CHAPTER 9

Implementing Web Security Using auth-lib.pl

OVERVIEW

A natural extension of most applications is the capability to restrict or track access through the application. To do this, we need a system designed to authenticate users when they first start using the application. This is where **auth-lib.pl** comes in.

auth-lib.pl is a Perl authentication library that provides the three core capabilities that any security library needs. First, it allows the administrator of a Web application to maintain a registration list or password file of users who are authorized to access the application (or who have recently registered, depending on the rules that the administrator sets up). Second, it allows users to log in to the application either through the Web server's built-in HTTP authentication or through a CGI form. Figure 9.1 shows an example of the CGI-based logon form. Once users

are logged in, **auth-lib.pl** tracks them by using unique session files that are created when users complete a successful login. To support all these functions, the authentication library contains a great many configuration options, accommodating all sorts of security scenarios.

Figure 9.1 *The CGI-based logon form.*

Playing a major role in the flexibility of the authentication library is the ability to configure the fields of user information to store. By default, this library is configured with fields such as last name, first name, and E-mail address, and more fields can easily be added. Core fields include username, password, and the names of security groups a user belongs to. By using group names, a programmer can make a CGI script fully configurable. For example, you may want to allow only certain people to view a groupware calendar and other people to be able to post new events or change existing ones.

The authentication library can take advantage of CGI form-based login and authentication on its own, or it can integrate with Web server–based authentication. Web server–based authentication is useful if your company owns and maintains its own Web server and can configure it to log users in. CGI-based authentication, on the other hand, means that people log in using an HTML form that is processed by a CGI script. For people on the Internet who share a Web server on an Internet service provider, CGI-based authentication means that you need not lose authentication capability if you are not allowed to change the Web server configuration.

When a user logs in, all the configurable field information, such as username and last name, is stored in a session file. This file is associated with a uniquely generated *session code* that is passed from screen to screen of your Web application, providing the application with the user's information at all times. This is typically referred to as *maintaining state*.

Normally, Web servers have no concept of a continuous session. When a user goes from page to page, the Web server sends each page on an entirely new connection. The Web server has no knowledge of which pages have been previously viewed. The authentication library gets around this limitation by setting up a unique session code and assigning it to a hidden form variable. This session code is sent as a parameter to each CGI script access for the application. This technique allows the application to look at the session file to maintain the state of the application. The state of the application is always dependent on the user's most recent action, and this information is reflected in the session file.

Installation and Usage

When the authentication library is installed from the accompanying CD-ROM, your directories and files should be similar to Figure 9.2. This figure also shows the permissions needed on the files and directories. The additional **html-auth** files are a sample application for protecting HTML documents by using **auth-lib.pl**.

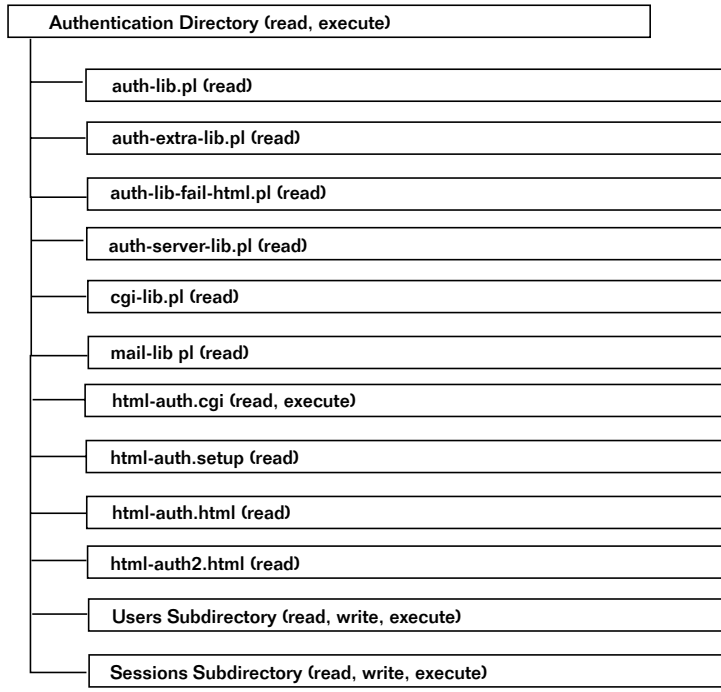

Figure 9.2 *Directory structure of the authentication library along with sample HTML filter scripts.*

Although the diagram shows the authentication library and html-auth CGI scripts in the same Authentication directory, the library files can be placed in any directory. By default, most of our applications that use this library store all their library files, including the authentication library, in a subdirectory called Library underneath the main scripts directory.

Authentication Library Files

auth-lib.pl is the core of the authentication library. You must `require` this library in your CGI scripts to enable authentication. This file needs to be readable by the Web server.

auth-extra-lib.pl is loaded by **auth-lib.pl** if any logon, registration, or searching functions need to be performed. This file must be readable.

auth-extra-html.pl is loaded by **auth-extra-lib.pl** and provides Perl code that prints the HTML related to logon, registration, and search functions in the authentication library. This file must be readable.

auth-lib-fail-html.pl is loaded by **auth-lib.pl** when a session is not valid. It outputs an HTML screen informing the user of the error. This file must be readable.

auth-server-lib.pl is loaded by **auth-extra-lib.pl** if Web server–based authentication is being used. This file must be readable.

auth.setup is a sample setup file for authentication. CGI scripts that use this library typically have their own setup file in which the authentication variables are specified.

mail-lib.pl is necessary so that the authentication library can send E-mail. **cgi-lib.pl** is necessary for any sample CGI applications that may be running in the same directory. These files should be readable.

The **Users** subdirectory is where the user file for CGI-based authentication is stored by default. This directory should be readable, writable, and executable.

The **Sessions** subdirectory is where sessions are created by the authentication library. This directory should be readable, writable, and executable.

html-auth Sample Authentication Application Files

html-auth.cgi is a sample application that uses authentication. Its purpose is to protect HTML files from being viewed by unauthorized visitors. This file must be readable and executable.

html-auth.setup is the setup file for **html-auth.cgi**. It includes all the setup variables that the authentication library needs. This file must be readable.

html-auth.html and **html-auth2.html** are simple HTML files that are used as example protected HTML files in this chapter.

Server-Specific Setup and Options

The authentication library assumes that certain setup variables are set before the main routine, `GetSessionInfo`, is called. These variables are usually set up in the CGI script that uses the authentication library. For example, the authentication variables for the bulletin board system script reside in the **bbs.setup** file.

`$auth_lib` contains a path to the directory where the authentication library is stored. Typically, libraries that are commonly used should be stored in a central directory. If you want to keep the authentication library in the same directory as the other CGI script that is using it, simply set `$auth_lib` to ".", which is shorthand notation for "the current directory I am in."

`$auth_server` is set to `on` if you are using server-based authentication. Web servers can be configured to ask the user for a username and password when certain pages are viewed or scripts are executed. If you set `$auth_server` to `on`, the authentication library reads the username from the Web server that has logged the user on. From there, the authentication library can be modified to look up other information about the user, such as first name, last name, E-mail address, and more, based on the username.

`$auth_cgi` is set to `on` if you are not using server-based authentication. CGI authentication is typically used in place of server-based authentication if you are on a shared server and do not have permissions to modify the Web server setup files to add access restrictions.

If you do not wish to use any authentication at all, you would set both `$auth_cgi` and `$auth_server` to `off`. You might use this option if you wish your application to be free of security checks. For example, the bulletin board script described later on in this book uses the authentication library to provide first name, last name, and E-mail information about the user when the user posts a message. However, if authentication is turned off, the BBS program makes users type that information when they post. Some sites prefer to have the BBS totally open to all users so that they do not have to type information unless they want to post a message.

You should not set both `$auth_cgi` and `$auth_server` to `on` at the same time. If both variables are set to `on`, the code that makes a session based on `$auth_server` being set to `on` will override the code that makes a session based on `$auth_cgi` set to `on`. We advise that you do not try this at home—or at least on your home page.

Many of the variables discussed here are CGI security–specific. When you're using CGI-based security, the authentication library must do much more work. The library must log users onto the system, allow them to register on the system if they are not yet there, and search for accounts if they forget their username. Variables related to these functions are specifically associated with the CGI-based security.

`$auth_user_file` is set to the directory and filename of the file that contains information about the users who are allowed to get into the system. This variable is used only when CGI authentication is on. When you use Web server–specific security, the user's password and username are stored in a Web server–specific file. This file is not needed if you are using Web server–specific security instead of CGI-based security. A sample of a user file is shown next:

```
gunther|mypassword|readaccess|Gunther|Birznieks
selena|selenapass|readaccess:writeaccess|Selena|Sol
```

`$auth_alt_user_file` is set to the directory and filename of the file that contains information about the users who have registered in the system. Normally, when users enter their information into the registration screen, their data is stored in `$auth_user_file`. Placing a value in `$auth_alt_user_file`, however, sends registrations to an alternative file instead of the original one. This arrangement allows you to check registrations before you decide to move them to the user file. This technique is also CGI security–specific.

`$auth_allow_register` is set to `on` if you wish to allow users to register at the logon screen using CGI-based security. Figure 9.3 shows an example of a registration screen.

