egnite

Internals of Nut/TLS

This document will introduce you to the inner workings of Nut/TLS, the TLS 1.2 server implementation for the Nut/OS embedded operating system.

Author: Daniel Otte

Status: Rev-1.1 11/2013

Contents

1 Overview

1.1 Features

Supported cipher suites:

- TLS_RSA_WITH_AES_128_CBC_SHA256 (0x00,0x3C)
- TLS_RSA_WITH_AES_128_CBC_SHA (0x00,0x2F)
- TLS_RSA_WITH_AES_256_CBC_SHA256 (0x00,0x3D)
- TLS_RSA_WITH_AES_256_CBC_SHA (0x00,0x35)
- TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x00,0x0A)
- TLS_RSA_WITH_RC4_128_SHA (0x00,0x05)
- TLS_RSA_WITH_RC4_128_MD5 (0x00,0x04)
- TLS_RSA_WITH_NULL_SHA256 (0x00,0x3B)
- TLS_RSA_WITH_NULL_SHA (0x00,0x02)
- TLS_RSA_WITH_NULL_MD5 (0x00,0x01)

1.2 A few words about TLS

TLS means Transport Layer Security and is available in several versions, which are all available as RFCs. We are here concerned only with version 1.2 which is specified in RFC 5246. TLS is also the successor of SSL, as the new name did not propagate very well, most people still say "SSL" when they mean TLS.

TLS forms a tunnel for sensitive data over insecure channels. It neither defines what data is transported nor over which channel the secure channel is established. A common combination is transferring HTTP data via TLS over a TCP connection. This is widely known as HTTPS.

TLS is basically a record protocol operating over a reliable stream (like a TCP connection), with two specified additional protocols operating on top of it. A third protocol would be the application stream which also has a specified protocol ID.

Each record transmitted over a TLS connection may be encrypted and/or signed by the transmitting party.

The two important protocols which are part of TLS and are operating over the record connection are the handshake protocol and the alert protocol.

The handshake protocol allows establishing a secure connection using asymmetric cryptography while the alert protocol handles error and warning messages, which are also used to gracefully close the connection.

It is important to know that the standard explicitly forbids application layer communication before the first successful handshake is done.

1.3 Basic operation of Nut/TLS

Nut/OS offers you to use TCP connections like normal streams in C. Nut/TLS extends this ides to use TLS connections also like normal streams. It even allows you to operate your connection over any bidirectional stream (not only TCP connections but also serial/UART lines).

The Nut/TLS core is implemented as Nut/OS driver. It offers the standard interface of Nut/OS drivers and special functions, which are necessary for it to work.

The usage schema is quite simple:

1. Initialization of the driver
2. Opening an Nut/TLS stream
3. Assigning a stream to the Nut/TLS stream
4. Doing a handshake
5. Send and receive data
6. Close the Nut/TLS stream

1.3.1 Driver initialization

The initialization of the driver is started by the following code:

```
NutRegisterDevice(&devTLS, 0, 0);
```

This causes the creation of a dedicated thread which flushes the write buffers in fixed intervals and the allocation of memory for the resumption table.

1.3.2 Opening a Nut/TLS stream

A Nut/TLS stream is opened like a regular file with fopen(). The name must start with "tls:" to indicate that the Nut/TLS driver should be used to open this file. The prefix "tls:" is followed by an identifier which is unique for each distinct TLS connection running at the same time. The identifier may consist of any combination of the letters "A" to "Z", "a" to "z", the numerical characters "0" to "9" and the character "_". Opening an existing connection will result in a stream which is an alias to the already opened stream.

1.3.3 Assigning a stream to a Nut/TLS stream

Due to the flexibility of Nut/TLS it is necessary to explicitly assign a stream to operate on the Nut/TLS stream. This stream may be of any kind, but must be bidirectional. In most cases this will be a stream obtained from a TCP socket connection. The stream to operate on is also called the **base stream**.

1.3.4 Handshake

After the base stream is assigned, a basic record layer connection is established. This connection does neither provide confidentiality nor integrity. To secure the

connection a **handshake** has to take place. Within this handshake the cryptographic parameters are negotiated and cryptographic keys are exchanged which will protect following data.

