Cookies and Sessions[1]

Prof. Dr. Stefan Zander

November 18, 2019

[1] Selected lecture of the module "Entwicklung Webbasierter Anwendungen"

Objectives:

- *Understand the logic behind cookies and sessions*
- *Create cookies and sessions in PHP*
- *Learn about possible limitations and security risks*

Contents

As we will see, sessions and cookies work perfectly togehter

---

1 Motivation

HTTP is a **stateless protocol** with no «memory»; this means that no data could be saved or retained by the protocol between two subsequent requests. In terms of the protocol that means, that a subsequently issued HTTP request has no knowledge about previously issued requests and their status as well as about the data generated during the first request. This behavior has some advantages in terms of simplicity and scalability and is one major driving factor for the vast growth of the WWW, but also yields serious implications and limitations in some scenarios—which we will discuss in the course of this lecture. This lecture therefore introduces two solutions, namely **cookies** and **sessions**, that deal with this limitation. We discuss both technologies by means of the PHP language.

All data available during the first call of a page are lost in a second call of the same page.

2 Cookies

The concept of cookies has been developed by Netscape in the mid 90's[2]. Since then, the original specification has been iteratively revised and new functionalities have been incorporated. It is available under the URL `www.ietf.org/rfc/rfc2965.txt`.

[2] The original specification document is still available under the URL `http://wp.netscape.com/newsref/std/cookie_spec.html`

2.1 What is a Cookie?

A cookie represents textual information that is exchanged between a server and its clients. The server sends a request for storing one or more cookies on the client as part of the HTTP header. An example HTTP header entry is illustrated below[3]

```
Set-Cookie: Language=PHP
Set-Cookie: Version=7.0
```

[3] Cookie data are stored in the HTTP general header part. Inspecting the HTTP response issued by the `cookies.php` file shows the complete HTTP header information.

```
HTTP/1.1 200 OK
Date: Mon, 18 Dec 2017 06:34:20 GMT
Server: Apache/2.4.27 (Unix) OpenSSL/1.0.2l PHP/7.1.9 mod_perl/2.0.8-dev Perl/v5.16.3
X-Powered-By: PHP/7.1.9
Set-Cookie: Language=PHP
Set-Cookie: Version=7.0
Content-Length: 478
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
```

Upon receiving that information, the Web browser then processes them in accordance to its configuration:

1. The cookie is either **saved**,
2. or it is **rejected**,
3. or the Web browser **asks** the user whether to accept or reject it.

If the cookie was accepted, the cookie information is either held in memory or stored on the **client's hard drive** depending on the Web browser's configuration. During the initial response, the Web server has no knowledge about how the client processes its cookie request, i.e., whether it was accepted or rejected. How a cookie is processed on the client side is communicated back to the server only with the next client's request: When the cookie was accepted, the client includes it in the HTTP header of every request:

Usually, cookie data are stored in a database file.

Sever has no information about how cookies are processed on the client side

The server receives information about whether a cookie was accepted or not only with the following request.

```
Cookie: Language=PHP; Version=7.0
```

The Web server is then able to recognize a client and access this information to include it in further processing steps. If a request for cookies was not accepted, the respective header field is not set. The following example demonstrates a HTTP request send by the client that includes cookie information:

Listing 1: A client's HTTP request with cookie data included

```
1  GET /dev/dev/php/session_cookies/cookies.php?do=print HTTP/1.1
2  Host: localhost
3  Connection: keep-alive
4  Upgrade-Insecure-Requests: 1
5  User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_1)
       AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84
       Safari/537.36
6  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
       image/webp,image/apng,*/*;q=0.8
7  Referer: http://localhost/dev/dev/php/session_cookies/cookies.php?
       do=set
8  Accept-Encoding: gzip, deflate, br
9  Accept-Language: en-GB,en;q=0.9,en-US;q=0.8,de;q=0.7
10 Cookie: Language=PHP; Version=7.0
```

2.2 *Storage Location*

Some Web browsers such as Google's Chrome browser stores cookie data in a SQLite database[4]. Under Windows, the database file is stored at the following location:

[4] For more information see `https://stackoverflow.com/questions/31021764/where-does-chrome-store-cookies`

```
C:\Users\your_username\AppData\Local\Google\Chrome\User Data\Default\
```

In Mac OS the location is

```
~/Library/Application Support/Google/Chrome/Default/Cookies
```

and in Linux

```
~/.config/google-chrome/Default/
```

Please note that special tools are needed to read SQLite database files. Otherwise, cookie data can be accessed from within the browser.