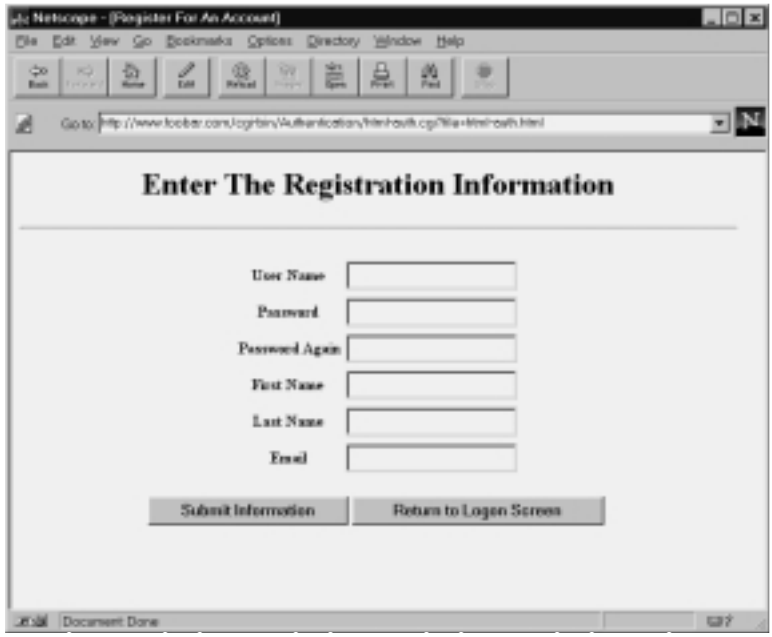

Figure 9.3 *CGI-based HTML registration page.*

`$auth_allow_search` is set to `on` to allow users to search for their username in case they forgot it at the CGI-based logon screen. Figure 9.4 shows an example of a search screen.

`$auth_default_group` is the security group assigned to users when they register for access using the CGI-based security mechanism. You can use this variable if you have a CGI that script calls the authentication library and you wish to distinguish whether a user has certain types of access, such as create, delete, or modification. The typical default is "normal." A user's groups determine the user's rights in the CGI script that is using the authentication library.

If `$auth_generate_password` is set to `on`, the CGI security mechanism generates user passwords when they register for the site instead of allowing them to choose their own password. The generated passwords are E-mailed to the users after they have submitted the registration. Figure 9.5 displays an example HTML registration form when this variable is set to `on`. Notice that users are no longer prompted to enter a password of their own choosing.

Figure 9.4 *CGI-based HTML search page.*

Figure 9.5 *CGI-based security HTML page for registration when* ***auth-lib.pl*** *generates the password.*

`$auth_use_cleartext` is set to `off` in order to trigger the storage of passwords in the user file in encrypted form for CGI-based security. This arrangement is useful on a shared server where someone may be able to snoop in your user file and find out the passwords. If the passwords in the user file are encrypted, it will be more difficult for someone to break into your site.

`Cleartext` indicates that password information is sent as "clear text" that anyone can see. Thus, it is `off` when you do not want your passwords to be stored as clear text. It is administratively easier but less secure to keep the passwords unencrypted. When passwords are stored as plain text, you can look up a forgotten password quickly, or you can change the password manually. If the passwords are stored in encrypted form, it is difficult to change a password by editing the user file manually; you have to figure out what the new encrypted form of the password should be.

Using the `crypt` function in Perl to encrypt passwords may not work on some operating systems. If you are using another OS such as Windows NT, you may need to turn off password encryption to make this library work.

`$auth_check_duplicates` is set to `on` for CGI-based security when you wish to make sure that a person cannot register with the same username twice at the same site.

`$auth_add_register` is set to `on` for CGI-based security when you want the authentication library to add the user's registration to the designated user file (`$auth_user_file` or `$auth_alt_user_file`, described previously).

`$auth_email_register` is set to `on` for CGI-based security when you wish a user's registration to be E-mailed to you instead of or in addition to being written to a file.

`$auth_admin_from_address` is the source E-mail address that the authentication library uses for CGI-based security whenever it E-mails information to a user or to you.

`$auth_admin_email_address` is the destination E-mail address for you whenever the authentication library E-mails information about a CGI-based registration.

To send E-mail, the authentication library relies on the mail-lib.pl library discussed in Chapter 7. Remember to include this program in the directory that holds the authentication library files.

`$auth_session_length` is the number of days that an authentication session is valid. Whenever a user logs in through the authentication library, a session file is created that is valid only for a short time. The authentication library then uses that session file to read the user's information based on a session ID code that is sent to the library after the initial logon. In that way, users need not type their username and password for every screen of the authenticated CGI application. Although this number represents the value in days, it is important to note that the value can be a fraction. If you want to have sessions last 1/24th of a day, you would set the value to `"1 / 24"` (one divided by 24). It is a good idea to keep this session file around for the maximum time the user will be using the application. Otherwise, the session file might be deleted before the user has finished using the Web program, and the user will get an error because the session is no longer valid.

It can be useful to set up a session to be an outrageous length, such as 8000 days. In this case, users can typically bookmark their sessions. If the session IDs become invalid after a mere day or so, users cannot bookmark the URLs related to the Web script. This is because when they try to go back, the session ID is not valid and they must log in again. Setting the session length to a huge number alleviates that problem.

However, be aware that keeping the sessions and their files around lessens the security of the site, because a user can try different session IDs to break into your application instead of just a username and password. In addition, keeping the session files around for a long time clutters the directory with small session files and takes up disk space.

`$auth_session_dir` is the directory where the session files are stored. It is highly recommended that this be a separate subdirectory under the main

scripts area so that the session files do not get mixed up with the main scripts. In addition, the directory that holds the session files must be writable by the Web server. You do not want the directory where your scripts reside to be writable.

`$auth_register_message` is set for CGI-based security to send users a brief message after they have registered to the system. You may need to change this variable depending on how the previous parameters are set. For example, if you have the authentication library set to generate passwords and send them via E-mail, the message should say something like, "Thanks, you will be E-mailed your password within the next few minutes."

`$auth_password_message` is set for CGI-based security to E-mail users a brief message related to their automatically generated password if you turn that option on.

The authentication library stores a set of core information for each user: username, password, and the groups the user belongs to. In addition to these fields, many CGI applications require additional fields of information about a user. For example, the bulletin board system requires first name, last name, and E-mail information. The information related to these extra fields is stored in a couple of arrays. One array (`@auth_extra_fields`) contains the field names, and the other array (`@auth_extra_desc`) contains the descriptions of the fields.

Field names should generally be lowercase alphabetic characters and contain no spaces. In addition, field names must start with `auth_`. The underscore ("_") character after `auth` is important. The authentication library uses the `auth_` prefix to figure out which form variables have been passed to it that belong to the authentication library. The descriptions can contain fancy formatting. For example, a field name for first name would be `auth_first_name`, whereas a field description would be `First Name`.

If you create extra field names, be sure to prefix them with `auth_`. The authentication library relies on this designation to determine which form variables are related to the authentication process and which ones came from the previous Web application that happens to use this script.

In addition, if you are collecting E-mail information about the users, always name the E-mail form variable `auth_email`. The authentication library looks for the `auth_email` form variable when determining the address to send password information to. This happens when the library is configured to E-mail users a computer-generated password after they register.

`$auth_logon_title` contains the title that is printed between the `<TITLE>` HTML tags in the CGI-based security logon screen. `$auth_logon_header` is the header that is printed between the `<H1>` HTML tags in the same page.

The following sample setup file illustrates the typical syntax:

```
$auth_lib = ".";
$auth_server =                  "off";
$auth_cgi =                     "on";
$auth_user_file =               "Users/user.dat";
$auth_alt_user_file =           "";
$auth_allow_register =          "on";
$auth_allow_search =            "on";
$auth_default_group =           "normal";
$auth_generate_password =       "off";
$auth_check_duplicates =        "on";
$auth_add_register =            "on";
$auth_email_register =          "off";
$auth_admin_from_address =      "wwwadmin\@foobar.com";
$auth_admin_email_address =     "gunther\@foobar.com";
$auth_session_length = 2;
$auth_session_dir = "./Sessions";
$auth_register_message =
   "Thanks, you may now log on with your new username
        and password.";
$auth_password_message =
  "Thanks for applying to our site, your password is";
@auth_extra_fields = ("auth_first_name",
                      "auth_last_name",
                      "auth_email");
@auth_extra_desc = ("First Name",
                    "Last Name",
                    "Email");
$auth_logon_title = "Submit BBS Logon";
$auth_logon_header =
       "Enter Logon Information For The BBS";
```

Using the GetSessionInfo Function

`GetSessionInfo` is the only routine you need to call directly in order to use the authentication library. All the other subroutines support `GetSessionInfo`. Here is a sample usage of `GetSessionInfo`:

```
($session, $username, $group,
 $firstname, $lastname, $email) =
 &GetSessionInfo($session, "$bbs_script", *in);
```

The parameters to `GetSessionInfo` are the session ID, the name of the current script, and a reference to the associative array that contains all the form variables read via the `ReadParse` function in **cgi-lib.pl**.

The `$session` variable is typically read from a hidden form variable that has been previously set. If `$session` is not set, the `GetSessionInfo` function creates one based on the user's logon. When a session has been created, the main CGI script can then pass `$session` as a hidden variable so that the next time the script is activated, the session will be available.

The second parameter contains information related to the name of the script from which the authentication library is being called. Frequently, when you're using CGI-based security, the authentication library will have the user fill out HTML forms related to logging in and registration. When this happens, the authentication library needs the original script name to figure out where to post the form information for processing. Remember that the authentication library is only a library of routines; it is not, by itself, a CGI program. It must be called from another CGI script.

The third parameter is the associative array containing all the form variable entries from the previous form. `%in` is the default array used in **cgi-lib.pl**. However, if you read your form variables into a different associative array, such as `%form_data`, you use `*form_data` to call `GetSessionInfo`. The asterisk indicates that the array is passed as a reference to the location of its value in memory rather than making another copy of the values and passing them to the subroutine, a technique that would be inefficient.

`GetSessionInfo` returns an array of values. The first value is the current valid session ID. If there was no previous session ID, `GetSessionInfo` creates one. The subsequent values in the list of returned data relate to

information about the user who has logged on, such as username, groups, first name, last name, and E-mail address.

