Last updated: Fri, 26 Feb 2010

# Cookies

PHP transparently supports HTTP cookies. Cookies are a mechanism for storing data in the remote browser and thus tracking or identifying return users. You can set cookies using the setcookie() or setrawcookie() function. Cookies are part of the HTTP header, so setcookie() must be called before any output is sent to the browser. This is the same limitation that header() has. You can use the output buffering functions to delay the script output until you have decided whether or not to set any cookies or send any headers.

Any cookies sent to you from the client will automatically be included into a $_COOKIE auto-global array if variables_order contains "C". If you wish to assign multiple values to a single cookie, just add *[]* to the cookie name.

Depending on register_globals, regular PHP variables can be created from cookies. However it's not recommended to rely on them as this feature is often turned off for the sake of security. $HTTP_COOKIE_VARS is also set in earlier versions of PHP when the track_vars configuration variable is set. (This setting is always on since PHP 4.0.3.)

For more details, including notes on browser bugs, see the setcookie() and setrawcookie() function.

User Contributed Notes

## Cookies

**john at host89 dot net**
28-Oct-2009 01:24

```
I'm currently developing a secure system utilizing PHP session cookies, and rather than trying to
deal with recreating a session every time the script runs (which would hurt server performance), I'm
having the script send a 2nd cookie to the client which contains an MD5 of their username along with
a random value (so that it's harder to generate a matching key if somebody is trying to hijack the
session.) The original random value can be stored as $_SESSION['rndvalue'] or something of the like,
and easily re-hashed and compared to the cookie. If it isn't valid, just a simple session_destroy();
does the trick. For higher security, the random value could even be changed at every new page, and
to make leeway for that double-click phenomon, save the old one as 'rndvalueold' with the expiration
time as 'rndvalueexpire' or something. This will also let users use the back button even if the
session ID is passed via GET or POST.
```

**Henry**
08-May-2009 02:15

```
It is better to note not to attach your cookies to and IP and block the IP if it is different as
some people use Portable Browsers which will remember the cookies.  It is better to show a login
screen instead if the IP does not correspond to the session cookie's IP.
```

**ingen at stocken.ws**
19-Nov-2006 08:26

```
If you want a secured session not tied to the client IP you can use the valid-for-one-query method
below, but to safeguard against a scenario where the legitimate user clicks twice, you can use a
shutdown function (register_shutdown_function)*.

It will check to see if the script terminated prematurely (connection_aborted), and reset the valid
session ID. That way, it's still valid when the user makes the second request. If the script ends
properly, the new session ID will be used instead.

Now, since you can't set a cookie from the shutdown function (after output has been sent), the
cookie should contain both the previous valid session ID and the new one. Then the server script
will determine (on the next request) which one to use.

:: Pseudo example:
::
:: [Start of script:]
::
:: 1. Get the session ID(s) from cookie
:: 2. If one of the session ID's is still valid (that is, if there's a storage associated with it -
in DB, file or whatever)
::  ____2.1. Open the session
:: 3. Generate a new session ID
:: 4. Save the new session ID with the one just used in cookie
:: 5. Register shutdown function
::
:: [End of script (shutdown function):]
::
:: 1. If script ended prematurely
:: ____1.1. Save session data using the old Session ID
:: 2. Else
:: ____2.1. Save session data using the new Session ID
:: ____2.2. Make sure the old session ID is added to a list of ID's (used for the purpose described below)
:: ____2.3. Trash the old session storage
```

There's still the possibility of some deviant network sniffer catching the session cookie as it's sent to the client, and using it before the client gets the chance to. Thus, successfully hijacking the session.

If an old session ID is used, we must assume the session has been hijacked. Then the client could be asked to input his/her password before data is sent back. Now, since we have to assume that only the legitimate user has the password we won't send back any data until a password is sent from one request.

And finally, (as a sidenote) we could obscure the login details (if the client has support for javascript) by catching the form as it is sent, take the current timestamp and add it to the form in a dynamically generated hidden form object, replace the password field with a new password that is the MD5 (or similar) of the timestamp and the real password. On the serverside, the script will take the timestamp, look at the user's real password and make the proper MD5. If they match, good, if not, got him! (This will of course only work when we have a user with a session that's previously logged in, since we know what password (s)he's supposed to have.) If the user credentials are saved as md5(username+password), simply ask for both the username and password, md5 them and then md5 the timestamp and the user cred.

---

If you need a javascript for md5: http://pajhome.org.uk/crypt/md5/md5src.html

---

* You could use session_set_save_handler and make sure the session ID is generated in the open function. I haven't done that so I can't make any comments on it yet.

