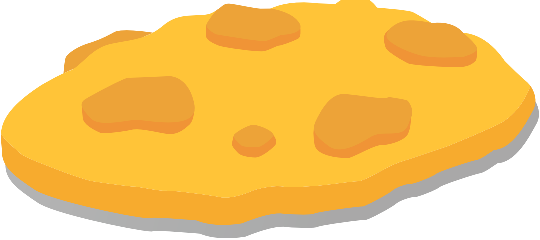
Security of Cookies

Cookies are the way HTTP ensures that users who send a request to a website are allowed access to visit restricted areas. This paper will examine the topic of cookie security by evaluating security measures that are used to protect the content of sensitive cookies and users against attacks.

Ziyahan Albeniz
February 2019

Table of **Contents**

3 HTTP and Ticket System
3 How Do Cookies Work?
3 Cookie Attributes
3 name:
3 domain:
4 path:
4 secure:
4 expires:
4 max-age:
4 Modifying Cookies with Javascript
5 Protecting Session Cookies With httpOnly
6 Introduction to Session Cookies
7 Analyzing Sessions
7 session_start()
9 Other Options to Hide Sessions
10 Cookie Attributes in Terms of Security
10 name[=value] Attribute
10 Cookie Headers
11 Solution to a Secure Session
11 domain Attribute
12 Example of Setting the domain Attribute
13 path Attribute
14 Example of Setting path Attribute
15 Solution
15 expires and max-age
15 httpOnly Flag
15 secure Flag
16 SameSite Attribute
17 Strict and Lax Values
17 Strict
17 Lax
18 Examples of the Importance of Prefixes
19 __Secure- Prefix
19 __Host- Prefix
19 Extra Measures
20 Conclusion



This paper will take a close look at cookie security and shed light on various security measures that should be implemented in order to protect the content of sensitive cookies, and to protect users against a range of different attacks.

HTTP and Ticket System

In many regards, the most common internet protocol, HTTP, works like the ticket system in a subway. For HTTP servers, just like train personnel, it's impossible to identify every user when they interact with thousands of visitors on a daily basis. For ticket inspectors, service tickets (just like cookies) are the only way to differentiate users and find out whether they are authorized to ride with the subway. HTTP similarly needs definitive proof that the user sending the request is allowed to do so, at least when it comes to visiting restricted areas, such as an admin panel. HTTP does so by using cookies and session IDs.

How Do Cookies Work?

When it was first introduced, the HTTP protocol was only used to provide static content, such as HTML files. But new methods to interact with visitors were developed relatively quickly. One of the key features that was required to ensure proper communication between server and client was to have a singular value (an ID) to distinguish the requests of user A from those of user B. In 1994, a Netscape employee, Lou Montulli, realized the necessity of this ID when he was designing a shopping application. He decided to adapt a magic cookie concept that was widely used in Unix.

Soon after, cookies began to be supported with the release of the first version of Netscape Navigator. Later, all browsers supported cookies. Besides a few security additions, cookies preserved their initial structure. Despite rumors stating otherwise, cookies are part of a simple database that does not contain any malicious code, with each cookie able to hold up to 4KB of data.

Cookie Attributes

Let's take a look at the cookie attributes. Before we begin, you should note that the attributes must be separated with a semicolon and a space.

name:

Although cookies are generally set in the format `name=value`, browsers do not strictly expect it to be this way. Therefore, cookies can be set with only the name. Some web applications set cookies without a value in lieu of a boolean data type. If the cookie name exists, its value is seen as being true. If it doesn't, the value is seen as being false. Below is such a cookie, that indicates whether the user wants to open the page in full-screen mode.

```
Set-Cookie: fullScreen; path=/;
```

A website may set this kind of cookie to remember some settings you've made in the future.

domain:

This is the attribute that specifies the `domain` to which the cookie will be sent when you visit a website. Think of it in terms of train tickets again. You can't use a train ticket from Madrid to Barcelona to board an airplane. In the same way, browsers won't allow you to use a cookie for google.com on amazon.com. This is an optional attribute. If it's not specified, the domain name of the site that set the cookie will be used implicitly.

A cookie may be also be used in multiple subdomains belonging to the same domain. For instance, a cookie set for example.com, may be sent with the requests to mail.example.com, calendar.example.com or crm.example.com.

```
Set-Cookie: Scanner=Netsparker; domain=example.com
```

path:

The `path` attribute is another optional attribute for cookies. It will specify the exact path that needs to be present in the URL for the cookie to be sent.

If the path is '/', the browser will send the cookie along with all the requests to example.com, regardless of the path. If the path is `/foo`, all the requests to example.com/foo and example.com/foo/baz will contain this cookie.

If no path attribute is given, the default path value, which is the page on which the cookie was set, will be used.

```
Set-Cookie: Scanner=Netsparker; path=/foo
```

secure:

When you mark the cookie as `secure`, you make sure that, in addition to the domain and path matching above, the connection type has to be HTTPS for the cookie to be sent. If this optional attribute isn't given, the cookie will be sent to all requests that match the domain and path, regardless of its state of security. You can only set this attribute in an HTTPS request. You should keep in mind that this variant of a same origin policy is fundamentally different from the same origin policy that JavaScript adheres to, in that it doesn't take the protocol into account by default.

expires:

The optional `expires` attribute specifies the amount of time a cookie will be stored in the browser. If it's not specified, the cookie will be stored as long as the browser is open. It's deleted when the browser is closed. The cookies that have the `expires` attribute set to a date in the distant future, are known as Persistent Cookies. The expected format for the `expires` attribute is: Wdy, DD-MM-YYYY HH:MM:SS GMT:

```
Set-Cookie: Scanner=Netsparker; domain=example.com; path=/; expires=Sun, 21-02-2018 08:25:01 GMT
```

If you want to delete a cookie from the client browser, you set a past date as the `expires` value, since cookies are deleted automatically after they expire:

```
Set-Cookie: Scanner=Netsparker; domain=example.com; path=/; expires=Sun, 21-02-1977 08:25:01 GMT
```

max-age:

This one is nearly identical to the `expires` attribute, but it uses seconds instead of an actual date.

```
Set-Cookie: Scanner=Netsparker; domain=example.com; path=/; max-age=86400
```

If the value is 0 or lower, the cookie will be removed from the user's browser.

