Computer Science

Reza Mohammadi and Martin Jarl

Implementation analysis of openSSL over SCTP

Bachelor's Project

June 2004:14

Implementation analysis of openSSL over SCTP

Reza Mohammadi and Martin Jarl

This report is submitted in partial fulfillment of the requirements for the Bachelor’s degree in Computer Science. All material in this report which is not our own work has been identified and no material is included for which a degree has previously been conferred.

Reza Mohammadi and Martin Jarl

Approved, 2004-06-03

Advisor: Johan Garcia

Examiner: Stefan Lindskog

Abstract

TietoEnator in Karlstad develops a protocol stack based on SS7 (Signaling System Nr 7). There is a desire to increase the security provided by SS7 when it is used in an IP network (Internet Protocol network). A possible solution is to use TLS (Transport Layer Security).

There is an existing implementation of TLS called openSSL. TietoEnator has decided to use openSSL in order to test how TLS functions with SS7. The openSSL code is designed to run on top of the transport layer protocol TCP (Transmission Control Protocol). However, SS7 uses SCTP (Stream Control Transmission Protocol) at the transport layer. To use openSSL with SS7 openSSL must be adapted to SCTP.

This document analyses parts of the existing SS7 environment, including SCTP, and the openSSL code. The openSSL code analysis concentrates on the parts of openSSL that communicate with TCP, because these are the parts that need to be adapted to SCTP. The analysis shows that there are at least three different design approaches to the adaptation of openSSL to SCTP. One approach involves translation of each TCP call into a SCTP call. However, this is not possible because of the differences between TCP and SCTP. Another approach involves creation of a translating software module between the TCP calls and SCTP. This demands too much time for a test implementation but is suitable for the production version. The last approach involves rewriting the parts of openSSL that communicate with TCP. This requires a lot of openSSL modifications but is adequate for a test implementation. An initial implementation according to the third approach is made as a part of this work.

Contents

List of Figures

List of tables

1 Introduction

TietoEnator in Karlstad develops a protocol stack, based on SS7 (Signaling System Nr 7), used in traditional telephony networks.

A telephony network consists of two separate nets. One net is used for transmission of speech and data. The other net is used for signaling. The signaling net delivers control information used to setup, supervise and terminate telecommunication connections in the transmission net.

SS7 is divided into two functional parts. One part is responsible for packing and unpacking signaling messages. The other part is responsible for the transfer of messages. More information about SS7 can be found in [10] and [17].

In today's telecommunication world attempts are made to integrate the TDM-based (Time Division Multiplexing) SS7 with data communication networks. By using appropriate protocols in the data communication network, the integration process can be made easier. It is advantageous to use IP (Internet Protocol [12]) at the network layer, a specific transport layer protocol called SCTP (Stream Control Transmission Protocol [21]) and a few adaptation protocols. The idea is to send the SS7 signaling traffic over the data communication network.

TietoEnator strives to increase the security provided for data transmission in SS7 when used in the data communication network environment. One possible way to improve security is to use a protocol called TLS (Transport Layer Security). The problem with this protocol is that it reduces performance. TietoEnator wishes to implement a test version of the TLS protocol to see if the loss of performance is acceptable.

There is an existing implementation of TLS called openSSL [11]. Instead of implementing TLS from scratch this implementation can be used. The problem with openSSL is that it is designed to run on top of TCP (Transmission Control Protocol [13]). In SS7 the correspondent of TCP is SCTP (Stream Control Transmission Protocol). The parts of openSSL that communicates with TCP needs to be adapted to SCTP. This document describes how the adaptation of openSSL to SCTP can be done.

The work described in this document is limited to the Unix openSSL implementation, even though the openSSL code is designed to run on other platforms as well. The design and to some extent the analysis of openSSL, described in this document, are concentrated on the initialization of a connection and on data transmission. The encryption functionality is not described.

To understand how the adaptation of openSSL over SCTP can be done, a description of the individual protocols TLS and SCTP must be given. A description of network security is needed to understand TLS. Chapter 2 describes some security issues and some methods that provide security. The environment to which openSSL is to be adapted is described in chapter 3. This environment consists not only of SCTP but also of CP (Common Parts). CP is a software module used in TietoEnator's SS7 environment to encapsulate platform dependencies. Chapter 4 contains a presentation of the TLS protocol and an analysis of its implementation, openSSL. Some conceivable design approaches and the design selected for the work are described in chapter 5. Problems encountered during this work are described in chapter 6. Finally, conclusions that the authors of this document have come to during this work are described in chapter 7.

2 Network security

This chapter is a short introduction to network security. The presentation here is based on [5], [8] and [19], where further information can be found.

When sending messages over a communication link, several security issues arise. It is common to deal with the following security aspects:

- Confidentiality: The sender wants to keep messages unreadable for everyone except the intended receiver.
- Message Integrity: The receiver wants to be able to detect if a message has been changed during transmission.
- Authentication: The sender and the receiver want to be able to verify each other's identity.

To achieve confidentiality the sender and the receiver must agree on some method (typically secret-key encryption) to transform messages before and after sending. Confidentiality is discussed in section 2.1. MAC (Message Authentication Code) calculations, commonly used to provide message integrity, are described in section 2.2. Authentication can be provided with a method called digital signing, described in section 2.3. The consequences of implementing security in different layers in the protocol stack are investigated in section 2.4.

2.1 Confidentiality – encryption

Encryption is the process of substituting and/or rearranging elements in a message before sending.

Substitution means replacing each element (i.e. bit or group of bits) of a message with another different element. So, elements are mapped into new elements. Rearranging means changing the order of elements in a message. The substitution and rearrangement must be done in a manner so that the receiver can reverse it. The process of reversing the substitution and rearrangement is called decryption.

To make decryption reversible the receiver must use the same algorithm for substitution and rearrangement as the sender, but reversed. If someone else knows this algorithm he or she can possibly decrypt the message. To prevent this, the sender needs to be able to make the algorithm produce different outputs depending on some input parameter. If this input parameter is not known the message will be hard to decrypt. Therefore, only the sender and the intended receiver should know this input parameter. The input parameter is called a key.

The origin message (i.e. the message before the encryption) is called plaintext. The encrypted message is called ciphertext. When the message has been successfully decrypted it is the same as the origin message and called plaintext again.

2.1.1 Secret-key encryption

Secret-key encryption is a method used to prevent anyone but the sender and the intended receiver from understanding the messages sent over a communication link.

Secret-key encryption is also called conventional encryption, symmetric encryption or single-key encryption. In secret-key encryption the sender and receiver use the same key as input to the encryption/decryption algorithm. If the key is kept secret it will be hard for anyone else to decrypt the sent messages.

The encryption algorithm takes the plaintext and the secret key as input and produces a ciphertext. The ciphertext is then transmitted over a data communication link. When the ciphertext reaches the receiver it is decrypted using the receiver's copy of the secret key. This process is illustrated in Figure 2.1.

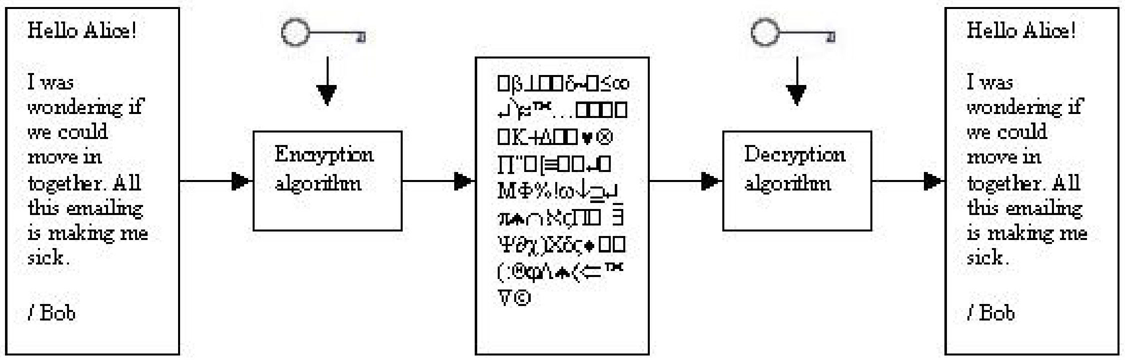
Hello Alice!
I was wondering if we could move in together. All this emailing is making me sick.
/ Bob
Encryption algorithm
Decryption algorithm
Hello Alice!
I was wondering if we could move in together. All this emailing is making me sick.
/ Bob

Figure 2.1: Encryption decryption process

Examples of well-known secret-key encryption algorithms are DES (Data Encryption Standard [18]), AES (Advanced Encryption Standard [18]), RC2 (Rivest's Cipher 2 [19]) and RC4 (Rivest's Cipher 4 [19]).

2.1.2 Public-key encryption

Public-key encryption is often used when exchanging the secret keys, which are used in secret-key encryption. Distribution of secret keys using public-key encryption is described in section 2.1.2.2. Another use of public-key encryption, called digital signing, helps the receiving end-point (i.e. computers connected to a network) to verify the identity of the sender. Digital signing is described in section 2.3.

Public-key encryption is also called asymmetric encryption. Public-key encryption uses two keys, one public and one private. A message encrypted with a certain public key can (hopefully) only be decrypted with its matching private key. Consider two people, Alice and Bob, who want to communicate. Both Alice and Bob generate their own public and private keys. The private key is, as implied by the name, kept private. The public key is either distributed to the other person or kept in a public register from which it can be fetched.

If Alice wants to send a message to Bob, she uses Bob's public key to encrypt the message and then sends it to Bob. When Bob receives the message he uses his private key to decrypt it.

2.1.2.1 Key management and certificates

There is one problem with public-key encryption. Consider Alice and Bob exchanging their public keys. How can Alice be sure that the received public key belongs to Bob? Someone else, let's say Trudy, could pretend to be Bob and send Alice her public key. Later when Alice wants to send a message to Bob she uses Trudy's public key (believing it is Bob's). Now, Trudy can read this message.

The solution to this problem is called certificates, which is only briefly mentioned in this document. Certificates consist of a public key and a user identification number. Instead of exchanging public keys Alice and Bob exchange certificates. A trusted Certificate Authority usually signs the certificate. More information about certificates can be found in [8].

2.1.2.2 Distribution of secret keys using public-key encryption

One problem with secret-key encryption is the distribution of the secret keys. How can Alice and Bob select the same secret key? Alice could select the secret key and send it to Bob, but how? She does not want to send it in plaintext and she cannot encrypt it as long as they have not agreed on the secret key.

Public-key encryption can solve this problem. Alice can send the secret key to Bob using public-key encryption. The following example explains how:

1. Alice encrypts the entire message using secret-key encryption and a secret key.

2. Alice encrypts the secret key using Bob's public key.
3. Alice attaches the encrypted secret key to the message and sends it to Bob.
4. Bob decrypts the secret key using his private key.
5. Bob decrypts the message using the secret key.
6. Alice and Bob can now communicate using secret-key encryption and the secret key.

2.1.3 Where to put encryption

Encryption can be used in different protocol layers (i.e. link layer, network layer, transport layer or application layer). The kind of security achieved depends on the choice of layers that use encryption.

If encryption is incorporated in the link layer, it is referred to as link encryption. Link encryption means that each node (e.g. router, switch) will encrypt frames (i.e. link layer messages) before sending them to the next node. This provides high security, but demands a lot of work, since each frame must be both decrypted and encrypted at each node.

The transport layer or the application layer can also handle the encryption of messages. This placement of encryption is called end-to-end encryption. The problem with end-to-end encryption is that a message header cannot be encrypted, because the header contains the destination address. If the header was to be encrypted, the lower layers would not be able to route the packet through the network. Since the header cannot be encrypted, end-to-end encryption secures the message data, but not the traffic pattern.

When both link encryption and end-to-end encryption are used the security provided is higher than with just one of them. End-to-end encryption secures the message data and link encryption secures the traffic pattern. The packet will be secured during the entire trip, except while it is in the memory of a switch or router where the header (traffic pattern) is not secured.

2.2 Message integrity – MAC

MAC (Message Authentication Code) calculations is a common way to provide message integrity. If Alice wants to send a message to Bob she calculates a MAC using the message itself together with the secret key as input to a hashing function. She then sends the MAC along with the message. When Bob receives the message he calculates his own MAC using the message and his copy of the secret key. If the MAC he calculated matches the one Alice sent along with the message he assumes that the message has not been altered during transmission.

If an intruder, let’s say Trudy, alters the message during transmission she would probably not be able to create the correct MAC for the altered message, since she does not have the secret key.

Examples of well-known MAC calculation algorithms are MD5 (Message Digest Algorithm 5 [9]) and SHA-1 (Secure Hash Algorithm [9]).

2.3 Authentication – digital signing

Digital signing uses public-key encryption to verify the identity of the sender. The sender, let’s say Alice, encrypts the message using her own private key. The receiver, Bob, can then verify that the message comes from Alice by decrypting it with Alice’s public key.

Digital signing does not provide the same kind of security as normal public-key encryption or secret-key encryption. A signed message can be read by anyone who knows the sender’s public key. Assuming that the origin of the public keys can be trusted, digital signing simply lets the receiver verify the sender of the message.

2.4 Where to put security

Network security must be incorporated in a protocol. Some possible placements of this protocol are in the application layer (e.g. Kerberos [24], S/MIME [14][15], PGP [6], SET [7]), the transport layer (e.g. TLS [1]) and the network layer (e.g. IPSec [23]). In this section transport layer security and application layer security will be discussed. Note the difference between this section and section 2.1.3 (Where to put encryption). This section discusses the entire security concept (i.e. confidentiality/encryption, message integrity and authentication) whereas section 2.1.3 discusses encryption only.

Transport layer security means that security is incorporated in the transport layer protocol. Security can also be placed in a new protocol (e.g. TLS) on top of the transport protocol as a part of the transport protocol suite, as shown in Figure 2.2. Since the new protocol is a part of the transport protocol suite, it is transparent to the application protocol. This means that the application protocol does not need to make any adjustments to the new protocol.

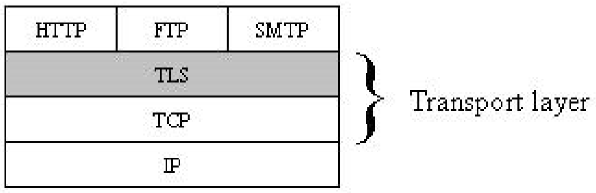

Figure 2.2: Transport layer security stack, from [19]

In application layer security, the security is incorporated in an application layer protocol, as illustrated in Figure 2.3. The advantage with this solution is that the security can be adapted to the needs of a specific application.

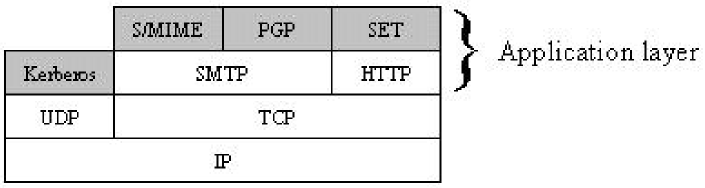

Figure 2.3: Application layer security stack, from [19]

The placement of security, interesting for this work, is the transport layer in a telephony signaling network environment.

2.5 Chapter summary

This chapter describes secret-key encryption that is used to provide confidentiality. Public-key encryption is a method used to distribute the secret keys used for secret-key encryption.

Encryption can be used at different layers in the protocol stack. To secure both data and traffic pattern it is favorable to use encryption in more than one layer. Encryption at the transport layer will secure the data while encryption at the network layer will secure the traffic pattern.

