Restricting Access to Your Applications

Lesson 13a

This lesson explains how to use Apache (on the server side) to restrict access to parts of a Web site based on the identity of the user or on information about the request. On the application side of things, you can create your own mechanism for user validation and check the validity of your users through cookies.

In this lesson, you will learn

- How to restrict access based on the user, client IP address, domain name, and browser version
- How to use the user management tools provided with Apache
- How to store and retrieve cookie information
- How to use cookies for authentication

Authentication Overview

Authorization and authentication are common requirements for many Web sites. Authentication establishes the identity of parties in a communication. You can authenticate yourself through something you know like a password or a cookie, through something tangible such as an ID card or a key, through something intrinsically part of you like your fingerprint or your retina, or through any combination of these elements. In the context of a Web site, authentication is usually restricted to the use of passwords and certificates.

Authorization deals with protecting access to resources. You can authorize access based on several factors, such as the IP address the user is coming from, the user's browser type, the content the user is trying to access, or who the user is (previously determined via authentication).

Apache includes several modules that provide authentication and access control and that can be used to protect both dynamic and static content. You can either use one of these modules or implement your own access control at the application level and provide customized login screens, single sign-on, and other advanced functionality.

Client Authentication

Users are authenticated for tracking or authorization purposes. The HTTP specification provides two authentication mechanisms: basic and digest. In both cases, the process is the following:

1. A client tries to access restricted content in the Web server.

2. Apache checks whether the client is providing a username and password. If not, Apache returns an `HTTP 401` status code, indicating user authentication is required.

3. The client reads the response and prompts the user for the required username and password (usually with a pop-up dialog box).

4. The client retries accessing the Web page, this time transmitting the username and password as part of the HTTP request. The client remembers the username and password and transmits them in later requests to the same site, so the user does not need to retype them for every request.

5. Apache checks the validity of the credentials and grants or denies access based on the user identity and other access rules.

In the basic authentication scheme, the username and password are transmitted in clear text, as part of the HTTP request headers. This poses a security risk because an attacker could easily peek at the conversation between server and browser, learn the

username and password, and reuse them freely afterward.

The digest authentication provides increased security because it transmits a digest instead of the clear text password. The digest is based on a combination of several parameters, including the username, password, and request method. The server can calculate the digest on its own and check that the client knows the password, even when the password itself is not transmitted over the network.

NOTE:

A digest algorithm is a mathematical operation that takes a text and returns another text, a digest, which uniquely identifies the original one. A good digest algorithm should make sure that, at least for practical purposes, different input texts produce different digests and that the original input text cannot be derived from the digest. MD5 is the name of a commonly used digest algorithm.

Unfortunately, although the specification has been available for quite some time, only very recent browsers support digest authentication. This means that for practical purposes, digest authentication is restricted to scenarios in which you have control over the browser software of your clients, such as in a company intranet.

In any case, for both digest and basic authentication, the requested information itself is transmitted unprotected over the network. A better choice to secure access to your Web site involves using the HTTP over SSL protocol, as described in lesson "Setting Up a Secure Web Server."

User Management Methods

When the authentication module receives the username and password from the client, it needs to verify that they are valid against an existing repository of users. The usernames and passwords can be stored in a variety of ways, including the file- and database-based mechanisms provided for by Apache. Third-party modules provide support for additional mechanisms such as Lightweight Directory Access Protocol (LDAP) and Network Information Services (NIS).

Apache Authentication Module Functionality

Apache provides the basic framework and directives to perform authentication and access control. The authentication modules provide support for validating passwords against a specific back end method (file, database, and so on). Users can optionally be organized in groups, easing management of access control rules.

Apache provides three built-in directives related to authentication that is used with any of the authentication modules: `AuthName, AuthType`, and `Require`.

`AuthName` accepts a string argument, the name for the authentication realm. A realm is a logical area of the Web server that you are asking the password for. It will be displayed in the browser pop-up window.

