Chapter 7
Certificates and Public Key Cryptography

We concluded the previous chapter by describing the standard way of enabling HTTPS on an Apache web server. Our main goal was to protect the information exchanged between clients and the server from adversaries eavesdropping on communication. In this chapter, we will brush up on public key cryptography and examine its use in more detail.

7.1 Objectives

You will learn how to create public and private keys using OpenSSL. Given a key pair you will learn how to create a *certificate signing request* in the X.509 format. This can be sent to a certificate authority to request a certificate that binds the key pair to the name provided in the signing request. You will also learn to issue certificates yourself and to implement certificate-based client authentication on an Apache web server.

7.2 Fundamentals of Public Key Cryptography

Public key cryptography, also called asymmetric cryptography, is widely used both to encrypt and to digitally sign messages. In contrast to symmetric cryptography, where parties share the same secret key, public key cryptography uses a pair of keys consisting of a public and a private key. The public key is so named as it may be made public for all to see and use, whereas an agent keeps his private key to himself. The security of public key cryptography relies on the fact that there is no feasible way to derive the private key from the corresponding public key.

Figure 7.1 depicts the use of public key cryptography for encryption. Here a message m is encrypted by someone using Alice's public key. As Alice has the private key, she can decrypt the ciphertext c, recovering m. Moreover, provided she

D. Basin et al., *Applied Information Security*,
DOI 10.1007/978-3-642-24474-2_7,

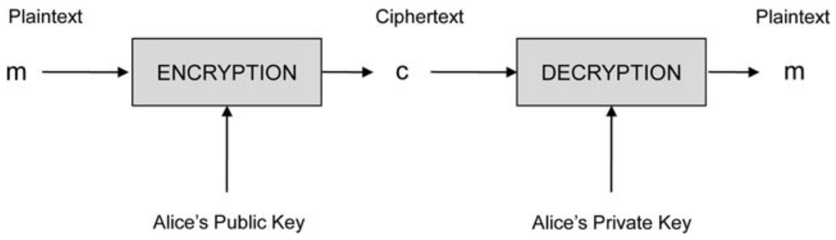

Fig. 7.1 Public key cryptography

is the sole holder of her private key — that is, this key has not been compromised by an adversary — then nobody else can decrypt *c*.

If we consider public key encryption as the counterpart to symmetric encryption, public key cryptography also enables a counterpart to message authentication codes, namely *digital signatures*. Similar to a public key encryption scheme, a signature scheme also requires two keys, which in this context are called the *signing key* and the *verification key*. For a signature scheme, we require that the signing key cannot be derived from the verification key and it is therefore safe to publicly disclose the verification key, enabling everybody to verify signatures. Creation of signatures, however, is restricted to the holder of the corresponding signing key. For some public key encryption schemes such as RSA it is possible to design secure signature schemes where private keys can be used as signing keys. Signatures can then be verified using the corresponding public key as the verification key. The use of public key cryptography in message signing is depicted in Fig. 7.2.

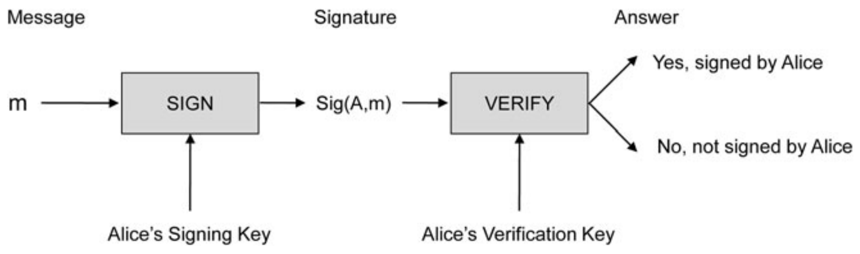

Fig. 7.2 Digital signatures

Problem 7.1 Briefly explain the components of an asymmetric encryption scheme and the required properties.

7.3 Distribution of Public Keys and Certificates

The use of public key encryption has several advantages over symmetric encryption. One advantage is that fewer keys need to be generated and distributed.