Modifying Cookies with JavaScript

Using the `document.cookie` object, you can generate cookies and change their content using JavaScript.

You can read the cookies the client browser sent to your website by reading the value of the `document.cookie` attribute. They are stored in the following format:

```
Cookie1=Cookie1Val; Cookie2=Cookie2Val;
```

You can use the following code to set cookies using JavaScript:

```
document.cookie = "name=Netsparker; domain=example.com; path=/";
```

Protecting Session Cookies With httpOnly

Cookies can be read, modified and deleted by JavaScript. Although there are countless benefits to storing various user data for JavaScript to read, some cookies should be off limits for JavaScript. One example is a session cookie, which is only of use to the server, but should not be accessible by JavaScript, as it allows easy session hijacking if an XSS vulnerability is present.

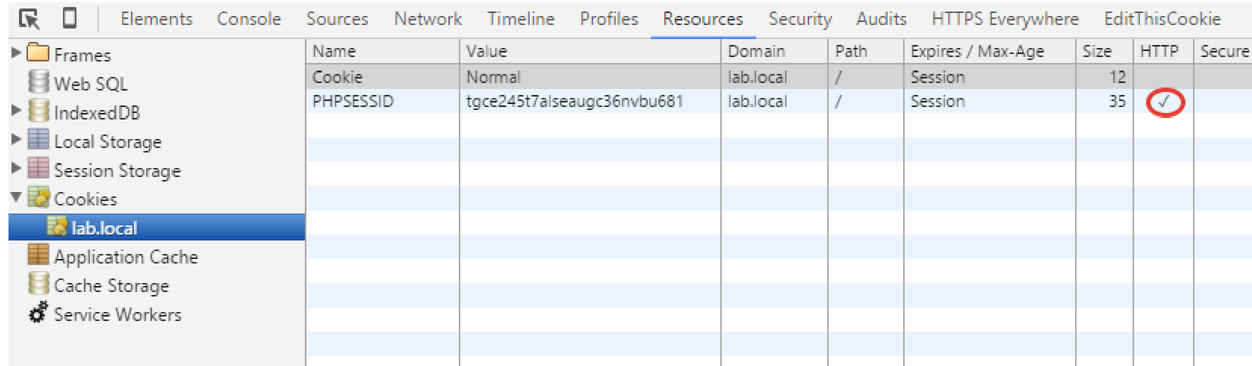
In 2002, Microsoft developed the httpOnly flag, which was later added to the HTTP Cookie specifications, and has since been supported by all modern browsers to prevent this specific security risk.

If the cookie has an `httpOnly` flag set, the browser will only send it together with HTTP requests, but will not make it available to JavaScript, hence the name httpOnly.

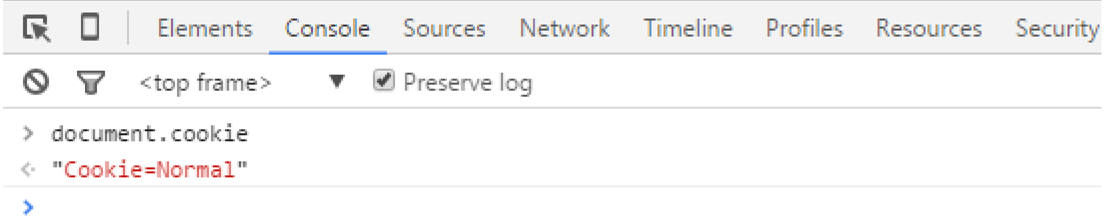
You can label a cookie with `httpOnly` by simply adding the `httpOnly` attribute in the Set-Cookie header:

```
Set-Cookie: PHPSESSID=tgce245t7alseaugc36nvbu681; domain=lab.local; path=/; htt-
pOnly
```

This screenshot shows the Google Chrome Developer console.



It shows two cookies: one is called `Cookie` and the value 'Normal', the other is called `PHPSESSID` with a session ID as its value. Additionally, the `PHPSESSID` cookie is marked as 'httpOnly', indicated with a checkmark highlighted with a red circle.



If you read the value of `document.cookie`, you will see that only the cookie that wasn't marked as 'httpOnly' (`Cookie`) is available for JavaScript, unlike the `PHPSESSID` cookie.

Introduction to Session Cookies

Let's take a look at how cookies and session handling have developed since 1994.

All visitors to your websites are of equal importance. However, some visitors may have advanced privileges, and are therefore able to access certain content that others can not. These visitors may be page editors or those who purchase services from you.