`AuthType` specifies the type of browser authentication: basic or digest.

`Require` enables you to specify a list of users or groups that will be allowed access. The syntax is `Require user` followed by one or more usernames, or `Require group` followed by one or more group names. For example

```
Require user joe bob
```

or

```
Require group employee contractor
```

If you want to grant access to anyone who provides a valid username and password, you can do so with

```
Require valid-user
```

With the preceding directives, you can control who has access to specific virtual hosts, directories, files, and so on. Although authentication and authorization are separate concepts, in practice they are tied together in Apache. Access is granted based on specific user identity or group membership. Some third-party modules, such as certain LDAP-based modules, allow for clearer separation between authentication and authorization.

The authentication modules included with Apache provide

- Back-end storage Provides text or database files containing the username and group information
- User management Supplies tools for creating and managing users and groups in the back-end storage
- Authoritative information Specifies whether the results of the module are authoritative

NOTE:

Sometimes users will not be allowed access to a particular realm because their information is not found in the user database provided by the module, or because no authentication rules matched their information. In that case, one of two situations will occur:

- If the module specifies its results as authoritative, a user will be denied access and Apache will return an error.

- If the module specifies its results as not authoritative, other modules can have a chance of authenticating the user.

This enables you to have a main authorization module that knows about most users, and to be able to have additional modules that can authenticate the rest of the users.

File-Based Authentication

The `mod_auth` Apache module provides basic authentication via text files containing usernames and passwords, similar to how traditional Unix authentication works with the `/etc/passwd` and `/etc/groups` files.

Back-End Storage

When using back-end storage methods, you need to specify the file containing the list of usernames and passwords and, optionally, the file containing the list of groups.

The `users` file is a Unix-style password file, containing names of users and encrypted passwords. The entries look like the following, on Unix, using the crypt algorithm:

```
admin:iFrlxqg0Q6RQ6
```

and on Windows, using the MD5 algorithm:

```
admin:$apr1$Ug3.....$jVTedbQWBKTfXsn5jK6UX/
```

The `groups` file contains a list of groups and the users who belong to each one of them, separated by spaces, such as in the following entry:

```
web: admin joe Daniel
```

The `AuthUserFile` and the `AuthGroupFile` directives take a path argument, pointing to the `users` file and the `groups` file. The `groups` file is optional.

User Management

The Apache distribution includes the `htpasswd` utility on Unix and `htpasswd.exe` on Windows; they are designed to help you manage user password files. Both versions are functionally identical, but the Windows version uses a different method to encrypt the password. The encryption is transparent to the user and administrator. On Linux/Unix, the first time you add a user, you need to type

```
/usr/local/apache2/bin/htpasswd -c file userid
```

where `file` is the password file that will contain the list of usernames and passwords, and `userid` is the username you want to add. You will be prompted for a password, and the file will be created. For example, on Linux/Unix, the command

```
/usr/local/apache2/bin/htpasswd -c /usr/local/apache2/conf/htusers
admin
```

will create the password file `/usr/local/apache2/conf/htusers` and add the `admin` user.

Similar functionality exists on Windows, where the command-line operation might look something like the following:

```
htpasswd -c "C:\Program Files\Apache Group\Apache2\conf\htusers"
admin
```

The `-c` command-line option tells the `htpasswd` executable that it should create the file. When you want to `add` users to an existing password file, do not use the `-c` option; otherwise, the file will be overwritten.

It is important that you store the password file outside the document root and thus make it inaccessible via a Web browser. Otherwise, an attacker could download the file and get a list of your usernames and passwords. Although the passwords are encrypted, when you have the file, it is possible to perform a brute-force attack to try to guess them.

Authoritative

The `AuthAuthoritative` directive takes a value of `on` or `off`. By default, it is `on`, meaning that the module authentication results are authoritative. That is, if the user is not found or does not match any rules, access will be denied.

