PHP 5 Form Handling

The PHP superglobals $\_GET and $\_POST are used to collect form-data.

## **PHP - A Simple HTML Form**

The example below displays a simple HTML form with two input fields and a submit button:

## **Example**

<html>  
<body>  
  
<form action="welcome.php" method="post">  
Name: <input type="text" name="name"><br>  
E-mail: <input type="text" name="email"><br>  
<input type="submit">  
</form>  
  
</body>  
</html>

[**Run example »**](http://www.w3schools.com/php/showphp.asp?filename=demo_form_post)

When the user fills out the form above and clicks the submit button, the form data is sent for processing to a PHP file named "welcome.php". The form data is sent with the HTTP POST method.

To display the submitted data you could simply echo all the variables. The "welcome.php" looks like this:

<html>  
<body>  
  
Welcome <?php echo $\_POST["name"]; ?><br>  
Your email address is: <?php echo $\_POST["email"]; ?>  
  
</body>  
</html>

The output could be something like this:

Welcome John  
Your email address is john.doe@example.com

The same result could also be achieved using the HTTP GET method:

## **Example**

<html>  
<body>  
  
<form action="welcome\_get.php" method="get">  
Name: <input type="text" name="name"><br>  
E-mail: <input type="text" name="email"><br>  
<input type="submit">  
</form>  
  
</body>  
</html>

[**Run example »**](http://www.w3schools.com/php/showphp.asp?filename=demo_form_get)

and "welcome\_get.php" looks like this:

<html>  
<body>  
  
Welcome <?php echo $\_GET["name"]; ?><br>  
Your email address is: <?php echo $\_GET["email"]; ?>  
  
</body>  
</html>

The code above is quite simple. However, the most important thing is missing. You need to validate form data to protect your script from malicious code.

|  |  |
| --- | --- |
| **Note** | **Think SECURITY when processing PHP forms!**  This page does not contain any form validation, it just shows how you can send and retrieve form data.  However, the next pages will show how to process PHP forms with security in mind! Proper validation of form data is important to protect your form from hackers and spammers! |

## **GET vs. POST**

Both GET and POST create an array (e.g. array( key => value, key2 => value2, key3 => value3, ...)). This array holds key/value pairs, where keys are the names of the form controls and values are the input data from the user.

Both GET and POST are treated as $\_GET and $\_POST. These are superglobals, which means that they are always accessible, regardless of scope - and you can access them from any function, class or file without having to do anything special.

$\_GET is an array of variables passed to the current script via the URL parameters.

$\_POST is an array of variables passed to the current script via the HTTP POST method.

## **When to use GET?**

Information sent from a form with the GET method is **visible to everyone** (all variable names and values are displayed in the URL). GET also has limits on the amount of information to send. The limitation is about 2000 characters. However, because the variables are displayed in the URL, it is possible to bookmark the page. This can be useful in some cases.

GET may be used for sending non-sensitive data.

**Note:** GET should NEVER be used for sending passwords or other sensitive information!

## **When to use POST?**

Information sent from a form with the POST method is **invisible to others** (all names/values are embedded within the body of the HTTP request) and has **no limits** on the amount of information to send.

Moreover POST supports advanced functionality such as support for multi-part binary input while uploading files to server.

However, because the variables are not displayed in the URL, it is not possible to bookmark the page.

|  |  |
| --- | --- |
| **Note** | **Developers prefer POST for sending form data.** |

Next; lets see how we can process PHP forms the secure way!

