A Secure Electronic Tender System

Jonas Hallberg, Alf Bengtsson, and David Lindahl

Abstract

Using World-Wide Web technology to implement a secure electronic tender system offer possibilities considering both cost-effectiveness and security. In this report the structure and protocol of a system utilizing these possibilities are proposed. Moreover, a prototype implementation illustrating the proposed system is described. Security issues not handled by the prototype are described, which should be valuable when designing a final implementation.

1. Introduction

When something is put out for tender, the goal is to receive the best possible offer to supply the services or goods in question. To achieve this several steps have to be performed. First, a *tender invitation*, which captures the desired characteristics of the requested services or goods in a precise and unambiguous manner, has to be written. Second, the tender invitation has to be distributed in such a way that as many as possible of the parties potentially interested in responding will become aware of its existence. Third, the parties interested in delivering the requested services or goods, hereafter referred to as the *suppliers*, have to write *tenders* capturing their offers. Fourth, the tenders have to be delivered to the party requesting them, that is the *buyer*, preferably in a secure way. Here, security implies confidentiality, integrity, and non-repudiation. Fifth, the buyer has to decide whether any of the submitted tenders is fulfilling the requirements and, if this is the case, decide which of the suppliers to offer the contract to. The tender process is followed by the contract phase in which a contract is agreed upon and signed by the buyer and the supplier in order to close the deal.

In this report a project aiming at designing a secure electronic tender system is described. The report is submitted as a partial fulfillment of the examination criteria for the "Applied Network Security" course given at the Department of Computer and Information Science, Linköping University, spring 1999. In the project World-Wide Web (WWW) technology is utilized to support the execution of steps two and four of the tender process described above.

1.1 Motivation

The tender process described above is quite complicated, especially considering the security aspects during the second and fourth steps. A secure electronic tender system would, as the name suggests, alleviate the security problems and decrease the effort required to distribute and store all the tender invitations and tenders. Thus, enabling the application of the process to less significant purchases and decreasing the risk of mistakes, misunderstandings, and falsifications. All this leads to the mother of all motivations, that is, decreased costs.

1.2 Problem Formulation

Designing a secure electronic tender system demands careful consideration of the requirements implied by the terms secure and electronic. In this case, electronic means that WWW technology should be used in order to benefit from the connectivity and low distribution costs of the Internet. Thus, a framework for distribution of tender invitations and tenders via the Internet has to be created.

Being secure, on the other hand, requires that the confidentiality, integrity, and non-repudiation characteristics of the documents are properly protected. Confidentiality is usually not an issue for tender invitations. However, in some cases the mere fact that the buyer would like to purchase a specific service or goods is sensitive information. For tenders, on the other hand, confidentiality is essential from the perspective of both the buyer and the supplier. Integrity is obviously a central issue considering all involved documents and parties, since the contents of a received document have to be trustworthy. Furthermore, non-repudiation is required in order for a party to be able to depend on the offer made by another party. Thus, a protocol with security mechanisms guaranteeing these characteristics of the documents has to be designed.

1.3 Contributions

The main contributions of this work are:

- The design of a system structure and a protocol fulfilling the requirements specified in Section 1.2 above.
- A prototype implementation illustrating the use of the proposed protocol.
- A discussion of the strengths and weaknesses of the prototype implementation.

1.4 Report Layout

In Section 2 a background is given by discussing the WWW techniques necessary for the creation of a secure electronic tender system. In Section 3 the system structure and protocol is detailed and in Section 4 the prototype implementation is described. Section 5 contains an example illustrating the use of the prototype implementation. Finally, in Section 6 conclusions are drawn.

2. Background

We want to use the World-Wide Web, enhanced with security functionality, to build our Electronic Tender System. In this chapter we give a broad survey of the WWW and the security enhancements we use. For detailed descriptions we refer to (Garfinkel and Spafford 1997). The survey consists of four sections

- Digital signatures and PKI. Digital signatures are used to achieve authentication of tenders and tender invitations. They are based on a public-key crypto systems where the public keys are authenticated by a PKI. The keys are also indirectly used to generate transmission encryption keys to achieve confidentiality.
- WWW-servers, basic functionality. The server shall serve two types of requests from the clients. The clients want to read data (a static data file or some dynamically generated data) or the clients want to submit data (to be stored in a file or to be input to an application). The basic means to implement a security policy, i.e. control which client (or preferably which user) that is authorized to read or submit, are quite rude.
- WWW-clients, basic functionality of clients, mainly browsers.
- WWW-security. The de facto enhancements for better security in WWW-servers and clients.

2.1 Digital signatures and PKI

The digital signature is the tool to obtain authentication of a message. It guarantees both that the message is unchanged and that the message was put together by the signer and no one else. The digital

signature is based upon cryptographic operations. For details about different algorithms etc see (Menezes, van Oorschot, and Vanstone 1999). For a lot of reasons a scheme based on public key cryptography should be chosen. The private key is used by the signer. The corresponding public key is used to verify the signature.

As always in public key systems two conditions are absolutely crucial. The signer must never disclose her private signing key. The verifier must be sure that the public key that he uses to verify the signature is tied to the signer in an undeniable way. Under these two conditions we get the desired properties of the digital signature – the message, including the signature itself, is authentic and the signer cannot deny her signing the message. Unfortunately these two conditions don't hold very well in off-the-shelf WWW-systems.

The first condition, the confidentiality of the private signing key, is met by hiding the key in a file protected from unauthorized reading by the operating system and protected by some encryption. The access is most often controlled by a password mechanism. There is no established standard for this which means that the user must have one signing key per application. So you end up with many passwords and many keys accessible from outside the operating system, altogether a not very desirable situation. The need is obvious for a standardized system, preferably utilizing smart cards.

The second condition, tying together the signer's public key with her identity, is obtained by a Public Key Infrastructure, PKI. It consists of a hierarchy of Certification Authorities, CA's. The signer presents her public key to a CA that knows her and therefore can issue a certificate, a message that essentially says "I hereby guarantee that the public key xxxxx (an encoding of the public key) is tied to yyyyy (the identity)". The CA also uses its own private key to compute a digital signature of the certificate. The certificate and the CA's signature are handed over to the signer who can present them to the verifier. If the verifier trusts the CA and beforehand has got the CA's public key he can verify and trust the certificate and thus the signer's verification key. If not, the procedure is repeated recursively. The CA gets a certificate, that proves his own key and identity, from another CA one level higher in the hierarchy. Sooner or later the signer and verifier find a common CA they both trust and they can build a chain of certificates to prove the keys. Possibly this common CA is the highest CA in the hierarchy, the root-CA which everyone is supposed to trust. This whole system - CA's, how keys are handled, how certificates are encoded, when are certificates valid and what to do when they are not etc - constitutes the Public Key Infrastructure, see (Ashley and Vandenwauver 1999). In our project we used a small PKI set up by another project.

A particularly crucial, and hard to implement, part of the PKI is the handling of CRL, Certificate Revocation List. This is the list of certificates that for some reason have turned invalid before their expiration dates. This is a kind of black list that should be communicated to all parties in order to check the validity of certificates. We have not implemented CRL in our project.

In off-the-shelf WWW-systems you have the same problem with the PKI as described with the private key. Due to lack of common standards and API you get for instance one database with certificates for each application.

2.2 WWW-servers

The communication protocol mainly used on the World-Wide Web is HTTP, HyperText Transfer Protocol. In its simplest form it is used by a client to request data files, of different multimedia types, from a WWW-server. These requests should be served without the client logging in or otherwise authenticating itself. The data request either consists of a file name, which results in transmission of this particular data file, or it consists of a program name, which results in transmission of the output of the program. In the latter case the client could also send small amounts of data as input to the requested program.

