Diploma Project

September 7, 2000 - October 30, 2000

Mini Web Server supporting SSL

Implementation for IPC@CHIP

A. Zingg
B. Lenzlinger

Table of Contents

1 Abstract

1.1 English Abstract

IPC@CHIP is a matchbox-sized single chip computer (80186, 20 MHz, 256 kilobytes of flash memory for programs, 512 kilobytes of RAM) featuring an integrated Ethernet interface and 24-volt IO ports. Hence it is possible to connect appliances to the Internet, to control them remotely, and to transfer data. Consequently, security issues become of major importance. Access must be restricted to authorized personnel, and the entire transmission has to be encrypted.

This is precisely the goal of our diploma project: Development of a web server capable of encrypting the entire data transmission using SSL, despite the lack of powerful hardware.

The first task consisted of the porting of the freely available software OpenSSL to the IPC@CHIP platform. Because of the limited hardware this task turned out to be very complex and time-consuming. The original OpenSSL library (larger than 1.5 megabyte) had to be reduced to a few hundred kilobytes. The entire ASN.1 and X.509 part of the library was removed because we implemented our own functions for handling certificate and private key. To reduce code size a special format was developed for the storage of the private key. In addition, a software tool was developed to convert private keys from the old format into the new one.

Because of License Agreements the already existing IPC@CHIP web server could not be equipped with SSL functionality. Therefore we developed our own web server with CGI support: Mini Webserver. Due to lack of space no client authentication was implemented, and so the user's identity must be checked in a CGI program (login name and password are transmitted securely). As the CGI interface is fully compatible with the IPC@CHIP web server, all present CGI programs written for IPC@CHIP are executable on Mini Webserver.

The result of our diploma project is a tiny (101 kilobytes), fully functional, CGI-capable web server supporting SSL handshake and encoding. The used cipher suite (RSA with 1024-bit encryption, RC4 with 128-bit encryption) is nowadays considered to be very safe and is widely used. Thus no more restrictions are set on the usage of IPC@CHIP, not even for safety-relevant applications.

Winterthur, October 30, 2000

Bernhard Lenzlinger Andreas Zingg

1.2 Deutsche Zusammenfassung

Der IPC@CHIP von Beck ist ein Minicomputer (80186, 20 MHz, 256 Kilobyte Flash-Speicher für Programme, 512 Kilobyte RAM) in der Grösse einer Streichholzschachtel mit integrierter Ethernet-Schnittstelle und frei ansteuerbaren 24-Volt Ausgängen. Damit lassen sich über das Internet weltweit Geräte und Maschinen ansteuern und Daten transferieren. Diese Daten, vor allem aber der Zugriff auf die Geräte, muss Berechtigten vorbehalten sein.

Genau an diesem Punkt setzt die Aufgabe unserer Diplomarbeit ein. Ziel ist es, trotz beschränkter Hardware-Bedingungen einen Webserver zu realisieren, der es ermöglicht, den gesamten Datenverkehr mit SSL zu verschlüsseln.

Zuerst musste die frei erhältliche Software OpenSSL auf den IPC@CHIP portiert werden. Wegen der Hardwareumgebung stellte sich dieser Punkt als sehr aufwendig und zeitraubend heraus, denn die ursprüngliche OpenSSL-Library (grösser als 1,5 Megabyte) musste auf wenige hundert Kilobyte verkleinert werden. Da eigene Funktionen das Handling von Zertifikat und privatem Schlüssel übernehmen, konnten die gesamten ASN.1- und X.509-Funktionen weggelassen werden. Zur Unterstützung dieser Funktionen und zur Reduzierung der Codegrösse wurde ein spezielles Format für die Speicherung des privaten Schlüssels entwickelt. Dazu gehört selbstverständlich auch ein Werkzeug zur Konvertierung des bisherigen Formates in das neu entwickelte.

Da der schon existierende Webserver des IPC@CHIP aus lizenzrechtlichen Gründen nicht mit der SSL-Funktionalität ausgestattet werden konnte, entwickelten wir von Grund auf einen eigenen Webserver mit CGI-Unterstützung. Aus Platzgründen konnte keine Client-Authentifizierung implementiert werden, weshalb die Identität des Benutzers (Login und Passwort werden verschlüsselt übermittelt) in einem CGI-Programm überprüft werden muss. Bei erwähntem CGI-Interface wurde auf vollständige Kompatibilität zum IPC-Webserver geachtet, so dass alle bisherigen CGI-Programme auch auf unserem Webserver lauffähig sind.

Das Produkt unserer Diplomarbeit ist ein winziger (101 Kilobyte), voll funktionsfähiger Webserver mit SSL-Verschlüsselung und CGI-Unterstützung. Die verwendeten Verschlüsselungsalgorithmen (RSA mit 1024-Bit-Schlüssel, RC4 mit 128-Bit-Schlüssel) gelten heute als sehr sicher und sind weit verbreitet. Damit sind dem Einsatz des IPC@CHIP auch für sicherheitsrelevante Anwendungen keine Grenzen mehr gesetzt.

Winterthur, 30. Oktober 2000

Bernhard Lenzlinger Andreas Zingg

2 Task Description

For the original task description, written in German, please refer to appendix D.

Mini Webserver supporting SSL

Description:

Using the IPC@CHIP by Beck it is possible to connect a web server to the Internet for little money. This single chip computer has direct access to 24-volt IO ports. Hence it is possible to connect machines and appliances to the Internet and control them remotely. Consequently, security issues become of major importance. Access must be restricted to authorized people, and the entire transmission has to be encrypted.

The standard SSL protocol is capable of these requirements.

The goal of this project is the porting of the source code of the OpenSSL library to the IPC@CHIP in order to run a mini web server supporting SSL handshake and encryption.

Tasks:

- Definition of the minimal part of the OpenSSL library to suit the following requirements:
  - Support of SSL version 3
  - No client authentication
  - Strong encryption: RC4 with 128 bit key size
  - MD5 MAC for authentication
- Porting of the SSL stack to the IPC@CHIP environment
- Development of a simple https server listening on port 443. The user is prompted for login name and password. After correct login the user is given a page that shows statistics about the processor state, or that enables the user to set and reset the IPC@CHIP's IO ports.
- Writing of an installation manual and a user's guide for the SSL stack and the mini web server.
- Documentation of the project.

3 Introduction

This thesis is structured as follows:

- **Chapter 1: Abstract**
  Description of the results that were accomplished at the end of the diploma project.

- **Chapter 2: Task Description**
  Description of the tasks of the diploma project.

- **Chapter 3: Introduction**
  Introduction to all used technologies that are required for understanding the approach to the project. Overview of IPC@CHIP, SSL, and X.509 certificates.

- **Chapter 4: Development Environment**
  Short description of the hardware and software environment used to implement SSL for the IPC@CHIP. The setup of those is described as well.

- **Chapter 5: Approach**
  Approach to the project in chronological order.

- **Chapter 6: Report**
  Detailed report of the work done, ordered by topics.

- **Chapter 7: Manual - SSL for IPC@CHIP**
  Installation Manual and Programmer's Guide for *SSL for IPC@CHIP*.

- **Chapter 8: Conclusion & Prospects**
  A look back onto the project, and a view into the future of *SSL for IPC@CHIP*.

- **Chapter 9: Glossary**
  Short explanation of terms used throughout the thesis.

- **Appendices**
  Bibliography, Configuration, Links, Certificates.

3.1 Overview IPC@CHIP

The development of the Industrial PC (IPC) nowadays shows two tendencies.
On the one hand the IPC is being developed into a more and more complex system. Therefore the system not only increases in features but also in size and costs. The goal of this approach is that controlling, positioning, visualization, and data management can all be handled by a single system.
On the other hand there is the miniaturization of the IPC. In order to accomplish minimal space, costs and power consumption, this IPC lacks many features that are typical for a "grown-up" PC or a complex IPC (mentioned above). It has neither monitor, nor keyboard, nor hard disk. Instead, the BIOS is constructed in a way that the first serial port acts as a terminal connector. Hence, an external system is necessary to develop applications and load them to the IPC. The IPC itself, though, is of very compact size and of little cost (around 100 DM).

3.1.1 General Description

The IPC@CHIP of Beck–IPC GmbH [5] is of the latter category described in the previous section ("the smaller the better"). As it is a single chip solution, there is no need for external RAM or Flash memory. The hardware includes the micro processor (Intel 80186), 512 kilobytes of RAM, 512 kilobytes of flash memory, built-in Ethernet, two serial ports, watchdog, data/address bus, programmable I/O and power-fail detection. It is the fastest and easiest solution to make an appliance fit for the Internet. As a rule tiny IPC's do not feature great software support. The IPC@CHIP is an exception to this rule: it already includes the most widely used internet functions:

- **Webserver supporting CGI**
  The integrated CGI-capable web server can be used to supply a web-interface to any device. The user can now use his Internet browser, handheld or even mobile phone not only to monitor data (status, temperature, ...), but also to control the device.
- **FTP server**
  An integrated FTP server loads applications, graphics and other files to the chip or retrieves log files.
- **Modem Connection**
  Connection of a modem to the integrated PPP server is possible without problems.
  In addition to the PPP server the IPC@CHIP features an integrated PPP client to automatically connect to the internet whenever the device needs to send emails.
- **Telnet**
  The integrated telnet server provides a console connection to the IPC@CHIP over the internet.

- **Real Time Operating System**
  Underneath the aforementioned services, a real time operating system is running to allow simultaneous use of all of these services. The operating system is DOS-like, the mere difference is the ability to run tasks simultaneously. An application, therefore, has to be written as a 16-bit DOS application and is started as a task. Applications can handle CGI requests, control the device, communicate to other devices or send emails when necessary.
- **TCP / IP**
  The TCP-IP stack is *the* means of communication to the rest of the world. Available are 64 sockets (TCP and UDP).

As a summary, the IPC@CHIP enables a developer to program applications for almost every Internet communication and data transmission facility.

3.1.2 Technical Features

CPU	`Intel 80186`
Operating Frequency	`20 MHz`
Bus Width	`16 Bit`
RAM Size	`512 KByte`
Flash Memory Size	`512 KByte`
IO Space	`6 x 256 Byte`
DMA Channels	`2`
User Programmable IO Ports	`14`
User Programmable Timer	`3 x 16 Bit`
User Programmable Interrupts Extern	`6`
User Programmable Interrupts Intern	`8`
Ethernet Connection	`1 x 10BaseT`
Serial Ports	`2`
Package	`DIL32`

Table 3.1: Technical Features of the IPC@CHIP

For more details about the IPC@CHIP, refer to [5], or to the enclosed CD-ROM (`[CDROM]/beck/documentation`).

3.2 Overview Secure Sockets Layer

For this chapter it is assumed that you are familiar with the basic concepts of public-key cryptography, as summarized in the document [4].

This chapter describes the Secure Sockets Layer (SSL) protocol, a security protocol that provides privacy over the Internet. The protocol allows client/server applications to communicate in a way that cannot be eavesdropped. Server's are always authenticated, clients can optionally be authenticated.

3.2.1 Motivation

The Transmission Control Protocol/Internet Protocol (TCP/IP) controls the transport and routing of data over the Internet. Other protocols, such as the HyperText Transport Protocol (HTTP), Lightweight Directory Access Protocol (LDAP), or Internet Messaging Access Protocol (IMAP), run "on top" of TCP/IP in the sense that they all use TCP/IP for data transmission.
The SSL Protocol is designed to provide privacy between two applications (a client and a server). Second, the protocol is designed to authenticate the server, and optionally authenticate the client. SSL runs above the TCP/IP protocol because it requires a reliable transport protocol for data transmission and reception, and below higher-level protocols such as HTTP, FTP or TELNET. Figure 3.1 shows the location of the SSL protocol between the application layer and network layer.

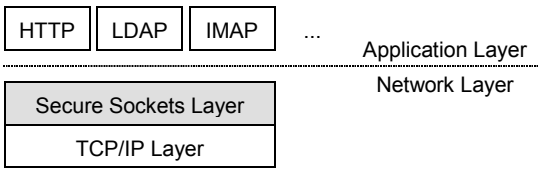


Figure 3.1: SSL is a Separate Protocol Layer for Security Only

The advantage of the SSL protocol is that it is application protocol independent. A higher level application protocol can layer on top of the SSL Protocol transparently. The SSL Protocol can negotiate an encryption algorithm and session key as well as authenticate a server and client before the application protocol transmits or receives its first byte of data. All of the application protocol data is transmitted encrypted, ensuring privacy.

The SSL protocol provides "channel security", which has three basic properties:

- The channel is private. Encryption is used for all messages after a simple handshake is used to define a secret session key.
- The channel is authenticated. The server endpoint of the conversation is always authenticated, while the client endpoint is optionally authenticated.
- The channel is reliable. The message transport includes a message integrity check (using a message authentication code (MAC) e.g. MD5 or SHA).

The SSL protocol includes two sub-protocols: the SSL record protocol and the SSL handshake protocol. The SSL record protocol defines the format used to transmit data. The SSL handshake protocol involves using the SSL record protocol to exchange a series of messages between an SSL-enabled server and an SSL-enabled client when they first establish an SSL connection. This exchange of messages is designed to facilitate the following actions:

- Authenticate the server to the client.
- Allow the client and server to select the cryptographic algorithms, or ciphers, that they both support.
- Optionally authenticate the client to the server.
- Use public-key encryption techniques to generate shared secrets.
- Establish an encrypted SSL connection.

For more information about the handshake process, see the following section on the next page.

3.2.2 SSL Handshake Process

The cryptographic parameters of the session state are produced by the SSL handshake protocol, which operates on top of the SSL record layer. When a SSL client and server first start communicating, they agree on a protocol version, select cryptographic algorithms, optionally authenticate each other, and use public-key encryption techniques to generate shared secrets.

These processes are performed in the handshake protocol, which can be summarized as follows:

1. The client sends a **ClientHello** message to which the server must respond with a **ServerHello** message, else a fatal error occurs and the connection fails. The ClientHello and ServerHello are used to establish security enhancement capabilities between client and server. The ClientHello and ServerHello establish the following attributes: Protocol Version, Session ID, Cipher Suite, and Compression Method. Additionally, two random values are generated and exchanged: ClientHello-Random and ServerHello-Random.
2. Following the Hello messages, the server sends its **Certificate** message that includes its certificate, if it is to be authenticated. Additionally, a **ServerKeyExchange** message may be sent, if it is required (e.g. if their server has no certificate, or if its certificate is for signing only). If the server is authenticated, it may request a certificate from the client (**CertificateRequest**), if that is appropriate to the cipher suite selected. Now the server sends the **ServerHelloDone** message, indicating that the hello-message phase of the handshake is complete. The server then waits for the client to respond.

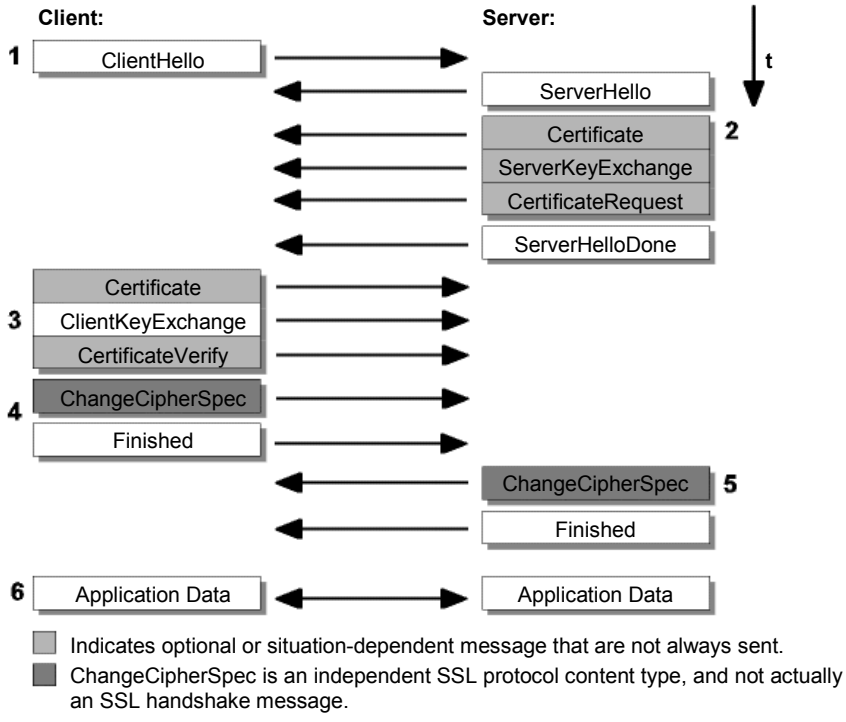


Figure 3.2: The SSL Handshake Process

3. If the server has sent a CertificateRequest message, the client must send either the **Certificate** message or a 'no certificate' alert. The **ClientKeyExchange** message is now sent, and the content of that message will depend on the public key algorithm selected between the ClientHello and the ServerHello. If the selected algorithm is RSA, the client sends an encrypted premaster secret, which afterwards is applied in generating the master secret. If the client has sent a certificate with signing ability, a digitally-signed **CertificateVerify** message is sent to explicitly verify the certificate.
4. At this point, a **ChangeCipherSpec** message is sent by the client, and the client copies the pending Cipher Spec into the current Cipher Spec. The client then immediately sends the **Finished** message using the new algorithms, keys, and secrets.
5. In response, the server will send its own **ChangeCipherSpec** message, transfer the pending to the current Cipher Spec, and send its **Finished** message under the new Cipher Spec.
6. At this point, the handshake is complete and the client and server may begin to exchange application layer data.

Resuming a Previous Session

When the client and server decide to resume a previous session or duplicate an existing session (instead of negotiating new security parameters) the message flow occurs as follows:

1. The client sends a **ClientHello** message using the Session ID of the session to be resumed.
2. The server then checks its session cache for a match. If a match is found, and the server is willing to re-establish the connection under the specified session state, it will send a **ServerHello** message with the same Session ID value.
3. At this point, both client and server must send **ChangeCipherSpec** messages and proceed directly to **Finished** messages.

Once the re-establishment is complete, the client and server may begin to exchange application layer data. If a Session ID match is not found, the server generates a new session ID and the SSL client and server perform a full handshake.

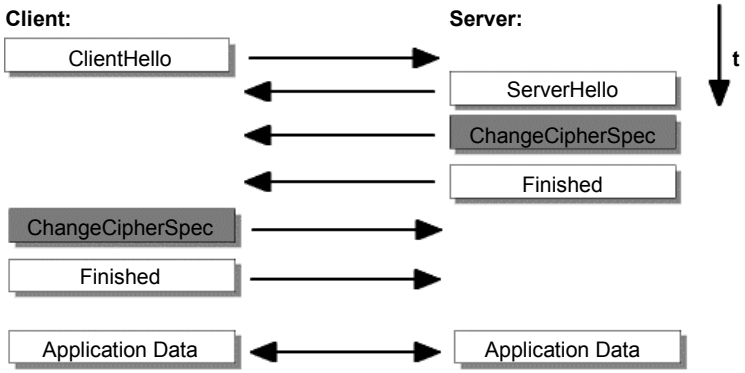


Figure 3.3: Only Six Messages are Required to Resume an SSL Session

3.2.3 SSL Protocol Specification

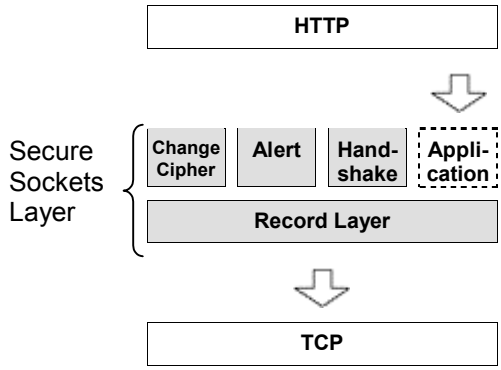


Figure 3.4: SSL Protocol Architecture

The SSL protocol itself consists of several different components organized as figure 3.4 illustrates. Four different sources create SSL messages: the Change Cipher Spec protocol, the Alert protocol, the Handshake protocol, and an application like HTTP. The Record Layer protocol accepts all of these messages. It formats and frames them appropriately, and passes them to a transport layer protocol such as TCP for transmission.