Sample Application: HTML Filtering

This book contains a sample application that demonstrates the authentication library in the simplest possible setting. This application prevents a user from viewing certain HTML files without logging in through the authentication library. In addition, the HTML files have the word `session` filtered to include the current session ID so that further HTML files can be linked and protected using the current authentication session. Earlier, Figure 9.2 showed the directory structure of the authentication library, including the **html-auth.cgi** program.

The files **auth-lib.pl**, **auth-extra-lib.pl**, **auth-extra-html.pl**, **auth-lib-fail-html.pl**, and **auth-server-lib.pl** are part of the core authentication library. The **auth.setup** file is a sample setup file for authentication variables. The **Sessions** directory is where all the authenticated sessions are written to. The **Users** directory contains the user file for the CGI-based authentication. All the **html-auth** files—such as **html-auth.cgi**, **html-auth.setup**, **html-auth.html**, and **html-auth2.html**—belong to this sample HTML filtering and protection CGI script sample.

The setup file **html-auth.setup** is configured to use CGI-based authentication by default. Users are also allowed to register and gain immediate access to the HTML files by default. The users are stored in the **Users** directory, and the session files are stored in the **Sessions** directory.

Html-auth.cgi is the CGI program that calls the authentication library. It accepts the `file` form variable to determine which HTML file is to be read. To test this, you can call `html-auth.cgi?file=html-auth.html` as a URL in your Web browser. If this script were installed in the **cgi-bin** directory of www.foobar.com, we would refer to the script as follows:

```
http://www.foobar.com/cgi-bin/html-auth.cgi?file=html-auth.html
```

The logon screen would come up. After the user logged on, the **html-auth.html** file would be filtered and would be output to the Web browser. Figure 9.6 shows this process.

***Figure 9.6 html-auth.html** after it has been filtered by **html-auth.cgi**.*

The URL reference inside **html-auth.html** has the session ID added to it so that **html-auth.cgi** need not ask for the logon username and password again when it attempts to use **html-auth.cgi** to access the second HTML file, **html-auth2.html**. Figure 9.7 displays an example of what happens when the link is pressed. Notice, from the URL location identifier in the figure, that **html-auth.cgi** is still used to filter the next HTML file. To understand how **html-auth.cgi** works, we need to look at the code for these HTML files.

HTML-AUTH.HTML

The code that appears next is the **html-auth.html** file. It contains the HTML code for a sample Web page that you might want to protect unauthorized viewers from seeing. To activate this protection, you reference this page using the **html-auth.cgi** program as `html-auth.cgi?file=html-auth.html`.

```
<HTML>
<HEAD>
<TITLE>
Authentication Test
</TITLE>
</HEAD>
<BODY>
<H1>Authentication Test</H1>
<HR>
SESSION=> session<P>
Click
<A HREF=html-auth.cgi?file=html-auth2.html&session>
 Here </A>
for a reference to another HTML file using
the same html-auth.cgi script. Look at the
html-auth.html file for an example of how to call
other html files that you wish to protect with the
authentication library.
<P>
</BODY>
</HTML>
```

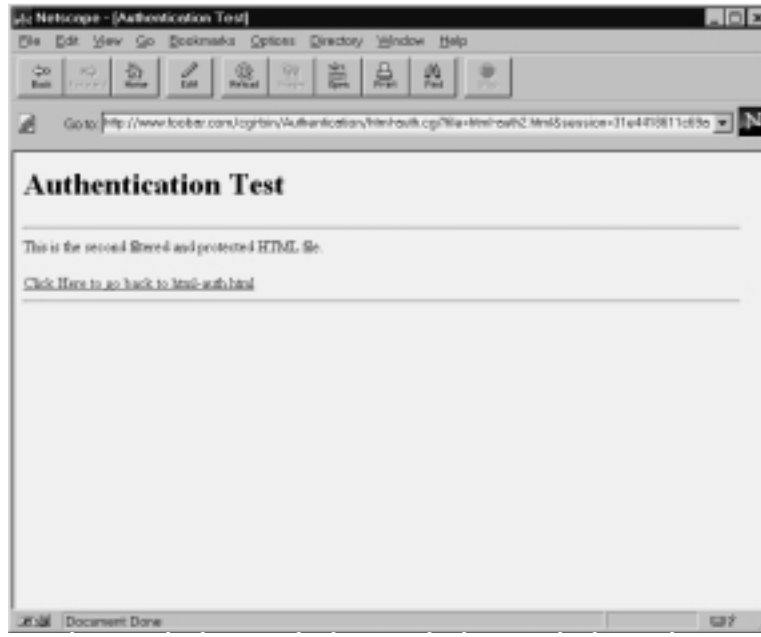

Figure 9.7 *A second* ***html-auth.cgi*** *filtered HTML file.*

For this protection of the HTML files to be effective, you should place them in a directory where the CGI script can read them but where a user cannot get to that HTML file by directly typing a URL to it.

When the user calls this HTML file using **html-auth.cgi**, a logon screen is displayed. When the user successfully enters a valid logon, the HTML file is read into the CGI script and is output to the user's Web browser. If the script sees a line that contains the word `session`, **html-auth.cgi** adds an equal symbol (=) to the end of `session` along with the session ID. Thus, if the session ID were 2000, the part of the HTML code that reads

```
<A HREF=html-auth.cgi?file=html-auth2.html
&session>
```

would be replaced with

```
<A HREF=html-auth.cgi?file=html-auth2.html
&session=2000>
```

The subsequent page, **html-auth2.html**, that the user may wish to read, would be protected by the same authentication library that protects the **html-auth.html** file without the user having to reauthenticate (log on again).

HTML-AUTH2.HTML

The **html-auth2.html** file is read by the user's Web browser if he or she clicks on the hyperlink in **html-auth.html**. It links to **html-auth.html** to demonstrate that the same session ID can be used to display many pages. This is important, because we do not wish to have the user log on to the site more than once.

```
<HTML>
<HEAD>
<TITLE>
Authentication Test
</TITLE>
</HEAD>
<BODY>
```

```
<H1>Authentication Test</H1>
<HR>
This is the second filtered and
protected HTML file.
<P>
<A HREF=html-auth.cgi?file=html-auth.html&session>
Click Here To Go Back to html-auth.html</A>
<HR>
</BODY>
</HTML>
```

A more detailed discussion of this sample application appears in the next section.

Design Discussion

The main function of the authentication library is to check to see whether a current session ID exists. If no session exists for the user, the authentication library needs to create a session. However, before a session can be created, the user needs to log in through an HTML form (for CGI-based authentication) or through the Web server's own authentication mechanism. The CGI-based authentication is more complex, because the authentication library has functionality to allow people to register as well as search for previously assigned usernames. Figure 9.8 shows the basic work flow options of this library.

auth-lib.pl

This short library file is the fulcrum of the authentication process. It contains code to see whether a session file exists and returns the information in the file if there is a valid session code. If there is no valid session file, **auth-lib.pl** reads another auxiliary library file, **auth-lib-extra.pl**, which contains all the other main functions of the authentication library. These functions are split into two Perl files for efficiency. The `GetSessionInfo` routine in **auth-lib.pl** is called every time CGI scripts execute that use authentication. However, the functions in **auth-extra-lib.pl** are called only when the initial session must be created and the user needs to log on or register for the CGI application.

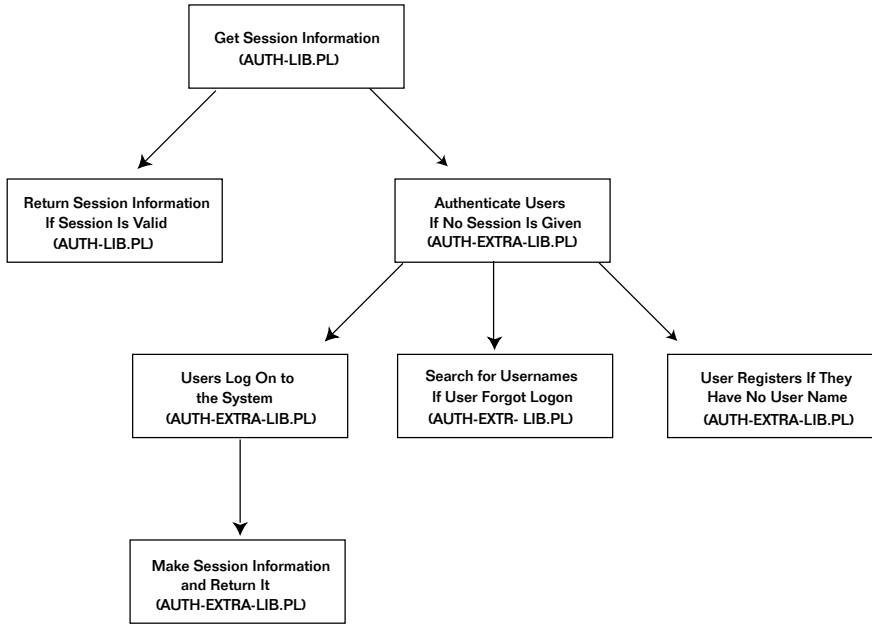

Figure 9.8 *Basic flowchart for the authentication library.*

THE GETSESSIONINFO SUBROUTINE

The `GetSessionInfo` function accepts as parameters a session number, a script name, and a reference to an associative array containing form variables. The session number tells `GetSessionInfo` whether there is a valid session file of information that it should look for. If there is no valid session number, `GetSessionInfo` logs the user on and creates a session file containing the user's information. The script name is needed for CGI-based authentication, because the logon and registration HTML forms that are displayed to the user need to post the information back to the original script. The original form variables are necessary, because the logon and registration processing used in CGI-based security needs to preserve them from screen to screen by embedding them as hidden variables in the HTML forms.

In addition to the authentication form variables, other form variables can be sent to the authentication library. For example, to activate the BBS, you send a `forum` form variable to the **bbs_forum.cgi** script. However, if the authentication library requires the user to log on via another HTML form, the `forum` form variable needs to be maintained. The authentication library does this by making sure that any form variable that does not begin with `auth_` is passed from form to form until the user logs on to the script. At that point, the original HTML form variables are sent to the script for processing.