The HTTP protocol is layered right above the TCP protocol. In the basic version, HTTP version 1.0, there is no persisting TCP-sessions. Each request requires its own session. So the basic HTTP 1.0 is quite simple. It mainly consists of three commands from the client - HEAD to get information, e.g. multimedia type, concerning the file, GET to also get the data file and POST to input data to a process in the server, e.g. to append data in a database. There are extensions in later versions, such as HTTP 1.1, but they are not implemented everywhere. Examples are PUT to send a data file from the client and methods to achieve some rudimentary authentication by sending a password unencrypted.

The security functions are almost non-existent in a WWW-server that only implements basic HTTP 1.0. Often the only restriction is that you can only request a file or a program that is stored in a specific branch of the file system in the server. Common extensions to somewhat improved security are for instance Access Control Lists. The ACL could be hardcoded in the configuration file of the server or it could be placed in a file, for instance named .htaccess, for each node in the branch. The authentication of the client are all the same imperfect, since it is often based on the IP-address of the client or on a password sent as clear text.

Similar deficiencies exist when a client POSTs input data to a server program. The program doesn't get authenticated information about which client is issuing the request. The program could be a specific application program with the POSTed data as input. It also could be a script that should be interpreted by a general script interpreter, for instance Perl. Some servers can interpret scripts themselves without running a separate script interpreter. Obviously these programs and scripts must be written with great care otherwise security will be compromised.

2.3 WWW-clients

The WWW-client uses the HTTP-protocol to request data from the server or to submit data to a program or script controlled by the server. The client could be any type of application. The most common application is the browser used to display the requested data. Embedded in the data is control information, written in the standardized HTML-language, that tells the browser how data should be displayed.

Due to the enormous popularity of the WWW, and the many types of multimedia information, some browsers have become huge programs trying to integrate a lot of peripheral functions, for instance electronic mail. But you can also find smaller browsers focusing on the core task, to display HTML files.

A simple browser can display a HTML-file. It can also display a form where the user can type data that the browser POSTs to the server. To extend this basic functionality there was a need for a scripting language that could interpret scripts embedded in the HTML-file. The de facto language is JavaScript which is implemented in most browsers. One example of use of JavaScript is to describe some calculations to do on data from a form before they are POSTed in order to detect typos. One problem with JavaScript is that it is not properly standardized.

The next level of extended browsers have implemented languages with more functionality than JavaScript. The most well known language is Java. Many browsers have hooks where you can add extensions called plug-ins. Such a plug-in could be an interpreter to yet another scripting language, e.g. Tcl/Tk or Perl, or it could be a plug-in for some type of multimedia, e.g. video. As an alternative to plug-ins the browser could launch a separate helper application to which it submits the data.

The security implications of these different extensions are equally serious as those on the server side and the need for authentication is evident. The functionality in scripts and plug-ins must furthermore be properly restricted. From JavaScript it should for instance not be possible to access the file system or the network at all. From Java you are granted access in a restricted way, within a 'sandbox'. From extensions built on Microsoft's ActiveX you have no restrictions other than authentication. Code that

comes from an authorized origin can do anything. For a discussion on different aspects of these matters, mobile code, see (Persson 1998).

2.4 WWW-security

In the previous sections we noticed that the WWW-security should be enhanced in many ways. This could be done at two levels.

At end-to-end level where the applications, in the clients and the server respectively, control encryption, authentication and access policy themselves. To achieve this the necessary functions - encryption, digital signing, key handling etc. - should be callable via a standardized API usable by all applications. As already mentioned, many of these functions ought to reside in a carefully protected environment, for instance a smart card. This is not the case in today's systems. There is for instance no ubiquitous PKI-system. So existing enhancements are restricted. In our project we use Netscape's PKI-implementation and a JavaScript function specific to Netscape.

The enhancements could also be at the transport layer. There have been proposals for extensions to the HTTP protocol but the de facto extension is SSL, Secure Socket Layer, in an intermediate layer just above TCP layer which means that it can be used by other protocols besides HTTP. The standardized version of SSL is named TLS.

The SSL-protocol consists of two parts, one handshake part and one transmission part. The handshake part is the opening of a persistent session within which the transmission can be split in many TCP-connections. The handshake has two results, one mandatory and one optional. The mandatory result is that the client and the server, using a negotiated method, build up a mutual master secret. The optional result is that the client or the server, or both of them, authenticates itself. To achieve this it sends a list of certificates, up to the root certificate, that the other party can verify. The authentication is at the SSL-level only. There is for instance no standardized way to hand over the certificates to the application.

In the transmission part of SSL the two parties each computes a key based on the master secret, and negotiates an encryption method, to be used for transmission encryption. This can be done several times within a session. This means that you achieve confidentiality at the transmission level, but data is decrypted before it is handed on to the application.

In SSL-capable WWW-browsers the convention is that if the address of the link starts with https:// instead of http:// then SSL should be used. The most common use of SSL, for instance in systems for shopping on the Internet, is to only use transmission encryption. In this case you get confidentiality on the Internet but you don't necessarily get authentication, although most WWW-servers send a list of certificates that makes server authentication possible.

3. System Structure and Communication Protocol

In order to construct the secure tender system a protocol will have to be specified. The protocol is the definition of how the communication between the parties will be structured in order for the system to work. As indicated in previous chapters the protocol set up must handle at least two problems:

- Authenticity of the offers placed on the server by the buyer.
- Confidentiality and authenticity of the tenders placed on the server by the provider.

This is accomplished through the use of encryption and digital signatures as detailed below.

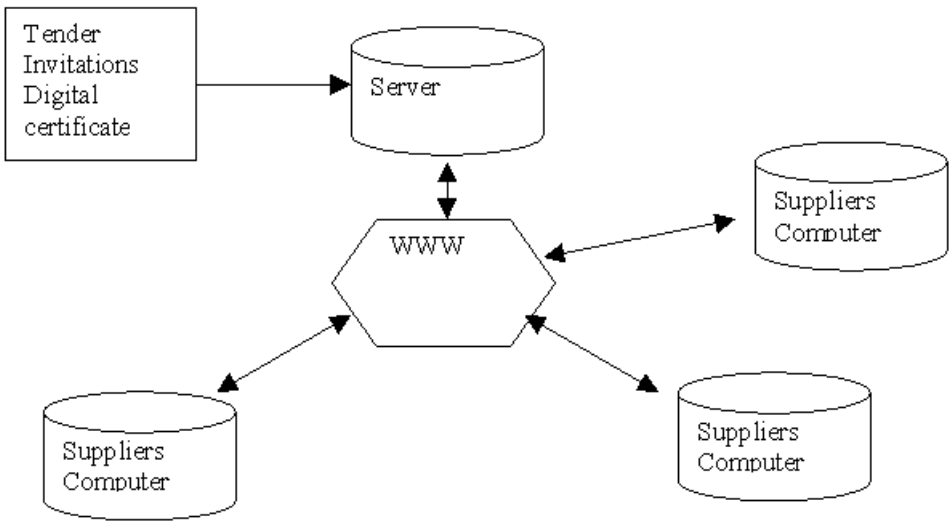
3.1 System structure

The entire system will consist of a web server on which the invitations and tenders will be located, the world wide web and at least one other computer. The owner of the system, the buyer, will be putting the tender invitations directly on the server. The prospective supplier will connect his computer with the server through the Internet.

The encryption is handled through the use of the SSL protocol in the contacts between the computers. This is automatic when the supplier is using one of the reasonably modern browsers and hence is a very practical solution as it is transparent to the user and requires no knowledge of encryption techniques on their part.

Since this feature was used as-is we do not take credit for it and will not discuss it further in this report.

