secure_x509_extended_key_usage_extension_parse(certificate,
                      NX_SECURE_TLS_X509_TYPE_PKIX_KP_SERVER_AUTH);

    if(status != NX_SUCCESS)
    {
        if(NX_SECURE_X509_EXT_KEY_USAGE_NOT_FOUND)
        {
            printf("Extended key usage extension not found or specified key usage OID not
                    provided in extension.\n");
            /* The certificate was valid but the specified OID was not found. The
               application can decide whether to continue or abort the TLS handshake. */
            return(NX_SECURE_X509_KEY_USAGE_ERROR);
        }
        else
        {
            /* The extension or certificate was invalid. */
            return(status);
        }
    }

    /* The specified OID was found, return success! */
    return(NX_SUCCESS);
}
```

See Also

nx_secure_tls_session_certificate_callback_set,
nx_secure_x509_key_usage_extension_parse,
nx_secure_x509_crl_revocation_check,
nx_secure_x509_common_name_dns_check,
nx_secure_x509_extension_find

nx_secure_x509_ extension_find

Find and return an X.509 extension in an X.509 certificate

Prototype

```
UINT  nx_secure_x509_extension_find(
                        NX_SECURE_X509_CERT *certificate,
                        NX_SECURE_X509_EXTENSION *extension,
                        USHORT extension_id);
```

Description

This service is intended to be called from within a certificate verification callback (see *nx_secure_tls_session_certificate_callback_set)* and is an advanced X.509 service.

The function will search for a specific extension within an X.509 certificate based on an OID and return whether the OID is present, along with a structure containing references to the relevant raw extension data. The extension_id parameter is an integer mapping of the OIDs which is used internally by NetX Secure X.509 and TLS to avoid passing the variable-length OID strings as parameters.

The helper functions provided for specific extensions (such as *nx_secure_x509_key_usage_extension_parse*) call nx_secure_x509_extension_find internally to obtain the extension data.

The relevant OIDs for known X.509 extensions are given in the table below.

The NX_SECURE_X509_EXTENSION structure contains pointers into the X.509 certificate that allow helper functions such as *nx_secure_x509_key_usage_extension_parse* to quickly decode the raw extension DER-encoded ASN.1 data.

For information on specific extensions, see RFC 5280 (X.509 specification) or the reference for the appropriate helper functions if available.

The current version of NetX Secure X.509 has limited support for X.509 extensions. More helper functions will be added in the future.

This service is an advanced feature for users familiar with X.509 extensions and DER-encoded ASN.1. It is provided to enable those users to access extensions for which NetX Secure X.509 does not currently *provide helper functions. For those extensions without helper functions, you will have to parse the raw DER-encoded ASN.1 yourself.*

NetX Secure Identifier	OID Value	Description	Helper function?
NX_SECURE_TLS_X509_TYPE_DIRECTORY_ATTRIBUTES	2.5.29.9	Directory Attributes – basic information attributes about certificate subject	No
NX_SECURE_TLS_X509_TYPE_SUBJECT_KEY_ID	2.5.29.14	Used to identify a specific public key	No
NX_SECURE_TLS_X509_TYPE_KEY_USAGE	2.5.29.15	Provides information on valid uses for the certificate public key	Yes
NX_SECURE_TLS_X509_TYPE_SUBJECT_ALT_NAME	2.5.29.17	Provides alternative DNS names to identify the certificate	Yes[8]
NX_SECURE_TLS_X509_TYPE_ISSUER_ALT_NAME	2.5.29.18	Provides alternative DNS names to identify the certificate's issuer	No
NX_SECURE_TLS_X509_TYPE_BASIC_CONSTRAINTS	2.5.29.19	Provides basic certificate usage constraint information	No
NX_SECURE_TLS_X509_TYPE_NAME_CONSTRAINTS	2.5.29.30	Used to constrain certificate names to specific domains	No
NX_SECURE_TLS_X509_TYPE_CRL_DISTRIBUTION	2.5.29.31	Provides URIs for CRL distribution	No
NX_SECURE_TLS_X509_TYPE_CERTIFICATE_POLICIES	2.5.29.32	List of certificate policies for large PKI systems	No

[8] The SubjectAltName extension is parsed as part of the DNS name check in the service *nx_secure_x509_common_name_dns_check*.

NX_SECURE_TLS_X509_TYPE_CERT_POLICY_MAPPINGS	2.5.29.33	List of CA certificate policies	No
NX_SECURE_TLS_X509_TYPE_AUTHORITY_KEY_ID	2.5.29.35	Used to identify a specific public key associated with a certificate signature	No
NX_SECURE_TLS_X509_TYPE_POLICY_CONSTRAINTS	2.5.29.36	CA policy constraints	No
NX_SECURE_TLS_X509_TYPE_EXTENDED_KEY_USAGE	2.5.29.37	Additional OID-based key usage information	Yes
NX_SECURE_TLS_X509_TYPE_FRESHEST_CRL	2.5.29.46	Provides information for obtaining delta CRLs	No
NX_SECURE_TLS_X509_TYPE_INHIBIT_ANYPOLICY	2.5.29.54	CA certificate field indicating that AnyPolicy cannot be used	No

OIDs and mappings for X.509 Extensions

Parameters

certificate Pointer to certificate being verified.

extension Return structure containing extension data pointer and length.

extension_id OID integer mapping from table above.

Return Values

NX_SUCCESS (0x00) Specified extension OID found and data returned.

NX_SECURE_X509_MULTIBYTE_TAG_UNSUPPORTED (0x181) ASN.1 multi-byte tag encountered (unsupported certificate).

NX_SECURE_X509_ASN1_LENGTH_TOO_LONG (0x182) Invaild ASN.1 field encountered (invalid certificate).

NX_SECURE_X509_INVALID_TAG_CLASS (0x190) Invalid ASN.1 tag class encountered (invalid certificate).

NX_SECURE_X509_INVALID_EXTENSION_SEQUENCE (0x192) Invalid extension encountered (invalid certificate).

NX_SECURE_X509_EXTENSION_NOT_FOUND	(0x19B)	The given extension OID was not found in the provided certificate.
NX_PTR_ERROR	(0x07)	Invalid certificate or extension pointer.

Allowed From

Threads

Example

