Introduction to Web Security

An Overview of Technologies for Security, Authentication, and Privacy on the World-Wide Web

Adam Cain
National Center for Supercomputing Applications
acain@ncsa.uiuc.edu
http://www.ncsa.uiuc.edu/People/acain

Abstract:

This document provides an introduction to the issues, technologies and applications of security on the World Wide Web. The complex area of security is first decomposed into several smaller components. After a short discussion of system security issues concerning Web browsers and servers, we turn our attention to technologies designed to provide authentication and privacy in Web communications. A brief review of available cryptographic tools is provided. Then we discuss the goals, mechanics and relative merits of such schemes as PGP/PEM support, Secure-HTTP, Message Digest Authentication, Kerberos support, and the Secure Sockets Layer approach. We close by pointing out the inadequacy of these technologies to the application of online digital payment, and speculate about the future of Web security developments.

Keywords:
WWW, Security, Cryptography, Authentication

Introduction

With the recent explosion in popularity of the World Wide Web, there has been a growing panic based on the realization that "the Web is not secure!" Businesses wishing to set up shop in the emerging Home-page Shopping Network fear that a marauding "hacker" might vandalize their virtual storefronts, or spoof their electronic payment systems and rob them blind. Information providers look for trustworthy ways of limiting access to sensitive documents to particular individuals, as well as ways of verifying an individual's identity in the first place. Meanwhile, users of the Web bemoan their lack of privacy in this new environment.

Unfortunately, these concerns are well founded. Both the World Wide Web and the Internet which forms its foundation were not originally designed to meet the security requirements that are now being demanded of them. This is not necessarily a tragedy, as this "open" design is largely responsible for the tremendous growth of the Web that we

witness today. However it does complicate the task of "adding security" to a medium which is fundamentally non-secure.

Consequentially, we now see a great diversity in proposed solutions to the many aspects of Web security and network security in general. And so in this document we present an overview of the security concerns pertaining to the World Wide Web, and a survey of some of the many technologies designed to address them.

What is "Security" ?

One of the reasons that Web security is currently such a *big* issue is that the topic of network security is itself such a complex, multi-faceted area. Without presenting a complete theory of network/computer security (interested readers may refer to [1] or [2] for a more thorough treatment), we must define a few concepts simply to understand the security objectives of the systems discussed in this document. So we may (imperfectly) subdivide the area of network security into four intertwined components:

- **System Security/Integrity**

  Before we can enable any "secure" use of the Web, we must first be sure that the computer systems involved do not have any serious security "holes" by which intruders may gain undesirable access. This means that a computer's operating system, system software, administrators and users ensure that the machine is not exposed to unintended use.

- **Authentication Mechanisms**

  We will require a variety of methods for verifying the identity of a user or other entity on the network. Authentication mechanisms are means of gaining trust in the identity of a particular party, even when communicating across an untrusted network (i.e., where attackers may control the communication links and alter data flowing across them). These mechanisms may involve passwords, challenge/response systems, digital signatures, or even hardware token-based schemes such as smartcards.

- **Authentication Mechanisms**

  We will require a variety of methods for verifying the identity of a user or other entity on the network. Authentication mechanisms are means of gaining trust in the identity of a particular party or message, even when communicating across an untrusted network (i.e., where attackers may control the communication links and alter data flowing across them). In the case of authenticating individuals, these mechanisms may involve passwords, challenge/response systems, or even hardware token-based schemes such as smartcards. We may also wish to verify the authenticity (legitimate source and content) of particular messages or data; in this case, digitally signatures and data integrity checksums can be useful.

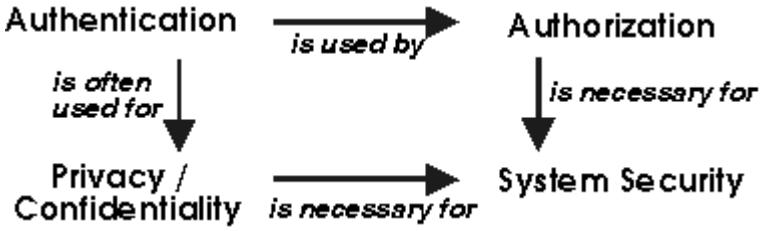
- **Access Control / Authorization Systems**

  Once we know the identity of a party on the network, we may also wish to know what operations are allowed for that entity. Is a particular user authorized to view or even modify a given object on the network? Authorization systems consist both of technologies, such as file permissions, access control lists and digital credentials, as well as human-oriented policies and protocols.

- **Privacy Technologies**

  To ensure the confidentiality of particular data, we need mechanisms for protecting the privacy of communications on the network. Outside of using dedicated, trusted links (i.e., *not* on the Internet), this typically requires that channels, messages or even entire files be encrypted in such a way that the data is only accessible by the intended recipient(s).

The interaction of these components is far from simple, and one depiction of their complex interaction is shown below:



We note that for system security to be effective, we require that the system's authorization policy be implemented (in both its human and technological dimensions), and that data confidentiality also be enforced. Clearly, authorization systems make use of authentication mechanisms, since one cannot determine the allowed actions for a particular individual until the identity of that individual is trusted to some degree.

Interestingly enough, the individual privacy provided to users on a computer or network is not necessarily guaranteed by the successful implementation of the other security components. A system may be perfectly secure, in terms of its system integrity, but be lacking any user privacy at all; consider, for example, the case of the system depicted in George Orwell's 1984 [3]. On the network, when privacy measures are provided they are usually tightly linked to the mechanisms used for authentication. This means that we combine the step of ensuring that data is "for your eyes only" with the process of verifying who "you" are.

We will see that in all but the simplest Web applications, we will require novel schemes for authenticating Web clients and servers, implementing access control over Web documents, and ensuring data privacy during the communications involved. However, for *any* use of

the World Wide Web, we would like to make sure not to expose our computers to new vulnerabilities. So let's first take a look at the system security concerns of using the Web.

System Security Considerations for Using the WWW

The powerful flexibility of both Web browsers and Web servers comes at a price: each introduces a few system security weaknesses. We study these system security concerns for browsers and servers in turn.

Web Browsers

The Web browsers of today are some of the largest, most sophisticated and feature-filled programs on the Internet. The system security threat associated with using these do-it-all applications is that an intruder may try to gain unauthorized access to a browser user's computer by exploiting a security hole in either the browser or the programs used in conjunction with the browser. Remember that installing and configuring a Web server is a very simple process. So it is not unimaginable that even a juvenile prankster might set up a server on the WWW with the sole purpose of attempting to break into the computer of any browser who drops by. Moreover, the attacker's task may be easier still if the browser's system may be compromised merely by persuading the user to view a particular web document or click on a certain hyperlink.

The main security weakness of browser use is not actually a problem with the browser program itself, but rather with its use of external, "helper" applications. These applications, often called "viewers," are used by the browser to handle document types that the browser doesn't know how to present to the user. Examples include "ghostscript" for viewing PostScript files, or a program which plays MPEG movies. Some of these programs may themselves have security holes which can be penetrated when the user attempts to view a particular document obtained from a Web server. For instance, if the "ghostscript" application is not carefully configured for use with the browser, a PostScript document may use it to execute arbitrary commands on the browser's machine.

A particularly severe example of the security problems involved in using external viewers is the fact that browsers may be configured to execute any Unix shell script sent to it by the server. This is an attacker's dream. While in the early days of Web development, this was thought of as a useful feature, currently most browsers have this function turned off as part of their default configuration. However, if a user's "mailcap" file contains the line,

```
application/x-csh;  csh -f %s
```

this hole will be opened up again. System administrators are well advised to take measures to guard against this potential weakness, at least by warning users of its implications (see [4]).

At a more basic level, Web browsers are potentially vulnerable to certain attacks merely

due to the fact that these programs are extremely large. Any software engineer will tell you that whenever a program comprises hundreds of thousands of lines of source code, there are inevitably many bugs in the program, and some of those bugs may give rise to security holes. A classic example of this phenomenon is the recent discovery that most all browsers today do not handle extremely long URL's (longer than 356 characters). These URL's cause a string buffer in the browser to overflow, and the excess data may be put on the computer's stack and executed as program code [5]. This bug is almost identical to the hole in "fingerd" exploited by the Internet worm [6], and it may allow an attacker to execute pre-compiled commands on the target machine.

How does one guard against these vulnerabilities? It is interesting and a bit frightening to note that firewalls will *not* help much, since the security hole is in the client-side handling of document data. In essence, the attacks mentioned above employ a "trojan horse" web document (a PostScript or HTML document, for example) which will pass right through a firewall. So the only real defense for these attacks is to make sure that the browser and viewer applications used on the client's machine are up-to-date and free of all known security holes. In the near future, we will see new "portable code" languages, such as Sun's Java [7], which address the need for securely executing applications obtained on the Web.

Web Servers

Security concerns are even more important on the server side of the Web, since such information servers may be well known and thereby function as particularly inviting targets for attackers. Setting up and maintaining a web server are processes which require a healthy amount of care and knowledge if new security holes are to be avoided.

Fortunately, Web servers are substantially simpler programs than Web browsers, so we may not have to worry as much about security-related bugs due to the large size of the server's program code. Nonetheless, there have been discoveries of bugs in Web servers which constitute security holes. For instance, version 1.3 of the NCSA httpd server exhibited the same "string buffer overflow" bug as described above, allowing knowledgeable attackers to execute arbitrary commands on the server by appending pre-compiled program code to the end of a long URL string [8]. If a Web server is properly configured (i.e., it runs as user "nobody" with minimal privileges), this vulnerability should not, by itself, allow untold damage to be done on the server machine. Nonetheless, sophisticated attackers may use this small hole to open more serious security holes; so naturally, upgrading to a more recent version of this particular Web server is highly recommended.

The most serious security vulnerability of Web servers has to do with its ability to run Common Gateway Interface programs (CGI scripts), and the ease by which one may write a CGI script with security holes of its own. If CGI programs are not carefully written, they may allow malicious client-side attackers to execute arbitrary commands on the Web server host. Frequently, authors of CGI scripts place too much trust in the input provided by the client, and this can lead to the execution of an attacker's shell commands embedded in text strings.

