Smart Card Applications and Mobility in a World of Short Distance Communication

CASTING Project

a cooperation between
ETH Zürich, Distributed Systems Group
and
Swisscom AG Bern, Corporate Technology

Michael Rohs, Harald Vogt
{*rohs, vogt*}*@inf.ethz.ch*

January 2001

Abstract

The CASTING project is concerned with the application of smart card technology in combination with short distance wireless communication. This report focuses on secure access to Web pages, meaning that the right person has access to his or her personal Web pages in a manner that respects integrity, authenticity and confidentiality. This requires authentication of users, which is achieved by providing users with public-key certificates and the corresponding private keys, stored in a mobile device. The mobile device contains a smart card that stores the user's certificate and the user's private key and executes the necessary operations to convince the remote Web server about the identity of the local user.

The client application that is adapted for user authentication is Netscape, version 4.7x. The Web server used is Apache, extended with the Apache-SSL module. The long distance link between Web server and Netscape is secured with the SSL/TLS protocol. The Web server is configured to ask for user authentication, which is provided by the user's mobile security device. This device is accessible wirelessly via a short distance link that is in turn secured via the SECTUS protocol. Netscape is adapted with a custom security module that implements the PKCS #11 interface.

The CASTING project is part of a cooperation between the Distributed Systems Group at ETH Zurich and Swisscom AG Bern, Corporate Technology. This report describes phase 2 of the CASTING project. Phase 1 was done by researchers at EPFL and focused on protocols for secure spontaneous connections. The work described in this report was done at ETH Zurich and funded by Swisscom. Contact persons are Prof. Friedemann Mattern at ETH Zurich, and Karin Busch and Michael Deichmann at Swisscom.

Contents

Chapter 1

Introduction

1.1 Secure Web Access via Netscape, PKCS #11, and Mobile Phones

The basic usage scenario that we envision within the CASTING project [46, 52, 55, 56] is secure access to personal Web pages from fixed PCs via the Netscape Internet browser. We assume that there is a pool of fixed PCs that users can choose from. The PCs do not store any identification information about a single user, so that they can possibly be employed by any user. The PCs are connected to the Internet. Further we suppose that users can access personal Web pages and Web services that are located on some HTTP server on the Internet.

This server is reponsible for maintaining privacy of the personal data that it contains. It shall only provide this data if it is convinced that the requesting party is authorized to access it. Authorization requires unforgeable identification of the requesting party. Traditionally this has been done using login names and passwords, but password schemes have some well-known drawbacks. They are, for example, often poorly chosen and therefore easy to guess. A better approach for identifying and authenticating users is to use a public key scheme, i.e. providing users with a public-key certificate and a private key. Once the identity of a requesting party is proven, standard cryptographic protocols, like Transport Layer Security (TLS) can be used to transfer the data securely from the HTTP server to the fixed terminal of the user.

As the fixed PCs do not contain user-specific data, it is assumed that user certificate and public-key are stored on a mobile device, like a mobile phone, that the user carries around. More precisely, we assume that user certificate and public key are stored in the *Subscriber Identification Module (SIM)* within the mobile phone. Upon requesting personal data on the fixed terminal, the user certificate that is needed to identify the user shall be transferred to the fixed PC via a short distance wireless link and then be sent to the remote HTTP server. The private key never leaves the SIM card and is used during the authentication procedure as described below. In effect, the mobile phone is used as a wireless smartcard reader for the

fixed terminal.

An essential aspect in this setup is the short distance wireless link between the mobile phone and the terminal. The setup of this connection must happen spontaneously, i.e. it must not require any configuration by the user. Potential wireless link technologies are IrDA or Bluetooth. An issue is that the wireless link is not physically secure. IrDA traffic for example can easily be eavesdropped. More critical, however, is that the association between the user's mobile phone and the terminal is not given in advance. It must be made sure that the right mobile is connected with the right terminal. Otherwise it might be possible for an attacker to employ the user's phone to get access to the user's personal data.

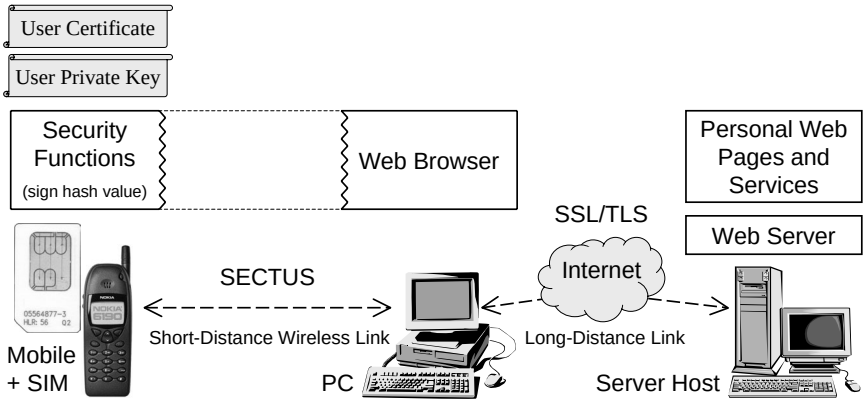

Figure 1.1: CASTING Overview

The overall setting is depicted in figure 1.1. There exists a short-distance wireless link between the mobile phone and the PC in front of the user. A second link is established between the PC and the Web server that holds content and services personalized to the user. The short-distance link is secured with the SECTUS protocol; the long-distance Internet connection with SSL/TLS. Critical security functions that normally reside inside the Web browser are now performed by the SIM inside the mobile phone. The SIM contains the necessary user data, such as the user's certificate and private key. This is an idealized situation, though, because performance constraints prohibit the execution of all security-related functions on the SIM itself. The actual split of security functionality between SIM and PC, as used in the implementation, is detailed in the following chapters.

We first describe how the long-distance link is secured. In chapter 3 we then explain the specifics of securing the spontaneously established short-distance link and the pairing of the user's mobile phone and PC. Chapter 4 gives a review of public-key certificates and secure Web servers. The adaptation of Netscape for an external security module is described in chapter 5. Our implementation is outlined in chapter 6.

Chapter 2

Securing the Long-Distance Link

2.1 Transport Layer Security

The user's personal data needs to be sent across the Internet from the HTTP server to the terminal of the user. To maintain confidentiality, authenticity, and integrity of this data we chose to use the well-established SSL/TLS protocol.

In the following the SSL/TLS protocol is shortly described. Especially the way in which the client, i.e. the fixed terminal, proves its identity is investigated. The points in the protocol at which the user must provide personal data (e.g. the client certificate) or must perform cryptographic operations are identified.

2.2 SSL/TLS Overview

The goal of the SSL/TLS protocol is to provide communications privacy over the Internet. It allows two parties to communicate in a way that is *designed to prevent eavesdropping, tampering, or message forgery*. The SSL (Secure Sockets Layer) protocol [11] was originally developed by Netscape Communications and is superseded by the TLS (Transport Layer Security) protocol. TLS [2] is based on SSL and is being standardized by the Internet Ingineering Task Force (IETF). In the following we will discuss TLS. TLS version 1.0 is nearly identical to SSL version 3.0. For an introduction to SSL/TLS see [51].

TLS is located between the transport layer (TCP) and the application layer (e.g. HTTP). TLS requires a reliable transport protocol, which is TCP in our case. The application data it carries will be HTTP requests and responses in our application scenario. See [43] for a description of HTTP over TLS.

The TLS protocol is based on the client-server principle. Typically servers hold resources that clients want to access in a secure manner. Therefore TLS allows mutual authentication between a server and a client. It also allows to establish an encrypted and authentic channel between both machines.

- **TLS server authentication** lets a user confirm the identity of the server.

The server needs to be equipped with a public-key certificate that contains identification information of the server together with the public key of the server, signed by a certificate authority (CA) that must be trusted by the client software. See [38, 39, 40, 41, 42, 53] for more information on certificate authorities and public-key infrastructures.

- **TLS client authentication** conversely allows a server to confirm the identity of a user that tries to access resources. The same technique as in server authentication is used: The client needs to posses a certificate that contains identification information and a public key. This certificate needs to be signed by a CA known to and trusted by the server. Client authentication is a very important point in our scenario, because the server's resources must only be given to authorized clients.

- **A secure channel** between client and server is established before the first byte of application data is exchanged between both parties. Data that is exchanged on such a channel is encrypted and therefore can only be deciphered by client and server. It is also immune against tampering, i.e. unauthorized modification of the data exchanged. A secure channel is established in a so-called *handshake phase* that uses public-key cryptography to establish a shared secret between both parties. Application data is then encrypted with symmetric cryptography using the generated shared secret as the encryption and decryption key. The reason for using a combination of asymmetric (public key) and symmetric (private key) cryptography lies within their complementing properties. Public key cryptography can be used to distribute shared secrets, while asymmtric cryptography requires less computing resources and is therefore much faster.

TLS is divided in two layers. At the lower layer the *TLS record protocol* is responsible for encrypting, fragmenting, and optionally compressing higher layer data into records of at most 2^{14} bytes. These records can carry different types of content. The most important ones are (encrypted) application data and handshake messages.

The handshake messages belong to the *TLS handshake protocol*, which is part of the upper layer of the TLS protocol. The handshake protocol uses the record protol to exchange a set of handshake messages in order to first negotiate the security parameters of a TLS connection.

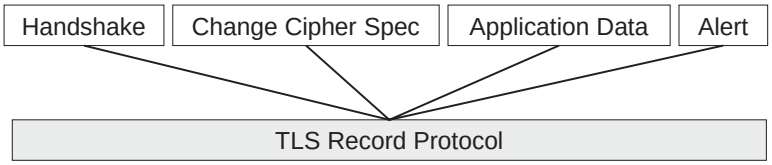

Figure 2.1: TLS protocol layers

Figure 2.1 shows the various sub-protocols of the TLS protocol. TLS records of the lower layer carry one or more messages of the upper layer as their payload. Upper layer messages may even be fragmented into multiple records.

2.3 TLS Record Protocol

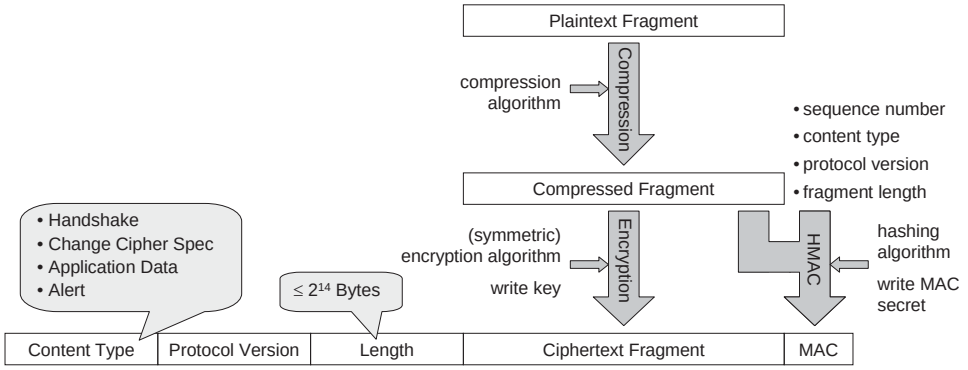

Figure 2.2: Assembling a TLS record

Figure 2.2 gives an overview of how the TLS record layer works. The plaintext fragment is, potentially a part of, a higher layer message. It can also contain multiple higher layer messages if they are all of the same type. The type is given in the content type field of the record. After optional compression[1] the fragment is encrypted. To ensure that the record is not altered in transit, it is protected with a message authentication code (MAC) that is computed over all fields (i.e. content type, protocol version, length, and compressed fragment) of the record. The MAC also contains a sequence number to prevent replay attacks.

2.3.1 Algorithms

TLS can be used with a wide variety of cryptographic algorithms for compression, encryption, and message authentication. A particular choice of algorithms is called a *cipher suite*. The cipher suites of interest in our scenario are those that use RSA[22] as the algorithm for key exchange, because we assume that the user is equipped with an RSA public-key certificate and a corresponding private key.

Table 2.1 shows cipher suites that use RSA for key distribution. Initially no encryption and no message authentication algorithms are negotiated. Therefore the initial cipher suite is always TLS_NULL_WITH_NULL_NULL.

[1] This feature is seldom used in practice.

CipherSuite	Key Exchange	Bulk Cipher	Hash
TLS_NULL_WITH_NULL_NULL	none	none	none
TLS_RSA_WITH_RC4_128_MD5	RSA	RC4_128	MD5
TLS_RSA_WITH_RC4_128_SHA	RSA	RC4_128	SHA
TLS_RSA_WITH_IDEA_CBC_SHA	RSA	IDEA_CBC	SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA	RSA	3DES_EDE_CBC	SHA
TLS_RSA_WITH_DES_CBC_SHA	RSA	DES_CBC	SHA

Table 2.1: TLS cipher suites that use RSA for key distribution

2.3.2 Connection States

The TLS record protocol always operates under a specific *connection state*. The current connection state specifies how to encrypt, compress, and hash the payload data that is transported in record protocol blocks, however it does not contain the key distribution and authentication methods. This information belongs to the handshake protocol as it is only needed during the setup phase of a connection. The connection state contains the choice of cipher suite and its parameters. It can be subdivided into four logical connection sub-states.

Connection Substate	Description
Current write state	Currently effective parameters for writing data
Current read state	Currently effective parameters for reading data
Pending write state	Parameters for writing, not effective yet, currently under negotiation
Pending read state	Parameters for reading, not effective yet, currently under negotiation

Table 2.2: Current and pending connection states

Table 2.2 shows that there are different connection states for the two directions of a connection. There is also a distinction between the current connection state – the one which is currently effective – and the pending connection state – the one that is not yet in effect. The parameters of the first cannot be modified anymore, while the parameters of the second are negotiated between client and server by means of the handshake protocol as described below. After the negotiation of parameters for the pending connection state is complete, it can be turned into the current connection state by a switching operation. This switching operation is the point in time when the new parameters come into effect. The switching operation is performed by sending a *change cipher spec* message, which is the only message of the *change cipher spec protocol*.

Each of the four states contains the items shown in figure 2.3.

```
enum { server, client } ConnectionEnd;
enum { null, rc4, rc2, des, 3des, des40, idea } BulkCipherAlgo-
rithm;
enum { stream, block } CipherType;
enum { true, false } IsExportable;
enum { null, md5, sha } MACAlgorithm;
enum { null(0), (255) } CompressionMethod;

struct {
    ConnectionEnd          entity;
    BulkCipherAlgorithm    bulk_cipher_algorithm;
    CipherType             cipher_type;
    uint8                  key_size;
    uint8                  key_material_length;
    IsExportable           is_exportable;
    MACAlgorithm           mac_algorithm;
    uint8                  hash_size;
    CompressionMethod      compression_algorithm;
    opaque                 master_secret[48];
    opaque                 client_random[32];
    opaque                 server_random[32];
} SecurityParameters;
```

Figure 2.3: Security parameters of a connection state (for a syntax description, see [2])

2.3.3 Keys and Key Generation

The symmetric encryption algorithms that are defined for use with TLS are RC4 [26], RC2, DES [33], 3DES, DES40, and IDEA [28]. Depending on the needs of the application and on export restrictions different key sizes can be chosen. For our application we consider a key size of 128 bits as desirable.