As the administrator of the webpage, you need to integrate a login form for users to introduce themselves and therefore receive the special treatment they require. Since HTTP is a stateless protocol, it can't really tell users apart or remember whether they are already logged in. Instead of asking users to repeatedly fill out the login form to access each page they visit, you can use this cookie to automatically authorize them. (Warning: Do not use code like this in production. It's insecure!)

```
Set-Cookie: isLogged=Yes;
```

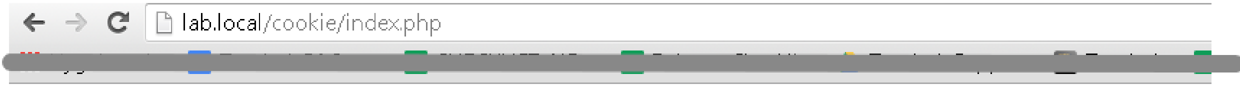
This cookie allows the website to save the user information, and identify them when required. We can set multiple cookies within the browser limit in one HTTP response by using the Set-Cookie header. After a successful login, the response will look like this:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Set-Cookie: isLogged=Yes;
Set-Cookie: username=Customer;
```

It'll look like this from the server side:

```
<?php
if($_COOKIE["isLogged"]=="Yes") {
 echo "Welcome ".$_COOKIE["name"];
} else {
 echo "Please log in!";
}
?>
```

However, nothing prevents a user with malicious intentions from crafting a cookie that would log them into the website, even though they don't have an account. This can easily be done with JavaScript and document.cookie, or by using curl or a similar tool.



Please log in!

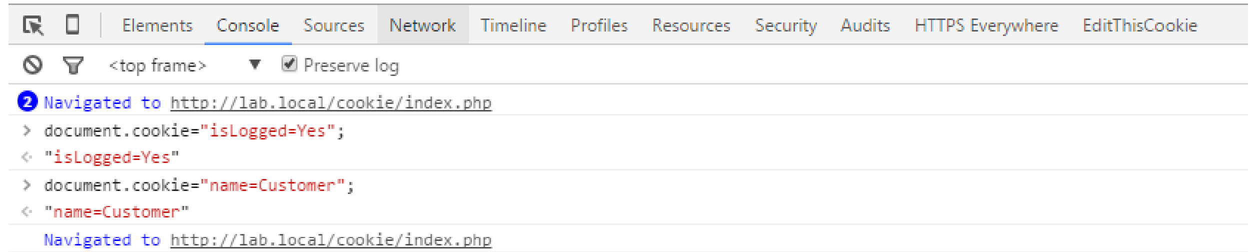
Elements Console Sources Network Timeline Profiles Resources Security Audits HTTPS Everywhere EditThisCookie

<top frame> ▼ Preserve log

```
Navigated to http://lab.local/cookie/index.php
> document.cookie="isLogged=Yes";
< "isLogged=Yes"
> document.cookie="name=Customer";
```

After the cookie is manipulated, the user can log in:

Welcome Customer



Analyzing Sessions

Instead of exposing the variables and values in a cookie, which may affect our web applications' workflow, decision mechanisms, and authorization processes, the session mechanism was developed to protect those elements from potential attacks.

With sessions, the critical data is kept away from the client-side by generating IDs that refer to the sensitive data on the server-side. These IDs are then stored on the client-side in the form of cookies. What makes them secure is that they are typically very long and unpredictable, which makes it practically impossible for an attacker to guess another user's session ID.

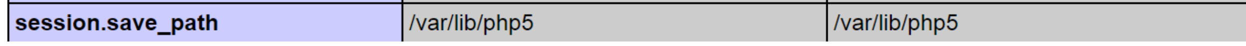
Every time the user makes a request to the server, the browser sends the user's session ID to the server, which allows it to identify the user and access the stored data for the respective session ID.

session_start()

In PHP applications, when the session_start() function is called, the first thing PHP will do is to check whether a cookie was sent or not (by default, its name is PHPSESSID).

session.name	PHPSESSID	PHPSESSID

You can view the path to the directory that contains the PHP session files using the phpinfo() function.



session.save_path	/var/lib/php5	/var/lib/php5

If the PHPSESSID cookie is not sent with the request, PHP will request the browser to set a cookie with the name PHPSESSID.

```
▼Response Headers     view source
   Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
   Connection: Keep-Alive
   Content-Length: 0
   Content-Type: text/html
   Date: Fri, 19 Feb 2016 11:26:29 GMT
   Expires: Thu, 19 Nov 1981 08:52:00 GMT
   Keep-Alive: timeout=5, max=100
   Pragma: no-cache
   Server: Apache/2.4.7 (Ubuntu)
   Set-Cookie: PHPSESSID=sp39odgr48v3e5d3dsl6dvcn46; path=/
   X-Powered-By: PHP/5.5.9-1ubuntu4.14
```

This cookie contains a long and secure session ID.

```
▼Request Headers     view source
   Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
   Accept-Encoding: gzip, deflate, sdch
   Accept-Language: tr-TR,tr;q=0.8,en-US;q=0.6,en;q=0.4
   Cache-Control: max-age=0
   Cookie: PHPSESSID=sp39odgr48v3e5d3dsl6dvcn46
   Host: lab.local
   Proxy-Connection: keep-alive
   Upgrade-Insecure-Requests: 1
   User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/48.0.2564.109 Safari/537.36
```

Once the session_start function checks whether the PHPSESSID value is valid, it'll proceed to step two: check whether there's already a session file that contains the ID of the PHPSESSID cookie.

```
root@virtualweb-VirtualBox:/var/lib/php5# ls -l
total 4
drwxr-xr-x 5 root     root     4096 Eki 12 16:20 modules
-rw------- 1 www-data www-data    0 Şub 19 13:26 sess_sp39odgr48v3e5d3dsl6dvcn46
root@virtualweb-VirtualBox:/var/lib/php5#
```

If the file doesn't exist, it will create it in the following format: "sess_**PHPSESSIDVALUE**". But if the file exists, it'll open the file, deserialize the serialized object it contains and store it in a global variable called $_SESSION.

Here's an example of how the data is stored in session files:

```
<?php
session_start();
if(authenticate($user, $passwd)) {
 $_SESSION["loggedIn"] = "yes";
 $_SESSION["last_login"] = date("Y-m-d H:i:s");
} else {
 echo "Try again!";
}
```

```
root@virtualweb-VirtualBox:/var/lib/php5# ls
modules   sess_sp39odgr48v3e5d3dsl6dvcn46
root@virtualweb-VirtualBox:/var/lib/php5# cat sess_sp39odgr48v3e5d3dsl6dvcn46
loggedIn|s:3:"yes";last_login|s:19:"2016-02-19 13:48:20";root@virtualweb-Virtual
Box:/var/lib/php5#
```

To see what's stored in the session variable you can call the var_dump function with $_SESSION as its argument.