For example, consider a CGI script which sends email to an address specified in an HTML form. The shell script might contain the line,

```
system("/usr/lib/sendmail -t $USER < input_file")
```

for sending the contents of the file, "input_file," to the email address contained in the variable "USER." Now, let's say that a malicious client supplies a value of the USER variable which contains a semicolon followed by a different command. This might look like:

```
USER = " whatever@wherever.com < /dev/null ; rm -rf / "
```

When the server performs the "system" call, the supplied string will be interpreted as two commands: the first sends a blank message to the indicated address, and the second causes some serious problems on the server machine. Here again, the damage is limited by the permissions of the web daemon's user ID (hopefully "nobody"). However the solution to this problem is to be much more paranoid with the client-supplied input. For tips on how to write CGI programs with such attacks in mind, see [9] and [10]. Naturally, the appropriate policy for a Web server administrator is to either not run CGI scripts at all, or to carefully scrutinize them before they are used.

There are many other important precautions to take when setting up a Web server, including:

- Run the server as a special user ("nobody") with limited privileges
  -- *do not run it as **root**!!*
- Turn off server-side includes
- If possible, use a separate machine for web server; minimally configured, outside the firewall
- Keep up with **SAIC** and **CERT** security bulletins
- Use **TCP-wrapper** for improved access control and logging

For more information on these security measures, see [11], [12], and [13].

Security Requirements for Sophisticated Web Use

Assuming we have survived the process of securing the host computers running our browsers and servers, we may now consider some of the other components of Web security with respect to our desired uses of the Web. In addition to the usual fare of publicly browsing and serving Web pages, we would also like to enable the following applications:

- Sharing sensitive information with particular groups of individuals

  This requires remote authentication of client users, in order to perform access control on the server side.

- Exchanging, publishing important information in a trustworthy manner

  When submitting purchase orders or publishing official documents, we would like to know that both the information source is authentic and that the data has not been altered in transit. This not only requires authentication of Web clients and servers, but perhaps also the authentication of messages or documents by means of digital signatures.

- Digital payment: buying and selling goods, services, information

  Electronic payment on the Web may involve authenticating the parties involved in the transaction (particularly the seller). However, it will almost certainly involve validation of the payment itself, perhaps by a third party such as a credit card acquirer.

- Private communications on the Web

  In all of these situations, we would probably either desire or require that the information exchanged on the Web not be visible or modifiable by bandits roaming the untrusted network.

In short, for most sophisticated uses of the World Wide Web, we will require the ability to perform *secure communications* between *authenticated agents*.

Current State of Web Security

With these requirements in mind, we now ask what is achievable given the current standards of the Web. Basically, the situation on the Web today is that of a *Free-For-All*. The identity of browser users is almost always unknown, and one usually can't even be perfectly certain of a Web server's authenticity; if a nearby attacker were to intercept a browser's request for a particular home page and then send back a bogus home page in response, it would be difficult for the browser user to notice the substitution.

The anonymity of Web clients means that server-side access control to Web documents is not easily achieved. And in almost all current Web communications, the information is unprotected in transit; given the relative ease of setting up a "packet sniffer" on the network, this means that privacy is basically nonexistent on the Web.

It is interesting to contrast the current state of security on the Web with that of the modern telephone network. When we use the phone, we can be fairly certain of the identity of an individual on the end of the line, often due to the recognition of his or her voice. But more importantly, we place a lot of trust in the infrastructure itself. When we dial a particular number, it is generally unlikely that the call will be intercepted or monitored by an attacker. Also the directory services provided by the phone system are widely published and considered to be reliable ways of locating a particular party. The situation is much less trustworthy on the Web.

To date, the common use of the World Wide Web has not suffered greatly from these security deficiencies. The Web was designed as a public medium, where practically anyone can provide information for everyone to view. In fact, this is one of the major reasons why it has flourished. It is only recently, with the arrival of Web-based commerce and the publishing of official documents on the Web, that network espionage has become a commercially interesting activity. And consequentially, the need for strong security mechanisms on the Web has gained a growing amount of attention.

Basic Authentication: A Modest Proposal

There is *one* authentication mechanism that has been part of the Web from the start. "Basic Authentication" (sometimes called "User Authentication") was defined in the original specification for the HyperText Transfer Protocol (HTTP), and is therefore supported by all HTTP-compliant Web browsers and servers [14].

Basic Authentication is the familiar username/password form of weak authentication, and it works as follows. The Web server may be configured so that access to particular documents requires a valid username and password to be submitted as part of the client's HTTP request. The server maintains files containing lists of usernames along with the corresponding passwords stored in one-way encrypted form. Similar to the `/etc/passwd` file on Unix machines, the passwords are stored this way so that anyone viewing the username/password files will not be able to deduce the plaintext password from its one-way encrypted form. When the Web server receives a request for a document protected by Basic Authentication, the server one-way encrypts the password supplied in the request, and compares the result against what is stored in the username/password file for the supplied username. If the values match, the user has supplied the correct password and is given access to the document.

A typical HTTP exchange involving the use of Basic Authentication is shown below. The browser initially asks for the document (the first HTTP "GET" request) without any knowledge that it is protected. The server then response with a "401 - Unauthorized", and indicates in the "WWW-Authenticate" header that the client must use Basic Authentication in order to retrieve documents in that area (the "SuperSecret" realm). When the browser receives this response, it prompts the user for a username and password. It then "uuencodes" (a trivially simple scrambling method) the username:password pair and includes the resulting string in the "Authorization" line of the re-submitted request. In this case, the user has provided a correct username/password combination, and so the server then provides the document in its response. Note that the browser keeps track of the Authorization information used, so any subsequent requests for documents in the same directory of the same server will be accompanied by the same username/password information. This is much more convenient than requiring the user to enter a username and password for every document in a protected area.

1. First the browser sends the request without an Authorization line

```
GET /protected/secrets.html HTTP/1.0
Accept: */*
User-Agent: NCSA_Mosaic/2.7
```

2. The server notices that Basic Authentication is required for the requested document and sends back a 401

```
HTTP/1.0 401 Unauthorized
Date: Mon, 16 Oct 1995 04:12:49 GMT
Server: NCSA/1.4.2
Content-type: text/html
WWW-Authenticate: Basic realm="SuperSecret"

<HEAD><TITLE>Authorization Required</TITLE></HEAD>
<BODY><H1>Authorization Required</H1>
Browser not authentication-capable or
authentication failed.
</BODY>
```

3. After prompting the user, the browser resubmits the request, with the username/password in the Authorization line

```
GET /protected/secrets.html HTTP/1.0
Accept: */*
User-Agent: NCSA_Mosaic/2.7
Authorization: Basic Zm51cmc6eWFnZ2EteWfnZ2E=
                        this is the "username:password" string, uuencoded
```

4. The username/password is valid, so the server returns the document

```
HTTP/1.0 200 Document follows
Date: Mon, 16 Oct 1995 04:13:05 GMT
Server: NCSA/1.4.2
Content-type: text/html
Content-length: 227
```

-- Document starts here --

How useful is Basic Authentication? Its main strengths are that it is a very simple scheme, in that users are already accustomed to providing username/password pairs when they login or use applications such as ftp. Also it is worth mentioning that the Web server administrator has full control of who has access to the documents, since each allowed user must be explicitly included in the username/password files. Of course this also means that administration can turn into a nightmare if the list of allowed users is large or rapidly changing (as in the case of a university or perhaps a large corporation). This system can also be problematic if the administrator must manage *several* different web server

machines; keeping all of the username/password files up-to-date and consistent could require a lot of work.

However the most troubling weakness of Basic Authentication is that it is insecure. The passwords are essentially traveling over the network in plaintext, since anyone who has gone through the trouble to set up a packet sniffer for capturing passwords is probably clever enough to know how to use "uudecode." It should be noted, nonetheless, that this problem is shared by the standard "telnet" and "ftp" applications which people use all the time. So Basic Authentication can be thought of as providing a similar level of (in)security as most of the network applications in use today. We'll see how a new scheme, called "Message Digest Authentication," can serve as a simple replacement for Basic Authentication and solves the problem of passwords traveling in the clear.

Using Network Domain for Access Control

Almost all web servers in use today also have the ability to control access to documents based on the client's network address using a method called "IP filtering." To use this mechanism, the server administrator specifies that certain documents may only be viewed by clients connecting from particular network domains. For instance, the administrator for a web server at FooBar University may allow access only to clients on a host whose name end with "`.foobar.edu`". The lines below show examples of the syntax used to specify such an access control scheme, for both the NCSA httpd [15] and the CERN httpd [16] web servers.

```
"allow from .foobar.edu"          (NCSA server)
"* : get, put  : @128.141.*.*"    (CERN server)
```

Note that when a client requests a document protected by this manner, the user is not prompted for a username, password or anything else. The Web server simply looks at the client's IP address and compares it to the masks (like those shown above) specified in its access control configuration files.

This method of access control has a few advantages. It is usually much simpler to administer than Basic Authentication, and it somewhat more secure. However, it is only as secure as DNS (the Domain Name Service), which is not exactly bulletproof. The sophisticated attack of "IP spoofing," which gained some press attention recently, can also circumvent this access control scheme. Yet these vulnerabilities are not as important as the fact that administrators may have difficulty specifying allowed access in terms of particular network domains. Quite often, authorized individuals will be scattered among several domains, each of which may also be accessible by unauthorized parties.

Combining Basic Authentication and IP-Filtering

Most popular Web servers currently support the ability to use both Basic Authentication and IP-Filtering together. This enables a server administrator to, for example, require that users provide a valid password *or* connect from a particular network domain. Alternatively, the user may be required to both enter a valid password *and* be within a permitted domain. See [16] and [17] for details of configuring the server to use both of these mechanisms.

At this point we conclude that the authentication/authorization systems currently available in all Web browsers and servers are somewhat useful, though not what we would like. So far, we have seen only weak forms of authentication (passwords, instead of digital signatures, for instance), and none of these mechanisms provide for privacy in our Web communications -- everything is traveling across the Web in the clear, begging to be captured by eavesdroppers. How can we achieve our requirements of secure communications between authenticated agents? Fortunately, some cryptographic techniques come to our rescue.