Record Protocol

In SSL, all data sent is encapsulated in a record, an object which is composed of a header and some non-zero amount of data. The record provides a common format to frame alert messages, handshake messages, security escapes and application data transfers. This record is called record layer and it is used by both the client and the server at all times. Figure 3.5 emphasizes the record layer's position in the SSL architecture.

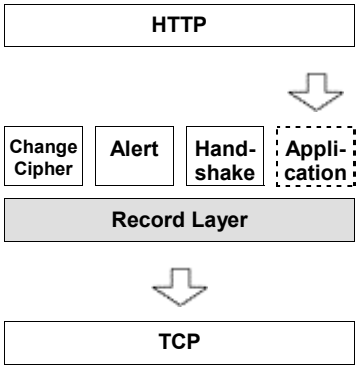


Figure 3.5: All SSL Messages are Formatted by the Record Layer

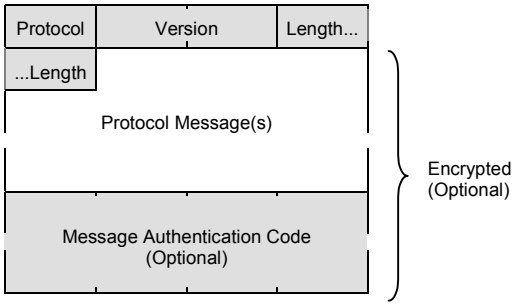


Figure 3.6: Record Layer Encapsulates all Protocol Messages

Figure 3.6 shows the structure of record layer formatting. The record layer formatting consists of 5 bytes that precede other protocol messages and, if message integrity is active, a message authentication code (MAC) at the end of the message. It is also responsible for encryption if that service is active.

The SSL specification defines four different higher-layer protocols (Alert protocol, ChangeCipherSpec protocol, Handshake protocol or Application Data protocol) that the record layer can encapsulate. For any particular message, the protocol field within the record header indicates the specific higher-layer protocol.

Change Cipher Spec Protocol

The Change Cipher Spec protocol exists to signal transitions in ciphering strategies. As figure 3.7 shows, it has the same position in the SSL architecture as other protocols, including Alert, Handshake, and Application Data Protocol. The protocol consists of a single message, which is encrypted and compressed under the current (not the pending) CipherSpec.

The Change Cipher Spec message is sent by both the client and server to notify the receiving party that subsequent records will be protected under the just-negotiated CipherSpec and keys. Reception of this message causes the receiver to copy the read pending state into the read current state. The client sends a ChangeCipherSpec message following KeyExchange and CertificateVerify messages (if any), and the server sends one after successfully processing the KeyExchange message it received from the client. An unexpected ChangeCipherSpec message should generate an UnexpectedMessage alert. When resuming a previous session, the ChangeCipherSpec message is sent after the Hello messages.

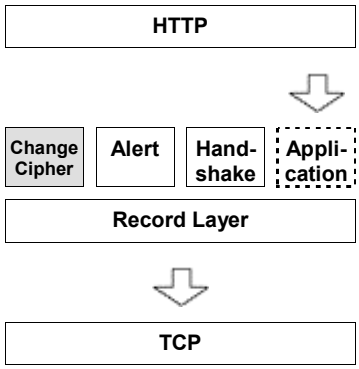


Figure 3.7: ChangeCipherSpec Messages are a Separate Protocol

Alert Protocol

Systems use the alert protocol to signal an error or caution condition to the other party in their communication. This function is important enough to warrant ist own protocol. As figure 3.8 illustrates, the Alert protocol, like all SSL protocols, uses the record layer to format its messages.

Alert handling in the SSL connection protocol is very simple. When an error is detected, the detecting party sends a message to the other party. Alerts can either be warnings or fatal. Fatal alerts represent significant problems with the communication, and cause the client and server to abort the secure connection. Warnings are not quite so drastic. A system receiving such an alert may decide to allow the present session to continue. However, both parties are required to "forget" any session-identifiers associated with a failed connection.

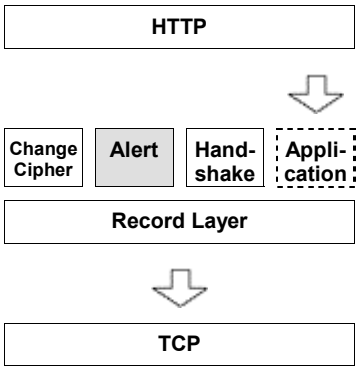


Figure 3.8: The Alert Protocol Signals Error Conditions

Handshake Protocol

As figure 3.9 illustrates, the Handshake protocol messages are encapsulated in the Record protocol. They are composed of two parts: a single byte message type code, and some data. The client and server exchange messages until both ends have sent their Finished message. That indicates that they are satisfied with the Handshake protocol conversation. While one end may be finished, the other may not, therefore the finished end must continue to receive Handshake protocol messages until it too receives a Finished message. For more detailed information about the handshake process refer to the previous section.

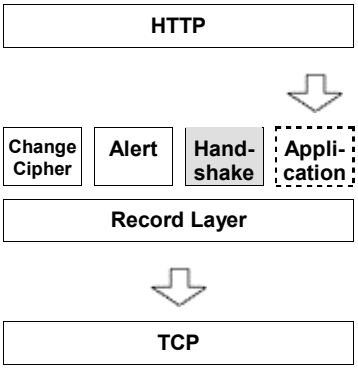


Figure 3.9: The Handshake Protocol Handles Session Negotiation

Handshake Protocol Types
HelloRequest
ClientHello
ServerHello
Certificate
ServerKeyExchange
CertificateRequest
ServerHelloDone
CertificateVerify
ClientKeyExchange
Finished

Table 3.2: Handshake Protocol Types

Table 3.2 lists the messages that the handshake protocol consists of:

Application Data Protocol

Application Data messages are carried by the record layer and are fragmented, compressed and encrypted based on the current connection state. The messages are treated as transparent data to the record layer.

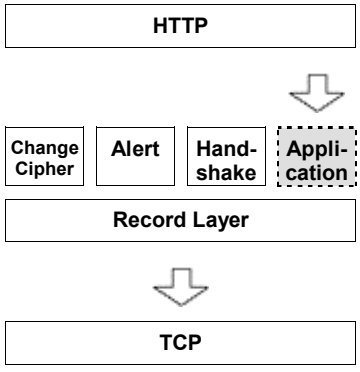


Figure 3.10: Applications Use the Record Layer directly

3.2.4 Creating Cryptographic Parameters

The encryption and message authentication code algorithms of SSL rely on a collection of secret information known only to the communicating parties. Establishing that information securely is one of the three major purposes of the SSL handshake (the other two are authenticating identity and negotiating cipher suites).

Master Secret

The starting point for all the shared secret information is the master secret. The master secret is, in turn, based on the premaster secret. When RSA is used for server authentication and key exchange, the 48-byte premaster secret is generated by the client, encrypted with the server's public key, and sent to the server. The server uses its private key to decrypt the premaster secret. Both parties then know the premaster secret and input it, along with the hexadecimal value of characters in ASCII code and the random values each chose for its Hello message, into secure hash functions. From now on both systems will have the same master secret. (The premaster secret should be deleted from memory once the master secret has been computed.)

Figure 3.11 illustrates how each party calculates the master secret from the premaster secret, the ASCII characters and the random values, and figure 3.12 shows the same steps in the form of an equation.

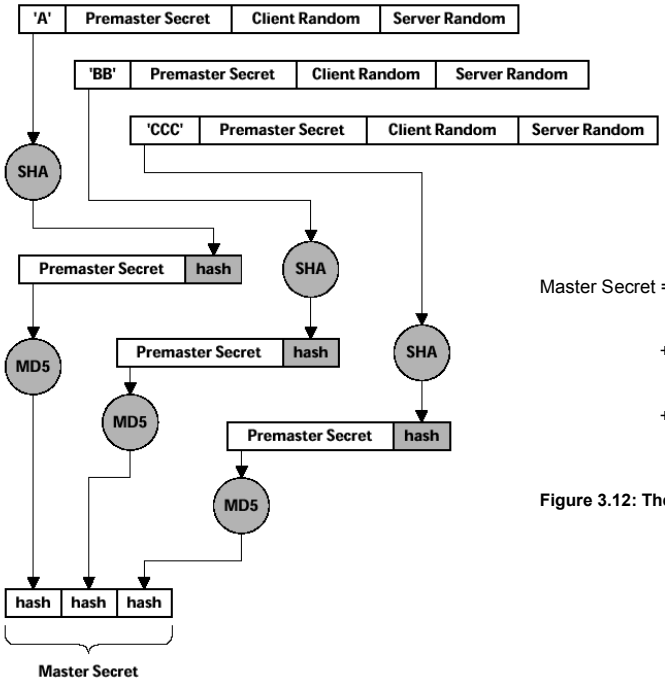


Figure 3.11: SSL uses Hash Functions to Generate the Master Secret

Master Secret =
MD5(Premaster Secret + SHA('A' + Premaster Secret +
ClientHello-Random + ServerHello-Random))
+
MD5(Premaster Secret + SHA('BB' + Premaster Secret +
ClientHello-Random + ServerHello-Random))
+
MD5(Premaster Secret + SHA('CCC' + Premaster Secret +
ClientHello-Random + ServerHello-Random))

Figure 3.12: The Master Secret Requires Six Hash Calculations

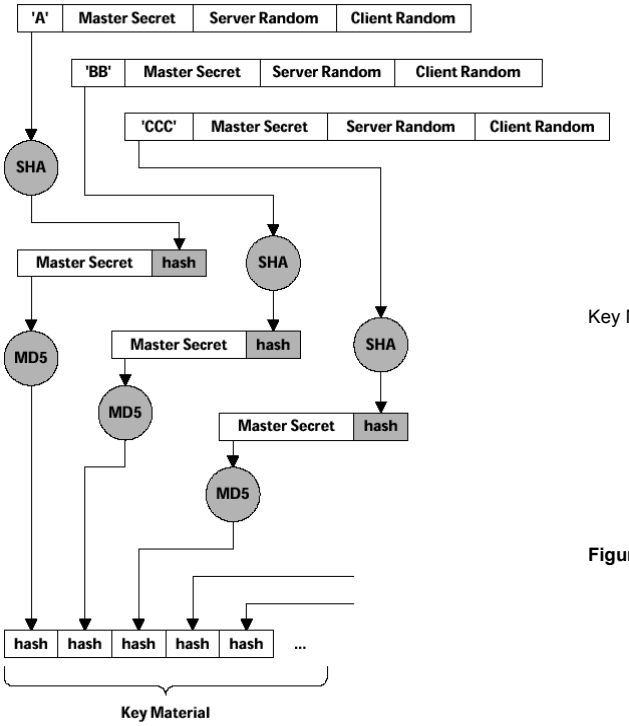
Key Material (Secret Information)

Once each system has calculated the master secret, it is ready to generate the actual secret information needed for the communication. The first step in that process is determining how much secret information is necessary for the key elements. The exact amount depends on the particular cipher suite and parameters that the two parties have negotiated, but generally consists of the information listed in table 3.3. Each party selects the information from that table that is appropriate for the negotiated cipher suite. Then it counts the number of bytes each value requires, based on the negotiated cipher suite parameters. The result is the size of the required secret information.

Parameter	Secret Information
MAC Write Secret Client	The secret value included in the MAC code for messages generated by the client.
MAC Write Secret Server	The secret value included in the MAC code for messages generated by the server.
Secure Write Key Client	The secret key used to encrypt messages generated by the client.
Secure Write Key Server	The secret key used to encrypt messages generated by the server.
Client Write IV	The initialization vector for encryption performed by the client.
Server Write IV	The initialization vector for encryption performed by the server.

Table 3.3: Shared Secret Information

To create shared secret information, both parties use a process very similar to the aforementioned process that calculates the master secret. Figure 3.13 illustrates the approach and figure 3.14 shows the calculation as an equation. They first calculate the SHA hash of the ASCII character ‘A’ followed by the master secret, followed by the server’s random value (from ServerHello), followed by the client’s random (from ClientHello).



Key Material =
MD5(Master Secret + SHA('A' + Master Secret + ClientHello-Random + ServerHello-Random))
+
MD5(Master Secret + SHA('BB' + Master Secret + ClientHello-Random + ServerHello-Random))
+
MD5(Master Secret + SHA('CCC' + Master Secret + ClientHello-Random + ServerHello-Random))
+ ...

Figure 3.14: The Master Secret Seeds Calculations of Key Material

Figure 3.13: The Generation of the Key Material Requires at Least Six Hash

Systems then calculate the MD5 hash from the master secret followed by the result of the intermediate SHA hash. If the resulting 16 byte value is not sufficient for all the secret information, they repeat the process, but with the ASCII characters ‘BB’ instead of ‘A’. The parties continue repeating this calculation (with ‘CCC’, then ‘DDDD’, then ‘EEEEE’, and so forth) as many times as necessary to generate enough secret information. The result is split into the values listed in table 3.3.

3.3 Overview X.509 Certificates

Users of a public key shall be confident that the associated private key is owned by the correct remote subject (person or system) with which an encryption or digital signature mechanism will be used. This confidence is obtained through the use of public key certificates, which are data structures that bind public key values to subjects. The binding is asserted by having a trusted certificate authority (CA) digitally sign each certificate. A certificate has a limited valid lifetime which is indicated in its signed contents. Because a certificate's signature and timeliness can be independently checked by a certificate-using client, certificates can be distributed via untrustworthy communications and server systems. They also can be cached in unsecured storage in certificate-using systems.

One particular international standard is widely accepted as the appropriate format for public key certificates. That standard is from the International Telecommunications Union (ITU). It is universally known by its ITU specification number: X.509. This section presents a short overview of the X.509 standard.

3.3.1 Basic Certificate Fields

Certificates described by the X.509 standard may be signed with any public key signature algorithm. RSA and DSA are the most popular signature algorithms used in the Internet. Signature algorithms are always used in conjunction with a one-way hash function. The signature algorithm and one-way hash function used to sign a certificate is indicated by use of an algorithm identifier which appears in the Signature Algorithm field in a certificate. The contents of the parameter components for each algorithm vary; details are provided for each algorithm.

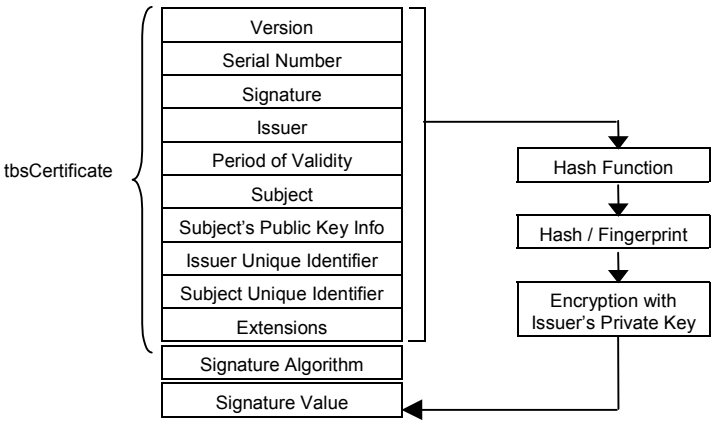


Figure 3.15: General Structure of an X.509 Certificate

The data to be signed (e.g. the one-way hash function output value) is formatted for the signature algorithm to be used. Then, a private key operation (e.g. RSA encryption) is performed to generate the digital

signature. This digital signature is then ASN.1-encoded and included in the certificate in the Signature Value field.

Figure 3.15 illustrates the general structure of an X.509 certificate and the creation of the digital signature. The Certificate is a sequence of three required fields (tbsCertificate, Signature Algorithm and Signature Value). The particular fields are described in detail in the following subsections.

3.3.1.1 TbsCertificate

The field contains the names of the subject and issuer, a public key associated with the subject, a validity period, and other associated information. The fields are described below.

- **Version**
  The Version field identifies the particular version of the X.509 standard to which the certificate conforms. Up to now, the latest version of the X.509 standard is 3.

- **Serial Number**
  The serial number is an integer assigned by the certificate authority (CA) to each certificate. It must be unique for each certificate issued by a given CA.

- **Signature**
  The Signature field identifies the cryptographic algorithm used to sign the certificate, as well as any parameters pertinent to that algorithm. This information is actually repeated in the Signature Algorithm field of the certificate. Most implementations choose to use the information from that section, effectively ignoring this value.

- **Issuer**
  The Issuer field identifies the CA that has signed and issued the certificate. It takes the form of a distinguished name. That is a hierarchy, often starting with a country and then dividing into state or province, organizations, organizational units, and so on. Theoretically, a distinguished name may extend all the way to an individual.

- **Period of Validity**
  The certificate validity identifies both the earliest and latest times that the certificate is valid. Outside of the bounds, the certificate is not to be considered valid.

- **Subject**
  The Subject field identifies the entity that owns the private key being certified. Like the Issuer field, this field takes the form of a distinguished name.

- **Subject's Public Key Info**
  This field contains the subject's public key and identifies the algorithm and its parameters with which the key is used. As an example, if the public key algorithm is

RSA, then this field will contain the modulus and public exponent.

Note: This information is different from the information in the Signature / Signature Algorithm and Signature Value fields of the certificate. Those two fields identify the algorithm of the *CA*'s public key, the key used to sign the certificate. This field identifies the *subject*'s public key.

- **Issuer Unique Identifier**
  This optional field permits two different issuers to have the same Issuer distinguished name. Such issuers would be distinguished from each other by having different values for the Issuer Unique Identifier. In practice, this field is rarely used.

- **Subject Unique Identifier**
  This optional field permits two different subjects to have the same distinguished name. For example, two different people in the same organization might be named John Miller. Such subjects would be distinguished by different values for this field. In practice, this field is rarely used.

- **Extensions**
  The Extensions field provides methods for associating additional attributes with users or public keys and for managing the certification hierarchy. CA frequently use this area for miscellaneous information related to the certificate.

3.3.1.2 Signature Algorithm

The Signature Algorithm field identifies the cryptographic algorithm used by the CA to sign this certificate. This field must contain the same algorithm identifier as the Signature field mentioned above.

3.3.1.3 Signature Value

The Signature Value field contains a digital signature computed upon the DER (Distinguished Encoding Rules) encoded tbsCertificate. The DER encoded tbsCertificate is used as the input to the signature function. This digital signature is ASN.1-encoded and included in the certificate's Signature Value field.

By generating this digital signature, a CA certifies the validity of the information in the tbsCertificate field. In particular, the CA certifies the binding between the public key material and the subject of the certificate.

4 Development Environment

4.1 Infrastructure

4.1.1 Hardware

- PC's: 2 Windows NT (ksy108.zhwin.ch, ksy109.zhwin.ch)
  1 SuSE Linux 6.2 (ksy110.zhwin.ch)
- IPC@CHIP SC12 (Chip)
- IPC@CHIP DK40 (Evaluation Kit)
- IPC PS1 SM14 (Programming Cable)
- Power Supply (No. HF 77.3)
- LAN Analyzer DA-320 (by Wandel & Goltermann) plus appropriate Software (Domino Core 2.4)

4.1.2 Software

- IP address for IPC@CHIP (160.85.134.67)
- DNS entry for IPC@CHIP (mini.zhwin.ch)
- TeraTerm Pro 2.3
  WS-FTP 5.08
  ChipTool 2.2.6.1
  PKLite 2.01 (for compression)
  WinZip 7.0
- Netscape Navigator 4.75 (128 bit encryption)
  MS Internet Explorer 5.5 (128 bit encryption)
- OpenSSL 0.9.5a
  (available on CD-ROM: `[CDROM]\OpenSSL\openssl-0.9.5a.tar.gz`)
- Borland C++ 3.1 Compiler
- MS Visual C++ 6.0

4.2 Setting up IPC@CHIP

The setting up of the IPC@CHIP is described in detail in the file `startup.pdf` (available on the enclosed CD-ROM: `[CDROM]\beck\documentation\setup\startup.pdf`). For this project BIOS version 0.69 Medium was flashed to the chip (see section 4.2.3 on page 22).

4.2.1 Configuration

IPC@CHIP is capable of obtaining the IP configuration from a Dynamic Host Configuration Protocol (DHCP) server. Despite this fact all IP information, such as IP address, standard gateway, and subnet mask, were stored in a file (`chip.ini`) and loaded to the IPC@CHIP's flash memory. An example of `chip.ini` is printed in appendix B. The DNS entry *mini.zhwin.ch* belongs to the IP address *160.85.134.67*.