Example 7.1. Suppose we would like any pair of agents in a group of n agents to be able to exchange messages secretly. This requires a set of $\frac{(n-1)\times n}{2}$ symmetric keys, since every agent must share a key with every other agent. Public key cryptography reduces this overhead to n key pairs. Moreover, if an arbitrarily large group of agents wishes to encrypt messages for a server, the agents only need the server's public key.

A second advantage concerns *how* keys are distributed. For symmetric encryption, the keys must be exchanged secretly between each pair of agents. This is not required with public key cryptography: Alice can publish her public key on her web page or in any publicly accessible directory. However, there is an important pitfall that users of public key cryptography must be aware of: namely those wishing to communicate confidentially with Alice must get the *right* public key.

Example 7.2. Suppose that Bob wishes to encrypt a message for Alice. He therefore needs Alice's public key which he either receives from Alice upon request or which he downloads from Alice's web page. If Mallet controls the communication channel used by Bob to receive Alice's public key, Bob has no way to verify whether Alice's public key is *authentic*, i.e., if it originates from her.

What Bob receives is just a bit string, possibly manipulated by Mallet. Consider what happens when Mallet replaces Alice's public key with his own one. Bob then erroneously uses Mallet's key to encrypt a message for Alice and it is Mallet who can read Bob's message, not Alice.

Problem 7.2 In the scenario in Example 7.2, Alice sends her public key over an insecure communication channel to Bob. There is the possibility for an adversary to mount a man-in-the-middle attack.

1. Describe schematically how such a man-in-the-middle attack works. What messages are exchanged between Alice, Bob, and the adversary.
2. How could this kind of attack be prevented? What must Alice and Bob be aware of if they want to use public key cryptography as described in the scenario?

The key point is that Bob must get Alice's public key in a way that guarantees its authenticity. Contrast this with symmetric cryptography. There key distribution must ensure both the key's confidentiality and authenticity. If Alice and Bob want to share a symmetric key they can exchange it by meeting in person privately. Alternatively, they may use a preexisting secure channel, that is, a communication channel that ensures the authenticity and secrecy of its content.[1] In contrast, with public key

[1] This may in turn be guaranteed by some physical means, such as a shielded untappable cable between Alice and Bob or by using cryptography, which in turn requires predistributed keys.

cryptography, Alice can publish her key and need not ensure its secrecy. However, recipients must be able to rely on its authenticity. This may be formalized as the requirement that Alice communicates her public key to others over an authentic channel.

Problem 7.3 How can public key cryptography be used by two honest agents, say Alice and Bob, to create a shared secret in the presence of an adversary, if Alice and Bob did not share a secret beforehand?

Problem 7.4 Suppose Alice publishes her public key on her web page. How can Bob determine its authenticity?

In order to guarantee the authenticity of a public key, one often relies on *certificates* which are issued by *certificate authorities*. A certificate is an electronic document that binds a given public key to an identity and is thus used to verify that the public key belongs to an individual. This electronic document is digitally signed by the *certificate authority* (CA) that has issued the certificate. Other agents, who have the authority's signature verification key, can then verify the certificate's authenticity.

Summing up, a certificate is a signed assertion binding an identity to a public key. So why should we trust a certificate? If we look at this carefully, we see that two notions come into play. First, we must trust the certificate authority to correctly issue certificates, for example, to carefully verify the identity of the holder of the corresponding private key, protect his own signing key so that others cannot forge his signature, etc. Second, we must hold an authentic copy of the certificate authority's signature verification key to verify that the certificate has indeed been issued by the claimed authority. In this way we can check the authenticity of the asserted binding.

Given that you trust that a certificate authority correctly issues certificates, the next question is how do you obtain the certificate authority's (public) signature verification key?

Problem 7.5

1. Consider the case where you connect from your computer to your bank's web site, for example, to use online banking services. Typically, the bank's login page is secured using HTTPS, which uses certificate-based authentication. In view of what has been said in this chapter so far, it is clear that your client must have received the bank's certificate authentically beforehand. How is this done?
2. Assume that you have successfully established an HTTPS session to your bank's web site. What are the resulting security guarantees for both endpoints of the session, i.e., you and your bank?