Using `mod_auth`

Listing 1 shows a sample configuration, restricting access to the `private` directory in the document root to authenticated users present in the `htusers` password file. Note that the optional `AuthGroupFile` directive is not present.

Listing 1. File-Based Authentication Example

```
1: <Directory /usr/local/apache2/htdocs/private>
2: AuthType Basic
3: AuthName "Private Area"
4: AuthUserFile /usr/local/apache2/conf/htusers
5: AuthAuthoritative on
6: Require valid-user
7: </Directory>
```

Database File-Based Access Control

Storing usernames and passwords in plain text files is convenient, but this method does not scale well. Apache would need to open and read the files sequentially to look for a particular user. When the number of users grows, this operation becomes very time-consuming. The `mod_auth_dbm` module enables you to replace the text-based files with indexed database files, which can handle a much greater number of users without performance degradation. The `mod_auth_dbm` module is included with Apache but is not enabled by default. Enabling this module occurs when configuring Apache to be built, using the `--enable-module=dbm` option.

Back-End Storage

The `mod_auth_dbm` module provides two directives, `AuthDBMUserFile` and `AuthDBMGroupFile`, that point to the `database` files containing the usernames and groups. Unlike plain text files, both directives can point to the same file, which combines both users and groups.

User Management

Apache provides a Perl script (`dbmmanage` on Unix and `dbmmanage.pl` on Windows) that allows you to create and manage users and groups stored in a database file. Under Unix, you might need to edit the first line of the script to point to the location of the Perl interpreter in your system. On Windows, you need to install the additional MD5 password package. If you are using ActiveState Perl, start the Perl package manager and type

```
install Crypt-PasswdMD5
```

To add a user to a database on Linux/Unix, type

```
dbmmanage dbfile adduser userid
```

On Windows, type

```
perl dbmmanage.pl dbfile adduser userid
```

You will be prompted for the password, and the user will be added to the existing database file or a new file will be created if one does not exist.

When adding a user, you can optionally specify the groups it belongs to as comma-separated arguments. The following command adds the user `daniel` to the database file `/usr/local/apache2/conf/dbmusers` and makes it a member of the groups `employee` and `engineering`:

```
dbmmanage /usr/local/apache2/conf/dbmusers adduser daniel
employee,engineering
```

If you ever need to delete the user `daniel`, you can issue the following command:

```
dbmmanage dbfile delete daniel
```

The `dbmmanage` program supports additional options. You can find complete syntax information in the `dbmmanage` manual page or by invoking `dbmmanage` without any arguments.

NOTE:

Apache 2.0 provides an additional utility, `htdbm`, that does not depend on Perl and provides all the functionality that `dbmmanage` does.

Using Apache for Access Control

The `mod_access` module, enabled by default, allows you to restrict access to resources based on parameters of the client request, such as the presence of a specific header or the IP address or hostname of the client.

Implementing Access Rules

You can specify access rules using the `Allow` and `Deny` directives. Each of these directives takes a list of arguments such as IP addresses, environment variables, and domain names.

Allow/Deny Access by IP Addresses

You can deny or grant access to a client based on its IP address:

```
Allow from 10.0.0.1 10.0.0.2 10.0.0.3
```

You can also specify IP address ranges with a partial IP address or a network/mask pair. Additionally, you can specify the first one, two, or three bytes of an IP address. Any IP address containing those will match this rule. For example, the rule

```
Deny from 10.0
```

will match any address starting with 10.0, such as 10.0.1.0 and 10.0.0.1.

You can also utilize the IP address and the netmask; the IP address specifies the network and the mask specifies which bits belong to the network prefix and which ones belong to the nodes. The rule

```
Allow from 10.0.0.0/255.255.255.0
```

will match IP addresses 10.0.0.1, 10.0.0.2, and so on, to 10.0.0.254.