SC12 Serial Number	`0016F`
MAC Address	`00 30 56 F0 01 6F`
IP Address	`160.85.134.67`
Subnet Mask	`255.255.240.0`
Gateway	`160.85.128.1`

Table 4.1: Configuration of the IPC@CHIP

4.2.2 Tools

Each tool mentioned below is available on the enclosed CD-ROM.

- **TeraTerm Pro**
  TeraTerm Pro was used as the terminal program to communicate with the IPC@CHIP. The correct settings are shown below:

Baud Rate	`19200`
Data Bits	`8`
Parity	`none`
Stop Bits	`1`
Flow Control	`none`

Table 4.2: Correct Settings for TeraTerm Pro

- **WS-FTP Lite**
  Since IPC@CHIP hosts a FTP server, WS-FTP Lite was used for data transfer from and to the IPC@CHIP. The exact login information (login name and password) are stored in the initialization file `chip.ini` and can be altered (see appendix B). If not specified in `chip.ini`, the login is *ftp* and the password *ftp*.

- **ChipTool**
  ChipTool is the tool to configure IPC@CHIP, to update its BIOS and to search for running chips in the local network. The way to perform a BIOS update is described in detail in the next section.
  Mentioned here is the function to search for running chips. Use *Chip* | *Find* to start the search. The result is presented in figure 4.1.

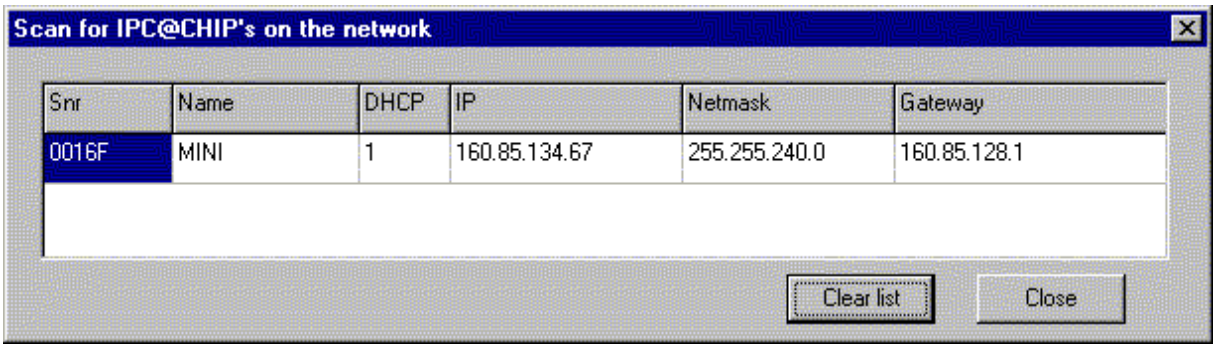


Figure 4.1: Status Information of all IPC@CHIP in the Local Network as indicated by ChipTool

- **PKLite**
  IPC@CHIP has approximately 416 kilobyte of free RAM for user programs (BIOS version 0.69 Medium). In contrast, free space on the flash disk is limited to 256 kilobytes. Obviously, programs could be quite large, according to the RAM size, but they do not fit to the disk size. The solution to this is PKLite. PKLite is a file compression program used to compress executable files to 45 to 50 percent of the original size. It might also reduce that start-up time of many programs because of reduced drive access time. PKLite compresses your files and adds a small amount of extraction code at the beginning of each executable file. When you run an application that has been compressed by PKLite, the program automatically expands into memory and runs. The compression process does not change the operation of the program; it merely reduces the disk space required to store the program. PKLite, in this project, was used as follows:

  ```
  pklite.exe -b serv.exe
  ```

  `serv.exe` is compressed, and a copy of the full-size original is stored as `serv.bak`.

4.2.3 BIOS Update

To perform a BIOS update there are two possibilities; by direct connection to the EXT port of IPC@CHIP, and by TCP connection. The only difference is which button to press. For the direct connection the start button in the “Use RS-232 interface” section must be pressed, for TCP the start button in the “Use TCP” section.
Bios | *Program Flash* initiates the BIOS update. After the flash memory has been programmed, the IPC@CHIP automatically reboots.

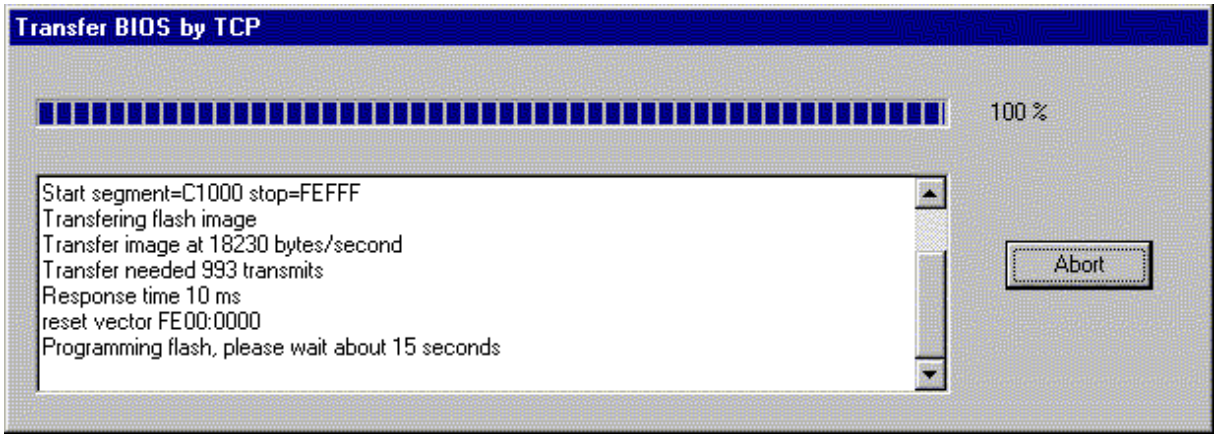


Figure 4.2: BIOS Update to Version 0.69 Medium

4.3 Setting up OpenSSL

The necessary conditions for the installation of OpenSSL are a working C compiler and the scripting language Perl. Perl is installed by default on Linux, whereas on Windows NT it must be installed separately.

4.3.1 Installation on Windows NT

It is assumed that Microsoft’s Visual C++ is installed in the directory `c:\programme\devstudio\vc98`. Should it have been installed at another location, the path in step 3 must be adjusted.

Prior to the installation of OpenSSL the scripting language Perl must be installed on the computer. This is accomplished by running
`[CDROM]\activeperl\nt\instmsi.exe`, followed by
`[CDROM]\activeperl\activeperl-5_6_0_613.msi`.

For installing OpenSSL the file `[CDROM]\openssl\openssl-0.9.5a.tar.gz` must be unpacked into the directory `g:\openssl` using WinZip.

The next steps are as described below:

```
1.  g:\
2.  cd \openssl
3.  set path=c:\programme\devstudio\vc98\bin;%path%
4.  vcvars32
5.  set DIGEST=no-md2 no-mdc2 no-ripemd
6.  set CIPHER=no-rc2 no-rc5 no-idea no-des no-bf no-cast
7.  set PKC=no-das no-dh
8.  set SSLVER=no-ssl2
9.  set OPTS=no-asm
10. perl Configure BC-16
11. perl util\mkfiles.pl > MINFO
12. perl util\mk1mf.pl %DIGEST% %CIPHER% %PKC% %SSLVER% %OPTS%
    BC-MSDOS > ms\msdosbc.mak
```

For the various options refer to the file `Install` in `g:\openssl` .

IMPORTANT: This is not a full installation but rather the initial start position for this project. From this point on the generated make file (msdosbc.mak) and several C source files and header files need modification and adjustment in order to produce a working SSL implementation for the IPC@CHIP. For the detailed approach refer to chapter 5 and 6.

4.3.2 Installation on Linux

The installation of OpenSSL on Linux is straightforward. Just follow these instructions:

```
1. cp [CDROM]/openssl/openssl-0.9.5a.tar.gz /usr/local
2. cd /usr/local
3. tar –xzf openssl-0.9.5a.tar.gz
4. cd openssl-0.9.5a
5. ./config
6. make
7. make test
8. make install
```

For means of compatibility OpenSSL is by default installed into `/usr/local/ssl` .
For the various options refer to the file `Install` in `/usr/local/openssl-0.9.5a` .

The include files are stored in `/usr/local/ssl/include`, the libraries in `/usr/local/ssl/lib` .
For compilation *gcc* must be run with the options `-L/usr/local/ssl/lib -lssl -lcrypto`. Otherwise compiler and linker cannot find the OpenSSL libraries.

4.4 Setting up Borland C

Borland C++ 3.1 for DOS is an antique 16-bit software, but for our purpose it is the number-one choice. BC is delivered on floppy disks. To install it, the install program on the first diskette has to be run.

At this point, some very important options have to be adjusted for compiling proper code for the IPC@CHIP. Adjust them under *Options* | *Compiler* | *Code Generation* and *Options* | *Compiler* | *Advanced Code Generation*, respectively.

Memory Model	`Large`
Instruction Set	`80186`
Floating Point	`Emulated`

Table 4.3: Configuration of Borland C++ 3.1

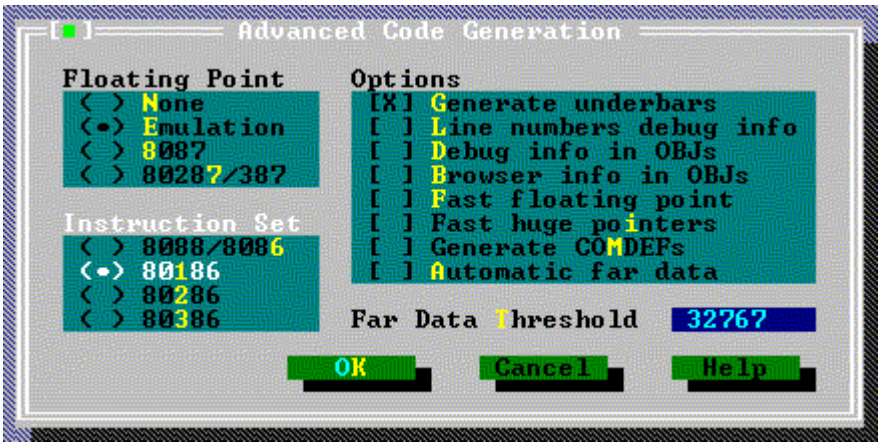


Figure 4.3: Configuration of Borland C++ 3.1

The various defines plus the remaining options are not listed here, because they first have to be defined (see section 6.5 on page 40). The defines and options finally used are listed in the make file (`opssl bc.mak`, available on the enclosed CD-ROM).

Borland C++ 3.1 could not be included on the CD-ROM due to the License Agreement.

4.5 Setting up fswcert

fswcert was originally written by Andreas Gruenbacher. In our SSL implementation project a modified version is used to convert private keys from the PEM format into the HST format.

The installation procedure of the modified *fswcert* is described below:

```
1. cp [CDROM]/tools/fswcert/fswcert.tar.gz  ~/target_dir
2. cd ~/target_dir
3. tar –xzf fswcert.tar.gz
4. cd fswcert
5. make
```

The compilation will issue a couple of warnings, which can be ignored altogether.
The usage of the tool *fswcert* and the details of the HST format are described in the report in section 6.3 on page 30.

5 Approach

This chapter gives a short description in chronological order of how the different tasks of this diploma project were tackled. For a detailed description refer to chapter 6: Report, where the different tasks are ordered by topic.

As written in the task description the main goal of this diploma project was the development of a web server capable of SSL encryption and CGI support. The main task was split into several smaller tasks. These consisted of the following:

1. **Size reduction of OpenSSL**
   OpenSSL is a huge software library. It is far too voluminous for the purpose of this diploma project, and since there is only 256 kilobytes of flash memory available for user programs in the IPC@CHIP, it had to be stripped down drastically.

2. **Adaptation to the IPC@CHIP environment**
   The original library was developed on a 32-bit environment, whereas IPC@CHIP only provides a 16-bit environment. This adaptation was mainly due to performance, but in some cases the 32-bit code simply did not work and had to be adapted.

3. **Implementation of a web server**
   The best solution would have been the integration of the SSL encryption part into the already existing web server on the IPC@CHIP, but Beck answered negatively to a request. They said they were not able to provide their web server source code due to a license agreement.

OpenSSL features every cipher suite and algorithm that can be thought of (most of which are considered to be weak nowadays). Hence, it was necessary to select only one cipher suite out of the available ones. That cipher suite had to be strong and widely used, that is to say supported by common web browsers like Netscape Navigator, or MS Internet Explorer. The decision was reached to support a handshake with RSA_WITH_RC4_128_MD5. This implies that the RSA algorithm is used for asymmetric encryption (generation of key material), RC4 (128 bit strong) for symmetric encryption, and MD5 as a message digest. Therefore the support of all the other algorithms was cancelled, except the support of SHA, which is needed for generating the key material (see figures 3.11 and 3.13 in chapter 3: Introduction). The cancellation of the cipher support was mainly done with define tags in the make file. Nevertheless, some source files had to be adjusted manually.
Also manually adjusted were all the type irregularities. This was always done in parallel to the other tasks. Under Linux, where SSL originates from, the data type `int` is 32 bit wide, whereas under DOS, the environment of IPC@CHIP, the type `int` is only 16 bit wide. This whole adaptation consumed a lot of time, and the first complete compilation of SSL for IPC@CHIP was not due until 2 weeks of work were done. Since only warnings occurred and no errors, it was a functioning compilation. However, the resulting libraries still occupied too much space to be loaded to the IPC@CHIP.

Prior to further reducing of code size, the decision was raised to implement a test server in order to check the overall file size. Still there was hope that only a small number of functions were called and therefore linked into the executable file. After compilation and linking the executable file turned out to be slightly smaller than 500 kilobytes. There was the possibility of compressing it using PKLite. This would reduce the code to approximately 45 to 50 percent of its original size. The compressed code would fit into the flash memory, but expanded it would take up more memory than was available (around 410 kilobytes). Important to point out is the fact that PKLite needs up to 50 kilobytes of memory for its expanding algorithms, which means that the uncompressed executable must not be greater than 350 kilobytes. As a result, there was still plenty of code to reduce.

While looking into the map file, produced by the linker, a discovery was made. Many functions were linked from the libraries into the executable file, but never called. This was caused partly by macro definitions, partly by the code itself. Many function were called from a macro. Although the macro was never used, it still occupied space. Therefore they had to be eliminated laboriously. The other cause was, as aforementioned, the code. In numerous switch clauses were cases that could never be reached for they were not supported. An example is given in figure 5.1. As only RSA is supported, the Diffie-Hellman and the Fortezza case are never reached. The compiler, though, linked all the functions into the executable file, even those that were never called. Therefore each switch similar to the example had to be adjusted. This work was done with great effort, but it showed a great result: The code reduction was noticeable.

```
   ...
switch (type)
{
    case RSA     : do RSA stuff
    case DH      : do Diffie
                   Hellman stuff
    case Fortezza: do Fortezza stuff
    case default : do nothing
}
   ...
```

Figure 5.1: Example Switch

Despite the macro elimination the libraries ASN1, PEM, and X509 could not be removed entirely. Therefore other ways were sought to read certificate and private key, which are stored in PEM format. A new format for the private key (HST, refer to section 6.3.2 in the next chapter for details) was created, and also functions to read the new format. The certificate was read in PEM format, translated to ASN.1, stored in memory, translated again to DER (binary ASN.1), and finally transmitted over the secure channel. Reading the certificate directly in DER format would save much processing time and, even more important, code. After these enhancements the three libraries ASN1, PEM, and X509 could be removed completely.

Finally, the libraries and the aforementioned test server (*serv*) were compiled and linked, so that after compression with PKLite the executable could be loaded to the IPC@CHIP. The file size of the uncompressed executable was around 200 kilobytes, the compressed version slightly under 100 kilobytes. During the SSL handshake the test server suddenly terminated. To find the reason, several calls to `printf()` were inserted into the code. It was discovered that Netscape Navigator 4.75 sent an SSL version 2.0 ClientHello message. Therefore SSL for IPC@CHIP had to be extended to be capable of version 2.0 request handling. This was quite an easy task for we had done a similar procedure before with version 3.0 request handling. After the extension *serv* functioned well with the Navigator as client. In contrast to

Netscape, which does not check the certificate chain, Microsoft Internet Explorer 5.5 did not function because of an invalid certificate signature. After the generation of a new certificate set (CA certificate and server certificate), Internet Explorer still was not able to perform the SSL handshake successfully. Only after the import of the CA certificate into Internet Explorer did it function properly. As a summary, Navigator does neither check the certificate chain, nor the match between distinguished name (in a server certificate the distinguished name is the DNS entry of the server) and the domain name of the server. Internet Explorer, in contrast, refuses to connect to a secure server if distinguished name and domain name do not match, or if the issuer of the server certificate is unknown or not trusted.
Now that these problems were solved, the authors decided to implement TLS version 1.0 functionality, a feature whose implementation was a matter of only an hour.

Since the test server was working, it was time to develop the web server. The web server had to be developed from scratch because Beck was not allowed to provide the source code of their IPC@CHIP web server due to a license agreement. The development of the web server (*mws*) itself was more of an extension of the test server and was done without experiencing major problems. The implementation of CGI functionality was the most interesting task while developing *mws*. In order to be fully compatible to Beck's web server, *mws* makes use of interrupt `0xAB`. The authors wrote an interrupt service routine (ISR) by themselves, implementing interrupt `0xAB` functions `0x01` and `0x02`. These are the main functions to install (`0x01`) and remove (`0x02`) a CGI function. The challenge was to return values from the ISR in the processor registers. The solution was a trick described in section 6.8.3.2 (page 48) in the next chapter. As long as CGI programs written for the IPC@CHIP web server do not use interrupt `0xAB` functions `0x03` to `0x09`, they can be run with *mws*.

The final result of our diploma project is a tiny (101 kilobytes), fully functional, CGI-capable web server supporting SSL handshake and encoding (SSL version 3.0 and TLS version 1.0). The used cipher suite (RSA with 1024-bit encryption, RC4 with 128-bit encryption) is nowadays considered to be very safe and is widely used. Thus no more restrictions are set on the usage of IPC@CHIP, not even for safety-relevant applications.
For software developers all SSL libraries (except ASN1, PEM, and X509) were adopted to the 16-bit DOS environment and are available to use in their own software projects.

6 Report

6.1 BIOS on the IPC@CHIP

This diploma project needed the smallest possible BIOS in order to have as much memory as possible available to user programs. At the beginning of the work the current BIOS version was 0.68. The web server was built in, although this was not necessary. After a request for a special version without built-in web server Beck provided a pre-release of BIOS 0.69. As it turned out this pre-release was somewhat unstable. Therefore, another solution was sought and found in BIOS version 0.69 which was released approximately one month later. BIOS 0.69 is Beck's first release that offers various versions. These versions all support a different set of functions. Table 6.1 shows the configurations:

	Tiny	Small	Medium	Large	MediumPPP	LargePPP
Serial	x	x	x	x	x	x
File System	x	x	x	x	x	x
Ext Disk			x	x	x	x
XMODEM	x	x	x	x	x	x
Ethernet		x	x	x	x	x
TCP-IP		x	x	x	x	x
I2C	x	x	x	x	x	x
Hardware API	x	x	x	x	x	x
CFG server		x	x	x	x	x
Webserver				x		x
FTP Server			x	x	x	x
Telnet Server			x	x	x	x
PPP server					x	x

Table 6.1: A Range of BIOS' that Differ in Size and Function Sets

For the diploma project BIOS 0.69 Medium is the best choice, as it is the smallest version that supports telnet and ftp server but not the web server. Additionally, BIOS 0.69 Medium features diverse bug fixes. It turned out to have even more memory space available for users (424848 bytes) than the pre-release (410416 bytes).

The BIOS is available on the home page of Beck [5] or on the enclosed CD-ROM (`[CDROM]/bios/0.69`). For details refer to the file `history.txt` in the same directory.

6.2 Supported Cryptographic Algorithms

The supported cipher suite is RSA_WITH_RC4_128_MD5, according to the task description (see chapter 2). This section introduces this suite and the pertinent algorithms.