Review of Cryptography

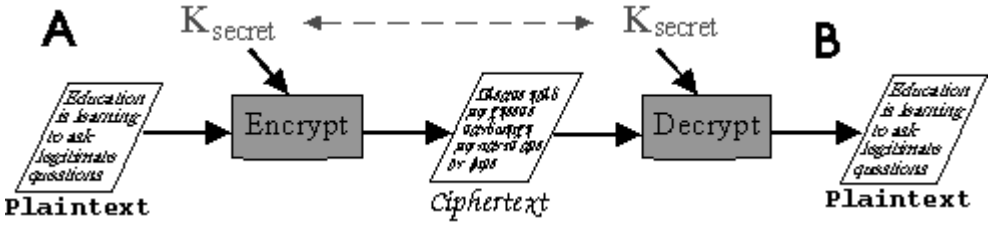
The field of cryptography is one of the richest and most challenging areas of our time. In fact, entire books, and even entire national agencies have been devoted to its study. For our purposes, we only need to understand the basics of cryptographic tools used by the Web security systems surveyed in later sections. Readers already familiar with cryptography may wish to skip to the "Adding Cryptography to the Web" section.

Cryptography is the art of transforming information to and from a form in which it is useless to all but the intended recipient. These transformations are performed using cryptographic *algorithms* in conjunction with particular *keys* held by the sender and receiver. There are a variety of algorithms which can be used to accomplish this, and the choice of algorithm used in a cryptographic scheme is usually not a secret. Rather, it is the possession of the correct key which is the basis of the security provided by these systems. The way in which keys are used in a cryptographic scheme is what distinguishes it as being in one of two main categories of cryptographic systems: symmetric, or asymmetric cryptography.

Symmetric cryptography for privacy

In symmetric cryptography, the same key is used both to *encrypt* and to *decrypt* data. The unencrypted message to be sent is often called the *plaintext*, and this data is encrypted to produce the *cyphertext*. Often, the processes of encrypting and decrypting are called *encyphering* and *decyphering*, respectively. The key is essentially a number, and in symmetric cryptography this key is assumed to be known by both communicating parties, but not to the eavesdropping adversary.

Consider the scheme depicted in the diagram below. Alice ("A") and Bob ("B") wish to communicate securely across an untrusted network. Fortunately, they both have algorithms for performing symmetric encryption and decryption; they also share a secret key, K_{secret}. In this scenario, Alice encrypts the plaintext using the secret key and sends the resulting cyphertext to Bob over the untrusted communication link. Bob may then decrypt the cyphertext using the same secret key and thereby obtain the Alice's plaintext message. Any attacker intercepting the cyphertext in transit will be unable to recover the plaintext without knowledge of K_{secret}.



Symmetric cryptographic schemes have been in use since the days of Julius Caesar. The classic Caesar cipher involved replacing each letter of the plaintext with the letter three away from it in the alphabet. In this case, the secret key is simply "3" and the encryption/decryption algorithms may be performed by hand. Another famous symmetric cryptosystem is the *Enigma* machine used in World War II. These days, computer algorithms such as DES, RC4, and IDEA [18] are used to rapidly encrypt and decrypt data.

The modern algorithms used in symmetric cryptography have been studied for some years, and are generally believed to be effective. However it is important to note that the ultimate security of these systems resides in the unguessibility of the keys used. Thus, the key length is of prime importance, as the attacker must not be easily able to try all possible key values and thereby break the code. Generally, symmetric keys shorter than 40 bits in length are considered to provide minimal security. 128-bit keys are thought to be much more secure, as it would require many *billions* of MIPS-years (years of computation on a 1-MIPS machine) to try all possible keys. For more information on symmetric cryptography systems, see [19] and [20].

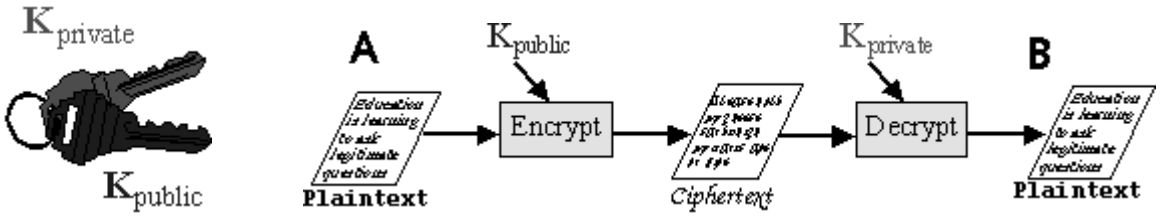
With all the speed and usefulness of symmetric cryptosystems, there is one major problem: how do the communicating parties get the secret key in the first place? After all, if they have a secure channel for exchanging secret keys, why not use that channel for communicating the secret message as well? Fortunately, public-key cryptography addresses this problem.

Public-key cryptography for signatures and spontaneous exchanges

In asymmetric cryptography, often called "public-key cryptography," each user has a unique *key pair* consisting of a *private key* and a *public key*. These keys are actually numbers which are mathematically related in particular ways. The important feature is that a user's private key is to be kept completely secret to the user, while the public key may be, and *should* be, widely known. And despite the mathematical relationship between the two keys, it must be impossible, or at least *computationally infeasible*, to determine the private key given the corresponding public key.

Public-key cryptographic algorithms have the amazing property that anything encrypted with one key can *only* be decrypted using the other key of the pair. This is a most unusual characteristic, and it enables some interesting applications. For starters, it allows two parties to communicate securely without the need for pre-arranged secret keys.

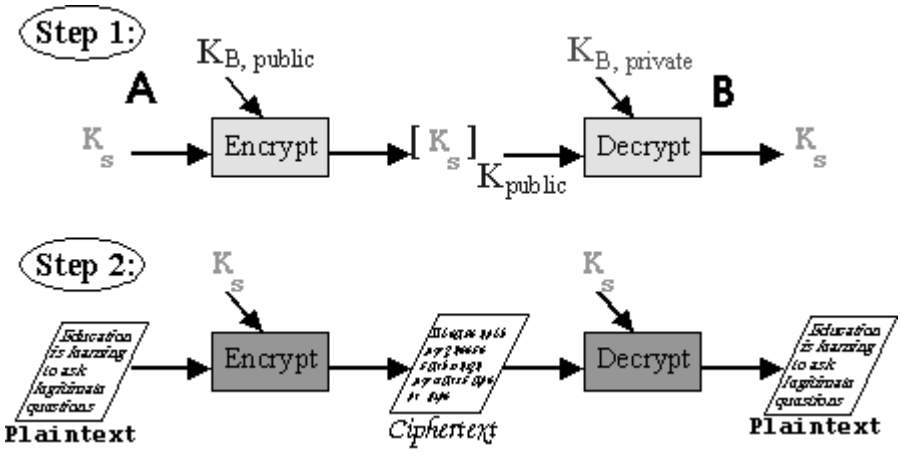
Consider the scenario depicted in the diagram below. Alice knows Bob's public key, and may use it to encrypt a secret message for him. The resulting cyphertext may then only be decrypted using Bob's private key, which is presumably known exclusively to Bob.



The most popular asymmetric cryptography scheme involves use of the "RSA algorithm" developed by Rivest, Shamir and Adleman. The RSA algorithm involves modular exponentiation of very large numbers (roughly 500 to 1000 bits long), which is substantially slower than symmetric algorithms. The computational infeasibility of finding the private key given the public key depends on the ease at which one may multiply two large numbers together, compared to the difficulty involved in going the reverse direction (starting with one large number and factoring it into two other large numbers).

Public-key Cryptography for Key exchange

Since asymmetric cryptography is so much slower than symmetric algorithms (by a couple orders of magnitude), the scheme depicted above is almost never used for data encryption in practice. Instead, a symmetric key is exchanged using public-key cryptography, and then that symmetric key, or *session key*, is used to symmetrically encrypt and decrypt the data. This combination of asymmetric and symmetric encryption works as depicted in the figure below.

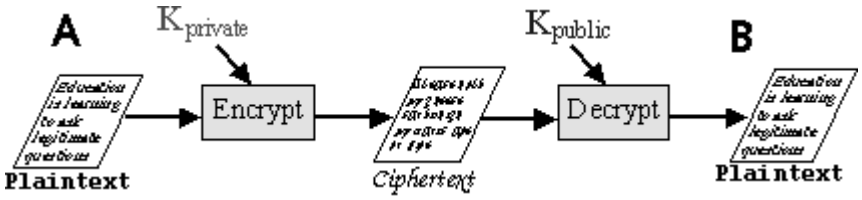


In Step 1, Alice generates a random session key, K_s. It is important this be a truly random number in the sense that an attacker monitoring Alice's activities should not be able guess the value of K_s. Alice then encrypts the session key using Bob's public key and sends the resulting cyphertext to Bob, who uses his private key to decrypt it and recover K_s, the session key. Now that Alice and Bob have securely agreed upon a secret session key, they may use it with a symmetric encryption/decryption algorithm, as shown in Step 2.

This system combines the speed of symmetric cryptographic algorithms with the ability for spontaneous secure communications provided by public-key cryptography. Note that it only requires one of the two parties to have an asymmetric key pair (Bob, in this example). Successful completion of this two-step process also serves to authenticate Bob. If the receiving party in Step 1 is able to successfully recover the session key, K_s, then that party must hold Bob's private key, and thus that party must be Bob.