```
/* Function to parse a Basic Constraints X.509 extension. */
UINT _basic_constraints_extension_parse(NX_SECURE_X509_CERT *certificate)
{
const UCHAR             *current_buffer;
ULONG                    length;
UINT                     status;
NX_SECURE_X509_EXTENSION extension_data;

    /* Find the Basic Constraints extension in the certificate. */
    status = _nx_secure_x509_extension_find(certificate, &extension_data,
                              NX_SECURE_TLS_X509_TYPE_BASIC_CONSTRAINTS);

    /* See if extension present - it might be OK if not present! */
    if (status != NX_SUCCESS)
    {
        return(status);
    }

    /* The extension_data structure now points to the raw extension ASN.1
      (DER-encoded). */
    current_buffer = extension_data.nx_secure_x509_extension_data;
    length = extension_data.nx_secure_x509_extension_data_length;

   /* Extension ASN.1 parsing… */

   return(NX_SUCCESS);
}
```

See Also

nx_secure_tls_session_certificate_callback_set,
nx_secure_x509_key_usage_extension_parse,
nx_secure_x509_crl_revocation_check,
nx_secure_x509_common_name_dns_check,
nx_secure_x509_extended_key_usage_extension_parse

nx_secure_x509_key_usage_extension_parse

Find and parse an X.509 Key Usage extension in an X.509 certificate

Prototype