- **RSA**
  RSA is a public key encryption standard. Due to the tremendous computing power the algorithm needs to encrypt and decrypt ciphertext, RSA is only used during handshake procedure. The premaster secret, which the master secret and key material is derived from, is securely transmitted by RSA encryption. Afterwards, RSA is not used anymore.

- **RC4 128 bit**
  RC4 is a stream cipher with 128-bit key size. After the generation of the session keys all messages are secured by RC4. RC4 is a fast algorithm. The delay due to encrypting the data is barely noticed, even on the IPC@CHIP (80186, 20 MHz).

- **MD5**
  MD5 is a message digest algorithm and therefore converts an arbitrarily long data stream into a digest of fixed size (128 bits). MD5 is used for the generation of master secret and key elements (see section 3.2.4), as well as for the protection of the integrity of application data.

- **SHA1**
  SHA1 is a message digest like MD5, but the digest size is 160 bits. SHA1 is used for the generation of master secret and key elements only (see section 3.2.4).

6.3 Certificate and Key Formats of Mini Webserver

In every step that was done the goal of reducing the size of the SSL implementation was kept in mind. It turned out that the ASN.1 and X.509 libraries took up a great amount of space. If there was be a way to eliminate these two libraries, the program size would decrease substantially. The utilization of the ASN.1 functions happens to be limited to certificate transmission and client authentication. As the SSL implementation for IPC@CHIP does not support client authentication (see task description, chapter 2), certificate transmission only was subject to analysis.

6.3.1 Certificate Format

Prior to sending the Certificate Message, the certificate must be read from a file where it is stored in PEM format. The certificate is read, translated, and kept in an internal structure, the same structure that hosts the private key. To be fed into the Certificate message the certificate must be read from this structure and, once more, translated into the DER format (binary coded ASN.1). All in all a tremendous overhead that could

be skipped if the certificate was stored in the file directly in DER format. In this case, the certificate could be read from the file and directly fed into the Certificate message without involving any translation. The only remaining problem is the internal structure where the certificate is saved. Strangely enough the certificate part of the structure is never accessed at runtime. Hence it is superfluous to store the server certificate in the internal structure. It can be read from the file in DER format when needed and fed directly into the Certificate message.

The next step was to rewrite the function `ssl3_send_server_certificate()` in the file `s3_srvr.c`. Instead of getting the certificate from the internal structure and translating it, the function simply reads the certificate from the disk in the proper sending format (DER). As `ssl3_send_server_certificate()` is invoked from within `ssl3_accept()`, the function `SSL_CTX_use_certificate_file()` (in the file `ssl_rsa.c`) in the main server program became obsolete and was removed (see chapter 7: Manual - SSL for IPC@CHIP).

Due to this enhancement the ASN.1 functions were entirely removed from the program.

```
int ssl3_send_server_certificate(SSL *s)
{
    FILE* fcert;
    unsigned char *p;
    struct stat fcert_stat;
    BUF_MEM *buf;

    if (s->state == SSL3_ST_SW_CERT_A) {
        if((fcert = fopen("s_cert.der", "rb")) == NULL) {
            printf("file not exist\n");
        }
        // Get data associated with "fcert":
        if(!fstat(fileno(fcert), &fcert_stat)) {
            buf = s->init_buf;

            // 7 byte: type, message length and certificate chain length
            // 3 byte: certificate 1 length
            if(!BUF_MEM_grow(buf, (int)(fcert_stat.st_size + 10))) {
                s->init_num=(int)0;
            }
            else {
                // Attempt to read in 'fcert_stat.st_size' bytes:
                p = (unsigned char*)&(buf->data[10]);
                fread(p, sizeof(unsigned char), fcert_stat.st_size, fcert);

                // insert message type: 11
                p = (unsigned char*)&(buf->data[0]);
                *(p++)=SSL3_MT_CERTIFICATE;

                // insert message length
                l2n3((int)(fcert_stat.st_size + 6), p);

                // insert certificate chain length
                l2n3((int)(fcert_stat.st_size + 3), p);

                // insert certificate 1 length
```

```
                l2n3((int)(fcert_stat.st_size), p);

                s->init_num = (int)(fcert_stat.st_size + 10);
            }
            s->state = SSL3_ST_SW_CERT_B;
            s->init_off = 0;
        }
    }
    return(ssl3_do_write(s,SSL3_RT_HANDSHAKE));
}
```

6.3.2 Private Key Format

An approach similar to the one in the previous section was made for the private key.

The private key originally was read in the PEM format, translated and stored in an internal structure in so called BIGNUM's (structure to represent big numbers). This translation used X.509 functions and produced unnecessary overhead. If the private key could be read in the same format as it is stored internally, that overhead would disappear. For this reason the HST (**H**ex **ST**ring) format was created. A file in HST format consists of eight strings (each one on a separate line) that represent the hexadecimal values of the private key numbers (for further details about the structure of a private key refer to [8], page 36, section "Private-key Syntax"). These lines can be read and translated into BIGNUM's using the function `BN_hex2bn()`. The BIGNUM's are then stored in the private key part of the aforementioned internal structure (see last section). The lines in a HST file consist of:

```
Modulus           :
Public Exponent   :
Private Exponent  :
Prime 1           :
Prime 2           :
Exponent 1        :
Exponent 2        :
Coefficient       :
```

Figure 6.1: Private Key in the HST Format; Each Line Represents a Hexadecimal Number

The now obsolete function `SSL_CTX_use_privatekey_file()` was removed from the code. Instead, a new function `SSL_CTX_MINI_use_privatekey_file()` was created. This function reads a private key file in HST format, translates each line into a BIGNUM and stores them in the structure EVP_PKEY (private key part of the internal structure introduced in the previous paragraph). The match between private key and public key is not checked since the certificate is no longer stored internally in memory.

Due to this enhancement the X.509 functions could be entirely removed from the program.

```
short SSL_CTX_MINI_use_privatekey_file( SSL_CTX *ctx, const char *file )
{
    FILE *f;
    EVP_PKEY *pk;
    RSA *r;
    BIGNUM *big;
    short i = 1;
    char buf[500];

    f = fopen( file, "r" ); // read only
    if (!f) return( -1 ); // error opening file
    ctx->cert->key = &(ctx->cert->pkeys[0]);
    ctx->cert->valid = 0;
    if(ctx->cert->key->privatekey == NULL) ctx->cert->key->privatekey = EVP_PKEY_new();
    pk = ctx->cert->key->privatekey;
    if (!pk) return( -1 ); // no memory allocated
    if (pk->pkey.rsa == NULL) pk->pkey.rsa = RSA_new();
    pk->type = EVP_PKEY_RSA;
    CRYPTO_add(&pk->references, 1, CRYPTO_LOCK_EVP_PKEY);
    r = pk->pkey.rsa;
    if (!r) return( -1 ); // no memory allocated
    while (!feof( f )) {
        if (fgets( buf, 500, f ) != buf) {
           // nothing
        }
        else {
            big = BN_new();
            BN_hex2bn( &big, buf );
            switch (i) {
                case 1: if (r->n == NULL)      // modulus
                            BN_free( r->n );
                        r->n = big;
                        break;
                case 2: if (r->n == NULL)      // public exponent
                            BN_free( r->e );
                        r->e = big;
                        break;
                case 3: if (r->n == NULL)      // private exponent
                            BN_free( r->d );
                        r->d = big;
                        break;
                case 4: if (r->n == NULL)      // prime 1
                            BN_free( r->p );
                        r->p = big;
                        break;
                case 5: if (r->n == NULL)      // prime 2
                            BN_free( r->q );
                        r->q = big;
                        break;
                case 6: if (r->n == NULL)      // exponent 1
                            BN_free( r->dmp1 );
                        r->dmp1 = big;
                        break;
                case 7: if (r->n == NULL)      // exponent 1
                            BN_free( r->dmq1 );
                        r->dmq1 = big;
                        break;
```

```
                case 8: if (r->n == NULL)      // coefficient
                            BN_free( r->iqmp );
                        r->iqmp = big;
                        break;
                default: break;
            }
            i++;
        }
    } // while
    fclose( f );
    return( 0 );
}
```

6.3.3 Format Conversion

As the Mini Webserver formats are defined now (DER for the certificate, HST for the private key), the original certificate and private key must be converted into these new formats.

- **Certificate Conversion : PEM to DER**
  The conversion from PEM into DER is performed by *openssl*. A properly working OpenSSL environment (see section 4.3) on Linux is crucial for this task.

- **Private Key Conversion : PEM to HST**
  The conversion from PEM into HST cannot be done with *openssl*. Instead, an existing tool by the name of *fswcert* was found on [9]. After minor modifications, *fswcert* was just the right tool for the particular requirements of this project. The modified *fswcert* (see below) is now capable of converting PEM into HST. For the installation procedure refer to section 4.5.

An example of both PEM-to-DER and PEM-to-HST conversion is given in section 6.4.2 on page 37.

6.3.3.1 fswcert

fswcert was originally written by Andreas Gruenbacher for the FreeS/WAN project. He has hereby contributed a tool based on OpenSSL 0.9.5a, that allows the painless extraction of public keys, private keys and ASN.1 distinguished names from X.509 certificates and RSA or PKCS#12 key files.

The output of the original *fswcert* is roughly what is required for the HST format in this SSL implementation project. However, there are differences that must not be ignored. The original *fswcert* command line instruction

```
fswcert -k -- type rsa s_key.pem > s_key.hst
```

produced the following output:

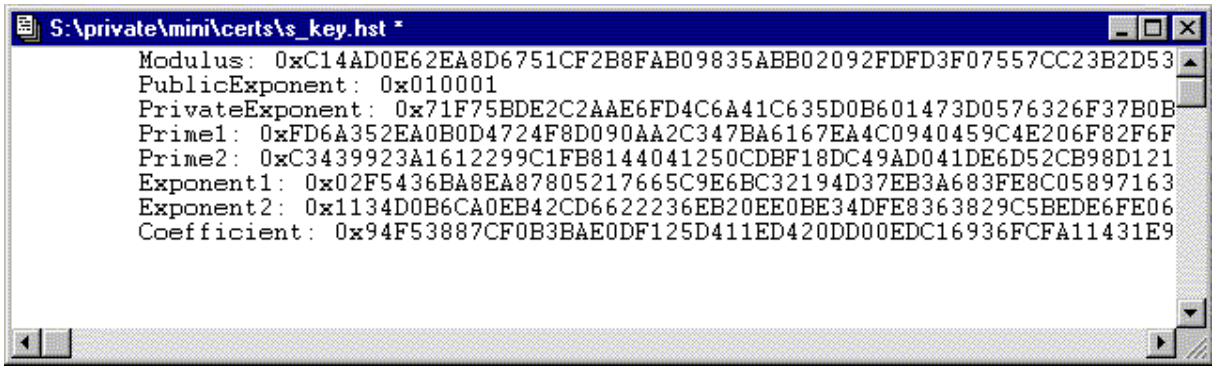


Figure 6.2: Private Key in the Output Format of the Original *fswcert*

Given the expected output as shown in figure 6.1, merely minor customization is necessary. The textual description in front of the number ("*Modulus:*", "*Public Exponent:*", ...) plus the hex identifier (`0x`) must be eliminated. Subsequent to the modification of the print macro in the source code, the output of *fswcert* produced an output according to the requirements of the HST format.

The modified version as well as the original version of *fswcert* are available on CD-ROM (`[CDROM]/tools/fswcert`).

6.4 Generating Certificates

For a primer on X.509 Certificates refer to section 3.3.

Certificates are essential for the proper use of SSL. Each server must possess a server certificate. For a client the certificate is optional as long as client authorization is not demanded by the server. Should this be the case, a secure connection cannot be established without a valid client certificate. The SSL implementation for IPC@CHIP does not support client authorization. In the next paragraphs follows a detailed description of how to generate certificates. A working installation of OpenSSL on Linux (see chapter 4.3.2 on page 23) is inevitable. Should the exact syntax be of interest, further details can be found in the SSL handbook [7], page 18.

The certificates used for testing the SSL implementation for IPC@CHIP are printed in appendix E. They are also available on the enclosed CD-ROM in both the PEM format and the Mini Webserver format (DER and HST, respectively).

6.4.1 CA Certificate

Every certificate, whether server or client certificate, must be signed by a Certificate Authority. By default Internet browsers have several CA certificates whom they trust, e.g. Thawte, Verisign, or Entrust. Since the price to obtain a server certificate signed by aforementioned Certificate Authorities is rather high, we created our own CA by using OpenSSL running on Linux.

Step 1: Generation of a 2048 bit key pair, secured by password and IDEA cipher

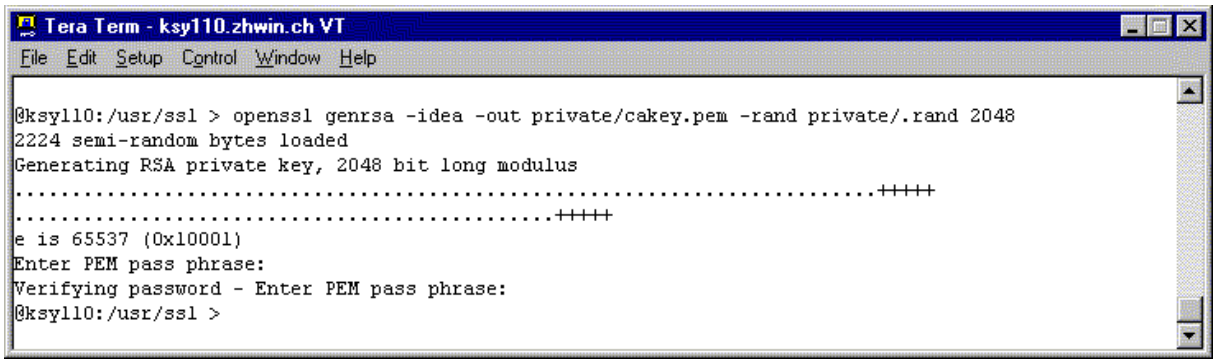


Figure 6.3: Generation of the Key Pair for the Root Certificate

Step 2: Creation of a self signed CA certificate

IMPORTANT: the tag nsCertType in the configuration file openssl.cnf must be set to “sslCA, emailCA, objCA”.

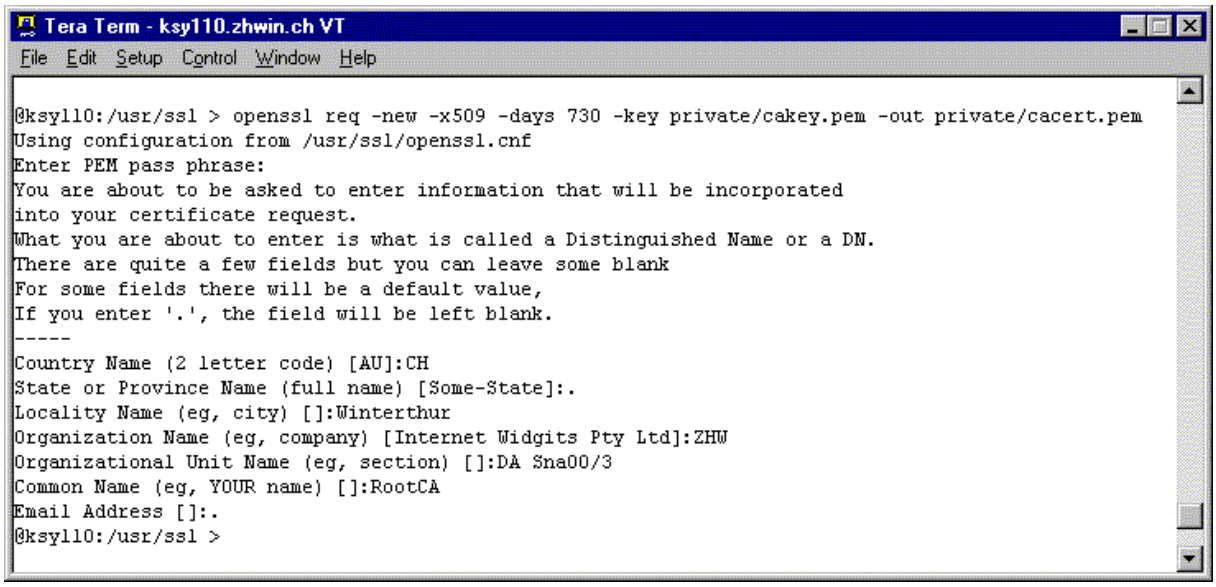


Figure 6. 4: Creation of a Self Signed CA Certificate

Step 3: Keeping track of generated certificates

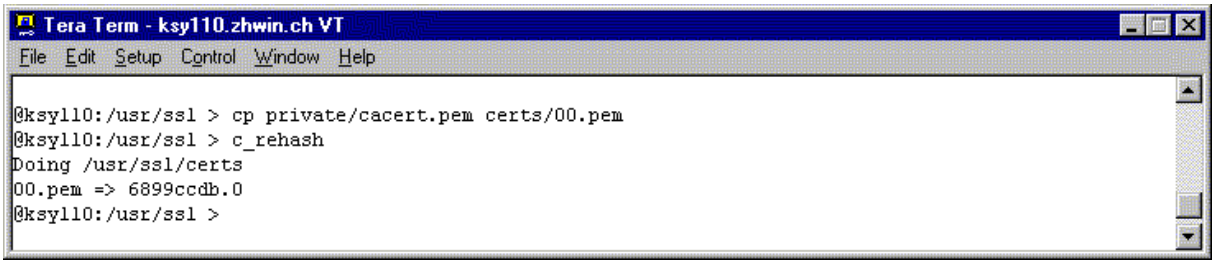


Figure 6. 5: Keeping Track of the Certificates

Step 4: Conversion of the CA certificate into the PKCS12 format in order to make it importable for Internet browsers

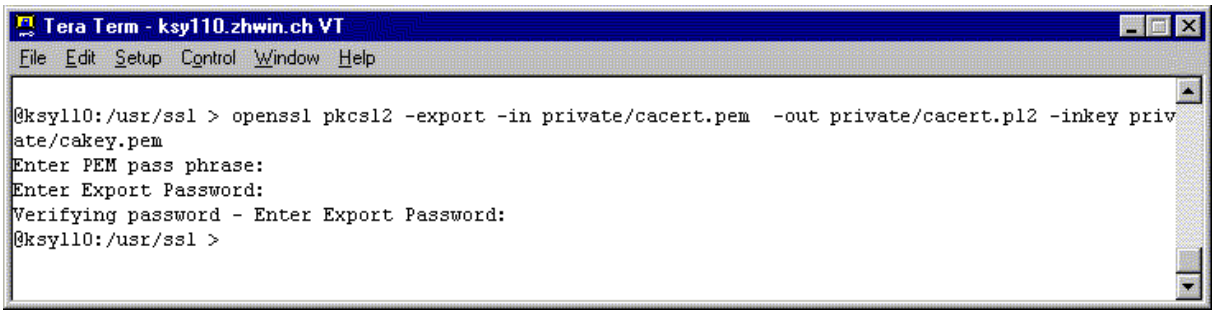


Figure 6. 6: Conversion from PEM Format to PKCS12 Format

6.4.2 Server Certificate

The generation of a server certificate is similar to the generation of a CA certificate. The difference lies in the tags: for a server certificate nsCertType must be set to “server”, and nsSslServerName to the server’s domain name (in this case: mini.zhwin.ch).
In order to sign a server certificate, a valid CA certificate and the corresponding private key must exist.

Step 1: Generation of a 1024 bit key pair, not secured by a cipher

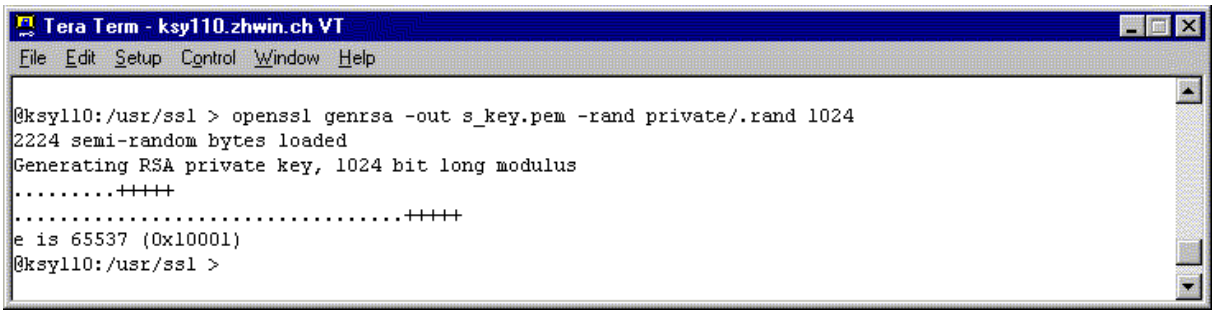


Figure 6. 7: Generation of a Key Pair for the Server Certificate

Step 2: Creation of a request which has to be signed by the CA certificate

IMPORTANT: the tag nsCertType must be set to “server”, nsSslServerName must be set to the server’s domain name (in this case: *mini.zhwin.ch*).