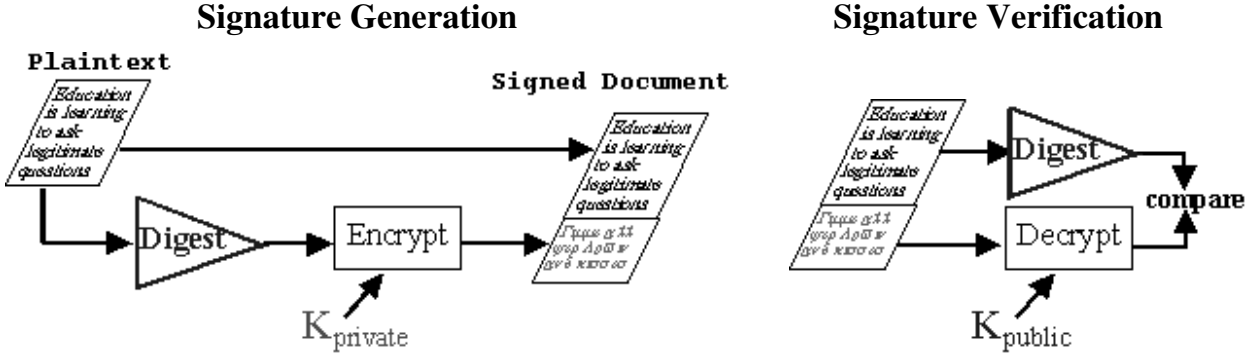
Digital signatures:

Using asymmetric cryptography becomes even more interesting if we reverse the roles of the public and private keys. Consider the scheme depicted in the diagram below. Alice encrypts the plaintext using her *private* key, instead of her or Bob's public key as in the examples above. The resulting cyphertext is by no means private, because anyone who knows Alice's public key (which presumably *everyone* knows) may decrypt the message. However, since Alice is the only one who holds her private key, only Alice could have created such a document. In this way, Alice can prove that a particular message came from her and her alone. Hence, we have a type of *digital signature*.



As in the case of using public-key cryptography for private communications, asymmetrically encrypting an entire document is tremendously slow. So what is usually done in practice involves encrypting a hash value which uniquely identifies the document, and then appending the encrypted hash to the document, forming a "signed document." This scheme is depicted below, and works as follows:

The signing party uses a *message digest function* to compute a *one-way hash* of the entire document. The resulting hash value is on the order of 128 bits long, and it uniquely represents the document in that it is computationally infeasible for an attacker to find two documents which produce the same hash result when put through a message digest function. This is the "one-way" property of the function. Given the output of the message digest function, it is practically impossible to alter the document, or find another document, such that the hash value is the same.



To produce a digital signature, the signing party encrypts the document's hash value using the author's private key. The resulting cyphertext is appended to the document and the new signed document is distributed. Anyone wishing to verify the signature on a signed document first finds the author's public key and then uses it to decrypt the hash value. If this hash value matches the result of running the received document through the message digest function, then we know that the document is authentic and has not been altered since it was signed.

Along with public-key cryptography's marvelous gift of enabling spontaneous secure communications and digital signatures, comes one complication: how do we know that a particular public key belongs to a particular individual? If we cannot trust that a public key corresponds to a particular author, for example, digital signatures verified using that public key become worthless. Consider the case in which an attacker wishes to impersonate Bob. The attacker might generate his own key pair and then broadcast his public key as belonging to Bob. If Alice then encrypts a message for Bob using the attacker's public key, the attacker has thwarted our security system altogether. To address the need for trust in the ownership of a public key, we may use *digital certificates*.

Digital Certificates:

A digital certificate is essentially a digitally signed data structure which can be used to vouch for the ownership of a particular public key. The entity which vouches for this assertion by signing the certificate is called the *Certification Authority* (CA). To illustrate the use of digital certificates by means of example, let's say that Bob wishes to have the U.S. Postal Service issue him a certificate. Bob would probably have to walk into the Post Office, give them his public key and probably show them several forms of identification (perhaps a driver's license or birth certificate) to prove that he is indeed Bob. Once the Post Office is convinced of Bob's identity, they prepare a data structure containing, at least, the following information:

- **User Name** Bob's name, plus perhaps his address, organizational affiliation, etc.
- **User's Public Key** A numerical representation of Bob's public key
- **Name of CA** In this case, the U.S.P.S. is issuing the certificate
- **Other attributes** perhaps the policy used by the CA to verify the user's identity

This data structure is then digitally signed using the private key of the Certification Authority (the U.S.P.S., in this example). Bob walks away with the digital certificate and uses it in subsequent communications to prove that the U.S.P.S. asserts that the public key contained in the certificate belongs to Bob.

Obviously, anyone wishing to verify the signature on the digital certificate must possess and trust the public key of the Certification Authority. Does this mean we might have Certification Authorities issuing certificates to other Certification Authorities in order to vouch for the latter's public key? Unfortunately, it probably does. If two parties do not trust each other's public keys, there may be *chains* of certificates to be verified before the parties can communicate in a trustworthy fashion. This suggests the need for a hierarchy of Certification Authorities, with an all-trusted public key owned by the CA at the top. For this to work in practice, computer networks will need online services for looking up public keys and digital certificates. This sort of *Public-Key Certificate Infrastructure* might behave similar to the Domain Name Services (DNS) of today's Internet. However the infrastructure has not been built yet, although there are currently several different design proposals and even testbed implementations. The handling of digital certificates online, and the organization of a Public-Key Certificate Infrastructure is sure to be a contentious issue in the near future.

Adding Cryptographic Features to the Web

Now that we have the tools for securing communications and authenticating agents, what's keeping us from using them on the Web? There are many reasons why integrating cryptography with the Web is a difficult task, including the following:

- This requires extending HTTP, HTML and perhaps even proposing new standards
- Web browsers and servers must be augmented with cryptographic features
- There is a large variety of cryptographic algorithms, hardware, and software to support. We cannot limit ourselves to one encryption algorithm, for example, because it may be proven to have a severe weakness.
- Several countries have export/import restrictions on cryptographic technologies, complicating the task of software distribution and collaborative development.
- There are network infrastructure components (certification authorities, key management systems, etc.) which must be in place before we have truly scalable solutions.
- Web security is but one piece of the larger puzzle of network/system security (i.e., we want all of our network security mechanisms to work together).

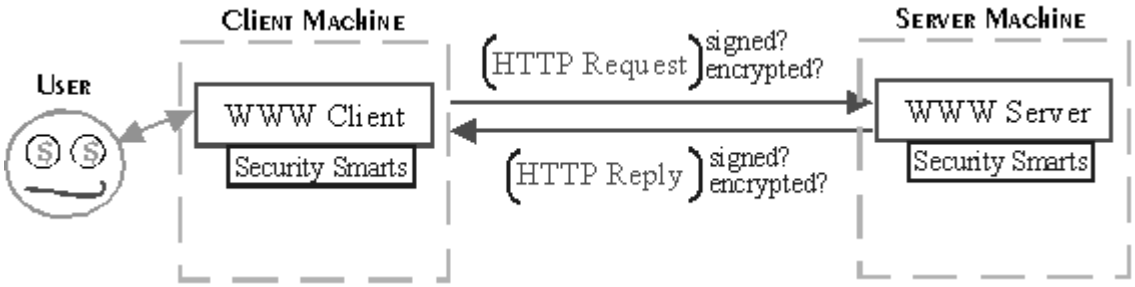
Nonetheless this hasn't stopped a large number of institutions from jumping into the game of putting cryptography into the Web. In fact, there are so many different schemes offered for solving the Web security problem that we cannot hope to cover all of them in this document.

Why are we doing this?

At this point, it is useful to review why we wish to incorporate cryptography into the Web in the first place. Looking back at the "Security Requirements for Sophisticated Web Use" section, we notice that most all of our security needs are achieved if we are able to use:

- Digital Signatures to authenticate client HTTP requests
- Digitally signed HTTP server responses for authenticating server
- Signed documents to verify author, prevent document forgery
- Public and symmetric-key cryptography for privacy (encrypted HTTP messages)

To accomplish this, we may use an architecture like the one depicted below. This scheme is sometimes called "application layer security," in which the web browser and server applications have built-in cryptographic mechanisms (the "Security Smarts") which are used to enhance the HTTP messages exchanged on the Web.



We will see that this is not the only possible approach to adding security to the Web, however it adequately describes the architecture employed by the first several of the Web security schemes surveyed in this document. The important feature of this scheme is that the HTTP messages themselves contain components which are digitally signed and/or encrypted.

PGP/PEM Authentication, Encryption in NCSA Mosaic/httpd

To demonstrate how the encryption/signing of HTTP messages may be accomplished, NCSA XMosaic and httpd were augmented with features which allowed them to run external programs to handle message encryption, signing, etc. We explain these "hooks" (simple calls to external applications) to PGP and PEM now, for both historical and expository reasons. Once the simple PGP/PEM scheme is understood, it will be easier to understand how S-HTTP works.

PGP stands for "Pretty Good Privacy," and is a general-purpose cryptographic package written by Phillip Zimmerman [21], [22]. Privacy-Enhanced Mail, or PEM [23], is a similar program which resulted from efforts of the IETF (Internet Engineering Task Force). Both are stand-alone applications which are widely available and can be used for digitally signing documents, encrypting data using symmetric or public-key cryptography, and managing public keys and digital certificates.

The main idea behind the PGP/PEM hooks in XMosaic, httpd is that many users will already have one of these programs installed, for use with protecting email messages, netnews postings, and the like. So why not have the Web browser and server simply use these applications to handle the security enhancement of HTTP messages? This is an easy way of allowing for signed, encrypted HTTP messages without adding all the necessary cryptographic functions directly to the Web applications.