**PHP** This and the next chapters show how to use PHP to validate form data.

## **PHP Form Validation**

|  |  |
| --- | --- |
| **Note** | **Think SECURITY when processing PHP forms!**  These pages will show how to process PHP forms with security in mind. Proper validation of form data is important to protect your form from hackers and spammers! |

The HTML form we will be working at in these chapters, contains various input fields: required and optional text fields, radio buttons, and a submit button:

The validation rules for the form above are as follows:

|  |  |
| --- | --- |
| **Field** | **Validation Rules** |
| Name | Required. + Must only contain letters and whitespace |
| E-mail | Required. + Must contain a valid email address (with @ and .) |
| Website | Optional. If present, it must contain a valid URL |
| Comment | Optional. Multi-line input field (textarea) |
| Gender | Required. Must select one |

First we will look at the plain HTML code for the form:

## **Text Fields**

The name, email, and website fields are text input elements, and the comment field is a textarea. The HTML code looks like this:

Name: <input type="text" name="name">  
E-mail: <input type="text" name="email">  
Website: <input type="text" name="website">  
Comment: <textarea name="comment" rows="5" cols="40"></textarea>

## **Radio Buttons**

The gender fields are radio buttons and the HTML code looks like this:

Gender:  
<input type="radio" name="gender" value="female">Female  
<input type="radio" name="gender" value="male">Male

## **The Form Element**

The HTML code of the form looks like this:

<form method="post" action="<?php echo htmlspecialchars($\_SERVER["PHP\_SELF"]);?>">

When the form is submitted, the form data is sent with method="post".

|  |  |
| --- | --- |
| **Note** | **What is the $\_SERVER["PHP\_SELF"] variable?**  The $\_SERVER["PHP\_SELF"] is a super global variable that returns the filename of the currently executing script. |

So, the $\_SERVER["PHP\_SELF"] sends the submitted form data to the page itself, instead of jumping to a different page. This way, the user will get error messages on the same page as the form.

|  |  |
| --- | --- |
| **Note** | **What is the htmlspecialchars() function?**  The htmlspecialchars() function converts special characters to HTML entities. This means that it will replace HTML characters like < and > with &lt; and &gt;. This prevents attackers from exploiting the code by injecting HTML or Javascript code (Cross-site Scripting attacks) in forms. |

## **Big Note on PHP Form Security**

The $\_SERVER["PHP\_SELF"] variable can be used by hackers!

If PHP\_SELF is used in your page then a user can enter a slash (/) and then some Cross Site Scripting (XSS) commands to execute.

|  |  |
| --- | --- |
| **Note** | **Cross-site scripting (XSS) is a type of computer security vulnerability typically found in Web applications. XSS enables attackers to inject client-side script into Web pages viewed by other users.** |

Assume we have the following form in a page named "test\_form.php":

<form method="post" action="<?php echo $\_SERVER["PHP\_SELF"];?>">

Now, if a user enters the normal URL in the address bar like "http://www.example.com/test\_form.php", the above code will be translated to:

<form method="post" action="test\_form.php">

So far, so good.

However, consider that a user enters the following URL in the address bar:

http://www.example.com/test\_form.php/%22%3E%3Cscript%3Ealert('hacked')%3C/script%3E

In this case, the above code will be translated to:

<form method="post" action="test\_form.php/"><script>alert('hacked')</script>

This code adds a script tag and an alert command. And when the page loads, the JavaScript code will be executed (the user will see an alert box). This is just a simple and harmless example how the PHP\_SELF variable can be exploited.

Be aware of that **any JavaScript code can be added inside the <script> tag!**A hacker can redirect the user to a file on another server, and that file can hold malicious code that can alter the global variables or submit the form to another address to save the user data, for example.

## **How To Avoid $\_SERVER["PHP\_SELF"] Exploits?**

$\_SERVER["PHP\_SELF"] exploits can be avoided by using the htmlspecialchars() function.

The form code should look like this:

<form method="post" action="<?php echo htmlspecialchars($\_SERVER["PHP\_SELF"]);?>">

The htmlspecialchars() function converts special characters to HTML entities. Now if the user tries to exploit the PHP\_SELF variable, it will result in the following output:

<form method="post" action="test\_form.php/&quot;&gt;&lt;script&gt;alert('hacked')&lt;/script&gt;">

The exploit attempt fails, and no harm is done!

## **Validate Form Data With PHP**

The first thing we will do is to pass all variables through PHP's htmlspecialchars() function.

When we use the htmlspecialchars() function; then if a user tries to submit the following in a text field:

<script>location.href('http://www.hacked.com')</script>

- this would not be executed, because it would be saved as HTML escaped code, like this:

&lt;script&gt;location.href('http://www.hacked.com')&lt;/script&gt;

The code is now safe to be displayed on a page or inside an e-mail.

