Secure WWW Transactions Using Standard HTTP and Java Applets

F. Bergadano, B. Crispo, and M. Eccettuato
Universitá di Torino, Italy

Secure WWW Transactions Using Standard HTTP and Java Applets

F. Bergadano
Dipartimento di Informatica, Università di Torino, Italy
bergadan@di.unito.it
B. Crispo
Dipartimento di Informatica, Università di Torino, Italy
Cambridge University Computer Laboratory, England
bc201@cl.cam.ac.uk
M. Eccettuato
Dipartimento di Informatica, Università di Torino, Italy

Abstract

Can users access information on the Web securely with their unchanged, normal broswers, and yet without relying on the cryptographic software contained in those browsers? In this paper we show that this is possible, with a communication architecture based on Java applets. This is important, because cryptographic functions need be separated from both the user interface and the communications routines. It must be possible to acquire the source code for the relevant modules and alternative software vendors must be available, in order to avoid hidden trapdoors and undetected implementation problems. Our approach is alternative to solutions at the protocol level (e.g., SSL), because the unchanged HTTP/TCP/IP stack is maintained. Moreover, it does not require the installation of proxies.

1 Introduction

The explosive growth in information that becomes available through the Web has led to the development of new applications. Some of those applications, such as electronic commerce or teleworking, are particularly critical because they require secure communications between clients and servers. For those applications, WWW transactions must offer security services such as authentication, secrecy, data integrity and non repudiation. Some browser vendors and standardisation groups have already proposed some solutions to this problem, in a way that we consider still unsatisfactory from users' point of view.
Standardisation groups have proposed to secure HTTP transactions at the protocol level. Even if these solutions enhance the security of the communications between WWW clients and servers, they are hidden below the application level, where users are often unable to access data and system components. Moreover, new protocols require new client and server software, that may not become as widespread as standard HTTP-based implementations. Browser vendors, sometimes, provide solutions at the application level. However, for commercial reasons they may not provide the source code of their solutions. The choice of cryptographic modules may also be limited by local regulations.
We believe that new approaches are needed.

In this paper we describe a framework based on Java applications and Java applets [3] to secure HTTP transactions. We have implemented a system that allows users to perform all the encryption and authentication work outside the browser, by using applets and other locally installed software.

Software is also installed on the WWW server and performs corresponding encryption and authentication procedures. All the software we use can be easily studied and analysed. Moreover users can eventually integrate their own modules to perform the cryptographic operations without affecting the validity of our approach. Our solution enforces strong security without requiring modifications in the existing HTTP protocol [7, 12] or in the available commercial browsers. Any browser supporting Java may be used.

2 Related Work

There are already several different approaches used to secure HTTP transactions. They can be classified as follows:

1. **Application protocol solutions.** The HTTP protocol is changed so that the header includes key management features and encryption/signature becomes possible. A well known example of this kind is the SHTTP protocol [11]. Another modification of HTTP that supports server group authentication is described in [17].

2. **Session layer solutions.** HTTP communication is implemented on top of a modified transport API, that guarantees end-to-end privacy and authentication. A well known example following these principles is SSL [13, 10].

3. **Transport layer solutions.** HTTP communications can also be secured at the transport level, securing the underlying TCP connections using tunnelling protocols such as PPTP [15] or SSH [21] [20], or at the network level using, e.g., IPv6 [9]. The Point-to-Point Tunnelling Protocol was designed to provide authentication and encryption over a public TCP/IP network using the common Point-to-Point Protocol (PPP), thus creating a Virtual Private Network (VPN). PPTP works by encapsulating the virtual network packets inside of PPP packets, which are in turn encapsulated in Generic Routing Encapsulation packets sent over IP from the client to the gateway PPTP server and back again. PPTP does not specify particular algorithms for authentication and encryption. Instead, it provides a framework for negotiating particular algorithms. In practice most commercial products use the Microsoft Windows NT version of the protocol because it is already a part of the operating system. Even if PPTP is secure in theory, a recent flow discovered in its Microsoft implementation [20] suggests to evaluate this solution carefully. The SSH protocol can be used as a generic transport layer encryption mechanism, providing both host authentication and user authentication, together with privacy and integrity protection. SSH uses a packet-based binary protocol that works on top of any transport that will pass a stream of binary data, e.g. TCP/IP. It was originally designed to provide secure remote login but the current version also supports secure forwarding of arbitrary TCP/IP connections. The SSH server can listen for a socket on the desired port, dedicated for this purpose, automatically forwards the request and data over the secure channel, and makes the connection to the specified target port from the other side. If the target port is the port dedicated to the HTTP server, SSH can be used to secure HTTP connections. IPv6 defines two headers (Authentication and Encapsulation Security Payload) to support authentication and encryption at network level.