Mechanics of PGP/PEM hooks

The way the PGP/PEM-enhanced Web communications work is illustrated in the HTTP trace below, and may be explained as follows. First, assume that the browser user and the web server in question both have asymmetric key pairs, and that each knows and trusts the other's public key (this is a *big* assumption which we will discuss later). The server administrator specifies that a particular document directory can only be accessed by clients who authenticate themselves by digitally signing (and also encrypting) their HTTP request. When the server receives a normal, unsigned request for such a document it sends back a "401- Unauthorized" message to the client, indicating that the browser must sign and encrypt the HTTP request using PGP, for example. In the scheme depicted below, the server also tells the client, via the WWW-Authenticate line, the PGP username of the server. The client then sends the request to PGP, which uses the public key of that username to encrypt the HTTP request, after it has first digitally signed it using the user's private key. The signed, encrypted HTTP request is then encapsulated within the body of *another* HTTP request by means of adding a few generic HTTP headers to the signed,

encrypted data. This new request is sent to the server which may then use PGP to unencrypt the encapsulated request and verify the client's signature (using the client user's public key). Assuming the new request is acceptable, the server prepares a HTTP response and this is likely to be a "200" message containing the requested document. The server gives this message to PGP, which signs it with the server's private key and then encrypts the result with the client's public key. This signed, encrypted response is then encapsulated within another, generic HTTP response and sent back to the client. Finally, the browser uses PGP to decrypt the encapsulated message, and verify the server's signature. The resulting, plaintext HTTP message may then be processed by the browser (usually this will involve displaying the requested document). [24]

---

1. First the browser sends the request without PGP enhancements

```
GET /pgp_protected/index.html HTTP/1.0
Accept: */*
User-Agent: Mosaic/X 2.4
```

---

2. The server notices that PGP Authentication, Encryption is required for the requested document and sends back a 401

```
HTTP/1.0 401 Unauthorized
Date: Mon, 16 Oct 1995 04:12:49 GMT
Server: NCSA/1.4.2
Content-type: text/html
WWW-Authenticate: PGP entity="web_server@paranoid.ncsa.uiuc.edu"
```

-- HTML for 401 error message here --

---

3. The browser uses PGP to sign the HTTP request (with the client's private key) and encrypt the resulting data (with the server's public key)

```
GET / HTTP/1.0
Accept: */*
Authorization: PGP entity="nick_danger@ncsa.uiuc.edu"
Content-type: application/x-www-pgp-request

-----BEGIN PGP MESSAGE-----
Version: 2.6.2

LYtqNDjzttho36gowAh7xhoosymZigJKOWwK33EE3HhG123+9s2Zp/
IIaOZip  this is the real HTTP request, signed and encrypted  I3Vc0DNi
mx4GS3iJHK3JP2NY1M1UydxUoShcIIsx1YGlltIkkahyyETCiPTYw5
-----END PGP MESSAGE-----
```

4. After decrypting and verifying the signed request, the server returns the document, digitally signed (using its private key) and encrypted (with the client's public key)

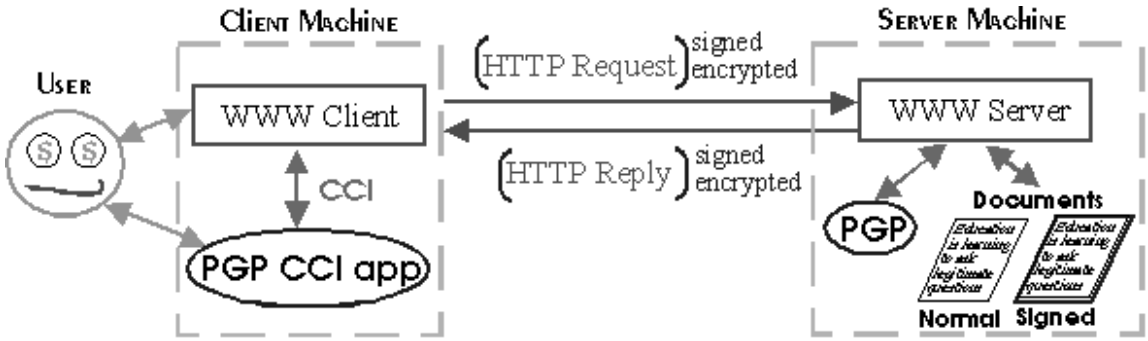
```
HTTP/1.0 200 Document follows
Server: NCSA/1.4.2
Content-type: application/x-www-pgp-reply
Content-length: 227

-----BEGIN PGP MESSAGE-----
Version: 2.6.2

jDqcVMsqo9RT3OGSa5pHDu3+dOmHAqs561CZqVKg4PV1xxIKa8WErD
IIaOZip   this is the real HTTP response, signed and encrypted   I3Vc0DN
mx4GSMoent1d8eGBeKvRSVPpFHKuMZ5fdlGllsdfHGUIE3sf0SRC1H
[etc.]
-----END PGP MESSAGE-----
```

The main feature to understand about this scheme is the way in which HTTP messages are signed, encrypted, and then encapsulated in another HTTP message. This is done by prepending a few HTTP headers to the signed, encrypted data, so that the recipient of the message knows how to "unwrap" the security-enhanced data in the message body.

Status and Future of PGP/PEM Scheme

The initial implementation of the PGP/PEM hooks in Mosaic and httpd was largely a "proof of concept." It suffered from a few flaws, such as requiring that public keys be exchanged between the Web server and browser in advance, and also requiring that each user's pass-phrase (the sacred string allowing access to the user's private key) be stored *in cleartext* within shell scripts used to interface Mosaic/httpd with PGP or PEM. Consequentially, these features did not become very widely used.

Support for PGP, and to a lesser extent PEM, in Web browsers and serves is being revised currently. The new implementation addresses the problems mentioned above, and uses a general-purpose browser interface (CCI, the Common Client Interface) to free the browser of crypto-specific features and also make PGP support easier to add to various Web browsers. This novel scheme is depicted below.



In this design, the signing and encrypting of HTTP messages is much the same as before. However, the browser now communicates with PGP by means of the Common Client Interface. Essentially, the "PGP CCI app" is a separate application which serves as a user-friendly front end to PGP and handles all of the details of message signing, encryption, etc. The PGP CCI app may also be called upon to pre-encrypt a HTTP request before it is sent at all. To do this, it must know to use the correct public key for that server, and it obtains information from special attributes in the hyperlink for the requested document. For example, an HTML anchor corresponding to a document protected by this new PGP-CCI scheme might look as follows:

```
<A HREF="http://www.topsecret.org/the-plans.html"
  PGPPUBKEYID="E9B2BB1D"
  PGPUSER="Topsecret Webserver <web-admin@topsecret.org>"
> Click here to see the secret plans! </A>
```

It is important to note that in order to pre-encrypt a request, information such as the server's identity must be known, either from the hyperlink or by some other means.

The use of a PGP-CCI application has the additional advantage of enabling the elegant handling of signed/encrypted *documents*, as opposed to HTTP messages. By this we mean that the author of the HTML document might use PGP to digitally sign the entire document before depositing it on the Web server for distribution. Then, when the browser obtains such a signed document, it uses the PGP CCI app to verify the signature and report the security status of the document (i.e., "signed by web-admin@topsecret.org") to the user while the browser displays the plaintext HTML document. This is an important type of application, because there are many cases where we may not care at all from which Web server we retrieved a particular document; our only concern is that the document bears the official signature of a particular entity. To learn more about the PGP-CCI scheme, see [25].

The ultimate utility of using PGP or PEM with Web browsers and servers is difficult to judge. It is certainly the case that using external cryptography programs, such as PGP or PEM, is somewhat slower than if the crypto-related features were implemented in the Web application itself. So we do not gain anything in terms of performance using this scheme. However, it does address the fact that programs like PGP are already being used with other applications, such as email, and it would be nice if users did not have to have separate security solutions for each application. It may also be advantageous to let external programs handle the complicated task of encryption, signing, and public key management, allowing the browser to focus exclusively on communicating on the Web. On the other hand, security smarts fully integrated into the browser will probably be easier to use.

A much more sophisticated way of dealing with encapsulated, security-enhanced HTTP messages is found in S-HTTP, to which we now turn our attention.

EIT's Secure-HTTP

Secure-HTTP [26], defined by Allan Schiffman and Eric Rescorla of Enterprise Integration Technologies (EIT), is a security-enhanced version of HTTP. Similar to the PGP/PEM scheme described above, S-HTTP exchanges may be signed and/or encrypted in many different ways. In addition to specifying a variety of mechanisms for encryption, key exchange, and authentication in Web messages, S-HTTP also defines ways in which browsers and servers may negotiate what security enhancements are required or supported. For example, the client may require that the server's response be encrypted; similarly, the server may require that a client's request be digitally signed, in order to authenticate the browser user.

The protocol specification for S-HTTP [27] defines support for several different cryptographic algorithms, including DES, RC2, RC4, and IDEA for symmetric encryption, RSA and DSS for public-key encryption (for key exchange), and MD2, MD5 or SHS for computing message digests. The algorithms supported or required by the client and server are also negotiable through S-HTTP.

How Does S-HTTP Work?

To fully understand the detailed mechanics of S-HTTP, one must endure a read-through of the specification itself [27]. Many new, S-HTTP-specific headers are defined, along with a few new HTML tags, hyperlink anchor attributes and HTTP methods. To gain a high-level understanding of how S-HTTP works, we describe a few of the new definitions.

- New protocol designator in anchor: `shttp`

  This means that any documents to be retrieved using S-HTTP must have URL's of the form "**`shttp:`**`//www.wherever.com/whatever.html`", rather than "**`http:`**`//www.wherever.com/whatever.html`".

- New protocol method: `Secure`

  The first line of a S-HTTP request is "`Secure * Secure-HTTP/1.1`", instead of "GET /whatever.html HTTP/1.0", as in the case of a standard HTTP Get request.

- Hyperlink anchors indicating server identity, required cryptographic parameters, etc.

  Similar to the PGP-CCI above, S-HTTP anchor attributes mat specify the certified name (CN) of the server, the cryptographic options specific to the given URL, and even nonce values (random numbers) to be used when preparing a S-HTTP request. For instance, the anchor below specifies that the client must encrypt the request using the public key corresponding to the "Secret Server" entity.

```
<A HREF="shttp://www.reallysecret.org/nothing_much.html"
   DN="CN=Secret Server, OU=Persona Certificate,
       O=RSA Data Security, Inc., C=US"
   CRYPTOPTS="SHTTP-Privacy-Enhancements: recv-required=encrypt"
> You don't really want to click here. </A>
```

- New HTML elements: CERTS, CRYPTOPTS

  The server's digital certificate (or even several certificates) may be embedded in the HTML document using the <CERTS> tag. This is useful in allowing the client to determine if the server's public key is trusted before using it to prepare a S-HTTP request. An embedded certificate stored in "PKCS-7" format would look something like what is shown below.

```
<CERTS FMT=PKCS-7>
MIAGCSqGSIb3DQEHAqCAMIACQAExADCEw0BBwEAAKCAMIIREgRGF0yXi/2
VatCFcmiin8RBnojLvhXWAU/CBkwfcR+RAAANVAbsTJUC1GMLCM1cAEEUM
cVMsqo9RT3OGSa5pHDu3+dOmHAqs561CZqVKg4PV1xxIKa89RTrB3OGSa5
[etc.]
</CERTS>
```

  The CRYPTOPTS tag may be used to define default cryptographic parameters for hyperlinks in a particular HTML document.