To achieve authenticity a PKI is used. This requires a trusted third party (TTP) who will issue a individual certificate for the user. This certificate contains the users public key, information about the user and a hash sum of these encrypted with the TTP's private key. This certificate is then sent together with the digital signature the user creates from the message with his private key. The signatures are made with Netscape built-in features and require no knowledge of programming on the part of the users. It is however not automatic and is therefore included in the protocol description below.



3.2 Encryption

The encryption utilized in the protocol will be handled by using SSL for all the transactions. The guarantee of the correctness of the cryptographic implementation will therefore rest on the providers of the browser and web server.

3.3 Protocol Definitions

B will denote the buyer.
P will denote the provider.
W will denote the web server.
CA will denote the Trusted Third Party that has issued the certificate used in the protocol.

S will denote a digital signature
S_b will denote the digital signature of the buyer.
S_p will denote the digital signature of the provider.
$S_x(Y)$ will denote a digital signature of X signing Y.

C will denote a certificate containing individual information about the owner, the owner's public key and these two signed with the digital signature of the CA.

C_{CA} will denote the certificate of the CA.
C_B will denote the certificate of the buyer.
C_P will denote the certificate of the provider.

3.4 Protocol Structure (encryption excluded)

The buyer will write a tender invitation, sign it with his private key and add his digital certificate (acquired beforehand from the CA). He will then place the tender invitation and his digital signature on the web.

An interested supplier will download the tender invitation and the signature to her own machine. She will then verify that it is indeed the buyer who has signed the invitation. To accomplish this she will need a copy of the CA's certificate and a verification program.

The CA's certificate is probably available through the Internet if not by other means. It is possible that the buyer will have a chain of certificates each verified by the former up until the certificate issued by the CA. It will then be necessary for the supplier to decide whether or not she trusts EVERYONE in the chain

After the supplier has verified the identity of the buyer and the validity of the invitation she goes online and accesses the web server of the buyer. There she either writes or cut-and-paste a tender. She then signs it with her own private key, adds her certificate and sends it to the server where it is stored.

The buyer accesses the server and performs a similar procedure of verification. If he feels so inclined he will then accept the offer and send a formal agreement but this is not included in the protocol. To function sociologically it will probably be necessary to include an automatic reply to make sure that the supplier knows that her offer has reached its destination correctly.

Below is the protocol in formal notation:

B=>W : C_B, S_b(Tender Invitation), Tender Invitation
P=> W : Request Tender Invitations
W=>P: C_B, S_b(Tender Invitation), Tender Invitation

(If the CA's certificate is not in the supplier database it must be acquired, possibly as follows:

P => CA: Request C_{CA}
CA=>P: C_{CA})

P verifies that the Tender Invitation is authentic.

P =>W: C_P, S_p(Tender), Tender
B =>W: Request Tenders
W=>B: C_P, S_p(Tender), Tender
B =>CA: Requests C_{CA}
CA=>B: C_{CA}

B verifies that the Tender is authentic and that it corresponds to a valid Tender Invitation.

4. Implementation

To illustrate the use of the protocol described in the previous section a prototype has been implemented. To decrease the amount of code to be written, the prototype utilizes already available tools to perform the main functions of document signing and signature verification. The structure of the implementation is depicted in Figure 1. On the right side is the WWW-server, which performs three major tasks.

1. It delivers the HTML and JavaScript code as requested by the client browsers.
2. It accepts the digitally signed documents posted by the client browsers and executes Perl scripts in order to extract the document text and the signature data. The document text and signature data are stored in files with unique names created using the name of the tender invitation and the current time and date according to the server clock.
3. It delivers the stored tender invitation document and signature files as requested by the client browsers.

Note that the WWW-server, as illustrated by Figure 1, is controlled by the buyer.

In the middle of Figure 1 are the client browsers interfacing with the WWW-server and the users. The client browsers implement user interfaces enabling the buyer and the suppliers to input their tender invitations and tenders respectively. Digital signatures for the documents are created, using a JavaScript function, before the documents are posted to the WWW-server. Finally, the browsers allow the suppliers to retrieve the tender invitation document and signature files from the WWW-server and store them locally.

On the left side of Figure 1 are instances of the signature verification tool. This tool can be used to verify the signatures created by the JavaScript function and to output data contained in the signature file, such as the chain of certificates required in order to verify the certificate used to create the signature.

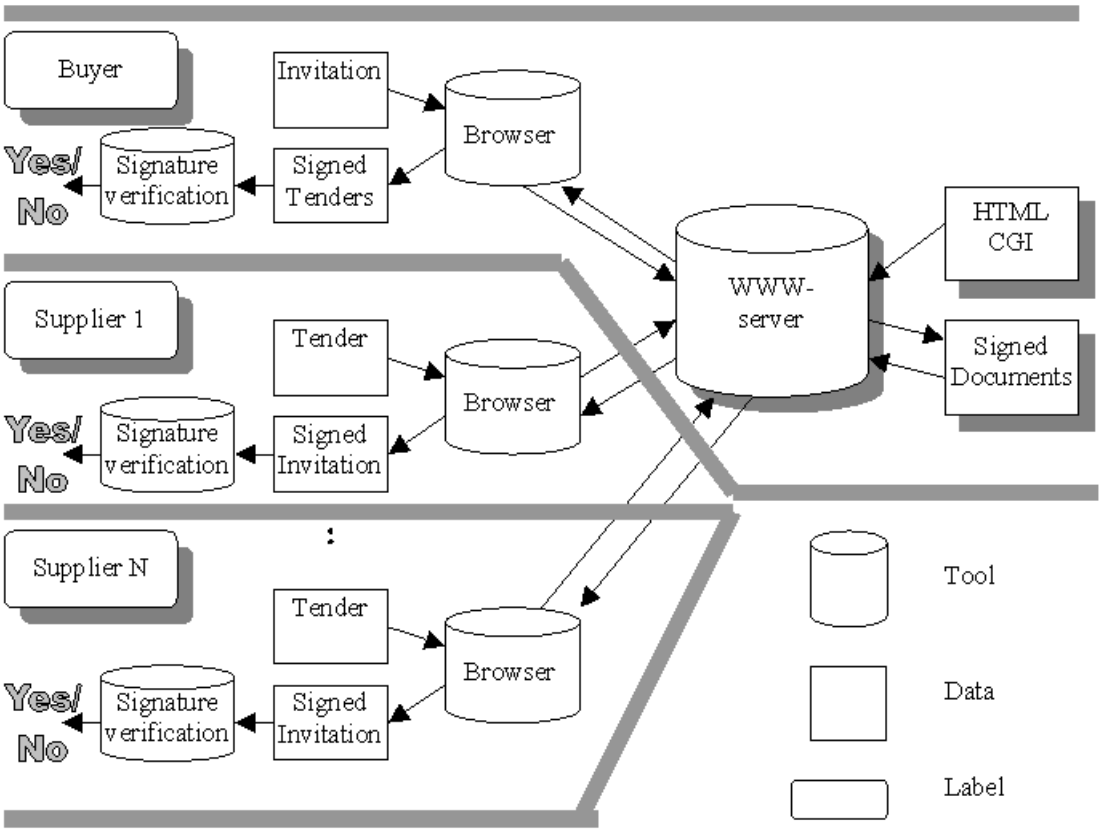


Figure 1: Structure of the implementation.

4.1 Tools Used

4.1.1 WWW-Server

In order to fulfill the confidentiality requirements the WWW-server has to support SSL. Therefore, we used an SSL-capable version of Apache, called Apache-SSL. The functionality of this server package will not be discussed here, but some configuration issues will appear in Section 4.2 below, where the file structure is discussed.

4.1.2 Client Browser

Since the secure electronic tender system is a WWW-based application, it should, in principle, be possible to use any WWW-browser. However, the JavaScript method used to create the digital signatures has to be available, which is the case for Netscape Communicator 4.04 and later versions. Thus, the use of the latest Netscape browser, or at least version 4.04, is prescribed.