```
UINT  nx_secure_x509_key_usage_extension_parse(
                              NX_SECURE_X509_CERT *certificate,
                              USHORT *bitfield);
```

Description

This service is intended to be called from within a certificate verification callback (see *nx_secure_tls_session_certificate_callback_set)*. It will search for the Key Usage extension and if found, will return the Key Usage bitfield in the "bitfield" parameter.

The bits, as defined by the X.509 specification (RFC 5280) are given in the table below. A bitwise AND with the appropriate bitmask (and checking for non-zero) will give the value of each bit.

Note that the DER-encoding of the bitfield eliminates extra zeroes so the actual position of the bits in the raw certificate data will likely be different from their positions in the decoded bitfield. The supplied bitmasks are only intended to be used on the decoded bitfield returned by *nx_secure_x509_key_usage_extension_parse* and not with the raw DER-encoded certificate data.

In the certificate verification callback, the error return code NX_SECURE_X509_KEY_USAGE_ERROR is reserved for application use. If there is an error in checking key usage, this value may be returned from the callback to indicate the reason for failure.

NetX Secure Identifier	Bit position	Description
NX_SECURE_X509_KEY_USAGE_DIGITAL_SIGNATURE	0	Certificate can be used for digital signatures
NX_SECURE_X509_KEY_USAGE_NON_REPUDIATION	1	Certificate can be used to verify digital signatures other than those for certificates and CRLs
NX_SECURE_X509_KEY_USAGE_KEY_ENCIPHERMENT	2	Certificate can be used to encrypt symmetric keys (key transport)
NX_SECURE_X509_KEY_USAGE_DATA_ENCIPHERMENT	3	Certificate can be used to directly encrypt raw user data (uncommon)
NX_SECURE_X509_KEY_USAGE_KEY_AGREEMENT	4	Certificate can be used for key agreement (as with Diffie-Hellman)

NX_SECURE_X509_KEY_USAGE_KEY_CERT_SIGN	5	Certificate can be used to sign and verify other certificates (the certificate is a CA or ICA certificate).
NX_SECURE_X509_KEY_USAGE_CRL_SIGN	6	Certificate public key is used to verify signatures on CRLs
NX_SECURE_X509_KEY_USAGE_ENCIPHER_ONLY	7	Used with Key Agreement bit (bit 4) – when set, certificate key can only be used to encrypt during key agreement. Undefined if Key Agreement bit is not set.
NX_SECURE_X509_KEY_USAGE_DECIPHER_ONLY	8	Used with Key Agreement bit (bit 4) – when set, certificate key can only be used to decrypt during key agreement. Undefined if Key Agreement bit is not set.

Bitmasks and values for X.509 Key Usage Extension

Parameters

certificate Pointer to certificate being verified.

bitfield Return the entire bitfield from the extension.

Return Values

NX_SUCCESS (0x00) Key usage extension found and bitfield returned.

NX_SECURE_X509_MULTIBYTE_TAG_UNSUPPORTED
(0x181) ASN.1 multi-byte tag encountered (unsupported certificate).

NX_SECURE_X509_ASN1_LENGTH_TOO_LONG
(0x182) Invaild ASN.1 field encountered (invalid certificate).

NX_SECURE_X509_INVALID_TAG_CLASS
(0x190) Invalid ASN.1 tag class encountered (invalid certificate).

NX_SECURE_X509_INVALID_EXTENSION_SEQUENCE
(0x192) Invalid extension encountered (invalid certificate).

NX_SECURE_X509_EXTENSION_NOT_FOUND
(0x19B) The Key Usage extension was not found in the provided certificate.

NX_PTR_ERROR	(0x07)	Invalid certificate or bitfield pointer.

Allowed From

Threads

Example

```
ULONG certificate_verification_callback(NX_SECURE_TLS_SESSION *session,
                                        NX_SECURE_X509_CERT* certificate)
{
UINT status;
USHORT key_usage_bitfield;

    /* Key usage - extract key usage bitfield. */
    status = nx_secure_x509_key_usage_extension_parse(certificate, &key_usage_bitfield);

   /* Check certificate for a few specific key usage bits. */
   if((key_usage_bitfield & NX_SECURE_X509_KEY_USAGE_DIGITAL_SIGNATURE) == 0 ||
      (key_usage_bitfield & NX_SECURE_X509_KEY_USAGE_NON_REPUDIATION)   == 0 ||
      (key_usage_bitfield & NX_SECURE_X509_KEY_USAGE_KEY_ENCIPHERMENT)  == 0)
    {
        printf("Expected key usage bitfield bits not set!\n");
        return(NX_SECURE_X509_KEY_USAGE_ERROR);
    }

    /* The specified bits were set, return success! */
    return(NX_SUCCESS);
}
```

See Also

nx_secure_tls_session_certificate_callback_set, nx_secure_x509_extended_key_usage_extension_parse, nx_secure_x509_crl_revocation_check, nx_secure_x509_common_name_dns_check, nx_secure_x509_extension_find

Appendix A

NetX Secure Return/Error Codes

NetX Secure TLS 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 TCP/IP error codes – TLS values begin at 0x101 and TCP/IP values are below 0x100. X.509 return values start at 0x181. Refer to the NetX TCP/IP documentation for information on TCP/IP 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.

Error Name	Value	Description
NX_SECURE_TLS_ALLOCATE_PACKET_FAILED	0x111	An internal packet allocation for a TLS message failed.
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.

Error Name	Value	Description
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.
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 recognized 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

NetX™ Secure User Guide

Publication Date: Rev.5.17 Mar 4, 2019

Published by: Renesas Electronics Corporation

NetX™ Secure User Guide

Renesas Electronics Corporation