The handshake is normally initiated by the client after establishing the record connection. To proceed the handshake the `make_handshake()` function is called. It just takes a pointer to the Nut/TLS stream as parameter and returns zero on success.

1.3.5 Send and receive data

When the handshake is done, data exchange can begin. Exchanging data is as simple as using `fread()`, `fwrite()`, `fprintf()`. All functions which can handle streams will also work with Nut/TLS streams.

1.3.6 Destruction of a Nut/TLS stream

The connection is destroyed by calling `fclose()` with a pointer to the Nut/TLS stream as parameter.

1.4 Drivers point of view

1.4.1 Reading data

When data is requested by the application, the driver first tries to fulfill the request by using already buffered data. If the drivers buffer does not contain enough data, a packet is read from the base stream, decrypted in-place and then chained into the buffer. If the request can still not be fulfilled, it reads the next packet, decrypts it and chains it into the buffer. This is repeated until the request can be fulfilled or an error occurs. If an error occurs, the data which could be read is returned to the application. The EOF flag and error indicator are set according to the reason of the error.

1.4.2 Sending data

If the data to send fits into the write buffer it is simply copied into the write buffer. If the write buffer is full or the data does not fit into the write buffer, the content of the write buffer is encrypted and send to the base stream. The remaining data is send in a packet with a size which is equal to the size of the write buffer. If there is a last block, which is smaller than the write buffer, it is copied into the write buffer.

Encrypting and sending the packet to the base stream is, opposed to the reading behavior, handled by the lower layer functions. This avoids additional copying of the data.

1.5 Applications point of view

Nut/TLS appears as filter to the application. It behaves like a normal C-style stream (like `stdin`, `stdout` and `stderr`). The stream is bidirectional and binary only. All the tools, which are able to operate on this kind of streams, are adequate for use with Nut/TLS streams (like `fread()`, `fwrite()`, `fprintf()`, `. . .`).

But before normal operation can take place, it is necessary to properly start the connection. Therefore a stream has to be assigned to the Nut/TLS stream and a handshake has to be done. Since Nut/TLS is quite flexible it is possible to run TLS connections over nearly any other bidirectional stream. A stream to operate on has to be assigned to the Nut/TLS stream. This is done by `int tls_ctl_set_stream (FILE* tls_file, FILE* base_stream)`.

1.6 How the parts fit together

The driver-style implementation allows using TLS connections like normal streams. But this abstraction covers only the TLS record level and the TLS alert protocol. The implementation of handshake itself uses this level of abstraction and can therefore not be integrated into the driver. Therefore it is necessary to initiate a handshake manually after opening the connection. Also renegotiation is not implemented for this and security reasons.

1.7 Implementation content

1.7.1 Header files

`hexdump.h`	hex dump routines for debugging purposes
`sec_memcmp.h`	secure version of `memcmp()`
`server.key.h`	declaration of TLS private key structures
`tls_alert_protocol.h`	declaration of alert functions
`tls_alert_types.h`	declaration of alert related types and values
`tls_certificate.h`	declaration of TLS certificate structures
`tls_cipher_suites.h`	declaration of TLS cipher suites related types and values
`tls_crypto.h`	declaration of cryptographic core functions
`tls_driver.h`	declaration of driver related functions and types
`tls_handshake.h`	declaration of handshake function
`tls_handshake_protocol.h`	
`tls_random.h`	declaration of PRNG related functions
`tls_record_layer.h`	declaration of record-level TLS functions
`tls_resumption.h`	declaration of functions and types related to TLS session resumption

1.7.2 Implementation files

`Hexdump.c`	hex dump routines for debugging purposes
`nuttls_test_system.cert.tls.elf`	certificate wrapped into an elf-object
`sec_memcmp.c`	secure version of `memcmp()`
`server.key.c`	structures containing the private key material
`tls_alert_protocol.c`	alert protocol related functions
`tls_cipher_suites.c`	structures and functions related to TLS cipher suites
`tls_crypto.c`	functions handling low level symmetric crypto
`tls_driver.c`	driver related functions and structures
`tls_handshake.c`	functions for the TLS handshake protocol
`tls_random.c`	PRNG wrapper around entropium[1]
`tls_record_layer_common.c`	functions shared between TLS record layer transmit and receive functions
`tls_record_layer_rx.c`	functions handling the reception of TLS record layer packets
`tls_record_layer_tx.c`	functions handling the transmission of TLS record layer packets
`tls_resumption.c`	functions for managing the resumption table
`tls_test.c`	demo application implementing a HTTPS server

2 Basic data structures

The two most important structures are `tls_connection_t` and `tls_record_connection_t`. `tls_connection_t` contains the driver related data, while `tls_record_connection_t` contains the TLS related data.