The JavaScript method *crypto.signText* can be used to digitally sign strings of text (Netscape 1998a). The method takes the string to be signed and one or more arguments concerning the certificate to be used as input. The output consists of a string containing the digital signature and various other data. The syntax is:

```
crypto.signText(string, CA_option, [CA_name1, [CA_name2, ...]])
```

Where,

- *string* is the text to be signed,
- *CA_option* takes one of the values "ask" or "auto". The certificate used for the signing is taken from the certificate database managed by Netscape Communicator. The value of *CA_option* decides whether the selection of the certificate will be done manually, by the user, or automatically, by the system.
- *CA_namei* specifies the distinguished name of a certification authority, CA, whose certificates are accepted for signing the text. In this way, the number of certificates possible to select for the user or the system (depending on the value of *CA_option*) can be limited. Any number of distinguished names can be supplied. On the other hand, if none is given, no restrictions are made on the possible selections.

Figure 2 illustrates the certificate selection process. In this case, the user is asked to select the certificate to be used when signing the text.

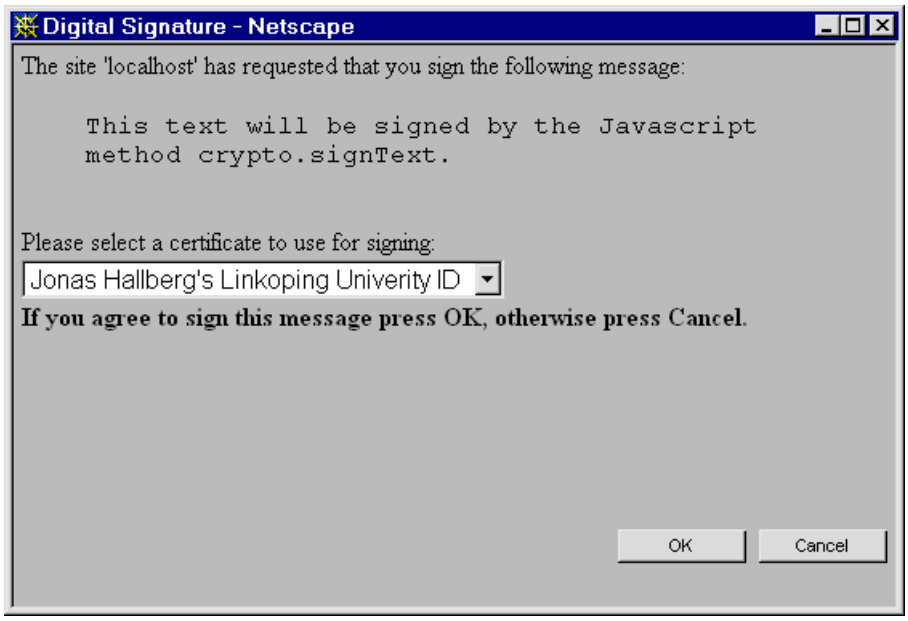


Figure 2: Certificate selection.

When the certificate has been selected *crypto.signText* uses the secure hash algorithm, SHA-1, to get a digest of the input text and then applies RSA to create the signature from this digest and the current time. If no error has occurred during the execution, the method returns a signed base-64 encoded PKCS (Public-Key Cryptography Standards) #7 object (RSA 1993), including:

- A list of certificates, including the certificate used to produce the signature and other certificates, if needed, to complete the chain up to a trusted CA.
- Signing time, the date and time when the signature was created. This attribute is authenticated together with the digest.
- Message digest, the digest of the text to be signed, produced by the selected hash algorithm, in this case SHA-1.
- Encrypted digest, the result of encrypting the message digest, and other authenticated attributes, with the private key corresponding to the selected user certificate.

Attributes that are named in PKCS #7, but not included in the returned object are:

- The contents, that is, the actual text that has been signed.
- Certificate-revocation lists, CRLs, this attribute is supposed to contain enough information to verify that none of the certificates has been revoked by their issuer.

The lack of support for CRLs is a severe shortcoming in the current implementation of the *crypto.signText* method. In fact, it will render the prototype of the Electronic Tender System incapable of detecting certificates that have been withdrawn before their expiry date.

If the *crypto.signText* method, for some reason, cannot complete the creation of the signature, then one of the following three error-messages replaces the PKCS #7 object as the output:

- *error:noMatchingCert*, no certificate fulfilling the restrictions enforced by the *CA_name* parameters was found. That is, the user does not have a certificate issued by any of the named CAs.
- *error:userCancel*, the signing process was interrupted by the user selecting the cancel button in the certificate-selection dialog box, shown in Figure 2.
- *error:internalError*, the process could not be completed for some other reason than no matching certificate or user cancel.

Actually, as indicated by the description above, the occurrence of the *error:internalError* message is not necessarily triggered by an error internal to the method. For example, the lack of a self-signed certificate, in the certificate database, terminating the chain of certificates to be included in the PKCS #7 object, results in this error message being returned. This is, of course, somewhat misleading.

4.1.3 Signature Verification Tool

To verify the signatures of the signed documents the signature verification tool from Netscape is used (Netscape 1998b). The signature verification tool is a stand-alone application, with a command line-based user interface, available for the Solaris and Windows NT platforms. The syntax is:

```
Signver [-i sign|data] [-d filename] [-s filename] -D directory_name –v|-V|-A
```

Where,

- *-i sign|data*, assigns standard input as the source of the PKCS #7 object or the signed data,
- *-d filename*, assigns file referred to by *filename* as the source of the signed data,
- *-s filename*, assigns file referred to by *filename* as the source of the PKCS #7 object,
- *-D directory_name*, assigns the directory referred to by *directory_name* as the holder of the certificate database,
- *-v*, returns the result of the signature verification,
- *-V*, returns the result of the signature verification and, if the signature is not valid, the reason why the signature is invalid,
- *-A*, returns the contents of the PKCS #7 object.

There are also other options (Netscape 1998b). However, they are neither fully implemented nor required by the electronic tender system and, thus, not included here.

The output depends on which of the options *–v*, *–V*, or *–A* that was specified. If *–v* was selected, the output is "*signatureValid=yes*" or "*signatureValid=no*" depending on whether the signature was approved or not. If *–V* was selected and the signature is rejected, the output is "*signatureValid=no:reason*", where *reason* can be, for example, "*Bad PKCS7 signature*". When *-A* is specified, a pretty print of the PKCS #7 object is returned as output.

To verify the signature of a tender, where the tender and the PKCS #7 object are stored in the files *tender* and *tender.sig* respectively and the certificate database is in the current directory, the following is the most straightforward command-line input.

```
signver –D . –d tender –s tender.sig –v
```

To increase the user-friendliness of the signature verification tool, we have implemented a graphical user interface in Tcl/Tk. The interface, depicted in Figure 3, allows the user to set the *-d*, *-s*, and *-D* parameters using standard file selection dialogues. The desired output is simply produced by selecting the appropriate button, *Verify* or *All Info*, and is presented in pop-up windows. The result of selecting *All Info* is depicted in Figure 4.

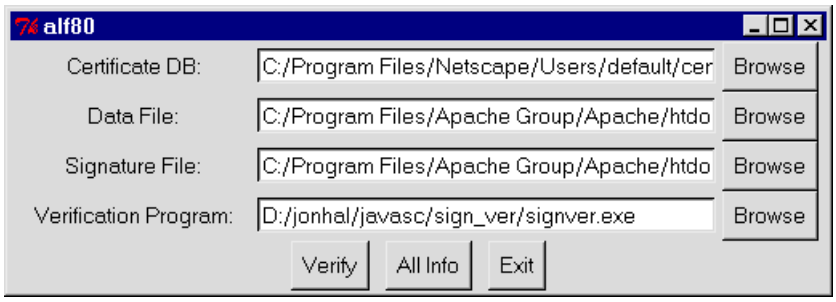


Figure 3: The user interface designed for the signature verification tool.