MAC calculations are commonly used to provide message integrity. Digital signing, which uses public-key encryption, lets the sender and the receiver authenticate each other.

The methods described in this chapter (secret-key encryption, public-key encryption, MAC calculations and digital signing) are used by the TLS protocol to provide basic network security. TLS and openSSL (i.e. a TLS implementation) are described in chapter 4.

3 Analysis of existing environment

This chapter describes the SS7 environment developed by TietoEnator into which openSSL is to be incorporated.

TietoEnator uses a transport layer protocol called SCTP (Stream Control Transmission Protocol). The general presentation of SCTP in section 3.1 is based on [22]. A more detailed specification of SCTP can be found at [21].

In TietoEnator's SS7 implementation, communication with SCTP cannot be done directly. It must be done via another software module called CP (Common Parts). CP encapsulates platform dependencies, which helps to provide platform independent user programs (e.g. openSSL). An overview of the CP/SCTP implementation at TietoEnator, based on [4] and [16], is given in section 3.2.

3.1 Stream Control Transmission Protocol (SCTP) – general information

SCTP is a transport layer protocol that is used in TietoEnator's SS7 implementation.

SCTP is capable of delivering messages in strict order or with no respect to order (within streams). Streams and other SCTP concepts are described in section 3.1.1.

SCTP allows the user to specify one or more destination addresses for the same end-point (multi-homing). The different destination addresses may have different network paths leading to them. If there is a failure on the path leading to one destination address, SCTP can redirect messages to another destination address (path selection). Multi-homing, path-selection and some other SCTP properties are described in section 3.1.2.

The SCTP initialization, data transmission and shutdown procedures are described in section 3.1.3.

3.1.1 Concepts

The concepts association, stream and the SCTP packet format are explained in this section.

3.1.1.1 Association and stream

An association is a connection between two SCTP end-points. In one association there can be many streams. A stream transfers the actual data and control messages between the end-points.

Within a specific stream messages are usually delivered in strict order. However, the SCTP user can choose to bypass this service to provide a delivery where the order of messages is not important. In this case SCTP will always deliver a message to the higher layer directly after it is received, even if an earlier message (with lower sequence number) is yet not received.

Different streams, however, always operate independent of each other. So, message loss in one stream does not affect any other streams.

3.1.1.2 SCTP packet format

The SCTP packet consists of a header and some chunks, as illustrated in Figure 3.1. The chunks can be of different types. There are data chunks and control chunks. The data chunks carry the messages from the upper layer, whereas the control chunks contain control information used by SCTP.

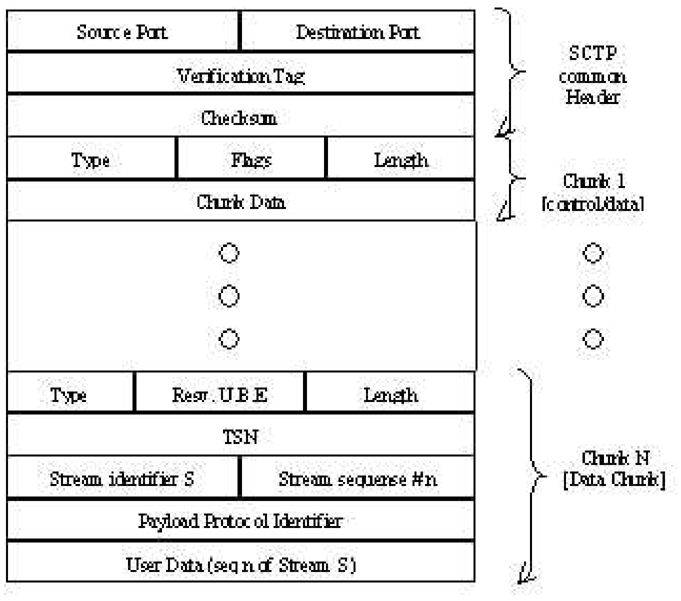

Figure 3.1: SCTP packet format, from [22]

3.1.2 Properties

This section describes the SCTP properties multi-homing, path selection and congestion control.

At the initialization the SCTP end-points exchange a list with IP destination addresses. Each IP-address may represent an alternative path through the network. One of the IP-addresses represents the primary path that messages will take in the first place. The other IP-addresses represent alternative paths that can be used if failure occurs at the primary path. Path selection is only of interest if an end-point is multi-homed (i.e. can be reached via more than one IP-address).

To test if there is failure on a path, an end-point occasionally sends a heartbeat chunk. When the other end-point receives a heartbeat chunk it replies with a heartbeat acknowledgement. If the first end-point does not receive a heartbeat acknowledgement within a certain time, it assumes there is failure on the path.

Congestion control is managed for the entire association, which has a congestion window that limits the number of bytes that may be sent without an acknowledgement.

3.1.3 Initialization, data transmission and shutdown

During the initialization some information used by the association is stored in a cookie. The cookie is digitally signed with a MAC.

SCTP uses a four-way handshake illustrated in Figure 3.2. The handshake starts when the client sends an INIT chunk, which the server acknowledges with an INIT ACK. The client then sends a COOKIE ECHO, which is acknowledged with a COOKIE ACK. The four-way handshake defends against denial of service attacks (i.e. attacks that prevent legitimate users from using some sort of service). The cookie mechanism shields from blind attacks (i.e. an attack where the attacker sends messages to an end-point but is not able to receive any responses).

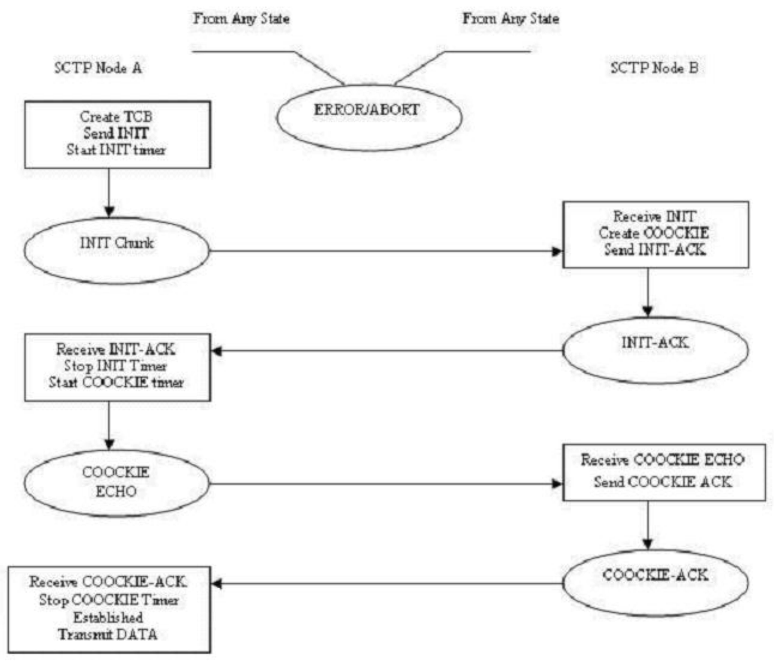

Figure 3.2: SCTP initialization, from [22]

During transmission, data chunks and control chunks may be sent in the same packet. When an end-point receives a data chunk it acknowledges the receipt with a SACK chunk, as shown in Figure 3.3. The HEARTBEAT chunk is, as described in section 3.1.2, used to test if there is any failure on the path.

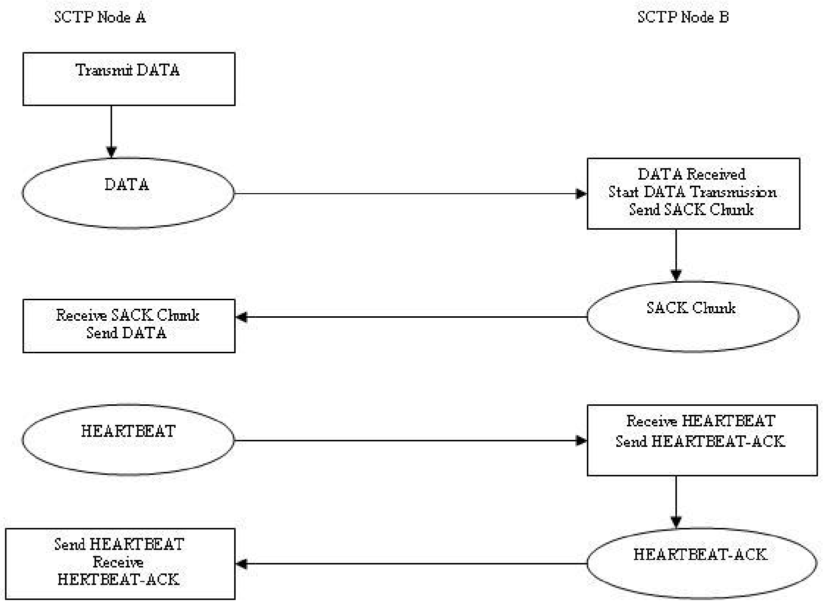

Figure 3.3: SCTP data transmission, from [22]

The termination is done with three messages, SHUTDOWN, SHUTDOWN ACK and SHUTDOWN COMPLETE, as shown in Figure 3.4.

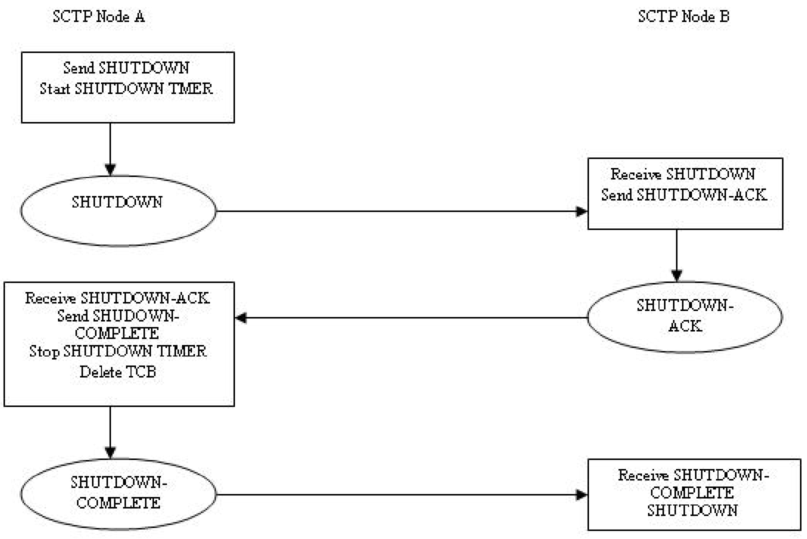
SCTP Node A
SCTP Node B
Send SHUTDOWN Start SHUTDOWN TMER
SHUTDOWN
Receive SHUTDOWN Send SHUTDOWN-ACK
SHUTDOWN-ACK
Receive SHUTDOWN-ACK Send SHUDOWN-COMPLETE Stop SHUTDOWN TIMER Delete TCB
SHUTDOWN-COMPLETE
Receive SHUTDOWN-COMPLETE SHUTDOWN

Figure 3.4: SCTP shutdown, from [22]

3.2 Common Parts (CP) and SCTP – in TietoEnator's SS7 environment

This section describes TietoEnator's SS7 environment to which openSSL is to be adapted. This environment consists partly of TietoEnator's version of SCTP. However, it is not possible or desirable for openSSL to communicate directly with SCTP. All communication must pass through CP (Common Parts). The reason is to achieve platform independency. So, openSSL must communicate with SCTP via CP.

A short description of the functionality of Common Parts is given in section 3.2.1, followed by a more detailed description of the CP/SCTP API functions in section 3.2.2. The CP/SCTP API functions will be used by openSSL.

3.2.1 Common Parts

The main task of CP (Common Parts) is to enable SS7 to run on the different platforms used at TietoEnator. CP isolates platform dependencies into one software module, by providing timer handling, memory handling, communication facilities, log/trace possibilities, list handling and interrupt handling.

Table 3.1 gives a short description of the functionality and services provided by CP.

Functionality/Service	**Description**
Common Data Types and Simple Operations	Provides type definitions for various integer types and some macros.
Timer Management	Functionality for creating, changing and canceling timers. CP notifies the requesting user when a timer expires.
Extended Memory Handling	Provides independent memory management.
Communication	Provides platform independent communication facilities.
System Log and Trace	Provides a uniform way for logging errors and other events.
Extended List Handling	Provides functions for list operations (e.g. add an element).
Interrupt Handling	Lets users specify the actions taken at different UNIX signals (only for the Unix platform).
Common Parts Configuration	There is a configuration file that contains some CP settings. This block reads the information in this file.
Common Parts Management	This block manages the communication between the CP user and the CP manager.
Internal Time	This block manages internal time of the systems.

Table 3.1: CP Functionality/Services, based on [4]

3.2.2 CP/SCTP API functions

This section describes the CP and SCTP interface towards the upper layer (e.g. applications or openSSL). API functions needed to set up a connection, to transfer data and to close a connection are described briefly in Table 3.2. Other functions, not needed for this work, are not described.

Function	**Description**	**Used by**
EINSS7CpRegisterMPOwner	This function is used to register the message port owner. The message port owner is simply the ID of the CP user.	client, server

EINSS7CpRegisterRemoteCPMgmt	This function sets the address of the process that uses CP. This address consists of an IP address and a port number. This function must also be provided with the CP manager ID. The CP manager handles the communication between the CP user and CP.	client, server
EINSS7CpMsgInitiate	This function initializes communication facilities and configures CP. It must be called before any of the other Msg-functions.	client, server
EINSS7CpMsgPortOpen	This function is used to set the reusability of message buffers. A message buffer is the memory location where a message is stored before sending. In a reusable message buffer it is possible to read the content of a message after the message has been sent. Most important, when this function has been called it is possible to set up a connection to another user.	client, server
EINSS7_00SctpRegFunc	This function is used to specify the functions that should be called at different events.	client, server
EINSS7CpMsgConnInst	This function sets up (or reestablishes) a connection between two users. It enables these users to start sending messages to each other. Note that this is not the function used by a client to connect to a server. For this purpose 'EINSS7_00SctpAssociateReq' is used. However, both the server and the client need to call this function.	client, server
EINSS7_00SctpBindReq	This function is used to register a new SCTP user.	client, server
MsgRecvEvent	This function is used to receive messages and other events like errors and alerts. Note that a CP message contains a SCTP function call. 'EINSS7_00SctpHandleInd' should be used to interpret the CP message.	client, server
MsgSend	This function is used to send messages.	client, server
EINSS7_00SctpHandleInd	This function is called by the SCTP user to interpret the messages received by 'MsgRecvEvent'.	client, server

EINSS7_00SctpInitializeReq	This function is used to register a set of IP-addresses and a port for the local end-point. The set of IP-addresses is used for the SCTP multi-homing functionality (see section 3.1.2 for more details).	client, server
EINSS7_00SctpAssociateReq	This function is used by the SCTP client to initiate an association to another end-point. The other end-point is specified by an IP-address and a port number.	client
EINSS7_00SctpSetPrimaryReq	This function allows the SCTP user to specify a primary destination IP-address (see section 3.1.2 for more details).	client, server
EINSS7_UnbindReq	This function deregisters an SCTP user.	client, server
EINSS7CpRelInst	This function closes the connection set up by 'EINSS7CpMsgConnInst'.	client, server
MsgClose	This function disables the communication facilities for a specific CP user.	client, server
MsgExit	This function disables communication facilities for all CP users.	client, server

Table 3.2: CP/SCTP API functions, based on [4] and [16]

Figure 3.5 shows how client and server use the CP/SCTP API. The first six functions, called by both client and server, initialize CP. Next both client and server register as SCTP users with 'EINSS7_00SctpBindReq'. 'EINSS7_00SctpInitializeReq' registers the IP-addresses and port numbers for the local end-points. All request functions (ending with 'Req') are followed by an 'MsgRecvEvent' and an 'EINSS7_00SctpHandleInd', which are used to confirm that the request function succeeded. The client calls 'EINSS7_00SctpAssociateReq' to set up an association with the server. The server uses 'MsgRecvEvent' to listen for the association initiated by the client. The 'EINSS7_00SctpHandleInd', called next, is used to send an association confirmation to the client. The second pair of 'MsgRecvEvent' and 'EINSS7_00SctpHandleInd' that follows the 'EINSS7_00SctpAssociateReq', called by the server, is used to receive the association confirmation from the server. At this stage, the client and the server are able to exchange messages. 'EINSS7_UnbindReq', 'EINSS7CpRelInst', 'MsgClose' and 'MsgExit' are used to close connections and free resources.