You can also specify the network mask via high-order bits. For example, you could write the previous rule as

```
Allow from 10.0.0.0/24
```

Allow/Deny Access by Domain Name

You can control access based on specific hostnames or partial domain names. For example, `Allow from example.com` will match `www.example.com`, `foo.example.com`, and so on.

NOTE:

Enabling access rules based on domain names forces Apache to do a reverse DNS lookup on the client address, bypassing the settings of the `HostNameLookups` directive. This has performance implications.

Allow/Deny Access Based on Environment Variables

You can specify access rules based on the presence of a certain environment variable by prefixing the name of the variable with the string `env=`. You can use this feature to grant or deny access to certain browsers or browser versions, to prevent specific sites from linking to your resources, and so on. For this example to work as intended, the client needs to transmit the `User-Agent` header.

For example

```
BrowserMatch MSIE iexplorer
Deny from env=iexplorer
```

Because the client sends the `User-Agent` header, it could possibly be omitted or manipulated, but most users will not do so and this technique will work in most cases.

Allow/Deny Access to All Clients

The keyword `all` matches all clients. You can specify `Allow from all` or `Deny from all` to grant or deny access to all clients.

Evaluating Access Rules

You can have several `Allow` and `Deny` access rules. You can choose the order in which the rules are evaluated by using the `Order` directive. Rules that are evaluated later have higher precedence. `Order` accepts one argument, which can be

`Deny,Allow`, `Allow,Deny`, or `Mutual-Failure`. `Deny,Allow` is the default value for the `Order` directive. Note that there is no space in the value.

```
Deny,Allow
```

`Deny,Allow` specifies that `Deny` directives are evaluated before `Allow` directives. With `Deny,Allow`, the client is granted access by default if there are no `Allow` or `Deny` directives or the client does not match any of the rules. If the client matches a `Deny` rule, it will be denied access unless it also matches an `Allow` rule, which will take precedence because `Allow` directives are evaluated last and have greater priority.

Listing 2 shows how to configure Apache to allow access to the `/private` location to clients coming from the internal network or the domain `example.com` and deny access to everyone else.

Listing 2. Sample `Deny,Allow` Access Control Configuration

```
1: <Location /private>
2: Order Deny,Allow
3: Deny from all
4: Allow from 10.0.0.0/255.255.255.0 example.com
5: </Location>
```

`Allow,Deny`

`Allow,Deny` specifies that `Allow` directives are evaluated before `Deny` directives. With `Allow,Deny`, the client is denied access by default if there are no `Allow` or `Deny` directives or if the client does not match any of the rules. If the client matches an `Allow` rule, it will be granted access unless it also matches a `Deny` rule, which will take precedence.

Note that the presence of `Order Allow,Deny` without any `Allow` or `Deny` rules causes all requests to the specified resource to be denied because the default behavior is to deny access.

Listing 3 allows access to everyone except a specific host.

Listing 3. Sample `Allow,Deny` Access Control Configuration

```
1: <Location /some/location/>
2:  Order Allow,Deny
3:  Allow from all
4:  Deny from host.example.com
5: </Location>
```

`Mutual-Failure`

In the case of `Mutual-Failure`, the host will be granted access only if it matches an `Allow` directive and does not match any `Deny` directive.

Combining Apache Access Methods

In previous sections, you learned how to restrict access based on user identity or request information. The `Satisfy` directive enables you to determine whether both types of access restrictions must be satisfied in order to grant access. `Satisfy` accepts one parameter, which can be either `all` or `any`.

`Satisfy all` means that the client will be granted access if it provides a valid username and password and passes the access restrictions. `Satisfy any` means the client will be granted access if it provides a valid username and password or passes the access restrictions.

Why is this directive useful? For example, you might want to provide free access to your Web site to users coming from an internal, trusted address, but require users coming from the Internet to provide a valid username and password. Listing 4 demonstrates just that.