7.4 Creating Keys and Certificates

To create keys and certificates, we will use OpenSSL [15]. This is an open-source toolkit that implements protocols such as SSL and provides a general-purpose cryptographic library.

▷ On **`alice`** create a private key with the command:

```
alice@alice:$ openssl genrsa -out alice.key 1024
```

The above command generates an RSA private key, i.e., a private exponent and two prime numbers that build the modulus. The key (more precisely the exponent and the two primes) is saved without protection in the file `alice.key`. If your private key is to be protected by a passphrase you can add the option `-des3`, which encrypts the key using triple DES. Note that the public exponent is set by default to 65537. To display all components of the private key file, use the command: `openssl rsa -text -in alice.key`.

Given a public–private key pair, we may want a certificate authority to bind the key pair to our identity. We therefore create a *signing request*. This includes our public key as well as information about the key holder's identity.

▷ Create a signing request for the key `alice.key` with the command:

```
alice@alice:$ openssl req -new -key alice.key -out alice.csr
```

After entering the command to create a signing request, you are asked questions, such as your name, address, e-mail address, etc. After the signing request `alice.csr` has been created, you can display its content using the command `openssl req -noout -text -in alice.csr`. You could now submit the certificate signing request (the file `alice.csr`) to a certificate authority.

Apart from having your key signed by a certificate authority, another option is to create a *self-signed certificate*, where you sign your certificate with your own key.

Problem 7.6 Self-signed certificates may at first seem a bit odd and merely a work-around in cases where you do not want to pay for a certificate issued by an "official" certificate authority. In particular, self-signed certificates do not seem to provide any kind of security guarantee. Why does it still make sense to allow self-signed certificates, even if you would only accept certificates signed by "official" certificate authorities?

The main standard for certificates is the X.509 standard, issued in 1988 by the International Telecommunication Union (ITU).

▷ After creating the certificate signing request `alice.csr`, we can create a self-signed certificate in the X.509 format with the command:

```
openssl x509 -req -days 365 -in alice.csr -signkey alice.key
-out alice.crt
```

Having created the certificate, you can now check its validity using the command `openssl verify alice.crt`. Note that file the file extensions `.pem` and `.crt` both indicate Base64 encoded PEM (Privacy Enhanced Mail) format of the certificate. In the following we use both extensions interchangeably.

In the following we configure Alice's HTTPS engine to use the certificate `alice.crt` and private key `alice.key`. We therefore assume that you have followed the configuration steps in Sect. 6.8, where we have enabled the SSL module on Alice's Apache web server.

To install the certificate and the corresponding private key, copy the certificate and the key into the right directory:

```
sudo cp alice.crt /etc/ssl/certs
sudo cp alice.key /etc/ssl/private
```

To enable the generated key and certificate to be used by the HTTPS engine of Alice's web site, change the lines

```
SSLCerticateFile /etc/ssl/certs/ssl-cert-snakeoil.pem
SSLCertificateKeyFile /etc/ssl/private/ssl-cert-snakeoil.pem
```

in the file `/etc/apache2/sites-enabled/default-ssl` to point to the corresponding locations and names of your files. In the case of the key and the self-signed certificate created above, the corresponding lines would be:

```
SSLCerticateFile /etc/ssl/certs/alice.crt
SSLCertificateKeyFile /etc/ssl/private/alice.key
```

In this way we have equipped the web server on **`alice`** with a key pair and a self-signed certificate. To activate the changes in **`alice`**'s HTTPS engine, we must restart the Apache daemon with the command `sudo /etc/init.d/apache2 restart`. If we now connect from **`mallet`** to **`alice`** using HTTPS, we receive a security warning saying that the certificate presented by the server is self-signed.

7.5 Running a Certificate Authority

In order to allow Alice to create key pairs and the corresponding certificates, she must set up a *certificate authority* (CA). To set up a CA on **`alice`**, proceed as follows.