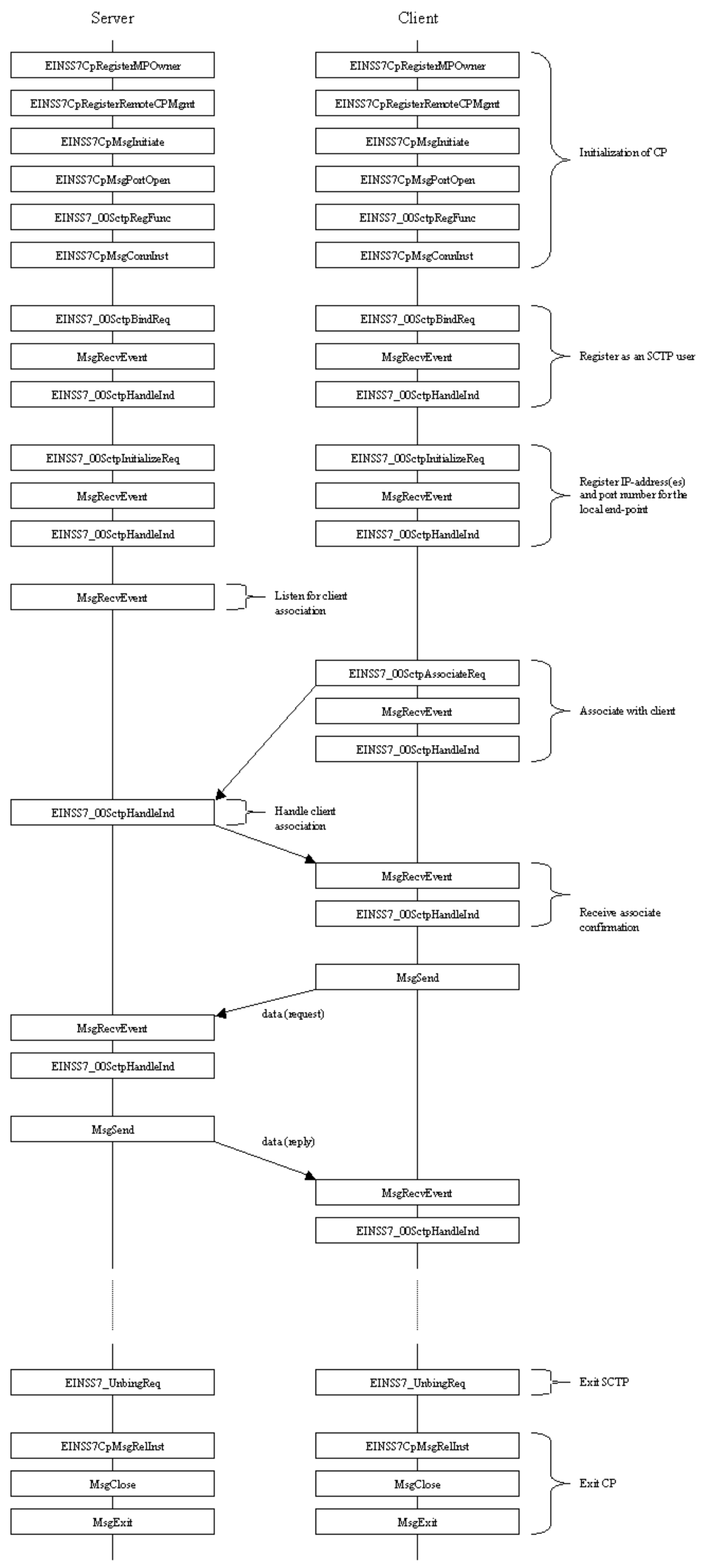

Figure 3.5: Client and server CP/SCTP API usage

3.3 Chapter summary

This chapter starts with a general description of SCTP, which is the transport layer protocol used in TietoEnator's SS7 environment. SCTP provides multi-homing, which gives the user an opportunity to select multiple destination IP-addresses that messages can be delivered to. The user can prioritize these IP-addresses by arranging them in a specific order. SCTP will deliver messages to the first IP-address in the first place, the second IP-address if there is failure on the path to the first one and so on. This is called path selection. SCTP also provides congestion control. SCTP uses a four-way handshake that together with a cookie mechanism shields from denial of service attacks and blind attacks.

In TietoEnator's SS7 implementation communication with SCTP cannot be done directly. Communication with SCTP must pass through CP (Common Parts). The main task of CP is to provide platform independency, which enables SCTP user applications to run on different platforms. This chapter gives a short description of the CP/SCTP API functions that will be used by openSSL.

Chapter 4 will describe the environment that is to be adapted to the existing environment described in this chapter.

4 Analysis of added environment

This chapter describes TLS and openSSL, a TLS implementation, which is to be adapted to the TietoEnator SS7 environment.

TLS is an attempt to provide confidentiality, message integrity and to let end-points authenticate each other. TLS uses the encryption, MAC calculations and message authentication, discussed in chapter 2, to provide this security. A general description of the TLS implementation is given in section 4.1. This description is based on [1] and [19].

Originally openSSL was designed to run on top of TCP. It communicates with TCP via the socket API. Therefore a short description of the socket API, based on [1], will be given in section 4.2. Thereafter an analysis of the openSSL code, based on [11], is given in section 4.3. More information about the socket API can be found in [20].

4.1 Transport Layer Security (TLS) – general information

TLS is a protocol that provides confidentiality (i.e. only sender and receiver can read the messages sent), message integrity (i.e. receiver can detect if a message has been altered) and authentication (i.e. sender and receiver can verify identity of each other). TLS is used at the transport layer, on top of some transport layer protocol (e.g. TCP or SCTP). TLS becomes a part of the transport layer suite, which gives the application layer the freedom to ignore the presence of TLS.

4.1.1 Architecture

TLS consists of a basic layer called the TLS record protocol and three higher layers. The higher layers are the handshake protocol, the change cipher protocol and the alert protocol. The TLS architecture is shown in Figure 4.1.

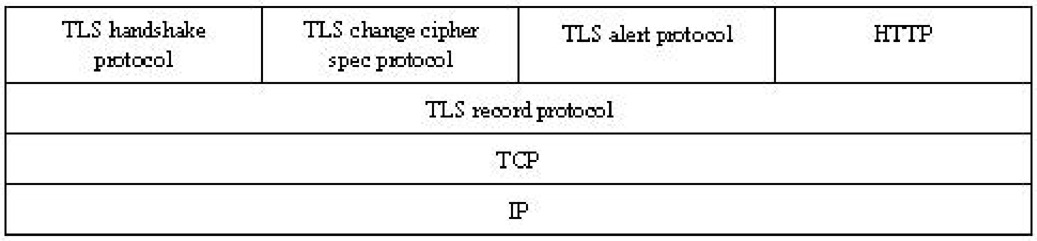

Figure 4.1: TLS architecture, from [19]

There are two important TLS concepts, session- and connection state, which need describing before presenting the different sub protocols of TLS.

A session is an association between a client and a server. The handshake protocol is used to negotiate the parameters in the session state. The session state can be shared among many connections.

The session state contains the end-points certificates. It specifies the compression method, the cipher specification (i.e. algorithms for encryption and MAC calculations). The session state also contains the master secret (i.e. a combination of symbols), which is used to generate the encryption and MAC keys. A more detailed specification of the session state is given in appendix B.1.

A connection is the working environment of the TLS record protocol. The connection state contains the keys needed for MAC calculations and encryption of messages. These keys are generated from the master secret in the session state. The connection state also maintains sequence numbers for sent and received messages. More detailed information about the parameters in the connection state can be found in appendix B.2.

Each end-point maintains one connection state for reading and one for writing messages. For both reading and writing each end-point maintains a current and a pending state. The current state is the one currently used whereas the pending state is the state that will be used when the security parameters are renegotiated.

4.1.2 TLS record protocol

The TLS record protocol provides basic security (i.e. confidentiality and message integrity) for the TLS handshake protocol and other higher layer protocols.

A TLS message consists of a header, some application data and a MAC. The header specifies the content type (i.e. the higher layer protocol), the TLS version and the length of the message. A more detailed description of the TLS record protocol header can be found in appendix B.3. The data and the MAC are encrypted, but the header is not. The TLS record format is shown in Figure 4.2.

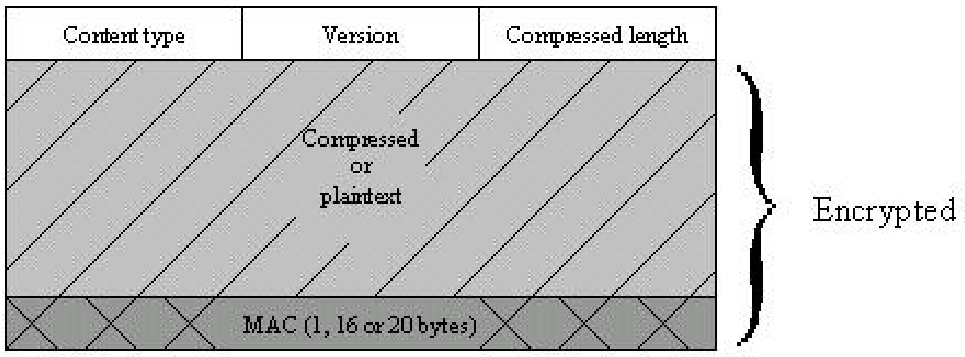

Figure 4.2: TLS record format, from [19]

4.1.2.1 What to do before sending and after receiving a message

When an upper layer message is delivered to the TLS protocol there are a few operations that TLS needs to do with the message before delivering it to the lower layer protocol (e.g. TCP or SCTP).

1. Fragmentation: Upper layer messages are fragmented into blocks of 2^{14} bytes or less.
2. Compression: The blocks are optionally compressed. As default, compression is not used in TLS. (The compression may not increase the content length with more than 2^{10} (1024) bytes.)
3. Add MAC: The MAC is calculated using a hash function (e.g. MD5 or SHA-1), the MAC secret and the message itself.
4. Encryption: The message is encrypted using the encryption algorithm (e.g. RC2, DES, RC4) defined in the session state (cipher specification) and the secret key. (Encryption may not increase the message content length with more than 2^{10} (1024) bytes. So, the total message content length may not exceed 2^{14} + 2048 bytes.)

5. Append TLS record header: Information about content type (i.e. higher layer protocol), TLS version and the length of the compressed data is added to the fragment.

When a lower layer message is delivered to TLS, TLS needs to do the following operations on the message before delivering it to the upper layer:

1. Remove TLS record header.
2. Decrypt the message.
3. Verify message integrity (using the MAC).
4. Decompress the message (if compression is used).
5. Reassemble the message.

4.1.3 TLS handshake protocol

The handshake protocol is used to authenticate the client and the server and to negotiate the encryption algorithms, the MAC algorithm and the encryption keys.

The encryption- and MAC algorithm and the encryption keys will be referred to as a set of security parameters. The handshake protocol defines one set of security parameters called the current state. These security parameters are used by the record layer. The handshake protocol also defines one or more sets of security parameters called the pending state. The pending state is simply a list of future security parameters.

A TLS handshake message consists of three parts, as illustrated in Figure 4.3. The first field specifies the type of handshake message sent (e.g. client_hello, server_hello and so on). The next field contains the length of the message content (i.e. the length of the content field). The content field depends on the message type. A detailed description of the different message types is given in appendix B.4.

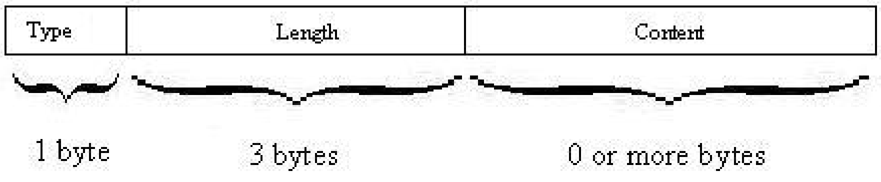

Figure 4.3: Handshake message format, from [19]

4.1.4 TLS change cipher protocol

The change cipher spec protocol is used to change the pending state into the current state. The change of security parameters once in a while gives an intruder less time to discover algorithms or keys.

A change cipher spec message always looks the same. It consists of a single byte with value 1. The change cipher spec message format is shown in Figure 4.4.

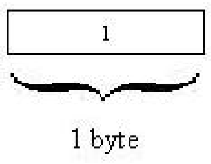

Figure 4.4: Change cipher spec message format, from [19]

4.1.5 TLS alert protocol

The alert protocol is used to report errors and warnings. The alert message consists of two fields, shown in Figure 4.5. The level field specifies the severity of the message. The alert can be a warning or a fatality. If the alert is a warning the receiver may determinate weather to terminate or continue the connection. If the alert is fatal the connection must be terminated. Other connections in the session may continue when a fatal alert is received, but no new connections may be initiated. The alert field is a description of the alert. The different alerts are listed in appendix B.5.

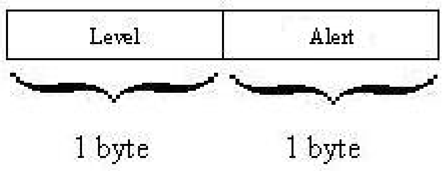

Figure 4.5: Alert message format, from [19]

4.1.6 Connection process

The setup of a TLS connection can be divided into four phases. The setup is shown in Figure 4.6. Asterix (*) indicates that a message is optional.