Listing 4. Mixing Authentication and Access Control Rules

```
1: <Location /restricted>
2: Allow from 10.0.0.0/255.255.255.0
3: AuthType Basic
4: AuthName "Intranet"
5: AuthUserFile /usr/local/apache2/conf/htusers
6: AuthAuthoritative on
7: Require valid-user
8: Satisfy any
9: </Location>
```

NOTE:

Access control based on connection or request information is not completely secure. Although it provides an appropriate level of protection for most cases, the rules rely on the integrity of your DNS servers and your network infrastructure. If an attacker gains control of your DNS servers, or your routers or firewalls are incorrectly configured, he can easily change authorized domain name records to point to his machine or pretend he is coming from an authorized IP address.

Limiting Access Based on HTTP Methods

In general, you want your access control directives to apply to all types of client requests, and this is the default behavior. In some cases, however, you want to apply authentication and access rules to only certain HTTP methods such as `GET` and `HEAD`.

The `<Limit>` container takes a list of methods and contains the directives that apply to requests containing those methods. The complete list of methods that can be used is `GET, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE, PATCH, PROPFIND, PROPPATCH, MKCOL, COPY, MOVE, LOCK`, and `UNLOCK`.

The `<LimitExcept>` section provides complementary functionality, containing directives that will apply to requests not containing the listed methods.

Listing 5 shows an example from the default Apache configuration file. The `<Limit>` and `<LimitExcept>` sections allow read-only methods but deny requests to any other methods that can modify the content of the file system, such as `PUT`. For more information on the myriad options available here, see the Apache documentation at http://httpd.apache.org/docs-2.0/mod/core.html.

Listing 5. Restricting Access Based on Rule

```
 1:  <Directory /home/*/public_html>
 2:    AllowOverride FileInfo AuthConfig Limit
 3:    Options MultiViews Indexes SymLinksIfOwnerMatch IncludesNoExec
 4:    <Limit GET POST OPTIONS PROPFIND>
 5:      Order Allow,Deny
 6:      Allow from all
 7:    </Limit>
 8:    <LimitExcept GET POST OPTIONS PROPFIND>
 9:      Order Deny,Allow
10:      Deny from all
11:    </LimitExcept>
12:  </Directory>
```

In the next section, you'll learn about restricting access on the application side based on information found in cookies.

Restricting Access Based on Cookie Values

In, "Working with Cookies and User Sessions," you learned all about the structure of a cookie and how to set and access cookie variables in PHP. The next few sections will show some practical uses of cookies for authentication purposes.

Suppose you created a login form that checked for values against a database. If the user is authorized, you send a cookie that says as much. Then, for all pages you want to restrict only to authorized users, you check for the specific cookie. If the cookie is present, the user can see the page. If the cookie is not present, the user is either sent back to the login form, or a message regarding access restrictions can be printed to the screen.

We'll go through each of these steps in the next few sections.

Creating the Authorized Users Table

When you're integrating user accounts into a Web-based application, it is common to store the user-specific information in a database table. The information in this table can then be used to authorize the user and grant access to areas of the site that are specifically for these "special" users.

The following table creation command will create a table called `auth_users` in your MySQL database, with fields for the ID, first name, last name, email address, username, and password:

```
mysql> create table auth_users (
  -> id int not null primary key auto_increment,
```

```
  -> f_name varchar(50),
  -> l_name varchar(50),
  -> email varchar(150),
  -> username varchar(25),
  -> password varchar (75)
  ->);
Query OK, 0 rows affected (0.03 sec)
```

The following `INSERT` command puts a record in the `auth_users` table for a user named John Doe, with an email address of `john@doe.com`, a username of `jdoe`, and a password of `doepass`:

```
mysql> insert into auth_users values ('', 'John', 'Doe',
'john@doe.com',
  -> 'jdoe', password('doepass'));
Query OK, 1 row affected (0.00 sec)
```

This `INSERT` command should be self-explanatory, with the exception of the use of the `password()` function. When this function is used in the `INSERT` command, what is stored in the table is in fact not the actual password, but a hash of the password.