▷ On **`alice`** execute the following steps:

1. Create the directories that hold the CA's certificate and related files:

   ```
   alice@alice:$ sudo mkdir /etc/ssl/CA
   alice@alice:$ sudo mkdir /etc/ssl/CA/certs
   alice@alice:$ sudo mkdir /etc/ssl/CA/newcerts
   alice@alice:$ sudo mkdir /etc/ssl/CA/private
   ```

2. The CA needs a file to keep track of the last serial number issued by the CA. For this purpose, create the file `serial` and enter the number 01 as the first serial number:

   ```
   alice@alice:$ sudo bash -c "echo '01' > /etc/ssl/CA/serial"
   ```

3. An additional file is needed to record certificates that have been issued:

   ```
   alice@alice:$ sudo touch /etc/ssl/CA/index.txt
   ```

4. The last file to be modified is the CA configuration file `/etc/ssl/openssl.cnf`. In the `[CA_default]` section of the file you should modify the directory entries according to the setting of your system. To do this, modify `dir` to point to `/etc/ssl/CA`.
5. At this point, create the self-signed root certificate with the command:

   ```
   sudo openssl req -new -x509 -extensions v3_ca -keyout
   cakey.pem -out cacert.pem -days 3650
   ```

6. Having successfully created the key and the certificate, install them into the correct directory:

   ```
   alice@alice:$ sudo mv cakey.pem /etc/ssl/CA/private/
   alice@alice:$ sudo mv cacert.pem /etc/ssl/CA/
   ```

7. Now we are ready to sign certificates. Given a certificate signing request (e.g., `key.csr`), the following command will generate a certificate signed by Alice's CA:

   ```
   sudo openssl ca -in key.csr -config /etc/ssl/openssl.cnf
   ```

   The certificate is then saved in `/etc/ssl/CA/newcerts/` as `<serial-number>.pem`.

Problem 7.7 Create a new key pair and use the CA to sign the public key. Explain the commands you use.

Aside from issuing certificates, certificate authorities also provide a service where certificates may be revoked. Revoking a certificate declares it to be invalid. If a public key for encryption is revoked, then no one should encrypt messages with it. If a signature verification key is revoked, then no one should accept messages signed with the corresponding signing key, even if signature verification would succeed.

Certificate revocation is often necessary whenever a private key is lost or compromised (e.g., stolen) by an adversary.

Example 7.3. Suppose that Alice has a private key used for decryption. If she loses it then she no longer wishes to receive messages encrypted with the corresponding public key as she cannot decrypt them. Even worse, if her private key is compromised, the adversary can decrypt messages sent to her. In both cases Alice should revoke the certificate associated with her public key. Now suppose that Alice's signing key is compromised. This would be disastrous as now the adversary can forge her signature. Again, Alice must revoke the associated certificate.

Problem 7.8 Suppose Alice loses her signing key. For example, it was stored on a token, which breaks. To what extent is this problem? Should she revoke the associated public key certificate?

To revoke a certificate, the person or entity identified in the certificate must contact the certificate authority. After successful authentication of the certificate holder, the certificate authority revokes the corresponding certificate and makes the revocation public on the *certificate revocation list* (CRL). Whenever a certificate is used, one must check on the corresponding CA's revocation list whether the certificate has not been revoked in the mean time. Certificate-handling APIs usually perform these checks automatically.

Problem 7.9

1. Key revocation plays an important role in the context of public key cryptography. Why is this not the case for symmetric keys?
2. How can you use asymmetric cryptography for communication so that even if your private key is ever compromised, all communication prior to the compromise still remains secret? (This property is called *perfect forward secrecy*.)

We use the following command to revoke a certificate `cert.crt` on host **`alice`**:

```
sudo openssl ca -revoke cert.crt -config /etc/ssl/openssl.cnf
```

The command updates the file `index.txt` that keeps track of the issued certificates. The corresponding certificate is therefore marked as revoked.