S-HTTP also defines ways of using pre-arranged keys (by means of "shared secret" strings exchanged out-of-band), as well as performing key exchange using Kerberos tickets, which are briefly described later. In all, S-HTTP provides a great deal of flexibility in the security enhancements employed for a given message exchange.

It is important to understand that despite the complexity of S-HTTP, the basic idea is still much the same as with the PGP/PEM scheme described above. Essentially, a normal HTTP message (with a few added S-HTTP related headers) is encrypted and/or signed, and then it is encapsulated within another message, an S-HTTP message. The S-HTTP headers added to the security-enhanced, encapsulated content give the recipient just enough information so that the message may be unwrapped. These headers include the ones given below:

```
Content-Privacy-Domain: PEM, PKCS-7, or PGP
Content-Transfer-Encoding: 8-bit, Base64, or 7-bit
Prearranged-Key-Info:  ....
```

When the final S-HTTP message is put together, it looks something like the S-HTTP "Get" request shown below:

```
Secure * Secure-HTTP/1.1
Content-Privacy-Domain: PKCS-7
Content-Transfer-Encoding: 7BIT

hGwDYXYcVtJTtGkBAwCMPi/JMfueTMManx3tGZs9pmeGMQw7FOjOvfX
YriB3DWkd8w    this is the actual HTTP request, encrypted    7E9aFYct
DD42hvRtOn/TLYtqNDjzttho36gowAh7xhoosymZigJKOWwK33EE3H5
```

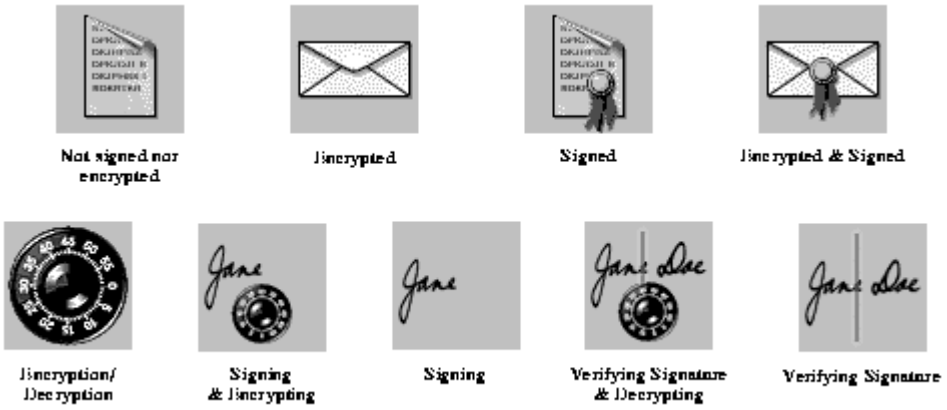
Secure NCSA Mosaic/httpd the "reference implementation"

After defining the S-HTTP specification, reference implementations of an S-HTTP-compatible Web browser and server were made using NCSA Mosaic and httpd. These programs show what user-interface features can be helpful for S-HTTP-capable applications.

Secure NCSA Mosaic looks a lot like the standard NCSA Mosaic browser, although there is an extra icon next to the familiar spinning globe. This new icon indicates the security status of the document currently in view, and this may correspond to one of four values:

- **plaintext:** the document was obtained without the use of any security enhancements
- **signed:** the document was signed by the S-HTTP server, but not encrypted
- **encrypted:** the document was sent encrypted, but not signed
- **signed and encrypted:** the server both signed and encrypted the document

The figure below shows the icon for each of these values. Also, when the browser is in the process of encrypting, signing, decrypting, or verifying signed data, the icon displays corresponding graphics to inform the user of its activity.



If the user clicks on the security status icon, additional S-HTTP-related details will be displayed in a pop-up window. For instance, if the document was signed, the window will display the S-HTTP server's distinguished name, as well as the algorithms used for signing the document (e.g., MD5 with RSA).

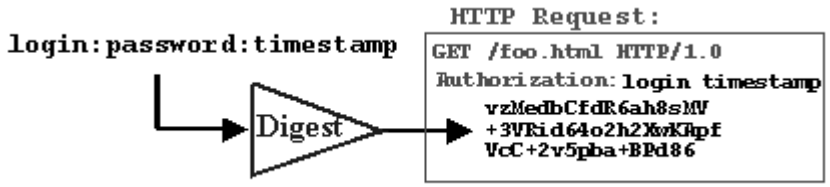
There are other interesting security-related user interface features in the S-HTTP reference implementations. For instance, when a user must digitally sign a request for a S-HTTP protected document, a pop-up window prompts the user for the choice of public key pair, and the corresponding digital certificate, to use to compute the signature. There are also menu items which the user may select in order to add or remove public keys for Certification Authorities; in this way, the user may choose which CA she trusts. For more information on the features and operation of Secure Mosaic and Secure httpd, see [28] and [29].

While we're on the application level, we next discuss two simpler schemes for Web authentication in a stronger fashion than what is provided by Basic Authentication.

Message Digest Authentication

Message Digest Authentication is intended to be a simple replacement for Basic Authentication, with the main improvement being that passwords are not sent in cleartext over the network.

Suppose that a browser user and a server administrator have agreed upon a secret password in advance. The server administrator then configures the Web server such that access to particular documents requires that the client prove that she knows the secret password. We may use Message Digest (one-way hash) functions to allow the browser user to prove this knowledge without sending the password across the network at all. This may be done in a manner similar to the scheme depicted in the diagram below:



The basic idea of this scheme is that the browser creates a string containing the username (login), the user's password, and a timestamp used to detect replays of the request. This string is then sent through a Message Digest function, producing a digest value (128 bits in length) which is converted to a text representation. The browser includes the digest string, along with the login and timestamp, in the Authorization line of the request. The server may look up the user's password in the server's configuration files, and then perform the same message digest computation. If the server's digest value matches the one submitted with the client's request, then the browser must have used the correct password, and so the client is granted access to the document. Note that the one-way property of the message digest function implies that any eavesdropper on the network who intercepts the request will not be able to deduce the user's password using the Authorization string. And if the attacker tries to replay the client's request, the server will detect this attack by noticing that the timestamp has already been used in a recent request.

The mechanics of the actual protocol are slightly more complicated. Similar to Basic Authentication, when the browser asks for the protected document without any Authorization information, the server sends back a "401 - Unauthorized" message with the WWW-Authenticate line containing a field which tells the browser to use Message Digest Authentication. This WWW-Authenticate line also contains a nonce value (random number) provided by the server, which essentially functions as a timestamp in that it allows for the detection of replayed requests. The browser concatenates the login, password, nonce value, requested URL, and some other information to form the string that is sent

through the digest function. The behavior of the server is also a bit more detailed. To learn more about the exact mechanics of Message Digest Authentication, see [30].

What are the relative merits of this simple Web security system? Its main strength is indeed its simplicity, as it does provide a fairly direct replacement for Basic Authentication. Its exclusive use of Message Digest functions is advantageous in that such algorithms are non-proprietary and not export-controlled (as opposed to security schemes which employ data encryption). This makes the distribution of software supporting Message Digest Authentication much easier.

There is one minor complication involved in this scheme. The server password files used in this authentication system must not be world readable, as they contain sufficient information for someone to impersonate an authorized user. The passwords in these files are not encrypted, as in Basic Authentication, because the server must be able to compute the same sort of digest computation as the client. These passwords aren't stored in the clear either, which makes the situation a bit less worrisome. However, these files must only be readable by the web daemon, and other users authorized to change their contents.

Kerberized Mosaic/httpd

Kerberos is a network security system which was developed at MIT and has been in use for some years now. It is a rather complicated system, and we can only hope to outline some of its fundamental characteristics in this document. Interested readers are encouraged to see [31] for a more complete introduction to Kerberos.

The Basics of Kerberos

Kerberos is what is known as a "trusted third party" security system and it is best suited for protecting network communications within a relatively small network environment known as a Kerberos "realm." Using Kerberos-compatible applications, clients and servers within a realm may mutually authenticate as well as establish an encrypted channel between them, even across an untrusted network. The system works as follows:

All users and network services in a particular realm have a unique secret key which is known only to themselves and to the realm's trusted third party, called "kerberos." This kerberos server maintains a database of all the realm's secret keys. When a client wishes to use a network service within the realm, the client must obtain a kerberos "ticket" for that service. This ticket is actually a cryptographically-protected data structure, and it is sent from the client to the server when requesting a service. In order to obtain tickets for network services, the client must authenticate herself to the realm's kerberos server. Usually a user does this when initially logging into the network; if the user provides the correct secret key (a function of the user's password), she receives a "Ticket-Granting Ticket" (TGT) which may be used to obtain tickets for other network services.

The Kerberos protocol used for authenticating clients and servers using these tickets is somewhat complex. Yet, since all of these operations are handled by Kerberos libraries built into the "kerberized" network applications, the user need not be concerned about their mechanics. The important feature is that a kerberized client application may use these tickets to authenticate the user to the kerberized server applications, and that the server may then authenticate itself to the client. Also, the kerberos ticket contains a session key which may be used by the client and server to set up an encrypted channel.

Kerberos has been under development for over 10 years, and has been widely used and tested. There are currently two versions of Kerberos in use today: V4 and V5 (the most recent). Unfortunately, they are only moderately compatible. Both versions are based entirely on symmetric cryptography (using DES), although there are several proposals for augmenting Kerberos with public-key cryptographic features.