When you view the contents of the `auth_users` table, you will see the hash in the password field, as follows:

```
mysql> select * from auth_users;
+----+--------+--------+--------------+----------+------------------+
| id | f_name | l_name | email        | username | password         |
+----+--------+--------+--------------+----------+------------------+
|  1 | John   | Doe    | john@doe.com | jdoe     | 2fae5c9d478ec4b1 |
+----+--------+--------+--------------+----------+------------------+
1 row in set (0.00 sec)
```

Although it may look like it is encrypted, a hash is in fact not an encrypted bit of information. Instead, it is a "fingerprint" of the original information. Hashes are generally used, like fingerprints, to perform matches. In this case, when you check your user's password, you will be checking that the hash of the input matches the stored hash. Using hashes alleviates the need and security risk of storing actual passwords.

Creating the Login Form and Script

After you authorize users in your table, you need to give them a mechanism for proving their authenticity. In this case, a simple two-field form will do, as shown in Listing 6.

Listing 6. User Login Form

```
 1: <html>
 2: <head>
 3: <title>Login Form</title>
 4: </head>
```

```
 5: <body>
 6: <H1>Login Form</H1>
 7: <FORM METHOD="POST" ACTION="userlogin.php">
 8: <P><STRONG>Username:</STRONG><BR>
 9: <INPUT TYPE="text" NAME="username"></p>
10: <P><STRONG>Password:</STRONG><BR>
11: <INPUT TYPE="password" NAME="password"></p>
12: <P><INPUT TYPE="SUBMIT" NAME="submit" VALUE="Login"></P>
13: </FORM>
14: </body>
15: </html>
```

Put these lines into a text file called `userlogin.html`, and place this file in your Web server document root. Next, you'll create the script itself, which the form expects to be called `userlogin.php` (see Listing 7).

Listing 7. User Login Script

```
 1: <?php
 2: //check for required fields from the form
 3: if ((!$_POST[username]) || (!$_POST[password])) {
 4:   header("Location: userlogin.html");
 5:   exit;
 6: }
 7:
 8: //connect to server and select database
 9: $conn = mysql_connect("localhost", "joeuser", "somepass")
10:  or die(mysql_error());
11: mysql_select_db("testDB",$conn) or die(mysql_error());
12:
13: //create and issue the query
14: $sql = "select f_name, l_name from auth_users where username =
15:  '$_POST[username]' AND password = password('$_POST[password]')";
16: $result = mysql_query($sql,$conn) or die(mysql_error());
17:
18: //get the number of rows in the result set; should be 1 if a
match
19: if (mysql_num_rows($result) == 1) {
20:
21:          //if authorized, get the values of f_name l_name
22:          $f_name = mysql_result($result, 0, 'f_name');
23:          $l_name = mysql_result($result, 0, 'l_name');
24:
25:          //set authorization cookie
26:          setcookie("auth", "1", 0, "/", "yourdomain.com", 0);
27:
28:        //create display string
29:        $display_block = "<P>$f_name $l_name is authorized!</p>
30:        <P>Authorized Users' Menu:
31:        <ul>
32:        <li><a href=\"secretpage.php\">secret page</a>
33:        </ul>";
34:
35: } else {
36:
37:          //redirect back to login form if not authorized
38:          header("Location: userlogin.html");
39:          exit;
40: }
```

```
41: ?>
42: <HTML>
43: <HEAD>
44: <TITLE>User Login</TITLE>
45: </HEAD>
46: <BODY>
47: <? echo "$msg"; ?>
48: </BODY>
49: </HTML>
```

Put these lines into a text file called `userlogin.php`, and place this file in your Web server document root. In a moment, you'll try it out, but first let's examine what the script is doing.

