A Guide to Building Secure Web Applications

Essential PHP Security

O'REILLY®

Chris Shiflett

CHAPTER 4

Sessions and Cookies

This chapter discusses sessions and the inherent risks associated with stateful web applications. You will first learn the fundamentals of state, cookies, and sessions; then I will discuss several concerns—cookie theft, exposed session data, session fixation, and session hijacking—along with practices that you can employ to help prevent them.

The rumors are true: HTTP is a stateless protocol. This description recognizes the lack of association between any two HTTP requests. Because the protocol does not provide any method that the client can use to identify itself, the server cannot distinguish between clients.

While the stateless nature of HTTP has some important benefits—after all, maintaining state requires some overhead—it presents a unique challenge to developers who need to create stateful web applications. With no way to identify the client, it is impossible to determine whether the user is already logged in, has items in a shopping cart, or needs to register.

An elegant solution to this problem, originally conceived by Netscape, is a state management mechanism called cookies. Cookies are an extension of the HTTP protocol. More precisely, they consist of two HTTP headers: the `Set-Cookie` response header and the `Cookie` request header.

When a client sends a request for a particular URL, the server can opt to include a `Set-Cookie` header in the response. This is a request for the client to include a corresponding `Cookie` header in its future requests. Figure 4-1 illustrates this basic exchange.

If you use this concept to allow a unique identifier to be included in each request (in a `Cookie` header), you can begin to uniquely identify clients and associate their requests together. This is all that is required for state, and this is the primary use of the mechanism.

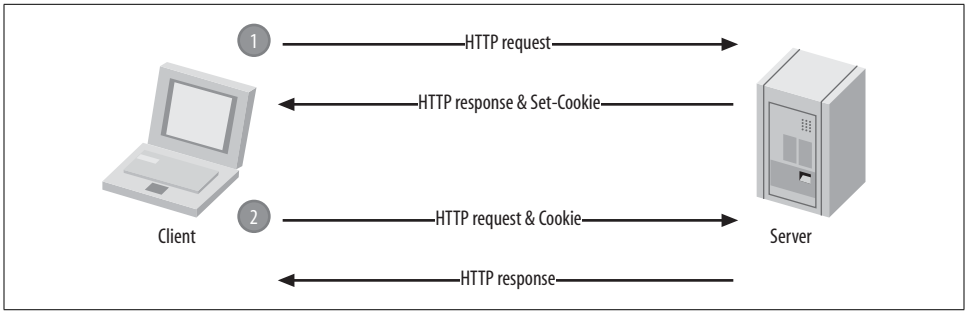
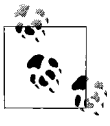


Figure 4-1. A complete cookie exchange that involves two HTTP transactions



The best reference for cookies is still the specification provided by Netscape at *http://wp.netscape.com/newsref/std/cookie_spec.html*. This most closely resembles industry support.

The concept of session management builds upon the ability to maintain state by maintaining data associated with each unique client. This data is kept in a session data store, and it is updated on each request. Because the unique identifier specifies a particular record in the session data store, it's most often called the session identifier.

If you use PHP's native session mechanism, all of this complexity is handled for you. When you call `session_start()`, PHP first determines whether a session identifier is included in the current request. If one is, the session data for that particular session is read and provided to you in the `$_SESSION` superglobal array. If one is not, PHP generates a session identifier and creates a new record in the session data store. It also handles propagating the session identifier and updating the session data store on each request. Figure 4-2 illustrates this process.

While this convenience is helpful, it is important to realize that it is not a complete solution. There is no inherent security in PHP's session mechanism, aside from the fact that the session identifier it generates is sufficiently random, thereby eliminating the practicality of prediction. You must provide your own safeguards to protect against all other session attacks. I will show you a few problems and solutions in this chapter.

Cookie Theft

One risk associated with the use of cookies is that a user's cookies can be stolen by an attacker. If the session identifier is kept in a cookie, cookie disclosure is a serious risk, because it can lead to session hijacking.