To create a certificate revocation list we first create the corresponding directory and serial number file:

```
sudo mkdir /etc/ssl/CA/crl
sudo bash -c "echo '01' > /etc/ssl/CA/crlnumber"
```

If necessary, the entry in the configuration file `/etc/ssl/openssl.cnf` pointing to the directory of the certificate revocation list must be changed accordingly. The following command takes the necessary information from the file `index.txt`, creates a certificate revocation list and places it in the designated directory. Note that the certificate revocation list must be recreated after the revocation of a certificate.

```
sudo openssl ca -gencrl -out /etc/ssl/CA/crl/crl.pem
```

At this point the certificate revocation list must be made available to the public to prevent misuse of the corresponding public and private keys.

To verify if a given certificate `cert.crt` has been revoked and published on a given certificate revocation list crl.pem we proceed as follows. First we concatenate the certificate of the responsible certificate authority cacert.pem with the revocation list crl.pem to build the file `revoked.pem`, then we use `openssl verify` with the `-crl_check` option to verify that `cert.crt` is indeed in the revocation list.

```
cat /etc/ssl/CA/cacert.pem /etc/ssl/CA/crl/crl.pem > revoked.pem
openssl verify -CAfile revoked.pem -crl_check cert.crt
```

7.6 Certificate-Based Client Authentication

In the last few sections, you have learned how to use `openssl` to generate key pairs, certificate signing requests, and how to operate a certificate authority. In this section, we use certificates to authenticate clients who want to connect to Alice's web site. The scenario described below is kept as simple as possible. For more complicated scenarios see the web site of modSSL [2] and the user manual you will find there.

The primary goal in this section is not only to authenticate the server using a server certificate, but also to authenticate the client using a client certificate that has been issued by the server. A typical application would be one in which all the clients are known to the server, and where an administrator is responsible for generating and distributing the certificates. Consider, for example, an internal network where the administrator could distribute the client certificates to all network machines.

In the last section, we created a key pair, calling the private key, for example, `testkey.key`, and the corresponding certificate `01.pem` (this corresponds to the case where `testkey.key` is the first key our CA has generated a certificate for). In order to use the certificate within a browser, we must create a corresponding certificate in the PKCS#12 format.

▷ Create the PKCS#12 certificate as follows:

1. Use:

   ```
   openssl pkcs12 -export -in 01.pem -inkey testkey.key
   -out testkey.p12 -name "MyCertificate"
   ```

2. You are asked to enter an *export password*, which you should carefully note since you will need it later to install the certificate on the remote host.

We must now install the PKCS#12 certificate in the browser that we will use to connect to the web site. In our example, we will install the certificate on **`mallet`**. Before doing so, we configure the web server on **`alice`** to accept connections that have been authenticated using a valid certificate issued by Alice's CA. Note that `cacert.pem` denotes the certificate of the signing key that has been used to sign the certificates.

▷ Add the following lines to the file `/etc/apache2/httpd.conf`:

```
SSLVerifyClient require
SSLVerifyDepth 1
SSLCACertificateFile /etc/ssl/CA/cacert.pem
```

Having added these lines to the Apache configuration file, restart the web server on **`alice`** with `sudo /etc/init.d/apache2 restart`. Having restarted the web server, try to connect from **`mallet`** to the HTTPS site https://alice using Firefox. Obviously, it is no longer possible to connect to the site. We must therefore install the certificate on **`mallet`**.

▷ Install the certificate on **`mallet`** as follows.

1. Copy the certificate file `testkey.p12` to a directory on **`mallet`**, for example, using the following command on **`mallet`**:

   ```
   mallet@mallet:$ scp alice@alice:/home/alice/testkey.p12 .
   ```

2. Now add the certificate to Firefox by importing the certificate using the Firefox option *Edit → Preferences → Advanced → Encryption → View Certificates (tab page "Your Certificates") → Import*.

At this point you should be able to connect to https://alice, possibly after restarting Firefox. You will then be asked to confirm certificate-based authentication requested by the web site using the corresponding certificate.

7.7 Exercises

Question 7.1 Briefly characterize the following building blocks of a public key infrastructure (PKI): *public key*, *private key* and *certificate*.