4. **Architectural solutions.** On the HTTP client side, the browser is configured to use an HTTP proxy. User requests will then be forced through the proxy, where encryption and signature services are available. On the HTTP server side, two solutions are possible: (1) a proxy is installed that handles encryption and signatures, and exchanges data with the actual server, or (2) the HTTP server is modified so as to guarantee secure communication with the client side.

5. **Application layer solutions.** Our proposal, uses no proxies, does not

change HTTP or socket APIs at the TCP level. Instead, HTTP communications are forced through a separate TCP connection with the help of applets within the normal browser environment. The most similar approach in the existing literature is found in [19], where CGI [14] programs are used instead of applets and secure communication is achieved with PGP [22].

Each approach has drawbacks and advantages, that may make it appropriate for different applications and different commercial contexts. Approaches number 1 and 2 are natural and technically correct. However, they typically require the implementation of a browser that supports the proposed techniques. As a consequence, strong encryption may be prevented due to regulatory considerations by the browser vendor. Moreover, there is no public access to most browsers' source code. This requires the user's trust in the cryptographic functions implemented in the browser. If, and when, the source code is delivered, the situation will obviously be improved. However, compatibility and extension software regulation will need to be addressed with care.

Solutions cited at number 3, a part for the problems of the specific solution, all present the limit to address the problem of end-to-end authentication at the transport level. In an environment where the same workstation can be shared among different users the same authenticated connection can be easily shared by a malicious user. A solution capable to provide end-to-end authentication at the application level guarantees better security.

Our approach (number 5) and approach number 4 may use commercial browsers and HTTP servers as they are available now. Therefore, they do not suffer from the above problems. However, they introduce more overhead in the communication structure. Our approach, though, is only active when needed, i.e. web pages that do not require privacy or authentication are retrieved normally. Proxies, on the other hand, seem to be hard to enable on a case by case basis. Moreover, if proxies are implemented on a separate computer, the WWW transaction's security may be at risk on the path from the browser to the proxy.

3 Our Proposal

WWW transactions are usually performed by two parties: WWW browsers (clients) and httpd processes (servers). Information flows initially from a client to a server. Users click on a URL, or they fill a form and submit the data, and a request is sent to the correspondent server. Then the server processes the request and sends back to the client the information requested, usually as an HTML page, which can embed several different document formats (e.g. postscript, jpeg, mpeg, text, etc).

Our solution splits these two communications in several steps, in such a way that additional security services can be implemented. The user interface is unchanged, any Java-enabled browser will be acceptable. It is also worth to remark the differences between applets and applications in our solution. While applets have to be implemented using Java, the applications could be coded using any programming language. We have chosen to use Java also for the applications only to have an homogeneous developing environment.

In the paper we assume that we can rely upon an already existing authentication infrastructure. Thus we will not describe details related to public key distribution and management. We suppose as well that users can easily and securely get the server's public keys. We also assume that users can perform encryption/decryption operations as well as signing/verification of data.

Each step in the secure communication procedure is explained below.

- *User request (Figure 1).* Access to the remote information will be obtained by the user with the browser in the usual way: by clicking on the relevant hyper-

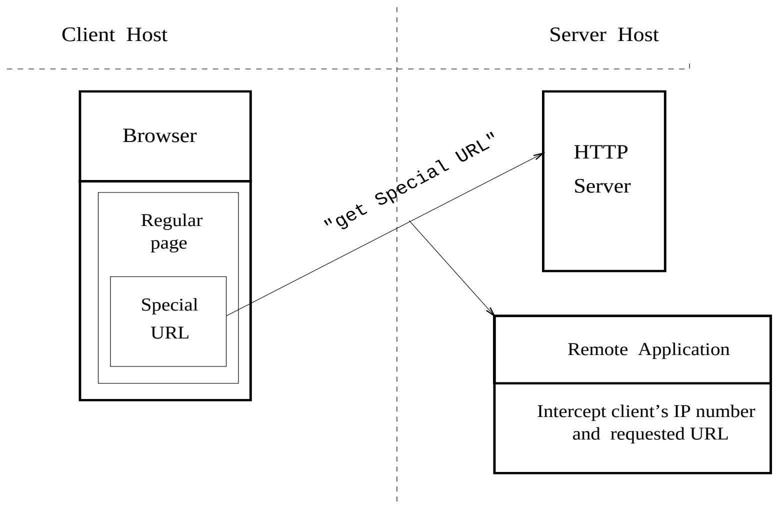

Figure 1:

link or by opening the URL directly[1]. The client's IP address, as well as the requested URL will also be intercepted, on the server side, by a running process that will later perform the actual communication, and which we shall call the *remote application*.

- *Retrieving a remote applet (Figure 2).* The selected URL will not correspond to the actual information on the server. Instead, it will reference a simple applet that the browser will run to initialise the subsequent communication environment. This will be called the *remote applet.*

  Such applets are programmed in Java and are run by the browser on the client side. In general, there has been concern regarding the security of local execution of mobile code, such as the applets. Most browsers have very strong restrictions on the execution of applets. In particular, remote applets cannot read or write local files, they cannot execute other programs in the local environment and they can only communicate with the computer they originate from. In our proposed architecture, the remote applet could be screened for security problems with even more care. The remote applet, in fact, is an extremely simple and short piece of software (see the Appendix). Moreover, the applet's code is known and is always the same, for our proposed architecture. The browser could then check whether the remote applet corresponds to a predefined pattern, and proceed to execution only if that is the case. In other words, even with very strict security policies, and in environments where running general remote applets is disallowed, the browser could agree to receive and run a predefined and limited set of specific applets. Our remote applet (received as described in Fig. 2) would be among those. At the implementation level, the browser needs to store a hash value for the allowed applets. When a remote applet is retrieved, it is run locally only if its hash value is among those previously authorised.

This whole procedure, and also the retrieval of the remote applet, is needed only if privacy and/or authentication of HTTP transactions are required. Otherwise, the base information can be immediately transmitted to the browser on the client side, and displayed to the user. The WWW server administrators will then decide whether security services are required for each of the available pages.

[1]At that point, the user will expect some kind of response from the server, e.g. some HTML page, and we shall call this the *base information.* At the level of the browser's interface, the user will obtain the base information in a transparent way, as if the information was actually stored under the selected URL. The base information will be displayed normally in the browser window when the whole procedure is completed.

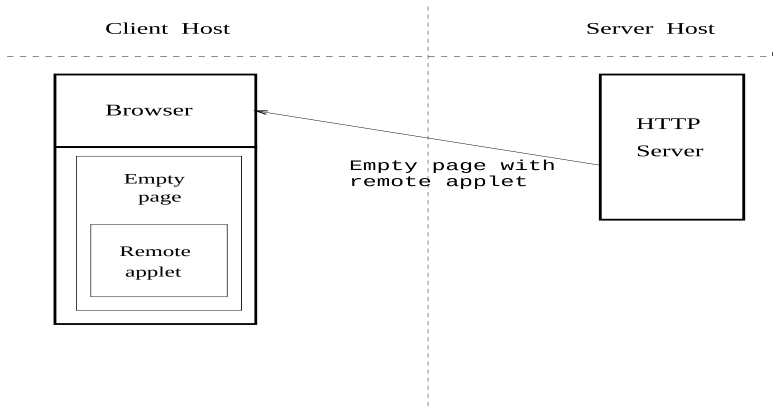

Figure 2:

If not, a page is stored under its address as advertised on the Web. Otherwise, the standard remote applet is stored in a page under that address, and the actual information is obtained by the user through the procedure explained below.

- *Opening a local applet.* The remote applet basically only does one thing: it forces the browser to open a fixed-name local URL. For instance, this URL could be "file:/security/applets/local.html". The remote applet can do this through a standard Java method called Showdocument. After this, it terminates. The remote applet is then only needed to open a local page. However, it cannot be eliminated from the architecture. First, the user will reference the base information by clicking on a remote URL, as found in hyperlinks and search index results. Therefore, the local URL, that is needed later, cannot be opened by the user in a natural way, and this must be done by the remote applet. Second, by requesting the remote URL, the browser will allow the server to record on its files the client's IP address and the URL that was referenced. This information is needed later on the server side. At this point, the remote applet has completed its life cycle and control is passed to the local URL. The local URL will reference a more substantial applet, that will control the following interaction with the user. This will be called the *local applet.* The local applet is considered trusted within this study. In fact, it is not loaded from the network at this time. It could be obtained separately, through a channel that allows for authentication.