`$session_file` is set up as a local variable to store the name of the session file. `@fields` is set up as an array of fields of user information that will be returned to the CGI script that calls `GetSessionInfo`. These fields include the session ID, username, groups, and more, depending on how the authentication library is configured.

```
sub GetSessionInfo {
local($session, $main_script, *in) = @_;
local($session_file, @fields);
```

If the session is not defined—that is, if it is equal to “”—the library does a `require` to load all the auxiliary routines for authentication and calls the `VerifyUser` routine to log the user into the system. `VerifyUser` also allows the user to register for the system as well as searches for prior used usernames depending on how security is configured for the library.

```
if ($session eq "") {
    require "$auth_lib/auth-extra-lib.pl";
    @fields = &VerifyUser($main_script, *in);
     } # End of if
```

If the session is defined, however, the program reads the session file and returns the information.

```
else {
```

`$session_file` is defined as the name of the session file. The session files consist of a session ID that is suffixed with a **.dat** extension. When the ses-

sion file is opened, it is checked for success. If the open file operation is a failure, then **auth-lib-fail-html.pl** is called; it prints an error message, and the authentication library exits. This usually happens when the session ID passed to the script is not valid. In this case, "not valid" means that there is no longer a session file associated with the session ID, or perhaps someone was trying to fake a session ID.

```
$session_file = "$session.dat";
open (SESSIONFILE, "$auth_session_dir/$session_file") ||
     (require "$auth_lib/auth-lib-fail-html.pl" && exit);
```

If the session ID is valid, the session file is read. The session file should consist of one line of fields that are pipe-delimited (separated by the pipe (|) symbol). A `chop` removes the newline at the end of the last field. (A file read typically reads the whole line, including the newline character that separates lines in a file.) Finally, the session file is closed.

```
 while (<SESSIONFILE>) {
        chop;
     @fields = split(/\|/);
     }
close (SESSIONFILE);
```

The session ID is placed at the beginning of the array of fields using the `unshift` operator. The final step in `GetSessionInfo` is to return the array of fields.

```
    unshift(@fields, $session);
     } # End of else

# return the array of fields;
@fields;
} # End of GetSessionInfo
```

auth-lib-fail-html.pl

This file contains Perl code that prints a message stating that the session file had a problem trying to open. For efficiency, this message is placed in a separate file. Most of the time, the session file will be valid. So the error code is placed outside the normal required authentication library

to make sure that only the Perl code needed to parse and compile will be interpreted. Figure 9.9 shows an example of the failure message.

```
print <<__FAILHTML__;
<HTML><HEAD>
<TITLE>Error Occurred</TITLE>
</HEAD>
<BODY>
<CENTER>
<H1>
Sorry, there appears to be a problem accessing a session file.
</H1>
</CENTER>
</BODY></HTML>
__FAILHTML__
```

***Figure 9.9 auth-lib-fail-html.pl**'s error message example.*

auth-lib-extra.pl

auth-lib-extra.pl contains all the extra routines needed to perform CGI-based authentication as well as create sessions. Because these routines

are typically called only when the session is created and a user needs to log on, they have been placed in a separate file to make loading the original library file (**auth-lib.pl**) more efficient.

In addition, the HTML output in this library file has been separated into yet another file to make it easier for you to edit the look of the screens without altering the authentication routines. Thus, when the core authentication routines are updated, it will be less likely that you will need to redo your specific HTML changes. The following code loads this extra HTML code.

```
require "$auth_lib/auth-extra-html.pl";
```

When you're modifying the Perl code that prints the HTML, be careful to escape any special characters such as @ with a backslash.

THE VERIFYUSER SUBROUTINE

The `VerifyUser` routine is the heart of the logon and user authentication process. It processes a CGI-based security logon and registration as well as server-based security. If the logon is valid, a session ID is created and the appropriate user information is passed to the `GetSessionInfo` routine discussed earlier. `VerifyUser` returns exactly the same fields as `GetSessionInfo` does, so `GetSessionInfo` can pass the information to the CGI script that called it.

As parameters, `VerifyUser` gets the main CGI script name and the reference to the associative array containing all the form variable information. Variables containing the session ID (`$session`), security groups (`$group`), username (`$username`), and all the other user information fields (`@extra_fields`) are declared in this routine for processing by the various security checking routines. In addition, `$bad_logon_message` is declared as a variable that is used to pass logon problems, such as using an invalid password, to the user as an HTML message.

```
sub VerifyUser {
    local ($main_script, *in) = @_;
    local ($session, $group, $username,
```

```
        @extra_fields,
        $bad_logon_message);
```

`$bad_logon_message` is initialized to contain no message. If the previous logon has failed, `$bad_logon_message` will contain a message to print to the user.

```
$bad_logon_message = "";
```

There are two ways of authenticating, or logging in, a user. CGI-based authentication uses this script's routines and HTML forms to log the user into the system. Server-based authentication uses the Web server's built-in authentication and relies on the environmental variable for the `REMOTE_USER` name containing a valid value set by the Web server itself.

When CGI-based authentication is used, the value of the submit buttons on the HTML forms—related to logging in, registration, searching for past usernames, and more—determine the type of processing that the authentication library should perform.

In the following code, if the form variable `auth_logon_op` has a value, the script knows that the user has pressed a button on an HTML form; the button's name was `auth_logon_op`. The only form with this button is the logon HTML form. In this case, the script calls the subroutine `&CgiLogon` to process the logon variables for username and password.

```
if ($auth_cgi eq "on") {

     if ($in{'auth_logon_op'} ne "") {
         ($bad_logon_message, $session,
          $username, $group, @extra_fields) =
           &CgiLogon($main_script, *in);
     } # End of auth_logon processing
```

Another case is `auth_search_op`. If this button is pressed, the subroutine is called to start searching for usernames. In addition, this routine is not called unless you have configured the script to allow searching using the `$auth_allow_search` variable. `auth_search_screen_op` is another form variable that is set when the HTML form should print using the `PrintSearchPage` routine for searching past usernames.

```
if ($in{'auth_search_op'} ne "" &&
    $auth_allow_search eq "on") {
    &SearchUsers($main_script, *in);
    exit;
}

if ($in{'auth_search_screen_op'} ne "" &&
    $auth_allow_search eq "on") {
    &PrintSearchPage($main_script, *in);
    exit;
}
```

When the `auth_register_op` submit button on the registration HTML form is clicked on and registration has been allowed by setting the `$auth_allow_register` variable to `on`, the `RegisterUser` function is called to register the user.

```
if ($in{'auth_register_op'} ne "" &&
    $auth_allow_register eq "on") {
    &RegisterUser($main_script, *in);
    exit;
} # End of register screen processing
```

Similarly, when the `auth_register_screen_op` submit button is pressed on the logon HTML form, `PrintRegisterPage` is called to print the HTML form for registering a user.

```
if ($in{'auth_register_screen_op'} ne "" &&
    $auth_allow_register eq "on") {
       &PrintRegisterPage;
    exit;
} # End of register screen
```

As a catch-all, the logon HTML form is printed using the `PrintLogonPage` function if the session ID has not yet been generated or if the user has pressed the `auth_logon_screen_op` submit button on one of the previous HTML forms. This ends the CGI-based security processing.

```
if ($in{'auth_logon_screen_op'} ne ""
    || ($session eq "")) {
    &PrintLogonPage($bad_logon_message,
      $main_script, *in);
    exit;
```

```
     } # End of Logon Screen

    } # End of Auth_CGI
```

The next section of the script processes Web server–based security if the `$auth_server` variable has been set previously to `on`. The username is retrieved using the `REMOTE_USER` environmental variable set by the Web server. If there is a username, the **auth-server-lib.pl** library file is executed to gather the rest of the user information.

```
    if ($auth_server eq "on") {
     $username = $ENV{'REMOTE_USER'};
     if ($username ne "") {
# The following calls a site-specific server based
# authentication routine
         require "auth-server-lib.pl";

     } # End of if username ne ""

    }  # End of if AUTH_SERVER is on
```

If neither CGI nor server-based security is used, the extra fields as well as the username to return are set to a list of blank strings. The group name is set to the default group so that a script that relies on security will be able to use the default group for security even without the rest of the user information. Finally, a session is created with this fake information by a call to `MakeSessionFile`.

The final part of this routine returns a session ID and the user information to `GetSessionInfo`.

```
if ($auth_server ne "on" && $auth_cgi ne "on") {
     @extra_fields = ();
     $username = "";
     $group = $auth_default_group;
     foreach (@auth_extra_fields) {
         push (@extra_fields, "");
     }
     $session = &MakeSessionFile(@extra_fields);
    } # End of if neither server or cgi auth is on
```

```
($session, $username, $group, @extra_fields);

} # End of VerifyUser
```

The CgiLogon Subroutine

The next routine, `CgiLogon`, attempts to log the user in to the authentication library using CGI-based security. The form variables from the HTML logon screen are interpreted against the user file to see whether the user is valid. If the logon is invalid, an error message is generated as part of the `$bad_logon_message` variable. If the logon is successful, the session is created and the user information is passed to `VerifyUser`.

In addition, variables are set aside for the logon processing. `$form_password` is the password as it was entered into the form. `$session` is the session ID. `@fields` contains the fields of user information. `$bad_logon_message` contains an error message if the logon becomes unsuccessful for any reason. And finally, the `$user_matched` variable is set to `true` if the username was found in the file. Most of these variables are set to 0 or "" initially. `$form_password` is set to the form variable for the password that was entered on the HTML logon form.

```
sub CgiLogon {
    local ($main_script, *in) = @_;
    local($password, $username, $group, @extra_fields);
    local($form_password, $session, @fields);
    local($bad_logon_message);
    local($user_matched);

    $bad_logon_message = "";
    $session = "";
    @fields = ();
    $form_password = $in{"auth_password"};
```

Next, the file containing user information is opened to scan whether the logon information matches one of the records. The `$user_matched` flag is set to 0 initially to indicate that no match yet exists.