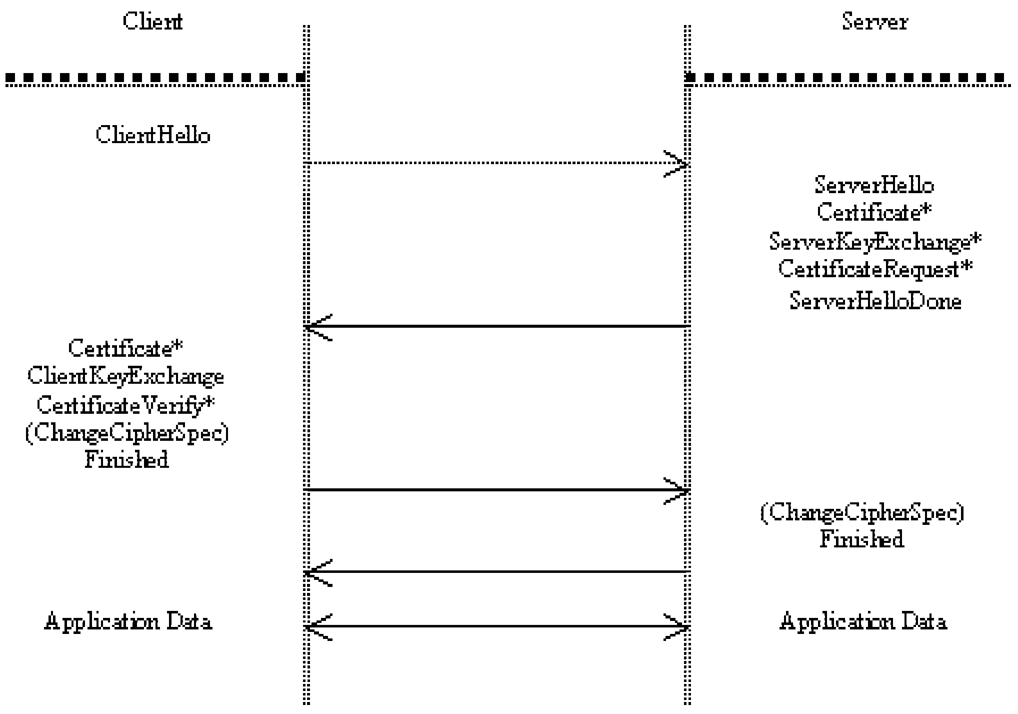

Figure 4.6: TLS connection setup, from [1]

Phase one establishes security capabilities. During the first phase the following messages are sent:

1. Server sends a hello request message. When a connection is established for the first time this message is not sent (since it is the client who initiates the connection). Later, however, the server can send this message when it is time to renegotiate the security parameters.
2. Client sends client hello message.
3. Server sends server hello message.

Phase two authenticates the server and exchanges keys. The following messages are sent during this phase:

4. Server sends its certificate message (optional).
5. Server sends key exchange message (optional).
6. Server sends certificate request message (optional).
7. Server sends server hello done.

Phase three authenticates the client and exchanges keys. The following messages are sent:

8. Client sends its certificate message (optional).

9. Client sends key exchange message.
10. Client sends certificate verify (optional).

Phase four finishes the negotiation of security parameters by sending these messages:

11. Client sends change cipher spec message.
12. Client sends a finished message.
13. Server sends change cipher spec message.
14. Server sends a finished message.

4.2 Socket API

This section will describe the important functions in the socket API. The socket API is used by openSSL to communicate with TCP. The socket calls are of great interest when adapting openSSL to SCTP. Table 4.1 gives a short description of the most important socket API functions.

Socket call	**Description**
socket	'socket' must be called by both client and server. 'socket' lets the user specify the protocol (TCP, UDP or IP) to be used. The 'socket' call returns a file descriptor, called a socket, which will be used in the other socket calls.
bind	'bind' is used only by the server. 'bind' lets the server associate the socket with a specific port on which it intends to listen.
listen	'listen' lets the server listen for incoming connections. When the server calls 'listen', execution will pause until a client connects to the server. When 'listen' returns control the server may accept the incoming connection, if no errors occurred.
connect	'connect' is used by the client to connect to a server. The client must specify the address of the server and the port to connect to. The server must listen to this port.
accept	The server uses 'accept' to accept the connection initiated by the client's call to 'connect'. 'accept' should only be called when 'listen' has returned control and no errors have occurred. 'accept' will return a new socket (file descriptor) that will handle the new connection. The old socket is usually used to continue listening to the original port. The new socket can be used for sending and receiving data.
write	'write' is used by the client and the server to send messages to each other.

read	‘read’ is used by the client and the server to receive messages from each other.
close	‘close’ is used by the client and the server to disable a socket (file descriptor) for the process that makes the call. ‘close’ disables the process from sending or receiving data via this socket. An attempt to send queued data will be made before the socket is disabled. If the socket is shared with another process, the other process may still use the socket.
shutdown	‘shutdown’ is used by the client and the server to free all resources held by the socket (file descriptor). ‘shutdown’ disables the socket for all processes.

Table 4.1: Socket API functions, based on [1]

The client and the server use the socket API in different ways, as illustrated in Figure 4.7. The server calls ‘socket’, ‘bind’, ‘listen’ to start listening for upcoming connections and ‘accept’ to accept them. The client calls ‘socket’ and ‘connect’ to connect to a server. Both client and server then use ‘write’ and ‘read’ to send and receive messages. Later they both use close or shutdown to terminate the connection.

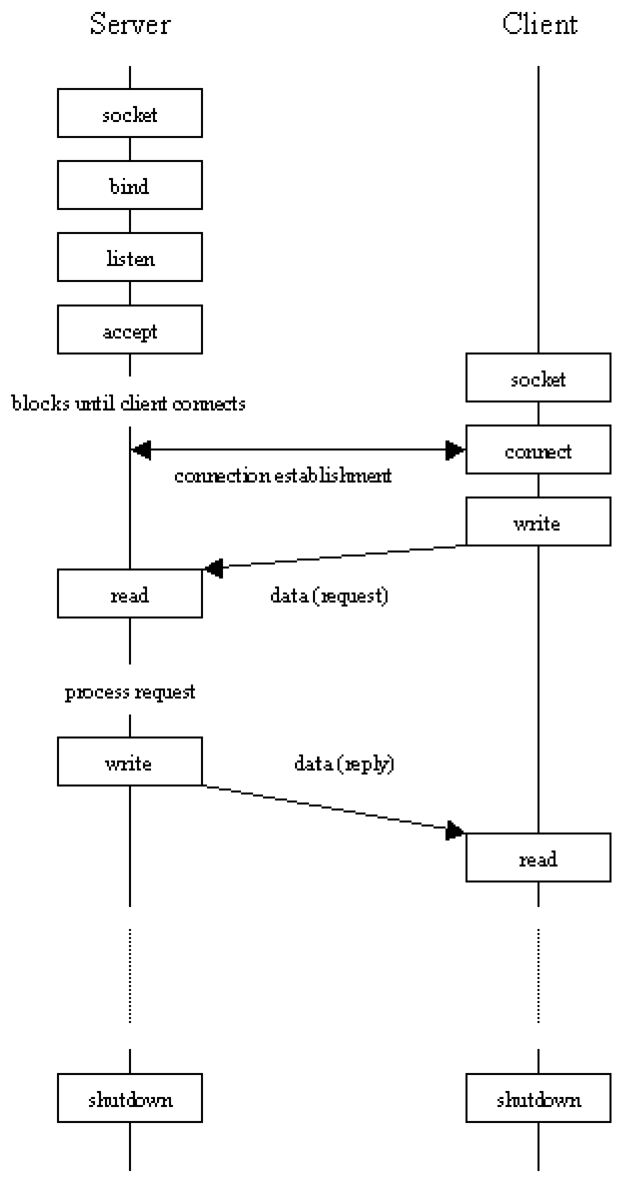

Figure 4.7: Client and server Socket API usage, based on [3]

4.3 openSSL – an existing implementation of TLS

This section describes the parts of openSSL that are of importance for this work. The openSSL code is designed to run on top of TCP (Transmission Control Protocol). Since TietoEnator's SS7 implementation uses CP and SCTP instead of TCP this section will concentrate on the parts of openSSL that communicate with TCP. These parts need to be changed in order to get openSSL to run on top of CP and SCTP.

Section 4.3.1 describes the file and folder structure in openSSL. An overview of the files that handle TCP communication is given in section 4.3.2. Section 4.3.3 explains how these files, and the functions in them, are used by openSSL client and server applications.

4.3.1 Interesting files and folders in openSSL

The openSSL code consists of numerous folders and files as shown in appendix C. Some of these files and folders are more interesting than others. Files that need consideration or even modification when adapting openSSL to CP and SCTP are considered very interesting.

Here is a summary of the interesting files and folders:

- openssl-0.9.7c (root folder)
    - Makefile: This Makefile builds the openSSL library. There are several Makefiles in the openSSL project, one for each subfolder that contains source code files. An important task of the root Makefile is to start and coordinate the work of the other Makefiles.
    - e_os.h: contains a few macros for reading and writing to a socket.
- openssl-0.9.7c/apps
    - s_socket.c: handles the TCP initialization for both client and server. This file contains a lot of socket calls.
    - s_server.c: used a lot when openSSL acts as a server.
    - s_client.c: used a lot when openSSL acts as a client.
    - app_rand.c: uses other files that make socket calls.
- openssl-0.9.7c/crypto/bio: Since this folder is under crypto (cryptography) it should (and probably does) have some connection to cryptographic computations. Since BIO stands for basic input output, the folder, 'bio', should contain the input and output functionality, like socket calls, needed for encryption. It should be noted that the openSSL code analysis concerning this folder is incomplete.
    - b_sock.c: contains a lot of TCP initialization functionality.
    - bss_acpt.c: contains a few socket calls.
    - bss_conn.c: contains a few socket calls.
    - bss_sock.c: contains functionality for reading and writing to sockets.
- openssl-0.9.7c/crypto/rand
    - rand_egd.c: contains some socket calls.
    - rand_unix.c: uses other files that make socket calls.

- openssl-0.9.7c/crypto/threads
    - mttest.c: uses other files that make socket calls.
- openssl-0.9.7c/crypto/x509
    - x509_lu.c: contains some socket calls.
- openssl-0.9.7c/ssl: This folder contains files that constitute the openSSL interface to user applications (at the application layer).
    - bio_ssl.c: uses other files that make socket calls.
    - ssl_lib.c: contains important library functions.
    - ssl_algs.c: contains the function used to initiate the library.
- openssl-0.9.7c/test
    - ssltest.c: uses other files to make socket calls.

4.3.2 Socket call files

This section lists the files that contain communication with TCP. Unfortunately there are several files spread out in the openSSL folder structure. The ideal situation would have been if all TCP communication were located to one folder with one or a few files.

Table 4.2 lists the files that contain socket calls (i.e. communicates with TCP). Socket calls in parenthesis are defined as macros in 'e_os.h' (e.g. closesocket is replaced with close by the preprocessor). In the source code the names within parenthesis will be found. Table 4.2 is not complete. Some files with socket calls have been left out because they are demos, test applications or because they are made for other platforms than Unix.

File	Socket calls
./e_os.h	(macros for) recv, send, shutdown, socket, close
./apps/s_socket.c	accept, bind, connect, listen, setsocketopt, socket
./crypto/bio/b_sock.c	accept, bind, connect, getsockopt, listen, setsockopt, socket, close (closesocket)
./crypto/bio/bss_acpt.c	shutdown, socket, close (closesocket), ioctl (ioctlsocket)
./crypto/bio/bss_conn.c	connect, setsockopt, socket, read (readsocket), write (writesocket), close (closesocket)
./crypto/bio/bss_sock.c	read (readsocket), write (writesocket)
./crypto/rand/rand_egd.c	connect, socket
./crypto/x509/x509_lu.c	shutdown

Table 4.2: Socket call files

Table 4.3 lists the location (i.e. file and function that makes the call) for each socket call. The numbers within parenthesis represent the order in which the socket calls are made for a specific location (function).

File	Socket call	Location (defined in function)
./e_os.h	recv	(defined as a macro)
	send	(defined as a macro)
	shutdown	(defined as a macro)
	socket	(defined as a macro)
./apps/s_socket.c	accept	do_accept
	bind	init_server_long
	connect	init_client_ip
	listen	init_server_long
	setsockopt	init_client_ip
	setsockopt	init_server_long
	socket	init_client_ip
	socket	init_client_long
./crypto/bio/b_sock.c	accept	BIO_accept
	bind (3)	BIO_get_accept_socket
	connect (5)	BIO_get_accept_socket
	getsockopt	BIO_sock_error
	listen (6)	BIO_get_accept_socket
	setsockopt (2)	BIO_get_accept_socket
	setsockopt	BIO_set_tcp_ndelay
	socket (1), (4)	BIO_get_accept_socket
	shutdown (closesocket)	acpt_close_socket
	ioctl (ioctlsocket)	BIO_ioctl_socket
./crypto/bio/bss_acpt.c	shutdown (closesocket)	acpt_close_socket

./crypto/bio/bss_conn.c	connect (3)	conn_state
	setsockopt (2)	conn_state
	socket (1)	conn_state
	read (readsocket)	conn_read
	write (writesocket)	conn_write
	shutdown (closesocket)	conn_close_socket
./crypto/bio/bss_sock.c	read (readsocket)	sock_read
	write (writesocket)	sock_write
./crypto/rand/rand_egd.c	connect (2)	RAND_query_egd_bytes
	socket (1)	RAND_query_egd_bytes
./crypto/x509/x509_lu.c	shutdown	X509_LOOKUP_shutdown

Table 4.3: Socket call locations

4.3.3 Socket call chains

This section describes the different function call sequences that end with socket calls. The idea is to give a better understanding of how the communication with TCP is done and which socket call locations that are of importance.

4.3.3.1 Server socket call chains

Figure 4.8 displays the socket calls initiated by the server in the 'apps' folder. The server application calls 'do_server' to set up a TCP connection. 'do_server' uses 'init_server' to create a socket, set some connection properties, bind the socket to a port and start listening for upcoming connections. The socket calls 'socket', 'bind' and 'listen' are used for this purpose. Next 'do_server' calls 'do_accept' to accept an upcoming connection. In the end this is done with the socket call 'accept'.

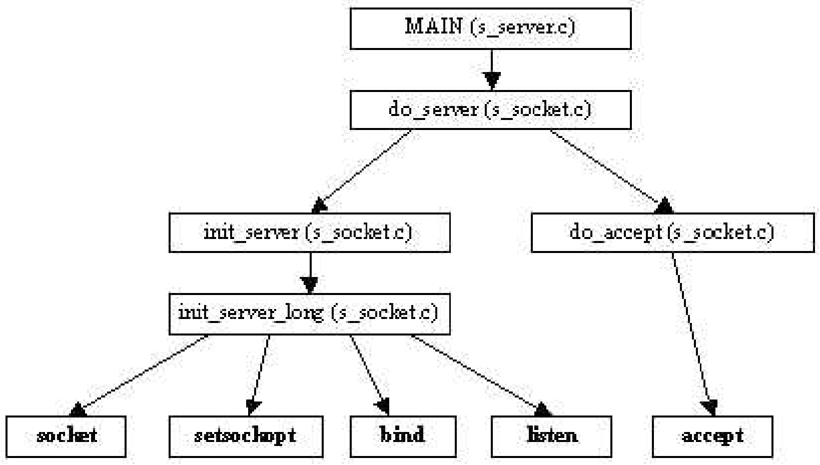

Figure 4.8: Server initialization socket call chains

4.3.3.2 Client socket call chains

Figure 4.9 illustrates the socket call chains initiated by the client in the 'apps' folder. The client creates a socket, sets some connection properties and connects to the server, by calling 'socket', 'setsockopt' and 'connect' respectively.

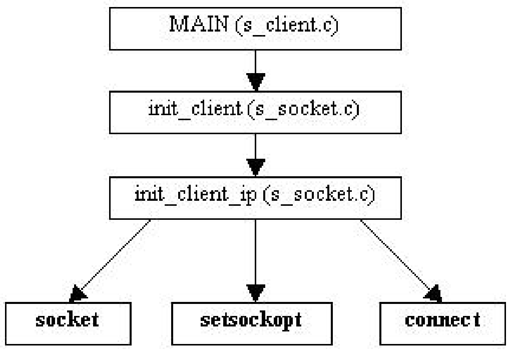

Figure 4.9: Client initialization socket call chains

4.3.3.3 Send and receive socket call chains

Figure 4.10 below illustrates the socket call chains used when sending messages. Examples of applications and functions that send messages are the server and the client in the 'apps' folder and the openSSL library function 'ssl_write' in 'bio_ssl.c'. The actual sending is done with the function 'write'.