- *Connection establishment (Figure 3).* On the HTTP client side, a *local application* will also be running and will communicate both with the local applet and with a remote application, running on the server machine. Such applications are actually general processes that run on the client and on the server machine. In our prototype implementation, such processes are being written as Java applications. For both of these communications, the local application will act as a server (although it runs on the HTTP client side). As a consequence, it will not need to know the IP address of the HTTP server machine. This is important as, e.g. in Netscape Communicator it is impossible to obtain this IP address without direct user input, because remote applets are very limited in their interaction with the computer they run on. In particular, current configurations would not allow the remote and local applet to communicate. This is a correct design choice in general, but we need a way around the problem of passing the HTTP server's IP address to the local application. This is accomplished by making the remote application start the TCP connection to the local application.

  On the HTTP server side, the remote application will be running and it will have obtained the IP address of the HTTP client machine as it was saved on the

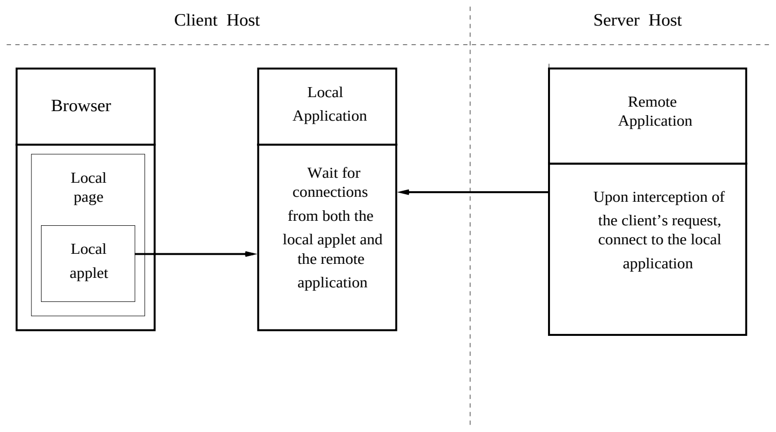
Client Host
Server Host
Browser
Local page
Local applet
Local Application
Wait for connections from both the local applet and the remote application
Remote Application
Upon interception of the client's request, connect to the local application

Figure 3:

server files as indicated in Fig. 1. Then, this application can behave as a client, and open a socket to the local application. This socket will be used for the most important part of the overall communication, including transmission of encrypted and authenticated user input and base information. As a first operation, the local and remote applications will handshake and exchange public key certificates. Protocols such as X.509 [16] or its variations [6, 8, 18, 2, 4] may be used for this purpose.

- *Secure server-to-client communication (Figure 4).* The remote application, then, will have available the URL initially referenced by the user on the client side browser, because it was stored on the server files (Fig. 1). This URL is used by the remote application to find the desired base information on the files of the HTTP server computer. Access control is performed based on user name, public key, and possibly other information obtained from the local application. In fact, the local application had previously obtained user information from the local environment, where user data and keys can be stored. At the operating system level, the user who is interacting with the browser could also undergo more sophisticated authentication procedures, that may be performed by the local applet based on passwords and/or biometric identification techniques. The local applet would then inform the local application of the verified user identity, and user data are communicated to the remote application. If the user has appropriate access rights, the base information is then encrypted and signed on the HTTP server side, and is sent to the local application.

- *Displaying the base information to the user (Figure 4).* On the HTTP client side, the local application will decrypt and authenticate the base information, and pass it to the local applet for displaying in the browser's window. The user will now see the base information, as if it were retrieved directly after clicking on the relevant link. The only difference is that the user will see a local URL with a fixed name in the browser's URL box, instead of the selected remote URL. This is against a desideratum of perfect transparency, but is actually a major advantage from a security point of view. In fact, the user can tell whether the information is local, and hence processed and received in a secure way.