The `while` loop continually reads each line of the file. Each line is then `split` into individual user fields that are separated by the pipe symbol in

the file. If the username that the user submitted is the same as a username in the file, the `$user_matched` variable is set to 1. Then the password that was entered is compared to the password in the user file. The password that was entered into the HTML logon form has been passed through the `AuthEncryptWrap` routine to encrypt the password in the same way that it appears in the file. If the password that the user entered has not been encrypted in this manner, it does not match the encrypted version that is already stored in the user file.

If the passwords match, the user fields are `split` into the `@fields` array. Because the password is the first element of the array and because we do not need to return its value, we strip it off the array by using the `shift` function. Finally, a session file is created with `MakeSessionFile` using this newly gathered user information.

```
open (USERFILE, "$auth_user_file");

$user_matched = 0;
while (<USERFILE>) {
 chop($_);
 ($password, $username, $group, $first_name,
  $last_name, $email) = split(/\|/, $_);
 if ($in{'auth_user_name'} eq $username) {
     $user_matched = 1;
 if (&AuthEncryptWrap($form_password, $password) eq
     $password) {
     @fields = split(/\|/, $_);
     shift(@fields); # Get rid of password field
     $session = &MakeSessionFile(@fields);
 } # End of if passwords match
 } # End of if username matches
} # End of While
close (USERFILE);
```

If no session was created but the username was found, the user probably did not have a matching password, so the program puts this information into the `$bad_logon_message` variable. If the username was not found, the user is notified and a different message is placed in `$bad_logon_message`. Finally, the `$bad_logon_message`, `$session`, and `@fields` user information is returned to `VerifyUser` for further processing.

```
    if ($user_matched == 1 && $session eq "") {
     $bad_logon_message =
         "<p><strong>Sorry, Your password did not";
     $bad_logon_message .=
         " match the username.</strong><p>";
    }
    if ($user_matched == 0) {
     $bad_logon_message =
         "<p><strong>Sorry, Your username was not";
     $bad_logon_message .=
         " found.</strong><p>";

    }
($bad_logon_message, $session, @fields);

} # End of CgiLogon
```

The MakeSessionFile Subroutine

The `MakeSessionFile` routine creates the session file that stores the current user information. `GetSessionInfo` later uses this session file to retrieve information about the currently logged-on user every time the current session ID is sent to the routine. This routine starts by accepting a list of fields that make up the user information and then returns the newly acquired session ID that is associated with the session file.

```
sub MakeSessionFile
{
local(@fields) = @_;
local($session, $session_file);
```

The first thing `MakeSessionFile` does is to call a routine (`RemoveOldSessions`) to remove old session files that are no longer being used. Then a new session ID is generated. We do this by generating a random number. The random number is first seeded in the `srand` function by combining the value of the process ID and the current time variable using the `or` operator (`|`). Finally, this random number, the time, and the process ID are converted to a long hexadecimal number that serves as the new session ID. A hexadecimal number is made up of digits that include the numbers 0 through 9 and the letters A through F instead of the 0 through 9 digits found in the

base 10 system. The session filename consists of the session ID plus a **.dat** extension.

```
&RemoveOldSessions;

# Seed the random generator
srand($$|time);
$session = int(rand(60000));
# pack the time, process id, and random $session into a
# hex number which will make up the session id.
$session = unpack("H*", pack("Nnn", time, $$, $session));

$session_file = "$session.dat";
```

Next, the session file is opened for creation in the directory specified by the `$auth_session_dir` variable, which is set in the script that uses this library. The user information fields are joined by the pipe symbol and are written to the file. Finally, the file is closed and the newly formed session code is returned from the subroutine.

```
open (SESSIONFILE,">$auth_session_dir/$session_file")
    || &CgiDie("Could Not Create Session File\n");
print SESSIONFILE join ("\|", @fields);
print SESSIONFILE "\n";

close (SESSIONFILE);

$session;

} # End of MakeSessionFile
```

The RemoveOldSessions Subroutine

The `RemoveOldSessions` procedure goes into the `$auth_session_dir` directory and removes all files that are older than `$auth_session_length` days. These variables are set by the script that calls this library. The `@files` array contains all the filenames in the current directory. `$file` is a temporary variable that holds the filename of the current file the program is checking for age.

The directory is opened using the `opendir` command, and the files in the directory are read into an array using the `readdir` command. The output from `readdir` is passed to Perl's internal `grep` function to make sure that the special filenames "." and ".." escape the removal process.

```
sub RemoveOldSessions
{
local(@files, $file);
# Open up the session directory.
opendir(SESSIONDIR, "$auth_session_dir") ||
    &CgiDie("Session Directory Would Not Open\n");
# read all entries except "." and ".."
@files = grep(!/^\.\.?$/,readdir(SESSIONDIR));
closedir(SESSIONDIR);
```

Each file is then checked for age using the `-M` operator. This operator returns the age of the file in days. If this age is greater than `$auth_ session_length`, the `unlink` function is called to delete the file.

```
foreach $file (@files)
        {
# If it is older than auth_session_length, delete it
        if (-M "$auth_session_dir/$file" >
               $auth_session_length)
                {
                unlink("$auth_session_dir/$file");
                }

        }
} # End of RemoveOldSessions
```

The SearchUsers Subroutine

`SearchUsers` searches for the users in the user file based on E-mail address and returns the username that matches the address. Sometimes users may forget their username, so this function provides a mechanism to allow them to find it again. `SearchUsers` accepts the main CGI script as a parameter so that it can print an HTML form that posts data to the original script. For the same reason, it also is passed the current form data in an associative array. The final output of this routine is an HTML form listing the found usernames if any were found.

$user_match keeps track of the currently matched usernames that are found later in the routine. $field_num is used to keep track of the element in the @auth_extra_fields array that contains the auth_email field. $username keeps track of the current username as the user file is searched. @extra_fields contains the fields that are read from the user file one line at a time. $auth_email is the form variable for the current E-mail address to search for. $user_list converts the usernames from the $user_match variable to an HTML list of names for later output to the user's Web browser. $form_tags contains a list of form variables passed by the previous CGI script that are not authentication-related. The library prints these as hidden variables on the forms it outputs so that the original CGI script form variables do not get lost during the CGI-based logon process.

```
sub SearchUsers {
    local($main_script, *in) = @_;
    local($user_match);
    local($field_num, $username, @extra_fields);
    local($auth_email);
    local($user_list);

    local($form_tags);

    $form_tags = &PrintCurrentFormVars(*in);
```

The first thing this function does is to open the user file for reading. The E-mail address being searched for is placed in the $auth_email variable, is changed to lowercase, and has spaces removed. This technique allows the search to be successful later even if the E-mail address is entered with a different case or with spaces embedded between characters. $user_match is set to nothing, because the program has not yet found any variables.

```
    open (USERFILE, "$auth_user_file") ||
        &CgiDie("Could Not Open User File\n");

    $auth_email = $in{'auth_email'};
    $auth_email =~ tr/A-Z/a-z/;
    $auth_email =~ s/ //g;
    $user_match = "";
```

The next part of this function iterates through the `@auth_extra_fields` array to find which element number in the list contains the `auth_email` field. Because the fields in `@extra_fields` are user-defined, you may have decided to use a different order of fields than is provided in the example. The index to the location in the array is stored in `$field_num`.

```
$field_num = 0;
foreach (@auth_extra_fields) {
 if ($_ eq "auth_email") {
     last;
 }
 $field_num++;
}
```

The file containing user information is parsed line by line until a match is found in the E-mail address. First, the extra newline is truncated from each line as it is read. Then any uppercase characters are converted to lowercase using the `tr` function. All the fields are `split` into an array using the pipe delimiter. The three fields are removed from the array, because they always consist of the password, username, and group information. They never contain the E-mail address. Finally, the E-mail address in the current user record is compared to the E-mail address that the user is searching for. If they match, the username is stored in the `$user_match` variable.

```
while (<USERFILE>) {
 chop($_);
 tr/A-Z/a-z/;
 @extra_fields = split(/\|/, $_);
 $username = $extra_fields[1];
 # Get rid of the first three fields
 shift(@extra_fields);
 shift(@extra_fields);
 shift(@extra_fields);
 if ($auth_email eq $extra_fields[$field_num]) {
     $user_match .= $username . "|";
 } # End of email match
} # End of While
close (USERFILE);
```

Next, the procedure checks to see whether there are any matches after the entire user file has been searched. If there are matches, they are converted

to HTML code consisting of the list of usernames separated by the <P> HTML tag.

```
if ($user_match ne "") {
 $user_list = $user_match;
 $user_list =~ s/\|/<p>/g;
}
```

Figure 9.10 *Example of a successful search result.*

If no users were found, `HTMLPrintNoSearchResults` is called to print an HTML form that displays a message. If users were found, `HTMLPrintSearchResults` displays an HTML form with the newly found usernames. Figure 9.10 contains an example of the search results that are returned if a successful match is found.

```
if ($user_match eq "") {
 &HTMLPrintNoSearchResults($main_script,
                  $form_tags);
} else {
 &HTMLPrintSearchResults ($main_script,
```

```
                    $form_tags,
                    $user_list);

    } # End of no users matched

} # End of SearchUsers
```

The RegisterUser Subroutine

The `RegisterUser` function is one of the more complex procedures in the authentication library. Many different setup variables affect how `RegisterUser` processes a user's enrollment in the library. Like many of the other functions here, `RegisterUser` accepts the current script name and form variables as parameters. The output of the function is an HTML form indicating whether the registration was a success or a failure. In addition, depending on how the setup variables have been configured, this function sends E-mail notification that a registration has occurred, or, if the program has been configured to choose a password automatically, it sends E-mail to users telling them their generated password.