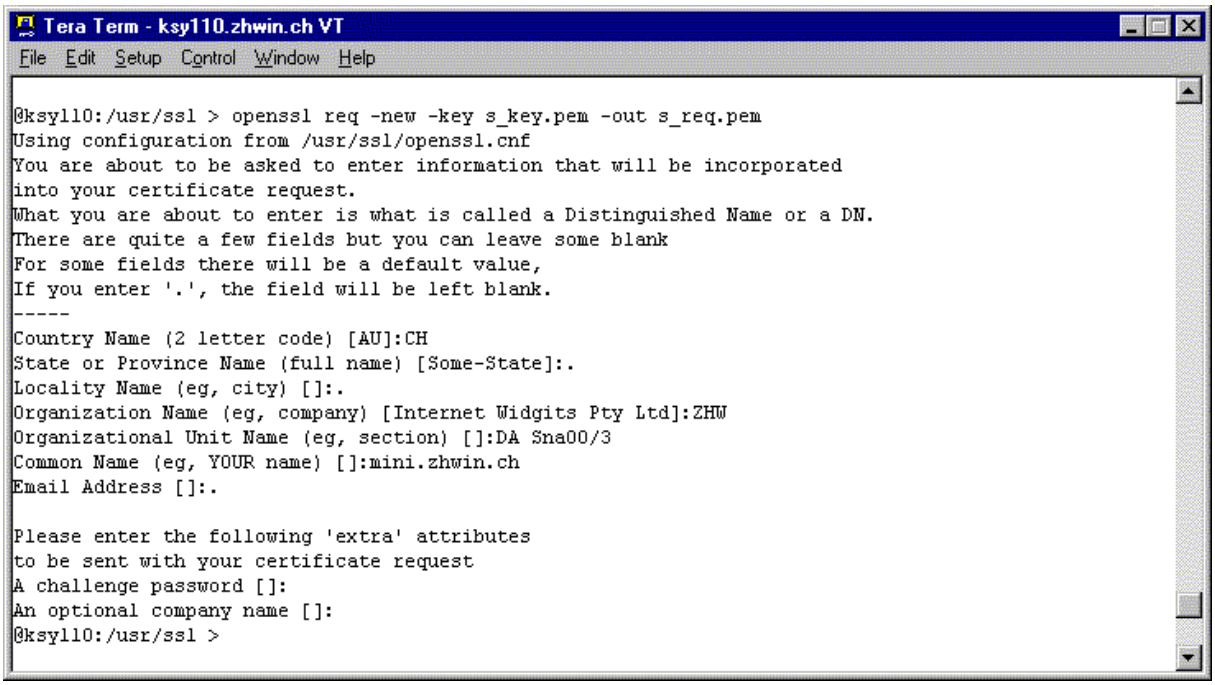


Figure 6. 8: Creation of a Request for Signing the Server Certificate

Step 3: Signing the server certificate

IMPORTANT: The program openssl searches for the files `index.txt` and `serial`. Since they do not yet exist they must be generated as described in figure 6.9. Unfortunately, these files are not mentioned in 7, but are essential for the signing procedure. The file `serial` must contain “01” for the number of the second certificate (“00” stands for the CA certificate).

The command `touch index.txt` creates an empty file `index.txt`.
The command `cat > serial` copies all input characters into the file `serial` and terminates after pressing `[Ctrl-D], [Ctrl-C]`.

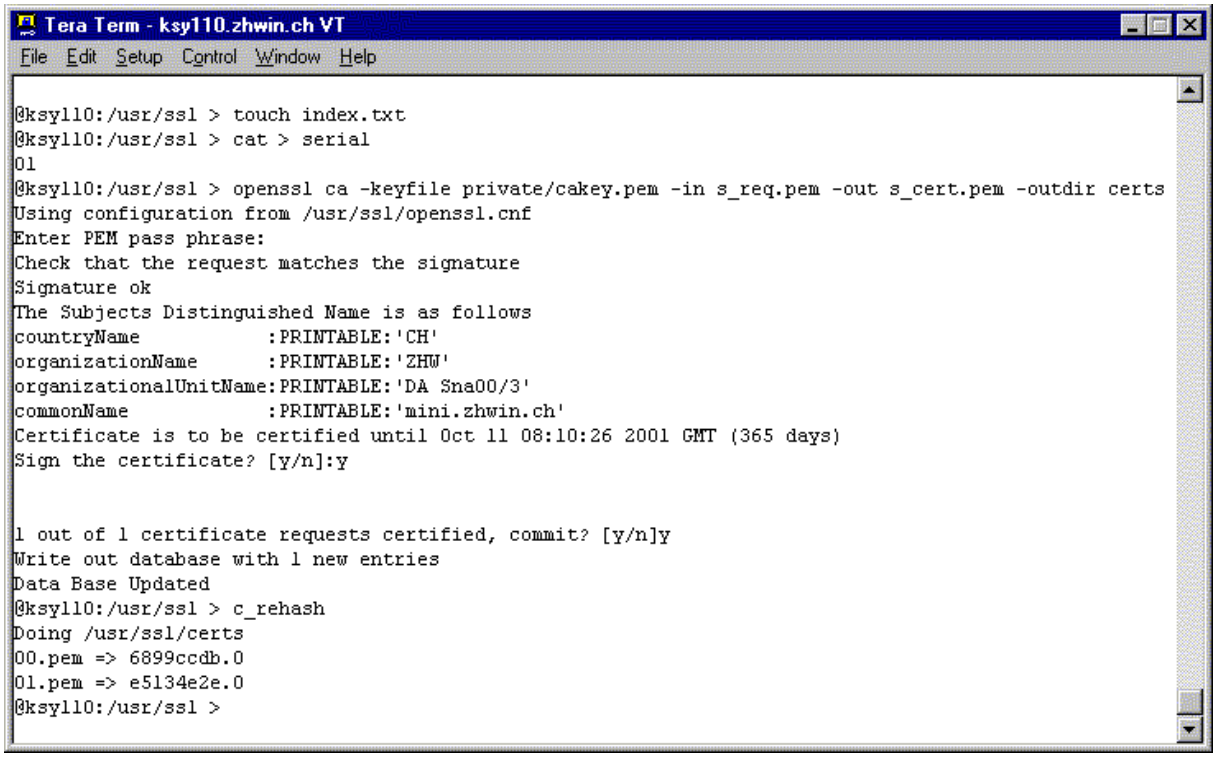


Figure 6.9: Signing the Server Certificate

Step 4: Conversion

In order to be readable for Mini Webserver the server certificate must be converted into the DER format, the private key into the HST format (for further detail on the formats used by Mini Webserver refer to section 6.3 in this chapter). The correct conversion is based on the proper installation of *openssl* and *fswcert* (see chapter 4). The first command line in figure 6.10 executes the conversion from PEM to DER, the second the conversion from PEM to HST.

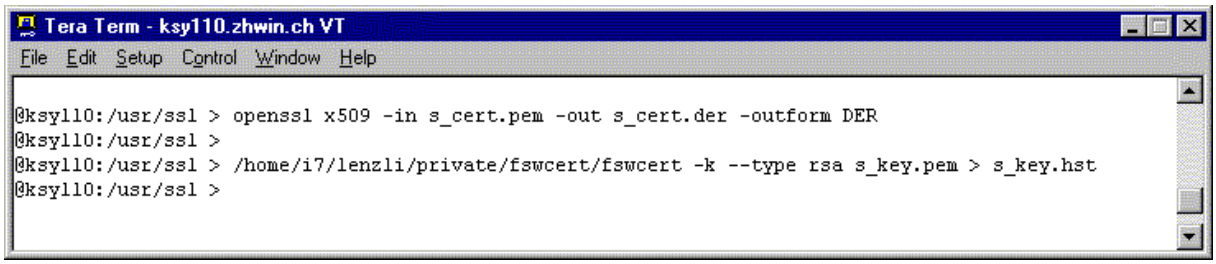


Figure 6.10: Conversion from PEM Format to the Formats of Mini Webserver (DER and HST)

6.5 Make File

The make file `opsslbc.mak` is the crucial part of the diploma project. The make file controls the rest of the SSL implementation for IPC@CHIP and decides on success or failure. If the make file functions correctly, all files are compiled, the libraries built and the necessary header files collected into one single directory.

A list of all files that were altered is available on the enclosed CD-ROM (`[CDROM]/SSLforIPC_CHIP/openssl/altered.txt`).

6.5.1 Inside the Make File

In order to control the adaptation of the OpenSSL package to the IPC@CHIP, the automatically generated make file had to be changed. As mentioned before several files had become obsolete (e.g. all files to do with ASN.1 and X.509), others had been added, some had been renamed. The following files had to be renamed because DOS and the Borland C++ compiler only support short file names:

`openssl/crypto/opensslconf.h`	→	`openssl/crypto/opensslc.h`
`openssl/crypto/md32_common.h`	→	`openssl/crypto/md32_c.h`
`openssl/crypto/stack/safestack.h`	→	`openssl/crypto/safestck.h`

Many lines that were initially in the make file have been cut out, e.g. all that handled ASN.1 or X.509.

The *make* tool internally invokes *bcc* (Borland's C compiler) and *tlib* (librarian). Since the length of the argument list is limited, *make* creates a configuration file which is passed to *bcc* and *tlib*, respectively.

Following is the part of the make file responsible for creating the configuration file `opssl.cnf`. All arguments are written into this file, and only the file name is then passed to *bcc*. This process makes it possible to deliver as many arguments to the compiler as needed.

```
#####################################################################
#        *Compiler Configuration File*
#
opssl.cfg: opsslbc.mak
  copy &&|
-c
-ml
-1
-f
-ff
-i250
-O
```

```
-Oe
-Ob
-Z
-d
-v-
-vi-
-y-
-wbbf
-wpin
-wamb
-wamp
-w-par
-wasm
-wpro
-wdef
-wsig
-wnod
-w-aus
-wstv
-wucp
-wuse
-w-stu
-weas
-wpre
-nG:\OPENSSL\TMP16BC
-I$(INCLUDEPATH)
-L$(LIBPATH)
-P-.C
-DL_ENDIAN;MSDOS;NO_IDEA;NO_RC2;NO_RC5;NO_MD2;NO_MDC2;NO_BF;NO_CAST;NO_DES;NO_DSA;NO_DH;
 NO_SSL2;NO_ERR;MINI;NO_RIPEMD;NO_SHA0
| opssl.cfg
```

For more information on the given options refer to [2], page 619 and following.

6.5.2 Using the Make File

The usage of the make file is straight forward. Should the configuration differ from the one described in chapter 4, the path specifications at the beginning of the make file can be altered individually.
The following steps show the usage of the make file:

1. `g:\`
2. `cd \openssl\ms`
3. `set path=g:\borlandc\bin;%path%`
4. `make –f opsslbc.mak`

After successful termination the program a states a message:
finished compiling and linking OpenSSL

The libraries can now be found in `g:\openssl\out16bc`,
the header files in `g:\openssl\inc16bc`.

6.6 Randomness

Randomness is a crucial part of cryptography for every key pair is generated using large random numbers. A key pair generated with insufficient unpredictability is far easier to break than a truly random key pair.

As with many other systems, the SSL implementation for IPC@CHIP generates random numbers using the service of a Pseudo Random Number Generator (PRNG). The random number generated by the PRNG is the output of an irreversible hash algorithm (SHA1), which the current state of the PRNG has been fed to. The output of truly random numbers implies that the PRNG's initial state must be truly random. To accomplish this requirement the PRNG must be seeded with unpredictable data such as mouse movements, keys pressed at random by the user, or random data from a seed file. Since it is virtually impossible to randomly press keys at every program start of the server, the random data is read from a 1024-byte seed file (`a:\rand.rnd`). After the generation of random numbers, the internal state of the PRNG must be saved into the seed file, otherwise the numbers generated at the next program start will lack uncertainty.

The initial random seed file was generated with the assistance of the program PGP, where randomness is generated, as mentioned above, by pressing keys at random and mouse movements. Thus, the Pseudo Random Number Generator is understood to be unpredictable.

6.7 Test Server

The Test Server program (`serv.exe`) provides the minimum functionality necessary for establishing an SSL connection. Since it returns only one HTML page and then terminates, the test server *serv* is not a web server, which at this stage is not necessary. The main goal of *serv* is to successfully establish an SSL connection and securely transmit data.

The program at startup creates all necessary SSL structures (context and session) after seeding the Pseudo Random Number Generator. Subsequently, the private key is read in HST format as described previously. The certificate is read later on during the handshake procedure. *Serv* is now listening on the standard SSL port 443 for a TCP socket connection to be established, the process of which is not described here. The SSL handshake itself is done in the function `ssl3_accept`. For a detailed description of the handshake procedure refer to section 3.2.2 on page 9. Right after the connection has been established the internal state of the PRNG must be saved to the random seed file in order to guarantee randomness at the next program startup (see previous section). The server then listens for incoming data (preferably a HTML Get Request) and transmits a HTML page to the client. A web server would continue listening at this point, but *serv* goes into the cleaning up phase where all allocated memory is wiped and freed.

In the next sections the SSL handshake between the test server and different clients is described in detail.

6.7.1 Netscape Navigator

To start with, Netscape Navigator 4.75 (128 bit encryption) was chosen as SSL client. Once the program was started, everything functioned fine, until a handshake error was detected. The ClientHello message could not be processed. In order to discover the source of the error, a LAN Analyzer was used to report the entire LAN traffic between client and server. The ClientHello message received by the server is illustrated in figure 6.11.

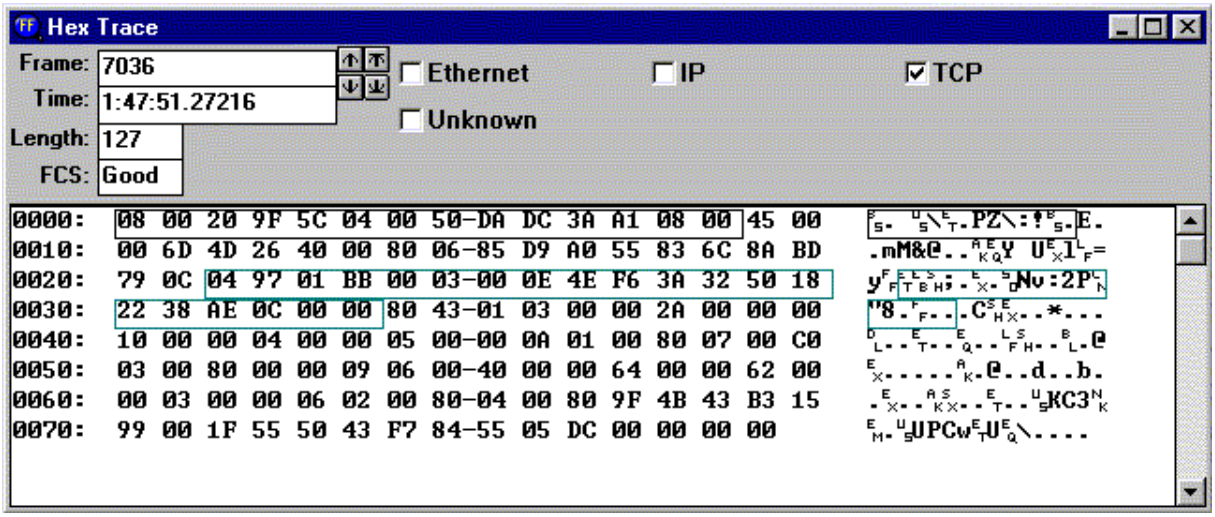


Figure 6.11: SSL version 2 ClientHello message

The hexadecimal numbers enclosed by a thin line represent headers. The content of the first frame is the Ethernet header or MAC header. Between the two frames is the IP header, and the second frame represents the TCP header. All following hexadecimal numbers subsequent to the TCP header are part of the ClientHello message. An SSL version 3.0 message normally starts with `16h` (the number of the handshake protocol), but this message starts with `80h`. The reason for this effect is that Navigator, in order to be backwards compatible, sends an SSL version 2.0 ClientHello message. Hence Navigator is capable of communicating with SSL servers that support only version 2.0. Notice that in this version 2.0 ClientHello the version field is set to `3.0`. In effect, a version number of 3.0 (or 3.1 for TLS version 1.0) is the hint for the server that, even though the client is sending a version 2.0 message, the client is capable of SSL version 3.0, or TLS version 1.0, respectively. A server only capable of version 2.0 ignores the “incorrect” version number and responds with a version 2.0 ServerHello. A version 3.0 server, in contrast, responds with an SSL version 3.0 ServerHello, or a TLS version 1.0 ServerHello, respectively. The rest of the message is relatively straightforward, but notice that version 2.0 ciphers are 3 bytes in length, rather than 2. This fact provides a convenient way for dual version clients to propose version 3.0 cipher suites within a version 2.0 ClientHello. The client simply prepends a single byte of 0 to the 2-byte cipher suite values. Since all legitimate 2.0 cipher suites start with a value other than 0, a dual version server will be able to recognize the modified version 3.0 cipher suites correctly.

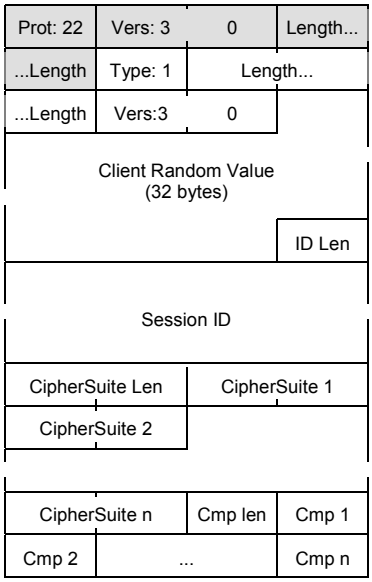


Figure 6.13: The ClientHello Message Proposes CipherSuites.

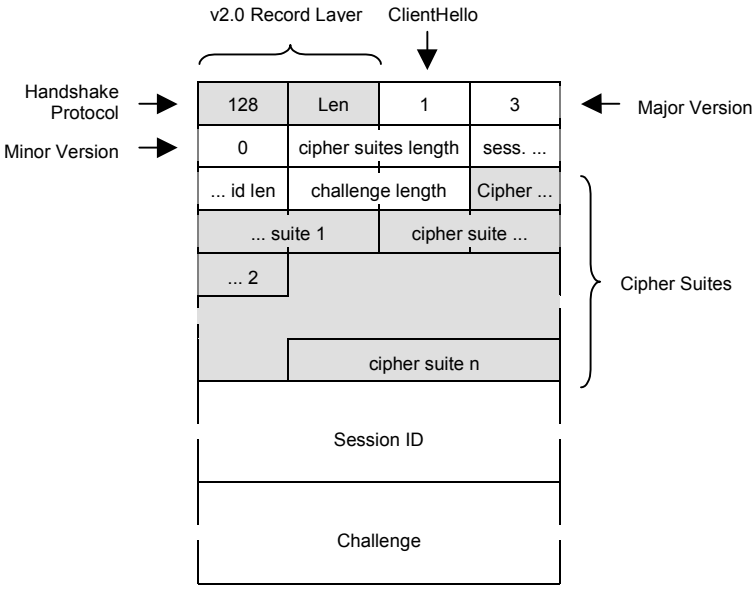


Figure 6.12: The ClientHello Message Proposes CipherSuites.