Question 7.2 A self-signed certificate is a certificate that was signed with the private key that corresponds to the certificate's public key. Do self-signed certificates make sense? Explain your answer.

Question 7.3

1. Suppose you have a public key and the associated certificate issued by a certificate authority (CA). What kind of guarantees should the certificate provide for you? How do you obtain the guarantee that the certificate really was issued by the respective CA?
2. Suppose that Alice wants to authenticate Bob using certificate-based authentication. Suppose further that Alice holds the CA's certificate and that Bob has a certificate for his public key issued by the CA. Explain the necessary steps of the authentication process.

Question 7.4 Suppose you are a new administrator of a small company which recently introduced a simple PKI to ensure secure e-mail communication for its employees. As a first step you analyze the IT infrastructure.

1. You recognize that the CA of the PKI synchronizes its clock daily with an unauthenticated time server. What security problems could arise from this? How could the problems be detected? What are possible countermeasures?
2. What possible security problems arise if an adversary manages to alter the clock used by the legitimate clients? What are appropriate countermeasures in this case?

Question 7.5 Figure 7.3 shows a simplified version of the *(simple) TLS handshake* between a client and a server, presented as a message sequence diagram. In the diagram we assume that:

- Cert(Pk,CertA) denotes a certificate issued by a certificate authority CertA, containing the public key Pk.
- The dashed arrows at the end of the protocols denote encrypted communication.
- The client and server *finished* messages at the end of the protocol include the name of the corresponding entity and a hash of all random strings in the protocol run.

1. Describe the guarantees that each of the involved parties (client and server) have after the execution of the protocol. For each of the guarantees argue (informally) why it should hold, i.e., what are the necessary prerequisites for the protocol to achieve the corresponding guarantee.
2. Give an example of an application scenario in which it makes sense to use this form of TLS handshake.
3. Assume that you want to connect to a web site using the TLS handshake as it is depicted in Fig. 7.3. When connecting to the web site, your browser interrupts the connection and shows a "Web Site Certified by an Unknown Authority" warning (in the case of Firefox). At which step of the protocol (in Fig. 7.3) does the

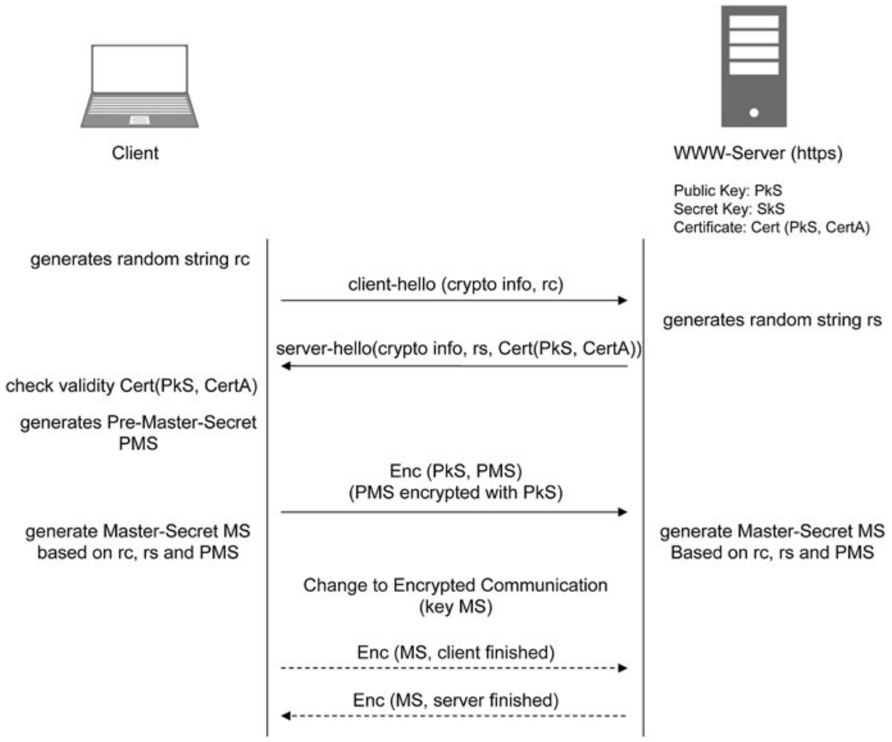
Client
WWW-Server (https)
Public Key: PkS
Secret Key: SkS
Certificate: Cert (PkS, CertA)
generates random string rc
client-hello (crypto info, rc)
generates random string rs
server-hello(crypto info, rs, Cert(PkS, CertA))
check validity Cert(PkS, CertA)
generates Pre-Master-Secret PMS
Enc (PkS, PMS)
(PMS encrypted with PkS)
generate Master-Secret MS based on rc, rs and PMS
generate Master-Secret MS Based on rc, rs and PMS
Change to Encrypted Communication (key MS)
Enc (MS, client finished)
Enc (MS, server finished)