Tools are needed to read SQLite database files

2.3 *Restrictions*

The following section discusses precautions that have been incorporated in the cookie specification in order to minimize misconduct.

Restrictions to minimize misconduct

- *Reading cookie data*
  The probably most relevant restriction is that **a cookie can only be read by the server that has set the cookie**. A cookie set by the Web page `www.h-da.de/` can not be read by the page `www.tu-darmstadt.de/` and vice versa. This restrictions also applies to sub-domains such as `www.h-da.de/fbi`, the cookies of which could not be retrieved by `www.h-da.de/eti` and vice versa. It is, however, possible to bind cookies to second-level domains (SLD/2LD). If a cookie is bound to the URL `.php.net`, it can be read from `www.php.net` as well as from `pecl.php.net` and `windows.php.net`. It is also possible to bind a cookie to a path within a domain such as `/regierung`. The cookies set by this site can, for instance, not be accessed from the site `/opposition`.

  Access restrictions

- *Cookie size*
  A cookie must not exceed the **maximum size of 4 kilobyte**, i.e., 4096 characters. The size restriction includes both *name* and *value*. If name and value are grater than 4 kilobyte, all characters that exceed this limit will be cut off.

  File size

- *Amount of Cookies accepted by Web browser*
  A Web server is instructed to accept a maximum of **300 cookies at all** and a maximum of **20 cookies per domain**. In case one of these numbers is exceeded, the oldest cookie (i.e., 301 or 21) will be deleted.

  Number of cookies

3 Working with Cookies

As indicated by the previous sections, setting and reading cookies is done in the HTTP header exclusively. Cookies are directly integrated in PHP; additional installations are not necessary. In order to use cookies, Web browser support must be enabled. PHP provides a number of convenience methods for setting and processing cookies.

Cookies are stored in the HTTP header

Cookies must be enabled by the Web browser

PHP offers convenience methods for working with cookies

3.1 Setting a Cookie

Cookies can be set in PHP using the function

```
setcookie(...)
```

Its syntax is as follows:

```
boolean setcookie ( string name [,
                    string value [,
                    int expire [,
                    string path [,
                    string domain [,
                    boolean secure [,
                    boolean httponly ]]]]]] )
```

Please note that parameters in square brackets `[ ]` are optional; only `name` and `value` are mandatory.

- `int expire`
  Sets the cookie's **expiration date** in seconds starting from 1.1.1970. Generally, two types of cookies exist that can be distinguished according to its expiration date:

  1. *Temporary Cookies* or sometimes called *Session Cookies* are only valid as long as the browser remains open. They will be deleted when the browser is closed.
  2. *Permanent Cookies* or persistent cookies contains a defined date that determines their validity.

  The expiration date can be set with the PHP functions `time() + 60 * 60 * 24` for a cookie the validity of which is one day, or using `mktime();` a cookie with expiration date `mktime(12, 0, 0, 12, 24, 2018)` is valid until Christmas Day 2018, 12 am. A cookie without expiration date is treated as a temporary cookie.

  Temporary cookies are deleted when the browser is closed

  Cookies without expiration data are treated as temporary (=session) cookies

- `string path`
  This parameter bounds the cookie to a specific path from which the cookie can be read and manipulated. For instance, a cookie set to the path `www.h-da.de/fbi/students` can not be read from `www.h-da.de/fbi/professors`. In general it is good practice to set cookies only on those sites that need them (e.g., on administrative sites) in order to increase performance.

  It is good practice to set cookies only on those sites that need them

- `string domain`
  Sets the domain from which a cookie can be accessed and manipulated. In cases where this parameter is not explicitly set, the Web browser sets the Web site's domain from its URL. As a consequence, if a Web site is requested using the site's IP address, only the address will be used and not the domain name.

- `boolean secure`
  If this token is set, a cookie will only be sent through secure HTTP connections via HTTPS. However, for security reasons, sensible data should never be stored in cookies but held server-side.