Line 3 checks for the two required fields the only two fields in the form:

`$_POST[username]` and `$_POST[password]`. If either of these fields is not present, the script will redirect the user back to the original login form. If the two fields are present, the script moves along to lines 9 - 11, which connect to the database server and select the database to use, in preparation for issuing the SQL query to check the authenticity of the user. This query, and its execution, is found in lines 14 - 16. Note that the query checks the hash of the password input from the form against the password stored in the table. These two elements must match each other, and also belong to the username in question, in order to authorize the user.

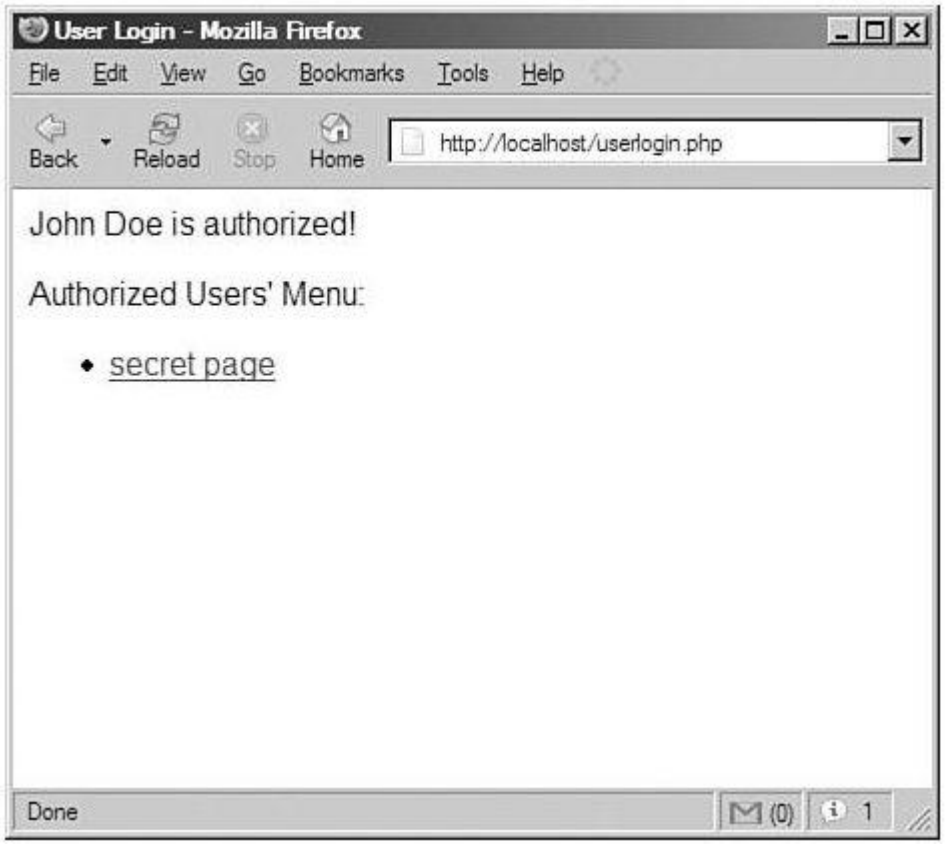
Line 19 tests the result of the query by counting the number of rows in the result set. The row count should be exactly 1 if the username and password pair represents a valid login. If this is the case, the `mysql_result()` function is used in lines 22 - 23 to extract the first and last names of the user. These names are used for aesthetic purposes only.

Line 26 sets the authorization cookie. The name of the cookie is `auth` and the value is `1`. If a `0` is put in the time slot, the cookie will last as long as this user's Web browser session is open. When the user closes the browser, the cookie will expire. Lines 28 - 34 create a message for display, including a link to a file we will create in a moment.

Finally, lines 35 - 40 handle a failed login attempt. In this case, the user is simply redirected back to the original login form.

Go ahead and access the login form, and input the valid values for the John Doe user. When you submit the form, the result should look like Figure 1.