As it turned out that browsers can send version 2.0 ClientHello messages, although they support version 3.0 or higher, the test server *serv* (and later on the Mini Webserver as well, of course) must be able to recognize and handle those messages correctly. *Serv* had to be reprogrammed, which turned out to be a change in one single line in the program:

version 3.0 only

```
    ...
SSLeay_add_ssl_algorithms();
meth = SSLv3_server_method();
ctx = SSL_CTX_new( meth );
    ...
```

dual version server

```
    ...
SSLeay_add_ssl_algorithms();
meth = SSLv23_server_method();
ctx = SSL_CTX_new( meth );
    ...
```

Of course the make file had to be updated for the sslv23 handshake is coded in separate files. Since this change was inevitable, the decision was reached to also implement TLS version 1.0. This change is very simple. Analogue to version23 the make file had to be adjusted for TLS support. TLS version 1.0 uses the same handshake procedure as SSL version 3.0. The only difference between TLS v1 and SSLv3 is that TLSv1 supports all cipher suites from SSLv3 except the Fortezza ciphers.

After the aforementioned corrections and enhancements *serv* functioned well with Netscape Navigator. Netscape does not need the CA certificate that signed the server certificate to function properly. If the distinguished name of a server certificate (at the

same time the domain name) does not match the site called by the client, Netscape merely displays a warning as shown in figure 6.14 (in order to get this warning, the test server was running on the machine *ksy123.zhwin.ch* with the certificate for *mini.zhwin.ch*; hence, the warning showed up).

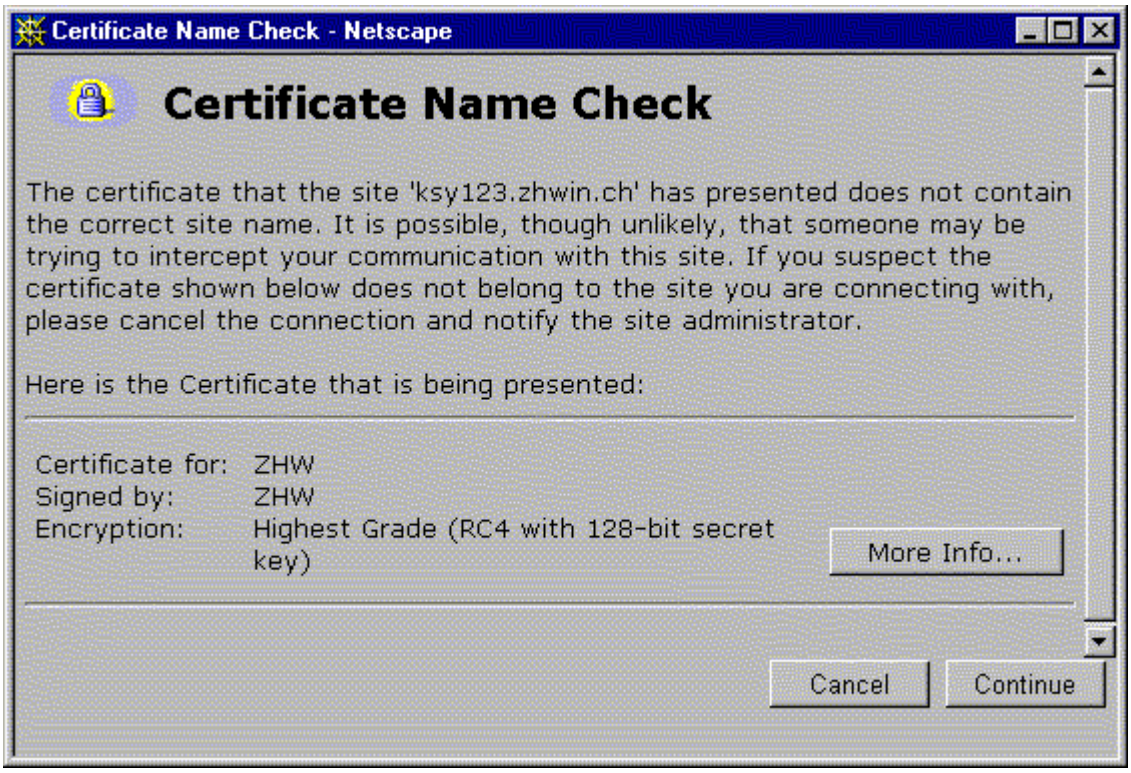


Figure 6.14: Netscape Displays a Warning if Certificate and Domain Name Do Not Match

6.7.2 Microsoft Internet Explorer

As a next step the handshake with MS Internet Explorer 5.5 (128 bit encryption) was examined. Here a strange phenomenon occurred. The first part of the handshake functioned (until the ServerHelloDone message of the sender; refer to section 3.2.2 on page 9). At this point Internet Explorer cancelled the handshake procedure for an unknown reason. The import of the server certificate into Internet Explorer discovered the cause for malfunction: the certificate contained an invalid signature. After generating an entirely new set of certificates (CA certificate, server certificate), MS Internet Explorer did not work properly before importing the Root Certificate. Once this was done, Internet Explorer was capable of communicating with *serv* without problems.

As a summary, MS internet Explorer needs the CA certificate that signed the server certificate to function properly. If the distinguished name of a server certificate (at the same time the domain name) does not match the site called by the client, Internet Explorer refuses to connect and terminates the handshake procedure, whereas Netscape Navigator merely displays a warning.

6.8 Mini Webserver

MWS is an extremely small (executable file size is 101 kilobytes), simple (no configuration except command line) and fast (consumes a minimum of system resources) secure web server supporting SSL version 3.0 and TLS version 1.0.

MWS is a program in an endless loop. Once started, it will appear in memory. There is no way to stop MWS, only a reboot invoked by the system or by the user can stop MWS.

6.8.1 Installing MWS on IPC@CHIP

The first step is to create the right directory structure as described below (Note: use TeraTerm Pro as terminal program to communicate with the IPC@CHIP):

1. `a:\`
2. `md www`
3. `md www\bin` (includes program files)
4. `md www\cgi-bin` (includes cgi binaries)
5. `md www\root` (www home (root))

For installing MWS, the file `[CDROM]\mws\mws.exe` must be copied into the created directory `a:\www\bin` using WS-FTP Lite.

To start the program manually, type `a:\www\bin\mws.exe`. If the program should be loaded automatically after rebooting the system, the file `[CDROM]\mws\autoexec.bat` must be copied into the root directory `a:\`.

In order to initiate an SSL connection, the secure server must have a valid certificate and an RSA private key. (See section 6.4 in this chapter for the generation of a private key and a certificate, and for the conversion of the created files into the right data format.) The file name of the private key must be `s_key.hst`, the certificate's file name `s_cert.pem`. The client can (and probably should) have a certificate. MWS does not currently provide client authentication.

Be reminded that the secure server certificate will be useless without the private key.

For initializing the internal random generator, MWS needs a random seed file. See section 6.6 in this chapter for the generation of the seed file `rand.rnd`.

Now all the important files have been created and must be copied into the directory `a:\www\bin`.

IMPORTANT: Before starting `mws.exe`, make sure the files `s_key.hst`, `s_cert.pem`, and `rand.rnd` are in the same directory as `mws.exe`.

Command Line Options

- The first parameter (optional) is a path to the home directory of MWS. By default it is set to `a:\www\root`.
- The second parameter (optional) is a port number. By default it is set to `443` for HTTPS (Standard SSL / TLS port).

Example

In order to run MWS on port 8000, type the following command line:

```
a:\www\bin\mws.exe a:\www\root 8000
```

6.8.2 Handling of HTTP-requests

The path to the home directory is taken from the command line as the first parameter, and the bind-port as the second; both are optional, do not use them if you are unsure about the correct usage. The default file name in the root directory is index.htm. If index.htm is not found in the root directory, MWS stops execution immediately and displays an error message “index.htm not found”. Make sure that index.htm exists in the directory the first command line parameter points to.

Only GET, HEAD and POST commands are handled.

6.8.3 Handling of CGI Applications

CGI applications are executed from the directory `a:\www\cgi-bin` only. All files requested from `a:\www\cgi-bin` will be treated as CGI programs and executed, even if they are not CGI applications. Hence, non-CGI files must not be stored in that directory.

The samples used to test CGI are provided on the enclosed CD-ROM (`[CDROM]\mws\cgi-bin`).

6.8.3.1 MWS CGI Application Programming Interface (MCAPI)

MWS allows the application programmer to build and install his/her own CGI functions from a DOS program. They will be executed by the MWS after a matching browser request comes in. The MCAPI is fully compatible with the CGI API developed by Beck. However, only the two interrupt functions InstallCGI (interrupt `0xAB`, function `0x01`) and RemoveCGI (interrupt `0xAB`, function `0x02`) are supported. Nevertheless, most existing CGI applications will run without any problems as only few functions make use of the interrupt functions `0x03` to `0x09`.

MCAPI-Functions

`BOOL CGI_install_cgi(const CGI_Entry *pNewEntry);`

pNewEntry points to a variable of the type `struct CGI_Entry` described in the header file mcapi.h. Defined for each entry is the Uniform Resource Locator (URL), the expected HTTP method (GET, HEAD, or POST) and the pointer to the function which will be executed if a matching browser request reaches the MWS. The return value equals 0 if the CGI function has been installed successfully, otherwise the return value is <> 0.

`BOOL CGI_remove_cgi(const char *pszCGIName);`

pszCGIName points to a character string that represents the name of the CGI function that is to be removed. The return value equals 0 if the CGI function has been removed successfully, otherwise the return value is <> 0.

For a full description of the MCAPI refer to the header file `mcapi.h` on the enclosed CD-ROM (`[CDROM]/mws/src`) and the CGI-API description (`[CDROM]/bios/0.69`).

6.8.3.2 MWS Interrupt Handler

The MCAPI provides the interrupt `0xAB` with a service number in the high byte of the AX register. The installation of the interrupt service routine (ISR) `CGI_interrupt_service_routine()` enables access to the CGI implementation of MWS. To feature full compatibility between MCAPI and CGI-API, the ISR has to return values in the registers AX and DX.

The keyword `interrupt` in the function declaration causes the compiler to generate the code sequences darkened in figure 6.15. At the entrance to the ISR all CPU registers are saved to the stack (e.g. register AX is saved with the command: `PUSH AX` in assembler mnemonic). After BP and SP are correctly set, the local variables are generated on the stack as shown in figure 6.16. Now the code between the braces (written by the programmer of the function) is executed. Shortly before leaving the function all registers are restored (e.g., `POP AX`) and the function ends at the `IRET` instruction as it is illustrated in the figure. The darkened chunks of the entry section and the final section of the ISR include the assembler mnemonic to visualize the described process. The problem is that the Interrupt Service Routine is supposed to return values in the registers AX and DX, but as all registers are restored after leaving, the ISR is not capable of returning values in a register. In order to solve this problem, a little trick is needed.
Figure 6.16 shows the stack area while the ISR is in process. SP points to the last stack entry, while the BP points to the first stack entry allocated when the ISR was invoked, in other words to the first item on top of the saved registers. By adding a negative offset to the BP the local variables (e.g. `segvar`, `offvar`) can be addressed. By adding a positive offset the saved registers are pointed to. The instruction `"MOV [BP + 10h], 5"` stores a return value of 5 into “AX”. It is actually stored to the memory AX is saved in, but as the so stored values are popped back into the registers and finally become return values, the value is at last stored in AX.

```
static void interrupt cgi_interrupt_service_routine(void)
{
          push ax              ; save all registers
          push bx
          push cx
          push dx
          push es
          push ds
          push si
          push di
          push bp
          mov bp, 89C1         ; load Data ..
          mob ds, bp           ; .. Segment
          mov bp, 89C1
          mov ds, bp

  // local variables
  unsigned short segvar, offvar; //segment and offset
  unsigned char func; // function code of interrupt
  unsigned short i;

          mov bp, sp           ; set bp
          sub sp, 000A         ; local variables size

  asm mov func, ah
  asm mov segvar, dx

  ...

} // end of function
          leave                ; remove local
          pop di               ; restore all registers
          pop si
          pop ds
          pop es
          pop dx
          pop cx
          pop bx
          pop ax
          iret                 ; end of function call
```

Figure 6.15: ISR Code Sequences.

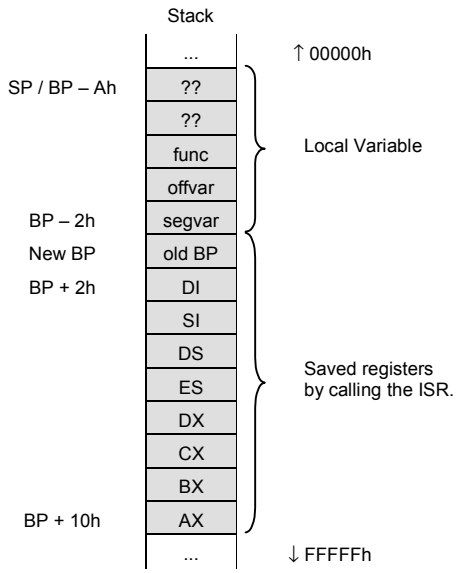


Figure 6.16: Stack After Interrupt Function Call.

6.9 Performance

In consideration of the processor being an antique 80186 operating at 20 MHz, the overall communication performance is surprisingly high. In other words, once the handshake is done there is no remarkable delay due to symmetric RC4 encryption. The only weak point is the RSA decryption of the ClientKeyExchange message (1024 bit). This tasks is highly time-consuming and could cause the user to cancel the handshake. Table 6.2 shows the time needed to perform an RSA decryption:

Key Length [bit]	Decryption Time [s]
1024	44
512	7

Table 6.2: RSA Decryption Time For Different Key Sizes

Therefore the user must be prepared to wait. As access is granted to authorized and instructed personnel only, this should not be a problem.

6.10 Size of the Libraries

Table 6.3 shows a comparison of the library sizes. Important to mention is the fact that under Linux only two libraries were created: *Crypto* and *SSL* (second column). Due to librarian limitations (Borland's *tlib*) these two libraries had to be split up into several smaller libraries (third column). As the source files were not changed at that phase of the project, not every function could be compiled and linked. The SSL library, for example, could not be linked at all (see third column). Therefore the library sizes shown in the table are smaller than they should be. At the end of the project several libraries were not used anymore since the functions were either not invoked, or rewritten by the authors. Some libraries are greater in size than at the beginning. This is caused by either the incomplete compiling, as explained earlier, or because the functions were declared as `inline` for means of performance.

	Original (Linux)	Beginning of Project	End of Project
ASN1	-	379	not used
BIO	-	49	59
BN	-	147	86
Crypto	1328	-	-
Crypto1	-	27	21
Crypto2	-	100	80
Evp	-	115	91
MD5	-	20	9
PEM	-	45	not used
PKCS12	-	75	not used
PKCS7	-	46	not used
RC4	-	1	5
RSA	-	44	38
SHA	-	1	19
SSL	249	0	174
X509	-	136	not used
X509v3	-	144	not used
Total	1577	1329	**582**

Table 6.3: Library Sizes of SSL for IPC@CHIP in Kilobytes

As a summary, the libraries created throughout the project are approximately 65 percent smaller than before. That is a huge step toward a successful implementation of the SSL stack on IPC@CHIP.

A list of all files that were altered is available on the enclosed CD-ROM (`[CDROM]/SSLforIPC_CHIP/openssl/altered.txt`).

7 Manual - SSL for IPC@CHIP

7.1 Installation Procedure

The proposition of the authors is that you use Borland C++ 3.1 for DOS. If you cannot get hold of that development tool, you can use any other compiler that builds 16-bit code for the Intel 80186 processor.

7.1.1 Libraries Only

Copy the library (`[CDROM]\SSLforIPC_CHIP\libraries`) and header files (`[CDROM]\SSLforIPC_CHIP\headers`) from the enclosed CD-ROM to your local hard disk drive. Enter the path to the directories containing the copied files into your project, and you are done.

7.1.2 Source Code

Copy the directory `[CDROM]\SSLforIPC_CHIP\openssl` including the subdirectories to your local hard disk drive (preferably to `g:\`). Run the make file (`openssl\ms\opsslbc.mak`) as described in section 6.5.2 in this chapter. After successful termination of the *make* utility the libraries are located in `openssl\out16bc`, the headers in `openssl\inc16bc`.

IMPORTANT NOTICE: The make file makes the assumption that the directory `openssl` is located in drive `g:\` by default. Should your configuration differ from the one described in chapter 4: Development Environment, you must adjust the path entries at the beginning of the make file manually.

7.2 Programmer's Guide

There is a random seed file available on the enclosed CD-ROM (`[CDROM]/mini`), but for security reasons you should generate your own seed file. A nice tool to perform this task is PGP (a variety of PGP versions are available under [10]).

7.2.1 Primer on SSL Server Programming

The following paragraph contains several points of importance. The numbers correspond to the numbers of the darkened code sequences in the subsequent section. The order must remain the same as shown below.

1. **Randomness**
   The first thing a program should do is initialize the Pseudo Random Number Generator. As explained in section 6.6 in the last chapter this is done by loading a random seed file and seeding the PRNG. This is done by calling the function `RAND_load_file()`. After loading the file it is highly important to check the status of the PRNG. The function `RAND_status()` returns `1` if the PRNG has been seeded with enough data, `0` otherwise. If PRNG has not been seeded correctly (with enough random data, to be precise), the generated numbers will not be random. Using the function `RAND_write_file()`, the state of the PRNG is written to the random seed file. IMPORTANT: This function must be invoked prior to the functions `SSL_read()` and `SSL_write()`. The reason is that the function call after reading or writing data through the secured channel caused a system error. So far it is not apparent whether this is a bug in the program sequence, in the compiler, or in the BIOS. Nevertheless, no error occurred when `RAND_write_file()` was called right after `SSL_accept()`.

2. **Server Method**
   As explained in section 6.7 in the last chapter the function `SSLv23_server_method()` must be invoked to configure the correct handshake function.

3. **Certificate File**
   Because the certificate file is saved in the DER format rather than in the "old" PEM format, the function `SSL_CTX_use_certificate_file()` was removed from the program. Within `SSL_accept()` a new function is called that reads the certificate in DER format and feeds it into the Certificate message. Hence, no explicit call of a read function is required.

4. **Private Key File**
   Prior to the handshake procedure the private key must be loaded. This is done by calling the function `SSL_CTX_MINI_use_privatekey_file()`, which replaces `SSL_CTX_use_privatekey_file()`

5. **TCP Handshake**
   The TCP handshake is done like in any other program.

6. **SSL Handshake**
   The entire SSL handshake is handled in one single function: `SSL_accept()`. As mentioned in the previous paragraph “Certificate File”, the reading and sending of the server certificate is handled in this function. A return value of –1 signals a handshake error. In this case the server program must either go into listening again, or terminate.

7. **Securely Read and Write**
   For reading from and writing to the secure communications channel, the functions `SSL_read()` and `SSL_write()` are used. The usage is similar to the read and write function that operate on a TCP connection.