```
<?php
session_start();
var_dump($_SESSION);
```

This is the result.

```
array(2) { ["loggedIn"]=> string(3) "yes" ["last_login"]=> string(19) "2016-02-19
13:48:20" }
```

Other Options to Hide Sessions

PHP stores the session data in text files by default. Even though it has several advantages, such as accessing and reading session files rapidly, you might want to store session files elsewhere for security measures.

```
root@virtualweb-VirtualBox:/var/lib/php5# ls -l
total 8
drwxr-xr-x 5 root     root     4096 Eki 12 16:20 modules
-rw------- 1 www-data www-data    0 Şub 19 13:53 sess_037k8in537467casg4r5kdn807
-rw------- 1 www-data www-data   57 Şub 19 13:53 sess_sp39odgr48v3e5d3dsl6dvcn46
root@virtualweb-VirtualBox:/var/lib/php5#
```

Since the *www-data* has permission for the directory, which holds the session data, you have to protect the files carefully against the risk of cookie data being stolen, because that puts all users into danger.

On the other hand, for those who prefer distributed systems for their web applications, storing session data in text files is not possible. Even though multiple machines can access the files in a folder system, the sequential access will slow down the system or lock it down altogether.

Thankfully, using PHP's session handler functions, you can direct and customize all the events from generating a cookie, choosing where it's stored, to it's removal. You have to use the session_set_save_handler() function to specify the functions which will be triggered in the events from the start to the end of the session.

```
bool session_set_save_handler ( callable $open , callable $close , callable $read
, callable $write , callable $destroy , callable $gc [, callable $create_sid ] )
```

Let's take a look the six parameters assigned to this function.

1. **callable $open(string $savePath, string $sessionName)**: It will be called together with the session_start function to start the session. It must return (bool) TRUE or (bool) FALSE.
2. **callable $close**: The supplied function acts as a destructor method for the classes. It must return (bool) TRUE or (bool) FALSE.
3. **callable $read(string $sessionId)**: This function must return a serialized session data. The value returned here will be deserialized and submitted to the super global $_SESSION.
4. **callable $write(string $sessionId, string $data):** It's called when the session is going to be ended or when the session values are going to be saved.
5. **callable destroy($sessionId):** When the session is ended, this function is called. It must return TRUE or FALSE.
6. **callable gc($lifetime):** It's called randomly by PHP, and should clear up the old session data.
7. **callable create_sid()**: This is an optional parameter that expects a function that is called when a new session key is required. It has to return a string.

Cookie Attributes in Terms of Security

In this section, we will take a look at all the components of the cookies that might make an attack surface and discuss the possible attacks, their effects, and methods of protection.

As we stated above, a cookie has determinants such as a name-value pair, expires, path, domain, and httpOnly and secure flags.

name[=value] Attribute

The only obligatory parameter is the name parameter. We can evaluate the name and the value differently from the attacker's perspective.

The name parameter isn't an attack vector on it's own, but it can give some idea of the application to the attacker, allowing the user to get to know the target and form an attack strategy.

For example, PHP applications use PHPSESSID cookie by default. You can find out the major platforms/frameworks and their cookie names in the table below:

Cookie Name	Framework / Platform
JSESSIONID	Java Platform
ASPSESSIONID	IIS WEB Server
ASP.NET_sessionid	Microsoft ASP.NET
CFID/CFTOKEN	Cold Fusion
zope3	Zope
cakephp	CakePHP
kohanasession	Kohana
laravel_session	Laravel
ci_session	Codeigniter

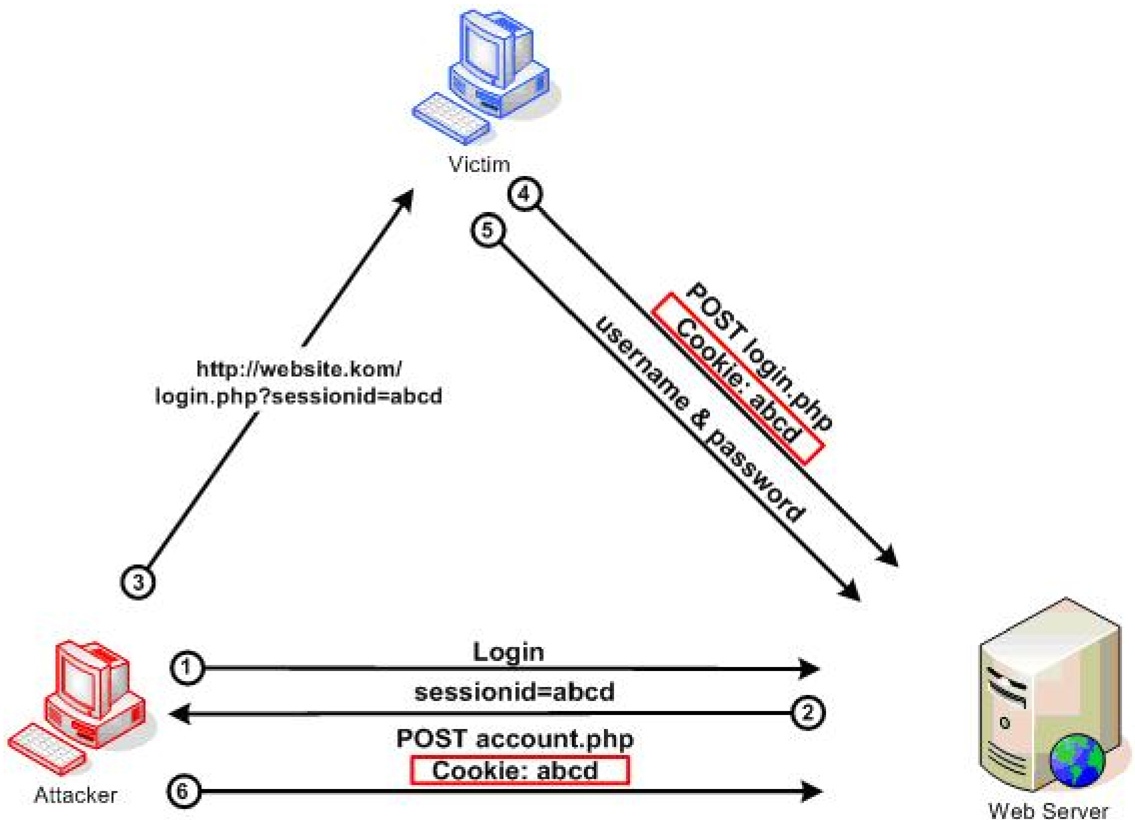
The value is the most important point that makes the user unique on the server side, and acts as a reference to their data on the server-side. If an attacker can read it, the user's session can be stolen. Therefore, the value set here must be hard to guess and unique. You can't for example use a value like user:admin and obfuscate it (e.g. by using base64). Instead you should always use a proper session ID.

Cookie Headers

But even if the session ID is cryptographically secure, attackers can still employ an attack known as Session Fixation. This means that instead of guessing the value of the session ID cookie, the attacker sets that cookie value for you. This attack often occurs if the cookie value is visible in the URL.

