CS255: Cryptography and Computer Security **Winter 2002**

Programming Project #2

Due: Wednesday, March 13, 2002, 11:59pm

Overview

In project 2, you will be expanding on the authentication system you built in project 1. Since the project 2 code does not depend too much on what you did in project 1, we will not be providing a solution. You will not be graded on project 1 features, but please come see us in office hours if you are concerned about your project 1 code.

As in the first project, you may work in groups of up to two on this project. No late days may be used. **Projects submitted after the due date will not be accepted** without advance permission from the instructor.

For project 2, you will need to do the following:

- Secure all traffic using SSL.
- Build and use a public key infrastructure using certificates.
- Use SSL client certificates to securely authenticate to merchants.
- Use a secure password authentication mechanism.
- Implement a secure and efficient online user revocation method.

We will examine each of these features in detail below. Since we have not yet covered in lecture all of the topics explored by this project, you may wish to start first on those aspects of the project that you can do immediately and save the other parts for later.

Secure Communication

The implementation of the authentication system in the first project had a fairly large security flaw, as some of you pointed out: Once a user had been authenticated to the merchant, an active attacker could remove the user from the network and use the user's credit card. This attack is generally known as "session hijacking". For project 2, we will eliminate this attack by using SSL to protect all Web communication with the merchant in addition to the authentication server.

Client Authentication Using Certificates

In project 1, we required users to visit the authentication server and enter their passwords each time they wanted to visit a store. Now we will use public key cryptography to enable the user to visit the authentication server only once, and then be able to authenticate to as many stores as they want.

We will do this using SSL client certificates, as follows: The user will visit the authentication server and enter their username and password. The authentication server will generate and sign a certificate and install it in the user's browser. The user can then visit merchant stores and present the certificate to each of them, without having to revisit the authentication server until the certificate expires or the user uses a different computer or browser.

Note that since the user now has a secure connection to the merchant, there is no longer a need for the authentication server to encrypt the credit card number; the user can enter it to the merchant directly over SSL. The user's password should still be kept secret only to the user and authentication server, though.

Public Key Infrastructure

In project 1 we provided the authentication server with an SSL key and certificate and told you to trust its security; for project 2, you will need to generate your own certificates. You will need to implement a public key infrastructure as follows:

1. Establish a CA private key and certificate that will be used for signing all the client and server certificates.
2. The CA private key should be password-protected and accessible only to the authentication server, which will use it to sign client certificates.
3. Distribute the CA certificate to all merchant servers so they can use it to verify client certificates. You may assume that you have a trusted mechanism for doing this (i.e., just copy the file over).
4. Create SSL server certificates for all the Web servers (authentication server and merchant servers), signed by the CA certificate. Make sure that all private keys are protected using a password.

Since all certificates used in the system are signed by the CA, the validity of any certificate can be verified knowing only the CA public key. If you like, you can install the CA certificate in your Web browser as a root certificate, and the SSL security warnings will go away. [1]

Secure Password Authentication

In project 1, we did not ask you to provide any security for the authentication server's password file. It contained simply a list of usernames and passwords, and if an attacker got hold of it, he could have broken the entire system. We now ask you to make the password file secure, so that even if a malicious entity can read it, they cannot obtain a certificate for a user whose password they do not know.

[1] Actually, since SSL certificates contain the hostname for the server they are issued for, you will probably still get name mismatch errors if you run your code on several different machines. You may ignore these. Some browsers support "wildcard" certificates, e.g. `*.stanford.edu`, but some do not.

Since the passwords will no longer be specified in the clear, you should provide a command-line tool for adding new users, changing passwords, and marking whether a user's account has been revoked (see "User Revocation" below).

Instead of specifying a particular method for doing secure password authentication, you should come up with your own design, based on the methods discussed in lecture and in the texts. You should be sure and discuss your implementation in your writeup. Explain your design, why you chose it, and what other solutions you considered. At the very least you should use one-way hashing and public salts when storing passwords on the authentication server.

User Revocation

One problem with using client certificates for authentication is that they are good for a long time. A certificate has an expiration date, but it will probably be 30 days, or a year, or longer, from the date of issue. In our system, we will want to be able to disable ("revoke") a user much more quickly. For example, if a user's password is stolen, certificates already issued should not continue to be valid. For this reason, we need a method for revoking user certificates more quickly.

One method would be to require the user to reauthenticate for every transaction. This would be inconvenient, and would destroy the advantage of the long-lived certificate. A better approach is what is known as a Certificate Revocation List (CRL), a signed list of revoked certificates issued by the CA (in our case, the authentication server). If the merchant has an up-to-date copy of the CRL, he can verify its signature, and know whether a user was valid as of the date on the CRL. However, if the number of revoked users is large, and the merchant needs to retrieve a new list often, this becomes cumbersome.