Figure 1. Successful login result.



Try to log in with an invalid username and password pair, and you should be redirected to the login form. In the next (and final) section, you will create the `secretpage.php script`, which will `read` the authentication cookie you have just set and act accordingly.

Testing for the `auth` Cookie

The last piece of this puzzle is to use the value of the `auth` cookie to allow a user to access a private file. In this case, the file in question is shown in Listing 8.

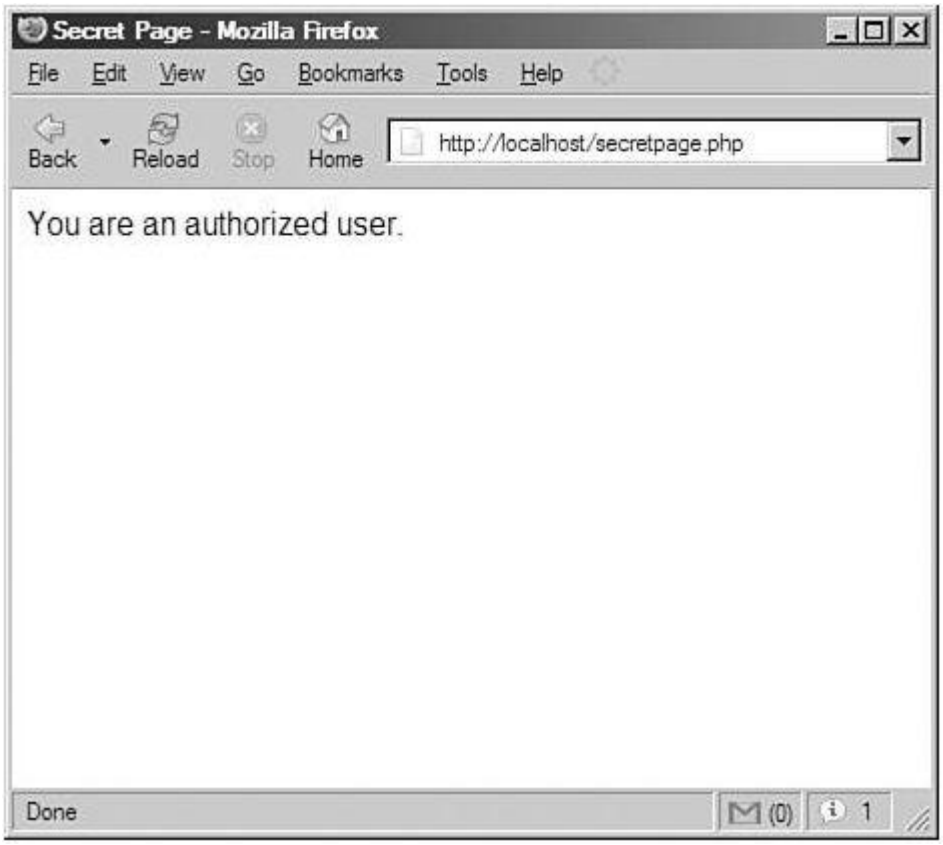
Listing 8. Checking for `auth` Cookie

```
1: <?php
2: if ($_COOKIE[auth] == "1") {
3:        $display_block = "<p>You are an authorized user.</p>";
4: } else {
5:          //redirect back to login form if not authorized
6:          header("Location: userlogin.html");
7:          exit;
8: }
9. ?>
```

```
10: <html>
11: <head>
12: <title>Secret Page</title>
13: </head>
14: <body>
15: <?php echo "$display_block"; ?>
16: </body>
17: </html>
```

From the menu shown in Figure 1, click the `secret page` link. Because you are an authorized user, you should see a result like Figure 2.

Figure 2. Accessing the secret page as an authorized user.



Close your browser and attempt to access `secretpage.php` directly. You will find that you cannot, and will be redirected to the original login form because the authentication cookie has not set.