- *Obtaining user input (Figure 5).* After displaying the base information, if the case requires it, the local applet will ask for user input, e.g., the user's name, credit card number and the amount to be payed. This input should then passed by the applet to the local application. However, this is not trivial. If the base information is a form, the user input will be collected by the browser, and then it will be sent to the HTTP server that is

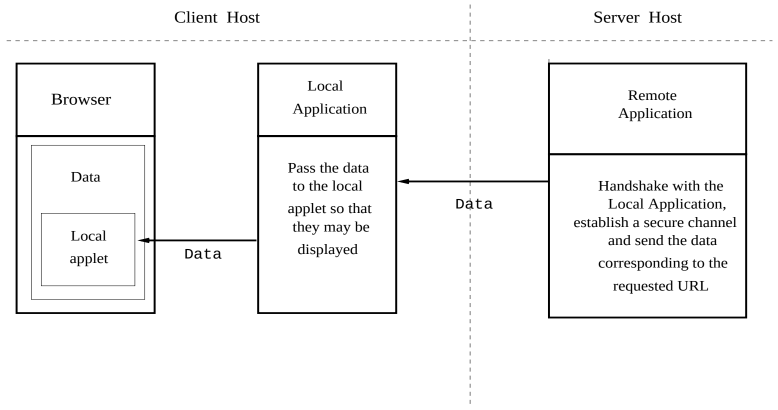
Client Host
Server Host
Browser
Data
Local applet
Data
Local Application
Pass the data to the local applet so that they may be displayed
Data
Remote Application
Handshake with the Local Application, establish a secure channel and send the data corresponding to the requested URL

Figure 4:

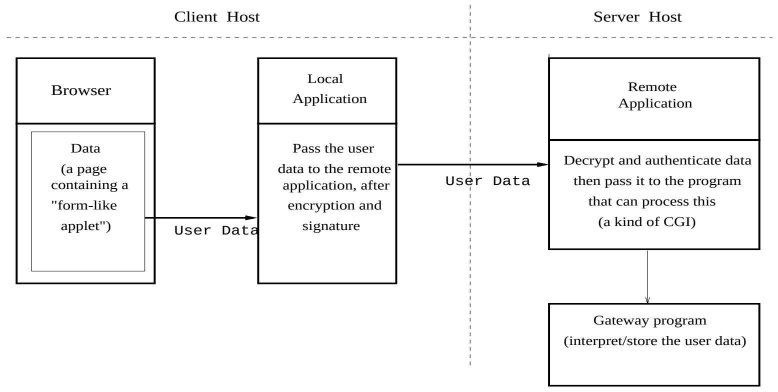
Client Host
Server Host
Browser
Data (a page containing a "form-like applet")
User Data
Local Application
Pass the user data to the remote application, after encryption and signature
User Data
Remote Application
Decrypt and authenticate data then pass it to the program that can process this (a kind of CGI)
Gateway program (interpret/store the user data)

Figure 5:

indicated in the form, in an insecure way. The local application will be cut out from this communication. We have considered two solutions to this problem: (1) installing a local HTTP server where user input could be redirected, and a general purpose CGI that transfers all data to the local application for further processing and transmission, and (2) requiring the HTTP server not to use forms, but a special parameterised Java applet that we provide and behaves like forms - when running in the browser, this applet will collect user data and immediately transfer it to the local application. Solution (2) was chosen in the implementation, because requiring an HTTP server on the original client host may be in many cases an excessive overhead. Solution (2) is cleaner and easier to install on general platforms, but requires Web sites to use our special syntax for forms.

- *Secure client to server communication (Figure 5).* The local application will now have received (1) user input obtained from our "form-like" applet and (2) the server's certified public key sent during connection establishment. The user input is then encrypted, signed, and sent to the remote application. There the plaintext information is saved in the appropriate files or passed to the needed end application, i.e. to something like a CGI script. We assume the end application does not need to send back results to the browser immediately - this can be done, but it makes the overall picture more complicated, and has not been included in the implementation. The user will then continue interacting with the browser normally, e.g. by going back to previous pages, or by clicking on new links that were displayed by means of the local applet.