```
http://www.example.com/index.php?PHPSESSID=Attacker
```

When you click such a link, or visit it unknowingly, e.g. through a hidden iframe on another page or as one of multiple redirects, your new session ID will be set to the string "Attacker". If you log into your account, PHP will write any session information, such as your username, email address, access level and other things into the file associated with this ID. However, the attacker now knows the value of your session ID, since he set it himself. All he has to do is to set his own session ID to the string "Attacker". Since PHP associates this session ID with your account and the attacker sends it to the vulnerable application, it will think that the attacker and you are the same person. This grants the attacker access to your account. There are ways to mitigate this, for example by storing the IP address of the user in the session and compare it to the IP of the user who makes the request. That means only one person can be logged in at any given time, but it might also cause issues for users whose IP often changes, such as TOR and mobile users.



"Simple example of Session Fixation attack" by OWASP is licensed under CC BY-SA 4.0.

Solution to a Secure Session

After a successful authorization process, the session keys must be regenerated and sent to the user with the code below. It's important to regenerate the session ID in order to avoid session fixation attacks.

```
<?php
$_SESSION['logged_in'] = FALSE;
if (check_login())
{
 session_regenerate_id();
 $_SESSION['logged_in'] = TRUE;
}
```

domain Attribute

The domain attribute is crucial for the security of cookies. Let's look at a website with subdomains to see why that's the case.

Before anything else happens the cookies' domain and the hostname of the requested URL are compared. The other criteria are checked only if that comparison was successful.

But what is considered as "same origin" yet again differs significantly from JavaScript's understanding of the term. While JavaScript requires an exact match of two hostnames, cookies are a little less picky as it's perfectly legal to set a cookie for a subdomain.

Example of Setting the domain Attribute

Let's assume there are two different accounts created on the webhoster *badsites.local*. Each of them has their own subdomain.

victim.badsites.local
attacker.badsites.local

If you visit the domains above, you can read the content their owners made available. However, if the owners want to change said content they need to login to *badsites.local* and do the necessary changes over the control panel.

When the user *victim* logs in to the system, this is the HTTP request their browser sends.

```
POST http://badsites.local/control.php HTTP/1.1
Host: badsites.local
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 35
```

```
username=victim&password=victim
```

There are three possible domain property values the cookie might contain. To make things even more complicated, different browsers might behave differently for each of them. The browsers used in the following test are Chrome 69.0.3497.100, Internet Explorer 11, Mozilla Firefox 44.0.2 and Edge 42.17134.1.0.

Case 1: Domain value of the cookie is not set:

```
HTTP/1.1 200 OK
Set-Cookie: PHPSESSID=ock3keuidn0t24vrf4hkvtopm0; path=/;
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
```

As you can see the domain parameter wasn't set. All of the tested browsers react similarly. Google Chrome, Edge, Internet Explorer 11 and Firefox will not send the cookie with the requests done for the subdomains.

There is an exception though, since older versions of Internet Explorer (such as 11.0.10240.17443 and below) will send this cookie even if one of the subdomains under *badsites.local* is requested. That means, that once the owner visits attacker.badsites.local, while being logged in to badsites.local, the attacker can read the victims cookies and take over their account.

```
GET / HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: tr,en-US;q=0.7,en;q=0.3
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; rv:11.0) like Gecko
Host: attacker.badsites.local
Cookie: PHPSESSID=ock3keuidn0t24vrf4hkvtopm0
```

Case 2: Cookie domain value is set to badsites.local:

You can see the second possibility below. The domain attribute might be set to badsites.local.