1 Entropium is part of the ARM-Crypto-Lib.

2.1 tls_connection_t

```
typedef struct _tls_connection_t tls_connection_t;

struct _tls_connection_t {
    char *name;
    tls_connection_t *next;
    tls_record_connection_t connection;
    void *write_buffer;
    size_t write_buffer_size;
    size_t write_buffer_fill;
    tls_read_buffer_block_header_t *read_buffer;
    uint8_t write_buffer_protocol;
    uint8_t closing;
    uint8_t write_lock;
};
```

2.1.1 name

The **name** field contains a pointer to a string of characters representing the name given to the connection during **fopen()**. The name is actual the part after the initial **"tls:"** prefix of the path.

2.1.2 next

The **next** pointer points to the next connection in memory, so that all connections form a linked list.

2.1.3 connection

The **connection** field is a **tls_record_connection** (not a pointer), holding the parameters of the TLS connection.

2.1.4 write_buffer

The **write_buffer** pointer points to some memory holding the write buffer.

2.1.5 write_buffer_size

write_buffer_size is the size of the buffer. The size is considered to be bytes.

2.1.6 write_buffer_fill

write_buffer_fill is the amount of actual data in the write buffer. Its unit is bytes.

2.1.7 read_buffer

read_buffer points to the first block header for the read buffer.

2.1.8 write_buffer_protocol

write_buffer_protocol is the numeric protocol ID to use for data transmission.

2.1.9 closing

closing is a flag which indicates if the connection is currently in the process of closing. If it is closing the field reads one else it reads zero.

2.1.10 write_lock

write_lock is a flag indicating if a write is in process. Write requests will be blocked while this field is non-zero.

2.2 tls_record_connection_t

```
struct _tls_record_connection_t {
 FILE *base_stream;
 uint32_t session_id;
 union __attribute__((packed)) {
 struct {
 uint8_t major;
 uint8_t minor;
 } names;
 uint16_t id16;
 } version;
 uint8_t secure;
 uint8_t post_clienthello;
 tls_record_structure_pattern_t tx_pre_pattern;
 tls_record_ciphertext_struct_t rx_pre_pattern;
 tls_sec_parameters_t sec_parameters;
 tls_record_states_t sec_states_local;
 tls_record_states_t sec_states_remote;
 tls_record_pending_state_t *pending;
 uint8_t pending_status;
 tls_error_state_t error_state;
};
```

2.2.1 base_stream

base_stream is a pointer to the base stream.

2.2.2 session_id

`session_id` is a 32-bit session ID assigned to the connection. It is automatically generated during handshake and can be used for session resumption.

2.2.3 version

The **`version`** field is a union holding the TLS protocol version used for communication. It can be used as 16-bit ID (**`id16`**) or as major (**`names.major`**) and minor (**`names.minor`**) part.

2.2.4 secure

The **`secure`** flag is used to signalize if a successful handshake has already took place. It reads zero before the initial handshake and one afterwards.

2.2.5 post_clienthello

The **`post_clienthello`** flag is used to signalize if checks for packet version should be relaxed. This is necessary since the initial **`clienthello`** may use a older record layer version for compatibility.

2.2.6 tx_pre_pattern

The **`tx_pre_pattern`** field holds precomputed offset information for sending packets. These data are computed after the crypto parameters are negotiated.

2.2.7 rx_pre_pattern

The **`rx_pre_pattern`** field holds precomputed offset information for receiving packets. These data are computed after the crypto parameters are negotiated.

2.2.8 sec_parameters

The **`sec_parameters`** field holds the negotiated crypto parameters (incl. **`master_secret`**, **`client_random`** and **`server_random`**).