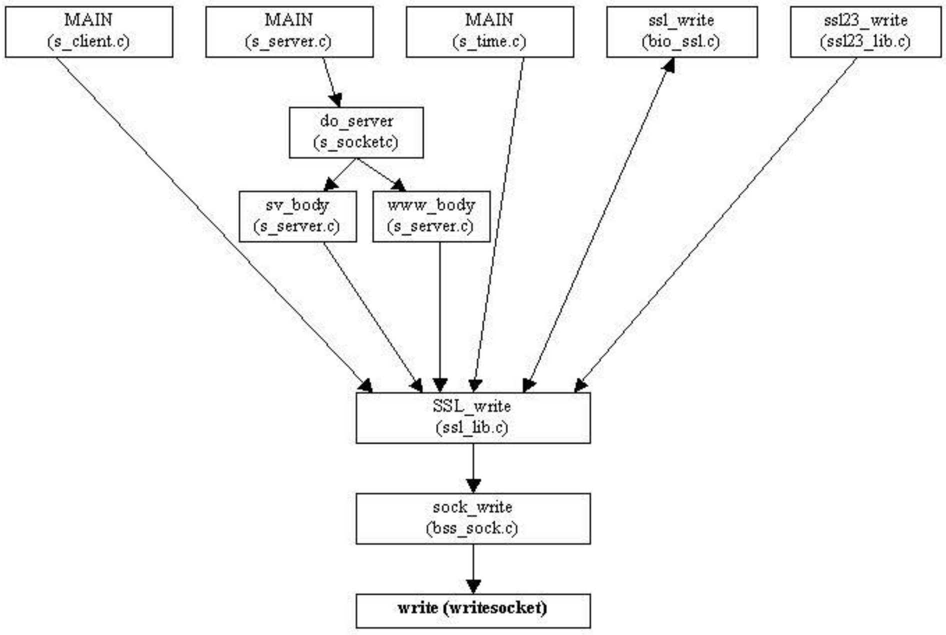

Figure 4.10: Sending socket call chains

Figure 4.11 shows the socket call chains used when receiving messages. The receiving is done pretty much by the same files that send messages. The client and server in the 'apps' folder receive messages and so does the openSSL library function 'ssl_read'. The actual receiving is done with the function 'read'.

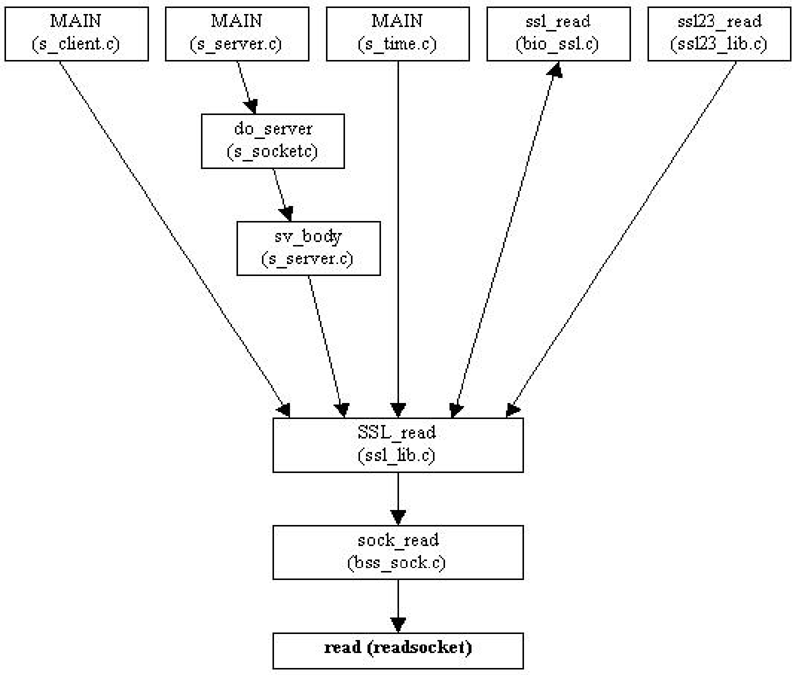

Figure 4.11: Receiving socket call chains

4.3.3.4 Cryptographic socket call chains

This section describes the socket call chains, (probably) used by cryptographic functionality. The code analysis done for these call chains is not complete and does not give much understanding of how these call chains work or when they are used. However, a brief description of them is included anyway.

Figure 4.12 shows the socket call in the file 'b_sock.c'.

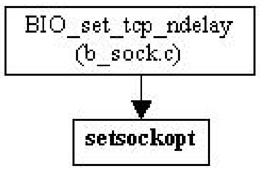

Figure 4.12: 'b_sock.c' socket call chain

Figure 4.13 illustrates the socket calls initiated by the file 'bss_acpt.c'.

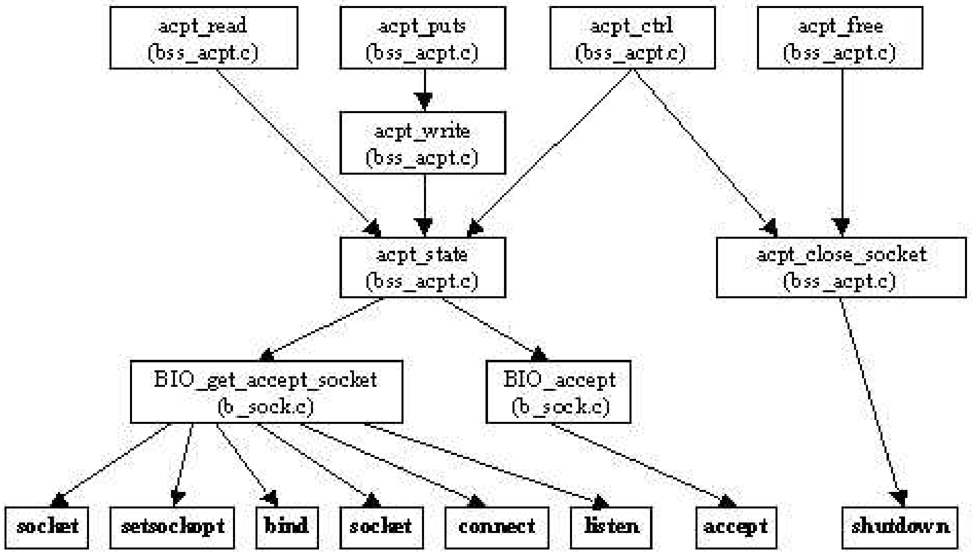

Figure 4.13: 'bss_acpt.c' socket call chains

Figure 4.14 displays the socket calls initiated by the file 'bss_conn.c'.

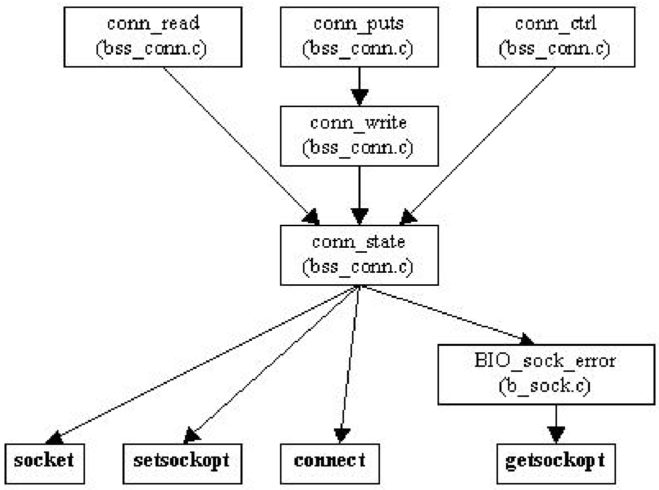

Figure 4.14: 'bss_conn.c' socket call chains

4.4 Chapter summary

This chapter describes TLS, which is a security protocol placed above the transport layer protocol. TLS becomes a part of the transport layer so that the application layer does not notice the presence of TLS. TLS consists of four sub protocols. The record layer protocol provides the basic security for the other three protocols and for the data transmission. The security parameters used by the record layer protocol are negotiated using the handshake protocol. The change cipher spec protocol is used to signal that it is time to change these security parameters. Error conditions are reported using the alert protocol.

TLS strives to guaranty confidentiality, message integrity and authentication between client and server.

The TLS implementation, openSSL, used for this work communicates with TCP via the platforms socket API. This chapter gives a short description of the socket API.

The end of this chapter analyzes openSSL. The interesting parts of openSSL are the ones communicating with TCP (i.e. the socket API). These parts need to be modified in order to adapt openSSL to CP/SCTP at TietoEnator. The openSSL connection is set up from the files 's_server.c', 's_client.c' and 's_socket.c'. Data transmission is handled in 'ssl_lib.c'.

The knowledge about openSSL, given in this chapter, together with the knowledge about TietoEnator's SS7 environment, given in chapter 3, lead to the openSSL over CP/SCTP design, described in chapter 5.

5 Design of openSSL over CP/SCTP

This chapter describes how openSSL can be adapted to TietoEnator's SS7 environment. Since openSSL was originally implemented to run on top of TCP, the adaptation of openSSL to CP/SCTP is the process of translating the TCP socket calls to CP/SCTP API functions. Section 5.1 discusses some different approaches to the adaptation of openSSL to CP/SCTP. The approach selected for this work is described in section 5.2.

5.1 Design considerations

This section describes different design approaches, which possibly can be used to adapt openSSL to TietoEnator's SS7 implementation. Independent of the design approach selected for this work, an understanding of how socket calls map to CP/SCTP calls is needed. An attempt to carry out this mapping is shown in Table 5.1. Remember the CP/SCTP API functions from section 3.2.2.

CP/SCTP function	Socket call
EINSS7CpRegisterMPOwner	socket
EINSS7CpRegisterRemoteCPMgmt	
EINSS7CpMsgInitInitiate	
EINSS7CpMsgPortOpen	
EINSS7_00SctpRegFunc	listen, accept
EINSS7CpMsgConnInst	
EINSS7_00SctpBindReq	bind
MsgRecvEvent	listen, accept, recv
MsgSend	send
EINSS7_00SctpHandleInd	
EINSS7_00SctpInitializeReq	
EINSS7_00SctpAssociateReq	connect
EINSS7_00SctpSetPrimaryReq	

Table 5.1: Mapping between CP/SCTP functions and openSSL socket calls

There are a few problems with this mapping:

1. Some CP/SCTP API functions have no corresponding socket calls. This is not necessarily a problem, but it still needs to be considered.
2. ‘EINSS7_00SctpBindReq’ maps to ‘bind’. In CP/SCTP both the client and the server need to do an ‘EINSS7_00SctpBindReq’ whereas with the socket API only the server should do a ‘bind’.
3. ‘EINSS7_00SctpRegFunc’ and ‘MsgRecvEvent’ map to ‘listen’ and ‘accept’. In the socket API the server calls ‘listen’ in order to listen for an upcoming connection. When the client connects to the server, the server calls ‘accept’ to accept the connection. In CP/SCTP this is handled differently. CP/SCTP has an event handling system. The function ‘EINSS7_00SctpRegFunc’ is used to specify the functions that should be called when different events occur. So, the server must implement a function that handles the accepting of a connection and register this function with ‘EINSS7_00SctpRegFunc’. Then ‘MsgRecvEvent’ is called to listen for upcoming connections. When the client connects to the server, CP will call the function specified with ‘EINSS7_00SctpRegFunc’.

5.1.1 Translation of each socket call into a CP/SCTP API function call

This design approach involves replacing each socket call (e.g. socket, bind, listen, connect and accept) with its corresponding CP/SCTP call. This approach is illustrated in Figure 5.1. The words ‘socket’, ‘bind’, ‘listen’ and so on represents calls to the socket API. Ellipses mark code locations that need modifications. The question marks represent the new CP/SCTP function calls.

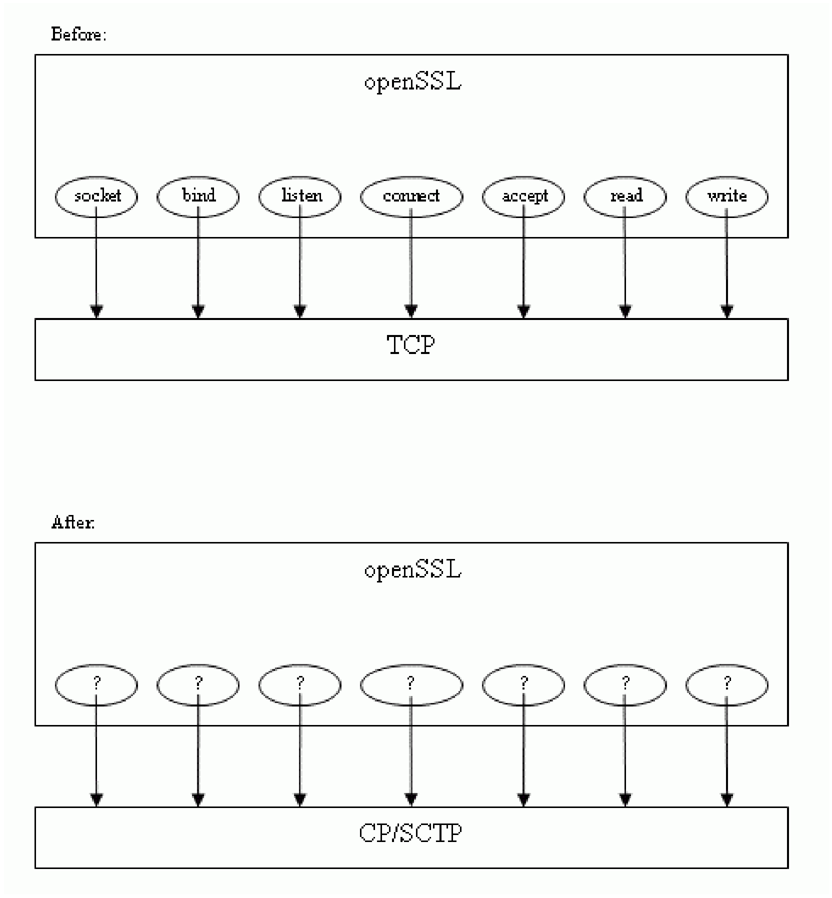

Figure 5.1: Translation of each socket call into a CP/SCTP API function call

One advantage with this design approach is the ease with which new openSSL versions can be adapted to CP/SCTP. At the arrival of a new openSSL version, all that needs doing is replacing each socket call with its corresponding CP/SCTP function. Another advantage with this approach is that it is easy to implement, since it does not require much understanding of the openSSL code. Most important is that this approach saves time and money.

The disadvantage with this approach is that it does not work, because of the problems described in section 5.1. Problem 1 could be solved without too much modification. However, problem 2 results in only the openSSL server calling 'EINSS7_00SctpBindReq' even though the openSSL client also needs to make this call. The 'bind' and 'accept' calls in openSSL need to be handled differently with CP/SCTP, because of problem 3.

5.1.2 Create a translator between the socket calls and the CP/SCTP API

This design approach involves creating new implementations of all socket calls, where each socket call implementation translates the original socket call functionality into the corresponding CP/SCTP API function or functions. The result of implementing this design is a socket-CP/SCTP translating software module. Figure 5.2 is an illustration of this design approach.