`@form_vars` keeps track of the form variables that are used in the registration process, such as the username, last name, first name, and other fields. `$group` contains the security group the user is given. This variable is initialized to the value contained in `$auth_default_group`. `@f` is an array that is used as a placeholder for fields in the routine. `@write_fields` is the array of items that is written to the user file upon a successful registration. `$password` contains the user's password. `$form_tags` is a placeholder for the HTML form variables that are passed to the original script. `$user_matched` is used as a flag to see whether the user already exists in the user file. `$userfile` is the name of the file that contains user data. `$user_email` is the E-mail address of the user. `$real_password` is the clear text version of the user's password. The `$password` variable is encrypted, so `$real_password` stores the nonencrypted version for further processing. `$random` is a randomly generated number from which a salt is generated for the encryption routine. A *salt* is basically a random two-character string added to the encryption routine. `$salt` contains the salt for the encryption routine. The encryption routine's behavior changes when different salts are added to it, providing extra security.

```
sub RegisterUser {
    local($main_script, *in) = @_;
    local($x, @form_vars, $group, @f);
    local(@write_fields, $password);
    local($form_tags, $user_matched);
    local($userfile,$user_email,$real_password);
    local($random,$salt);
    $form_tags = &PrintCurrentFormVars(*in);
    $group = $auth_default_group;
```

`@form_vars` contains a list of HTML form variables from the registration screen. If `$auth_generate_password` is `off`, the HTML registration screen prompts the user for a password and a confirmation of the password. These password variables are pushed into `@form_vars` in this case.

```
@form_vars = ('auth_user_name', @auth_extra_fields);

    if ($auth_generate_password ne "on") {
     push(@form_vars, "auth_password1");
     push(@form_vars, "auth_password2");
    }
```

The `%in` associative array contains all the form variables. In this case, all the variables in `@form_vars` are checked to see that they have a value and that the value contains no pipe symbols. Because pipe symbols delimit individual fields in the user file, problems would be caused if users were allowed to register using pipe symbols inside the fields. If there is a problem, the `HTMLPrintRegisterNoValidValues` routine is called to print the error message and the program exits. In addition, if there is an E-mail address designated with the `auth_email` form variable, that value is placed in the `$user_email` variable.

```
    foreach $x (@form_vars) {
     if ($in{"$x"} eq "" || $in{"$x"} =~ /\|/) {
         &HTMLPrintRegisterNoValidValues($main_script,
                              $form_tags);
         exit;
     } # End of if
     if ($x eq "auth_email") {
         $user_email = $in{"$x"};
     }
    } # End of Foreach form variable
```

If the password is not automatically generated, the passwords that are entered on the HTML registration form must match. Otherwise, an error HTML page is generated and the program exits.

```
if ($in{'auth_password1'} ne $in{'auth_password2'}) {
    &HTMLPrintRegisterNoPasswordMatch($main_script,
                          $form_tags);
    exit;
}
```

If `$auth_check_duplicates` is turned on, the user file is opened and scanned for duplicate usernames. In addition, this same routine is used again on the alternative user file (`$auth_alt_user_file`) if it is defined. `$user_matched` is set to 1 if any duplicates are found in the files. Setting `$user_matched` to 1 triggers the program to print an HTML error page and exit.

```
if ($auth_check_duplicates eq "on") {

     open (USERFILE, "$auth_user_file");
     $user_matched = 0;
     while (<USERFILE>) {
         chop($_);
         @f = split(/\|/, $_);
         if ($f[1] eq $in{'auth_user_name'}) {
          $user_matched = 1;
          last;
         }
     } # End of while userfile open
     close (USERFILE);

# Check for duplicates in the alternative
# file if it is defined.
     if ($auth_alt_user_file ne "") {
         open (USERFILE, "$auth_alt_user_file");

         while (<USERFILE>) {
          chop($_);
          @f = split(/\|/, $_);
          if ($f[1] eq $in{'auth_user_name'}) {
              $user_matched = 1;
              last;
          }
         } # End of while userfile open
```

```
        close (USERFILE);
    } # End of auth_alt_user_file

    if ($user_matched == 1) {
        &HTMLPrintRegisterFoundDuplicate($main_script,
                        $form_tags);
        exit;
    } # End of user matched (found duplicate)
  } # End of if check duplicates on
```

Because the random number generator is used several times in this subroutine, it is seeded in advance with the combined `$$` (process ID) and current `time` variable. The `random` variable is a placeholder for a string that contains the list of characters from which we choose a password or encryption salt.

```
srand(time|$$);
$random = "abcdefghijklmnopqrstuvwxyz1234567890";
```

When `$auth_generate_password` is `off`, the `$password` variable is set to one of the form variables. Otherwise, `$password` is randomly generated based on a string consisting of six alphabetic characters or numbers. `$real_password` is then set to the value of `$password`, because `$password` will be encrypted.

```
  if ($auth_generate_password ne "on") {
   $password = $in{"auth_password1"};
  } else {
   $password = "";
   for (1..6) {
          $password .= substr($random,int(rand(36)),1);
      }
  }
  $real_password = $password;
```

The salt for the encryption function is a two-character string that changes the behavior of the encryption routine. The encrypted password appears differently, depending on which salt is used. The program calls the `AuthEncryptWrap` routine instead of Perl's `crypt` routine, because `crypt` is not supported in some operating systems. This makes it easy to turn off

or replace the encryption routine, because it is encapsulated in its own subroutine.

```
    $salt = "";
for (1..2) {
     $salt .= substr($random,int(rand(36)),1);
    }
    $password = &AuthEncryptWrap ($password,
                      $salt);
    @write_fields = ($password, $in{'auth_user_name'},
                  foreach (@auth_extra_fields) {
     push(@write_fields, $in{"$_"});
    }
```

If the `$auth_add_register` variable is set to `on`, the user information is added to the user file. The information is instead added to the alternative user file if `$auth_alt_user_file` is specified in the library configuration. Whenever information is added to the user file, we use `AuthGetLock` to lock it first to make sure no one else is writing to the file at the same time. The file is then written to and the lock is removed using `AuthReleaseFileLock`. As with the Perl `crypt` routine, the script uses encapsulated routines instead of Perl's `flock` routine, because `flock` is not supported on some operating systems. In fact, `flock` is not even supported on all flavors of UNIX.

```
if ($auth_add_register eq "on") {

# Lock the file to make sure no one else will write to
# it if anyone else registers at the same time
#
&AuthGetFileLock ("$auth_user_file.lock");
    $user_file = $auth_user_file;
    if ($auth_alt_user_file ne "") {
     $user_file = $auth_alt_user_file;
    }
open (USERFILE, ">>$user_file");

print USERFILE join("\|", @write_fields) . "\n";
close (USERFILE);
&AuthReleaseFileLock ("$auth_user_file.lock");

} # End of auth_add_register
```

If the `$auth_email_register` variable is `on` and if your E-mail address has been specified, the authentication library E-mails you with the fact that the user registered on your system. Note that this part of the code requires the **mail-lib.pl** library discussed in an earlier chapter to send the message.

```
if ($auth_email_register eq "on" &&
    $auth_admin_from_address ne "" &&
    $auth_admin_email_address ne "") {
    require "$auth_lib/mail-lib.pl";
local($subject, $message);
$subject = "Register User";

$message = join(",", @write_fields) . "\n";
&send_mail($auth_admin_from_address,
     $auth_admin_email_address,
     $subject, $message);
} # End of Email Register
```

If the password is generated by the authentication library and the registration has been successful so far, the program E-mails the user's new password using the **mail-lib.pl** library. Finally, an HTML form is output to the user's Web browser indicating that the registration was a success.

```
if ($auth_generate_password eq "on") {
     if ($auth_email_register ne "on") {
         require "$auth_lib/mail-lib.pl";
     }
     $subject = "Registered User Password";
     $message = $auth_password_message;
     $message .= " $real_password.\n\n\n";
     &send_mail($auth_admin_from_address,
                $user_email,
                $subject, $message);
    }
    &HTMLPrintRegisterSuccess ($main_script,
                      $form_tags);

} # End of RegisterUser
```

The PrintCurrentFormVars Subroutine

`PrintCurrentFormVars` takes all form variables not related to authentication and returns a string containing the HTML code for hidden input fields relat-

ed to those form variables. The routine differentiates between the library form variables and regular CGI script variables because the authentication library variables always begin with `auth_`. The hidden input HTML tags are included with the forms that the library prints so that when the original CGI script is called again, the original form variables will still be intact.

```
sub PrintCurrentFormVars {
    local(*form_vars) = @_;
    local($x,$y,$form_tags);
    $form_tags = "";
    foreach $x (keys %form_vars) {
     if (!($x =~ /auth_/i)) {
         $y = $form_vars{"$x"};
         $form_tags .= qq!<input type=hidden name=$x
          value="$y">\n!;
     }
    }
    $form_tags;
} # PrintCurrentFormTags
```

The AuthGetFileLock Subroutine

The `AuthGetFileLock` function first looks to see whether a lock file exists. If there is a lock file, the routine waits for it to be released and then creates its own. A lock file ensures that no instance of the same program is writing to the user file at the same time. In case the program that originally made the lock file has crashed or has been killed for some reason before it could release the file, `AuthGetFileLock` waits for the file to be released only 60 seconds. After that, it is assumed that the file can be removed. The reason a hand-rolled lock file routine is used instead of the Perl `flock` function is that `flock` is not supported by some operating system platforms. Comments are included indicating the function that might be used if you decided to implement the `flock` routine.

```
sub AuthGetFileLock {
    local ($lock_file) = @_;

    local ($endtime);
    $endtime = 60;
    $endtime = time + $endtime;
#   We set endtime to wait 60 seconds
```

```
# The $endtime is used for a timeout of how long we
# want to keep waiting for the lock if someone else
# already has it open.
    while (-e $lock_file && time < $endtime) {
        # Do Nothing
    }
    open(LOCK_FILE, ">$lock_file");
#    flock(LOCK_FILE, 2);
} # end of AuthGetFileLock
```