The message authentication code of record layer messages is generated using HMAC [27], which is a keyed cryptographic hash function. In TLS the HMAC algorithm can be parameterized with either MD5 [44] or SHA [34]. It takes two inputs: a key – called a *MAC secret* – and data.

There are different keys for the two directions of a connection as well as for encryption and message authentication. Therefore, for each connection state the following keys are defined.

Key / Secret	**Purpose**
MAC write secret	Generating a MAC for a record
MAC read secret	Verifying a MAC of a received record
write key	Encryption of data before sending
read key	Decryption of data after reception

Using these keys content is encrypted, decrypted, resp., as follows:

$Enc_{bulk\_cipher\_algorithm}$(write encryption key, plaintext)
$Dec_{bulk\_cipher\_algorithm}$(read encryption key, ciphertext)

Message authentication codes are generated, verified, resp., like this[2]:

$HMAC_{mac\_algorithm}$(MAC write secret,
sequence number + content type +
protocol version + length + plaintext fragment)
$HMAC_{mac\_algorithm}$(MAC read secret,
sequence number + content type +
protocol version + length + plaintext fragment)

Because we use symmetric encryption algorithms, the key that is the write key on one side of a connection is equal to the read key on the other side. This key has to be exchanged between TLS client and TLS server in a secure way. To achieve this the key is based on a shared secret – called the *premaster secret* – that is securely exchanged between client and server during the handshake phase. The key exchange will be described in the next section. Here we focus on how the keys for encryption and MAC generation are produced from the premaster secret.

The premaster secret is a sequence of 48 random[3] bytes, known only to the client and the server. The premaster secret and two other (non secret) random values chosen independently by client and server during the handshake protocol are used to compute the so called *master secret*.

```
master_secret := PRF(pre_master_secret, "master secret",
                     SecurityParameters.client_random +
                     SecurityParameters.server_random)[0..47];
```

The master secret is in turn used to compute a *key block* that is long enough to provide all the keys needed. If encryption keys with a length of 128 bits are used and if no initialization vectors are required, then the generated key block needs to be 64 bytes in length: length(key block) = length(MAC write secret) + length(MAC read secret) + length(write key) + length(read key) = 16 + 16 + 16 + 16 = 64 bytes.

```
key_block := PRF(SecurityParameters.master_secret,
                 "key expansion",
                 SecurityParameters.server_random +
                 SecurityParameters.client_random);
```

The key block is partitioned as shown in figure 2.4 to obtain the needed keys. Some encryption algorithms also require initialization vectors (IVs), which are also obtained from the key block.

The function that computes the master key as well as the key block is called *pseudo random function (PRF)*. It is defined like follows.

[2]The + sign denotes the concatenation operation.

[3]Actually only 46 bytes are supposed to be randomly chosen. The first two bytes contain the protocol version number.

```
client MAC write secret = key_block[0        ..  m       -1]
server MAC read secret  = key_block[ m       .. 2m       -1]
client write key        = key_block[2m       .. 2m+ n    -1]
server write key        = key_block[2m+ n    .. 2m+2n    -1]
client write IV         = key_block[2m+2n    .. 2m+2n+i  -1]
server write IV         = key_block[2m+2n+i  .. 2m+2n+2i-1]
```

where m is the length of the MAC key, n is the length of the encryption key and i is the length of the initialization vector.

Figure 2.4: Partitioning of the key block.

```
PRF(secret, label, seed) :=  P_MD5(S1, label +[4] seed) XOR
                             P_SHA-1(S2, label + seed)
```

where `S1` is the first half and `S2` is the second half of `secret` and where `P_hash` is defined as

```
P_hash(secret, seed) :=
    HMAC_hash(secret, A(1) + seed) +
    HMAC_hash(secret, A(2) + seed) +
    HMAC_hash(secret, A(3) + seed) + ...
```

Function A is defined as

```
A(0) = seed
A(i) = HMAC_hash(secret, A(i-1))

HMAC_hash(key, data) := hash((key XOR opad) +
                        hash((key XOR ipad) + data))
```

2.4 TLS Handshake Protocol

The handshake protocol is responsible for the agreement on a common protocol version, a common cipher suite, for generating a shared secret and also for mutual authentication between client and server. All the security parameters that the record protocol needs are provided to it by the handshake protocol.

The handshake protocol can either work with or without client authentication. For our usage scenario client authentication is a major concern. Therefore we only consider this case. Figure 2.5 shows the message flow of the handshake protocol.

It is assumed that the server as well as the client are equipped with public-key certificates and corresponding private keys to authenticate each other during the TLS handshake protocol.

The first message is called the *client hello* message. It is sent from the client to the server and its most important contents are a 32-byte random value generated by the client and a list of cipher suites. This list contains the combinations of

[4] Again, here the + sign denotes the concatenation operation.

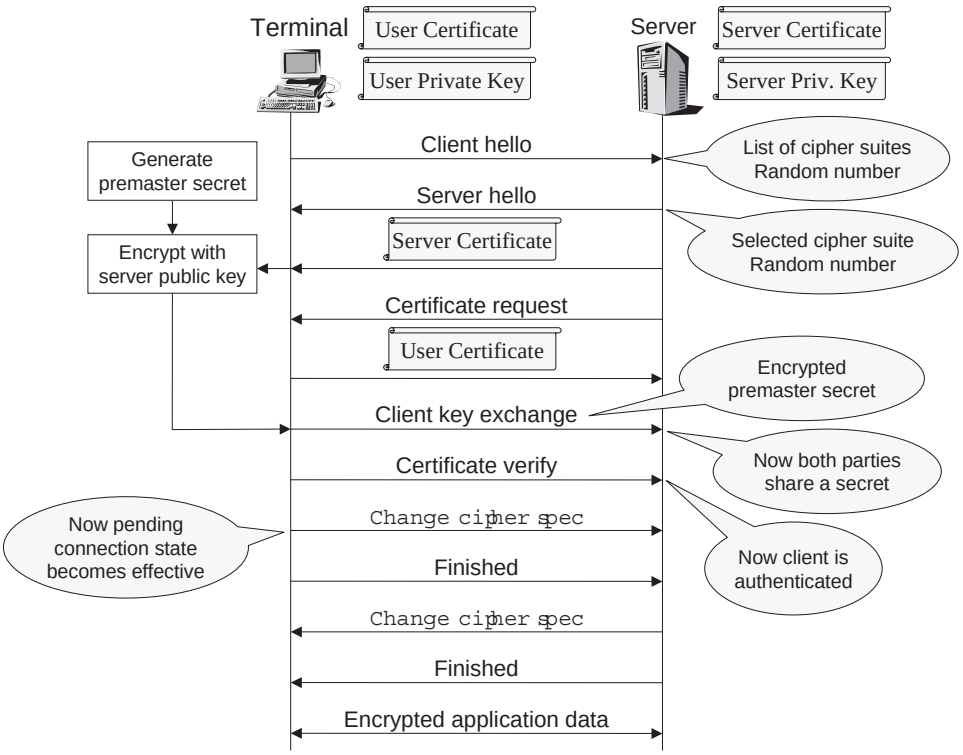

Figure 2.5: Message flow of the TLS handshake protocol

cryptographic algorithms that the client supports in the order of the client's preference. As described in section 2.3.1 a cipher suite defines a particular choice of key exchange algorithm, bulk cipher algorithm and message authentication algorithm.

The server responds with a *server hello* message. It contains a server generated 32-byte random value and one of the cipher suites selected from the list provided by the client. If the server does not find a suitable cipher suite, communication will be cancelled.

Next the server sends a *server certificate* message that contains the server's public-key certificate. The structure of public-key certificates is described in section 4.1.

The *certificate request* message sent by the server is optional for TLS, but essential for our application. It tells the client that it needs to provide a certificate and that it has to authenticate itself. The structure of this message is shown in figure 2.6. It contains a list of the types of certificates that the client is allowed to provide and a list of certificate authorities that the server knows of. The second list might be empty, which indicates that no specific certificate authority is required. We assume that our client is equipped with an RSA certificate. Therefore the client can only successfully complete the handshake protocol if the *client certificate type list* contains the type *rsa_sign*.

```
enum {
    rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4), (255)
} ClientCertificateType;

opaque DistinguishedName<1..2^16-1>;

struct {
    ClientCertificateType certificate_types<1..2^8-1>;
    DistinguishedName certificate_authorities<3..2^16-1>;
} CertificateRequest;
```

Figure 2.6: Certificate request message

The server ends its turn with a *server hello complete* message (not shown in figure 2.5).

The client now sends its certificate as requested by the server. The next message is called *client key exchange*. Since we only deal with RSA key exchange this message looks as follows:

```
struct {
    ProtocolVersion client_version;
    opaque random[46];
} PreMasterSecret;

struct {
    public-key-encrypted PreMasterSecret pre_master_secret;
} EncryptedPreMasterSecret;

struct {
    EncryptedPreMasterSecret exchange_keys;
} ClientKeyExchange;
```

The client generates a random value – the *premaster secret* – and encrypts it with the public key of the server that is contained in the server certificate. After sending this message client and server have a shared secret. This shared secret is used as described above to generate the keys for bulk cipher encryption and message authentication.

Yet one last and most important issue remains: that of client authentication. The server can be considered as authenticated to the client, because it can decrypt the client key exchange message and therefore obtain the shared secret only if it knows the private key corresponding to the server certificate it sent before. But the client still has to prove its identity to the server. This is achieved by the last message of the handshake protocol, called *certificate verify*.

In order to prove its identity the client has to sign a hash value that is generated over all handshake messages exchanged so far. All handshake message up to, but

```
TLS:
md5_hash = MD5(handshake_messages);
sha_hash = SHA(handshake_messages);

SSL:
md5_hash = MD5(master_secret +[5] pad_2 +
           MD5(handshake_messages + master_secret + pad_1))
sha_hash = SHA(master_secret + pad_2 +
           SHA(handshake_messages + master_secret + pad_1))

struct {
    opaque md5_hash[16];
    opaque sha_hash[20];
} hash;

struct {
    digitally-signed hash[36];
} Signature;

struct {
    Signature signature;
} CertificateVerify;
```

Figure 2.7: Certificate verify message

not including, the certificate verify message are concatenated in the chronological order sent, resp. received, to obtain the `handshake_messages` string as shown in figure 2.7. Then the MD5 and the SHA hash values are computed from this string and concatenateed to obtain the `hash` value. Note that SSL uses a slightly different scheme to obtain the hash value.

Now the private key of the user comes into play. It is needed to compute the signature from the hash value that is to be sent to the server. The signature can only be computed by an entity that knows the private key corresponding to the certificate. Therefore the server is convinced that the client is really who he claims to be and authentication is complete.

Fortunately this is the only place in the TLS protocol where the private key of the user is required. The hash value to be signed is very short (36 bytes) and the resulting signature is, depending on the size of the private key, also relatively short.

All parameters are now set. Until now, the *current* connection state remained untouched, because handshake messages only modify the *pending* connection state. The point where the completely negotiated pending connection becomes the new current connection is achieved with the next message, which is the one an only message of the *change cipher spec* protocol. It is a fixed message that simply signals the transition between connection states. The new parameters are communicated to the record protocol and it will process subsequent traffic accordingly.

To verify that the transition of connection states was successful the client sends one more handshake message: the *finished* message. It is the first message that is sent under the new connection state. It is kind of a “test packet” to check if encryption works properly.

```
verify_data = PRF(master_secret, "client finished"[6],
                  MD5(handshake_messages) +
                  SHA-1(handshake_messages))[0..11];

struct {
    opaque verify_data[12];
} Finished;
```

The finished message is decrypted and verified by the server and, if successful, application data can be sent over the connection. For the other direction – server to client – connection states are switched in the same way. The server also sends a *change cipher spec* message to make the negotiated pending connection state effective and a *finished* message to let the client verify that the server is able to do bulk cipher encryption according to the cipher suite and keys negotiated. Now, finally, application data can be securely exchanged.

2.4.1 Adapting TLS Client Authentication

As described in the introduction we would like to use terminals (client PCs) that are not configured in advance to hold user-specific data. In particular we would like not to entrust our private key to these PCs. The solution we envision is to split the cryptographic functionality of certificate (public-key) and private key and to leave the private key in the SIM card of the mobile. While the certificate is public information and can therefore be transferred to the terminal, the private key is very sensitive and can only adequately be protected by the SIM [8, 9, 10].

This means that all operations in which the private key is involved need to be executed by the SIM itself. Fortunately it is easy to split TLS functionality in a way to let the SIM do the private key operations and to let the terminal do the rest. As pointed out before, the only operation in which the private key is taking part is signing the 36-byte hash value. And this operation also takes place only once at the handshake phase of a TLS session. Therefore it is possible to use the combination of mobile phone and SIM as an authentication device, despite of its low bandwidth and computing capacity.

The next chapter will describe in detail how the hash value is securely transferred to the mobile and how the signature operation is performed.

[6] For the server it is ”server finished.”

Chapter 3

Securing the Short-Distance Link

3.1 Splitting Cryptographic Functionality

For the reasons described in the previous chapter we divided the cryptographic functionality that is needed for our usage scenario between the fixed PC and the mobile device that the user carries. These devices have different properties with respect to trustworthiness, personalization, computing power, and communication bandwidth. Table 3.1 summarizes these properties. It shows that both devices are rather complementary to each other with respect to the properties that are relevant within our context. The goal is to split the functionality in such a way such that the strengths of both devices are utilized.

Property	**Fixed PC**	**Mobile device / SIM**
trustworthiness	low—middle	high
personalization	no (a priori)	yes
computing power	high	low
communication bandwidth	high	low

Table 3.1: Properties of fixed PC versus mobile device/SIM

Taking the results of chapter 2 and the properties just described into account, the following “implementation axioms” seem to make sense.

- The private key is only known to the SIM.