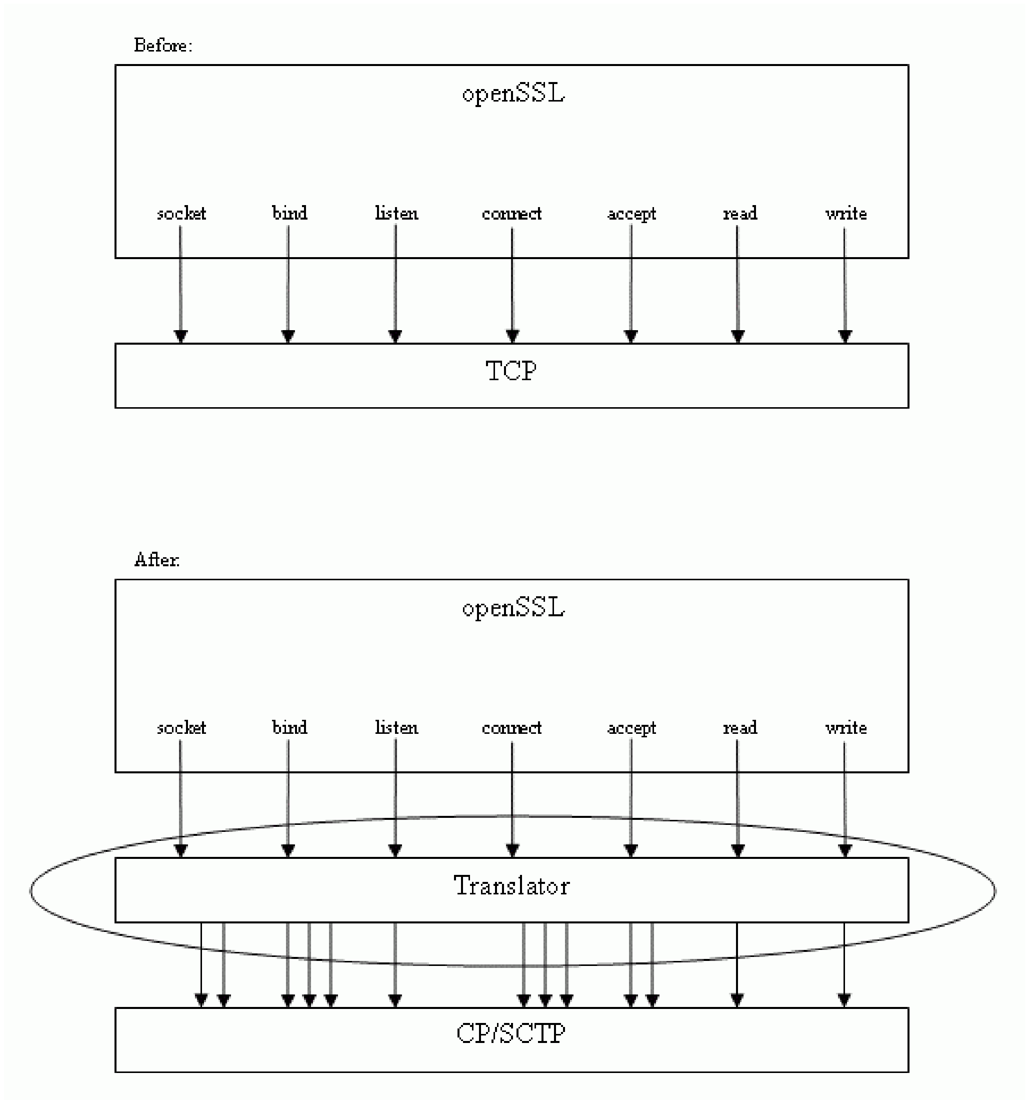

Figure 5.2: Create a translator between the socket calls and the CP/SCTP API

The following aspects need to be considered for this design approach:

1. The openSSL code contains a lot of includes to the h-file ‘socket.h’, which gives access to the socket library. These includes need to be replaced by a new h-file containing the definitions of the socket CP/SCTP translating functions.
2. Problem 2 in section 5.1 concerning the ‘EINSS7_00SctpBindReq’ to ‘bind’ mapping needs special treatment. Since both the SCTP client and server need to call ‘EINSS7_00SctpBindReq’ it should probably not be done in the translation of ‘bind’ since ‘bind’ is only called by the openSSL server.

3. Problem 3 in section 5.1 concerning the mapping between the socket functions 'listen' and 'accept' and the CP/SCTP functions 'EINSS7_00SctpRegFunc' and 'MsgRecvEvent' also needs a special solution. Clearly the translator needs to be fairly intelligent. A state (i.e. memory) is probably a good start.

One advantage with this design approach is that it requires only small modifications of a new openSSL version. However, this approach demands much understanding the socket API and the CP/SCTP API. The implementation of this design will be difficult and requires much time and money, but needs to be done only once.

5.1.3 Rewriting the parts of openSSL that communicate with the socket API

This design approach involves modifying the parts of openSSL that communicates with the socket API. For example, the initialization of the openSSL server illustrated in Figure 4.8 needs a complete rewriting of the function 'do_server'. This design approach is illustrated in Figure 5.3.

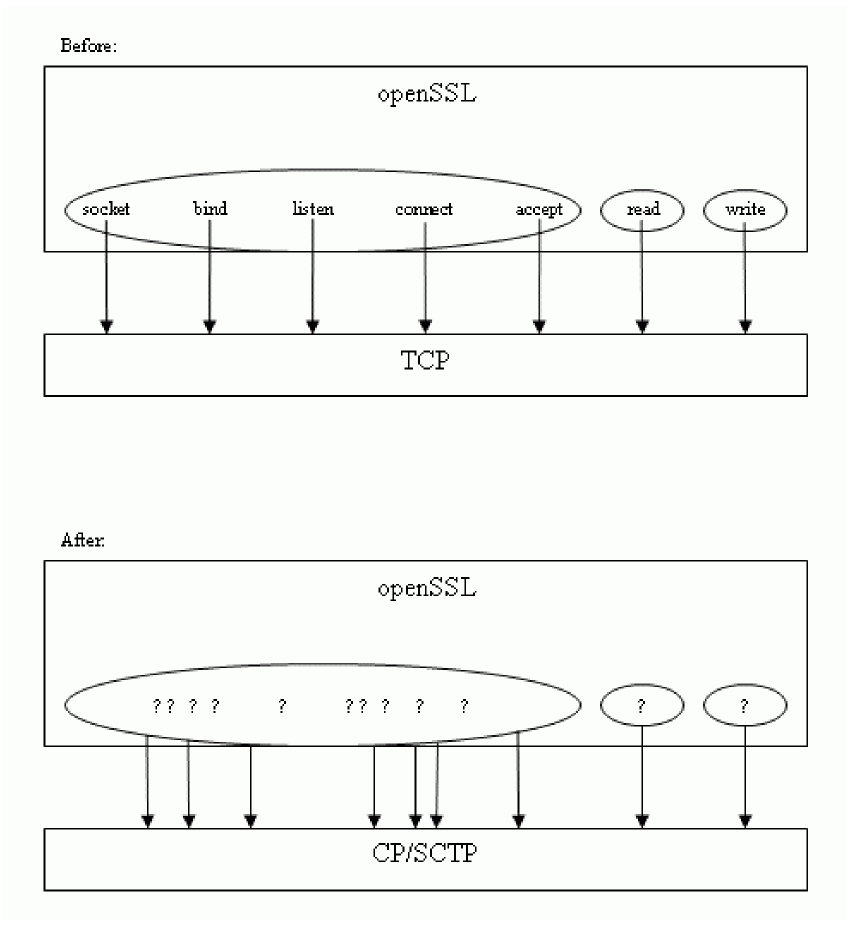

Figure 5.3: Rewriting the parts of openSSL that communicate with the socket API

There is no real advantage with this design approach except that it is viable. It is just a question of how extensive the rewriting will be. One disadvantage is that it requires a lot of understanding of the openSSL code, which implies that the implementation will be difficult. Another big problem appears at the arrival of a new openSSL version. Even though the new openSSL code might be similar to the one used and the understanding of the code comes easy, the entire implementation work needs to be redone. Both the initial design and the maintenance of the code (at new openSSL releases) require lot of time and money.

5.2 Selected design

Section 5.1 describes some different design approaches to the adaptation of openSSL to TietoEnator's SS7 implementation. The first approach, 'Translation of each socket call into a CP/SCTP API function call', described in section 5.1.1 is preferable for a test version of this work, because of its fast implementation. The second approach, 'Create a translator between the socket calls and the CP/SCTP API', described in section 5.1.2 is preferable for the final production version of this work, because of its simple maintenance. Since the first approach is impossible to implement the second approach or the third one, 'Rewriting the parts of openSSL that communicate with the socket API' presented in section 5.1.3, must be selected for this work. In the initial stages the third approach requires less work than the second approach does. Therefore the third approach is more desirable for a test version. Consequently design approach three, 'Rewriting the parts of openSSL that communicate with the socket API', is selected for this work.

The selected design is described in this section. Note that the description is limited to the initialization parts and the data transmission, since this is what the openSSL analysis in section 4.3 covered.

Modifications to the openSSL files and folders are described in section 5.2.1. The initialization of the openSSL client and server are described in section 5.2.2. Section 5.2.3 describes data transmission. Note that initialization and data transmission is mainly described by figures. The text describing the figures should be seen as a complement to them and not vice versa. Section 5.2.4 gives an overview of the openSSL Makefiles modifications.

5.2.1 File and folder structure

This section concentrates on pointing out the locations in the openSSL file and folder structure that need altering when adapting openSSL to CP/SCTP. The entire openSSL file and folder structure (including these modifications) can be found in appendix C.

Most important is that the folder 'TLSoverSCTP' should be added. The idea is that all CP/SCTP communication should pass through files in this folder. The folder should contain two important files. 'TLSoverSCTP.c' should contain function implementations for the initialization of CP/SCTP and for data transmission via CP/SCTP. 'TLSoverSCTP.h' should contain declarations for the functions in 'TLSoverSCTP.c'.

The openSSL file that handles the TCP initialization is 's_socket.c' in the 'apps' folder. The initialization functionality in this file must be rewritten. It must handle the initialization of CP/SCTP. This is done via calls to 'TLSoverSCTP.c'.

The openSSL send and receive functionality was originally handled by the openSSL library functions in 'ssl_lib.c' in the 'ssl' folder. This file needs to call the CP/SCTP send and receive functions instead of the data transmission functions in the socket API. This is also done via 'TLSoverSCTP.c'.

5.2.2 Client and server initialization

The initialization of the openSSL server originally involves the socket API calls 'socket', 'setsockopt', 'bind', 'listen' and 'accept', as illustrated in Figure 4.8. A complete rewriting of the function 'do_server' is needed, because of the mapping problem (3) with 'listen' and 'accept', described in section 5.1. 'do_server' branches off into two different functions. One of these functions calls 'listen' whereas the other one calls 'accept'. CP/SCTP handles listening and accepting differently with the functions 'EINSS7_00SctpRegFunc' and 'MsgRecvEvent'.

Instead of calling 'init_server' and 'do_accept', 'do_server' must call 'TLSoverSCTP_startServer' in 'TLSoverSCTP.c'. This function handles the CP/SCTP initialization, as shown in Figure 5.4. Remember the CP/SCTP API functions described in section 3.2.2. 'EINSS7_00SctpRegFunc' is called to specify the functions that should handle different events like, the receipt of a message or an upcoming connection. 'EINSS7CpRegisterMPOwner' is used to register the CP user. 'EINSS7CpRegisterRemoteCPMgmt' is used to register the CP manager, who handles the communication between the CP user and CP. This function is also used to specify the CP user's process (i.e. IP-address and port number). 'EINSS7CpMsgInitiate' is used to initialize and configure CP. ‘EINSS7CpMsgPortOpen’ specifies how message buffers shall be used and enables the CP user to setup a connection to another CP user. Next ‘EINSS7CpMsgConnInst’ sets up the connection to another CP user. Note that this is not the corresponding function to the socket API’s ‘connect’ and it’s meaning is unclear to the authors of this document. ‘EINSS7CpMsgConnInst’ is still a function that the server and the client need to call. More information about this function can be found in [4]. Next the server calls ‘EINSS7_00SctpBindReq’ to register the server as a SCTP user. ‘MsgRecvEvent’ is called to listen for upcoming connections and other events. ‘EINSS7_00SctpHandleInd’ interprets messages received with ‘MsgRecvEvent’. ‘EINSS7_00SctpInitializeReq’ is used to set the IP-addresses used for the SCTP multi-homing service, described in section 3.1.2. ‘EINSS7_00SctpSetPrimaryReq’ sets the primary destination IP-address (path) used for multi-homing.

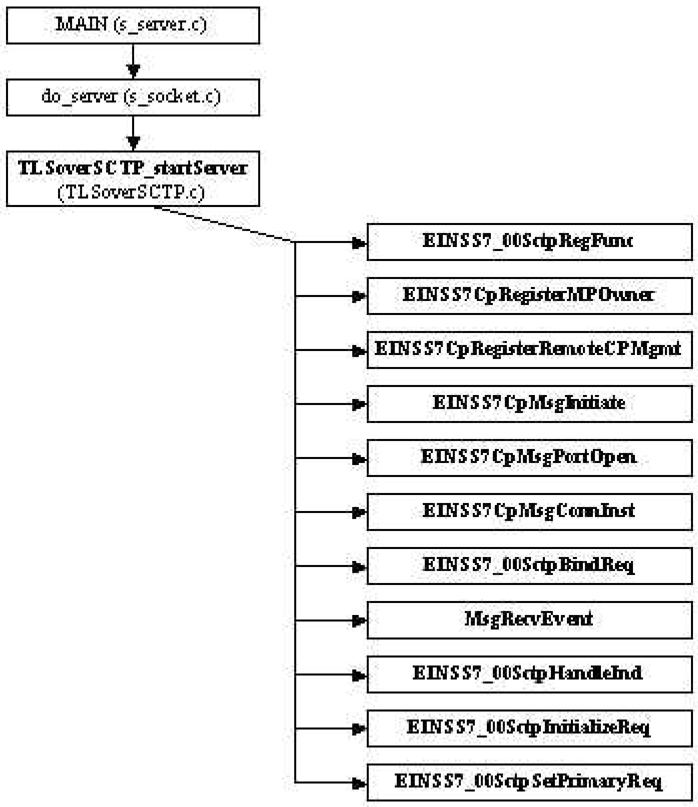

Figure 5.4: Server CP/SCTP initialization call chains

The openSSL client initialization was originally done with the socket calls ‘socket’, ‘setsockopt’ and ‘connect’ as shown in Figure 4.9. This call chain could be rewritten from ‘init_client’ or ‘init_client_ip’. To make the code structure more like the one used for the server initialization, rewriting is done from the ‘init_client’ function.

Instead of calling ‘init_client_ip’, as done in Figure 4.9, ‘init_client’ shall call ‘TLSoverSCTP_startClient’ in ‘TLSoverSCTP.c’, as shown in Figure 5.5. ‘TLSoverSCTP_startClient’ calls the same CP/SCTP functions, as does ‘TLSoverSCTP_startServer’ in Figure 5.4. However, the client initialization requires an extra call to ‘EINSS7_00SctpAssociateReq’, described in section 3.2.2. ‘EINSS7_00SctpAssociateReq’ sets up the association to the server.

