

PHP HACKS™

*Tips & Tools for Creating
Dynamic Web Sites*



O'REILLY®

Jack D. Herrington

**HACK**
#55**Fix the Double Submit Problem**

Use a transaction table in your database to fix the classic double submit problem.

I have a couple of pet peeves when it comes to bad web application design. One of the biggest is the wealth of bad code written to fix “double submits.” How often have you seen an e-commerce site that implores you, “Do not hit the submit button twice”?

This class problem results when a browser posts the contents of a web form to the server twice. However, if the user hits “submit” twice, this is exactly what the browser *should* do; it’s the server that needs to determine whether this is an error.

Figure 6-8 shows the double submit problem graphically. The browser sends two requests because the user clicks twice. The first submit is accepted, and before the HTML is returned, the second submit goes out. Then the first response comes in, followed by the second response.

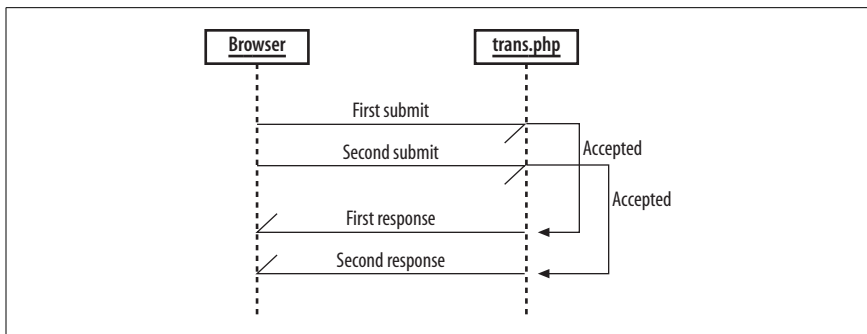


Figure 6-8. The double submit problem sequence diagram

Figure 6-9 illustrates a fix to the double submit problem; the first request stores a unique ID in the page being processed. That way, when the second request comes in with the same ID, the redundant transaction is denied.

The Code

Save the code in [Example 6-7](#) as *db.sql*.

Fix the Double Submit Problem

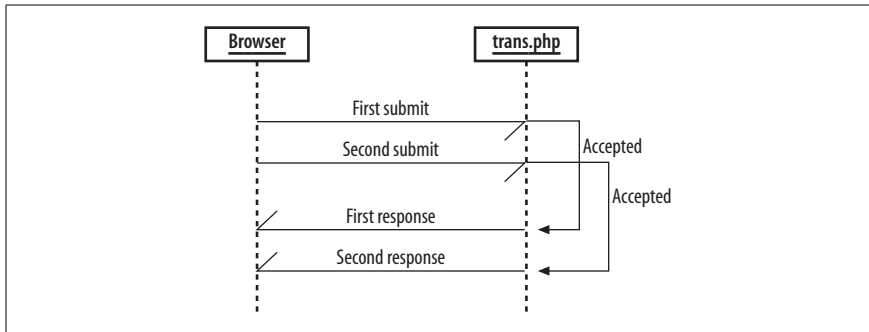


Figure 6-9. The double submit solution requires denying the second request

Example 6-7. The database code for the transaction checker

```

DROP TABLE IF EXISTS transcheck;
CREATE TABLE transcheck (
    transid TEXT,
    posted TIMESTAMP
);
  
```

Save the code in [Example 6-8](#) as *index.php*.

Example 6-8. The HTML form that has the transaction ID

```

<? require_once( "trans.php" ); ?>
<html>
<body>
<form action="handler.php" method="post">
<input type="hidden" name="transid" value="<?php echo( get_transid() ); ?>" />
Name: <input type="text" /><br/>
Amount: <input type="text" size="5" /><br/>
<input type="submit" />
</format>
  
```

Save the code in [Example 6-9](#) as *handler.php*.

Example 6-9. The code that receives the form data and checks the transaction

```

<? require_once( "trans.php" ); ?>
<html>
<body>
<?php if ( check_transid( $_POST["transid"] ) ) { ?>
This form has already been submitted.
<?php } else {
add_transid( $_POST["transid"] );
?>
Ok, you bought our marvelous product. Thanks!
<?php } ?>
  
```

Example 6-9. The code that receives the form data and checks the transaction (continued)

```
</body>
</html>
```

Save the code in [Example 6-10](#) as *trans.php*.

Example 6-10. The transaction checking library

```
<?php
require_once( "DB.php" );
$dns = 'mysql://root:password@localhost/transtest';
$db =& DB::Connect( $dns, array() );
if (PEAR::isError($db)) {
    die($db->getMessage());
}

function check_transid( $id )
{
    global $db;
    $res = $db->query( "SELECT COUNT(transid) FROM transcheck WHERE transid=?",
        array($id) );
    $res->fetchInto($row);
    return $row[0];
}

function add_transid( $id )
{
    global $db;
    $sth = $db->prepare( "INSERT INTO transcheck VALUES( ?, now() )" );
    $db->execute( $sth, array( $id ) );
}

function get_transid()
{
    $id = mt_rand();
    while( check_transid( $id ) ) { $id = mt_rand(); }
    return $id;
}
?>
```

Running the Hack

Upload the files to the server, and then use the `mysql` command to load the *db.sql* schema into your database:

```
mysql --user=myuser --password=mypassword mydb < db.sql
```

Next, navigate to the *index.php* page with your browser, and you will see the simple e-commerce form shown in [Figure 6-10](#).



Figure 6-10. The e-commerce form

Fill in some bogus data and click Submit. You should see the result shown in Figure 6-11, which shows a successful transaction. This is a good start, as it shows that we can successfully complete a transaction. Now we'll move on to denying redundant transactions.



Figure 6-11. A successful purchase

Click the Back button and click Submit again. You should see the result in Figure 6-12.

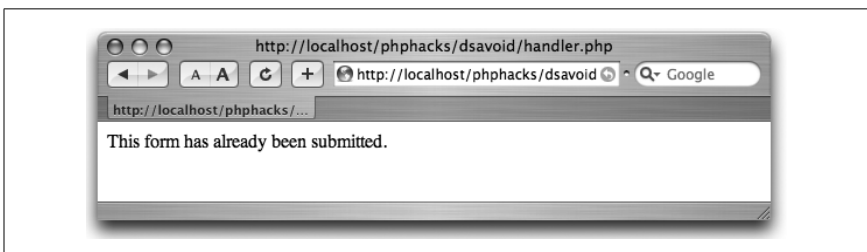


Figure 6-12. The result of a double submit

What happened is that `index.php` has requested a unique ID from the `trans.php` script. The `handler.php` script, which receives the form variables, first checks the ID to see whether it has been used already by calling the `check_transid()` function. If the ID has been used, the code should return the result shown in Figure 6-12.

If the ID is not in the database, we use the `add_transid()` function to add the ID to the database, and tell the user that the processing has been successful, as shown in [Figure 6-11](#).

The astute reader will note the race condition here. If another form submit comes in between the use of the `check_transid()` function and the call to the `add_transid()` function, you could get a double submit that *is* appropriate to process. If your database supports stored procedures, you can write a single transaction that will check to see whether the transaction has completed and then add the transaction to the completed list. This will avoid the race condition and ensure that you cannot have double submits.



At the time of this writing, MySQL did not support stored procedures, though it is in the feature request line for later releases.