  Never store sensible data in a cookie

- `boolean httponly`
  One possible security vulnerability of *cross-site scripting* is that external javascript code that is included in a site is in principle capable of reading cookie data. Depending on the browser's configuration, this might still be the case although this parameter is set to `true`.

  External JavaScript code can read cookie data

Since cookies are set as part of the HTTP header, working with cookies in PHP requires to call the function `setcookie()` prior to the first HTML output[5]. Listing 5 demonstrates the correct setting of a cookie in PHP:

`setcookie()` must be called prior to the first HTML output

[5] This restriction can be circumvented when writing to an output buffer (`output_buffering = true` in `php.ini`).

Listing 2: Setting cookies in PHP

```php
<?php
  header("Content-type: text/html; charset=UTF-8");
  setcookie("Language", "PHP");
  setcookie("Version", "7.0");
?>
<html>
<head>
  <title>Setting Cookies correctly</title>
</head>
<body>
  [...]
</body>
</html>
```

3.2 *Reading Cookie Data*

PHP automatically captures all cookies sent from the client to the server and stores them in the global array `$_COOKIE` from where they can be accessed. In order to access the value of a single cookie, the cookie's name must be used as key:

Listing 3: Reading a cookie value in PHP (not recommended)

```php
<?php
  echo htmlspecialchars($_COOKIE["Language"]);
?>
```

Caution: If no cookie with the given name exists, an error message will be printed depending on the configuration of the PHP Interpreter. Therefore, it is better to first check whether a cookie with the given name exists; this can either be done with the PHP function `array_key_exists()` or using `empty()` and `isset()`. The following code excerpt demonstrates this:

The PHP interpreter generates an error message if no cookie with the given name exists.

Listing 4: Favorable way of reading a cookie value with `isset()`

```php
<?php
  if (isset($_COOKIE["Language"])) {
    echo htmlspecialchars($_COOKIE["Language"]);
  }
?>
```

However, it is only possible to read a cookie but not its properties (path, domain, expiration date etc.) since those data are only stored on the client-side and are not transferred to the server.

A cookie's properties (path, domain, expiration date) are only stored client-side and can not be read server-side

3.3 *Deleting Cookies*

Cookies can be deleted in two different ways:

1. One way is to set the expiration date of a previously set cookie to a date that lies in the past; the Web browser thus deletes the cookie from the memory automatically. This can be achieved, for instance, by setting the expiration date to `0`, which works for both session and permanent cookies.

Set the expiration date to a past date (*Caution: This method only works if all other parameters are set exactly the same when the cookie was created.*)

Listing 5: Deleting a cookie by setting its expiration date to `0`

```
<?php
  setcookie("Language", "PHP", 0);
?>
```

   Be aware that for deleting cookies using this methodology, all previously set parameters need to be exactly the same when the cookie was set. If the value of only one remaining parameter (other than expire) differs, deleting a cookie will not work.

2. Another possibility is to set a cookie's value to an empty string (`""`). The Web browser recognizes this and automatically deletes the cookie—regardless of its expiration date.

Set the cookie's value to an empty (`""`) string.

3.4 *Final Thoughts*

Although cookies in general are very useful and help in providing a good user experience, the following principles, however, need to be considered when working with cookies:

1. Make sure that your Web site works even *without* cookies
2. Do not torment those users who have cookies enabled in their browsers

Particularly regarding the second point, it is not a good practice to send a lot of single cookies, e.g., one with each image etc. Another point is that cookies should be used with caution and with thriftiness, i.e., information should be bundled in one cookie rather than sending a large number of single cookies that would soon exceed the 300 number limit. Another point concerns the expiration date which should also be set rationally. It is rather stupid to set an expiration date to the far future, e.g., year 2041; in most cases, one year is more than sufficient.

Don't set a lot of single cookies

Bundle information in one cookie

Set a realistic expiration date

One big problem that is still present when using cookies is that of **spying on users**. The following scenario demonstrates this problem:

Cookies can be misused for spying on user data

Image you request a page from domain `www.website1.net` that includes an advertisement banner from the domain `www.adserver.`

net. However, www.adserver.net could set a cookie that might be sent back to the domain. If the user navigates to another Web page www.website2.net, which also includes banners from www.adserver.net, this site retrieves the previously created cookie set via the page www.website1.net. In doing so, www.adserver.net gains knowledge about the pages requested by a user—about the user's interests and her browsing behavior. This enables an advertisement company to create a user profile and provide more target-specific advertisements.