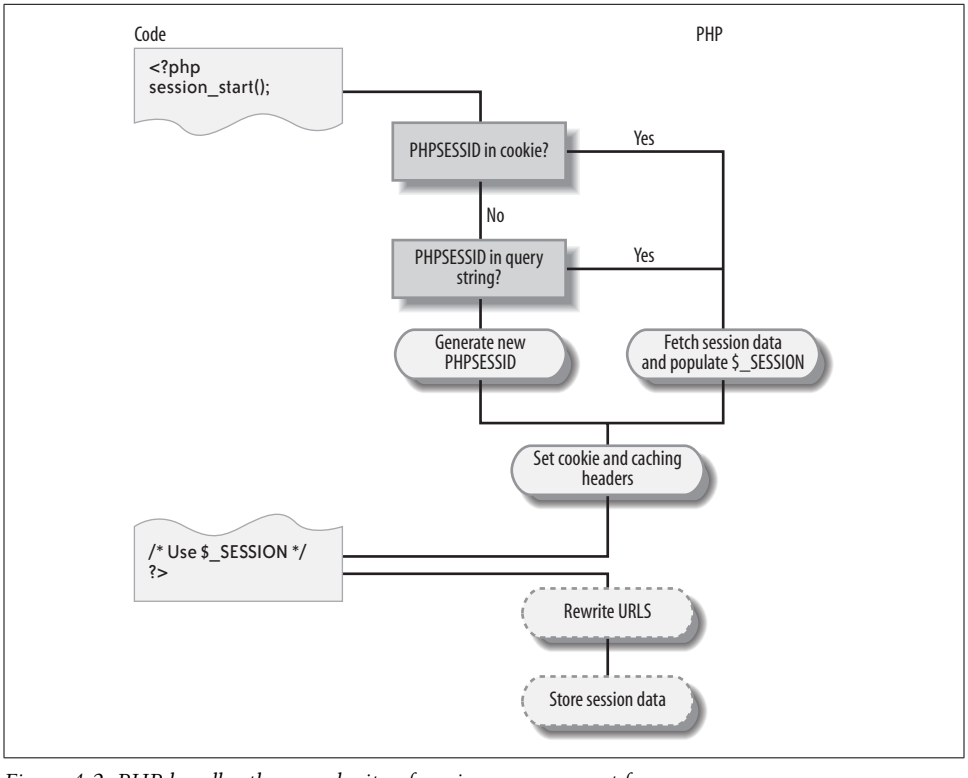


Figure 4-2. PHP handles the complexity of session management for you

The two most common causes of cookie disclosure are browser vulnerabilities and cross-site scripting (discussed in Chapter 2). While no such browser vulnerabilities are known at this time, there have been a few in the past—the most notable ones are in Internet Explorer Versions 4.0, 5.0, 5.5, and 6.0 (corrective patches are available for each of these vulnerabilities).

While browser vulnerabilities are certainly not the fault of web developers, you may be able to take steps to mitigate the risk to your users. In some cases, you may be able to implement safeguards that practically eliminate the risk. At the very least, you can try to educate your users and direct them to a patch to fix the vulnerability.

For these reasons, it is good to be aware of new vulnerabilities. There are a few web sites and mailing lists that you can keep up with, and many services are beginning to offer RSS feeds, so that you can simply subscribe to the feed and be alerted to new vulnerabilities. SecurityFocus maintains a list of software vulnerabilities at *http://online.securityfocus.com/vulnerabilities*, and you can filter these advisories by vendor, title, and version. The PHP Security Consortium also maintains summaries of the SecurityFocus newsletters at *http://phpsec.org/projects/vulnerabilities/securityfocus.html*.

Cross-site scripting is a more common approach used by attackers to steal cookies. An attacker can use several approaches, one of which is described in Chapter 2. Because client-side scripts have access to cookies, all an attacker must do is write a script that delivers this information. Creativity is the only limiting factor.

Protecting your users from cookie theft is therefore a combination of avoiding cross-site scripting vulnerabilities and detecting browsers with security vulnerabilities that can lead to cookie exposure. Because the latter is so uncommon (with any luck, these types of vulnerabilities will remain a rarity), it is not the primary concern but rather something to keep in mind.

Exposed Session Data

Session data often consists of personal information and other sensitive data. For this reason, the exposure of session data is a common concern. In general, the exposure is minimal, because the session data store resides in the server environment, whether in a database or the filesystem. Therefore, session data is not inherently subject to public exposure.

Enabling SSL is a particularly useful way to minimize the exposure of data being sent between the client and the server, and this is very important for applications that exchange sensitive data with the client. SSL provides a layer of security beneath HTTP, so that all data within HTTP requests and responses is protected.

If you are concerned about the security of the session data store itself, you can encrypt it so that session data cannot be read without the appropriate key. This is most easily achieved in PHP by using `session_set_save_handler()` and writing your own session storage and retrieval functions that encrypt session data being stored and decrypt session data being read. See Appendix C for more information about encrypting a session data store.