Kerberos and the Web

Particularly for institutions that are already using Kerberos for all of their network applications, it would be nice to be able to integrate Web browsers and servers into the Kerberos environment. In such a scheme, the kerberized Web browser sends over a ticket for the kerberized web server as a part of the HTTP request. The server can then respond with its part of the mutual authentication process. Thus, we may use this system to authenticate Web clients and servers in a very simple fashion. Assuming the server administrator has configured the Web server so that Kerberos authentication is required to access certain documents (see [32] for details), the protocol works as follows:

---

1. The browser sends the request without an Authorization line

```
GET /krb_protected/poems.html HTTP/1.0
Accept: */*
User-Agent: NCSA_Mosaic/2.7
```

---

2. The server notices that Kerberos (V4) Authentication is required for the requested document and sends back a 401

```
HTTP/1.0 401 Unauthorized
Date: Mon, 16 Oct 1995 04:12:49 GMT
Server: NCSA/1.4.2
Content-type: text/html
WWW-Authenticate: KerberosV4
```

-- HTML for 401 error message here --

---

3. After obtaining a Kerberos ticket for the Web server, the browser resubmits the request, with the ticket hex-encoded in the Authorization line

```
GET /krb_protected/poems.html HTTP/1.0
Accept: */*
User-Agent: NCSA_Mosaic/2.7
Authorization: KerberosV4 acain 0504003e4fc3e031eba696b4e42b19a33b18d
                                47ce  this is the hex-encoded kerberos ticket  f14a
                                c80fce026ff9352dba13c0246600e4d9571
```

---

4. The Kerberos ticket is valid, so the server returns the document along with a WWW-Authenticate header containing the server's authenticator

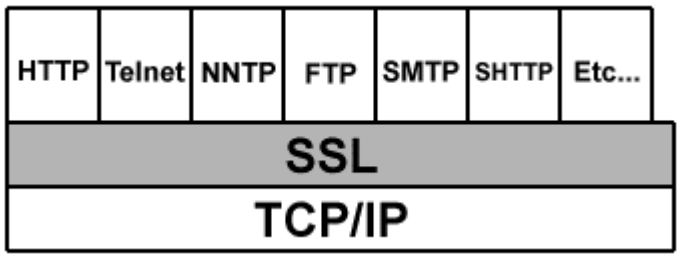
```
HTTP/1.0 200 Document follows
Date: Mon, 16 Oct 1995 04:13:05 GMT
Server: NCSA/1.4.2
Content-type: text/html
Content-length: 624
```
WWW-Authenticate: KerberosV4 [c3602905a92b683f] User authenticated

-- Document starts here --

In the server's second response, the cyphertext field of the WWW-Authenticate line is the message which authenticates the server to the client. Note that this is a very simple way of including Kerberos ticket exchange in HTTP messages. We could also use the session key in the ticket to enable encryption of the client request and the server response. This functionality is being developed currently. See [33] for recent information regarding the status and future of Kerberized Web applications.

Netscape's Secure Sockets Layer (SSL)

An alternative approach to improving security on the Web (and on the Internet in general) is offered by the Secure Sockets Layer (SSL) proposal [34], developed by Kipp Hickman and Taher Elgamal at Netscape Communications Corporation. The SSL design is also aimed at providing secure communications between authenticated agents, although it provides these services below the application layer, as indicated in the diagram below:



The main idea of SSL is that instead of the application (Web-specific, or otherwise) cryptographically protecting messages sent over the untrusted channel, the application may use a special set of socket routines which establish a secure channel through which any data may safely travel. To create a secure channel, the SSL routines use public-key cryptography for authenticating the communicating agents, as well as for exchanging a session key. Thereafter, the session key is used to symmetrically encrypt all communications on this connection. The basic SSL *hand-shake* protocol used to accomplish this is functionally identical to the protocol described in the "Public-key Cryptography for Key Exchange" section above, and works as follows:

1. The client sends a "hello" message to the server
2. The server sends over a digital certificate (which includes the server's public key)
3. The client creates a random session key and sends it encrypted using server's public key
4. After the necessary acknowledgment messages, the session key is used to encrypt all communications (using DES, RC2, RC4 or IDEA) sent in either direction thereafter

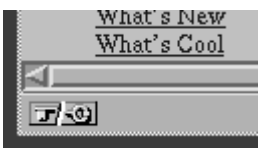
Note that this authenticates the server, since the server's private key must be used to recover the session key. The SSL specification also defines a protocol which can be used for authenticating both the client and the server at the point when the secure channel is being created.

SSL and the Web

The design of SSL makes it fairly easy to support in Web browsers and servers, as well as other network applications like ftp, telnet or net-news (NNTP). However, the browsers and servers must have some way of knowing when they are supposed to use SSL routines to retrieve a document, instead of the normal socket routines. To accomplish this, Netscape has proposed that URL's containing "`https`" in their protocol field be considered to correspond to documents served by SSL-capable servers. Thus, a hyperlink for such a document might look like "`https://www.verysecret.org/whatever.html`" and not have any other anchor attributes associated with it (the SSL routines figure out who the server is from the servers certificate during the hand-shake phase). Also, the SSL Web server must be running on a particular port used only for SSL HTTP services. Netscape has defined that 443 be the default port for this purpose.

To see how SSL may be used in Web browsers, we may consider its implementation in the Netscape Navigator (version 1.1). The Navigator reports the document security status in a few ways. There is a key icon at the lower left corner of the viewing area; when viewing a document retrieved without SSL, this key icon is broken and has a gray background; for a document obtained using SSL, the key is unbroken and the background is blue (see below).

<u>normal document:</u>



<u>doc received via SSL:</u>

There is also a "security status bar" just above the document viewing area. For documents retrieved using SSL, this thin line is blue, whereas it is gray for documents obtained without SSL. Additionally, the user may select the "Document Information" item in the "File" menu, and this pops up a window which, in the case of a document obtained using SSL, displays such information as the distinguished name of the server accessed, and the encryption parameters used in the SSL connection (for example, the size of the session key).

In the version 1.1 implementation of the Navigator, there is no facility for adding or removing Certification Authorities' public keys. In effect, there are a handful of public keys *hard-wired* into the client executable, which are considered to be the only trusted CA keys. Also, there is no support for client authentication in the version 1.1 implementation. Both of these shortcomings are to be addressed in the 2.0 releases of the Navigator.

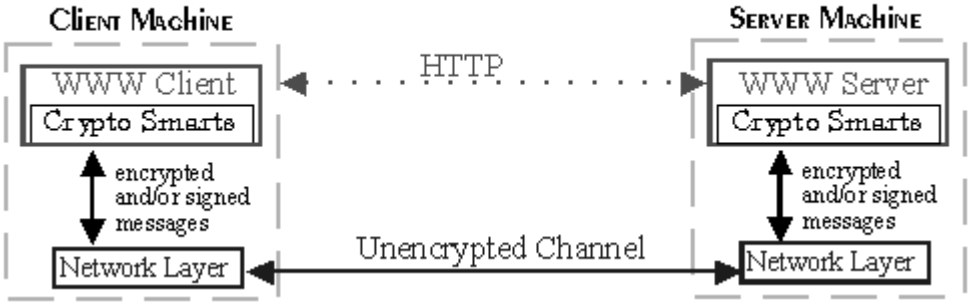
It is worth mentioning that in addition to the SSLREF reference implementation created by Netscape, there is at least one other implementation of SSL. The *SSLeay* package implemented by Eric Young in Australia (i.e., outside of the U.S.A.) is freely available for commercial and non-commercial use. [35]

And to further confuse the Web security landscape, Microsoft has recently publicized its "Private Communication Technology" (PCT), which is basically a slightly improved version of SSL. The improvements offered by PCT include fewer communications in the hand-shake phase (and thus, fewer round-trip delays), and a fix of a slight security weakness in SSL's client authentication protocol. Interested readers are encouraged to see [36] for details.

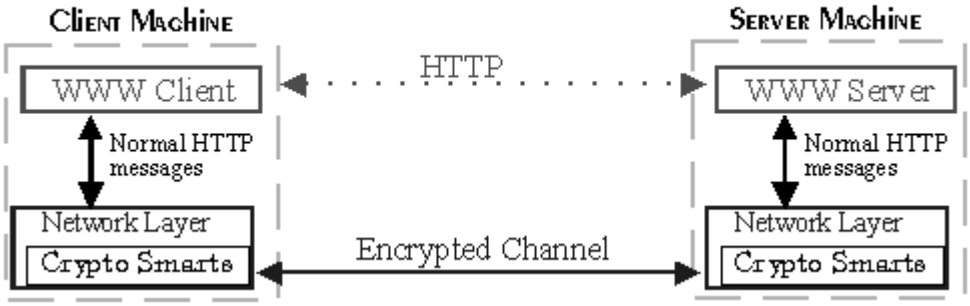
Contrasting SSL and S-HTTP

At this point, it is worth emphasizing the fundamentally different approaches to securing Web communications employed by SSL and S-HTTP. With S-HTTP, messages are cryptographically protected and thus can be safely sent over any connection; in this case, the security intelligence is at the application layer, which means that the browser intimately understands certificates, signatures, encryption, and the like. In contrast, the SSL/PCT approach alleviates the need for application-level security intelligence, since the cryptographic smarts are positioned closer to the network layer. With SSL, normal, unprotected, HTTP messages are sent through cryptographically protected channel.

S-HTTP: Application-level Security



SSL: Connection-level Security



So which approach is *better*? We cannot really say, as such a question amounts to comparing apples with oranges. However, we may make a few observations regarding the strengths and weaknesses of both schemes:

- SSL may offer slightly higher performance, since the cryptographic operations are simpler, and do not involve signing or verifying the signatures on messages.
- Use of SSL requires a new proxy protocol to pass through firewalls, whereas S-HTTP can use the same proxy system as regular HTTP.
- Signed messages, as in S-HTTP, allow for non-repudiation. This means that a signed S-HTTP message containing a vendor's receipt, for example, may be used later to prove that the vendor agreed to send a product which was paid for online. With channel-oriented approaches, non-repudiation is not possible.
- SSL is not Web or HTTP specific, and may be fairly easily supported in other network applications.
- Application-layer approaches like S-HTTP allow for the display and control of a wide range of security parameters. In essence, the user may deal with the security options rather explicitly, since the application has full knowledge and control of these options.
- Since SSL is a simpler scheme, it may be easier to use by browser users and server administrators. For example, precisely specifying the S-HTTP anchor attributes for all hyperlinks on a secure server may be more difficult than changing the URL protocol anchors from "`http`" to "`https`".
- There are other network security systems, such as Kerberos and the upcoming IPv6, which provide similar functionality to that of SSL and PCT.