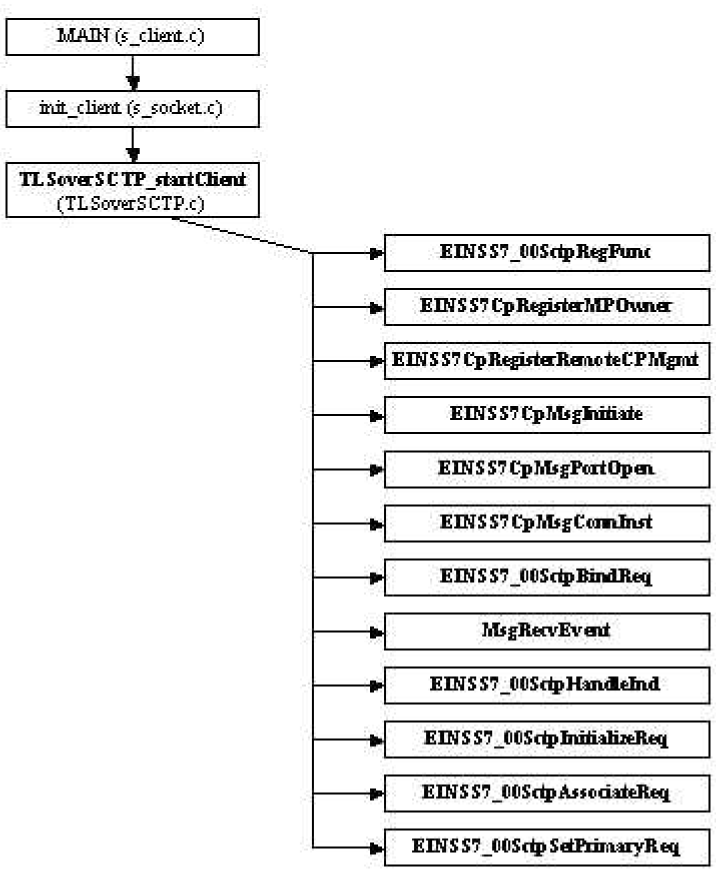

Figure 5.5: Client CP/SCTP initialization call chains

5.2.3 Data transmission

Data sending was originally done with the ‘send’ socket call, as displayed in Figure 4.10. This call chain may be modified so that the library function ‘SSL_write’ (via ‘sock_write’) calls ‘TLSoverSCTP_send’ in ‘TLSoverSCTP.c’, as illustrated in Figure 5.6. ‘TLSoverSCTP_send’ makes the call to the CP/SCTP API function ‘MsgSend’.

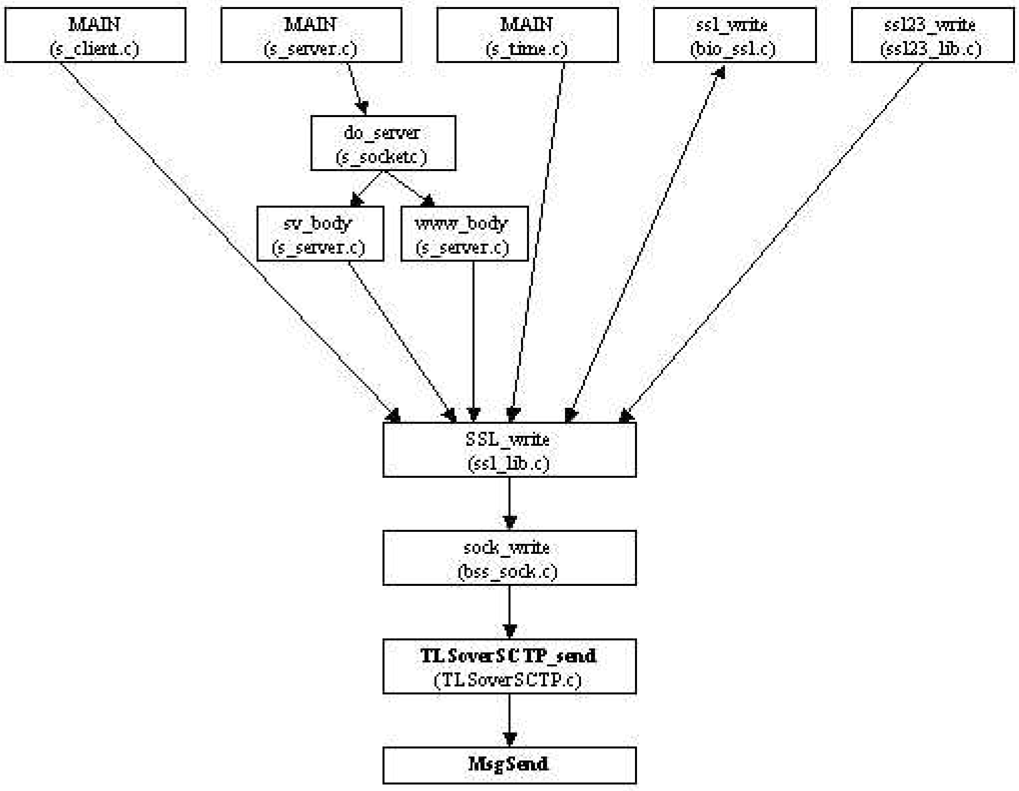

Figure 5.6: CP/SCTP sending call chains

The 'recv' socket call originally handled the receipt of data, as shown in Figure 4.11. The receiving call chain should be modified so that the 'SSL_read' function (via 'sock_read') calls 'TLSoverSCTP_recv'. 'TLSoverSCTP_recv' should then call the CP/SCTP API function 'MsgRecvEvent'. This is displayed in Figure 5.7.

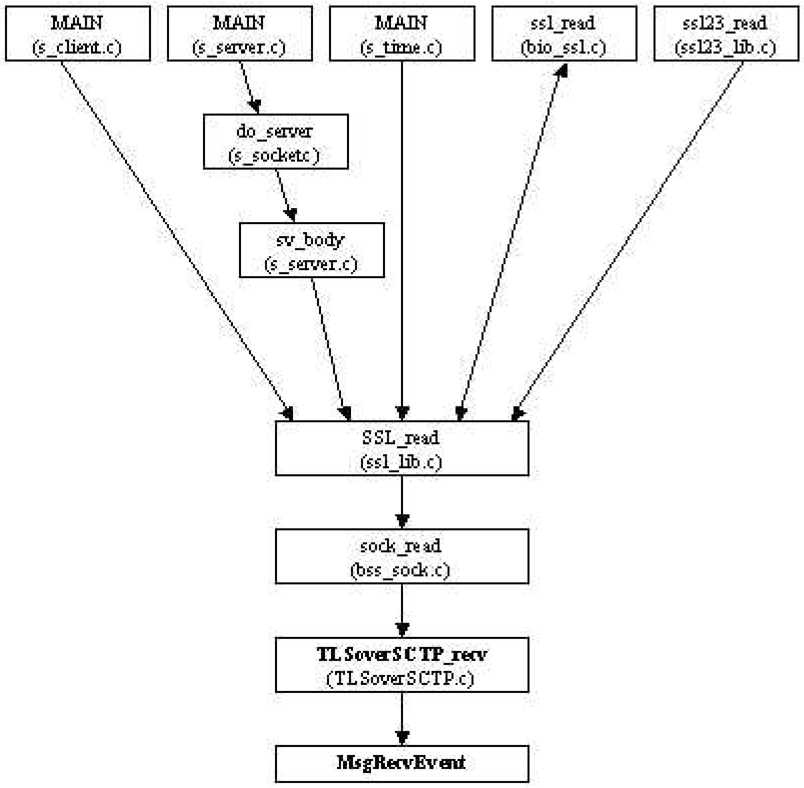

Figure 5.7: CP/SCTP receiving call chains

5.2.4 Makefiles

The Makefiles in openSSL handle code compilation and linking. The selected design, 'Rewriting the parts of openSSL that communicate with the socket API', requires some changes in a few existing Makefiles and the adding of a new Makefile, in the 'TLSoverSCTP' folder.

The Makefile in the 'TLSoverSCTP' folder should handle the compilation of 'TLSoverSCTP.c'. The Makefile must give this code access to the CP and SCTP libraries.

The Makefiles that need altering are the ones in the 'apps' folder and the 'ssl' folder. The Makefiles in the 'apps' and 'ssl' folder must provide access to the header-file and the object-files in the 'TLSoverSCTP' folder, since the initialization in the 'apps' folder and the data transmission in the 'ssl' folder call functions in 'TLSoverSCTP.c'.

5.3 Chapter summary

This chapter describes three design approaches for the adaptation of openSSL to CP/SCTP. The first design approach, 'Translation of each socket call into a CP/SCTP API function call', turns out to be impossible to implement. The second design approach, 'Create a translator between the socket calls and the CP/SCTP API', is suitable for the final production version of this work. Design approach three, 'Rewriting the parts of openSSL that communicate with the socket API', is the design selected for this work.

Rewriting the parts of openSSL that communicate with the socket API requires the creation of a new folder, 'TLSoverSCTP', containing files functioning as a port through which all openSSL-CP/SCTP communication must pass. Also the files 's_socket.c', which handles initialization, and 'ssl_lib.c', handling data transmission, must be modified. The cryptographic functionality is not considered in this design.

6 Problems

This chapter describes the problems encountered during this work.

One problem concerns the Makefiles in the openSSL code. The authors of this document have little or almost no experience of Makefiles, especially not as complicated ones as those in openSSL. The fact that openSSL contains multiple Makefiles, almost one in each folder, does not make this problem any less significant. The Makefile modifications needed involves providing some parts of the openSSL code with access to the CP and SCTP libraries. This is a complex process that requires a lot more time than first expected.

The Makefile problem was solved with help from staff at TietoEnator. However, the way this problem was solved is typical for a test implementation. There is probably a better and more understandable solution. The structure in the Makefiles is not satisfactory. Since the authors of this document have little Makefile experience and little time to get such an experience they can't refine this Makefile structure within the time frame of this work.

Another problem, related to openSSL, is the documentation of the code, or more correctly the lack of documentation. Documentation within the openSSL code files, explaining difficult code sections, is almost nonexistent. Documentation explaining the task and purpose of each function is only found in some cases. Unfortunately, such documentation could not be found where it was needed for this work. An overview of the functionality collected in each openSSL folder would also have been helpful, if it had existed. The documentation problem caused the openSSL code analysis to require much more time than expected. Because of the lack of documentation, there is also a risk that the openSSL analysis is misleading.

The documentation problem was solved by an extensive analysis of the openSSL code.

While reading the openSSL code, the authors of this document encountered a number of programming styles. The openSSL code is written by at least ten programmers, so there are numerous programming styles reflected in the code. The openSSL code analysis was slowed down, because of the frequent switching between different programming styles.

The openSSL code is written for more than one platform. The possibility to compile openSSL on different platforms is achieved by preprocessor directives controlling what parts of the code to compile on a specific platform. The preprocessor directives make the code hard to read, since the reader frequently must check if a specific part of the code is used or not. The checking may involve switching to other files, which takes time and perhaps causes the readers attention to focus on something less important than the code.

These problems where never solved. However, the analysis of the openSSL code revealed some of the hidden openSSL code structure.

Before the actual openSSL to CP/SCTP adaptation began, a lot of studying was required. Actually the chapters 2, 3 and 4 simply provide the knowledge needed to understand how the design in chapter 5 could be made. The studying done for this work involved general information about network security, the SCTP protocol and the CP/SCTP API functions, the TLS protocol and its implementation (openSSL), the Unix socket API and finally information about how Makefiles work.

7 Conclusions

The authors of this document have come to a few conclusions concerning a possible final production version of this work. As described in chapter 5, there are three possible design approaches to the adaptation of openSSL to CP/SCTP. The first approach, ‘Translation of each socket call into a CP/SCTP API function call’, is shown impossible to implement by the openSSL code analysis in section 4.3. The second one, ‘Create a translator between the socket calls and the CP/SCTP API’, is considered suitable for the final production version even though it requires a lot of work in the initial stages. The third approach, ‘Rewriting the parts of openSSL that communicate with the socket API’, is not suitable for the final production version because it requires much work at the arrival of a new openSSL version. However, this approach is acceptable for the test version described in this work.

During the analysis of the openSSL code the authors of this document realized that the best way to incorporate security in the SS7 protocol stack is not necessarily to use openSSL. The lack of documentation and the difficulty to read the openSSL code, explained in chapter 6, may overshadow the advantages with using openSSL. However, the proposed design approach for the final production version, ‘Create a translator between the socket calls and the CP/SCTP API’, does not require much understanding of the openSSL code and may therefore be used. The choice between implementing the TLS protocol from scratch and using openSSL is not obvious.

References

[1] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen and T. Wright, Request for Comments: 3546 Transport Layer Security (TLS) Extensions, June 2003.

[2] Breej's Guide to Network Programming, http://www.ecst.csuchico.edu/~beej/guide/net/html, 2001 10 08 (2004 04 16).

[3] OH-material from the course C & Unix (DAVC18) at Karlstad University, http://www.cs.kau.se/~ljh/CaU/course/davc18/oh/8_ipc.pdf, 2004 06 03.

[4] Common Parts Functional Specification, Ericsson, 15517-CAA 201 29 Uen AH, 2004 02 20.

[5] Cyclopedia Cryptologia, http://www.disappearing-inc.com, 2004 03 05.

[6] M. Elkins, Request for Comments: 2015 MIME Security with Pretty Good Privacy (PGP), October 1996.

[7] Y. Kawatsura, Request for Comments: 3538 Secure Electronic Transaction (SET) Supplement for the v1.0 Internet Open Trading Protocol (IOTP), June 2003.

[8] James F. Kurose and Keith W. Ross, Computer Networking: A Top-Down Approach Featuring the Internet (Second Edition), Addison Wesley, 2002.

[9] Peter Loshin, Big Book of IPsec RFCs: Internet Security Architecture, Morgan Kaufmann, February 2000.

[10] K. Morneault, R. Dantu, G. Sidebottom, B. Bidulock, J. Heitz, Request for Comments: 3331 Signaling System 7 (SS7) Message Transfer Part 2 (MTP2) – User Adaptation Layer, September 2002.

[11] OpenSSL project, openssl-0.9.7.c.tar.gz at www.openssl.org/source, 2003 09 30.

[12] J. Postel, Request for Comments: 0791 Internet Protocol, 1981 10 01.

[13] J. Postel, Request for Comments: 0793 Transmission Control Protocol, 1981 10 01.

[14] B. Ramsdell, Ed., Request for Comments: 2632 S/MIME Version 3 Certificate Handling, June 1999.

[15] B. Ramsdell, Ed., Request for Comments: 2633 S/MIME Version 3 Message Specification, June 1999.

[16] SCTP IETF IETF RFC 2960 Functional Specification – API, Ericsson 155 19-CAA 901 548 Uen, 2003 06 25.

[17] G. Sidebottom, Ed., K. Morneault, Ed., J. Pastor-Balbas, Ed, Request for Comments: 3332 Signaling System 7 (SS7) Message Transfer Part 3 (MTP3) – User Adaptation Layer (M3UA), September 2002.

[18] Gustavus J. Simmons, Contemporary Cryptology: The Science of Information Integrity, Wiley-IEEE Computer Society Pr; 1999 01 27.

[19] William Stallings, Network Security Essentials: Applications and Standards, Source, 2000.

[20] W. Richard Stevens, Unix Network Programming, volume one, second edition, 1998.

[21] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, V. Paxson, Request for Comments: 2960 Stream Control Transmission Protocol, October 2000.