We will also do two more things when the user submits the form:

1. Strip unnecessary characters (extra space, tab, newline) from the user input data (with the PHP trim() function)
2. Remove backslashes (\) from the user input data (with the PHP stripslashes() function)

The next step is to create a function that will do all the checking for us (which is much more convenient than writing the same code over and over again).

We will name the function test\_input().

Now, we can check each $\_POST variable with the test\_input() function, and the script looks like this:

## **Example**

<?php  
// define variables and set to empty values  
$name = $email = $gender = $comment = $website = "";  
  
if ($\_SERVER["REQUEST\_METHOD"] == "POST") {  
  $name = test\_input($\_POST["name"]);  
  $email = test\_input($\_POST["email"]);  
  $website = test\_input($\_POST["website"]);  
  $comment = test\_input($\_POST["comment"]);  
  $gender = test\_input($\_POST["gender"]);  
}  
  
function test\_input($data) {  
  $data = trim($data);  
  $data = stripslashes($data);  
  $data = htmlspecialchars($data);  
  return $data;  
}  
?>

[**Run example »**](http://www.w3schools.com/php/showphp.asp?filename=demo_form_validation_escapechar)

Notice that at the start of the script, we check whether the form has been submitted using $\_SERVER["REQUEST\_METHOD"]. If the REQUEST\_METHOD is POST, then the form has been submitted - and it should be validated. If it has not been submitted, skip the validation and display a blank form.

However, in the example above, all input fields are optional. The script works fine even if the user does not enter any data.

The next step is to make input fields required and create error messages if needed.

# Form Validation

This and the next chapters show how to use PHP to validate form data.

## **PHP Form Validation**

|  |  |
| --- | --- |
| **Note** | **Think SECURITY when processing PHP forms!**  These pages will show how to process PHP forms with security in mind. Proper validation of form data is important to protect your form from hackers and spammers! |

The HTML form we will be working at in these chapters, contains various input fields: required and optional text fields, radio buttons, and a submit button:

The validation rules for the form above are as follows:

|  |  |
| --- | --- |
| **Field** | **Validation Rules** |
| Name | Required. + Must only contain letters and whitespace |
| E-mail | Required. + Must contain a valid email address (with @ and .) |
| Website | Optional. If present, it must contain a valid URL |
| Comment | Optional. Multi-line input field (textarea) |
| Gender | Required. Must select one |

First we will look at the plain HTML code for the form:

## **Text Fields**

The name, email, and website fields are text input elements, and the comment field is a textarea. The HTML code looks like this:

Name: <input type="text" name="name">  
E-mail: <input type="text" name="email">  
Website: <input type="text" name="website">  
Comment: <textarea name="comment" rows="5" cols="40"></textarea>

## **Radio Buttons**

The gender fields are radio buttons and the HTML code looks like this:

Gender:  
<input type="radio" name="gender" value="female">Female  
<input type="radio" name="gender" value="male">Male

## **The Form Element**

The HTML code of the form looks like this:

<form method="post" action="<?php echo htmlspecialchars($\_SERVER["PHP\_SELF"]);?>">

When the form is submitted, the form data is sent with method="post".

|  |  |
| --- | --- |
| **Note** | **What is the $\_SERVER["PHP\_SELF"] variable?**  The $\_SERVER["PHP\_SELF"] is a super global variable that returns the filename of the currently executing script. |

So, the $\_SERVER["PHP\_SELF"] sends the submitted form data to the page itself, instead of jumping to a different page. This way, the user will get error messages on the same page as the form.

|  |  |
| --- | --- |
| **Note** | **What is the htmlspecialchars() function?**  The htmlspecialchars() function converts special characters to HTML entities. This means that it will replace HTML characters like < and > with &lt; and &gt;. This prevents attackers from exploiting the code by injecting HTML or Javascript code (Cross-site Scripting attacks) in forms. |

## **Big Note on PHP Form Security**

The $\_SERVER["PHP\_SELF"] variable can be used by hackers!