4 Sessions

As we will see, sessions and cookies work perfectly togehter

In the last section, we learned that *HTTP is a stateless protocol* and that cookies are one way to deal with that issue. However, the main problem associated with cookies from an end user perspective is that they can be used to harm users' privacy since they allow 3rd parties to create user profiles—at least to a certain degree. Another problem is that they can be disabled by the user's Web browser.

Problem #1: Cookies might be misused to harm users' privacy by creating user profiles.

Problem #2: Cookies can be disabled

Sessions are another possibility to deal with the stateless character of HTTP. The term «session» usually denotes the act of visiting a Web page by a user. When the user accesses a Web shop for 10 minutes before she moves to another page, she had a 10 minute session. When she returns to the shop, e.g., after 1 hour, a new session will usually be created.

Session denotes the act of visiting a website by a user.

PHP provides the possibility to create and manipulate session data. It stores data within a session object and provides some convenience methods for accessing and manipulating them. In general, those data are available as long as the session objects exists, i.e., as long as the session is active.

PHP provides convenience methods for working with sessions.

4.1 Background Information

When PHP stores data in a session object, they need to be serialized beforehand and are then stored on the Web servers file system, depending on its configuration. Each session will be identified through a 32-character long hexadecimal value, the so-called **Session ID**. The Session ID serves as identifier for the data stored in the session object. *Hence, the problem of storing client-specific data between different subsequent requests is reduced to transmitting the Session ID between server and client.* The data are stored on the server-side and can be accessed through a set of PHP convenience methods.

Session data are stored server-side

The Session ID serves as identifier of the session and the data stored in the session objects

Sensible data should never be stored in cookies!

Sessions require that the Session ID is transmitted in every request and response. This can be done using two different approaches:

The Session ID need to be transmitted in every request and response. Why?

1. The Session ID is stored in a cookie
2. The Session ID is appended to all URLs contained in a Web page

Using cookies for storing the Session ID seems counter intuitive

Cookies and sessions are the safest method of storing user data

since cookies can be disabled in the user's Web browser. However, cookies are the one and only useful method for storing session data since a session management without cookies yields some serious drawbacks, e.g., *session hijacking*.

Session management without cookies allows session hijacking

4.2 Session Management without Cookies

The central concept of using a session management without cookies is to append the Session ID to all URLs embedded in a Web page. This leads to generated URLs according to the following pattern:

```
http://myserver.org/myscript.php?PHPSESSID=eh2czw36afpn71r8kbt03q4p8005
```

As a consequence, every link embedded in Web page needs to be automatically extended by the PHP Interpreter using the Session ID so that it is not lost. However, the Session ID is available as a `GET` parameter and does not influence the processing of the script.

The PHP Interpreter needs to add the Session ID to every URL embedded in a Web page.

The parsing of a Web page and complementing all URLs with the Session ID is resource consuming and might result in *performance bottlenecks*, in particular for large pages with many links. However, PHP is capable of doing the complementation automatically when the corresponding flags are set in the `php.ini` file. The `php.ini` also contains the pattern declarations used by the PHP Interpreter for complementing the URLs contained in a Web page (such as `a`, `href`, `frame`, `src`, `form`[6] etc).

Adding the Session ID to a Web site's URLs is rather resource demanding.

URL rewriting is done automatically by PHP when flags are set in the `php.ini`.

[6] In forms, a hidden field is used to transmit the Session ID.

4.3 Using Sessions in PHP

In PHP, a session is started using the function

```
session_start();
```

This function needs to be called *explicitly* on every page that requires a session management and before any output is generated. However, PHP can be configured to automatically initiate the session management on *every* page[7] using the following option flag in the `php.ini`:

`session_start()` must be called prior to any output

[7] In terms of performance, it is not useful to include session mangement in every page.

```
session.auto_start = 1
```

The value of the `session.auto_start` is `0` per default[8].

[8] If `session.auto_start` is enabled, `session_start();` does not need to be called explicitly.