  The private key is a very critical piece of information that should never leave the SIM inside the mobile phone. Consequently all operations that involve the private key have to be done by the card. In our case the only operation that needs the private key is signing the hash code of the SSL/TLS client verification message. This occurs only once in a SSL/TLS session, i.e. during the handshake phase. Because the data that is signed and the signature that

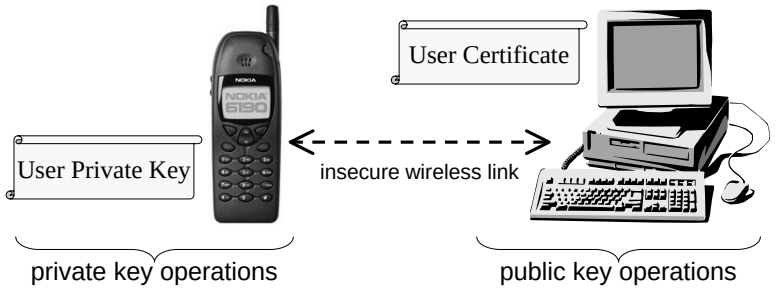

Figure 3.1: Split of functionality and wireless link

is produced are short, the low communication bandwidth between fixed PC and SIM is not an issue.

- The terminal should do as much work as possible, provided that security is not affected.

  The scarce computing resources of the SIM prohibit to do all cryptographic operations on the card itself. Instead we let the card perform only the critical operations (signing) and do everything else on the PC. Especially bulk cipher encryption is done on the PC.

  To make the identity of the user known to the Web server that provides the resources desired, the user certificate is transferred via the wireless link to the PC and from there to the Web server. Since the user certificate is public information that is protected against tampering, it can be sent without any protection. Transferring the certificate to the PC is needed, because the PC is not personalized to a particular user and does not store any certificates or other personal information.

- Depending on the needs of the SIM, a minimal number of cryptographic algorithms is used.

  We intend to only provide user certificates of a certain type, i.e. certificates that adhere to the X.509 standard [1, 12, 32, 50] and that contain RSA [45, 47] public keys. Therefore the method of choice for authenticating users is by generating signatures with their RSA private key. SSL/TLS is designed to use other algorithms for user authentication apart from RSA.

3.2 Ad-hoc Wireless Connection

In our scenario it is important that a user who is equipped with a mobile phone is able to connect to a fixed PC without much effort. The connection should happen spontaneously and without administration on the user's side. Wireless connections are suitable to meet these requirements. The two kinds of short distance wireless

connections that mobile phones offer are infrared [21] and – in the near future – Bluetooth [4].

It is assumed that the following steps happen when a user approaches a PC and switches on short-distance wireless connectivity on the mobile phone. First the physical connection – be it infrared or Bluetooth – is established. Then the user certificate is transferred to the fixed terminal to "personalize" the PC for that user. If private key operations need to be performed, the data to sign is transferred to the mobile phone and executed by the SIM card. Unfortunately, the short distance link, at least when using IrDA, is physically insecure. Therefore the short distance link also needs to be protected.

3.2.1 Possible Attacks on the Wireless Link

Common attacks on communication links are eavesdropping, generation of fake messages, replay of old messages, and interception and modification of messages. Eavesdropping is not an issue in our case, because we only transfer data (hash values) to sign and corresponding signatures via the link. The signatures are sent to the client unprotected and everybody who knows the user's public key is able to verify them and therefore to obtain the hash value that was signed.

Instead we need to protect against replay attacks and especially against fake messages and modified messages. We need to ensure that we only sign data that comes from the PC in front of the user. Otherwise an attacker might impersonate us by sending a hash value to our SIM using the signature for authentication in the SSL/TLS protocol. Such an attack is illustrated in figure 3.2.

Figure 3.2: Example for a possible attack: Bob impersonates Alice

3.3 The SECTUS Protocol

The SECTUS [7] protocol is used to ensure that the mobile device only signs data from the right terminal and conversely that the terminal only accepts data from

the right mobile. This protocol makes attacks like the one above impossible, by signing the data that is transferred between mobile phone and fixed terminal. SECTUS is a custom protocol that was developed especially for the purpose of user authentication and device pairing over short distance wireless links.

The goal of the SECTUS protocol is to establish a shared secret between mobile and PC that can be used to sign all data traffic over the wireless link. SECTUS stands for *secret transfer via the user* and this is also the main idea behind the protocol. The protocol assumes that while the wireless link between mobile/SIM and PC is insecure, there still exists a secure channel between both devices. This secure channel is the user herself. Figure 3.3 shows the details.

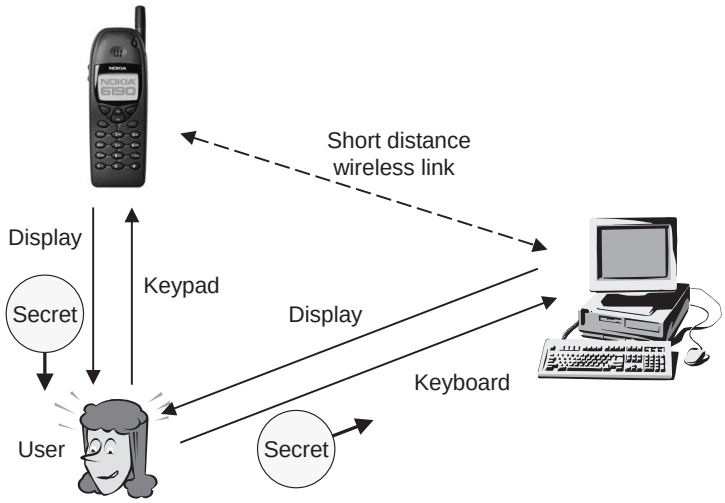

Figure 3.3: SECTUS: transfer of a secret via the user

Display and keyboard of mobile and terminal are modelled as channels to and from the user. The channels from mobile to user and from user to terminal are considered to be secure for the following reasons:

- Channel from mobile to user (display of mobile)

  This channel is considered to be authentic, timely, and, at least for a short amount of time, confidential, because it is assumed that the user can hide the contents of the display from being observed by attackers for a short time. Therefore this channel is denoted a *short time channel*.

- Channel from user to PC (keyboard of terminal)

  This channel is also considered authentic, timely, and confidential for a short period of time, because the assumption is that the user cannot be observed typing the secret into the terminal and that there is no echo displayed on the terminal screen. This is also a short time channel.

- Channel from mobile to SIM

It is also assumed that the link between mobile device and SIM is secure, meaning authentic, timely and confidential for an unlimited amount of time.

The full specification of SECTUS is given in chapter 2 of [7]. The security properties of the channels listed above are sufficient to transfer a secret from the SIM and mobile phone via the user to the terminal. This shared secret can then in principle be used as an authentication key. An issue is the short time confidentiality of the channels and therefore of the secret. For this reason, in our implementation the transferred secret is only used as a starting point to create a longer, and hopefully more secure, authentication key.

3.3.1 Establishing the Authentication Key

An issue when transferring data via the user is that this data can only have a very limited size to be convenient to a human user. A desirable length of a secret would for example be 128 bits. But this results in a printable representation of 22 characters, if capitalization is significant and if digits are used. For usability purposes the printable representation of the key should be as short as possible. On the other hand our application requires a certain minimum key length. In [7] a length of 40 bits is suggested. We would like to maintain this estimate, even if our key is only valid for a short period of time, i.e. less than 30 seconds.

According to [54] random codes that have to be memorized in short term memory should be either letters or digits, but not both. Also capitalization should not be significant, meaning that the code should either consist of lower or of upper case letters, but not of both. Also, the code should not be longer than 4 characters or 6 digits. If longer codes are inevitable, then they should be partitioned into segments of three letters or characters each.

Therefore, we use upper case letters for our user-transferrable secret. Since there are 26 upper case letters, at least nine letters are required to achieve a secret-length of 40 bits: $2^{43} > \mathbf{26^9} > 2^{42} > \mathbf{2^{40}} > 26^8$

For usability these nine upper case letters are partitioned into three triples. This results in more than 42 bits of randomness.

A 128 bit authentication key is derived from the 40 bit user-transferred secret in the following way. First the user has to activate the SIM applet for the CASTING protocol on her mobile phone. Then, after a physical wireless short-distance connection is established between the mobile and the fixed terminal, the mobile displays the one-time secret, called *r*, that is formatted as described above. The user reads the secret from the mobile's display and types it into a dialog box on the terminal's screen. As symbolized with the locks in figure 3.4, this secret transfer operation is considered to be secure. At this point the mobile and the terminal are associated to each other via the user – they share a secret. All other messages are sent via the unsecure wireless link between mobile and terminal.

Message m1 is the user certificate. Upon arrival at the terminal it is optionally verified. This step might be omitted, because the certificate will be verified by the

remote HTTP server anyway.

Next the terminal generates a 128 bit authentication key *s*, that is used for protecting data and that can potentially be used for a long period of time. The authentication key is encrypted with the public key of the user and the result is sent to the mobile (message m2). Only the card will be able to decipher the authentication key, using the private key.

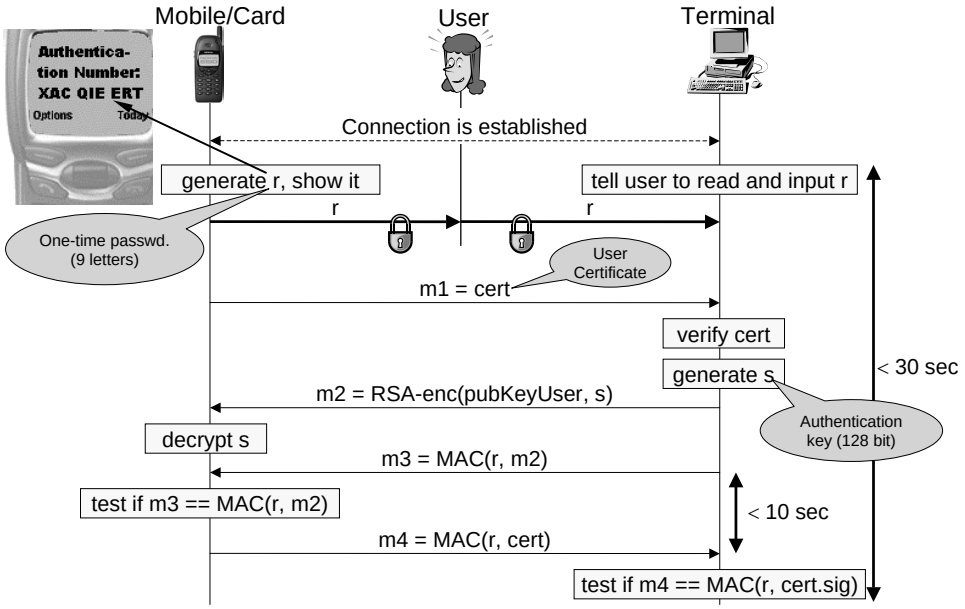

Figure 3.4: Establishing the authentication key **s** for the short distance link

Message m3 is the message authentication code for message m2 with r used as the MAC-key. In the implementation, HMAC [27] with MD5 is used as the message authentication algorithm. The mobile knows r and m2 and computes the message authentication code itself. It compares the result with the received message m3 and aborts, if they are not equal. If, on the other hand, verification is successful, then the mobile assumes that the sender knows r, and that the sender is indeed the terminal in front of the user, since this knowledge was transfered on a secure way via the user. To convince the terminal about the identity of the mobile, the SIM has to create a message authentication code of the user certificate and send it to the terminal (message m4). The terminal will also do a verification and abort if it fails.

In message m3 secret r is used for the first time. An attacking terminal that wants to generate unauthorized signatures from the mobile has just one try to guess r right to generate a correct message m3. This event is extremely unlikely, because the mobile aborts if verification fails – even if the attacker has unlimited computing power.

After m3 is sent however, r could be computed from the knowledge of m2 and

m3. If the message authentication algorithm is secure, this requires very much computing power, because different values for r have to be tried in a brute-force way. Precomputation is not possible, because m2 is not known in advance. To prevent an attacking mobile to succeed with a brute force approach, it is very important to limit the amount of time between sending message m3 and receiving message m4. A value of at most 10 seconds seems to be a reasonable value here. An attacking mobile that succeeded in finding r from m2 and m3 could send a fake certificate to the terminal that is not the user's original certificate. With the time constraint this event should not occur. Additionally a timeout value of 30 seconds is used for the overall authentication key generation algorithm. If any of these timeouts is reached, the algorithm aborts.

The result of the algorithm is the 128 bit authentication key **s** that should be secure for a long period of time. If r is revealed later, that knowledge is worthless to an attacker, because to find s, message m2 needs to be decrypted.

The authentication key is now used to protect messages that are exchanged between terminal and mobile from modification. This is achieved by appending a MAC to each message that is sent across the wireless link. This turns the wireless link into an authentic channel. As already mentioned, the short distance channel only needs to be authentic, but is not required to be confidential, because the SIM only performs signature operations that are later sent across the Internet unencrypted anyway.

3.3.2 Integration of SECTUS with SSL/TLS

As described in section 2.4 the client proves its identity by signing a hash value that is computed from all the handshake messages exchanged between client and server. The computation of the hash value can be done by the terminal, while signing the hash value can only be performed by the SIM within the mobile, because the private key is neccessary for this operation which is only known to the SIM.

After the hash value has been computed by the terminal, it is sent to the mobile via the wireless link. The hash value is secured by appending the MAC that is parameterized with the authentication key s that was established before. The SIM verifies the MAC and continues only if verification succeeds. After the signing operation is performed, the signature of the hash value is sent back to the terminal. From there it is passed on to the SSL/TLS server as the content of a *certificate verify* message. The integration of SECTUS with SSL/TLS is shown in figure 3.5.

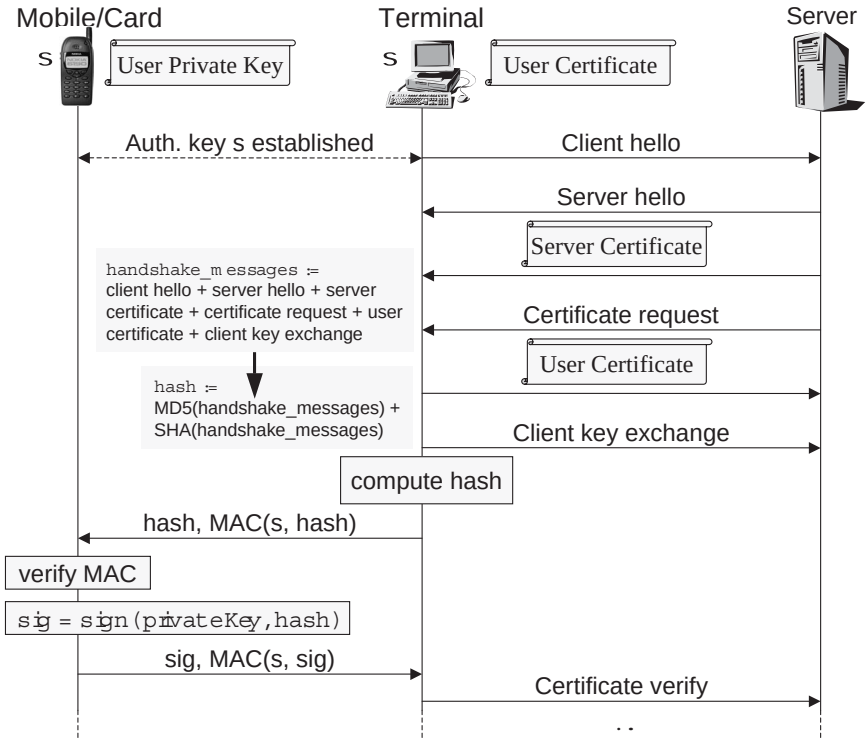

Figure 3.5: Integration of SECTUS with SSL/TLS

Chapter 4

Protecting Personal Web Pages and Services

4.1 Public Key Certificates

The purpose of public key certificates is to bind public key values to identities. Identities unambigously designate persons or systems. The binding is asserted by trusted third parties, so-called *certificate authorities (CAs)*. A CA signs this binding and thereby guarantees that a certain public key belongs to a certain identity. The CA signature also protects the certificate against modification. Therefore certificates can be sent across unsecure channels and stored in unsecured storage. Entities can simply check the validity of a certificate by verifying the CA signature.