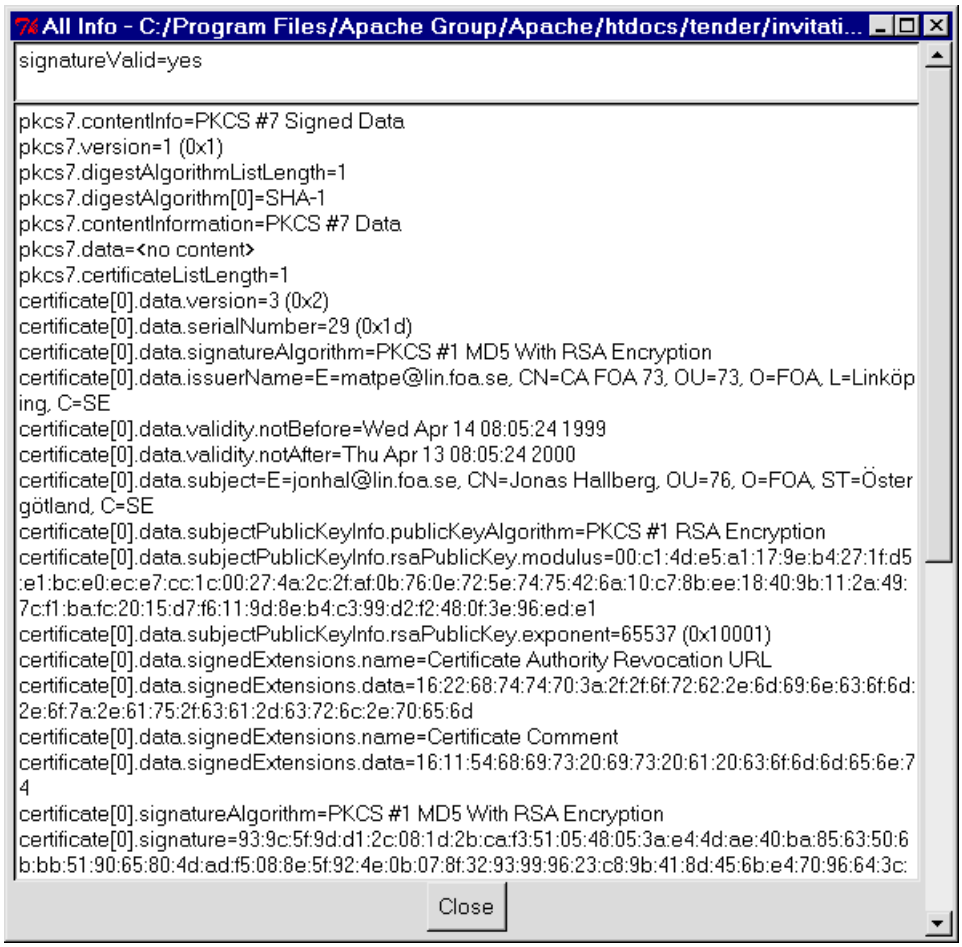


Figure 4: Pretty print of the PKCS #7 object displayed by the designed user interface.

4.2 File Structure

In this section the structure used to store the tender invitation, tender, signature, and code files is introduced and related security issues are discussed. The basic structure is depicted in Figure 5. The *tender* directory encapsulates all the files belonging to the electronic tender system, except for the CGI-scripts, which are contained in *cgi-bin*. The subdirectory *invitations* contains all the posted tender

invitations, while *tenders* contains all the submitted tenders.

The files *postinvitation.htm*, *submittender.htm*, and *tenders/index.htm* (paths relative to the *tender* directory) supplies the functionality for posting tender invitations, submitting tenders, and retrieving tender invitations respectively.

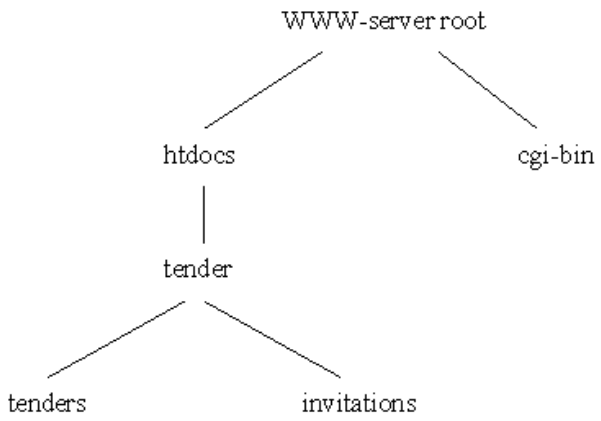


Figure 5: The file structure.

An issue of major importance is the protection against non-authorized access of the files. A first, and strong, line of defense is provided by SSL. Since only authenticated users are allowed access, those causing problems should be easy to trace. However, the rights to post tender invitations and read tenders have to be much more restricted, since only the buyer should be allowed to perform these tasks. For this purpose, the Apache-SSL directive *SSLFakeBasicAuth* is used (Laurie and Laurie 1999). Thus, the basic authentication process of Apache is simulated. The user names are extracted from the certificates using the SSLeay function *X509_NAME_oneline*, which, in principle, returns the distinguished name of the certificate holder. The passwords, on the other hand, are statically set to *password*. In this way, all the users will be authenticated, through their certificates, and associated with a unique user name. This is more effective and secure than the normal user name and password authentication procedure. To activate the *SSLFakeBasicAuth* authentication process the following two lines have to be inserted in the configuration file.

SSLFakeBasicAuth
SSLVerifyClient 2

SSLVerifyClient 2 ensures that the user have to provide a valid certificate.

The authorization is handled by dividing the users into the two groups of buyers and suppliers. The buyers are identified by putting their user names in the *user file* associated with the *SSLFakeBasicAuth* authentication process. In this way, the access rights that have to be granted exclusively to the buyer can be so. Thus, for the *invitations*-directory all access methods, except GET, are limited to the users included in the file *buyername* by the following lines.

<Directory "WWW-server root/htdocs/tender/invitations/">
AllowOverride none
AuthUserFile buyername
AuthType Basic
AuthName ElectronicTenderSystem
<LimitExcept GET>
Require valid-user

</LimitExcept>
</Directory>

For the tenders-directory, on the other hand, no exceptions are required and all access is limited to the users personifying the buyer with the following lines.

<Directory "WWW-server root/htdocs/tender/tenders/">
AllowOverride none
AuthUserFile buyername
AuthType Basic
AuthName ElectronicTenderSystem
Require valid-user
</Directory>

Accordingly, those users can view the tenders, and the corresponding signature files, using their browsers. The suppliers can not access the *tenders*-directory at all. This is not needed since the tenders are stored by a CGI-script.

Moreover, to prohibit non-authorized users from posting tender invitations, the following lines are added to the configuration file.

<Directory "WWW-server root/cgi-bin/">
<Files postinvitation.cgi>
AllowOverride none
AuthUserFile buyername
AuthType Basic
AuthName ElectronicTenderSystem
Require valid-user
</Files>
</Directory>

The file *buyername* should contain the user name of all the buyers. The format is:

User name:encrypted password

The simplest way to create this file is to take the user names from the error log file after an attempt by the corresponding user to access a restricted file or directory. The encrypted password is always set to *xxj31ZMTZzkVA.*

4.3 Glue

Obviously, some infrastructure has to support the JavaScript *crypto.signText* method and the signature verification tool in order to realize the electronic tender system. First of all, HTML with integrated JavaScript code is used to build the required user interfaces. Secondly, CGI-scripts, implemented in Perl, process the data posted to the WWW-server.