The AuthReleaseFileLock Subroutine

`AuthReleaseFileLock` is the opposite of the `AuthGetFileLock` routine. It removes the lock file using the `unlink` Perl function. If you decide to reprogram the routine to use `flock` instead, there are comments indicating the appropriate `flock` function to use.

```
sub AuthReleaseFileLock {
    local ($lock_file) = @_;

# 8 unlocks the file
#    flock(LOCK_FILE, 8);
    close(LOCK_FILE);
    unlink($lock_file);

} # end of ReleaseFileLock
```

The AuthEncryptWrap Subroutine

`AuthEncryptWrap` encrypts a string with the salt that is provided as a parameter and returns the newly encrypted string. If `$auth_use_cleartext` is `on`, the encryption does not occur. If your operating system does not support the Perl `crypt` function, you can write your own encryption routine inside this function or set the `$auth_use_cleartext` variable to `on` when you configure the authentication library.

```
sub AuthEncryptWrap {
    local ($field, $salt) = @_;

# If auth_use_cleartext is on, then we do not
# encrypt the password.
    if ($auth_use_cleartext ne "on") {
    $field = crypt ($field, $salt);
```

```
    }

    $field;

} # end of encrypt
```

auth-extra-html.pl

auth-extra-html.pl contains the routines that print the HTML forms for logon, registration, error messages, and other screens for the authentication library. By separating the HTML-related functions into their own file, you decrease the likelihood that this file will have to be replaced if there is an update to the logic in the main authentication routines that process the registration and logons. This is a benefit if you decide to make extensive cosmetic changes and do not wish to repeat the process whenever there is a minor bug fix or script update.

Typically, the routines here accept the current set of form variables and main CGI script name as parameters. This information is needed so that the HTML form can direct the user's Web browser to post the information to the original CGI script that uses the authentication library. In addition, the current CGI script form variables are passed from screen to screen to maintain the information that the user wanted to pass to the original CGI script. The variable `$form_tags` is generally used to store the HTML code related to hidden input fields that contain this previous CGI form information.

The PrintLogonPage Subroutine

`PrintLogonPage` prints the HTML form that logs the user in to the authentication library. `$bad_logon_message` is used as a parameter and contains an error message if the logon HTML page is being printed again as a result of a problem logging in. `$register_tag` is set up to contain the HTML code for the registration button if `$auth_allow_register` is set to `on` during configuration of CGI-based security. Similarly, `$search_tag` is used to store the HTML code for the search button if `$auth_allow_search` is set to `on`.

```
sub PrintLogonPage {
    local($bad_logon_message, $main_script, *in) = @_;
```

```
    local($form_tags);
    local($register_tag);
    local($search_tag);
```

You can override the logon page title and header by setting the `$auth_logon_title` and `$auth_logon_header` variables as part of the authentication library setup. If these variables are not set, defaults are chosen for them.

```
    if (length($auth_logon_title) < 1) {
        $auth_logon_title = "Submit Logon";
    }

    if (length($auth_logon_header) < 1) {
        $auth_logon_header = "Enter Your Logon Information";
    }
```

The following piece of code sets up the `$register_tag` and `$search_tag` variables with HTML code for the register and search buttons if they are allowed in your setup using `$auth_allow_register` and `$auth_allow_search`.

```
    $register_tag = "";
    $search_tag = "";

    if ($auth_allow_register eq "on") {
     $register_tag = <<__END_OF_REG_TAG__;
<input type=submit name=auth_register_screen_op
value="Register For An Account"><p>
__END_OF_REG_TAG__
}

    if ($auth_allow_search eq "on") {
     $search_tag = <<__END_OF_SEARCH_TAG__;
<input type=submit name=auth_search_screen_op
value="Search For Old Account"><p>
__END_OF_SEARCH_TAG__
    }
```

`$form_tags` is set to the HTML code of hidden input fields that are set to the form variables passed to the original CGI script. Finally, the HTML code is printed with all the variables that have been set.

```
   $form_tags = &PrintCurrentFormVars(*in);

    print <<__END_OF_LOGON__;
<HTML><HEAD>
<TITLE>$auth_logon_title</TITLE>
</HEAD>
<BODY>
<CENTER>
<H1>$auth_logon_header</h1>
<hr>
$bad_logon_message
<FORM METHOD=POST ACTION=$main_script>
$form_tags
<TABLE>
<TR><TH>Username</TH>
<TD><INPUT TYPE=TEXT NAME=auth_user_name></td></tr>
<tr><th>Password</th>
<td><input type=password name=auth_password></td></tr>
</TABLE><p>
<input type=submit name=auth_logon_op
value="Log On To The System"><p>
$register_tag
$search_tag
<hr>
</center>
</form>
</body>
</HTML>
__END_OF_LOGON__

} # End of PrintLogonPage
```

The PrintSearchPage Subroutine

`PrintSearchPage` prints the HTML form that asks users to enter the E-mail address to search for usernames on. This function sets up the hidden form tags for the original CGI script form variables and then prints the relevant HTML form.

```
sub PrintSearchPage {
local($main_script,*in) = @_;
    local($form_tags);
    $form_tags = &PrintCurrentFormVars(*in);
    print <<__END_OF_SEARCH__;
<HTML><HEAD>
<TITLE>Search For An Account</TITLE>
```

```
</HEAD>
<BODY>
<CENTER>
<H1>Search For Matching Username</h1>
<hr>
<h2>Enter Your Email Address To Search For A Matching Username</h2>
<FORM METHOD=POST ACTION=$main_script>
$form_tags
<TABLE>
<TR><TH>Email</TH>
<TD><INPUT TYPE=TEXT NAME=auth_email></td></tr>
</TABLE>
<p>
<input type=submit name=auth_search_op value="Submit Search">
<input type=submit name=auth_logon_screen_op value="Return to Logon
Screen">
<P>
</center>
</form>
</body>
</HTML>
__END_OF_SEARCH__

} # End of PrintSearchPage
```

The HTMLPrintSearchResults Subroutine

`HTMLPrintSearchResults` prints the HTML code related to a successful search for usernames. Again, this routine merely prints the HTML form that contains the variables related to the hidden HTML tags that correspond to the originally passed CGI script variables.

```
sub HTMLPrintSearchResults {
    local($main_script, $form_tags, $user_list) =
     @_;
     print <<__END_SEARCHRESULTS__;
<HTML><HEAD>
<TITLE>User Found</TITLE>
</HEAD>
<BODY>
<CENTER>
<H1>User Was Found In The Search</h1>
<hr>
<h2>List Of Users</h2>
<strong>$user_list</strong>
<FORM METHOD=POST ACTION=$main_script>
$form_tags
```

```
<input type=submit name=auth_logon_screen_op
value="Return to Logon Screen">
<hr>
</center>
</form>
</body>
</HTML>
__END_SEARCHRESULTS__

} # End of HTMLPrintSearchResults
```

The HTMLPrintNoSearchResults Subroutine

`HTMLPrintNoSearchResults` is just like `HTMLPrintSearchResults` except that it is called if no matching usernames were found.

```
sub HTMLPrintNoSearchResults {
    local($main_script, $form_tags) = @_;
     print <<__END_NOSEARCHRESULTS__;
<HTML><HEAD>
<TITLE>No Users Found</TITLE>
</HEAD>
<BODY>
<CENTER>
<H1>Sorry, No Users Found</h1>
<hr>
<h2>Sorry, No users were found that matched your email address</h2>
<FORM METHOD=POST ACTION=$main_script>
$form_tags
<input type=submit name=auth_logon_screen_op
value="Return to Logon Screen">
<hr>
</center>
</form>
</body>
</HTML>
__END_NOSEARCHRESULTS__

} # End HTMLPrintNoSearchResults
```

The PrintRegisterPage Subroutine

`PrintRegisterPage` prints the HTML form that lets the user enter and submit registration information. If the passwords are not automatically generat-

ed, the `$password_input` variable will contain HTML code for the password input fields. Similarly, if the password is being generated, the `$password_message` variable will contain a message that is sent to the user on the HTML form. The message tells users that the password will be sent via E-mail.

```
sub PrintRegisterPage {
local($main_script,*in) = @_;
local($form_tags);
local($more_form_input,$password_input, $x);
    $form_tags = &PrintCurrentFormVars(*in);
local ($password_message);

#
# We also check for the extra fields and output HTML
# asking for input on the extra fields.
#
$more_form_input = "";
for ($x = 0; $x <= @auth_extra_fields - 1; $x++) {
    $more_form_input .= <<__END_OF_EXTRA_FIELDS__;
<TR><TH>$auth_extra_desc[$x]</TH>
<TD><INPUT TYPE=TEXT NAME=$auth_extra_fields[$x]></td></tr>
__END_OF_EXTRA_FIELDS__
}
$password_input = "";
if ($auth_generate_password ne "on") {
    $password_input = <<__END_OF_PASSWORD_FIELDS__;
<tr><th>Password</th>
<td><input type=password name=auth_password1></td></tr>
<tr><th>Password Again</th>
<td><input type=password name=auth_password2></td></tr>
__END_OF_PASSWORD_FIELDS__
}
$password_message = "";
if ($auth_generate_password eq "on") {
$password_message = <<__PASSWORDMSG__;
Your password will be automatically generated and sent
to you via your E-mail address.
__PASSWORDMSG__
}
    print <<__END_OF_REGISTER__;
<HTML><HEAD>
<TITLE>Register For An Account</TITLE>
</HEAD>
<BODY>
<CENTER>
```

```
<H1>Enter The Registration Information</h1>
<hr>
<FORM METHOD=POST ACTION=$main_script>
$form_tags
<TABLE>
<tr><th>User Name</th>
<td><input type=Username name=auth_user_name></td></tr>
$password_input
$more_form_input
</TABLE>
<p>
<input type=submit name=auth_register_op value="Submit Information">
<input type=submit name=auth_logon_screen_op value="Return to Logon
Screen">
<P>
$password_message
</center>
</form>
</body>
</HTML>
__END_OF_REGISTER__

} # End of PrintRegisterPage
```