PHP also uses a *garbage collection* mechanism to avoid situations in which the directory for storing session data might grow too big.

The maximal life time of session data can be specified with the

```
session.gc_maxlifetime = 1440
```

value that determines the time in seconds before session data will be removed by the garbage collector. The above lifetime value causes the garbage collector to remove the session data when no link was clicked in a page within 24 minutes.

4.4 Writing Session Data

Working with sessions mainly involves reading and writing data from/to the session object

When working with sessions in PHP, a large share of programming work is related to writing data to and reading data from the super global session array `$_SESSION`.

Two steps are necessary for storing data in a session:

1. Initiate the session management by calling `session_start()` (unless `session.auto_start = 1` is set in the `php.ini`.)
2. Write data to the `$_SESSION` object

The following code excerpts stores two strings in the session:

Listing 6: Storing data in the session

```php
<?php
  session_start();
  // Further instructions...
  $_SESSION["language"] = "PHP";
  $_SESSION["version"] = "7.0";
?>
```

Call `session_start()` prior to any output.

Once initated, data can be stored any time in the session object.

Be aware that when cookies are used for session management, `session_start()` need to be called prior to writing HTML (e.g., set it in the head). However, when the session is initiated, data can be stored in the session at any time in the PHP script.

Whether a client accepts cookies can only by ascertained with the second request.

One problem still exists since PHP can ascertain whether cookies are enabled in the user's Web browser only with the *subsequently following request*. Although it is strongly advised against doing so, PHP can be configured to automatically add the Session ID to all links in the generated page when it is first called. Due to privacy and security issues, this should be avoided at all costs.

4.5 Reading Session Data

Data can only be read from the session after `session_start()` was called.

Data can be read from the session by accessing the `$_SESSION` array with the data key. Be aware that data can only be read from the session when `session_start()` was called before, which causes the `$_SESSION` array to be filled with the respective values.

Listing 7 shows how data can be read from the `$_SESSION` array.

Use the `isset()` function when reading data from the session

Escape data stored in the session to avoid code injection

Please note that when reading data from the session, it should first be checked whether the data exists using the `isset()` function. Additionally, the data value need to be *escaped* in order to avoid *code injection* and to not compromise the HTML output.

4.6 Deleting Session Data

PHP provides 2 ways to delete data from a session:

Listing 7: Reading session data

```php
<?php
  session_start();
  // Further instructions...
  if (isset($_SESSION["language"])) {
    echo htmlspecialchars($_SESSION["language"]);
  }
  if (isset($_SESSION["version"])) {
    echo htmlspecialchars($_SESSION["version"]);
  }
?>
```

1. Setting the session variable to an empty string or to `null`
2. Deleting the session variable using the `unset()` function

The result of both approaches is identical though the second variant using `unset()` is faster. The excerpt in Listing 8 uses variant 2 and 1 for the deletion of session variables:

Listing 8: Deleting session data

```php
<?php
  session_start();
  // Variant #2: Using unset
  if (isset($_SESSION["language"])) {
    unset($_SESSION["language"]);
  }
  // Variant #1: setting value to an empty string
  if (isset($_SESSION["version"])) {
    $_SESSION["version"] = "";
  }
?>
```

4.7 *Destroying the Session*

Completely destroying the session object

In order to completely destroy a session and its data, two steps are necessary:

1. Delete all data from the session array using the `session_unset()` function or reinitializing the session variable using `$_SESSION = array();`
2. Destroy the session using `session_destroy()`

Delete the session cookie

Sometimes, it might be necessary to also delete the session cookie, if it has been set before. This can be done with the `setcookie(...)` command in Listing 9.