The HTML and JavaScript code implements forms that enable the buyer and the suppliers to input the tender invitations and tenders, respectively. Figure 6 depicts the form used for submission of tender invitations, while Figure 7 depicts the form used for submission of tenders. When a tender invitation is constructed it has to be assigned a unique name consisting of only alphanumeric characters. Ideally this name would result in a general, if not complete, understanding of the contents of the tender invitation. Correspondingly, when a tender is to be submitted, the name of the invitation responding to has to be included. These names are supposed to be stated in the text (line) boxes at the top of the respective forms. In the text areas below these text boxes the tenders and tender invitations are to be inserted.

Finally, to post the inserted data to the WWW-server the *Sign and Submit* button at the bottom of the form is selected. Before the tender invitation or tender is posted to the WWW-server, the corresponding PKCS #7 object is created using the JavaScript method *crypto.signText*. The fractions of code implementing these forms are included in Appendix A.

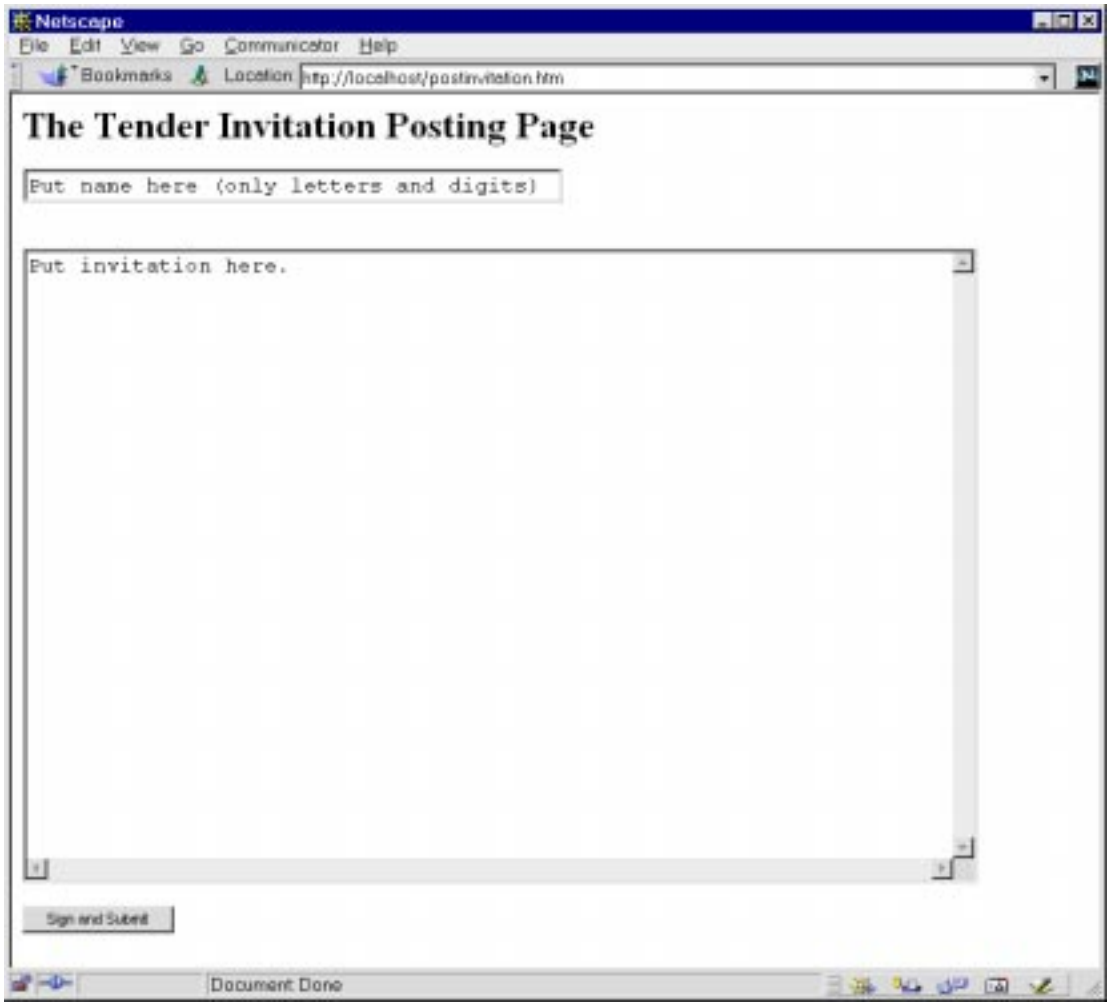


Figure 6: The tender invitation posting page.

Figure 7: The tender submission page.

When the WWW-server receives the posted data, it is processed by a Perl script. There are two Perl scripts, the code of which are included in Appendix A, corresponding to the two HTML-forms. The Perl scripts have two main tasks in common. First, the appropriate data, that is, the name of the tender, the signed text, and the PKCS #7 object, have to be extracted from the posting. Second, unique filenames, based on the name of the tender, have to be constructed for the files storing the signed text and the PKCS #7 object. Moreover, the tender invitation posting script builds an HTML file with links to all the stored tender invitations and the corresponding signature files.

The first task is solved similarly in the two scripts by parsing the data posted by the HTML-forms, which is read as a string from standard input. Since the data is URL-encoded, decoding has to be performed in order to restore the original form of the data. This is done by replacing all the plus signs with spaces and the hexadecimal numbers, *%x* where *x* is a two-digit hexadecimal number, with the corresponding characters. Obviously, it is of great importance that the data stored in the signed-text file is an exact copy of the text actually signed by *crypto.signText*. Here, some subtle *carriage return* '\r' and *linefeed* '\n' character problems arise because of:

1. The fact that some operating systems, such as Windows NT, make a distinction between text files and binary files and some, such as Linux and Solaris, do not. In those operating systems making the distinction a line in a text file ends with a carriage return followed by a linefeed, while in those system ignoring the difference the lines end with just a linefeed.
2. The HTTP-standard, which prescribes insertion of carriage returns in posted text.
3. The way Perl handles text files in an operating system doing the distinction, which means stripping away the carriage returns when a text file is read and inserting carriage returns when a text file is written.

4. The lack of carriage returns in the data the Perl scripts read from standard input. Actually, they are there, but encoded as *%0D*.

The experienced problems, which rendered the signature invalid, were:

1. When the system was tested in a Windows NT environment, the stored text files contained one extra carriage return per line. This was because of the Perl implementation of the print function, which adds an extra carriage return to each line, since it is assumed to have been stripped away when the file was read. This problem was simply solved by, contra-intuitively, storing the file as a binary file.
2. When the system was tested in a Unix environment, the carriage returns still remained in the signed text, because of the HTTP-standard. This, as above, resulted in the signed text files containing an extra carriage return on each line. This was solved by removing the solution introduced to deal with problem one and, instead, stripping away the carriage returns during the decoding of the data read from standard input.
3. When the client browser is executed under Windows NT and the WWW-server under Unix, the carriage-return problems remain and have not been solved for this prototype implementation.

One way to escape from this problem is to use a format that is more platform independent than plain text files, such as PDF, for the documents.

The second common task of the scripts, that is, to create unique filenames for the data files, is solved differently in the two scripts. When a tender invitation is posted, the name has to be supplied by the user. If the name is not unique, the posting will be refused and a new name has to be supplied. Given an exclusive name, the tender invitation text is stored in a file with that name and the signature is stored in a file with the same name and the extension "sig". The script dealing with tender submissions verifies that there is an invitation with the name supplied by the user submitting the tender. If this is not the case, the user will be asked to insert a valid name. When a valid invitation name has been supplied, unique names are constructed for the tender and the signature files by appending the current date and time to the name of the invitation. If, although most unlikely, the name is already taken, the script waits for two seconds before trying again.

