Improving Session Security

Because important information is normally stored in a session (you should never store sensitive data in a cookie), security becomes more of an issue. With sessions there are two areas to pay attention to: the session ID, which is a reference point to the session data, and the session data itself, stored on the server. A malicious person is far more likely to hack into a session through the session ID than the data on the server, so I'll focus on that side of things here. In the tips at the end of this section I mention two ways to protect the session data itself.

The session ID is the key to the session data. By default, PHP will store this in a cookie, which is preferable from a security standpoint. It is possible in PHP to use sessions without cookies, but that leaves the application vulnerable to *session hijacking*: If malicious user Alice can learn user Bob's session ID, Alice can easily trick a server into thinking that Bob's session ID is also *Alice's* session ID. At that point, Alice would be riding the coattails of Bob's session and would have access to Bob's data. Storing the session ID in a cookie makes it somewhat harder to steal.

Changing the Session Behavior

As part of PHP's support for sessions, there are over 20 different configuration options you can set for how PHP handles sessions. For the full list, see the PHP manual, but I'll highlight a few of the most important ones here. Note two rules about changing the session settings:

1. All changes must be made before calling **`session_start()`**.
2. The same changes must be made on every page that uses sessions.

Most of the settings can be set within a PHP script using the **`ini_set()`** function (discussed in Chapter 8):

Click here to view code image

```
ini_set(parameter, new_setting);
```

For example, to require the use of a session cookie (as mentioned, sessions can work without cookies but it's less secure), use

Click here to view code image

```
ini_set('session.use_only_cookies', 1);
```

Another change you can make is to the name of the session (perhaps to use a more user-friendly one). To do so, call the **`session_name()`** function:

```
session_name('YourSession');
```

The benefits of creating your own session name are twofold: it's marginally more secure and it may be better received by the end user (since the session name is the cookie name the end user will see). The **`session_name()`** function can also be used when deleting the session cookie:

Click here to view code image

```
setcookie(session_name(),'', time()-3600);
```

If not provided with an argument, this function instead returns the current session name.

Finally, there's also the **`session_set_cookie_params()`** function. It's used to tweak the settings of the session cookie:

Click here to view code image

```
session_set_cookie_params(expire, path, host, secure, httponly);
```

Note that the expiration time of the cookie refers only to the longevity of the cookie in the browser, not to how long the session data will be stored on the server.

One method of preventing hijacking is to store some sort of user identifier in the session, and then to repeatedly double-check this value. The *HTTP_USER_AGENT*—a combination of the browser and operating system being used—is a likely candidate for this purpose. This adds a layer of security in that one person could hijack another user's session only if they are both running the exact same browser and operating system. As a demonstration of this, let's modify the examples one last time.

To use sessions more securely:

1. Open **`login.php`** (refer to Script 12.8) in your text editor or IDE.
2. After assigning the other session variables, also store the *HTTP_USER_AGENT* value (**Script 12.12**):

Click here to view code image

```
$_SESSION['agent'] = sha1
→($_SERVER['HTTP_USER_AGENT']);
```

Script 12.12 This final version of the **login.php** script also stores an encrypted form of the user's *HTTP_USER_AGENT* (the browser and operating system of the client) in a session.

Click here to view code image

```
1   <?php # Script 12.12 - login.php #4
2   // This page processes the login form submission.
3   // The script now stores the HTTP_USER_AGENT value for added security.
4
5   // Check if the form has been submitted:
6   if ($_SERVER['REQUEST_METHOD'] == 'POST') {
7
8      // Need two helper files:
9      require('includes/login_functions.
       inc.php');
10     require('../mysqli_connect.php');
11
12     // Check the login:
13     list ($check, $data) = check_
       login($dbc, $_POST['email'],
       $_POST['pass']);
14
15     if ($check) { // OK!
16
17        // Set the session data:
18        session_start();
19        $_SESSION['user_id'] =
          $data['user_id'];
20        $_SESSION['first_name'] =
          $data['first_name'];
21
22        // Store the HTTP_USER_AGENT:
23        $_SESSION['agent'] = sha1
          ($_SERVER['HTTP_USER_AGENT']);
24
25        // Redirect:
26        redirect_user('loggedin.php');
27
28     } else { // Unsuccessful!
29
30        // Assign $data to $errors for
          login_page.inc.php:
31        $errors = $data;
32
33     }
34
35     mysqli_close($dbc); // Close the database connection.
36
37  } // End of the main submit conditional.
38
39  // Create the page:
40  include('includes/login_page.inc.php');
41  ?>
```

The *HTTP_USER_AGENT* is part of the **$_SERVER** array (you may recall using it way back in Chapter 1, "Introduction to PHP"). It will have a value like *Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1…*).