Certificates typically have a limited lifetime, which is included in the certificate attributes. Certificate authorites can be organized hierarchically, which means that a certificate of a CA is signed by its parent-CA. The origin of the hierarchy is called a *root CA*. The path from a client certificate up to a root CA is called a *certification path*. Root CAs are unique within organizations that act as trusted third parties, like Swisskey [40], VeriSign [42], Thawte [41], or Entrust [38]. Invalidated certificates are placed on public *certificate revocation lists (CRLs)*.

4.1.1 X.509 Certificates

The format of certificates used in the Internet is specified in RFC 2459 [12], which is one part of a family of standards for the ITU-T X.509 (formerly CCITT X.509) or ISO/IEC/ITU 9594-8 *public key infrastructure (PKI)* for the Internet. The currently used version is X.509 version 3, which has extension fields for additional information, such as additional identification information, revocation list URLs, certificate policy URLs, or key usage constraints. Certificate policies are documents that describe which policies a CA uses to issue client certificates, the security of the CA infrastructure, and other information concerning the organization that issued the certificate. Key usage constrains allow to describe the operations that the public

key that is contained in the certificate might be used for.

The fields of an X.509 certificte are shown in table 4.1.

Field	**Description**
Version	v1, v2 or v3
Serial number	Assigned by the issuer (which is a CA), unique for CA
Signature algorithm	Identifies the algorithm used to sign this certificate
Issuer	Identifies the entity that signed and issued the certificate
Validity period	Starting date and ending date of certificate validity
Subject	Identifies the subject associated with the public key
Subject's public key	Subject's public key and the algorithm with which it is used
Extensions	Standard extensions include additional issuer information, revocation list information, and certificate path information
Issuer's signature	CA's signature of the above fields, asserts their validity

Table 4.1: Attributes of X.509 certificates

Each certificate has a serial number that is unique for the CA that issued it. A certificate is therefore globally uniquely identifiable using the pair (serial number, issuer). The identifier for the algorithm that the issuer uses to sign the certificate is also given as a field. The fields for issuer and subject are formatted as so-called *distinguished names (DAs)*. This is a data type that uniquely identifies or "distinguishes" objects in an X.500 directory. It is defined in X.501. Distinguished names are sets of attribute-value pairs. Basic attributes for distinguised names are: common name (CN), country name (C), organization (O), organizational unit (OU), location (L). More information about attribute usage can be found in [50]. Table 4.2 shows an example X.509 certificate.

X.509 certificates are described using the *abstract syntax notation (ASN.1)* [23], which specifies a notation for defining types and values. Types can be constructed using other types. Types can be structured (sequences, sets, variants) or simple (integer, bit string, null, object identifier, octet string, printable string, UTC time, etc.). There are rules to transform the abstract notation into a binary representation. These rules are called *basic encoding rules (BER)* and *distinguished encoding rules*. BER describes how to encode values of an ASN.1 type as a string of octets. There is usually more than one encoding for each given ASN.1 type in BER. The distinguished encoding rules remove this ambiguity of BER by defining a unique encoding for each ASN.1 value. This is important if ASN.1 data has to be signed. The bit representation has to be unique, in order to obtain the same signature. There is also a *privacy enhanced mail (PEM)* format, which is the base-64 encoded form of DER.

4.1.2 Tools for Generating Certificates

OpenSSL [35] provides various tools that help with many aspects of a public-key infrastructure. These tools are available for Linux as well as for Windows and

Field	Description
Version	v3
Serial number	01 F1 00 00 0A 63
Signature algorithm	SHA1 with RSA
Issuer	CN = Swisskey ID Test CA Standard L = Zuerich OU = Identities not verified OU = Test Certificates only OU = 008510000000500000600 O = Swisskey AG C = CH
Validity period	not before: 10/26/2000, not after: 11/26/2000
Subject	E = someone@inf.ethz.ch CN = apachesslserver.inf.ethz.ch C = CH OU = 01.01.1950 OU = 008510001141100000179 O = Private Individual
Subject's public key	RSA (1024 bits): 30 81 89 02 ...
Extensions	Key Usage: Digital Signature , Client Authentication Revocation URL: https://crl.swisskey.ch/prodssl/get_status?sid= Certificate Policy URL: http://www.swisskey.ch/prod/cps

Table 4.2: Example X.509 certificate

take (almost) the same command line arguments.[1] For Linux these tools are documented in a set of man pages.

In the following it will be explained how to generate server and client certificates with these tools. The Swisskey CA allows to request test certificates via the Internet. These certificates can be generated without lengthy authentication procedures, which is very convenient for development purposes.

Generating a Server Certificate

The following command generates an (unencrypted) RSA key pair and a certificate request. The operation of the command is controlled by a configuration file. Some values can also be entered interactively. See the OpenSSL documentation for further information. When the tool asks for a common name (CN) the domain name of the Web server that contains the private Web pages has to be entered (e.g. `apachesslserver.inf.ethz.ch`).

```
openssl req -new -nodes -keyout keypair.pem -out request.pem
```

The new RSA key pair is stored in `keypair.pem`, the certificate request is

[1] There are also MacOS and VMS versions of these tools.

stored in `request.pem`. The key pair consists of the items shown below. The public key comprises just the modulus (n) and the public exponent (e), whereas the private key comprises the modulus (n) and the private exponent (d).

Modulus	n
Public exponent	e
Private exponent	d
Prime 1	p
Prime 2	q
Exponent 1	d mod (p-1)
Exponent 2	d mod (q-1)
Coefficient	(inverse of q) mod p

Table 4.3: Contents of key-pair file (no encryption)

The certificate request contains the public key information, subject identification information and other attributes from the configuration file. This information is signed with the subject's private key. The request is then sent to a CA, which has the ultimate control over the issuance of the requested certificate. It has to decide how to authenticate the requesting subject and which fields from the certificate template are to be put in the actual certificate. The CA may also add additional fields, like information about its issuance policy.

Request ID	ID for matching certificate request and reply
Certificate template	Desired field values of the certificate to be issued, the format is the same as for the certificate itself
Issuance controls	Specifies, e.g., how the certificate shall be published, how the requesting entity should be authenticated
Proof of possession	Signature of the above fields
Context information	Billing information, subscriber contact information

Table 4.4: Contents of certificate request file

Swisskey is a CA that allows to create test certificates that do not require any kind of authentication. To generate a *server* certificate the following steps are necessary:

- Point your Web browser to **http://www.swisskey.ch**
- Follow link **Swisskey ID for individuals**
- Follow link **Test Swisskey ID**
- If your browser does already contain the root certificate for the Swisskey Test CA, follow link **Install test root certificate**
- Follow link **Then apply for your test Swisskey ID**

- Choose **Applicant: Private individual**
- Choose **Purpose of usage: Server** (because we want to generate a **server** certificate)
- Some information about the subject has to be entered. It should match the data given in the certificate template. Make sure to choose **Directory release: None**, in order not to place this test certificate on any public certificate servers!
- When an input box is displayed, input the contents of file **request.pem** (including the first[2] and last[3] line).
- The certificate is now generated by the CA and can be downloaded and stored in a file.

Generating a Client Certificate

Creating a client certificate with the Swisskey Test CA differs a bit from the procedure described above.

- Point your Web browser to **http://www.swisskey.ch**
- Follow link **Swisskey ID for individuals**
- Follow link **Test Swisskey ID**
- If your browser does already contain the root certificate for the Swisskey Test CA, follow link **Install test root certificate**
- Follow link **Then apply for your test Swisskey ID**
- Choose **Applicant: Private individual**
- Choose **Purpose of usage: Browser/Client** (because we want to generate a **client** certificate)
- Some information about the subject has to be entered. It should match the data given in the certificate template. Make sure to choose **Directory release: None**, in order not to place this test certificate on any public certificate servers!
- An applet now generates the private key and performs the certificate request.

[2] `-----BEGIN CERTIFICATE REQUEST-----`
[3] `-----END CERTIFICATE REQUEST-----`

- The certificate can then be downloaded and is stored locally. For Netscape and Linux it is stored in **.netscape/swisskey/00xxx.p12**. For Internet Explorer and Windows it is stored in **C:\winnt\Java\swisskey\00xxxxx.pfx**. The format of these files is identical. It is called *PKCS #12* [49] – Personal Information Exchange Syntax Standard. They contain, in encrypted and MACed form, the whole certificate path from the root CA to the generated certificate as well as the the private key.

- The following command generates a printable form of all of these components:

```
openssl pkcs12 -nodes -in c00xxx.p12[4]
                      -out certpath+key.pem
```

- With a text editor the output file (certpath+key.pem) can now be edited and its components can be stored in separate files.

- The following command is useful to convert a base-64 encoded certificate to its binary (8 bits per byte) form:

```
openssl x509 -inform PEM -outform DER
             -in cert.pem -out cert.der
```

4.2 Apache-SSL

SSL and its successor TLS are the most widely used cryptographic protocols to make private Web pages and services securely accessible. Apache-SSL is an extension to the Apache [3] Web server that enables the delivery of Web content across SSL/TLS connections. It uses OpenSSL [35] as its cryptographic backend. Apache-SSL is free for both commercial and non-commercial use, makes available 128 bit encryption, and allows client authentication. Last but not least, its source code is open for inspection.

4.2.1 Apache-SSL Installation

The installation requires three components. The version numbers reflect the versions that have actually been used for the implementation. The Apache-SSL README.SSL file says, which versions of Apache and OpenSSL it requires (see section “Prerequisites“).

- OpenSSL: openssl-0.9.6.tar.gz,
  available at http://www.openssl.org/source/

- Apache-SSL extension: apache_1.3.14+ssl_1.42.tar.gz,
  available at http://www.apache-ssl.org/

[4] `.pfx`, respectively

- Apache HTTP server: apache_1.3.14.tar.gz,
  available at http://httpd.apache.org/

OpenSSL has to be installed first. There are several INSTALL files for different platforms. The README file gives general information.

Then the Apache-SSL extension patch has to be applied to the unpacked Apache package. The details are explained in file README.SSL of the Apache-SSL extension package. After the patch is applied, the standard Apache server installation procedure can be performed. For details, see the Apache INSTALL file and the documentation on the Apache Web site: http://httpd.apache.org/docs/.

4.2.2 Apache-SSL Configuration

General information on how to configure Apache is given in various readme files and on the Apache Web site (http://httpd.apache.org/docs/). The most important configuration file is **httpd.conf**, located in apache/conf. The installation produces a well documented example httpd.conf file. This main configuration file contains so-called *directives*, that adjust all aspects of Apache. Some of the Apache-SSL specific directives will be shortly described here. For a comprehensive description, see http://www.apache-ssl.org/docs.html.

- The TCP port Apache-SSL listens on is configured with the **Port** directive. The standard SSL/TLS port is 443. This port is only accessible if you are *root* on the installation machine.
- **SSLCertificateFile** points to the base-64 encoded certificate of the server. It seems that the (encrypted) private key has to be combined with the certificate, although if there is a directive called **SSLCertificateKeyFile**. The combined private key and certificate file of the server looks like this:

```
-----BEGIN RSA PRIVATE KEY-----
<base-64 encoding of (encrypted) private key>
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
<base-64 encoding of certificate>
-----END CERTIFICATE-----
```

- **SSLVerifyClient** specifies if client authentication is performed. **0** means no certificate is requested from the client. **2** means that a valid client certificate must be presented and verified. The latter value suits our needs.
- **SSLVerifyDepth** sets an upper limit on the length of the certificate path.
- **SSLCACertificatePath** is the path to a directory of base-64 encoded CA certificates. For each CA that appears in the certification path of a client certificate, there must be a corresponding CA certificate in this directory. If Swisskey test certificates are used for example, then certificates for “Swisskey

Test Root CA" and "Swisskey ID Test CA Standard" have to be present in this directory. Apache-SSL looks up these certificates using hash values of their common name. Therefore the individual certificate files need to be named with this hash value. Alternatively links to the original files may be established.

Hash values are computed with an OpenSSL tool:

```
openssl x509 -hash -noout -in SwisskeyTestRootCA.pem
```

A script that computes hash values and produces links to all certificates in the CA certificate directory might look like this:

```
#! /bin/sh
for c in *.pem; do
    ln -s $c `openssl x509 -hash -noout -in $c`
done
```

- Normally Apache-SSL sends a list of acceptable CAs to requesting clients during the SSL/TLS handshake protocol. This can be disbled, using the **SSLNoCAList** directive.

- **SSLSessionCacheTimeout** determines the period of time (in seconds) that a session key is valid, once negotiated. If the client makes new requests after this period is timed out, it will have to perform the whole SSL/TLS handshake protocol again. This includes client authentication, if enabled. For testing, it is convenient to set this parameter to a low value, e.g. to 10 seconds.

4.2.3 CGI Scripts

When user authentication is enabled, various SSL/TLS related parameters are made accessible to CGI scripts via CGI environment variables. These Apache-SSL specific parameters are shown in table 4.5.

This allows the CGI script to take different courses of action depending on the distinguished name (DN) of the client, the IP address of the client's machine[5], the strength of the encryption method and so on. Most importantly for our scenario, it enables the delivery of personal Web pages to individual users, without the requirement that each user has to enter an individual URL. The Web pages and services for the requesting user can simply be found as a function of the user's distinguished name.

It is even possible to export the whole certificate path to a CGI script using the directive **SSLExportClientCertificates**. The certificates can then be accessed via the enironment variables SSL_CLIENT_CERT and SSL_CLIENT_CERT_CHAIN_n, where n runs from 1 upwards.

There is also a module name **KeyNote** that allows access to pages based on client certificate or certificate authority.

[5]The IP address is given in the standard Apache CGI variable REMOTE_ADDR.