As a third task, the script dealing with the tender invitations automatically generates an HTML file referring to all the tender invitations and the corresponding signature files. In this way, the suppliers can download and validate the tender invitations they find interesting. The HTML file is built by finding the names of all files, in the tender invitation directory, for which there is a corresponding file with the appropriate signature extension. All these files are assumed to contain invitation tenders and an HTML-file linking to them and the accompanying signature files is created. Figure 8 depicts the result of displaying one of these automatically generated files.

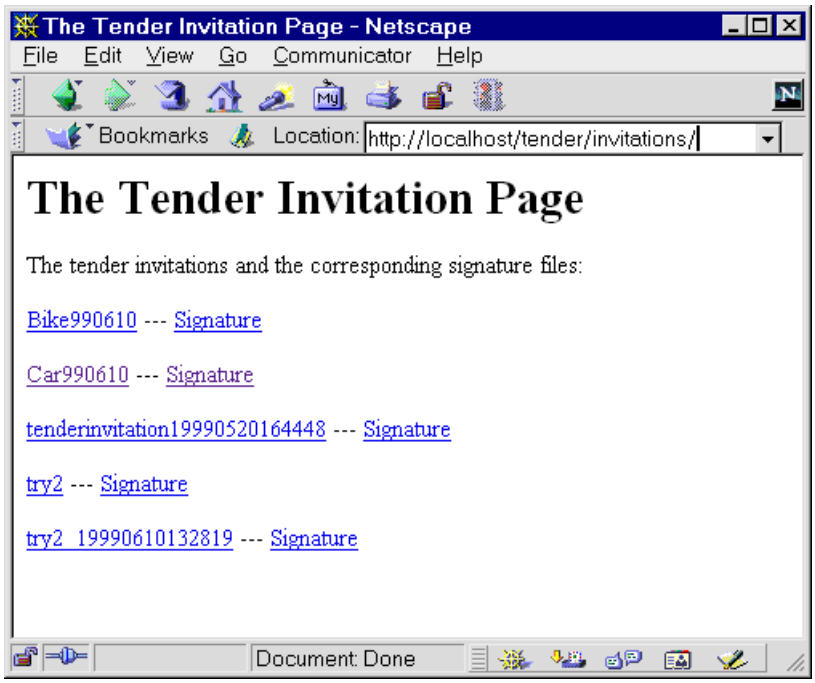


Figure 8: Automatically generated page listing currently available tender invitations.

4.4 Discussion

The implementation described in this section is a prototype. Therefore, we have not resolved the problem occurring when the client browser is executed in the Windows environment. Moreover, there are several security-related issues concerning the implementation that should be considered.

As is often the case, CGI-scripts can be used to cause unintended behavior. When the buyer controls the WWW-server, which we assume to be the case, the CGI-script problems can be prevented by letting the preparation of tender invitations be performed by stand-alone applications outside the realm of the WWW-server. A user with special privileges can then transfer the invitations to the appropriate directory. In this way, there will not exist any CGI-script, for posting of invitations, that can be accessed via the WWW-server. An alternative solution is to execute the CGI-script with another user identity than the one of the WWW-server. This can be achieved with the *suEXEC* feature of Apache (Apache Group 1999).

Actually, to strengthen the security the CGI-scripts executed with the user identity of the WWW-server should not own any files or have any write access rights. Consequently, also the CGI-script storing the tenders should be executed with the *suEXEC* feature of Apache.

5. Example

In this section a basic example is given. The example illustrates the steps to be taken and the resulting interaction, when the electronic tender system is used to administrate the tender process.

The buyer comes to the conclusion that a car needs to be bought. Accordingly, a tender invitation is written, specifying all the requirements on the car. The document is inserted into the HTML-form for tender invitation posting, depicted in Figure 6. Thereafter, the name *Car990610* is assigned to the tender invitation and the '*Sign and Submit*' button is selected to initiate the signing and posting to the WWW-server. The corresponding CGI-script, executed by the WWW-server, performs the following tasks:

- Verifies that the name *Car990610* is unique, that is, not used by any other invitation. If this is not the case, the buyer will be asked to post the invitation again, with a new name.
- Stores the tender invitation and the PKCS #7 object in the *tender/invitations* directory, using the filenames *Car990610* and *Car990610.sig* respectively.
- Generates an index file for the *tender/invitations* directory providing links to the available invitations and their signatures.

When the buyer has performed these steps, potential suppliers can read the tender invitation. Potential suppliers looking up the URL of the tender invitation page will be presented the page depicted in Figure 8, or something similar. The files can then be stored locally and the signature verified with the signature verification tool described in Section 4.1.3. Substantial convenience is added by the graphical user interface, illustrated in Figure 3.

The potential suppliers that would like to become the supplier write a tender answering to the requirements of the invitation. Thereafter, they employ the HTML-form on the tender submission page, depicted in Figure 7, to submit the tender. In order to do so the (file) name of the invitation, in this case *Car990610*, has to be supplied and the proper certificate has to be selected, when asked for by the *crypto.signText* method, as illustrated by Figure 2.

When the WWW-server receives the posted data, a CGI-script is executed to process it. The CGI-script performs the following tasks:

- Verifies that there is an invitation with the name *Car990610*. If this is not the case, the supplier will be asked to submit the tender again, supplying the correct invitation name. If this really would be the case here, then the buyer must have removed the invitation. This action is justified if the invitation has expired; else it means that the buyer has done a premature withdrawal of the tender.
- Creates unique filenames for the tender and the PKCS #7 of the form *invitationname_date&time*, where *date&time* is a 14-digit number built by concatenating the current year (four digits), month, day, hour, minute, and second. If the name is already taken then the script waits for two seconds before trying again.
- Stores the tender and the PKCS #7 object in the *tender/tenders* directory, using the filenames *Car990610_19990615175322* and *Car990610_19990615175322.sig* respectively, if that happened to be the current date and time.

With all these steps done, the buyer can retrieve the tenders responding to the invitation, verify the signatures, and select the most attractive offer. Since the tenders have been digitally signed, the buyer has a strong case if their submission has to be proved.

6. Conclusions

In this report a secure electronic tender system has been proposed and a prototype implementation of the system has been described and evaluated. The proposed system can be used to decrease the effort that has to be put into the tender process and, thus, increase its cost-effectiveness. Moreover, the security issues of confidentiality, integrity, and non-repudiation are handled. Except for an extension handling the contract phase following the tender process, the possible future extensions and improvements relate to the security and user convenience of the prototype implementation.

The usability of the implemented prototype is hampered by inter-platform compatibility problems. For example, the signature verification tool is only available for the Solaris and Windows NT platforms. Moreover, the infamous carriage-return problem appears when the client browser is executed under Windows NT and the WWW-server under Unix. This problem obviously has to be resolved for a non-prototype implementation.

A more tightly integrated implementation is required for non-experienced (or not security-aware) users. For example, the signature of the tenders could be verified automatically by the system upon arrival to the WWW-server. On the other hand, by separating the tools for tender invitation posting from the WWW-server the security will be improved, since the corresponding CGI-script will not be accessible from the outside any longer.

A problem, which has the potential to become exceedingly severe, is the lack of inter-application handling of certificates, as mentioned in Section 2, and the somewhat careless implementation of these routines in some software. This threatens the integrity of the whole certificate hierarchy.

Bibliography

Apache Group, *Apache 1.3 User's Guide*, 1999.
http://www.apache.org/docs

P. Ashley and M. Vandenwauver, *Practical Intranet Security – Overview of the State of the Art and Available Technologies*, Kluwer Academic Publishers, Boston, 1999.