```
HTTP/1.1 200 OK
Server: Apache/2.4.7 (Ubuntu)
```

```
X-Powered-By: PHP/5.5.9-1ubuntu4.14
Set-Cookie: PHPSESSID=1fr54qg3j9rf77toohcpcsk8h0; path=/; domain=badsites.local
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Length: 66
Content-Type: text/html
```

All of the tested browsers – IE, Edge, Chrome, and Firefox – will add the cookie generated by *badsites.local* to the requests to *attacker.badsites.local.*

```
GET / HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: tr,en-US;q=0.7,en;q=0.3
Host: attacker.badsites.local
Pragma: no-cache
Cookie: PHPSESSID=1fr54qg3j9rf77toohcpcsk8h0
```

Yet again, this may lead to an account takeover if you visit a malicious badsites.local subdomain.

Case 3: Cookie domain value is set as .badsites.local:

```
HTTP/1.1 200 OK
Set-Cookie: PHPSESSID=q3a20kfes2u6fgvgsrspv0rpf0; path=/; domain=.badsites.local;
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
```

IE, Edge, Chrome, and Firefox will add the cookie to requests made for *attacker.badsites.local.*

```
GET / HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: tr,en-US;q=0.7,en;q=0.3
Proxy-Connection: Keep-Alive
Host: attacker.badsites.local
Cookie: PHPSESSID=q3a20kfes2u6fgvgsrspv0rpf0
```

As seen in the cases above, cookie domain values have to be set with great care. If the correct values aren't given, your application might be at great risk. So what could the badsites.local host have done better to avoid all of this confusion and inconsistency between browsers? The answer is simple. They could have used the www subdomain for their control panel. Doing so restricts the cookies to *www.badsites.local* and its subdomains even if you leave the domain value empty.

In situations where you have to host websites with multiple users, do not host the control panel or other potentially risky websites under the main domain. For example, GitHub, the most popular hoster for git repositories also allows to upload HTML and markup files in order enable developers to create accompanying websites for their repositories. These may contain content such as product documentation, or personal websites. The sites are created under *github.com* as a repository containing these files. However, they are hosted on a completely different domain, namely *github.io*. http://github.io doesn't contain any content. Instead it just redirects to pages.github.com. This separation of content and content management makes it easy to avoid domain problems with cookies.

path Attribute

path is another important feature for the cookie security. Browsers check the *path* value right after they check *domain*.

Just like the domain attribute's value, the *path* value might result in critical security vulnerabilities, if not carefully set.

The *path* feature is optional, and the default value is "/", which means that the cookie will be sent together with every request, regardless of its path.

Unlike the *domain* attribute, the *path* value is checked from left to right.

But like the domain attribute, browsers behave differently. These differences may result in vulnerabilities in your applications.

Example of Setting *path* Attribute

We're going to test two cases where the *path* value does and does not have the "/" sign in the end. We'll be sending requests to http://badsites.local/victim/, http://badsites.local/victim/sub/, and http://badsites.local/victim-fake/ from Firefox, Edge, Chrome, and Internet Explorer browsers. The page that set the cookie is http://badsites.local/victim/

Case 1: Path value doesn't have "/" at the end (E.g. /victim)

```
HTTP/1.1 200 OK
Set-Cookie: PHPSESSID=r9evv4bft7luq4h7c4l5q8b1o4; path=/victim;
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
```

IE and Edge sends the cookie in all three requests:

```
GET /victim/ HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: tr,en-US;q=0.7,en;q=0.3
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; rv:11.0) like Gecko
Proxy-Connection: Keep-Alive
Host: badsites.local
Cookie: PHPSESSID=r9evv4bft7luq4h7c4l5q8b1o4
```

```
GET /victim/sub/ HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: tr,en-US;q=0.7,en;q=0.3
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; rv:11.0) like Gecko
Proxy-Connection: Keep-Alive
Host: badsites.local
Cookie: PHPSESSID=r9evv4bft7luq4h7c4l5q8b1o4
```

```
GET /victim-fake/ HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: tr,en-US;q=0.7,en;q=0.3
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; rv:11.0) like Gecko
Proxy-Connection: Keep-Alive
Host: badsites.local
Cookie: PHPSESSID=r9evv4bft7luq4h7c4l5q8b1o4
```

In Chrome and Firefox, a cookie is sent only with the requests to http://badsites.local/victim and http://badsites.local/victim/sub.

Case 2: path value has "/" at the end (E.g. /victim/)

```
HTTP/1.1 200 OK
Set-Cookie: PHPSESSID=j0sbcvo5h8q8a1g6n714kmaqo5; path=/victim/;
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
```

The Internet Explorer, Edge, Chrome, and Firefox browsers send the cookie only with the requests to http://badsites.local/victim and http://badsites.local/victim/sub.

Due to this difference in the browsers, the websites hosted under the sub directories can access the session variables set for other directories. Especially if you're using a module like the Apache *mod_userdir*, you have to be careful about the value set for the cookies.

Solution

If you're going to set a path value for the cookie, you should put a "/" at the end of the value.

expires and max-age

These attributes are optional and they specify the duration of storing the cookie. If they are not specified, the browser will treat them as a session cookie and will delete them once the browser is closed.

The purpose of the cookie is the deciding factor for the appropriate time-out settings. Large `expire` or `max-age` values might pose security risks when using devices that are shared among multiple users.

As mentioned above, `expire` and `max-age` can be used simultaneously. However, you should note that browsers will prioritize the `max-age` value if you use both attributes.

httpOnly Flag

Cookies are not only used to control the user session, but also to help applications customize their user preferences. Many scripts store their customization settings as cookies, and access and modify these settings using JavaScript.

For example, to set the user's language preferences, you set the following:

```
Set-Cookie: prefLanguage=en;
```

Or, if you want the browser to remember that the user wants to view the videos fullscreen, you set the following:

```
Set-Cookie: fullScreen=yes;
```

However, the cookies that hold the session data are only crucial to the server, since they hold the reference keys to the session data. Hence, they don't have to be read by JavaScript. In the presence of an XSS vulnerability, cookies read by JavaScript may make the exploitation of such a vulnerability easier by allowing session hijacking. Therefore, the session data cookies must have the `httpOnly` value set.

For example:

```
Set-Cookie: PHPSESSID=a984308kdf9845; path=/; httpOnly;
```

secure Flag

HTTP requests are transferred as plaintext between the client and the server. Someone listening to the network using a

Man in the Middle (MiTM) attack may acquire the session data, one of the most crucial types of data for web browsing.

You can prevent this threat by specifying the `secure` attribute when you create cookies. This attribute ensures that the cookies are only sent over a secure (HTTPS) connection.