Session Fixation

A major concern regarding sessions is the secrecy of the session identifier. If this is kept secret, there is no practical risk of session hijacking. With a valid session identifier, an attacker is much more likely to successfully impersonate one of your users.

An attacker can use three primary methods to obtain a valid session identifier:

- Prediction
- Capture
- Fixation

PHP generates a very random session identifier, so prediction is not a practical risk. Capturing a session identifier is more common—minimizing the exposure of the session identifier, using SSL, and keeping up with browser vulnerabilities can help you mitigate the risk of capture.

Keep in mind that a browser includes a `Cookie` header in all requests that satisfy the requirements set forth in a previous `Set-Cookie` header. Quite commonly, the session identifier is being exposed unnecessarily in requests for embedded resources, such as images. For example, to request a web page with 10 images, the session identifier is being sent by the browser in 11 different requests, but it is needed for only 1 of those. To avoid this unnecessary exposure, you might consider serving all embedded resources from a server with a different domain name.

Session fixation is an attack that tricks the victim into using a session identifier chosen by the attacker. It is the simplest method by which the attacker can obtain a valid session identifier.

In the simplest case, a session fixation attack uses a link:

```
<a href="http://example.org/index.php?PHPSESSID=1234">Click Here</a>
```

Another approach is to use a protocol-level redirect:

```
<?php

header('Location: http://example.org/index.php?PHPSESSID=1234');

?>
```

The `Refresh` header can also be used—provided as an actual HTTP header or in the `http-equiv` attribute of a `meta` tag. The attacker's goal is to get the user to visit a URL that includes a session identifier of the attacker's choosing. This is the first step in a basic attack; the complete attack is illustrated in Figure 4-3.

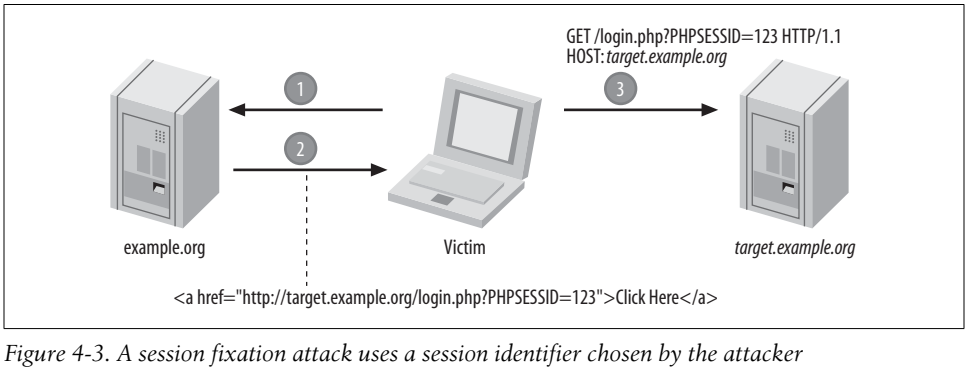


Figure 4-3. A session fixation attack uses a session identifier chosen by the attacker

If successful, the attacker is able to avoid the necessity of capturing or predicting a valid session identifier, and it is possible to launch additional and more dangerous types of attacks.

A good way to better understand this is to try it yourself. Begin with a script named *fixation.php*:

```
<?php

session_start();

$_SESSION['username'] = 'chris';

?>
```

Ensure that you do not have any existing cookies for the current host, or clear all cookies to be certain. Visit `fixation.php` and include `PHPSESSID` in the URL:

```
http://example.org/fixation.php?PHPSESSID=1234
```

This creates a session variable (`username`) with a value of `chris`. An inspection of the session data store reveals that `1234` is the session identifier associated with this data:

```
$ cat /tmp/sess_1234
username|s:5:"chris";
```

Create a second script, *test.php*, that outputs the value of `$_SESSION['username']` if it exists:

```
<?php

session_start();

if (isset($_SESSION['username']))
{
  echo $_SESSION['username'];
}

?>
```

Visit this URL using a different computer, or at least a different browser, and include the same session identifier in the URL:

```
http://example.org/test.php?PHPSESSID=1234
```

This causes you to resume the session you began when you visited *fixation.php*, and the use of a different computer (or different browser) mimics an attacker's position. You have successfully hijacked a session, and this is exactly what an attacker can do.

Clearly, this is not desirable. Because of this behavior, an attacker can provide a link to your application, and anyone who uses this link to visit your site will use a session identifier chosen by the attacker.