We will use a variant on CRLs known as Certificate Revocation Trees, or CRTs. The basic idea of CRTs is that instead of creating a list of revoked certificates, we create a hash tree, where the leaves correspond to the revoked users. The procedure to create a CRT is as follows:

1. Sort all the revoked certificates using a fixed ordering (e.g., lexicographically by username, or using the certificate serial number).
2. Create a binary tree, with the leaves being hashes of the identifier for the revoked certificates, using a collision-resistant hash function. Make sure that the leaves' order (left-to-right) retains the sorting.
3. Moving upwards in the tree, for each node, compute the hash of the concatenation of its children (which are themselves hash values).
4. Append a date to the root node's hash, and sign it.

The merchant can now query the authentication server to find out if a particular user's certificate has been revoked. If the certificate has been revoked, the authentication server can send the node corresponding to that certificate, the signed root of the tree, and all the intermediate nodes necessary for the merchant to verify the path through the hash tree. Thus with only $O(\log n)$ hash values (where n is the number of revoked users) and one signature, a revoked user can be identified.

The CRT can also be used to show that a certificate is valid: the signature can be used as proof that no corresponding leaf is in the tree. Since the users are sorted, the authentication server can provide hash values for enough nodes to prove that there are leaf nodes adjacent in the tree such that the certificate in question would fall between those users. Since the nodes are adjacent, it cannot, and the certificate has not been revoked.

In fact, the authentication server does not have to host the CRT directly. In your implementation, you should create a CRT cache server. The merchants will query the cache server, which can run a fast, unencrypted (no SSL) service to respond to revocation queries. Even though the cache server is unprotected and untrusted, this is secure, because the authentication server's signature validates the CRT. This allows the burden of online verification to be moved from the authentication server to a cache server. In a real system, there could be multiple cache servers, providing a cheap, distributed, local solution.

To be effectively used, the authentication server will need to generate new CRTs often enough to keep the list up-to-date. You can either have the authentication server send it directly to the cache, or have the cache request it from the authentication server. Either way, you should make sure that the cache is populated with a new signed CRT with frequent regularity (e.g., every 10 minutes). The merchant servers should verify that the timestamp for the CRT signature they receive is recent before letting a user authenticate. You may assume that the merchant and authentication servers have synchronized clocks that are no more than five minutes apart.

The above description of CRTs is not complete; it does not address the details of several cases; e.g., what if there are no revoked certificates, or the revoked nodes cannot form a complete binary tree? You should either come up with a complete CRT design yourself, or consult one of the referenced papers. Each paper has a slightly different implementation of CRTs, so you may want to look at several of them to find one that appeals to you. Either way, your design should be secure, and merchants should be able to verify whether a user's certificate is valid, or has been revoked.

Implementation note: the authentication server will now need, in addition to the keys for its SSL and CA certificates, an RSA or DSA signature key pair for revocation. You should not re-use the existing certificate keys. The corresponding public key can be distributed to the merchants along with the CA certificate.

References

1. Kocher, Paul, *A Quick Introduction to Certificate Revocation Trees*, 1988.

2. Berkovits, Shimshon and Jonathan C. Herzog, *A Comparison of Certificate Validation Methods for Use in a Web Environment.* `http://www.mitre.org/support/papers/tech_papers_01/berkovits_comparison/berkovits_comparison.pdf`

3. Wohlmacher, Petra, *Digital Certificates: A Survey of Revocation Methods.* `http://www.acm.org/sigs/sigmm/MM2000/ep/wohlmacher/`

Programming Environment

As in the first project, we have provided starter code that implements some of the project's skeleton for you. You may wish to continue using your project 1 code for the authentication and store servers, as the starter code for these servers has not changed much. You will need, however, to be sure to copy the new `weblib` directory and update your Makefiles, as the new skeleton code requires new files and libraries.

The starter code is located in the following directory:

```
/usr/class/cs255/proj2
```

The `auth` and `store` directories contain skeleton versions of SSL-enabled authentication and merchant servers. The `cache` directory contains a skeleton cache server, and the `pwdtool` directory a skeleton password tool.

New `weblib` Functionality

Here are some highlights of some of the new functions available in project 2. See the `weblib.h` file for more detail; functions new to project 2 are marked.

`GetSSLContext`: Returns the `SSL_CTX` context that will be used for incoming SSL requests. This function should be used to obtain the context during initialization, to set all required functions before calling *RunSecureWebServer()*.

Note that unlike the project 1 starter code, the `SSL_CTX` is not set to use specific private key and certificate files. You must do this manually in your initialization routines. See the OpenSSL handout for more details.

`GetSSLConnection`: When called from *DisplayWebContent()*, returns the `SSL` connection context for the current connection. You can use this result to get information about the status of the SSL connection, which is particularly useful when doing client authentication.