CGI Variable	**Description**
HTTPS	HTTPS is being used
HTTPS_CIPHER	SSL/TLS cipherspec
SSL_CIPHER	The same as HTTPS_CIPHER
SSL_PROTOCOL_VERSION	Self explanatory
SSL_SSLEAY_VERSION	Self explanatory
HTTPS_KEYSIZE	Number of bits in the session key
HTTPS_SECRETKEYSIZE	Number of bits in the secret key
SSL_CLIENT_DN	DN in client's certificate
SSL_CLIENT_<x509>	Component of client's DN
SSL_CLIENT_I_DN	DN of issuer of client's certificate
SSL_CLIENT_I_<x509>	Component of client's issuer's DN
SSL_SERVER_DN	DN in server's certificate
SSL_SERVER_<x509>	Component of server's DN
SSL_SERVER_I_DN	DN of issuer of server's certificate
SSL_SERVER_I_<x509>	Component of server's issuer's DN
SSL_CLIENT_CERT	Base-64 encoding of client cert
SSL_CLIENT_CERT_CHAIN_n	Base-64 encoding of client cert chain

Table 4.5: Apache-SSL CGI environment variables

Chapter 5

Netscape and PKCS #11

Having looked at the server side it is now time to consider the client side. Our client application is the Netscape Web browser. To branch the signing functionality out of Netscape into to a SIM in a mobile phone, Netscape has to be adapted.

All cryptography-related functionality of Netscape is located in a specific module. By default Netscape uses an internal module that handles all cryptographic functionality in software. This module can be replaced by an external module, like one that implements signing by external hardware in our case. An external module must conform to the *PKCS #11* specification, which defines the module's application programming interface (API) and externally accessible data structures.

This chapter describes the PKCS #11 specification in general, a GNU implementation by *TC TrustCenter GmbH* in praticular, and the steps that are necessary to configure Netscape for an external cryptographic module.

5.1 Public-Key Cryptography Standards (PKCS)

The *Public-Key Cryptography Standards (PKCS)* [24, 25, 37] are a family of standards, produced by *RSA Laboratories* (www.rsalabs.com), to achieve the interoperability of different implementations of public-key cryptography. An important motivation of RSA Laboratories is to accelerate the deployment of public-key cryptography. Rather than waiting for official standards, RSA tries to produce *de facto* standards. RSA is rather successful with this approach, as PKCS "standards" are widely used. Table 5.1 gives an overview of the PKCS standards defined so far.

5.2 PKCS #11 Cryptographic Token Interface Standard

External cryptography modules for Netscape need to conform to the *Cryptographic Token Interface Standard (PKCS #11)* [48], also called *Cryptoki*. PKCS #11 isolates an application from the details of a cryptographic device. It specifies a generic API to devices which hold cryptographic information and perform cryptographic functions.

PKCS #1	RSA Cryptography Standard
PKCS #3	Diffie-Hellman Key Agreement Standard
PKCS #5	Password-Based Cryptography Standard
PKCS #6	Extended-Certificate Syntax Standard
PKCS #7	Cryptographic Message Syntax Standard
PKCS #8	Private-Key Information Syntax Standard
PKCS #9	Selected Attribute Types
PKCS #10	Certification Request Syntax Standard
PKCS #11	Cryptographic Token Interface Standard
PKCS #12	Personal Information Exchange Syntax Standard
PKCS #13	Elliptic Curve Cryptography Standard
PKCS #15	Cryptographic Token Information Format Standard

Table 5.1: PKCS #i "standards" by RSA Laboratories

PKCS #11 follows an object-based approach, which favors technology independence (any kind of device) and resource sharing (multiple applications accessing multiple devices). The goal is to present to applications a common logical view of the device, abstracted as a *cryptographic token*.

5.3 PKCS #11 General Model

Figure 5.1 shows the general architectural model of PKCS #11 and its concrete instantiation with Netscape. The right part of the figure is just shown as an overview. It will be explained in detail in sections 5.6 and 5.7.

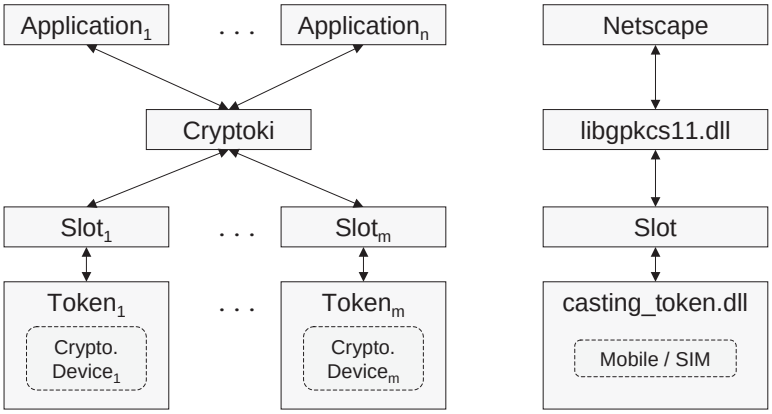

Figure 5.1: PKCS #11 general model and instantiation with Netscape

At the highest layer of the architecture are applications that need to perform cryptographic operations. The cryptographic functionality is implemented in one or more cryptographic devices. These devices are abstracted as so-called *tokens*

that hide the device-specific details. The tokens are inserted in *slots*, provided by Cryptoki. Applications connect to tokens that are present in slots. Multiple applications can access multiple tokens. Therefore, there needs to be some kind of multiplexing and mutual exclusion. The central Cryptoki component is responsible for managing access to tokens.

In our configuration, the cryptographic device is the SIM inside the mobile phone, which is abstracted with our token, called `casting_token.dll`. Netscape is configured to run with the GNU PKCS #11 library (details below), which is implemented as a dynamically loadable shared library. In our setup there is only one active application using Cryptoki – namely Netscape – and one filled slot – the one containing the CASTING token.

5.4 PKCS #11 Application Programming Interface

PKCS #11 provides about 70 functions to applications, which can be classified as follows:

- General-purpose (initialization, meta information)
- Slot and token management
- Session management
- Object management
- Encryption
- Decryption
- Message digesting
- Signing and MACing
- Verifying signatures and MACs
- Dual-purpose cryptographic functions
- Key management
- Random number generation
- Parallel function management

5.5 CASTING PKCS #11 Token

Our special PKCS #11 token performs all cryptographic operations for Netscape during an SSL/TLS protocol session. This includes the SSL/TLS handshake phase, especially the operations needed for user authentication, as well as the encryption of application data with SSL/TLS, using symmetric bulk cipher encryption.

Most of these operations are executed on the terminal. In particular, the symmetric encryption and message authentication during normal operation of SSL/TLS, i.e. once the security parameters have been established, are completely done by the terminal. Bulk cipher encryption is performance critical with respect to the application. Therefore it is a requirement to execute it on the terminal, given that the bandwidth to the mobile and the processing capabilities are very limited.

The user has her RSA key pair and her certificate stored on her mobile. The CASTING token reads the user certificate from the mobile and transfers it to the terminal. It will be queried by Netscape, just before the SSL/TLS session is established. In contrast, the user's private key never leaves the SIM.

This implies that all operations concerning the private key are executed on the SIM itself. Fortunately, the only operation on the client's private key within the SSL/TLS protocol is the computation of a signature during client authentication. This has to be done only once for a SSL/TLS session.

This split in functionality serves two goals. One the one hand, the private key is kept private in the user's SIM, which is absolutely crucial. On the other hand, the high performance during normal SSL/TLS operation is preserved.

5.6 GNU PKCS #11 (gpkcs11)

TC TrustCenter GmbH (http://www.trustcenter.de) offer an open source implementation of PKCS #11, called *gpkcs11* [36], that is distributed under the GNU Lesser General Public License (LGPL). We decided to use this implementation as a starting poing for the development of the CASTING token. gpkcs11 provides a software-only token that we adapted to our needs.

gpkcs11 is can be compiled for various platforms, such as Solaris 2.5.1/SPARC, Linux 2.0.36/i386, and Windows 9x, NT, and 2000. For Windows, workspace and project files for Microsofts Visual C++ 6.0 are included. As a prerequisite, an OpenSSL library is needed. The archive file for the sources is called `gpkcs11-0.6.1.tgz`. For the following discussion it is assumed that our adapted version of gpkcs11-0.6.1, i.e. the CASTING token implementation, is used and is unpacked to C:\. The implementation of the CASTING token is explained in detail in chapter 6.

The gpkcs11 installation, like Apache-SSL, uses OpenSSL as its cryptographic backend. This means that LIBEAY32.dll must be accessible via the system path. The CASTING token also requires the availability of the OpenSSL source code. In the following it is assumed that the OpenSSL package is installed at C:\.

For information on installation, compilation, and configuration of gpkcs11 see the `Readme.txt` file of the source distribution package.

5.7 Configuring Netscape for the gpkcs11 Module

The client application for our scenario is Netscape Communicator with 128 bit encryption. We used versions 4.73 and 4.75 for the Windows operating system. For our purposes Netscape‘s internal cryptographic module needs to be disabled and replaced by the external gpkcs11 library. As is shown on the right half of figure 5.1, each token for the gpkcs11 module (libgpkcs11.dll) is a separate library. In our configuration there is just one token present (casting_token.dll in figure 5.1). This allows gpkcs11 to use different tokens, simply by editing a gpkcs11 configuration file. Now the necessary configuration steps are explained in detail.

First the internal module needs to be disabled. To achieve this, press the **Security** button on Netscape‘s toolbar. After choosing **Cryptographic Modules**, a list of all installed cryptographic modules appears. They all have to be disabled as follows: Select the module in the listbox and press **View/Edit** (the topmost button shown in figure 5.2). For all tokens of the module, press **Config** and select **Disable this token**.

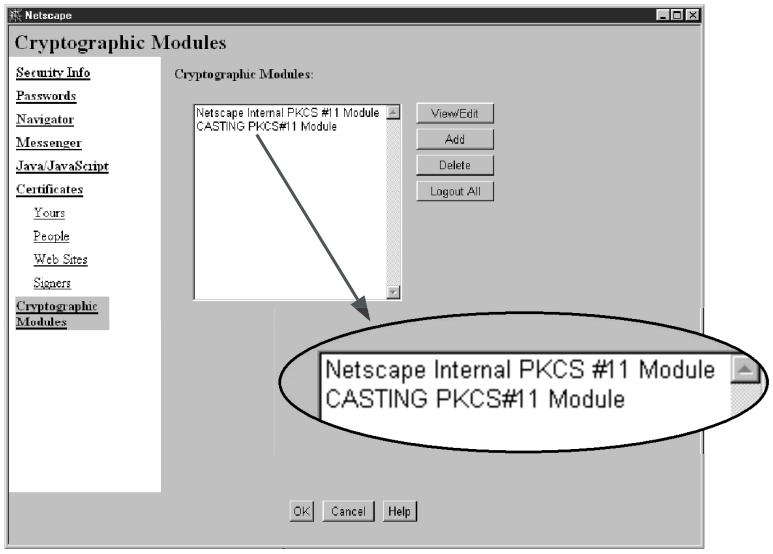

Figure 5.2: Netscape configuration with an external PKCS #11 module

Now the CASTING token has to be installed. In the **Cryptographic Modules** dialog box choose **Add**. A new dialog box appears with the caption **Create a New Security Module**. In the **Module Name** input field, type a name, e.g. `CASTING PKCS #11 module`. In the **Security Module File** enter the path to the gpkcs

module file[1] e.g. `C:\gpkcs11-0.6.1\winnt\libgpkcs11\Debug\libgpkcs11.dll`. The new token appears in the listbox of the **Cryptographic Modules** dialog box. Selecting the new module and pressing **View/Edit** and **More Info** shows information like in figure 5.3.

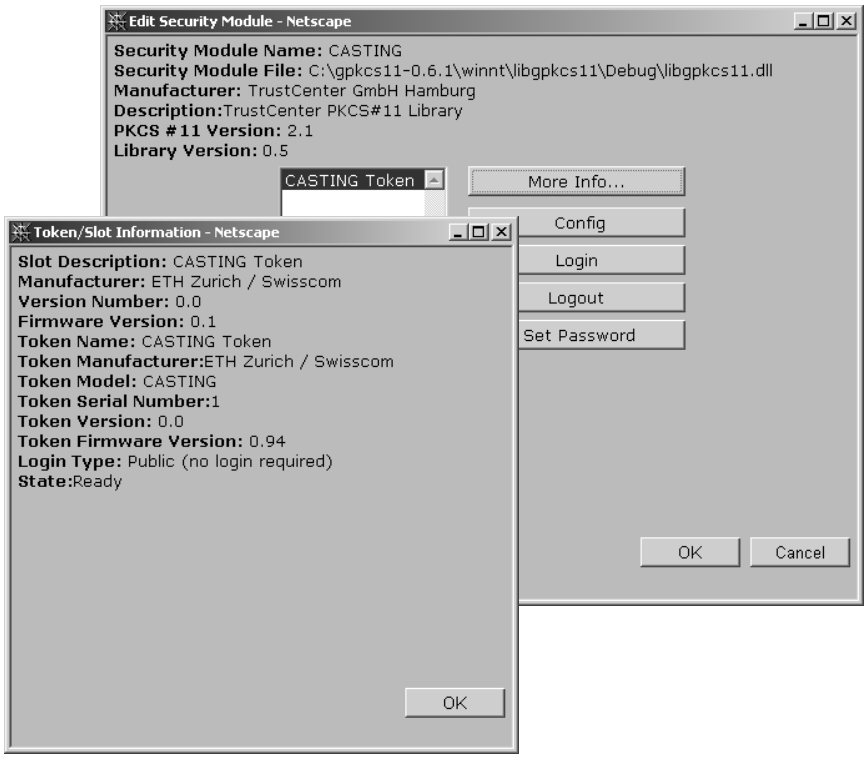

Figure 5.3: Netscape information about CASTING PKCS #11 token

The cryptographic methods of the token now have to be configured. This can be achieved by selecting CASTING module in the listbox of the **Cryptographic Modules** dialog box and pressing **View/Edit**. The **Edit Security Module** dialog box appears. Selecting the CASTING token in the listbox and pressing **Config** makes the **Configure Slot** dialog box appear. The token has to be enabled and the checkboxes have to be set exactly as shown in figure 5.4. RSA has to be enabled for the SSL/TLS handshake phase. RC4 is the bulk cipher algorithm of choice. SHA-1 and MD5 are needed for message authentication and key generation in SSL/TLS. The token also provides "publicly-readable certs" and a random number generator.

In the **Security** dialog box SSL/TLS version three has to be enabled: **Enable SSL (Secure Sockets Layer) v3**. Also all user certificates should be removed, as the mobile will provide the user certificate later. To simplify user interaction set **Certificate to identify you to a web site** to **Select Automatically**. The SSL/TLS

[1]Note: The name of the gpkcs11 module – usually called libgpkcs11.dll – needs to be entered here, **not** the name of the token.