7.2.2 Sample Code

```
#include <tcpip.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <errno.h>
#include <sys/types.h>
#include <openssl/rsa.h>
#include <openssl/crypto.h>
#include <openssl/x509.h>
#include <openssl/pem.h>
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <openssl/rand.h>

#define HOME   "a:\\"
#define CERTF  HOME "s_cert.pem"
#define KEYF   HOME "s_key.hst"
#define RANDF  HOME "rand.rnd"
#define PORT 443

#define HTML_PAGE "<HTML><HEAD><TITLE>SSL Encrypted Transmission</TITLE></HEAD> \
<BODY><P><BIG><BIG><B>SSL Testpage</B></BIG></BIG><P><P>The contents of this page \
have been transmitted using Secure Sockets Layer.</BODY></HTML>"

#define CHK_NULL(x) if ((x)==NULL) exit ( EXIT_FAILURE )
#define CHK_ERR(err,s) if ((err)==-1) { perror(s); exit( EXIT_FAILURE ); }

void main ()
{
    int err;
    int listen_sd;
    int sd;
    struct sockaddr_in sa_serv;
```

```
    struct sockaddr_in sa_cli;
    int client_len;
    SSL_CTX* ctx;
    SSL*     ssl;
    char     buf [4096];
    SSL_METHOD *meth;

    /* get randomness */
    RAND_load_file( RANDF, -1 ); /* -1: read entire file */
    if (!RAND_status()) {
        printf("not enough randomness\n");  exit( EXIT_FAILURE );
    }
    printf("\n\nseeding PRNG: ok\n");

    SSLeay_add_ssl_algorithms();
    printf( "adding SSL algorithms: ok\n" );
    meth = SSLv23_server_method();
    ctx = SSL_CTX_new( meth );
    if (!ctx) {
        printf( "SS_CTX_new: error\n" );  exit( EXIT_FAILURE );
    }
    printf( "SSL_CTX_new: ok\n" );

/*  if (SSL_CTX_use_certificate_file( ctx, CERTF, SSL_FILETYPE_PEM ) != 0) {
      printf( "use_certificate_file failed\n" );
      exit( EXIT_FAILURE );
  }
  printf("use_certificate_file failed\n");
*/

    if ((err = SSL_CTX_MINI_use_privatekey_file( ctx, KEYF )) != 0) {
        printf( "use_privatekey_file failed: %d\n", err );  exit( EXIT_FAILURE );
    }
    printf( "using private key file: ok\n" );
/*
  if (!SSL_CTX_check_private_key( ctx )) {
      printf( "Private key does not match the certificate public key\n" );
      exit( EXIT_FAILURE );
  }
*/

    /* ------------------------------------------------ */
    /* Prepare TCP socket for receiving connections */

    listen_sd = socket( AF_INET, SOCK_STREAM, 0 );
    CHK_ERR( listen_sd, "socket" );

    memset( &sa_serv, '\0', sizeof( sa_serv ) );
    sa_serv.sin_family      = AF_INET;
    sa_serv.sin_addr.s_addr = 0;
    sa_serv.sin_port        = htons( PORT );   /* Server Port number */

    err = bind( listen_sd, (struct sockaddr*) &sa_serv, sizeof( sa_serv ) );
    CHK_ERR( err, "bind" );

    /* Receive a TCP connection. */
    err = listen( listen_sd, 5 );
    CHK_ERR( err, "listen" );
```

1

2

3

4

5

```
    client_len = (int)sizeof( sa_cli );
    printf( "listening on port %d...\n", PORT );
    sd = accept( listen_sd, (struct sockaddr*) &sa_cli, &client_len );
    CHK_ERR( sd, "accept" );
    closesocket (listen_sd);

    /* ---------------------------------------------- */
    /* TCP connection is ready. Perform SSL handshake. */

    ssl = SSL_new( ctx );
    if (!ssl) {
        printf( "SSL_new failed\n" );  exit( EXIT_FAILURE );
    }
    printf( "SSL_new ok\n" );

    SSL_set_fd( ssl, sd );
    err = SSL_accept( ssl );
    if (err == -1) {
        printf( "SSL_accept: failed\n" );  exit( EXIT_FAILURE );
    }
    printf( "SSL_accept: ok\n" );

    if (RAND_write_file( RANDF ) <= 0) {
        printf( "writing random file to disk: failed\n" );  exit( EXIT_FAILURE );
    }
    printf( "writing random file to disk: ok\n" );

    /* ---------------------------------------------- */
    /* Data Exchange - Receive message and send reply. */

    err = SSL_read( ssl, buf, sizeof( buf ) - 1 );
    if (err == -1) {
        printf( "reading encrypted data: failed\n" );  exit( EXIT_FAILURE );
    }
    buf[err] = '\0';
    printf( "Got %d chars:'%s'\n", err, buf );

    err = SSL_write( ssl, HTML_PAGE, strlen( HTML_PAGE ) );
    if (err == -1) {
        printf( "SSL_write: Error\n" );  exit( EXIT_FAILURE );
    }
    printf( "SSL_write: done\n" );

    /* Clean up. */
    printf( "cleaning up...\n" );
    closesocket( sd );
    SSL_free( ssl );
    printf( "SSL_free: ok\n" );
    SSL_CTX_free( ctx );
    printf( "SSL_CTX_free: ok\n" );
}
```

5

6

1

7

8 Conclusion & Prospects

Looking back at the diploma project, a lot of time was spent on the search for documentation and information about OpenSSL, because in order to reduce the code, we had to understand OpenSSL (at least partly).
At some times it was hard to concentrate and to keep going, because both authors would have preferred to write SSL on their own instead of reducing the code. Hence they enjoyed replacing the certificate functions and especially the implementation of the interrupt service routine.

As a summary, the work on this project was interesting. The authors gained more insight into the internals of the SSL protocol. This certainly is a great improvement as security technologies become of major importance these days.

The co-operation of the authors was of mutual respect and produced a result to be proud of. The sharing of ideas, visions and knowledge was a great experience and an improvement of individual know-how.

A view into the future must not be missing at the end of this interesting project. In the software business there never is a lack of motivation and funds for future improvements. In the following list there are a few point mentioned that certainly are of interest for further developing. The order shall not be an indication for importance or individual preferences.

- **Performance Tuning for RSA Decryption**
  As mentioned in the report the decryption of the RSA-encrypted ClientKeyExchange needs about 44 seconds. Faster decryption certainly would be appreciated.

- **Client Authentication**
  If client authentication is implemented, the handling of authorized personnel will become much easier.

- **Additional Cipher Suites**
  With additional cipher suites compatibility increases. There might also be the need for stronger encryption algorithms as time passes by.

- **Better Web Server**
  mws is a web server that only allows one user at a time; multithreading for the web server is would be a great enhancement.

- **Secure FTP and Secure Telnet**
  The ftp and telnet servers are the weak points for a secure IPC@CHIP. Right now they have to be shut down in order to restrict unauthorized access. This hinders the update of secure server software. Secure FTP or a SSH server would enable a totally secured environment.

9 Glossary

Abstract Syntax Notation One (ASN.1). A Language for describing data and data objects, used to define X.509 public key certificates.

Alert Protocol. A component of the SSL protocol that defines the format of Alert messages.

Application Protocol. An application protocol is a protocol that normally layers directly on top of TCP/IP. For example: HTTP, TELNET, FTP, and SMTP.

Asymmetric Encryption. The technical term for public key encryption in which two different keys are used for encryption and decryption. One of the keys can be revealed publicly without compromising security.

Asymmetric Key Cryptography. Cryptography based on asymmetric encryption. Depending on the particular algorithms employed, asymmetric key cryptography can provide encryption/decryption or digital signature services.

Authentication. A security services that validates the identity of a communicating party.

Block Cipher. A cipher that encrypts and decrypts data only in fixed-size blocks.

Certificate. A public key certificate, digital information that identifies a subject and that subject's public key and is digitally signed by an authority that certifies the information it contains.

Certificate Authority (CA). An organization that issues certificates and vouches for the identities of the subjects of those certificates; also known as an issuer.

Cipher. An algorithm that encrypts and decrypts information.

Cipher Suite. A cipher algorithm and the parameters necessary to specify its use (e.g. size of keys).

Ciphertext. Information that has been encrypted using a cipher.

Client. The party that initiates communications. Clients communicate with servers.

Client Authentication. Client authentication allows a server to confirm a client's identity. SSL enabled server can use standard techniques of public key cryptography to check that a client's certificate is valid and has been issued by a certificate authority. This confirmation might be important if the server, for example, is a bank sending confidential financial information to a customer and wants to check the recipient's identity.

Cryptanalysis. The science concentrating on the study of methods and techniques to defeat cryptography.

Cryptography. The science concentration on the study of method and techniques to provide security by mathematical manipulation of information.

Decryption. The complement of encryption, recovering the original information from encrypted data.

Diffie-Hellman. A key exchange algorithm developed by W. Diffie and M.E. Hellman. First published in 1976.

Digest Function. A cryptographic function that creates a digital summary of information so that, if the information is altered, the summary (known as a hash) will also change. It is also known as a hash function.

Digital Signature Algorithm (DSA). An asymmetric encryption algorithm published as a U.S. standard by the National Institutes of Science and Technology. DSA can only be used to sign data.

Distinguished Encoding Rules (DER). A process for unambiguously converting an object specified in ASN.1 into binary values for storage or transmission on a network.

Distinguished Name. The identity of a subject or issuer specified according to a hierarchy of objects defined by the ITU.

Encryption. The process of applying a cipher algorithm to information, resulting in data that is unintelligible to anyone who does not have sufficient information to reverse the encryption.

File Transfer Protocol (FTP). An Internet application protocol for transferring files among computer systems. SSL can provide security for FTP communications.

Hash Function. A cryptographic function that creates a digital summary of information so that, if the information is altered, the summary (known as a hash) will also change. It is also known as a digest function.

HyperText Transfer Protocol (HTTP). The application protocol for Web browsing. SSL can add security to HTTP applications.

International Telecommunications Union (ITU). An international standards body responsible for communications protocols. The ITU published the X.509 standards for public key certificates.

Internet Engineering Task Force (IETF). An international standards body responsible for Internet protocols. The IETF publishes the Transport Layer Security specifications.

Internet Protocol (IP). The core network protocol for the Internet. IP is responsible for routing messages from their source to their destination.

Issuer. An organization that issues certificates and vouches for the identities of the subjects of those certificates. It is also known as a certificate authority.

Key. Information needed to encrypt or decrypt data. To preserve security, symmetric encryption algorithms must protect the confidentiality of all keys, while asymmetric encryption algorithms need only protect private keys.

Key Exchange Algorithm. An algorithm that allows two parties to agree on a secret key without actually transferring the key value across an insecure channel. The best known example is the Diffie-Hellman key exchange.

MAC Read Secret. A secret value input to a message authentication code algorithm for verifying the integrity of received data. One party's MAC read secret is the other party's MAC write secret.

MAC Write Secret. A secret value input to a message authentication code algorithm to generate message authentication codes for data that is to be transmitted. One party's MAC write secret is the other party's MAC read secret.

Master Secret. The value created as the result of SSL security negotiations, from which all secret key material (MAC read/write secret, secret read/write key) is derived.

Message Authentication Code (MAC). An algorithm that uses cryptographic technology to create a digital summary of information so that, if the information is altered, the summary (kwon as a hash) will also change.

Message Digest 5 (MD5). A digest function designed by Ron Rivest and used extensively by SSL. It converts an arbitrarily long data stream into a digest of fixed size.

Pretty Good Privacy (PGP). PGP is the world's defacto standard for email encryption and authentication.

Plaintext. Information in its unencrypted (and vulnerable) form before encryption or after decryption.

Premaster Secret. An intermediate value SSL implementation uses in the process of calculating key material for a session. The client usually creates the premaster secret from random data and sends it to the server in a ClientKeyExchange message.

Privacy. Privacy is the ability of two entities to communicate without fear of eavesdropping. Privacy is often implemented by encrypting the communications stream between the two entities.

Private Key. One of the keys used in asymmetric cryptography. It cannot be publicly revealed without compromising security, but only one party to a communication needs to know its value.

Pseudorandom Number. A number generated by a computer that has all the properties of a true random number.

Pseudo Random Number Generator (PRNG). A Function that generates truly random numbers if seeded properly. PRNG makes use of SHA1.

Public Key. One of the keys used in asymmetric cryptography. It can be publicly revealed without compromising security.

Public Key Certificate. Digital information that identifies a subject and that subject's public key and that is digitally signed by an authority that certifies the information it contains.

Public Key Cryptography. Cryptography based on asymmetric encryption in which two different keys are used for encryption and decryption. One of the keys can be revealed publicly without compromising the other key.

Rivest Cipher 4 (RC4). A stream cipher developed by Ron Rivest.

Rivest Shamir Adleman (RSA). An asymmetric encryption algorithm named after its three developers. RSA supports both encryption and digital signatures.

Secret Key. A key used in symmetric encryption algorithms and other cryptographic functions in which both parties must know the same key information.

Secret Key Cryptography. Cryptography based on symmetric encryption in which both parties must process the same key information.

Secure Hash Algorithm (SHA). A hash algorithm published as U.S. standard by the National Institutes of Science and Technology.

Secure Hash Algorithm 1 (SHA1). SHA1 is the successor of SHA. Programs should use SHA only when backward compatibility is required.

Secure Read Key. The secret key used to decrypt messages. One party's secret read key is the other party's secret write key.

Secure Sockets Layer (SSL). A separate network security protocol developed by Netscape and widely deployed for securing Web transactions.

Secure Write Key. The secret key used to encrypt messages. One party's secret write key is the other party's secret read key.

Server. The party in a communication that receives and responds to requests initiated by the other party.

Server Authentication. Server authentication allows a client to confirm a server's identity. SSL enabled client software can use standard techniques of public key cryptography to check that a server's certificate is valid and have been issued by a certificate authority. This confirmation might be important if the user, for example, is sending a credit card number over the network and wants to check the receiving server's identity.

Session. A Session is an association between a client and a server. Sessions are created by the handshake protocol. Sessions define a set of cryptographic security parameters, which can be shared among multiple connections. Sessions are used to avoid the expensive negotiation of new security parameters for each connection.

Session Identifier. The value SSL servers assign to a particular session so that it may be resumed at a later point with full renegotiation.

Session Key. In SSL there are four keys that are called session keys: client secret read key, client secret write key, server secret read key and server secret write key.

Signature. The encryption of information with a private key. Anyone possessing the corresponding public key can verify that the private key was used, but only a party with the private key can create the signature.

Stream Cipher. A cipher that can encrypt and decrypt arbitrary amounts of data, in contrast to the block ciphers.

Subject. The party who possesses a private key and whose identity is certified by public key certificate.

Symmetric Encryption. The technical term for secret key encryption in which encryption and decryption require the same key information.

Symmetric Key Cryptography. Cryptography based on symmetric encryption. Depending on the particular algorithms employed, symmetric key cryptography can provide encryption/decryption and message integrity services.

Transmission Control Protocol (TCP). A core protocol of the Internet that ensures the reliable transmission of data from source to destination.

Transport Layer Security (TLS). The IETF standard version of the Secure Sockets Layer protocol.

X.509. An ITU standard for public key certificates.

Appendices

A Bibliography

[1] Thomas, Stephen.
***SSL and TLS Essentials**. Securing the Web.*
Wiley Computer Publishing, ISBN 0471383546

[2] Achtert, Werner.
***Das grosse Buch zu C++**. Erfolgreich programmieren mit C++.*
Data Becker, ISBN 3815811775

[3] How SSL Works
http://developer.netscape.com/tech/security/ssl/howitworks.html

[4] Introduction to Public-Key Cryptography
http://developer.netscape.com/docs/manuals/security/pkin/index.htm

[5] http://www.beck-ipc.com

[6] http://www.openssl.org

[7] http://www.pca.dfn.de/dfnpca/certify/ssl/handbuch

[8] ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1d1.pdf

[9] http://www.strongsec.ch/freeswan/index.htm

[10] ftp://ftp.hacktic.nl/pub/crypto/pgp/

B Configuration of the IPC@CHIP

The IPC@CHIP can be reached via the DNS entry *mini.zhwin.ch* via the IP address *160.85.134.67*.

SC12 Serial Number	0016F
MAC Address	00 30 56 F0 01 6F
IP Address	160.85.134.67
Subnet Mask	255.255.240.0
Gateway	160.85.128.1

Table B.1: Configuration of the IPC@CHIP

All files and libraries were compiled using the file `chip.ini` that is shown below.

```
[IP]
DHCP=0
ADDRESS=160.85.134.67
NETMASK=255.255.240.0
GATEWAY=160.85.128.1

[DEVICE]
NAME=MINI

[FTP]
ENABLE=1

[TELNET]
ENABLE=0
```

C Contents of the Enclosed CD-ROM

Figure C.1 shows the contents of the enclosed CD-ROM.

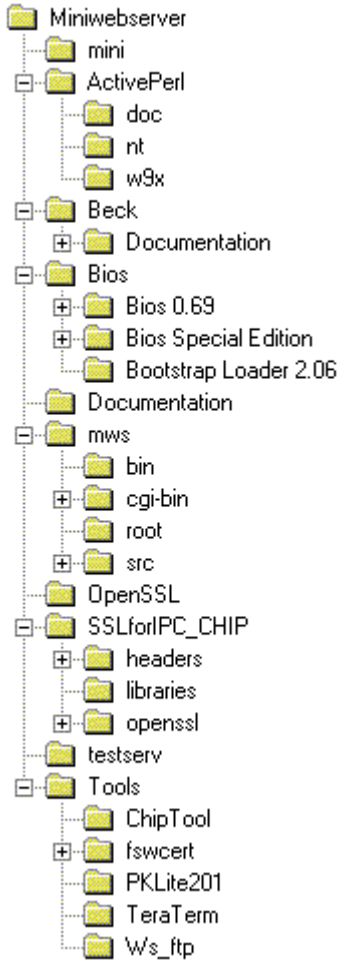


Figure C.1: Contents of the CD-ROM

D Original Task Description

Kommunikationssysteme (KSy)

Praktische Diplomarbeiten 2000 - Sna00/3

Mini-Webserver mit SSL

Studierende:

- Bernhard Lenzlinger, IT3b
- Andreas Zingg, IT3b

Termine:

- Ausgabe: Donnerstag, 7.09.2000 8:00 - 10:00 im E509
- Abgabe: Montag, 30.10.2000

Beschreibung:

Mit dem IPC@CHIP von Beck lässt sich für wenig Geld ein Web-Server mit integrierter Ethernet-Schnittstelle realisieren, der über 24 V Ein- und Ausgänge direkt Hardwareschnittstellen bedienen kann. Damit eröffnet sich die Möglichkeit, über das Internet weltweit Geräte ansteuern zu können. Als Konsequenz dieser globalen Vernetzung rücken natürlich Sicherheitsaspekte in den Vordergrund. Der Zugriff auf diese Geräte sollte nur Berechtigten vorbehalten werden. Weil offen über das Internet gesendete Passwörter leicht abgehört werden können, sollte ein Challenge/Response Protokoll für die Authentisierung verwendet werden. Zusätzlich sollte der gesamte Datenaustausch verschlüsselt werden können. Als Standardlösung bietet sich zu diesem Zweck der Secure Sockets Layer (SSL) an, der es erlaubt, mit dem Web-Server sicher zu kommunizieren.

In dieser Diplomarbeit soll der C-Source Code der OpenSSL-Library auf den IPC@CHIP portiert und damit ein SSL-fähiger Mini-Webserver realisiert werden.

Aufgaben:

- Bestimmung des minimalen Anteils der OpenSSL-Library, der notwendig ist, damit folgende Anforderungen erfüllt werden:
  - Unterstützung des SSL v3 Protokolls mit den untenstehenden Einschränkungen
  - Authentisierung des Mini-Webservers mittels eines Server-Zertifikats
  - Keine Client-Zertifikate
  - RC4 Verschlüsselung mit einem 128 Bit Schlüssel
  - MD5 MAC zur Authentisierung

- Portierung dieses minimalen Anteils des SSL-Stacks auf den IPC@CHIP Prozessor
- Realisierung eines minimalen https-Servers auf Port 443, der ein sicher übermitteltes Passwort entgegennimmt, prüft und anschliessend eine Resultatseite zurückgibt und oder eine I/O Schnittstelle des IPC@CHIP Prozessors steuert.
- Erstellen einer Installations- und Betriebsanleitung für den portierten SSL Stack und den https-Miniserver.
- Dokumentation der Diplomarbeit.

Infrastruktur / Tools:

- Raum: **E416**
- Rechner: 2 PCs, 2 SC12 IPC@CHIP Prozessoren mit DK40 Development Kit
- SW-Tools: C/C++ Compiler für 80186 Prozessoren, OpenSSL C-Library

Literatur / Links:

- Beck IPC@CHIP Seite
  http://www.beck-ipc.com/chip/
- OpenSSL Project
  http://www.openssl.org
- OpenSSL Handbuch
  http://www.pca.dfn.de/dfnpca/certify/ssl/handbuch/
- IETF Draft <draft-freier-ssl-version3-02.txt>
  The SSL Protocol Version 3.0
- SSL 3.0 Implementation Assistance
  http://home.netscape.com/eng/ssl3/traces/index.html
- Thomas, Stephen
  SSL and TLS Essentials, Wiley Computer Publishing, ISBN 0-471-38354-6.