Instead of you storing this value in the session as is, it'll be run through the **sha1()** function for slightly improved security. That function returns a 32-character hexadecimal string (called a *hash*) based on a value. In theory, no two strings will have the same **sha1()** result.

3. Save the file and place it in your web directory.
4. Open **loggedin.php** (Script 12.9) in your text editor or IDE.
5. Change the **!isset($_SESSION['user_id'])** conditional to (**Script 12.13**):

Click here to view code image

```
if (!isset($_SESSION['agent']) OR
→($_SESSION['agent'] != sha1
→($_SERVER['HTTP_USER_AGENT']) )) {
```

Script 12.13 This **loggedin.php** script now confirms that users accessing this page have the same *HTTP_USER_AGENT* as they did when they logged in.

Click here to view code image

```
1   <?php # Script 12.13 - loggedin.php #3
2   // The user is redirected here from login.php.
3
4   session_start(); // Start the session.
5
6   // If no session value is present, redirect the user:
7   // Also validate the HTTP_USER_AGENT!
8   if (!isset($_SESSION['agent'])
    OR ($_SESSION['agent'] != md5($_
    SERVER['HTTP_USER_AGENT']) )) {
9
10     // Need the functions:
11     require('includes/login_functions.inc.php');
12     redirect_user();
13
14  }
15
16  // Set the page title and include the
    HTML header:
17  $page_title = 'Logged In!';
18  include('includes/header.html');
19
20  // Print a customized message:
21  echo "<h1>Logged In!</h1>
22  <p>You are now logged in,
    {$_SESSION['first_name']}!</p>
23  <p><a href=\"logout.php\">Logout</a>
    </p>";
24
25  include('includes/footer.html');
26  ?>
```

This conditional checks two things. First, it sees if the **$_SESSION['agent']** variable is not set (this part is just as it was before, although *agent* is being used instead of *user_id*). The second part of the conditional checks if the **sha1()** version of **$_SERVER['HTTP_USER_AGENT']** does not equal the value stored in **$_SESSION['agent']**. If either of these conditions is true, the user will be redirected.

6. Save this file, place it in your web directory, and test in your browser by logging in.

Preventing Session Fixation

Another specific kind of session attack is known as *session fixation*. This approach is the opposite of *session hijacking*. Instead of malicious user Alice finding and using Bob's session ID, she creates her own session ID (perhaps by logging in legitimately), and then gets Bob to access the site using that session. The hope is that Bob would then do something that would unknowingly benefit Alice.

You can help protect against these types of attacks by changing the session ID after a user logs in. The **session_regenerate_id()** does just that, providing a new session ID to refer to the current session data. You can use this function on sites for which security is paramount (like e-commerce or online banking) or in situations when it'd be particularly bad if certain users (i.e., administrators) had their sessions manipulated.

Tip

For critical uses of sessions, require the use of cookies and transmit them over a secure connection, if at all possible. You can even set PHP to only use cookies by setting **`session.use_only_cookies`** to 1.

Tip

By default, a server stores every session file for every site within the same temporary directory, meaning any site could theoretically read any other site's session data. If you are using a server shared with other domains, changing the **`session.save_path`** from its default setting will be more secure. For example, it'd be better if you stored your site's session data in a dedicated directory particular to your site.

Tip

The session data itself can also be stored in a database rather than a text file. This is a more secure, but more programming-intensive, option. I show how to do this in my book *PHP 5 Advanced: Visual QuickPro Guide*.

Tip

The user's IP address (the network address from which the user is connecting) is not a good unique identifier, for two reasons. First, a user's IP address can, and normally does, change frequently (ISPs dynamically assign them for short periods of time). Second, many users accessing a site from the same network (like a home network or an office) could all have the same IP address.