2.2.9 sec_states_local

The **`sec_states_local`** field holds the contexts for cryptographic algorithms for the sending side.

2.2.10 sec_states_remote

The **`sec_states_remote`** field holds the contexts for cryptographic algorithms for the receiving side.

2.2.11 pending

pending points to a structure which contains the cryptographic parameters just negotiated but not yet active. It will be used after receiving a **change_cipher_spec message**.

2.2.12 pending_status

pending_status signalizes for which direction there are currently pending states.

2.2.13 error_state

error_state contains information about the last error which occurred.

3 Buffering

There are two buffers per connection, one for the receiving side and one for the transmitting side.

3.1 Write buffer

The write buffer implementation is quite simple. The buffer consists of a memory area allocated by **create_new_connection()** (in **tls_driver.h**). All data which should be send out normally are just copied into the buffer and the whole buffer is transmitted (by **tls_connection_send_buffer** in **tls_driver.h**) when it is full. An exception are messages which are to large for the buffer. In this case the buffer is send on the wire and then blocks of the message which all have the size of the buffer. If data remains (which will not fill a whole buffer) it is copied into the buffer.

3.1.1 Allocation

The whole allocation of the buffer happens in **create_new_connection()** (in **tls_driver.h**). The two macros **INITIAL_WRITE_BUFFER_SIZE** and **MIN_WRITE_BUFFER_SIZE** are used to configure the allocation process. The function first tries to allocate **INITIAL_WRITE_BUFFER_SIZE** bytes of memory, if that fails it successively reduces the memory request to the half of the previous request and tries again, until the requested is fulfilled or the requested size is below **MIN_WRITE_BUFFER_SIZE**. If the requested size is lower than **MIN_WRITE_BUFFER_SIZE** an error is returned.

3.2 Read buffer

Since TLS is a packet based protocol, always whole packets are received and buffered as such. The buffered packets are managed by a linked list of header blocks which contain the management information for those packets.

```
struct tls_read_buffer_block_header_st {
    uint8_t *head;
    uint8_t *buffer;
    tls_read_buffer_block_header_t *next;
    size_t remaining_bytes;
    uint8_t protocol;
};
```

4 Packet reception

The process of packet reception starts in **`read_packet_to_buffer()`** (in **`tls_driver.h`**) which is called if the read buffer could not fulfill a read request (see 1.4.1).

First it allocates three buffers:

- **`buffer`** for holding the packet content
- **`packet_info`** for holding information generated by packet decryption
- **`header`** for holding information about the memory block (for inclusion in the buffers linked list)

Then it reads the 5 byte packet header from the base stream into the just allocated buffer.

offset	size	meaning
0 (byte)	1 (byte)	sub-protocol
1 (byte)	2 (byte)	protocol version
3 (byte)	2 (byte)	remaining data bytes in packet

The standard limits the maximum amount of remaining bytes to 214 + 2048 (see RFC 5246 on page 21). This limit is checked, then optionally a compile time specified limit (the macro **`MAX_RX_PACKET_LENGTH`**). If the length checks are passed the size of the header buffer is expanded so it can hold the whole packet.

Then it reads the remaining part of the packet from the base stream into **`buffer`**.

Now the decryption routine **`tls_record_decode()`** gets called which decrypts the packet in place and returns data in an information structure (**`tls_record_packet_info_t`**).

Here the packet version is checked (for all packets except an initial **`client_hello`**) and the length is checked again and decryption is delegated to **`tls_crypto_decipher_packet()`**.

In **`tls_crypto_decipher_packet()`** the decryption work is delegated to the CBC or stream functions of the ARM-Crypto-Lib.

Back in **tls_record_decode()** the next thing is trying to compute the structure of the packet. This computation is accelerated by precomputed tables based on the negotiated parameters **(compute_pattern()**). After the structure is computed the Message Authentification Code (MAC) is checked (**check_mac()**) and then the message padding (**check_padding()**).

If any check goes wrong a proper alert message is generated and send to the base stream.

The driver, after **tls_record_decode()** returns, copies necessary information from **packet_info** into **header** and frees **packet_info**. The next step is checking the protocol, if it is set to **alert_proto(21)** and takes action depending on what kind of alert message was received (see 7 on page 13).