One should observe that there are two distinct phases in the proposed protocol: first, an HTTP request is sent by the client and a corresponding small remote applet is received and executed, second, a local application and a remote application exchange the actual data, that may be transmitted in an authenticated and private way. The two phases are not bound in a single session, and therefore there is a spoofing opportunity here. The client may connect to one server for the first phase (no cryptography here), and the attacker could plug in her remote application for the second phase. Although this is certainly possible, we assume the client's local application will have authentication mechanisms (e.g. based on public keys and certificates) that will allow it to verify the actual identity of the remote application's owner. If the bogus application does not have the private key associated to the Web server of the first phase, as listed in the initial HTTP request, the data transfer fails. In summary, the first phase is only a trigger for the second phase, where the whole authentication procedure is performed.

4 Conclusion

One of the main problems that needs to be solved, before electronic commerce can become widespread and an every day habit, is to build trust among users in the safety of using the Internet. One little step in this direction is represented by securing WWW transactions. The World Wide Web is one of the most used network applications. Browsers represent user interfaces that can be used for a great variety of services.

In this paper we have presented an alternative approach to secure HTTP transactions. We believe that solutions that claim to solve security weaknesses through the use of unscrutinable and proprietary software are unsatisfactory. Many times they simply shift the line where the weaknesses can be found, from the Internet to those specific solutions [5, 1]; they often add security holes to compatibility problems. On the other hand solutions that aim to change widely used standards such as the HTTP protocol are, we think, too optimistic.

The system we have described, and that is now under testing and freely available, provides a solution without these shortcomings. The software is available at `http://maga.di.unito.it/fb/SWWWT`. A prerequisite to use our solution is for users to have a Java virtual machine running in their environment. The drawbacks of our infrastructure are: - a little degradation in performance (only when a secure channel is necessary), and - with the current version of the software, users have to agree to use a form-like applet, instead of a normal HTML form, to submit their data to the server. Servers must also adapt to this special kind of forms.

We are working to implement an alternative to this solution, using an additional, local HTTP server as mentioned in Section 3, in such a way that users can choose the solution they wish.

5 Acknowledgement

The authors would like to thank Ross Anderson for providing helpful comments on a draft version of this paper. The authors also thank Cosimo Anglano for helpful discussions on the implementation.

References

[1] *Bugs in Microsoft Internet Information Server v 1.0* . `http://www.ntsecurity.com/News/bugs/iis-bug1.html`.

[2] *IETF - PKIX Working Group - Internet Public Key Infrastructure.* `http://www.ietf.org/html.charters/pkix-charter.html`.

[3] *Java Documentation and Distribution.* `http://www.javasoft.com/`.

[4] *NIST - Public Key Infrastructure Program.* `http://csrc.nist.gov/pki/`.

[5] R.J. Anderson, B. Crispo, J.H. Lee, C. Manifavas, F.A.P. Petitcolas, and V. Matyas Jr. *Global Trust Register 1998.* Nortghate Consulting Ltd., 10 Water End, Wrestlingworth, Bedfordshire SG19 2HA, England., 1998.

[6] F. Bergadano, B. Crispo, and M.T. Lomas. *Strong Authentication and Privacy with Standard Browsers. Journal of Computer Security*, 1997. Special issue on World Wide Web security - vol. 5 n. 3 pp. 191-212.

[7] T. Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*, May 1996. RFC 1945.

[8] B. Crispo and M. Lomas. *A Certification Scheme for Electronic Commerce.* In *Security Protocol Workshop*, volume LNCS series vol. 1189. Springer-Verlag, 1997.

[9] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification.* Technical report, December 1995. RFC 1883.

[10] T. Dierks and C. Allen. *The TLS Protocol, version 1.0*, November 1997. Internet Engineering Task Force Internet Draft.

[11] A. Schiffman E. Rescorla. *The Secure HyperText Transfer Protocol*, May 1996. Internet-Draft.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*, January 1997. RFC 2068.

[13] A. Freier, P. Karlton, and P. Kocher. *The SSL Protocol Version 3*, December 1995.

[14] S. Gundavaram. CGI Programming on the World Wide Web (Nutshell Handbook). O'Really & Associates, 1996.

[15] K. Hamzeh, G.S. Pall, W. Verthein, J. Taarud, and W.A. Little. *Point-to-Point Tunneling Protocol.* Technical report, `http://www.ietf.org/internet-drafts/draft-ietf-pppext-pptp-02.txt`, July 1997. Internet Draft.

[16] Information Technology - Open Systems Interconnection, Geneva. *Recommendation X.509* , June 1995. The Directory: Authentication Framework.