**kalla_durga at gmail dot com**
03-Feb-2006 08:10

In response to the solution posted in the comment below, there are some practical issues with this solution that must be kept in mind and handled by your code. I developed an application using a similar "use-it-once" key to manage sessions and it worked great but we got some complaints about legitimate users getting logged out without reasons. Turns out the problem was not tentative highjacking, it was  either:

A- Users double click on links or make 2 clicks very fast. The same key is sent for the 2 clicks because the new key from the first click didn't get to the browser on time for the second one but the session on the server did trash the key for the new one. Thus, the second click causes a termination of the session. (install the LiveHttpHeaders extension on firefox and look at the headers sent when you click twice very fast, you'll see the same cookie sent on both and the new cookie getting back from the server too late).

B- For any given reason, the server experiences a slow down and the response with the new key (which has replaced the old one on the server) is not returned to the browser fast enough. The user gets tired of waiting and clicks somewhere else. He gets logged out because this second click send the old key which won't match the one you have on your server.

Our solution was to set up a grace period where the old key was still valid (the current key and the previous key were both kept at all times, we used 15 seconds as a grace period where the old key could still be used). This has the drawback of increasing the window of time for a person to highjack the session but if you tie the validity of the old key to an IP address and/or user agent string, you still get pretty good session security with very very few undesired session termination.

**bmorency at jbmlogic dot com**
07-Oct-2005 12:14

In response to the solution posted in the comment below, there are some practical issues with this solution that must be kept in mind and handled by your code. I developed an application using a similar "use-it-once" key to manage sessions and it worked great but we got some complaints about legitimate users getting logged out without reasons. Turns out the problem was not tentative highjacking, it was  either:

A- Users double click on links or make 2 clicks very fast. The same key is sent for the 2 clicks

because the new key from the first click didn't get to the browser on time for the second one but the session on the server did trash the key for the new one. Thus, the second click causes a termination of the session. (install the LiveHttpHeaders extension on firefox and look at the headers sent when you click twice very fast, you'll see the same cookie sent on both and the new cookie getting back from the server too late).

B- For any given reason, the server experiences a slow down and the response with the new key (which has replaced the old one on the server) is not returned to the browser fast enough. The user gets tired of waiting and clicks somewhere else. He gets logged out because this second click send the old key which won't match the one you have on your server.

Our solution was to set up a grace period where the old key was still valid (the current key and the previous key were both kept at all times, we used 15 seconds as a grace period where the old key could still be used). This has the drawback of increasing the window of time for a person to highjack the session but if you tie the validity of the old key to an IP address and/or user agent string, you still get pretty good session security with very very few undesired session termination.

**mega-squall at caramail dot com**
23-Feb-2005 09:04

I found a solution for protecting session ID without tying them to client's IP. Each session ID gives access for only ONE querry. On the next querry, another session ID is generated and stored. If somebody hacks the cookie (or the session ID), the first one of the user and the pirate that will use the cookie will get the second disconnected, because the session ID has been used.

If the user gets disconnected, he will reconnect : as my policy is not to have more than one session ID for each user (sessions entries have a UNIQUE key on the collomn in which is stored user login), every entries for that user gets wiped, a new session ID is generated and stored on users dirve : the pirate gets disconnected. This lets the pirate usually just a few seconds to act. The slower visitors are browsing, the longer is the time pirates get for hacking. Also, if users forget to explicitly end their sessions .... some of my users set timeout longer than 20 minutes !

IMPORTANT NOTE : This disables the ability of using the back button if you send the session ID via POST or GET.

**James Olsen**
22-Jan-2005 04:14

Tying the session to the IP of the user is not a good idea. Some users, notably AOL users, are behind a rotating proxy which means their hits to the server will actually be coming from different IP addresses over the duration of their visit. Trying the session to the IP will not work properly for those users.

**myfirstname at braincell dot cx**
24-Sep-2003 03:47

[Editor's note: Wilson's comment has been deleted since it didn't contain much useful information, but this note is preserved although its reference is lost]

Just a general comment on Wilton's code snippet: It's generally considered very bad practice to store usernames and/or passwords in cookies, whether or not they're obsfucated.  Many spyware programs make a point of stealing cookie contents.

A much better solution would be to either use the PHP built in session handler or create something similar using your own cookie-based session ID.  This session ID could be tied to the source IP address or can be timed out as required but since the ID can be expired separately from the authentication criteria the authentication itself is not compromised.

Stuart Livings