One cause of this problem is that a session identifier in the URL is used to create a new session—even when there is no existing session for that particular session identifier, PHP creates one. This provides a convenient opening for an attacker. Luckily, the `session_regenerate_id( )` function can be used to help prevent this:

```
<?php

session_start();

if (!isset($_SESSION['initiated']))
{
  session_regenerate_id();
  $_SESSION['initiated'] = TRUE;
}

?>
```

This ensures that a fresh session identifier is used whenever a session is initiated. However, this is not an effective solution because a session fixation attack can still be successful. The attacker can simply visit your web site, determine the session identifier that PHP assigns, and use that session identifier in the session fixation attack.

This does eliminate the opportunity for an attacker to assign a simple session identifier such as `1234`, but the attacker can still examine the cookie or URL (depending upon the method of propagation) to get the session identifier assigned by PHP. This approach is illustrated in Figure 4-4.

To address this weakness, it helps to understand the scope of the problem. Session fixation is merely a stepping-stone—the purpose of the attack is to get a session identifier that can be used to hijack a session. This is most useful when the session being hijacked has a higher level of privilege than the attacker can obtain through legitimate means. This level of privilege can be as simple as being logged in.

If the session identifier is regenerated every time there is a change in the level of privilege, the risk of session fixation is practically eliminated:

```
<?php

$_SESSION['logged_in'] = FALSE;

if (check_login())
{
  session_regenerate_id();
  $_SESSION['logged_in'] = TRUE;
}

?>
```

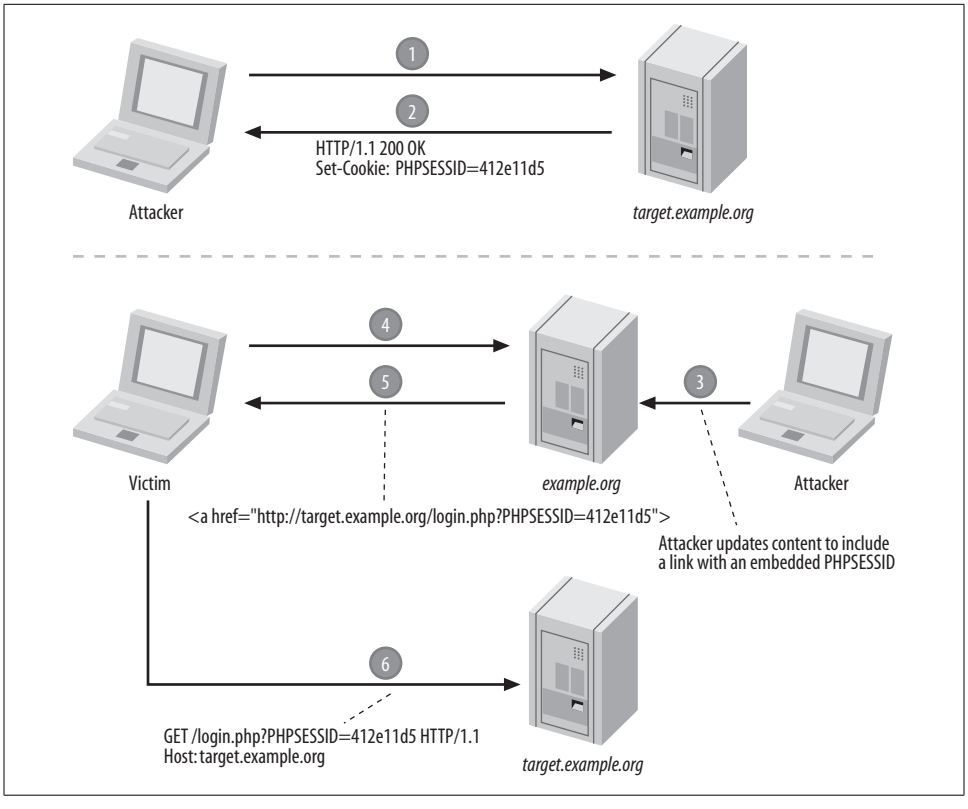


Figure 4-4. A session fixation attack can first initialize the session

I do not recommend regenerating the session identifier on every page. While this seems like a secure approach—and it is—it provides no more protection than regenerating the session identifier whenever there is a change in the level of privilege. More importantly, it can adversely affect your legitimate users, especially if the session identifier is being propagated in the URL. A user might use the browser's history mechanism to return to a previous page, and the links on that page will reference a session identifier that no longer exists.