THE HTMLPRINTREGISTERSUCCESS SUBROUTINE

If the user's registration process is successful, a new HTML form reports the success using `HTMLPrintRegisterSuccess`. This simple form leads users back to the logon screen if they press the **Return To Logon Screen** button. Figure 9.11 is an example of this success screen.

```
sub HTMLPrintRegisterSuccess {
    local($main_script, $form_tags) =
     @_;
    print <<__END_OF_REGISTER_SUCCESS__;
<HTML><HEAD>
<TITLE>Registration Added</TITLE>
</HEAD>
<BODY>
<CENTER>
<H2>You have been added to the user database</h2>
</center>
<hr>
<FORM METHOD=POST ACTION=$main_script>
$form_tags
```

```
<BLOCKQUOTE>
    $auth_register_message
</Blockquote>
<center>
<input type=submit name=auth_logon_screen_op value="Return to Logon
Screen")
</center>
</form>
</body>
</HTML>
__END_OF_REGISTER_SUCCESS__

} # End of RegisterSuccess
```

Figure 9.11 *HTML form indicating a successful registration has occurred.*

THE HTMLPRINTREGISTERFOUNDDUPLICATE SUBROUTINE

`HTMLPrintRegisterFoundDuplicate` prints an HTML form indicating that the user's registration was a failure because the username was already contained in the user file.

```
sub HTMLPrintRegisterFoundDuplicate {
    local($main_script, $form_tags) =
     @_;
print <<__END_OF_REGISTER_DUPLICATE__;
<HTML><HEAD>
<TITLE>Problem with Registration</TITLE>
</HEAD>
<BODY>
<CENTER>
<H1>Problem with Registration</h1>
</center>
<hr>
<FORM METHOD=POST ACTION=$main_script>
$form_tags
<BLOCKQUOTE>
Sorry, your username is already in the database
</Blockquote>
<center>
<input type=submit name=auth_logon_screen_op value="Return to Logon
Screen")
</center>
</form>
</body>
</HTML>
__END_OF_REGISTER_DUPLICATE__

} # End of HTMLPrintRegisterFoundDuplicate
```

The HTMLPrintRegisterNoPassword Subroutine

`HTMLPrintRegisterNoPassword` prints an HTML form indicating that the registration failed because the two passwords that the user entered did not match. Two separate password entries are necessary, because this field does not echo the original characters the user typed. Therefore, a second field is used to confirm the first password and make sure that a typo did not occur.

```
sub HTMLPrintRegisterNoPasswordMatch {
    local($main_script, $form_tags) =
     @_;

    print <<__END_OF_REGISTER_NOMATCH__;
<HTML><HEAD>
<TITLE>Problem with Registration</TITLE>
</HEAD>
```

```
<BODY>
<CENTER>
<H1>Problem with Registration</h1>
</center>
<hr>
<FORM METHOD=POST ACTION=$main_script>
$form_tags
<BLOCKQUOTE>
Sorry, the two passwords you typed in did not match.
</Blockquote>
<center>
<input type=submit name=auth_logon_screen_op value="Return to Logon
Screen")
</center>
</form>
</body>
</HTML>
__END_OF_REGISTER_NOMATCH__

} # End of HTMLPrintRegisterNoPasswordMatch
```

The HTMLPrintRegisterNoValidValues Subroutine

`HTMLPrintRegisterNoValidValues` prints yet another registration failure HTML form. This one is printed if the user has left out any values in the registration process.

```
sub HTMLPrintRegisterNoValidValues {
    local ($main_script, $form_tags) =
     @_;
    print <<__END_OF_REGISTER_NOVALUE__;
<HTML><HEAD>
<TITLE>Problem with Registration</TITLE>
</HEAD>
<BODY>
<CENTER>
<H1>Problem with Registration</h1>
</center>
<hr>
<FORM METHOD=POST ACTION=$main_script>
$form_tags
<BLOCKQUOTE>
Sorry, you need to enter a valid value for every field
</Blockquote>
<center>
<input type=submit name=auth_logon_screen_op value="Return to Logon
```

```
Screen")
</center>
</form>
</body>
</HTML>
__END_OF_REGISTER_NOVALUE__

} # End of HTMLPrintRegisterNoValidValues
```

auth-server-lib.pl

auth-server-lib.pl is called by the `VerifyUser` function in **auth-extra-lib.pl** if the method of security is Web server–based instead of CGI-based. This file has been broken out separately, because getting user information beyond the username in Web server–based security is usually quite server- and application-specific. For example, on a company intranet in which employees log into to the Web server itself, the rest of their information may need to be interfaced with an SQL database such as Sybase or Oracle from an employee database.

The following example code sets up a sample first name, last name, E-mail address, and security group that the user belongs to. If you were configuring your own server-based security, you would probably recode this routine to query a file or database of employee information. The final bit of the code calls the standard `MakeSessionFile` routine to create the session with the newly acquired user information. This call to `MakeSessionFile` must be included regardless of how you get the user information.

```
$firstname = "Gunther";
$lastname = "Birznieks";
$email = "gunther\@gunther.com";
$group = "$auth_default_group";

$session =
     &MakeSessionFile( ($username, $group, $firstname,
     $lastname,$email));
```

html-auth.cgi

html-auth.cgi is the sample CGI program included with the authentication library to demonstrate what **auth-lib.pl** is capable of. **html-auth.setup**

contains the sample authentication setup for this script. the purpose of **html-auth.cgi** is to protect certain HTML pages from being viewed unless the user logs in to the site.

Basically, you pass the name of the HTML file you wish to protect as a URL parameter to **html-auth.cgi**. For example, if you wished to protect **mypage.html**, you would call the program using the URL `html-auth.cgi?file=mypage.html`. If this program were in the **cgi-bin** of `www.foobar.com`, you would use the following URL:

```
http://www.foobar.com/cgi-bin/html-auth.cgi?file=mypage.html
```

If you have many pages that you wish to protect, you also need to reference the hyperlinks with this program and add the suffix `&session` to the URL parameter that is linked. This program filters the HTML page so that the tag `&session` is replaced, and the form variable is set equal to the current real session ID. The authentication library uses this session ID to make sure that the user has access to the page instead of asking the user to log in to read every page.

We append `&session` instead of `?session` to the html-auth.cgi program URL because there is already a parameter (`file`) that html-auth.cgi expects to see. Thus, the natural continuation of the URL is with an `&`. A `?` is not used because the `?` has already separated the script name from the `file` URL variable.

If you are referencing a CGI script that uses the authentication library—such as the bulletin board script discussed in this book—then you would refer to it in a protected HTML page. For the BBS example, to access the "Open" forum you would use a URL such as `bbs_forum.cgi?forum=open&session` so that the session ID would also be passed to the **bbs_forum.cgi** script.

`$lib` is set to the path where you place your library files. Then Steven Brenner's **cgi-lib.pl** is used to read the form variables `file` and `session`. Before the authentication library is processed, **html-auth.setup** is loaded using the `require` command.

```
$lib = ".";
require "$lib/cgi-lib.pl";
```

```
#
# The following should be a setup file with your
# authorization variables.  In this case,
# we use test.setup for testing purposes.
#
require "html-auth.setup";
require "$auth_lib/auth-lib.pl";

print &PrintHeader;
&ReadParse;
```

The session is read from the form variable `session` first. Then `GetSessionInfo` is called. Note that the main script name is **html-auth.cgi** in this case. In addition, the current form variables are passed to the authentication library. Remember, the **html-auth.cgi** script `requires` the `file` form variable to be sent in order to know which file to parse and print to the user's Web browser. Thus, by passing the form variables, the authentication library makes sure that the `file` variable is passed from authentication screen to screen using hidden variables until this script is called into action again after the user successfully logs in.

```
$session = $in{"session"};

($session, @fields) =
    &GetSessionInfo($session, "html-auth.cgi", *in);
```

`$htmlfile` is set to the file form variable. If there is no form variable `file`, an HTML form is printed that lets the user know that a parameter is required. Otherwise, the file is opened. A `while` loop reads every line of the file into memory. If the line contains the word `session`, it is replaced with `session=[THE REAL SESSION ID]`. For example, if the session ID were 1234, the word `session` would be replaced with `session=1234`. This is done to preserve the current session between protected pages. Finally, the altered line is printed to the user's Web browser. This process continues until the end of the HTML file and then the program ends.

```
$htmlfile = $in{'file'};

if ($htmlfile ne "") {
     open(HTMLFILE, "$htmlfile") ||
```

```
       &CgiDie("Could not open $htmlfile");

    while (<HTMLFILE>) {
        if (/session/) {
         s/session/session=$session/;
        }
        print $_;
    }

    close (HTMLFILE);
} else {
```

The following code prints the HTML code that tells users how to use this program if they forgot to leave off the `file=` parameter on the URL that called this script.

```
print <<__END_OF_HTML__;
<HTML>
<HEAD>
<TITLE>
Authentication HTML Filter
</TITLE>
</HEAD>
<BODY>
<H1>Authentication HTML Filter</H1>
<HR>
<BLOCKQUOTE>
<STRONG>Error: </STRONG> You forgot to include
a "file=" parameter on the URL of this CGI script.
You need this parameter to tell the
html-auth.cgi program which HTML files
to protect from viewing as well as filter
session IDs in the file.
<P>
For example, you may want to use
a URL in the form of:
<P>
http://www.foobar.com/cgi-bin/html-auth.cgi?file=html-auth.html
</BLOCKQUOTE>
<HR>
</BODY>
</HTML>

__END_OF_HTML__
 } # End of if $htmlfile is there
```