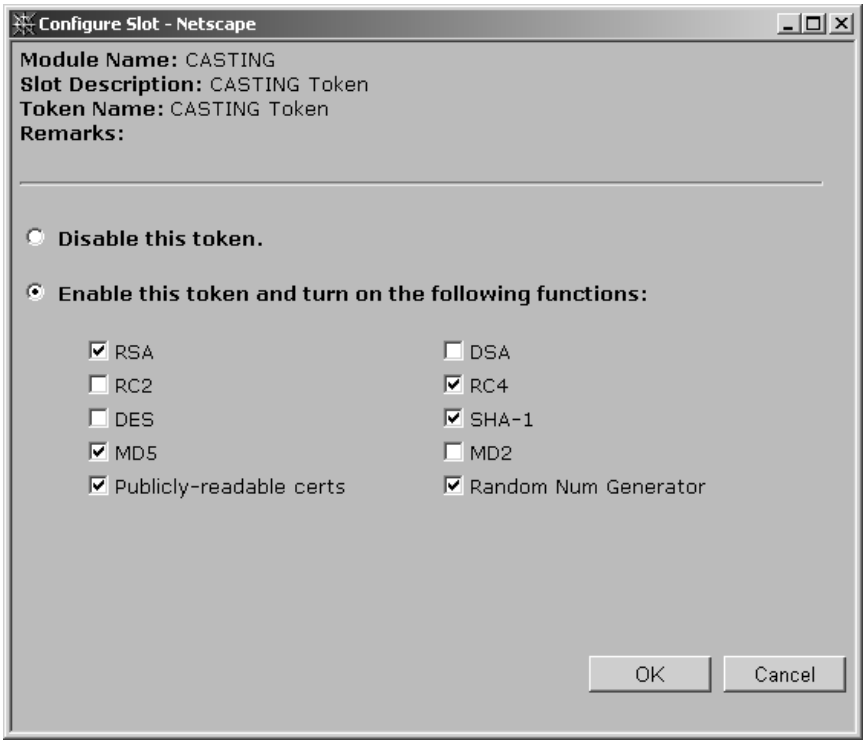

Figure 5.4: Configuration of CASTING PKCS #11 token

cipher suites for the CASTING PKCS #11 token should be set as in figure 5.5.

Having configured Netscape, it is now necessary to configure gpkcs11 to use the CASTING token. This is done via an initialization file, called `gpkcs11.ini`. This file needs to be put into the Windows system directory, i.e. C:\WINNT or C:\WINDOWS.

```
[PKCS11-DLL]
TokenList = CASTING-TOKEN
LoggingFile = C:\pkcs11.log
MemLoggingFile = C:\pkcs11mem.log
LoggingLevel = 3

[CASTING-TOKEN]
TokenDLL = C:\gpkcs11-0.6.1\winnt\ceay_token_IR\Debug\ceay_token.dll
InitSym = ceayToken_init
Port = COM4
```

The section [PKCS11-DLL] contains configuration for the gpkcs11 library. The TokenList attribute specifies the tokens present in that configuration. Each token has its own section. Our configuration contains only one token, which is described in section [CASTING-TOKEN]. The TokenDLL attribute names the shared library file for the token. The Port attribute is only relevant for the IrCOMM implementation of the CASTING token (see chapter 6). For more information on

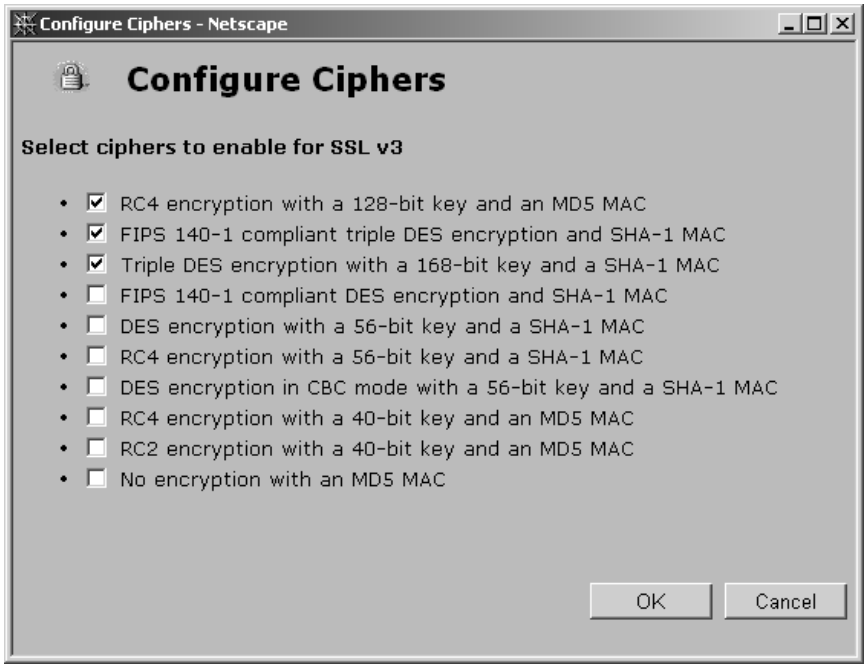

Figure 5.5: Preferred SSL/TLS cipher suites for CASTING PKCS #11 token

the attributes of the configuration file consult the documentation provided with the gpkcs11 distribution.

The CASTING token shared library (ceay_token.dll) depends on a number of other shared libraries. In particular it depends on WS2_32.dll for Windows sockets operations, LIBEAY32.dll which provides the cryptographic backend, and libgpkcs11.dll in which it is plugged in. These libraries therefore have to be accessible via the PATH environment variable. LIBEAY32.dll should be located in C:\openssl-0.9.5\out32dll; libgpkcs11.dll in C:\gpkcs11-0.6.1\winnt\libgpkcs11\-Debug.

Chapter 6

Implementation

This chapter deals with the main points of the CASTING implementation. The implementation consists of two parts, one running on the fixed terminal, the other on the SIM inside the mobile. The flow of messages exchanged between terminal and mobile and some of the issues involved with IrDA and Bluetooth short distance communication will be discussed. Finally, the demo setup, debugging, and testing tools are presented.

6.1 Implementation Overview

An overview of all of the components involved in the CASTING implementation is given in figure 6.1. Most of these components have been discussed before. This chapter concentrates on the implementation of the CASTING PKCS #11 token, the CASTING SIM applet resp. its simulation, and the implementation of the SECTUS protocol.

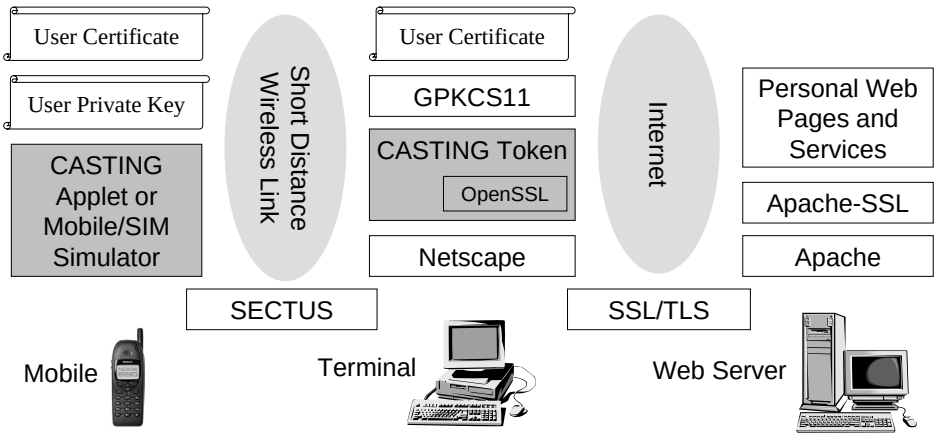

Figure 6.1: Implementation Overview

6.2 Development Software

The following software packages have been used during implementation:

OpenSSL 0.9.5a	Cryptographic backend, testing tools
gpkcs11 5.6.1	Included modificatons done by Felix Baessler
Netscape 4.73	With 128 bit encryption (Windows NT 4.0 and 2000)
ApacheSSL	apache_1.3.12/1.3.14 + openssl-0.9.5a/0.9.6 (Linux)
Visual C++ 6.0	Microsoft‘s development environment for Windows

These tools have been described before. Visual C++ is Microsoft‘s standard development environment for the Windows platform. The implementaton of the CASTING PKCS #11 token is based on a software-only token – called CEAY token – that comes with the gpkcs11 distribution package.

6.3 SECTUS Protocol Implementation

The simulator acts as a server that provides two operations. One is the execution of the SECTUS protocol, the other is signing the hash value. Prior to performing the latter operation there must have been established an authentication key between terminal and SIM. The operation of the simulator is shown in figure 6.2. The simulator reads the user certificate and key from files that are given as arguments on the command line. It waits until it receives an opcode and dispatches to the appropriate functions.

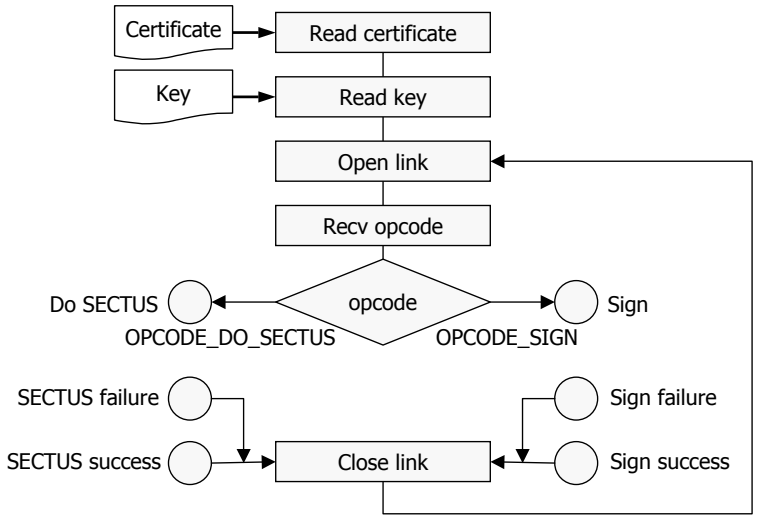

Figure 6.2: Control loop of the simulator

Figure 6.3 shows the processing of the SECTUS protocol. The actions of the mobile/SIM part are shown on the left; the actions of the terminal on the right. The

grey area in the middle represents the short distance wireless link. The timeline runs from top to bottom. SECTUS fails, when the certificate is not accepted by the terminal, when the user cancels the input dialog (shown in figure 6.4), when a MAC cannot be verified, or when a communication error occurs.

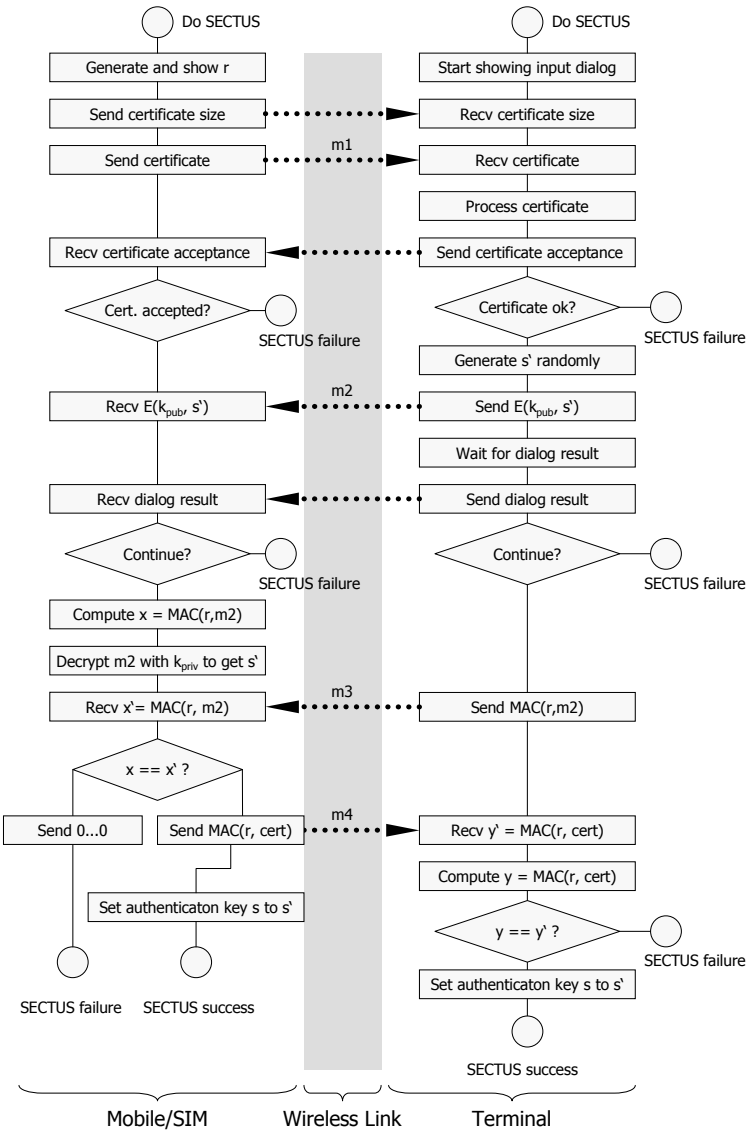

Figure 6.3: Implementation of the SECTUS protocol

Before the signing operation is performed it is first checked if the authentication key s is set. If this is not the case the operation is cancelled. The hash value to be signed as well as the signature are protected by MACs. The signature is computed using the private key of the user.

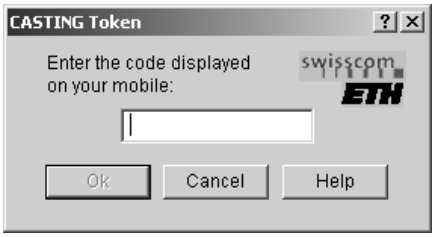

Figure 6.4: Dialog box for entering the one-time password of the SECTUS protocol

6.4 Implementation of the Short-Distance Link

There are several implementations of the CASTING PKCS #11 token that differ only in the realization of the short distance link. The following variants have been implemented:

Short distance link	**Implementation directory**
TCP/IP sockets (wired)	C:\gpkcs11-0.6.1\winnt\ceay_token
IrCOMM and RFCOMM Bluetooth	C:\gpkcs11-0.6.1\winnt\ceay_token_IR
IrSock (Windows 2000)	C:\gpkcs11-0.6.1\winnt\ceay_token_IRSock

The first implementation uses TCP/IP sockets to establish a wired short distance link. This was a first step to implement the SECTUS protocol. The second implementation is designed for IrCOMM infrared and Bluetooth RFCOMM. There are corresponding implementation directories for the mobile simulator, called *mobile_sim_SECTUS*, *mobile_sim_SECTUS_IR*, and *mobile_sim_SECTUS_IRSock*, respectively.