The `session_name()` function returns the session's default name as set in the php.ini, e.g.,

Listing 9: Deleting the session cookie

```php
<?php
  setcookie(session_name(), "killme", 0 , "/");
?>
```

```
session.name = PHPSESSID
```

The value can be dynamically retrieved and used to set a temporary cookie with expiration date `0` seconds. The following PHP program illustrates how to completely delete all session data together with the session cookie:

Listing 10: Deleting all session data together with the session cookie

```php
<?php
  session_start();
?>
<html>
<head>
  <title>Sessions</title>
</head>
<body>
<?php
  session_unset();
  session_destroy();
  setcookie(session_name(), "killme", 0, "/");
?>
<p>Everything has been deleted!</p>
</body>
</html>
```

5 *Using Sessions for Access Control*

Sessions are omnipresent and widely used in PHP-based Web applications. A typical use case scenario is to protected Web content from unauthorized access, i.e., sensitive information can only be retrieved after authentication and authorization via a login page. Listing 11, 12, and 13 demonstrate how this can be realized using PHP and sessions.

```php
// Login.inc.php
<?php
  session_start();

  if (!isset($_SESSION["login"]) || $_SESSION["login"] != "ok") {
    $url = $_SERVER["SCRIPT_NAME"];
    if (isset($_SERVER["QUERY_STRING"])) {
      $url .= "?" . $_SERVER["QUERY_STRING"];
    }
    header("Location: login.php?url=" . urlencode($url));
  }
?>
```

Listing 11: Checking whether a user is authenticated and authorized to request a protected resource («login.inc.php»)

```php
// protected_page.php
<?php
  require_once "login.inc.php";
?>
<html>
<head>
  <title>Protected Resource</title>
</head>
<body>
  <h1>Here is all the protected information</h1>
  <p> Protected content... </p>
</body>
</html>
```

Listing 12: The resource the content of which is protected («protected_page.php»)

```php
// login.php
<?php
  session_start();

  if (isset($_POST["user"]) && isset($_POST["password"])) {
    // usually checked against a user database
    if ($_POST["user"] == "jamesbond" &&
        $_POST["password"] == "topsecret") {
      $_SESSION["login"] = "ok";
      // redirect the user to the page from where she is comming
      $url = (isset($_GET["url"])) ? nl2br($_GET["url"]) : "index.
    php";
      header("Location: $url");
    }
  }
?>
<html>
<head>
  <title>Login</title>
</head>
<body>
  <form method="post">
    User: <input type="text" name="user" size="10" /><br />
    Password: <input type="password" name="password" size="10" /><br
     />
    <input type="submit" value="Login" />
  </form>
</body>
</html>
```

Listing 13: The login page («login.php») with stored user authorization credentials; usually such information will be retrieved from and checked against a user database.

6 Security Concerns

Sessions are easy to use and maintain in PHP but also yield security concerns, particularly when realized as *cookie-less sessions*. The reason for that is that in cookie-less sessions, the key to user data—the Session ID—is available as plain text since it is encoded in the Web site's URLs. The following scenario illustrates a potential security flaw that comes with cookie-less sessions.

In cookie-less sessions, the Session ID is available as plain text

Imagine the following situation:

A Web mail provider uses cookie-less sessions. You send one of the Webmail provider's clients an email that contains a link to your Web page. The link in the email points to a PHP script that will be executed when the recipient clicks on the link in the Webmail Web interface. The first thing the script does is to look at the `HTTP_REFERER` *header field; most Web browser use this header to send the previously visited page by a user. If you are lucky, the HTTP request issued by clicking on the link in the mail client contains the URL of the previously visited page, i.e., the mail provider's landing Web page together with the Session ID encoded in the URL, since the mail provider uses cookie-less sessions. The Session ID can then be extracted and used to steal user data or mimic a user's identity.*

This method is called *Session Hijacking*

In the early days of Web mail, some mail providers such as Lycos and GMX used cookie-less sessions and were vulnerable to session hijacking attacks. Meanwhile, almost all providers use temporary cookies to circumvent such attacks.

Although a number of counter measures exists that try to minimize such attacks (e.g., storing the user's current IP address in a separate session variable) no method provides full security. In case of storing the user's IP address, the server will evaluate the validity of the client's IP adress with every request and if the IP does not match the IP address stored in the session variable, the session will be immediately destroyed.

Counter measure #1: Evaluate the client's IP address in every request

Another possibility is to evaluate the `HTTP_REFERER` environment variable and determine the page from where the request was initiated. In case of a session hijacking attack, the value of this variable might be incorrect. Unfortunately, this method is also not entirely reliable since some Web browser either do not send this value or it might be compromised.

Counter measure #2: Evaluate the origin of the page that issed the request

In general, only use sessions in combination with cookies, i.e., solely store the Session ID in a cookie!