[22] Stream Control Transmission Protocol (SCTP), International Engineering Consortium 2003, http://www.iec.org/online/tutorials/sctp/index.html, 2004 02 18.

[23] R. Thayer, N. Doraswamy, R. Glenn, Request for Comments: 2411 IP Security Document Roadmap, November 1998.

[24] T. Ts'o, Request for Comments: 2942 Telnet Authentication: Kerberos Version 5, September 2000.

A Key term dictionary

This directory explains some of the words used in this document. The words are explained based on their meaning in this specific document. The explanations may not always be applicable in other contexts.

API	Application Programming Interface.
association	A connection between two SCTP end-points. In one association there may be many SCTP streams.
authentication	Authentication enables the sender and the receiver to verify each other's identity.
blind attack	An attack on an end-point. The attacker sends forged messages (with faked source address) to the attacked endpoint. The attack is blind if the attacker is not able to receive any responses from the attacked end-point. The attacker must guess what kind of messages to send to the attacked end-point.
block cipher	A block cipher is an encryption algorithm that takes a block of plaintext elements at a time and produces a block of ciphertext. A block cipher allows rearrangement of elements within the block.
call chain	A sequence of function calls where the first call starts the second call and so on.
certificate	A certificate consists of a public key and a user identification number. A certificate enables an end-point to verify the source of a public key.
certificate authority	A company that publishes certificates.
chunk	A piece of SCTP data or control information.

ciphertext	An encrypted message.
client	The end-point that initiates the communication between itself and another end-point.
compression	Reducing the size of a message (or any other data) in a way that is reversible.
confidentiality	Confidentiality is achieved when messages are kept unreadable for every one but the intended receiver.
congestion control	Congestion control is used by SCTP to control the speed at which messages are sent. If the network is congested (carries too much traffic) the sending speed is reduced.
connection state	The working environment of the TLS record protocol. The connection state contains the encryption keys etc.
cookie	A file containing information about an SCTP end-point.
CP	Common Parts. A software module developed by TietoEnator. The most important task of CP is to hide platform dependencies. CP gives programmers a good base to make platform independent network programs for TietoEnator's SS7 environment.
decryption	The reverse of encryption.
denial of service attack	The goal of this attack is to prevent legitimate users from using some service provided. This is usually done by attacking the machine(s) providing this service. A common approach is to consume all resources in the machine or simply to make the machine crash.
digital signing	A method used to verify the source of a message.
encryption	The process of replacing and rearranging elements in a message.

end-point	A computer connected to a network. Sometimes the word end-point is used in the sense of a process or a certain protocol on that computer.
fragmentation	The process of dividing a message into smaller parts before sending it.
frame	The messages at the link layer are called frames.
header	A piece of a message that among other things contains the destination address and the source address.
IP	Internet Protocol. The most common protocol used in the network layer.
IP-address	An address to a network layer node.
key	An input parameter to an encryption algorithm.
MAC	Message Authentication Code. MAC calculations are used to discover if a message has been altered during its transmission.
Makefile	A file that manages the compilation and linking of a program.
master secret	A combination of symbols used to derive encryption keys.
message integrity	Message Integrity is achieved when the receiver of a message is able to detect if the message has been changed during transmission.
multi-homing	Multi-homing is a SCTP property that allows the SCTP user to specify and prioritize multiple destination IP-addresses for the same end-point. If there is failure on the path to the first IP-address messages will be delivered to the second one (see path selection).
node	E.g. a switch, a router or an end-point.

openSSL	A software that implements the TLS protocol. The code is available for anyone to use and modify.
path	A specific combination of links and nodes in a network.
path selection	SCTP path selection handles the choosing of an appropriate destination IP-address (path) among those addresses specified by the SCTP user (see multi-homing). The selection of another IP-address is made if failure occurs on the path to the currently used address.
peer	See end-point
pending state	E.g. encryption keys that will be used when the TLS security parameters are renegotiated.
plaintext	A message that has not yet been encrypted or a message that has been encrypted and decrypted back to the original message.
protocol	A set of rules for data communication between two end-points.
protocol stack	A number of protocols used together.
public-key encryption	A method commonly used to distribute the secret keys used for secret-key encryption.
receiver window	SCTP uses a receiver window to keep track of how much memory space there is available for incoming messages.
SCTP	Stream Control Transmission Protocol. A transport layer protocol that provides multi-homing, path selection, independent data transmission in different streams etc.
secret-key encryption	A method used to provide confidentiality.
sequence number	A number identifying a message or the order in which a message should be delivered.

server	The end-point that does not initiate the communication between itself and another end-point.
session	An association (not in the SCTP sense) between two TLS end-points. See also session state.
session state	A session state is the compression method, the algorithms with belonging keys used for encryption and MAC calculations, certificates and a few other things needed to exchange data between two SCTP end-points.
socket	A file descriptor used to send and receive messages over a network using the socket API.
socket API	Standard network functions available to C/C++ programmers in Unix. The most important functions are ‘socket’, ‘bind’, ‘listen’, ‘connect’, ‘accept’, ‘read’, ‘write’, ‘close’ and ‘shutdown’
socket call	A call to one of the functions in the socket API.
SS7	Signaling System Nr 7. A protocol stack used in telephony networks.
SSL	Secure Socket Layer. SSL is an older version of TLS.
stack	See protocol stack.
stream	A stream is transferring the data between two SCTP end-points. There can be many streams in one SCTP association.
stream cipher	A stream cipher is an encryption algorithm that takes one plaintext element at a time and produces a ciphertext element. A stream cipher does not allow rearrangement of elements.

suite	The term suite has two meanings in this document. A protocol suite or a layer suite is simply the interface of the protocol or layer. A protocol or layer has two interfaces, one to the layer above and one to the layer below. A cipher suite is a set of security algorithms.
TCP	Transmission Control Protocol. A protocol at the transport layer. The most important features of TCP are that it provides reliable data transfer and congestion control.
TDM	Time Division Multiplexing. A method used to let multiple users share a data communication link. Each user gets a time slot during which it may send data.
TLS	Transport Layer Security. A protocol between the transport layer and the application layer. This protocol provides confidentiality, message integrity and authentication between client and server.
Unix	An operating system (platform).

B TLS

B.1 Session state

The session state contains the following parameters:

- Session ID: arbitrary byte sequence that identifies the session.
- Peer certificate.
- Compression method: The algorithm used to compress data.
- Cipher specification: One algorithm used to encrypt data and another one used for calculation of the MAC (used for checking the integrity of a message).
- Master secret: A secret (i.e. a combination of symbols) shared by client and server. This secret is used to generate keys for encryption and MAC calculations.
- Is resumable: A flag indicating weather a new connection can be established under the same session.

B.2 Connection state

The connection state defines the following parameters:

- Compression state: Information needed for the compression algorithm.
- Cipher state: The secret key used for encryption. It is generated from the master secret in the session state.
- MAC secret: The secret used for MAC calculations. It is also generated from the master secret in the session state.
- Sequence number: Each end-point maintains sequence numbers for sent and received messages. When the cipher spec is changed these sequence numbers must be set to zero.

B.3 TLS record protocol header

The TLS record protocol header consists of the following fields:

- Content type: This field specifies the higher-layer protocol. The enclosed fragment (i.e. the data) is delivered to this higher protocol.

- Version: The TLS protocol version.
- Compressed length: The length of the fragment (including the MAC).

B.4 TLS handshake protocol messages

The following messages are used by the handshake protocol to negotiate the security parameters:

- Hello request: The server sends a hello request message to notify the client that it should start the negotiation. The server may send this message at any time. If the client is currently negotiating a session this message will be ignored. If the negotiation is completed the client may respond with a client hello or ignore the hello request. If the client ignores the hello request the server will close the connection.
- Client hello: The client sends this message in order to start the negotiation. The client hello contains:
    - Client version: The highest TLS version supported by the client.
    - Random: A client generated random structure that is used when deriving the master secret.
    - Session ID: The ID of the session, which the client wants to use for this connection. If the client wants to establish a new session this field must be empty.
    - Cipher suite: A list of the cryptographic algorithms supported by the client. Each element in this list contains a MAC algorithm and a secret key encryption algorithm.
    - Compression method: A list of compression methods supported by the client.
- Server hello: The server sends this message as a response to a client hello. By sending this message the server accepts a set of algorithms suggested by the client.
    - Server version: The highest TLS version supported by the server.
    - Random: A server generated random structure.
    - Session ID: The session ID or a new session ID if the client left this field empty.
    - Cipher suite: The MAC- and secret key encryption algorithm selected by the server (among those suggested in the client cipher suite).

    - Compression method: The compression method selected by the server (among those suggested by the client).

- Server certificate: This message always follows the server hello message if the key exchange method requires a certificate.
- Server key exchange message: This message contains information used for public key encryption (secret key distribution). This message will be sent after the server certificate message or after server hello message if the sender is anonymous (i.e. if certificates are not used). If this message is sent or not depends on the key exchange method. With some methods, the server certificate message contains enough information and this message does not need to be sent.
- Certificate request: If the client is anonymous to the server, the server will send this message in order to certificate the client. If it is sent it follows the server key exchange message.
- Server hello done: This message is sent to indicate that the server is done sending its hello-associated messages.
- Client certificate: This message will be sent if the server requests a certificate from the client. This message can only be sent after the server hello done message.
- Client key exchange message: This message is used to change the keys used for encryption and MAC calculations. The message causes the pre master secret (i.e. the master secret in the pending state) to become the new master secret (in the current state).
- Certificate verify: The client sends this message only if the client has sent its certificate message earlier. The server can verify the source of the earlier messages using the certificate verify message.
- Finished: This message is sent to indicate that the key exchange and authentication succeeded. The client will send the finished message immediately after a change cipher spec message. The server will then sends its finished message. The finished message is encrypted using the new security parameters set by the change cipher spec message.

B.5 Alerts

Fatal alerts are:

- unexpected_message

- bad_record_mac
- decryption_failed
- record_overflow
- decompression_failure
- handshake_failure
- illegal_parameter
- unknown_ca
- access_denied
- decode_error
- export_restriction
- protocol_version
- insufficient_security
- internal_error.

Other alerts are:

- no_certificate_RESERVED
- bad_certificate
- unsupported_certificate
- certificate_revoked
- certificate_expired
- certificate_unknown
- decrypt_error
- user_cancelled (generally a warning)
- no_re_negotiation (always a warning)

C File and folder structure of openSSL

This is the openSSL file and folder structure. It includes the changes needed for the adaptation of openSSL to CP/SCTP. The superscripted numbers before each file or folder name marks the depth of the file or folder relative to the root folder (‘openssl-0.9.7c’). The following color codes are used:

- Added files
- Files with changed functionality (things that have changed are written in parenthesis)
- Very interesting files
- Interesting files (interesting functions are written in parenthesis)
- Directories

- ^{0} openssl-0.9.7c
    - ^{1} config
    - ^{1} configure
    - ^{1} e_os.h (macros for recv, send, shutdown, socket)
    - ^{1} e_os2.h
    - ^{1} INSTALL.com
    - ^{1} INSTALL.DJGPP
    - ^{1} INSTALL.MacOS
    - ^{1} INSTALL.OS2
    - ^{1} INSTALL.VMS
    - ^{1} INSTALL.W32
    - ^{1} INSTALL.WCE
    - ^{1} Makefile (added TLSoverSCTP library)
    - ^{1} Makefile.org
    - ^{1} Makefile.ssl
    - ^{1} Makefile.ssl.bak
    - ^{1} makevms.com
    - ^{1} openssl.doxy
    - ^{1} openssl.spec
    - ^{1} README
    - ^{1} README.ASN1
    - ^{1} README.ENGINE
    - ^{1} apps
        - ^{2} s_socket.c (socket calls: accept, bind, connect, listen, setsockopt, socket)
        - ^{2} s_server.c
        - ^{2} s_client.c
        - ^{2} app_rand.c
        - ^{2} Makefile
        - ^{2} Makefile.ssl

```
        ²democA
                ³private
        ²set
¹bin
¹bugs
¹certs
        ²expired
¹crypto
        ²aes
        ²asn1
        ²bf
                ³asm
        ²bio
                ³b_sock.c (socket calls: accept, bind, connect, getsockopt, listen,
                 setsockopt, socket)
                ³bss_acpt.c (socket calls: shutdown, socket)
                ³bss_conn.c (socket calls: connect, setsockopt, socket)
                ³bss_sock.c (socket calls: read (readsocket), write (writesocket))
                ³Makefile
                ³Makefile.ssl
        ²bn
                ³asm
                        ⁴alpha
                        ⁴alpha.works
                        ⁴x86
        ²buffer
        ²cast
                ³asm
        ²comp
        ²conf
        ²des
                ³asm
                ³t
                ³times
        ²dh
        ²dsa
        ²dso
        ²ec
        ²engine
                ³vendor_defns
        ²err
        ²evp
        ²hmac
        ²idea
        ²krb5
        ²lhash
        ²md2
        ²md4
        ²md5
                ³asm
```

- 2 mdc2
- 2 objects
- 2 ocsp
- 2 pem
- 2 perlasm
- 2 pkcs12
- 2 pkcs7
  - 3 p7
  - 3 t
- 2 rand
  - 3 rand_egd.c (socket calls: connect, socket)
  - 3 rand_unix.c
  - 3 Makefile
  - 3 Makefile.ssl
- 2 rc2
- 2 rc4
  - 3 asm
- 2 rc5
- 2 rc5
  - 3 asm
- 2 ripemd
  - 3 asm
- 2 rsa
- 2 sha
  - 3 asm
- 2 stack
- 2 threads
  - 3 mttest.c
- 2 txt_db
- 2 ui
- 2 x509
  - 3 x509_lu.c (socket calls: shutdown)
  - 3 Makefile
  - 3 Makefile.ssl
- 2 x509v3

1 demos
- 2 asn1
- 2 bio
- 2 easy_tls
- 2 eay
- 2 engines
  - 3 cluster_labs
  - 3 ibmca
  - 3 rsaref
  - 3 zencod
- 2 maurice
- 2 pkcs12
- 2 prime
- 2 sign
- 2 ssl

```
        2 state_machine
        2 tunala
        2 x509
1 doc
        2 apps
        2 crypto
        2 HOWTO
        2 ssl
1 include
        2 openssl
1 MacOS
        2 GetHTTPS.src
1 mx
1 os2
1 perl
1 shlib
1 ssl
        2 bio_ssl.c
        2 ssl_lib.c  (SSL_new,  SSL_accept,  SSL_connect,  SLL_read,  SSL_write,
          SSL_set_bio, SSL_set_fd)
        2 ssl_algs.c (SSL_library_init)
        2 Makefile
        2 Makefile.ssl
1 test
        2 ssltest.c
        2 Makefile
        2 Makefile.ssl
1 times
        2 090
        2 091
        2 x86
1 tools
1 util
        2 pl
1 VMS
1 man
        2 man1
        2 man3
        2 man5
        2 man7
1 bin
1 lib
        2 pkgconfig
1 misc
1 private
1 TLSoverSCTP
1 demos
1 doc
1 include
1 lib
```

- [1] macOS
- [1] man
- [1] misc
- [1] ms
- [1] os2
- [1] perl
- [1] private
- [1] shlibs
- [1] ssl
- [1] test
- [1] times
- [1] TLSoverSCTP
    - [2] TLSoverSCTP.h
    - [2] TLSoverSCTP.c
    - [2] Makefile
    - [2] Makefile.ssl