6.4.1 IrComm with Windows 9x and NT

IrCOMM [13] and also RFCOMM emulate a serial cable connection between two devices. An issue with this interface is that there is no notification of devices appearing and disappearing. The appropriate interfaces for device and service discovery, which exist in the IrDA specification, are hidden, because IrDA device manufacturers only provide the IrCOMM abstraction as an API. Using IrCOMM makes the implementation more clumsy than it would to be, if the appropriate interfaces were exposed to the API. IrCOMM treats the infrared connection like a file the can be written to and read from. The file name is “COMx” or “IrCOMMx”, where x is a virtual serial port (usually x is 4 or 5). The good thing about IrCOMM is that it can be used with legacy applications that use a serial port for communication.

The IrCOMM link proved to be unreliable. There were different problems, depending on the operating system used (Windows 95, 98, or NT 4.0). Sometimes opening the IrCOMM link failed, sometimes a write would hang for an indefinite amount of time.

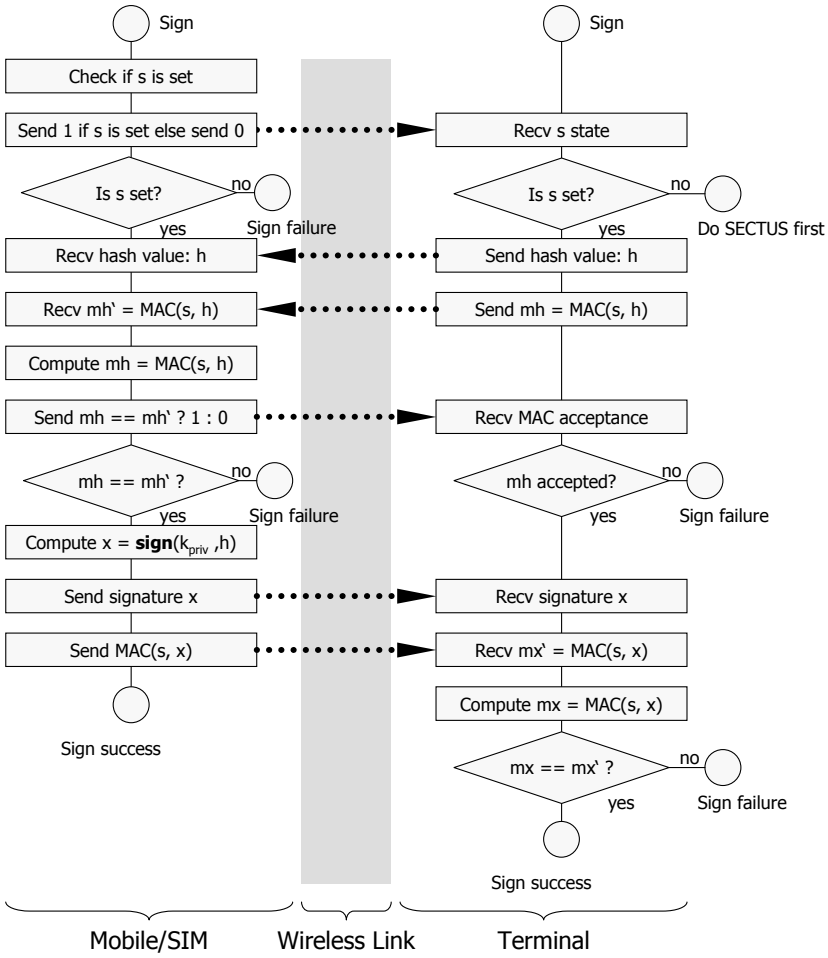

Figure 6.5: Implementation of the signing operation

6.4.2 IrSock with Windows 2000

Microsoft has recognized the problems with IrCOMM and does not support this interface for Windows 2000 anymore. Instead, infrared communication is based on the socket interface, familiar from TCP/IP sockets. Infrared support is better integrated into the operating system services and IrDA device discovery is possible. IrSock seems to be much more reliable than IrCOMM.

Development using IrSock requires the installation of the Windows 2000 *Device Driver Kit (DDK)*, which is accessible from Microsoft's developer Web site.

The device that provides a service typically acts as an IrSock server. The service is added to the IAS database local to the device on which it is located. The client can remotely query this database and find the service it wants to connect with. IrSock exposes TinyTP [16] over IrLMP [14] to the application programmer.

A good overview of IrDA is given [29]. In depth information on IrDA can be

found in [14, 15, 16, 17, 18, 19, 20].

6.4.3 Bluetooth Communication

Bluetooth [4] is a specification for short-distance (less than 10 m) radio frequency communication. Bluetooth is not yet integrated into the Windows operating system, but there is some activity by Microsoft in this area. There are no standard Bluetooth APIs for Windows yet. Therefore in our implementation we used RFCOMM, which has the same shortcomings as IrCOMM, but can be used with the IrCOMM version of our implementation with almost no modifications.

[30] discusses the Bluetooth protocol architecture. Bluetooth is specified in [5] and [6]. Bluetooth also considers authentication and confidentiality. This topic is discussed in [31].

For the project, two PCMCIA Bluetooth cards from *Digianswer* have been used.

6.5 CASTING PKCS #11 Token

In each implementation directory there is a Visual C++ workspace (called *ceay_token.dsw*) and source files. The most important source files are the following:

Source file	Functionality
ceay_token.c	Implementation of the CASTING gpkcs11 token
mobile_proxy.c	Handles the interaction with the counterpart on the mobile
comIR.c	Provides functions for communication with the mobile

6.5.1 ceay_token.c

The original software-only token had to be modified at several places to work properly. First, a longer key block had to be generated, from which clientMAC, serverMAC, clientKey, and serverKey are derived. For 128-bit encryption 64 bytes of key material are needed. The original implementation just generated 48 bytes of key material. To get a larger key block more “salt” and more iteratons while generating the key material are necessary. The key block is generated like this:

```
key_block =
   MD5(master_secret + SHA('A' + master_secret +
                           ServerHello.random +
                           ClientHello.random)) +
   MD5(master_secret + SHA('BB' + master_secret +
                           ServerHello.random +
                           ClientHello.random)) +
   MD5(master_secret + SHA('CCC' + master_secret +
                           ServerHello.random +
                           ClientHello.random)) +
```

```
MD5(master_secret + SHA('DDDD' + master_secret +
                        ServerHello.random +
                        ClientHello.random)) +
[...];
```

Another major problem was that the functions `GetOperationState` and `SetOperationState` were not implemented, although Netscape apparently needs them at different occasions.

A minor error was that the random number generator was never seeded. This led to an error condition when calling `RSA_public_encrypt`, which is a function provided by the OpenSSL library. Initialization can be done by using function `RAND_seed`. Maybe it is important to mention here that OpenSSL 0.9.5a was used (the newest version of OpenSSL available), not version 0.9.4 that is recommened by TrustCenter. It should be noted that the randomness of the seed of the current implementation should be improved.

There were multiple minor bugs that were relatively easy to fix, like inserting a missing `break` statement, disabling some checks, and removing unneeded header files.

The original token read the user certificate from a local database, called *cryptdb*. This database was removed, as all user related information is stored inside the SIM. The CASTING implementation does not store any user related data persistently.

`CI_Ceay_OpenSession` and `CI_Ceay_Sign` are the most heavily adapted functions.

`CI_Ceay_OpenSession` first checks if the user certificate has changed, by calling `mobile_certificateChanged` of the mobile_proxy component. This happens if a new mobile device/SIM has connected to the terminal. If it has changed, the new certificate is formatted as two PKCS #11 objects: a certificate object and a public-key object. These are added to an internal cache (nonappropriately called `persistent_cache`). Then the change is propagated to all sessions. Finally the session is added to the session list.

`CI_Ceay_Sign` is called by Netscape when the signature of the SSL/TLS hash value has to be computed. For our purposes only the signature type `CKM_RSA_PKCS` is relevant. The actual cryptographic signing operation is no longer performed by the function itself, but outsourced to the mobile. Therefore the function just calls the functions `mobile_RSA_signLen` and `mobile_RSA_sign` located in the mobile_proxy component to determine the signature length[1] and to compute the signature from the 36 byte hash value.

6.5.2 mobile_proxy.c

This file is probably the best place to start exploring the CASTING implementation. The mobile_proxy module acts as a proxy for the mobile phone. In a separate

[1] The length of the signature depends on the size of the user's private key.

thread it continuously tries to establish short-distance connections to appearing mobile phones. When a connection is established, an input dialog is shown to enable the user to input the one-time password *r*. While the user inputs the password, the certificate is transferred in the background. This eliminates the perceived delay for the user.

`mobile_doSectus` preforms the SECTUS protocol, while `mobile_RSA_sign` is responsible for the signing operation. This function recognizes if an authentication key has been set and calls the `mobile_doSectus` function if necessary. The function `mobile_processCert` converts a DER-encoded X.509 certificate, as read from the mobile, into a PKCS #11 certificate object (type CKO_CERTIFICATE) and a PKCS #11 public key object (type CKO_PRIVATE_KEY[2]).

X.509 (from SIM)	**cert_attributes**	**privKey_attributes**
–	CKA_CLASS	CKA_CLASS
–	CKA_TOKEN	CKA_TOKEN
–	CKA_PRIVATE	CKA_PRIVATE
(1)	CKA_LABEL	CKA_LABEL
(2)	CKA_ID	CKA_ID
–	CKA_ALWAYS_SENSITIVE	CKA_ALWAYS_SENSITIVE
Subject	CKA_SUBJECT	CKA_SUBJECT
–	CKA_SENSITIVE	CKA_SENSITIVE
–	CKA_APPLICATION	
(3)	CKA_VALUE	
(4)	CKA_CERTIFICATE_TYPE	
Issuer	CKA_ISSUER	
Serial Number	CKA_SERIAL_NUMBER	
–	CKA_DERIVE	
–	CKA_MODIFIABLE	
–		CKA_LOCAL
Exponent		CKA_PUBLIC_EXPONENT
Modulus		CKA_MODULUS
(5)		CKA_KEY_TYPE
–		CKA_EXTRACTABLE
Version	–	–
Signature Alg.	–	–
Not Before	–	–
Not After	–	–
X.509v3 ext.	–	–

Table 6.1: Conversion of X.509 attributes (left column) to PKCS #11 object attributes (middle and right column)

[2]Despite its name it does not contain private key information.

1) Human-readable identifier for the certificate
2) PKCS #11-specific ID of the certificate needed to match the two PKCS #11 objects: certificate and private key
3) The whole x509 certificate in binary form (as we read it from the mobile/card)
4) Defined certificate types (we need CKC_X_509)
```
#define CKC_X_509 0x00000000
#define CKC_VENDOR_DEFINED 0x80000000
```
5) Defined key types (we need CKK_RSA)
```
#define CKK_RSA 0x00000000
...
```

6.5.3 comIR.c

The comIR module is responsible for the implementation of the short distance communication. It provides functions to open and close the link and to exchange data.

6.6 Usage, Testing, and Debugging

6.6.1 OpenSSL Server Tool

To test the terminal and the simulator, you need an HTTP server that is able to establish SSL connections and that asks the client for its certificate. This can be done by setting up an HTTP server like Apache with an SSL module. Alternatively a tool from the openssl distribution can be used to simulate the connection establishment between the terminal and an SSL HTTP server. *openssl s_server* is such a tool that acts like an SSL HTTP server. The following describes the setup of the openssl tool for Windows NT. It is also available for Linux with roughly the same options.

Setup for openssl s_server:

- Create a directory that contains the trusted CA certificates, like the certificates of the Swisskey Test CA described in chapter 4. The client certificate that is used for testing has to be signed by one of these CA certificates. The directory is used by the *openssl s_server* tool to verify the client certificate.
- Copy a server certificate and private key to the directory in which the tool is started (the working directory of the tool). Both may be located in a single file. The certificate is used to identify our test server to its clients.
- Start the tool (the following is a single command line):

```
openssl s_server -accept 8088 -no_dhe -no_tmp_rsa
                 -Verify 4 -CApath C:\swisskey_certs
                 -state -www
```

At startup the tool generates the following output:

```
verify depth is 4, must return a certificate
Loading 'screen' into random state - done
ACCEPT
```

This indicates that the tool is ready to accept incoming connection requests. It listens on the port given by the `accept` command line argument. The client must provide a valid certificate (option `Verify` in capital case[3]) and the certificate path must be no longer than four certificates. The client certificate is verified against the CA certificates in the `swisskey_certs` directory. The `www` argument (in lower case) causes the return of a status page to the client. The tool can even act as a bare bones Web server, using the `WWW` option (in capital case). For a list of options type `openssl s_server -h`. For more information see the OpenSSL documentation.

6.6.2 CASTING PKCS #11 Token

Load Visual C++ and open the `ceay_token.dsw` workspace. Now the token can be debugged or executed, which automatically starts Netscape.

6.6.3 Mobile/SIM Simulator

To start the simulator, simply provide the file names of the certificate and the (unencrypted) key as command line arguments:

```
mobile_sim_SECTUS <user X.509 certificate file> <user RSA key file>
```

For the IrCOMM version there is an additional third argument, giving the serial port to use (e.g. `COM5`).

Figure 6.6 shows an example output of the simulator. First, the CASTING protocol is performed to establish the authentication key. Then, a signature operation is executed to sign the SSL/TLS hash value.

6.6.4 Starting Netscape

Once configured, Netscape can be started in the usual way, e.g. by double-clicking its icon. It can also be started from within the Visual C++ environment for debugging. To start Netscape from Visual C++ the ceay_token project has to be open. Then the CASTING token DLL must be connected to Netscape. This is achieved by setting **Executable for debug session** in **general project options** to the path of the Netscape executable, e.g.
`c:\Program Files\Netscape\Communicator\Program\netscape.exe`.

[3] Note that case is significant for the options.

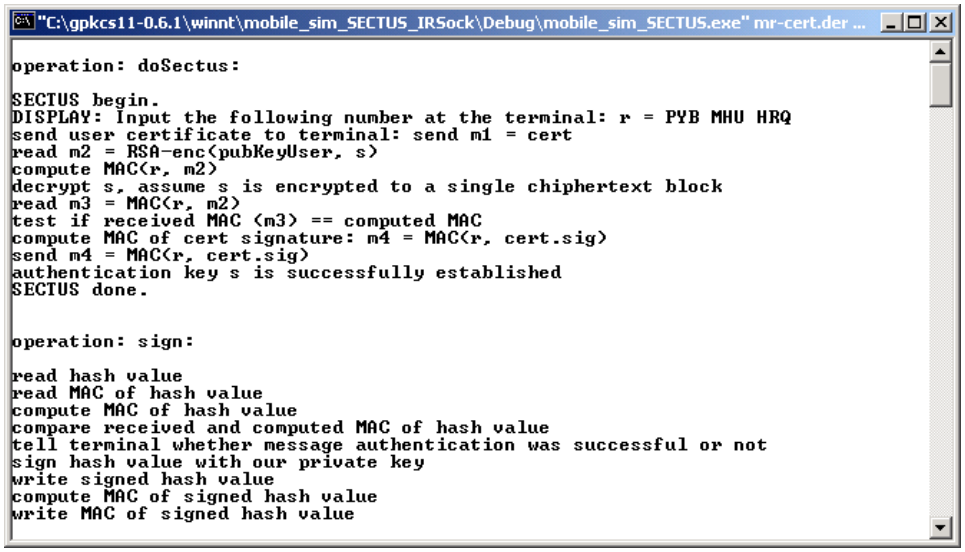

Figure 6.6: Example output of the simulator

6.6.5 Testing, Debugging and Demo Setup

When the simulator connects to the terminal, the user certificate is transferred. The one-time password of the SECTUS protocol is shown on the mobile display, and on the terminal screen an input dialog is presented to the user. After correct input of the one-time password, the authentication key is established. Now Netscape is ready to connect to an Apache-SSL server that requests client certificates or to the OpenSSL server tool.