```
Set-Cookie: PHPSESSID=a984308kdf9845; path=/; secure;
```

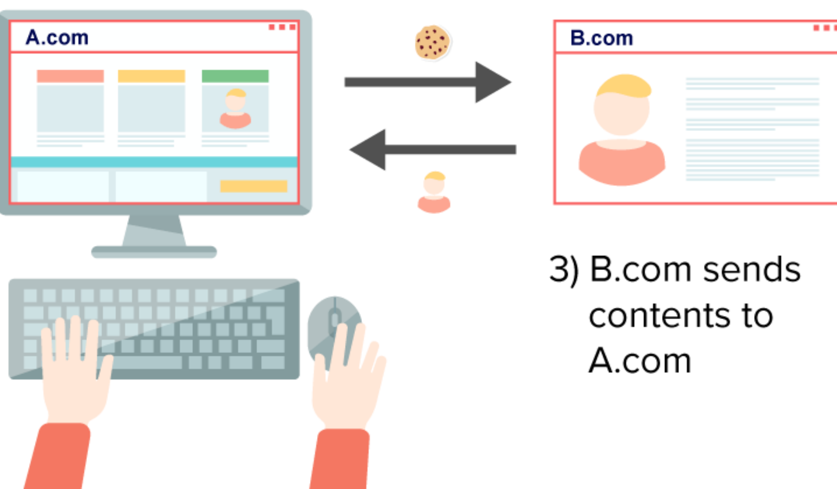
But you shouldn't just send the session cookies exclusively over HTTPS. Even if the cookie doesn't seem important first, it might have huge implications when it comes to the privacy of your users. There is a Blackhat USA talk from 2016, that explains this in detail.

SameSite Attribute

When browsers are instructed to open a website, they check whether there's a matching cookie in their cookie database. After successfully checking the domain, path and secure attributes, they will send the respective cookie with the request.

This doesn't just happen for the page you want to visit, but for all the external JavaScript, CSS, and image files from any third-party sources.

The critical point here is that when you visit website attacker.com, all requests made from attacker.com to example.com will contain the cookie from example.com. By default, this doesn't have any security implications for the content of the cookies. Even though the attacker instructs the browser to send example.com's cookies to example.com, attacker.com can't read them.



However, there is a common technique that takes advantage of the fact that cookies are always sent to the respective website, no matter where the request originated from, namely the Cross-Site Request Forgery (CSRF) attack. Technically speaking, nothing would prevent an attacker from creating a form like this:

```
<form method = "POST" action = "https://www.example.com/account/edit">
    <input name = "user_biography" value = "pwned by attacker.com!">
</form>
<script>document.forms[0].submit()</script>
```

Being able to change the user's profile is a rather harmless side effect. It becomes much more serious if you can change an email addresses. If an attacker changes the email address of the user to his *own* email address, he can then trigger

the password reset functionality and reset the user's password. This allows him to take over the account.

When you visit a website, your browser sends requests on everything found on that website, including the embedded code for social media sharing on the page like Facebook, Google Analytics and CDN resources, for example. The cookies set by the third-party resources on the browser are sent alongside these requests. The name of the website that sends the request is found in the `Referer` header in the request. In this way, third-parties use the cookies with the referrer data and discover the browsing habits of the users.

Normally, you can prevent this tracking by disabling the third-party cookies in Firefox and Chrome. However, this might have a negative impact on your browsing experience.

The SameSite cookie feature allows administrators to restrict to which requests cookies are added. The SameSite parameter allows you to remove cookies from certain requests if they weren't issued from their own website, instead of disabling all of them.

Setting the SameSite Cookie is pretty simple. All you have to do is to add `SameSite=Lax` or `SameSite=Strict` parameters to your cookie