[17] M. Kaiserswerth, A. Hutchison, and P. Trommler. *Secure World Wide Web Access to Server Groups.* In *Proceedings of the IFIP TC6/TC11 International Conference on Communication and Multimedia Security.* Chapman and Hall, September 1996.

[18] R.L. Rivest and B. Lampson. *SDSI - A Simple Distributed Security Infrastructure.* `http://theory.lcs.mit.edu/~cis/sdsi.html`, April 1996.

[19] B. Sanderson, J.D. Weeks, and A. Cain. *CCI-based Web Security: A Design Using PGP.* In *Proceedings of the 4th International World Wide Web Conference*, December 1995.

[20] B. Schneier and P. Mudge. *Cryptanalysis of Microsoft's Point-to-Point Tunnelling Protocol (PPTP).* In *Proceedings of the 5th ACM Conference on Computer and Communication Security*, 1998.

[21] T. Ylonen, T. Kivinen, and M. Saarinen. *SSH Connection Protocol*, November 1997. Internet Draft.

[22] P.R. Zimmermann. The Official PGP User's Guide. Boston: MIT Press, 1995.

6 Appendix

Here is the source code of the remote applet that is downloaded by the client. This version is for Unix, but our implementation also supports Windows 95. The comments present in the actual code are omitted here. The complete code distribution, also including a demo, is available from
`http://maga.di.unito.it/fb/SWWWT`

```
RemoteApplet.java

package SecureWWW.Server;
```

```
import java.applet.Applet;
import java.applet.AppletContext;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;

/** This applet is downloaded when the
* user clicks on a reserved URL. It begins
* the whole transaction, forcing the browser
* to show the document containing the local
* applet. */

public final class RemoteApplet
  extends Applet implements SecureInterface {

 private final static String
   defUNIXlocalloaderURLname =
   "file:/usr/local/Localloader.html";

 private static URL localloaderURL = null;

 private static boolean debug = false;

 public void init() {
   debug = getParameter("debug") != null ?
    getParameter("debug").equals("true") :
    false;

   boolean noDefault =
    getParameter
    ("Localloader.NoDefault") != null ?
    getParameter
    ("Localloader.NoDefault").equals("true")
    : false;

   if ((localloaderURLName =
    getParameter
     ("Localloader.UNIX")) != null)
    try {
      localloaderURL =
        new URL(localloaderURLName);
    }
    catch (MalformedURLException except) {
     if (debug) {
      System.err.println
       ("[RemoteApplet <" + this + ">]"
       + warnSpace + "Incorrect "
       + "URL name <"

          + localloaderURLName + ">"
          + warnSpace + "for "
             + "Localloader in parameter"
          + (noDefault ? "." :": trying "
          + "default."));
         showStatus
    ("[RemoteApplet] Incorrect parameter.");
        }
    }
    if (localloaderURL == null && !noDefault)
      localloaderURLName =
        defUNIXlocalloaderURLname;
 }

 if (localloaderURL == null)

  if (!noDefault)

    try {
      localloaderURL =
        new URL(localloaderURLName);
    }
    catch (MalformedURLException except) {
    }
  else {
    System.err.println
      ("[RemoteApplet <" + this + ">]"
      + "Incorrect "
      + "parameter or no parameter,
         and no default allowed"
      + errSpace
      + "for Localloader: halting.");
    showStatus("[RemoteApplet] No valid
       URL name for Localloader.");
  }
  if (localloaderURL != null)
      if (debug) {
        System.out.println
          ("[RemoteApplet <"
   + this + ">]"
   + okSpace + "Local file URL <"
   + localloaderURL + "> created.");
        showStatus("[RemoteApplet]
          Localloader URL created.");
      }
}

public void start() {
  if (localloaderURL != null) {
     getAppletContext().showDocument
 (localloaderURL);
     if (debug) {
        System.out.println
          ("[RemoteApplet <" + this + ">]"
          + okSpace + "Local file URL <"
          + localTloaderURL + "> shown.");
        showStatus
     ("[RemoteApplet]
       Localloader URL shown.");
     }
  }
}

public String[][]
 getParameterInfo() {
 String[][] info = {
    {"debug", "boolean",
    "Controls debugging
      informations generation."},
    {"Localloader.NoDefault", "boolean",
      "To impose an exclusive value."},
    {"Localloader.UNIX","local (file) URL",
      "For Solaris, Linux..."},
    return info;
 }
}
```