Fig. 7.3 Simplified TLS handshake

problem originate? Could an adversary make use of this problem? If your answer is "Yes", explain how, and if your answer is "No", explain why.

Question 7.6 Figure 7.4 presents a simplified version of a second TLS variant.

1. What is the main difference in terms of security guarantees with respect to the previous version presented in Fig. 7.3?
   In November 2009, a vulnerability in TLS was discovered (TLS renegotiation vulnerability). In the following we describe schematically how an HTTPS server handles a client-request. We refer to the simple TLS handshake in Fig. 7.3 as TLS-A, and the second TLS handshake in Fig. 7.4 as TLS-B.
   Consider an HTTPS server that contains both public content and secured content that is only allowed to be read (GET request) or changed (POST request) by authenticated users. The request handling is schematically shown in Fig. 7.5 and works as as follows:

   a. Every connection is initially secured by executing a TLS-A handshake between the server and the client.

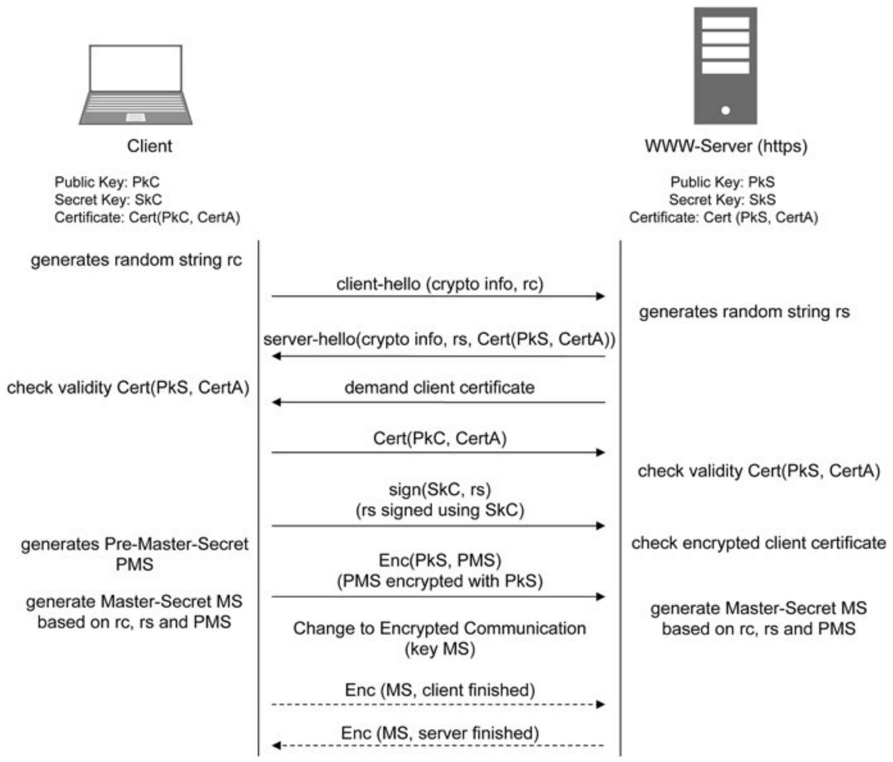
Client
Public Key: PkC
Secret Key: SkC
Certificate: Cert(PkC, CertA)
WWW-Server (https)
Public Key: PkS
Secret Key: SkS
Certificate: Cert (PkS, CertA)
generates random string rc
client-hello (crypto info, rc)
generates random string rs
server-hello(crypto info, rs, Cert(PkS, CertA))
check validity Cert(PkS, CertA)
demand client certificate
Cert(PkC, CertA)
check validity Cert(PkS, CertA)
sign(SkC, rs)
(rs signed using SkC)
generates Pre-Master-Secret PMS
check encrypted client certificate
Enc(PkS, PMS)
(PMS encrypted with PkS)
generate Master-Secret MS based on rc, rs and PMS
Change to Encrypted Communication (key MS)
generate Master-Secret MS based on rc, rs and PMS
Enc (MS, client finished)
Enc (MS, server finished)

