# Kernel TLS

## Overview

Transport Layer Security (TLS) is a Upper Layer Protocol (ULP) that runs over TCP. TLS provides end-to-end data integrity and confidentiality.

## User interface

### Creating a TLS connection

First create a new TCP socket and set the TLS ULP.

```
sock = socket(AF_INET, SOCK_STREAM, 0);
setsockopt(sock, SOL_TCP, TCP_ULP, "tls", sizeof("tls"));
```

Setting the TLS ULP allows us to set/get TLS socket options. Currently only the symmetric encryption is handled in the kernel. After the TLS handshake is complete, we have all the parameters required to move the data-path to the kernel. There is a separate socket option for moving the transmit and the receive into the kernel.

```
/* From linux/tls.h */
struct tls_crypto_info {
        unsigned short version;
        unsigned short cipher_type;
};

struct tls12_crypto_info_aes_gcm_128 {
        struct tls_crypto_info info;
        unsigned char iv[TLS_CIPHER_AES_GCM_128_IV_SIZE];
        unsigned char key[TLS_CIPHER_AES_GCM_128_KEY_SIZE];
        unsigned char salt[TLS_CIPHER_AES_GCM_128_SALT_SIZE];
        unsigned char rec_seq[TLS_CIPHER_AES_GCM_128_REC_SEQ_SIZE];
};


struct tls12_crypto_info_aes_gcm_128 crypto_info;

crypto_info.info.version = TLS_1_2_VERSION;
crypto_info.info.cipher_type = TLS_CIPHER_AES_GCM_128;
memcpy(crypto_info.iv, iv_write, TLS_CIPHER_AES_GCM_128_IV_SIZE);
memcpy(crypto_info.rec_seq, seq_number_write,
                                    TLS_CIPHER_AES_GCM_128_REC_SEQ_SIZE);
memcpy(crypto_info.key, cipher_key_write, TLS_CIPHER_AES_GCM_128_KEY_SIZE);
memcpy(crypto_info.salt, implicit_iv_write, TLS_CIPHER_AES_GCM_128_SALT_SIZE);

setsockopt(sock, SOL_TLS, TLS_TX, &crypto_info, sizeof(crypto_info));
```

Transmit and receive are set separately, but the setup is the same, using either TLS_TX or TLS_RX.

### Sending TLS application data

After setting the TLS_TX socket option all application data sent over this socket is encrypted using TLS and the parameters provided in the socket option. For example, we can send an encrypted hello world record as follows:

```
const char *msg = "hello world\n";
send(sock, msg, strlen(msg));
```

send() data is directly encrypted from the userspace buffer provided to the encrypted kernel send buffer if possible.

The sendfile system call will send the file's data over TLS records of maximum length (2^14).

```
file = open(filename, O_RDONLY);
fstat(file, &stat);
sendfile(sock, file, &offset, stat.st_size);
```

TLS records are created and sent after each send() call, unless MSG_MORE is passed. MSG_MORE will delay creation of a record until MSG_MORE is not passed, or the maximum record size is reached.

The kernel will need to allocate a buffer for the encrypted data. This buffer is allocated at the time send() is called, such that either the entire send() call will return -ENOMEM (or block waiting for memory), or the encryption will always succeed. If send() returns -ENOMEM and some data was left on the socket buffer from a previous call using MSG_MORE, the MSG_MORE data is left on the socket buffer.

## Receiving TLS application data

After setting the TLS_RX socket option, all recv family socket calls are decrypted using TLS parameters provided. A full TLS record must be received before decryption can happen.

```
char buffer[16384];
recv(sock, buffer, 16384);
```

Received data is decrypted directly in to the user buffer if it is large enough, and no additional allocations occur. If the userspace buffer is too small, data is decrypted in the kernel and copied to userspace.

`EINVAL` is returned if the TLS version in the received message does not match the version passed in setsockopt.

`EMSGSIZE` is returned if the received message is too big.

`EBADMSG` is returned if decryption failed for any other reason.

## Send TLS control messages

Other than application data, TLS has control messages such as alert messages (record type 21) and handshake messages (record type 22), etc. These messages can be sent over the socket by providing the TLS record type via a CMSG. For example the following function sends @data of @length bytes using a record of type @record_type.

```
/* send TLS control message using record_type */
static int klts_send_ctrl_message(int sock, unsigned char record_type,
                                  void *data, size_t length)
{
      struct msghdr msg = {0};
      int cmsg_len = sizeof(record_type);
      struct cmsghdr *cmsg;
      char buf[CMSG_SPACE(cmsg_len)];
      struct iovec msg_iov;   /* Vector of data to send/receive into.  */

      msg.msg_control = buf;
      msg.msg_controllen = sizeof(buf);
      cmsg = CMSG_FIRSTHDR(&msg);
      cmsg->cmsg_level = SOL_TLS;
      cmsg->cmsg_type = TLS_SET_RECORD_TYPE;
      cmsg->cmsg_len = CMSG_LEN(cmsg_len);
      *CMSG_DATA(cmsg) = record_type;
      msg.msg_controllen = cmsg->cmsg_len;

      msg_iov.iov_base = data;
      msg_iov.iov_len = length;
      msg.msg_iov = &msg_iov;
      msg.msg_iovlen = 1;

      return sendmsg(sock, &msg, 0);
}
```

Control message data should be provided unencrypted, and will be encrypted by the kernel.

## Receiving TLS control messages

TLS control messages are passed in the userspace buffer, with message type passed via cmsg. If no cmsg buffer is provided, an error is returned if a control message is received. Data messages may be received without a cmsg buffer set.

```
char buffer[16384];
char cmsg[CMSG_SPACE(sizeof(unsigned char))];
struct msghdr msg = {0};
msg.msg_control = cmsg;
msg.msg_controllen = sizeof(cmsg);

struct iovec msg_iov;
msg_iov.iov_base = buffer;
msg_iov.iov_len = 16384;

msg.msg_iov = &msg_iov;
msg.msg_iovlen = 1;

int ret = recvmsg(sock, &msg, 0 /* flags */);

struct cmsghdr *cmsg = CMSG_FIRSTHDR(&msg);
if (cmsg->cmsg_level == SOL_TLS &&
    cmsg->cmsg_type == TLS_GET_RECORD_TYPE) {
    int record_type = *((unsigned char *)CMSG_DATA(cmsg));
    // Do something with record_type, and control message data in
    // buffer.
    //
```

```
    // Note that record_type may be == to application data (23).
} else {
    // Buffer contains application data.
}
```

recv will never return data from mixed types of TLS records.

### Integrating in to userspace TLS library

At a high level, the kernel TLS ULP is a replacement for the record layer of a userspace TLS library.

A patchset to OpenSSL to use ktls as the record layer is here.

An example of calling send directly after a handshake using gnutls. Since it doesn't implement a full record layer, control messages are not supported.

## Statistics

TLS implementation exposes the following per-namespace statistics (`/proc/net/tls_stat`):

- `TlsCurrTxSw`, `TlsCurrRxSw` - number of TX and RX sessions currently installed where host handles cryptography
- `TlsCurrTxDevice`, `TlsCurrRxDevice` - number of TX and RX sessions currently installed where NIC handles cryptography
- `TlsTxSw`, `TlsRxSw` - number of TX and RX sessions opened with host cryptography
- `TlsTxDevice`, `TlsRxDevice` - number of TX and RX sessions opened with NIC cryptography
- `TlsDecryptError` - record decryption failed (e.g. due to incorrect authentication tag)
- `TlsDeviceRxResync` - number of RX resyncs sent to NICs handling cryptography