If the message is no fatal alert message or a **close_notify** it is chained into the linked list forming the read buffer.

5 Packet transmission

The process of packet transmission starts in **tls_connection_send_buffer()** (in **driver.h**) which is called when the write buffer is flushed (which occurs if the write buffer is full or an external flush is issued also see 1.4.2 on page 3.

tls_connection_send_buffer() itself first checks the validity of the given protocol number (it must be 20, 21, 22 or 23). After this basic check memory is allocated for the **pinfo** structure of type **tls_record_packet_info_t**. This structure is filled with the available data.

If the packet needs padding, a valid padding length is randomly selected by **random_pad_length()**.

Then a buffer for the packet is allocated. All further processing of the packet takes place in this buffer.

After the address of the buffer is fixed, the structure of the packet is computed with absolute values by **compute_pattern()**.

Now the buffer is filled with copied data and a header is prepended.

6 The handshake process

The handshake initializes the cryptographic parameters of a TLS connection. It is run at the beginning of a session and is initiated by the client. Multiple messages are exchanged of which some are optional. A normal handshake (as shown by figure 1) involves asymmetric cryptographic computations and so is very computing intense.

Client		Server
ClientHello	→	
	←	ServerHello
	←	Certificate
	←	ServerHelloDone
ClientKeyExchange	→	
[ChangeCipherSpec]	→	
Finished	→	
	←	[ChangeCipherSpec]
	←	Finished
ApplicationData	←→	ApplicationData

Figure 1: TLS handshake as done by Nut/TLS

The standard allows re-handshaking, which means doing a handshake via an already initialized session. This is not supported since it might be a security problem[2]. An alternative to do a "normal" full handshake is to do an abbreviated handshake to resume a session using common secrets that already have been computed between the two parties. This abbreviated handshake is shown in figure 2. It is very fast since only symmetric primitives are used to compute the new keys (derived from the common secret and the random values exchanged with the hello messages).

Client		Server
ClientHello	→	
	←	ServerHello
	←	[ChangeCipherSpec]
	←	Finished
[ChangeCipherSpec]	→	
Finished	→	
ApplicationData	←→	ApplicationData

Figure 2: Abbreviated TLS handshake as done by Nut/TLS

The handshake implementation (**`tls_handshake.c`**) works on top of the record layer. It is currently restricted to RSA cipher suites but can easily be replaced. The handshake handling routine **`make_handshake()`** is called directly by the application after opening the Nut/TLS stream file. **`make_handshake()`** waits for an incoming

2 See RFC 5746 for a description of the attack

client_hello message and then starts the negotiation. It returns zero on success and non-zero if an error occurred.

Since all handshake messages start with a three byte length field, a generic function is used to read handshake messages and allocate memory for them (**read_handshake_packet**).

The certificate message, which Nut/TLS sends to the client, is stored completely (with handshake header) in FLASH memory. It is created by specific tools (**cert_merge**) before compilation and is linked in during the linking stage. Since usage of the data positioned by the linker requires knowledge of the symbols referring to the binary blob, those are defined as macros (**CERT_BLOB_START**, **CERT_BLOB_END** and **CERT_BLOB_SIZE**).

The private RSA key is stored in **server.key.c** which defines multiple biginteger objects needed to do the RSA computations.

Users **must** replace the certificate and the key by self created ones where the key **should** be kept strictly secret.

7 Alert handling

Alert handling is directly done by the driver and the record layer. The main purpose of the alert protocol is to signal warnings, errors and regular termination of the communication. The last is especially important, since abnormal termination by a fatal error causes the session to be not resume-able. Handling of incoming alert messages is done directly after the packet is decrypted and before it would be stored in the read buffer. Every alert is send to the **tls_alert_receive()** function (in **alert_protocol.c**) and further checked and processed. This would be the place to handle specific errors if the application needs such handling. The return value of **tls_alert_receive()** signals, if a connection should be terminated or if it can further operate. If a fatal error or a close notify occurs, the connection will be terminated, which is handled by the function in the driver.

Sending alert messages (due to problems) happens by a call to **tls_alert_send()**. This function accepts an additional comment, which will be logged, if logging is enabled.