Garfinkel and Spafford, *Web Security & Commerce*, O´Reilly & Associates, 1997

B. Laurie, A. Laurie, *Apache-SSL Documentation*, 1999.
http://www.apache-ssl.org/docs.html

A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1999.

Netscape, *Signing Text from JavaScript*, 1998a.
http://developer.netscape.com/docs/manuals/security/sgntxt/index.htm

Netscape, *Using the Signature Verification Tool*, 1998b.
http://developer.netscape.com/docs/manuals/security/signver/index.htm

M. Persson, *Mobil kod och säker exekvering*, FOA-R--98-00807-503--SE (in Swedish), 1998.

RSA Laboratories, *PKCS #7: Cryptographic Message Syntax Standard*, Version 1.5, November 1993.
http://www.rsa.com/rsalabs/pubs/PKCS/html/pkcs-7.html

Appendix A

HTML and Javascript code for the tender invitation-posting page:

```
<HTML>

<HEAD>

<SCRIPT>

function signForm(theForm, theWindow, validation) {

        var signedText = theWindow.crypto.signText(theForm.tender.value, "ask");
```

```
        theForm.validation.value = signedText;
        theForm.submit();
        return true;
}
</SCRIPT>
</HEAD>
<BODY>
<H1>The Tender Invitation Posting Page</H1><P>
<FORM NAME = "tenderform" METHOD="post" ACTION="/cgi-bin/postinvitation.cgi">
        <INPUT NAME="name" SIZE=40 VALUE="Put name here (only letters and digits)">
        <BR><BR><BR>
        <TEXTAREA NAME="tender" ROWS=25 COLS=70>
                Put invitation here.
        </TEXTAREA><BR><BR>
        <INPUT TYPE="hidden" NAME="validation">
        <INPUT TYPE="button" NAME="sign" VALUE="Sign and Submit" ONCLICK="signForm(tenderform,
        window, tenderform.validation);">
</FORM>
</BODY>
</HTML>
```

HTML and Javascript code for the tender-posting page:

```
<HTML>
<HEAD>
<SCRIPT>
function signForm(theForm, theWindow, validation) {
        var signedText = theWindow.crypto.signText(theForm.tender.value, "ask");
        theForm.validation.value = signedText;
        theForm.submit();
        return true;
}
</SCRIPT>
</HEAD>
```

```
<BODY>
<H1>The Tender Submission Page</H1><P>
<FORM NAME = "tenderform" METHOD="POST" ACTION="cgi-bin/submittender.cgi">
        <INPUT NAME="name" SIZE=55 VALUE="Put filename of tender invitation responding to here">
        <BR><BR><BR>
        <TEXTAREA NAME="tender" ROWS=25 COLS=70>
        Put tender here.
        </TEXTAREA><BR><BR>
        <INPUT TYPE="hidden" NAME="validation">
        <INPUT TYPE="button" NAME="sign" VALUE="Sign and Submit" ONCLICK="signForm(tenderform,
        window, tenderform.validation);">
</FORM>
</BODY>
</HTML>
```

The code of the Perl script processing tender invitation postings:

```
#! /usr/local/bin/perl
sub decode {
        read (STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
        @pairs = split(/&/, $buffer);
        foreach $pair (@pairs)
        {
                ($name, $value) = split(/=/, $pair);
                $value =~tr/+/ /;
                $value =~ s/%0D//eg;
                $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
                $FORM{$name} = $value;
        }
}
print "Content-type: text/html\n\n";
&decode();
$dataSignature = $FORM{'validation'};
$dataToSign = $FORM{'tender'};
```

```
$filename = $FORM{'name'};
chdir "../htdocs/tender/invitations/";
print "<BR><B>File name: $filename</B><BR>";
if(open TENDERFILE, $filename) {
        close TENDERFILE;
        print "<BR><B>Name already used. Try a different name.</B><BR>";
        die;
}
close TENDERFILE;
open TENDERFILE, ">$filename" or die "Could not open file";
print TENDERFILE "$dataToSign";
close TENDERFILE;
$sigfilename = $filename . ".sig";
open SIGNATUREFILE, ">$sigfilename" or die "Could not open file";
print SIGNATUREFILE "$dataSignature";
close SIGNATUREFILE;
$value = $dataToSign;
$value =~ s/\n/<BR>\n/g;
print "<BR><B>Signed Data (in $filename):</B><BR>", "$value", "<BR>";
$value = $dataSignature;
$value =~ s/\n/<BR>\n/g;
print "<BR><B>Signature Data (in $sigfilename):</B><BR>", "$value", "<BR>";
opendir(DIR, ".") or die "can't open dir: $!";
@invitations = grep { /\w+/ && -f $_ . ".sig" } readdir(DIR);
closedir DIR;
open TENDERFILE, ">index.html" or die "Could not open file";
print TENDERFILE "<html>\n";
print TENDERFILE "<head>\n";
print TENDERFILE "<title>The Tender Invitation Page</title>\n";
print TENDERFILE "</head>\n";
print TENDERFILE "<body>\n";
print TENDERFILE "<h1>The Tender Invitation Page</h1>\n";
print TENDERFILE "The tender invitations and the corresponding signature files:<br><br>\n";
```

```
foreach (@invitations) {
        print TENDERFILE "<a href=\"$_\">$_</a> --- ";
        print TENDERFILE "<a href=\"$_.sig\">Signature</a><br><br>\n";
}
print TENDERFILE "</body>\n";
print TENDERFILE "</html>\n";
close TENDERFILE;
```

The code of the Perl script processing tender submissions:

```
#! /usr/local/bin/perl
sub decode {
        read (STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
        @pairs = split(/&/, $buffer);
        foreach $pair (@pairs)
        {
                ($name, $value) = split(/=/, $pair);
                $value =~tr/+/ /;
                $value =~ s/%0D//eg;
                $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
                $FORM{$name} = $value;
        }
}
sub createfilename {
        local ($filename);
        $filename = "_";
        @temp = localtime;
        @temp[5] += 1900;
        @temp[4]++;
        for($i = 5;$i >= 0;$i--) {
        if(@temp[$i] < 10) {
                $filename .= "0";
        }
```

```
		$filename .= @temp[$i];
		}
		return $filename;
}
print "Content-type: text/html\n\n";
&decode();
$dataSignature = $FORM{'validation'};
$dataToSign = $FORM{'tender'};
$filename = $FORM{'name'};
chdir "../htdocs/tender/invitations/";
print "<BR><B>Tender invitation name: $filename</B><BR>";
if(not open TENDERFILE, $filename) {
		print "<BR><B>No such tender invitation, try again.</B><BR>";
		die;
}
close TENDERFILE;
$tendfilename = $filename . createfilename();
chdir "../tenders/";
print "<BR><B>Tender name including local time: $tendfilename</B><BR>";
while(open TENDERFILE, $tendfilename) {
		close TENDERFILE;
		sleep 2;
		$tendfilename = $filename . createfilename();
		print "<BR><B>Tender name including local time: $tendfilename</B><BR>";
}
close TENDERFILE;
open TENDERFILE, ">$tendfilename" or die "Could not open file";
print TENDERFILE "$dataToSign";
close TENDERFILE;
$sigfilename = $tendfilename . ".sig";
open SIGNATUREFILE, ">$sigfilename" or die "Could not open file";
print SIGNATUREFILE "$dataSignature";
close SIGNATUREFILE;
```

```
$value = $dataToSign;

$value =~ s/\n/<BR>\n/g;

print "<BR><B>Signed Data (in $tendfilename):</B><BR>", "$value", "<BR>";

$value = $dataSignature;

$value =~ s/\n/<BR>\n/g;

print "<BR><B>Signature Data (in $sigfilename):</B><BR>", "$value", "<BR>";
```