`SendCertificateForm`: Use this function to issue a certificate to the Web browser. This function will automatically display a Web page that will cause the browser to generate a new private key and send a certificate request. This request will then be automatically passed to a function, which you can set using *SetIssueCallback()* during initialization, which can sign the certificate. You can also pass a string token which the browser will send back to the server and will be passed to the issue function.

This function will handle certificate requests with Netscape Navigator 4.x on any platform, or with Internet Explorer 4.0 or later on Microsoft Windows (version 5 or later is strongly recommended). Other platforms or browsers, including Netscape 6, are not supported.

`GetInsecureRemoteData`:
`GetSecureRemoteData`: These functions connect to a remote Web server, optionally using SSL, and retrieve a given URL, passing in arbitrary data and returning the result. You can use this to communicate directly between Web servers without involving the user. On the remote side, you can retrieve the passed-in data using *GetRedirectData()*, and send the result using *SendRawData()*.

`InstallTimer`: Installs a callback function which will be called a specific number of seconds in the future. This can be used for installing periodic events that are not triggered by Web requests. Note that each timer will be fired exactly once. Timers will not be called at the same time Web requests are being fulfilled.

Tips For Working With SSL In Web Browsers

Both Netscape Navigator and Internet Explorer have tools for examining and modifying the certificates they have stored for use in client and server authentication, as well as for viewing information about the current server's certificate. You should familiarize yourself with the appropriate tools in your browser.

- To view the certificate and other security information for the current SSL page, click the lock icon in the status bar. This works in both Netscape and IE. In Netscape, the information for the current page is under the "Security Info" section.
- To examine, modify or remove stored certificates in IE (version 5.0 or later), choose "Internet Options" from the Tools menu. Then navigate to the "Content" tab and click the "Certificates" button.
- To examine, modify or remove stored certificates in Netscape Navigator, click the lock icon in the status bar (it will be locked or unlocked depending on whether the current page is secure). Then click the "Certificates" link in the left-hand listing of options.
- When connecting to an SSL site that requests a client certificate, the browser may automatically select a certificate, or pop up a dialog to have you choose one. Internet Explorer will show a dialog if you have two or more matching certificates, or proceed automatically if you have zero or one. Netscape can be set to always proceed automatically, or to let you choose. It is strongly recommended that you set the latter preference: Select "Ask Every Time" in the Navigator section of the security dialog (click the lock icon in the status bar).
- Both Netscape and Internet Explorer keep information about SSL certificates, both for client and server, as long as you have your browser window open. If you have installed or changed client or server certificates, and want the browser to pick up the change, or if you want to use a different client certificate to authenticate to a site, you will probably need to close your browser window and open another one. Simply reloading the page will not work.

Help

- This project relies on more esoteric aspects of OpenSSL than the first project did. The following sources of information may prove useful:
  - A second OpenSSL handout, which you should have received along with this project handout, summarizes the public key cryptography and SSL aspects of the OpenSSL libraries.
  - In addition to the OpenSSL man pages and official documentation, a good Web site to visit is `<http://www.columbia.edu/~ariel/ssleay/>`. This is unofficial documentation of the cryptography library from SSLeay, the predecessor to OpenSSL. While it is somewhat out of date, it is the only online documentation available for some parts of the cryptography library (e.g., the ASN1 or X509 function calls), and you may find it useful.
  - If you wish to look at the OpenSSL source code to see exactly how a function is implemented, a copy of the version we are using has been placed in the course directory:

`/usr/class/cs255/example/openssl-0.9.6.c/`

- The class newsgroup will again be the primary place to look for answers and ask questions. Kudos to all students who answered questions for project 1.
- We will continue to hold some of our office hours in Sweet Hall. Please check the web page or the newsgroup for up to the minute office hour locations.
- As a last resort, you can email the staff at cs255ta@cs.stanford.edu.

Submission

In addition to your code, we would like you to include the certificates and key files necessary for the authentication server, at least two stores. Include instructions on how to use your system (be sure to tell us how to add and revoke users), including any passwords that might be necessary.

You should also submit a README containing the names, Leland usernames and Stanford ID numbers of the people in your group as well as a description of the design choices you made in implementing each of the required security features. Since there is a great deal of design work for this project, please don't skimp on the README.

You should be sure to include a complete description of your authentication system, as well as your solutions to all of the design problems presented. Also include an analysis of why your system is secure, and what choices you made or algorithms you decided against. Include any information you think might be relevant to us in grading your solution.

The project will be tested on one of the Leland Systems cluster computers, using the Solaris version of Netscape Navigator 4.79 to test the client certificates. It is highly recommended that you test your solution on this platform.

When you are ready to submit, make sure you are in your `proj2` directory and type
`/usr/class/cs255/bin/submit`.