Fig. 7.4 Simplified TLS handshake with client authentication

b. If the client requests access to a resource that requires client authentication, the server saves the request and initiates a so-called renegotiation by requesting the client to execute a TLS-B handshake.
c. If the TLS-B handshake (in Step 2) has been successfully executed, the server executes the saved request from the client.

2. Find a security problem in the way the server handles requests; describe informally how a potential adversary could use this weakness.
   Note that the browser does not distinguish between TLS-A and TLS-B sessions, and depending on the parameters received in the server-hello packet, it executes TLS-A or TLS-B.
3. Propose a fix to the problem you found in 2.

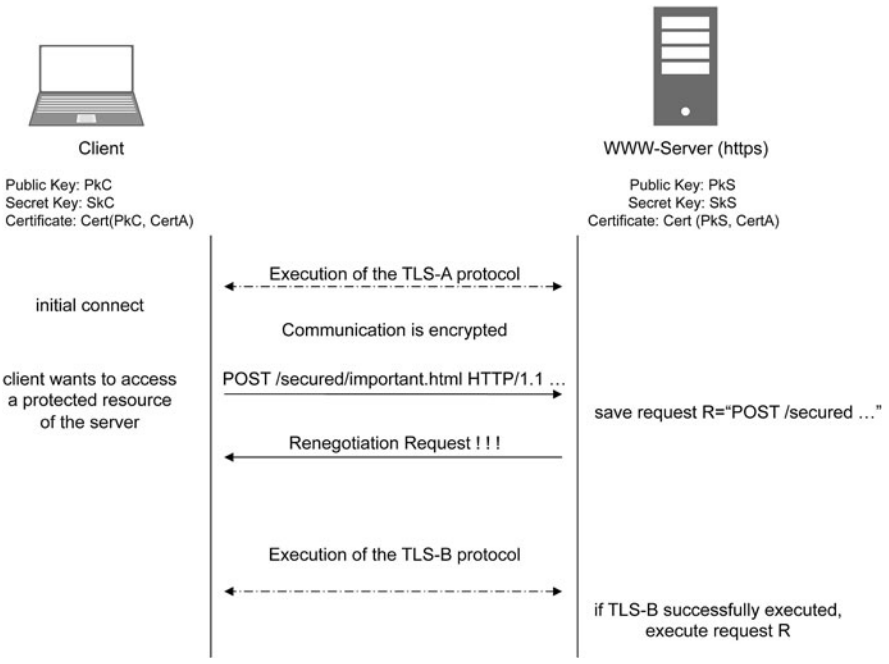
Client
Public Key: PkC
Secret Key: SkC
Certificate: Cert(PkC, CertA)
WWW-Server (https)
Public Key: PkS
Secret Key: SkS
Certificate: Cert (PkS, CertA)
Execution of the TLS-A protocol
initial connect
Communication is encrypted
client wants to access a protected resource of the server
POST /secured/important.html HTTP/1.1 …
save request R="POST /secured …"
Renegotiation Request ! ! !
Execution of the TLS-B protocol
if TLS-B successfully executed, execute request R

Fig. 7.5 TLS renegotiation