If PHP\_SELF is used in your page then a user can enter a slash (/) and then some Cross Site Scripting (XSS) commands to execute.

|  |  |
| --- | --- |
| **Note** | **Cross-site scripting (XSS) is a type of computer security vulnerability typically found in Web applications. XSS enables attackers to inject client-side script into Web pages viewed by other users.** |

Assume we have the following form in a page named "test\_form.php":

<form method="post" action="<?php echo $\_SERVER["PHP\_SELF"];?>">

Now, if a user enters the normal URL in the address bar like "http://www.example.com/test\_form.php", the above code will be translated to:

<form method="post" action="test\_form.php">

So far, so good.

However, consider that a user enters the following URL in the address bar:

http://www.example.com/test\_form.php/%22%3E%3Cscript%3Ealert('hacked')%3C/script%3E

In this case, the above code will be translated to:

<form method="post" action="test\_form.php/"><script>alert('hacked')</script>

This code adds a script tag and an alert command. And when the page loads, the JavaScript code will be executed (the user will see an alert box). This is just a simple and harmless example how the PHP\_SELF variable can be exploited.

Be aware of that **any JavaScript code can be added inside the <script> tag!**A hacker can redirect the user to a file on another server, and that file can hold malicious code that can alter the global variables or submit the form to another address to save the user data, for example.

## **How To Avoid $\_SERVER["PHP\_SELF"] Exploits?**

$\_SERVER["PHP\_SELF"] exploits can be avoided by using the htmlspecialchars() function.

The form code should look like this:

<form method="post" action="<?php echo htmlspecialchars($\_SERVER["PHP\_SELF"]);?>">

The htmlspecialchars() function converts special characters to HTML entities. Now if the user tries to exploit the PHP\_SELF variable, it will result in the following output:

<form method="post" action="test\_form.php/&quot;&gt;&lt;script&gt;alert('hacked')&lt;/script&gt;">

The exploit attempt fails, and no harm is done!

## **Validate Form Data With PHP**

The first thing we will do is to pass all variables through PHP's htmlspecialchars() function.

When we use the htmlspecialchars() function; then if a user tries to submit the following in a text field:

<script>location.href('http://www.hacked.com')</script>

- this would not be executed, because it would be saved as HTML escaped code, like this:

&lt;script&gt;location.href('http://www.hacked.com')&lt;/script&gt;

The code is now safe to be displayed on a page or inside an e-mail.

We will also do two more things when the user submits the form:

1. Strip unnecessary characters (extra space, tab, newline) from the user input data (with the PHP trim() function)
2. Remove backslashes (\) from the user input data (with the PHP stripslashes() function)

The next step is to create a function that will do all the checking for us (which is much more convenient than writing the same code over and over again).

We will name the function test\_input().

Now, we can check each $\_POST variable with the test\_input() function, and the script looks like this:

## **Example**

<?php  
// define variables and set to empty values  
$name = $email = $gender = $comment = $website = "";  
  
if ($\_SERVER["REQUEST\_METHOD"] == "POST") {  
  $name = test\_input($\_POST["name"]);  
  $email = test\_input($\_POST["email"]);  
  $website = test\_input($\_POST["website"]);  
  $comment = test\_input($\_POST["comment"]);  
  $gender = test\_input($\_POST["gender"]);  
}  
  
function test\_input($data) {  
  $data = trim($data);  
  $data = stripslashes($data);  
  $data = htmlspecialchars($data);  
  return $data;  
}  
?>

[**Run example »**](http://www.w3schools.com/php/showphp.asp?filename=demo_form_validation_escapechar)

Notice that at the start of the script, we check whether the form has been submitted using $\_SERVER["REQUEST\_METHOD"]. If the REQUEST\_METHOD is POST, then the form has been submitted - and it should be validated. If it has not been submitted, skip the validation and display a blank form.

However, in the example above, all input fields are optional. The script works fine even if the user does not enter any data.

The next step is to make input fields required and create error messages if needed.