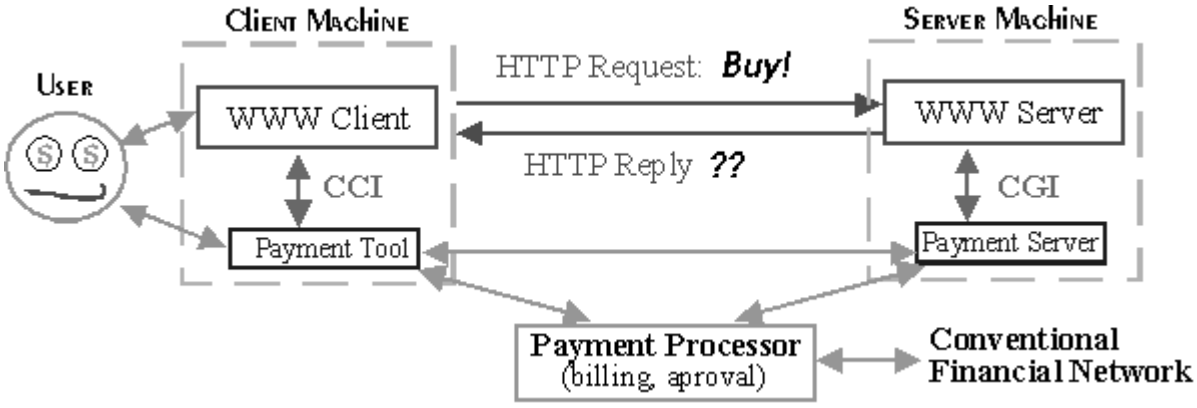
A Few Words About Digital Payment and the Web

It is very important to note that all of the Web security technologies covered here have been designed to achieve authentication and confidentiality of Web communications. This is not the same goal as enabling Web-based digital payment. In fact, none of the schemes presented in this document is very well suited for any except the simplest and least desirable forms of online payment.

We certainly may use the systems described here to securely transmit credit card information from the Web browser to the server; in the process we may also be able to authenticate the merchant server and perhaps even the browser user (the credit card owner). However, it would be even better to not let the merchant have the credit card numbers at all -- that information could instead be encrypted in such a way that only the credit card acquiring bank can read it. This makes sense when you consider that the merchant need only know that the payment is approved (after that, the merchant is compensated using the traditional credit card clearing mechanism). Furthermore, by using these simple credit card payment schemes, a nefarious merchant may collect a large amount of customer credit card numbers; then he may either open up a bogus shop and make phony payment requests for transactions which never occurred, or he may simply use these credit card numbers to purchase other online goods which do not require strong client authentication. These are attacks which we would like to be able to prevent.

In a more general sense, the Web security mechanisms covered here are inappropriate for realistic Web payment schemes largely because payment systems have such diverse requirements [37]. For low-cost items, such as online information, it is probably most important that the transactions be very fast and easy; bullet-proof security is less of a concern. However, for higher-cost items, such as hard goods, non-repudiation of payment is crucial. And in either of these cases, there may need to be a third party, such as an online bank, involved in the transaction -- so we cannot think of the Web client-server communications alone. Payment systems have many other issues involved. For instance, which comes first: the payment, or the delivery of the goods? Also, who manages online accounts, and who is legally liable when things break down? Another important issue is the provision of *privacy* in payment schemes; there may be cases where we would like our transactions to be audited, and there may be even more cases in which we'd like this information to be kept private.

Since online payment systems require much more than the security of the Web communications, most payment schemes under development today employ an architecture similar to the one depicted in the diagram below:



The important features of this architecture are as follows:

- For browser/server independence, payment systems use external applications.
- No special security features of HTTP communications are utilized, since the protection of payment data is handled by the payment applications themselves.
- If there is any communication between the payment applications and the Web applications, it is usually done using CGI on the server, and using CCI-like interfaces on the client.
- Payment processing usually involves a third party in transaction protocol.

Interested readers are invited to see [38] for a high-level summary of the different Web payment systems being developed currently.

Conclusions

Given the rather large number of Web security technologies covered here, and considering that there are other security schemes that were *not* mentioned, one might be inclined to ask how we are going to deal with this confusing scene. Will a particular security system come to dominate the Web? Probably not. Are there any efforts aimed at consolidating these different security mechanisms?

There are a few efforts geared towards forging a peaceful coexistence between Web security schemes. The Spyglass Security Plug-in Module architecture [39] defines a way in which different security "modules" may be used in a simple, user-configurable manner. The World Wide Web Consortium (W3C) is developing protocol extensions to HTTP [40] which may allow various application-level security schemes to use common features of HTTP 1.2. It has also been suggested that browses supporting portable-code languages like Java [7] may easily gain support for new security mechanisms simply by downloading the appropriate applets from a trusted source. In the meantime, Web application developers may use toolkits supporting both S-HTTP and SSL, such as those provided by Terisa Systems [41], to add support for two of the more popular Web security schemes in use today.

For now, there is no easy answer to the question of "how to secure the Web." Readers are well advised to critically analyze the relative merits of the Web security technologies presented here, and formulate their own conclusions on what systems are most appropriate for the security needs of their applications of interest.

References

[1] William Stallings, Network and Internetwork Security, Prentice Hall, 1995

[2] Steven Shaffer and Alan Simon, Network Security, AP Professional, 1994

[3] "Internet Security: Contexts, Issues, Users, Approaches," by Stephen Squires Presentation at the Federal WebMasters Workshop, July 11, 1995

[4] Security Concerns of Executing Shell Scripts Inside Mosaic
`http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/executing-shell-scripts.html`

[5] "Hack Netscape," September, 1995
http://www.c2.org/hacknetscape/

[6] William R. Cheswick and Stephen M. Bellovin, Firewalls and Internet Security: Repelling the Wily Hacker, Addison-Wesley, Reading Mass., 1994

[7] Sun's Java Language: The Security Story
`http://java.sun.com/1.0alpha2/doc/security/security.html`

[8] NCSA httpd Patch for Buffer Overflow
`http://hoohoo.ncsa.uiuc.edu/docs/patch_desc.html`

[9] CGI Security by Paul Phillips
`http://www.primus.com/staff/paulp/cgi-security/`

[10] Writing Secure CGI Scripts (NCSA)
`http://hoohoo.ncsa.uiuc.edu/cgi/security.html`

[11] NCSA httpd: Making your Setup More Secure
`http://hoohoo.ncsa.uiuc.edu/docs/tutorials/security.html`

[12] SAIC Security Documents
`http://mls.saic.com/docs.html`

[13] Introduction to TCP-wrapper
`ftp://ftp.win.tue.nl/pub/security/tcp_wrapper.txt.Z`

[14] Basic Protection Scheme for the WWW
http://www.w3.org/hypertext/WWW/AccessAuthorization/Basic.html

[15] Mosaic (httpd) User Authentication Tutorial
http://hoohoo.ncsa.uiuc.edu/docs/tutorials/user.html

[16] How to set up protected CERN server
http://www.w3.org/hypertext/WWW/Daemon/User/Config/AccessAuth.html

[17] Howto: Enhanced Access Control
http://hoohoo.ncsa.uiuc.edu/howto/access_control.html

[18] Bruce Schneier, Applied Cryptography, John Wiley & Sons, 1994

[19] Cryptography: The Study of Encryption
http://draco.centerline.com:8080/~franl/crypto/cryptography.html

[20] RSA's Frequently Asked Questions About Today's Cryptography
http://www.rsa.com/faq/faq_toc.html

[21] Simpson Garfinkel, PGP: Pretty Good Privacy, O'Reilly & Associates, 1994.

[22] Phillip Zimmerman, The Official PGP User's Guide, Cambridge Mass: MIT Press, 1995.

[23] RFC 1421: "Privacy-Enhancement for Internet Electronic Mail," by J. Linn
ftp://ftp.internic.net/rfc/rfc1421.txt

[24] Using PGP/PEM with Mosaic
http://hoohoo.ncsa.uiuc.edu/docs/PEMPGP.html

[25] CCI-based Web Security: A Design Using PGP
http://sdg.ncsa.uiuc.edu/~jweeks/www4/paper/current_rev.html

[26] Secure HTTP Description (includes pointers to S-HTTP FAQ and Documentation)
http://www.eit.com/projects/s-http/index.html

[27] S-HTTP Specification
http://www.commerce.net/information/standards/drafts/shttp.txt

[28] Secure Mosaic Home Page
http://www.commerce.net/software/SMosaic/

[29] Secure HTTPd Home Page
http://www.commerce.net/software/Shttpd/

[30] Message Digest Authentication Internet Draft
ftp://ds.internic.net/internet-drafts/draft-ietf-http-digest-aa-01.txt

[31] Kerberos: An Authentication Service for Open Network Systems
ftp://athena-dist.mit.edu/kerberos/doc/usenix.PS

[32] Using Kerberos with httpd 1.5
http://hoohoo.ncsa.uiuc.edu/beta-1.5/howto/kerberos.html

[33] Intro to Kerberos on the Web
http://snapple.ncsa.uiuc.edu/adam/khttp/intro.html

[34] The SSL Protocol
http://home.netscape.com/newsref/std/SSL.html

[35] SSLeay and SSLapps FAQ
http://home.mcom.com/newsref/std/SSL.html

[36] Private Communication Technology Protocol
http://www.microsoft.com/windows/ie/pct.htm

[37] Requirements for Network Payment: The NetCheque Perspective
ftp://prospero.isi.edu/pub/papers/security/netcheque-requirements-compcon95.ps.Z

[38] "Digital Commerce on the Web", from NCSA Access, Summer, 1995
http://www.ncsa.uiuc.edu/Pubs/access/95.2/95.2Contents.html

[39] Spyglass Security Protocol Module API
http://www.spyglass.com/techspec/spm_spec.html

[40] HTTP Security group of W3C
http://www.s3.org/hypertext/WWW/Security/Group/

[41] Terisa Systems Home Page
http://www.terisa.com/