If you regenerate the session identifier only when there is a change in the level of privilege, the same situation is possible, but a user who returns to a page prior to the change in the level of privilege is less likely to be surprised by a loss of session, and this situation is also less common.

Session Hijacking

The most common session attack is session hijacking. This refers to any method that an attacker can use to access another user's session. The first step for any attacker is to obtain a valid session identifier, and therefore the secrecy of the session identifier is paramount. The previous sections on exposure and fixation can help you to keep the session identifier a shared secret between the server and a legitimate user.

The principle of Defense in Depth (described in Chapter 1) can be applied to sessions—some minor safeguards can offer some protection in the unfortunate case that the session identifier is known by an attacker. As a security-conscious developer, your goal is to complicate impersonation. Every obstacle, however minor, offers some protection.

The key to complicating impersonation is to strengthen identification. The session identifier is the primary means of identification, and you want to select other data that you can use to augment this. The only data you have available is the data within each HTTP request:

```
GET / HTTP/1.1
Host: example.org
User-Agent: Firefox/1.0
Accept: text/html, image/png, image/jpeg, image/gif, */*
Cookie: PHPSESSID=1234
```

You want to recognize consistency in requests and treat any inconsistent behavior with suspicion. For example, while the `User-Agent` header is optional, clients that send it do not often alter its value. If the user with a session identifier of `1234` has been using Mozilla Firefox consistently since logging in, a sudden switch to Internet Explorer should be treated with suspicion. For example, prompting for the password is an effective way to mitigate the risk with minimal impact to your legitimate users in the case of a false alarm. You can check for `User-Agent` consistency as follows:

```
<?php

session_start();

if (isset($_SESSION['HTTP_USER_AGENT']))
{
  if ($_SESSION['HTTP_USER_AGENT'] != md5($_SERVER['HTTP_USER_AGENT']))
  {
    /* Prompt for password */
    exit;
  }
}
else
{
  $_SESSION['HTTP_USER_AGENT'] = md5($_SERVER['HTTP_USER_AGENT']);
}

?>
```

I have observed that some versions of Internet Explorer send a different `Accept` header depending upon whether the user refreshes the browser, so `Accept` should not be relied upon for consistency.

Requiring a consistent `User-Agent` helps, but if the session identifier is being propagated in a cookie (the recommended approach), it is reasonable to assume that, if an attacker can capture the session identifier, he can most likely capture the value of all other HTTP headers as well. Because cookie disclosure typically involves a browser vulnerability or cross-site scripting, the victim has most likely visited the attacker's web site, disclosing all headers. All an attacker must do is reproduce all of these to avoid any consistency check that uses HTTP headers.

A better approach is to propagate a token in the URL—something that can be considered a second (albeit much weaker) form of identification. This propagation takes some work—there is no feature of PHP that does it for you. For example, assuming the token is stored in `$token`, all internal links in your application need to include it:

```
<?php

$url = array();
$html = array();

$url['token'] = rawurlencode($token);
$html['token'] = htmlentities($url['token'], ENT_QUOTES, 'UTF-8');

?>

<a href="index.php?token=<?php echo $html['token']; ?>">Click Here</a>
```

To make propagation a bit easier to manage, you might consider keeping the entire query string in a variable. You can append this variable to all of your links, which makes it easy to refactor your code later, even if you don't implement this technique initially.

The token needs to be something that cannot be predicted, even under the condition that the attacker knows all of the HTTP headers that the victim's browser typically sends. One way to achieve this is to generate the token using a random string:

```
<?php

$string = $_SERVER['HTTP_USER_AGENT'];
$string .= 'SHIFLETT';

$token = md5($string);
$_SESSION['token'] = $token;

?>
```

When you use a random string (SHIFLETT in this example), prediction is impractical. In this case, capturing the token is easier than predicting it, and by propagating the token in the URL and the session identifier in a cookie, multiple attacks are needed to capture both. The exception is when the attacker can observe the victim's raw HTTP requests as they are sent to your application, because this discloses everything. This type of attack is more difficult (and therefore less likely), and it can be mitigated by using SSL.

Some experts warn against relying on the consistency of User-Agent. The concern is that an HTTP proxy in a cluster can modify User-Agent inconsistently with other proxies in the same cluster.

If you do not want to depend on User-Agent consistency, you can generate a random token:

```
<?php

$token = md5(uniqid(rand(), TRUE));
$_SESSION['token'] = $token;

?>
```

This approach is slightly weaker, but it is much more reliable. Both methods provide a strong defense against session hijacking. The appropriate balance between security and reliability is up to you.