```
Set-Cookie: CookieName=CookieValue; SameSite=Lax;
Set-Cookie: CookieName=CookieValue; SameSite=Strict;
```

Strict and Lax Values

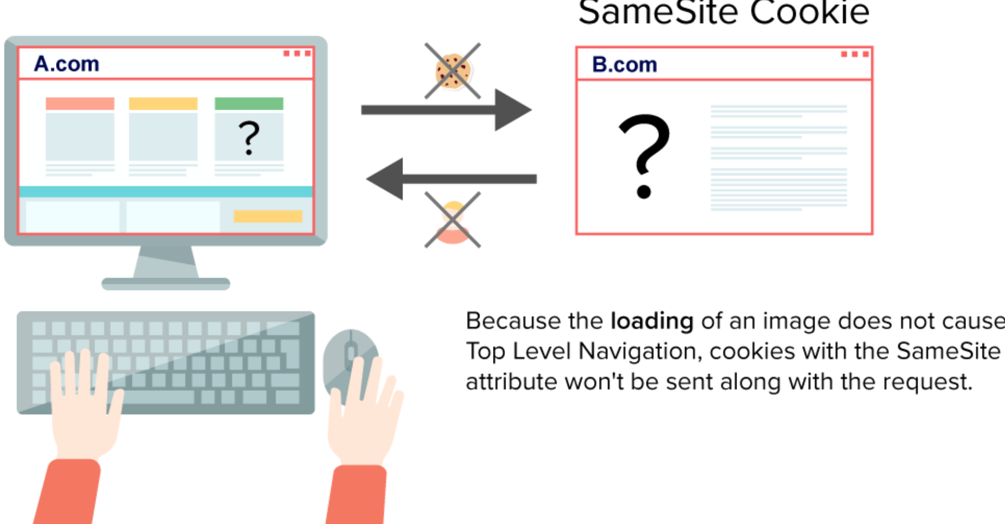
Strict and Lax refer to different levels of security.

Strict

If you use the `Strict` value, the cookies will not be sent when a third-party issues the requests. In some cases, this option might negatively impact your browsing experience. For example, if a website you visit by clicking on a link has the `SameSite=Strict` set, the website might request you to log in again, since the cookie won't be sent along with the request.

Lax

If you use the lax value, this allows cookies to be sent if the third-party issues a `GET` request that affects Top Level Navigation, which means that the request will change the address bar. Only those requests will be allowed to have the cookie sent with the Lax value.



You can load a source with the `iframe`, `img`, or `script` tags. These resources will be called with `GET` but they will not change the address in the address bar, since they don't need Top Level navigation. That's why the cookies will not be sent with requests made to those sources.

This table summarizes the request types and the cookies sent to them.

Request Type	Example Code	Cookies sent
Link		Normal, Lax
Perender	<link rel="prerender" href=".."/>	Normal, Lax
Form GET	<form method="GET" action="...">	Normal, Lax
Form POST	<form method="POST" action="...">	Normal
iframe	<iframe src="..."></iframe>	Normal
AJAX	$.get("...")	Normal
Image	<img src="...">	Normal

Cookie Prefixes

The most precise point in the relationship between cookie and session is that the server looks up the file in the database that matches the cookie value and transfers this to a session variable. If the server doesn't find a file, it makes a new one with the appropriate name.

The attacks are possible if developers do not take extra measures. For example, if the cookie is regenerated each time the session is online, the Session Fixation attack discussed above will not be possible. This is the first reason why cookie prefixes exist.

The second reason is similar. The individualizing factors of a cookie are `domain`, `path`, and `name`. The other flags, `secure`, `httpOnly`, and `SameSite` do not act as a differentiating factor in the browser's cookie jar.

Examples of the Importance of Prefixes

For example, let's say you set a cookie like this:

```
Set-Cookie: MyCookie=value; path=/;
```

The cookie may be overridden by these:

```
Set-Cookie: MyCookie=newvalue; path=/; secure;
Set-Cookie: MyCookie=newvalue; path=/; httpOnly;
```

A different scenario might be possible. Let's say an attacker takes advantage of an XSS vulnerability on your website but he can't access your cookie set with `httpOnly`. He overrides the secure session cookie using JavaScript, which results in an overwritten cookie and would even allow the attacker to carry out a session fixation attack.

We can break the attack down with the example below. This is your cookie:

```
Set-Cookie: MyCookie=value; path=/; httpOnly;
```

An attacker uses the code below to override the cookie:

```
document.cookie="MyCookie=AttackerValue; path=/;"
```

In the same way, if an attacker cannot access the cookie set as secure by intercepting your network requests, she can force the user's browser to connect over HTTP and reach that cookie.

Here's a dangerous scenario possible on IE or Edge browsers.

The cookie set by example.com:

```
Set-Cookie: MyCookie=SetByExample.com; path=/;
```

hacker.example.com returns the following:

```
Set-Cookie: MyCookie=hacked; path=/;
```

The cookie set by example.com, `MyCookie`, will have its value overridden to 'hacked', too.

What are the solutions to these issues? Cookie prefixes prevent cookies from overrides through non-*secure* connections, subdomains, and JavaScript.

__Secure- Prefix

When the `__Secure-` prefix is added to a cookie name, the cookie is accessible from HTTPS connections only.

Here's an example:

```
Set-Cookie: __Secure-MyCookie=value; path=/; secure;
```

__Host- Prefix

When the `__Host-` prefix is added to a cookie name, it performs the same job as `__Secure-`. But it also ensures that only the domain that sets the cookie can override the cookie. Subdomains cannot alter the cookie. The domain attribute should not be set and the path should be set as '/' for this prefix to work.

```
Set-Cookie: __Host-MyCookie=value; path=/; secure;
```

It is important to note that the attack above is only possible in IE and Edge browsers. The feature, suggested by Eric Lawrence, has been revised in 2016 by Mike West, and is no longer supported by IE and Edge browsers.

Extra Measures

Here are some recommendations:

- Use each cookie for a single task. Do not use the session cookies for other operations, such as password resetting. Using cookies for multiple tasks might result in application workflow complications, or vulnerabilities. Session Puzzling is an example of how dangerous it is to use cookies for multiple tasks.
- Use the optimal settings for each application session and choose appropriate `max-age` or `expires` values, depending on the application's type. For example, for a forum or a wiki site, a one hour expiration time is ideal, or for a banking application, five minutes would suffice. Once the time is up, the session variables will be removed from the server.
- When the session is ended, you have to ensure that the variables are no longer valid on both the server and client side. If the session isn't ended on the server side, attacks such as Cookie Replay might be possible.

Conclusion

In this article, we discussed how applications use cookies and session objects to allow for secure storage of session related data on the server side.

However, it is alarming to see that from the day their specification was released, cookies haven't been used properly, and people still don't set them correctly.

Many web applications are vulnerable to attacks due to the incorrect implementation of cookie values. Automatized tools cannot always test the features entirely. Therefore, developers have to set cookies with the greatest of care, and test how they behave in various browsers before releasing them into the wild.

Author:
Ziyahan Albeniz

Translators:
Sven Morgenroth
Umran Yildirimkaya