Winterthur, 7. September 2000

Dr. Andreas Steffen

E Certificates

E.1 Root CA Certificate

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 0 (0x0)
        Signature Algorithm: md5WithRSAEncryption
        Issuer: C=CH, L=Winterthur, O=ZHW, OU=DA Sna00/3, CN=RootCA
        Validity
            Not Before: Oct  9 12:08:05 2000 GMT
            Not After : Oct  9 12:08:05 2002 GMT
        Subject: C=CH, L=Winterthur, O=ZHW, OU=DA Sna00/3, CN=RootCA
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (2048 bit)
                Modulus (2048 bit):
                    00:c8:bd:60:25:ce:64:62:88:d9:b0:b3:b5:57:6b:
                    af:e2:3b:22:62:2b:4e:48:9b:b8:98:80:ef:f3:bc:
                    70:e0:f4:5e:56:3d:2d:3e:ec:89:4e:3a:ce:06:f8:
                    e3:4a:1e:75:cd:1c:f2:da:87:af:1d:e7:76:a2:8b:
                    65:2f:22:84:63:c3:9a:33:45:db:93:1b:2f:41:5e:
                    52:dd:9f:a5:70:f5:9f:77:c4:1d:76:93:8d:be:b2:
                    ab:0d:84:53:31:81:a4:7f:46:a6:70:c9:f8:70:35:
                    7b:ed:43:dc:41:7e:c3:4a:db:be:7e:16:76:6b:c5:
                    1c:b5:ac:56:48:fd:aa:4b:5b:e6:34:a6:0b:fb:2c:
                    47:46:67:5e:df:a5:41:73:7f:1c:ee:6e:c2:b5:6d:
                    72:3d:02:ac:fb:34:45:4b:bc:7e:1c:f9:15:26:3d:
                    11:77:26:f8:f6:4e:71:68:f5:70:0c:b7:a5:21:dc:
                    8a:1f:ac:db:fa:d9:ce:59:7f:a6:eb:60:93:fa:55:
                    a7:f8:19:23:f2:ec:20:e2:6a:bd:90:0a:dd:c0:8b:
                    d7:af:c4:48:aa:37:f2:9c:b4:e6:24:dc:02:d5:bd:
                    16:15:37:77:1f:18:80:47:16:20:81:4d:30:c7:5b:
                    23:64:c8:76:df:e0:3a:25:ea:7a:7a:ce:2a:fb:76:
                    c7:a1
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                63:DD:BF:C6:92:78:6B:08:BB:69:A2:63:84:9F:E7:08:1E:9D:A4:33
            X509v3 Authority Key Identifier:
                keyid:63:DD:BF:C6:92:78:6B:08:BB:69:A2:63:84:9F:E7:08:1E:9D:A4:33
                DirName:/C=CH/L=Winterthur/O=ZHW/OU=DA Sna00/3/CN=RootCA
                serial:00

            X509v3 Basic Constraints:
                CA:TRUE
            Netscape Cert Type:
                SSL CA, S/MIME CA, Object Signing CA
    Signature Algorithm: md5WithRSAEncryption
        5d:94:bf:06:1b:b8:1c:fb:fd:9a:2c:a9:69:9a:24:f2:9a:1c:
        3d:d2:2d:f8:6d:a3:e3:30:c9:0d:82:0f:ab:52:17:95:82:8a:
        06:5f:f7:df:1d:d0:7e:03:b2:36:5a:37:66:71:54:2d:29:b1:
        18:00:77:4b:07:db:30:f6:52:5d:72:bb:04:25:80:7a:8b:3d:
```

```
        8f:3e:e2:0e:88:97:4a:a6:f8:6b:c4:04:c6:0e:5c:61:05:64:
        0c:96:5d:b3:1b:bd:b0:6e:7c:57:8d:39:43:63:8d:f9:26:44:
        9d:64:22:f9:5b:4c:6c:cd:a9:b4:c7:a0:03:7c:aa:8f:30:16:
        ee:9c:c4:5f:ac:a0:d2:ec:10:85:85:50:55:39:c4:a1:5b:aa:
        4f:e1:84:8b:94:4a:60:cd:80:99:5b:25:02:73:a4:a0:de:66:
        7a:25:00:ae:e1:d6:c7:53:85:fe:76:16:c5:0f:ca:d9:b7:6b:
        f4:b3:e8:67:8c:56:9e:f0:c1:44:56:4d:8c:f3:1c:af:db:ba:
        ce:90:3f:74:94:ee:a5:bc:f5:3e:71:01:d4:0d:75:0c:98:fe:
        b1:ee:42:eb:18:2d:72:5c:69:f0:16:65:a9:6a:e7:c1:25:23:
        33:b8:1a:e9:2e:b5:69:7b:37:28:6c:99:cc:86:15:2c:86:2e:
        b6:2c:ee:86
-----BEGIN CERTIFICATE-----
MIID6zCCAtOgAwIBAgIBADANBgkqhkiG9w0BAQQFADBWMQswCQYDVQQGEwJDSDET
MBEGA1UEBxMKV2ludGVydGh1cjEMMAoGA1UEChMDWkhXMRMwEQYDVQQLEwpEQSBT
bmEwMC8zMQ8wDQYDVQQDEwZSb290Q0EwHhcNMDAxMDA5MTIwODA1WhcNMDIxMDA5
MTIwODA1WjBWMQswCQYDVQQGEwJDSDETMBEGA1UEBxMKV2ludGVydGh1cjEMMAoG
A1UEChMDWkhXMRMwEQYDVQQLEwpEQSBTbmEwMC8zMQ8wDQYDVQQDEwZSb290Q0Ew
ggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQDIvWAlzmRiiNmws7VXa6/i
OyJiK05Im7iYgO/zvHDg9F5WPS0+7IlOOs4G+ONKHnXNHPLah68d53aii2UvIoRj
w5ozRduTGy9BXlLdn6Vw9Z93xB12k42+sqsNhFMxgaR/RqZwyfhwNXvtQ9xBfsNK
275+FnZrxRy1rFZI/apLW+Y0pgv7LEdGZ17fpUFzfxzubsK1bXI9Aqz7NEVLvH4c
+RUmPRF3Jvj2TnFo9XAMt6Uh3IofrNv62c5Zf6brYJP6Vaf4GSPy7CDiar2QCt3A
i9evxEiqN/KctOYk3ALVvRYVN3cfGIBHFiCBTTDHWyNkyHbf4Dol6np6zir7dseh
AgMBAAGjgcMwgcAwHQYDVR0OBBYEFGPdv8aSeGsIu2miY4Sf5wgenaQzMH4GA1Ud
IwR3MHWAFGPdv8aSeGsIu2miY4Sf5wgenaQzoVqkWDBWMQswCQYDVQQGEwJDSDET
MBEGA1UEBxMKV2ludGVydGh1cjEMMAoGA1UEChMDWkhXMRMwEQYDVQQLEwpEQSBT
bmEwMC8zMQ8wDQYDVQQDEwZSb290Q0GCAQAwDAYDVR0TBAUwAwEB/zARBglghkgB
hvhCAQEEBAMCAAcwDQYJKoZIhvcNAQEEBQADggEBAF2UvwYbuBz7/ZosqWmaJPKa
HD3SLfhto+MwyQ2CD6tSF5WCigZf998d0H4DsjZaN2ZxVC0psRgAd0sH2zD2Ul1y
uwQlgHqLPY8+4g6Il0qm+GvEBMYOXGEFZAyWXbMbvbBufFeNOUNjjfkmRJ1kIvlb
TGzNqbTHoAN8qo8wFu6cxF+soNLsEIWFUFU5xKFbqk/hhIuUSmDNgJlbJQJzpKDe
ZnolAK7h1sdThf52FsUPytm3a/Sz6GeMVp7wwURWTYzzHK/bus6QP3SU7qW89T5x
AdQNdQyY/rHuQusYLXJcafAWZalq58ElIzO4GukutWl7NyhsmcyGFSyGLrYs7oY=
-----END CERTIFICATE-----
```

E.2 Server Certificate

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1 (0x1)
        Signature Algorithm: md5WithRSAEncryption
        Issuer: C=CH, L=Winterthur, O=ZHW, OU=DA Sna00/3, CN=RootCA
        Validity
            Not Before: Oct 11 08:10:26 2000 GMT
            Not After : Oct 11 08:10:26 2001 GMT
        Subject: C=CH, O=ZHW, OU=DA Sna00/3, CN=mini.zhwin.ch
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:c1:4a:d0:e6:2e:a8:d6:75:1c:f2:b8:fa:b0:98:
                    35:ab:b0:20:92:fd:fd:3f:07:55:7c:c2:3b:2d:53:
                    98:69:b2:80:3c:c5:19:eb:0e:d9:a0:aa:ce:1f:23:
                    33:6c:73:aa:1f:81:fb:e0:52:98:11:95:3c:5f:56:
                    5e:a1:4f:14:0f:31:52:23:43:e7:1f:02:8e:c7:e5:
                    bd:1c:14:27:a5:4f:0f:21:36:69:69:af:55:63:81:
```

```
                    66:02:7e:aa:a4:0a:00:b6:8a:38:d4:63:ce:b2:4c:
                    0d:b1:9a:68:97:00:11:77:11:54:28:af:91:f1:23:
                    1b:8c:91:3f:e3:99:d0:24:13
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            Netscape Cert Type:
                SSL Server
            Netscape Comment:
                OpenSSL Generated Certificate
            X509v3 Subject Key Identifier:
                BF:40:73:47:8A:DD:3D:5B:D5:5F:96:B4:5E:F5:23:0E:22:5F:E2:08
            X509v3 Authority Key Identifier:
                keyid:63:DD:BF:C6:92:78:6B:08:BB:69:A2:63:84:9F:E7:08:1E:9D:A4:33
                DirName:/C=CH/L=Winterthur/O=ZHW/OU=DA Sna00/3/CN=RootCA
                serial:00

            Netscape SSL Server Name:
                mini.zhwin.ch
    Signature Algorithm: md5WithRSAEncryption
        81:12:45:ce:4b:c8:66:a5:b8:ea:b2:ca:87:f2:30:03:38:e8:
        45:ac:14:df:9e:61:b8:90:68:7b:6b:ed:4a:76:29:e0:0f:e2:
        19:c0:be:a3:82:bf:ea:47:d2:19:36:98:b8:ae:0e:e5:08:ae:
        fa:29:02:1f:e1:10:8e:db:02:cc:6b:3e:d6:39:ba:a9:c7:9a:
        3e:fe:02:0a:8c:81:29:40:58:5c:58:30:3d:9e:d5:57:4a:90:
        fc:25:8a:14:a3:8d:7c:db:2f:b8:1d:3d:5a:f5:37:57:fc:99:
        55:7f:c2:64:79:ad:4e:29:59:d9:fc:29:2a:58:2e:48:25:2f:
        1e:03:6c:52:bb:bd:cf:2d:95:12:a4:05:fe:d4:d3:21:5b:6d:
        0d:05:58:12:65:45:51:01:3b:d3:da:fc:fe:56:8b:44:25:bc:
        60:d1:4f:05:1e:fa:80:24:2e:90:20:33:a8:44:08:7f:3f:ee:
        6f:a3:a0:f6:ee:30:90:e1:10:3f:2b:93:9e:33:0b:65:be:8e:
        82:91:ba:48:5a:44:25:eb:53:60:23:c4:bf:f1:6d:a4:c0:a1:
        de:8c:a8:52:9b:64:81:ab:77:ab:b5:a8:df:9a:e5:8f:4e:fb:
        fd:19:3f:2a:dc:c4:cb:18:12:8b:d9:e2:a8:7d:5c:b4:50:81:
        7d:7d:73:65
-----BEGIN CERTIFICATE-----
MIIDpDCCAoygAwIBAgIBATANBgkqhkiG9w0BAQQFADBWMQswCQYDVQQGEwJDSDET
MBEGA1UEBxMKV2ludGVydGh1cjEMMAoGA1UEChMDWkhXMRMwEQYDVQQLEwpEQSBT
bmEwMC8zMQ8wDQYDVQQDEwZSb290Q0EwHhcNMDAxMDExMDgxMDI2WhcNMDExMDEx
MDgxMDI2WjBIMQswCQYDVQQGEwJDSDEMMAoGA1UEChMDWkhXMRMwEQYDVQQLEwpE
QSBTbmEwMC8zMRYwFAYDVQQDEw1taW5pLnpod2luLmNoMIGfMA0GCSqGSIb3DQEB
AQUAA4GNADCBiQKBgQDBStDmLqjWdRzyuPqwmDWrsCCS/f0/B1V8wjstU5hpsoA8
xRnrDtmgqs4fIzNsc6ofgfvgUpgRlTxfVl6hTxQPMVIjQ+cfAo7H5b0cFCelTw8h
Nmlpr1VjgWYCfqqkCgC2ijjUY86yTA2xmmiXABF3EVQor5HxIxuMkT/jmdAkEwID
AQABo4IBDTCCAQkwCQYDVR0TBAIwADARBglghkgBhvhCAQEEBAMCBkAwLAYJYIZI
AYb4QgENBB8WHU9wZW5TU0wgR2VuZXJhdGVkIENlcnRpZmljYXRlMB0GA1UdDgQW
BBS/QHNHit09W9VflrRe9SMOIl/iCDB+BgNVHSMEdzB1gBRj3b/GknhrCLtpomOE
n+cIHp2kM6FapFgwVjELMAkGA1UEBhMCQ0gxEzARBgNVBAcTCldpbnRlcnRodXIx
DDAKBgNVBAoTA1pIVzETMBEGA1UECxMKREEgU25hMDAvMzEPMA0GA1UEAxMGUm9v
dENBggEAMBwGCWCGSAGG+EIBDAQPFg1taW5pLnpod2luLmNoMA0GCSqGSIb3DQEB
BAUAA4IBAQCBEkXOS8hmpbjqssqH8jADOOhFrBTfnmG4kGh7a+1KdingD+IZwL6j
gr/qR9IZNpi4rg7lCK76KQIf4RCO2wLMaz7WObqpx5o+/gIKjIEpQFhcWDA9ntVX
SpD8JYoUo4182y+4HT1a9TdX/JlVf8Jkea1OKVnZ/CkqWC5IJS8eA2xSu73PLZUS
pAX+1NMhW20NBVgSZUVRATvT2vz+VotEJbxg0U8FHvqAJC6QIDOoRAh/P+5vo6D2
7jCQ4RA/K5OeMwtlvo6CkbpIWkQl61NgI8S/8W2kwKHejKhSm2SBq3ertajfmuWP
Tvv9GT8q3MTLGBKL2eKofVy0UIF9fXNl
-----END CERTIFICATE-----
```

E.3 Configuration File openssl.cnf

```
#
# OpenSSL example configuration file.
# This is mostly being used for generation of certificate requests.
#

RANDFILE               = $ENV::HOME/.rnd
oid_file               = $ENV::HOME/.oid
oid_section            = new_oids

# To use this configuration file with the "-extfile" option of the
# "openssl x509" utility, name here the section containing the
# X.509v3 extensions to use:
# extensions           =
# (Alternatively, use a configuration file that has only
# X.509v3 extensions in its main [= default] section.)

[ new_oids ]

# We can add new OIDs in here for use by 'ca' and 'req'.
# Add a simple OID like this:
# testoid1=1.2.3.4
# Or use config file substitution like this:
# testoid2=${testoid1}.5.6

####################################################################
[ ca ]
default_ca             = CA_default                 # The default ca section

####################################################################
[ CA_default ]

dir                    = /usr/ssl                   # Where everything is kept
certs                  = $dir/certs                 # Where the issued certs are kept
crl_dir                = $dir/crl                   # Where the issued crl are kept
database               = $dir/index.txt             # database index file.
new_certs_dir          = $dir/newcerts              # default place for new certs.

certificate            = $dir/private/cacert.pem    # The CA certificate
serial                 = $dir/serial                # The current serial number
crl                    = $dir/crl.pem               # The current CRL
private_key            = $dir/private/cakey.pem     # The private key
RANDFILE               = $dir/private/.rand         # private random number file

x509_extensions        = usr_cert                   # The extentions to add to the cert

# Extensions to add to a CRL. Note: Netscape communicator chokes on V2 CRLs
# so this is commented out by default to leave a V1 CRL.
# crl_extensions       = crl_ext

default_days           = 365                        # how long to certify for
default_crl_days       = 30                         # how long before next CRL
default_md             = md5                        # which md to use.
preserve               = no                         # keep passed DN ordering

# A few difference way of specifying how similar the request should look
```

```
# For type CA, the listed attributes must be the same, and the optional
# and supplied fields are just that :-)
policy         = policy_match

# For the CA policy
[ policy_match ]
countryName                   = match
stateOrProvinceName           = optional
organizationName              = match
organizationalUnitName        = optional
commonName                    = supplied
emailAddress                  = optional

# For the 'anything' policy
# At this point in time, you must list all acceptable 'object'
# types.
[ policy_anything ]
countryName                   = optional
stateOrProvinceName           = optional
localityName                  = optional
organizationName              = optional
organizationalUnitName        = optional
commonName                    = supplied
emailAddress                  = optional

####################################################################
[ req ]
default_bits                  = 1024
default_keyfile               = privkey.pem
distinguished_name            = req_distinguished_name
attributes                    = req_attributes
x509_extensions               = v3_ca # The extentions to add to the self signed cert

[ req_distinguished_name ]
countryName                   = Country Name (2 letter code)
countryName_default           = AU
countryName_min               = 2
countryName_max               = 2

stateOrProvinceName           = State or Province Name (full name)
stateOrProvinceName_default   = Some-State

localityName                  = Locality Name (eg, city)

0.organizationName            = Organization Name (eg, company)
0.organizationName_default    = Internet Widgits Pty Ltd

# we can do this but it is not needed normally :-)
#1.organizationName           = Second Organization Name (eg, company)
#1.organizationName_default   = World Wide Web Pty Ltd

organizationalUnitName        = Organizational Unit Name (eg, section)
#organizationalUnitName_default=

commonName                    = Common Name (eg, YOUR name)
commonName_max                = 64

emailAddress                  = Email Address
```

```
emailAddress_max               = 40

# SET-ex3                      = SET extension number 3

[ req_attributes ]
challengePassword              = A challenge password
challengePassword_min          = 4
challengePassword_max          = 20

unstructuredName               = An optional company name

[ usr_cert ]

# These extensions are added when 'ca' signs a request.

# This goes against PKIX guidelines but some CAs do it and some software
# requires this to avoid interpreting an end user certificate as a CA.

basicConstraints=CA:FALSE

# Here are some examples of the usage of nsCertType. If it is omitted
# the certificate can be used for anything *except* object signing.

# This is OK for an SSL server.
# nsCertType = server

# For an object signing certificate this would be used.
# nsCertType = objsign

# For normal client use this is typical
# nsCertType = client, email

# and for everything including object signing:
nsCertType = client, email, objsign

# This is typical in keyUsage for a client certificate.
# keyUsage = nonRepudiation, digitalSignature, keyEncipherment

# This will be displayed in Netscape's comment listbox.
nsComment = "OpenSSL Generated Certificate"

# PKIX recommendations harmless if included in all certificates.
subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid,issuer:always

# This stuff is for subjectAltName and issuerAltname.
# Import the email address.
# subjectAltName=email:copy

# Copy subject details
# issuerAltName=issuer:copy

#nsCaRevocationUrl             = http://www.domain.dom/ca-crl.pem
#nsBaseUrl
#nsRevocationUrl
#nsRenewalUrl
#nsCaPolicyUrl
#nsSslServerName = mini.zhwin.ch
```

```
[ v3_ca]

# Extensions for a typical CA


# PKIX recommendation.

subjectKeyIdentifier=hash

authorityKeyIdentifier=keyid:always,issuer:always

# This is what PKIX recommends but some broken software chokes on critical
# extensions.
#basicConstraints = critical,CA:true
# So we do this instead.
basicConstraints = CA:true

# Key usage: this is typical for a CA certificate. However since it will
# prevent it being used as an test self-signed certificate it is best
# left out by default.
# keyUsage = cRLSign, keyCertSign

# Some might want this also
# nsCertType = sslCA, emailCA, objCA

# Include email address in subject alt name: another PKIX recommendation
# subjectAltName=email:copy
# Copy issuer details
# issuerAltName=issuer:copy

# RAW DER hex encoding of an extension: beware experts only!
# 1.2.3.5=RAW:02:03
# You can even override a supported extension:
# basicConstraints= critical, RAW:30:03:01:01:FF

[ crl_ext ]

# CRL extensions.
# Only issuerAltName and authorityKeyIdentifier make any sense in a CRL.

# issuerAltName=issuer:copy
authorityKeyIdentifier=keyid:always,issuer:always
```