The transferred user certificate can be displayed in Netscape, but only when an SSL connection has been established using that certificate. Netscape does not ask the CASTING PKCS #11 token for user certificates any earlier. Even if transferred, the certificate first remains internal to the CASTING token. To view the user certificate choose **Security** from the toolbar and select **Certificates/Yours**.

To connect to the OpenSSL server tool, input its URL into Netscape's location bar. E.g. assuming the server is located at **vs6.inf.ethz.ch**, type: **https://vs6.inf.ethz.ch:8088/**. If it is located on your local host, type: **https://localhost:8088/**

Netscape should now show connection and status information, which was generated by the OpenSSL server tool. If all is well, it should contain a line like this:
`Verify return code: 0 (ok)`

The openssl tool should display the following information about the connection. It gives detailed information about the steps of the SSL handshake protocol.

```
SSL_accept:before/accept initialization
SSL_accept:SSLv3 read client hello A
SSL_accept:SSLv3 write server hello A
SSL_accept:SSLv3 write certificate A
SSL_accept:SSLv3 write certificate request A
```

```
SSL_accept:SSLv3 flush data
depth=2 /
         CN=Swisskey Test Root CA/
         O=Swisskey AG/
         OU=008510000000500000903/
         OU=Test Certificates only/
         OU=Identities not verified/
         L=Zuerich/
         C=CH
verify return:1 depth=1 /
         CN=Swisskey ID Test CA Professional
         O=Swisskey AG/
         OU=008510000000500000701/
         OU=Test Certificates only/
         OU=Identities not formally verified/
         L=Zuerich/
         C=CH/
verify return:1 depth=0 /
         CN=Rene Frutiger/
         O=Swisscom AG, Bern/
         OU=008510000684600000518/
         OU=Corporate Information + Technology/
         2.5.4.17=3050/
         L=Bern/
         C=CH/
verify return:1
SSL_accept:SSLv3 read client certificate A
SSL_accept:SSLv3 read client key exchange A
SSL_accept:SSLv3 read certificate verify A
SSL_accept:SSLv3 read finished A
SSL_accept:SSLv3 write change cipher spec A
SSL_accept:SSLv3 write finished A
SSL_accept:SSLv3 flush data
ACCEPT
```

The simulator is called by Netscape to sign the hash value that was computed during the SSL/TLS handshake protocol. The steps of the SECTUS protocol are written to standard output by the simulator. The token writes the steps of the protocol into a log file, if the logging level is high enough. The location of the log file and the amount of logging information are specified in the `gpkcs11.ini` file.

The software package also contains some CGI scripts that were used for demonstration purposes. These scripts show the CGI environment variables that are passed to the script and return information specific to the user connecting.

An example Web page generated by the second CGI script is shown in figure 6.7. Information about the protocol versions, the server and the client host, and about the client and server certificates is presented.

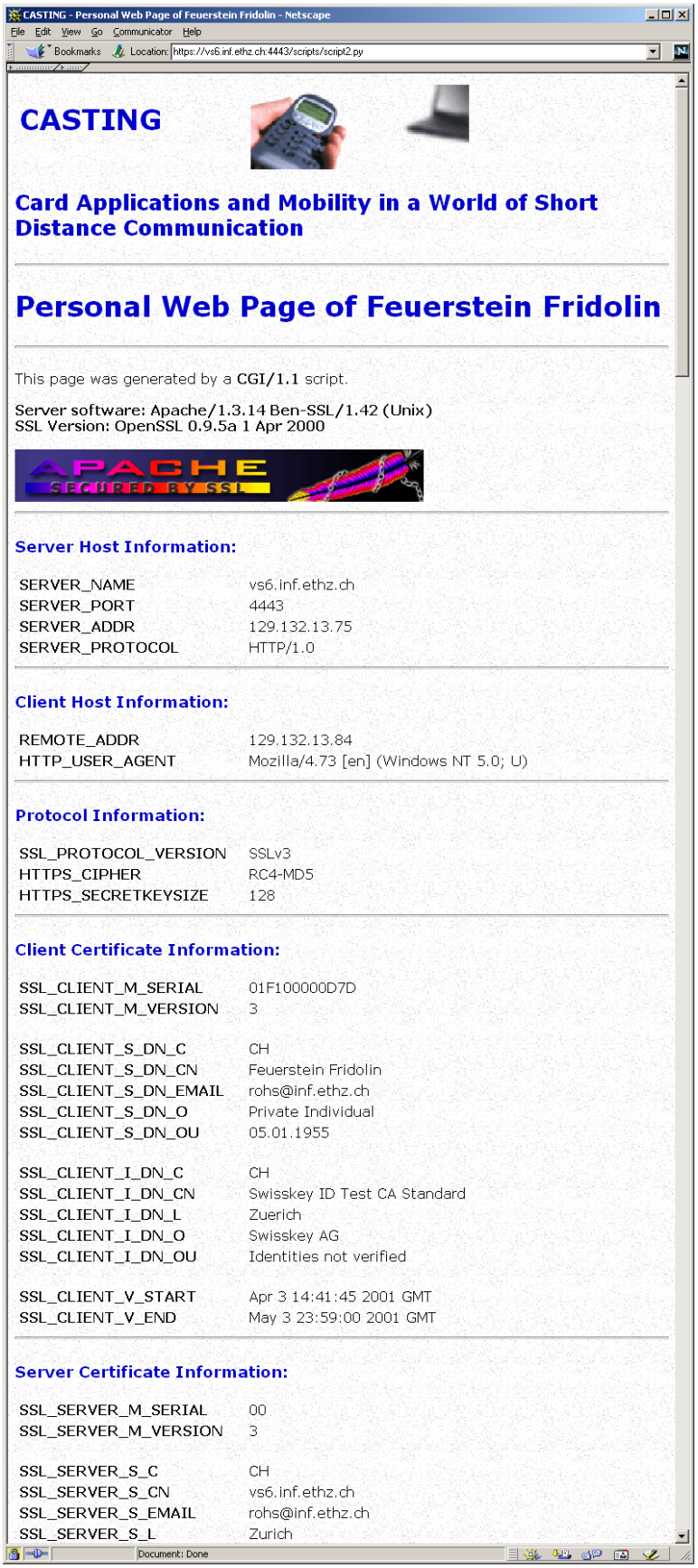

Figure 6.7: Personal Web page generated by script 2

Bibliography

[1] C. Adams and S. Farrell. Internet X.509 Public Key Infrastructure Certificate Management Protocols. Internet RFC 2510, Mar. 1999.

[2] C. Allen and T. Dierks. The TLS Protocol, Version 1.0. Internet RFC 2246, Jan. 1999.

[3] Apache HTTP Server Project. http://httpd.apache.org.

[4] Bluetooth Consortium. The Bluetooth Project Homepage. www.bluetooth.com.

[5] Bluetooth Special Interest Group. *Specification of the Bluetooth System – Volume 1: Core, v1.0 B*, Dec. 1999.

[6] Bluetooth Special Interest Group. *Specification of the Bluetooth System – Volume 2: Profiles, v1.0 B*, Dec. 1999.

[7] L. Buttyán. CASTING – Cryptographic Protocols. Research Report, Mar. 2000.

[8] European Telecommunications Standard Institute. *Digital cellular telecommunications system (Phase 2+); AT command set for GSM Mobile Equipment (ME) (GSM 07.07)*, 1997.

[9] European Telecommunications Standard Institute. *Digital cellular telecommunications system (Phase 2+); Specification of the SIM Application Toolkit for the Subscriber Identity Module – Mobile Equipment (SIM–ME) interface (GSM 11.14)*, 1998.

[10] European Telecommunications Standard Institute. *Digital cellular telecommunications system (Phase 2+); Specification of the Subscriber Identity Module – Mobile Equipment (SIM–ME) interface (GSM 11.11)*, 1998.

[11] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL Protocol, Version 3.0. home.netscape.com/eng/ssl3/draft302.txt, Nov. 1996.

[12] R. Housley, W. Ford, T. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. Internet RFC 2459, Jan. 1999.

[13] Infrared Data Association. *'IrCOMM': Serial and Parallel Port Emulation over IR (Wire Replacement), Version 1.0*, Nov. 1995.

[14] Infrared Data Association. *Link Management Protocol (LMP), Version 1.1*, Jan. 1996.

[15] Infrared Data Association. *Serial Infrared Link Access Protocol (IrLAP), Version 1.1*, June 1996.

[16] Infrared Data Association. *Tiny TP: A Flow-Control Mechanism for use with IrLMP, Version 1.1*, Oct. 1996.

[17] Infrared Data Association. *Serial Infrared Physical Layer Specification, Version 1.3*, Oct. 1998.

[18] Infrared Data Association. *Object Exchange Protocol (IrOBEX), Version 1.2*, Mar. 1999.

[19] Infrared Data Association. *Serial Infrared Link Access Protocol Specification for 16 Mb/s Addition (VFIR), Errata to IrLAP Version 1.1*, Jan. 1999.

[20] Infrared Data Association. *Serial Infrared Physical Layer Link Specification for 16 Mb/s Addition (VFIR), Errata To IrPHY Version 1.3*, Jan. 1999.
[21] Infrared data association. www.irda.org/, 1999.
[22] B. Kaliski and J. Staddon. PKCS #1: RSA Cryptography Specifications, Version 2.0. Internet RFC 2437, Oct. 1998.
[23] B. S. Kaliski, Jr. A Layman's Guide to a Subset of ASN.1, BER, and DER. ftp://ftp.rsasecurity.com/pub/pkcs/doc/layman.doc, Nov. 1993.
[24] B. S. Kaliski, Jr. An Overview of the PKCS Standards. ftp://ftp.rsasecurity.com/pub/pkcs/doc/overview.doc, Nov. 1993.
[25] B. S. Kaliski, Jr. Some Examples of the PKCS Standards. ftp://ftp.rsasecurity.com/pub/pkcs/doc/examples.doc, Nov. 1993.
[26] K. Kaukonen and R. Thayer. A Stream Cipher Encryption Algorithm "Arcfour". Internet Draft, www.cs-ipv6.lancs.ac.uk/ipv6/documents/standards/general-comms/internet-drafts/draft-kaukonen-cipher-arcfour-03.txt, July 1999.
[27] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. Internet RFC 2104, Feb. 1997.
[28] X. Lai. *On the design and security of block ciphers*, volume 1 of *ETH Series in Information Processing, Technische Hochschule Zurich*. Hartung-Gorre Verlag Konstanz, 1992.
[29] P. J. Megowan, D. W. Suvak, and C. D. Knutson. IrDA Infrared Communications: An Overview. www.extendedsystems.com/prodinfo/pdf/irda\%5Foverview.pdf.
[30] R. Mettala. Bluetooth Protocol Architecture, Version 1.0. www.bluetooth.com/developer/download/download.asp?doc=175, Sept. 1999.
[31] T. Muller. Bluetooth Security Architecture, Version 1.0. www.bluetooth.com/developer/download/download.asp?doc=174, July 1999.
[32] M. Myers, C. Adams, D. Solo, and D. Kemp. Internet X.509 Certificate Request Message Format. Internet RFC 2511, Mar. 1999.
[33] National Institute of Standards and Technology, Computer Systems Laboratory. *FIPS PUB 46-2: Data Encryption Standard (DES)*, federal information processing standards publication 46-2 edition, Dec. 1993.
[34] National Institute of Standards and Technology, Computer Systems Laboratory. *FIPS PUB 180-1: Secure Hash Standard*, federal information processing standards publication 180-1 edition, Apr. 1995.
[35] OpenSSL: The Open Source toolkit for SSL/TLS. www.openssl.org.
[36] GNU PKCS #11 (gpkcs11). Available at www.trustcenter.de/html/Produkte/TC_PKCS11/1490.htm.
[37] Public-Key Cryptography Standards. Available at www.rsasecurity.com/rsalabs/pkcs/.
[38] Entrust. www.entrust.com.
[39] NIST PKI Program. http://csrc.nist.gov/pki/.
[40] Swisskey Zertifizierungsstelle. www.swisskey.ch.
[41] Thawte Digital Certificate Services. www.thawte.com.
[42] VeriSign Root Certificates. www.verisign.com/repository/root.html.
[43] E. Rescorla. HTTP Over TLS. Internet RFC 2818, May 2000.
[44] R. L. Rivest. The MD5 Message-Digest Algorithm. Internet RFC 1321, Apr. 1992.
[45] R. L. Rivest, A. Shamir, and L. M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, Feb. 1978.
[46] M. Rohs and H. Vogt. Project CASTING – Task Description (CASTING Phase2, Part I). Draft, June 2000.
[47] RSA Security Inc. *PKCS #1 – RSA Cryptography Standard, Version 2.1*, Sept. 1999.

[48] RSA Security Inc. *PKCS #11 – Cryptographic Token Interface Standard, Version 2.10*, Dec. 1999.
[49] RSA Security Inc. *PKCS #12 – Personal Information Exchange Syntax, Version 1.0*, June 1999.
[50] S. Santesson, T. Polk, P. Barzin, and M. Nystrom. Internet X.509 Public Key Infrastructure Qualified Certificates Profile. Internet RFC 3039, Jan. 2001.
[51] Introduction to SSL. http://developer.netscape.com/docs/manuals/security/sslin/index.htm, 1998.
[52] Card applications and mobility in a world of short distance communication – CASTING. Project Definition, May 2000.
[53] I. VerisSign. Secure Wireless E-Commerce with PKI from VeriSign. Available at www.verisign.com, May 2000.
[54] J. Wandmacher. *Software-Ergonomie*. de Gruyter, 1993.
[55] E. Wiedmer. CASTING Project Monthly Report, June 2000.
[56] E. Wiedmer. Concept for CASTING Demonstrator: Authentication with Mobile. Report (Draft) for internal use, May 2000.