Ben Brandhorst

SDEV 325 6380

Detecting Software Vulnerabilities

Professor Luke Donoho

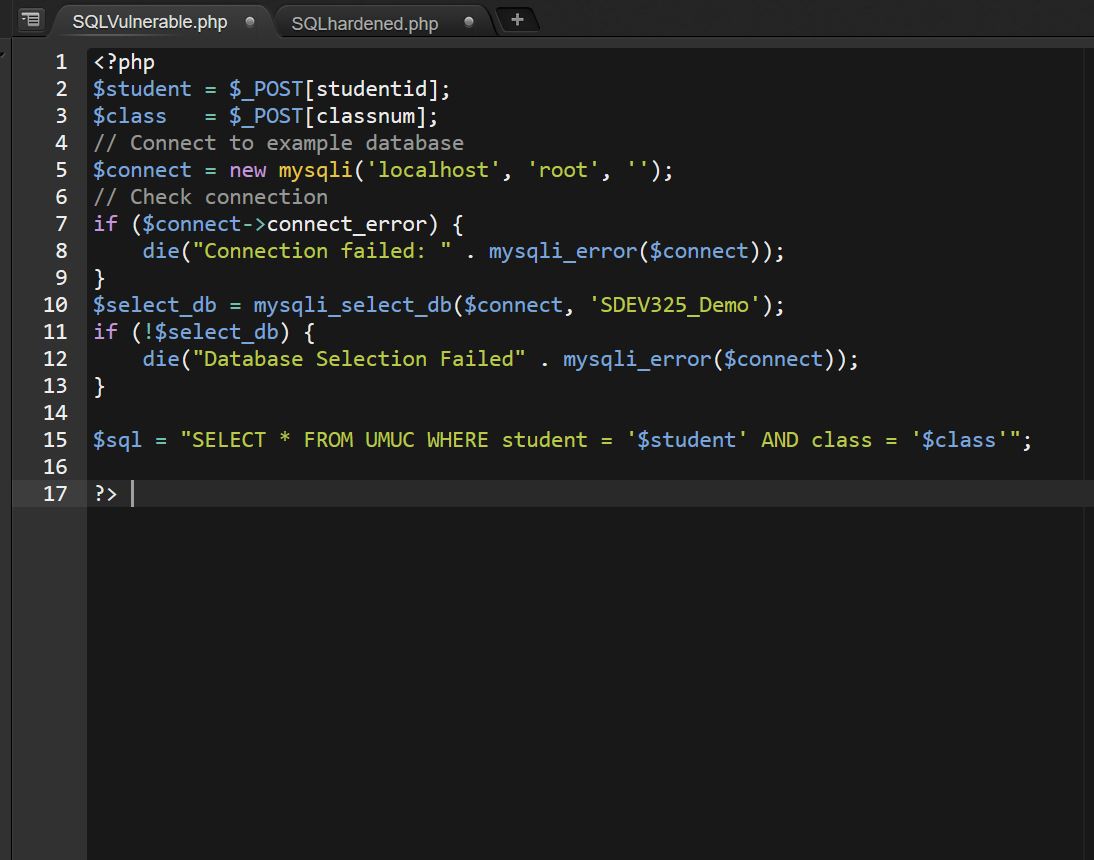
08/31/2019

**Demonstrating Insecure Interactions Between Components**

**Example #1**

**CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')**

**Vulnerable PHP Code**

****

The code in the example given is vulnerable to exploitation in a few different ways. The main vulnerability comes from accepting user generated input without sanitizing that input. In the above example **$student** and **$class** are both generated from a form that accepts keyboard input. The unsanitized input for those variables is then used in an SQL select statement and is vulnerable to injections.

A regular user might use input like **$student = ‘12345’** and **$class = ‘6380’** which would turn the SQL statement into: **$sql = “SELECT \* FROM SDEV325 WHERE Student = ‘12345’ AND Class = ‘6380’’;**

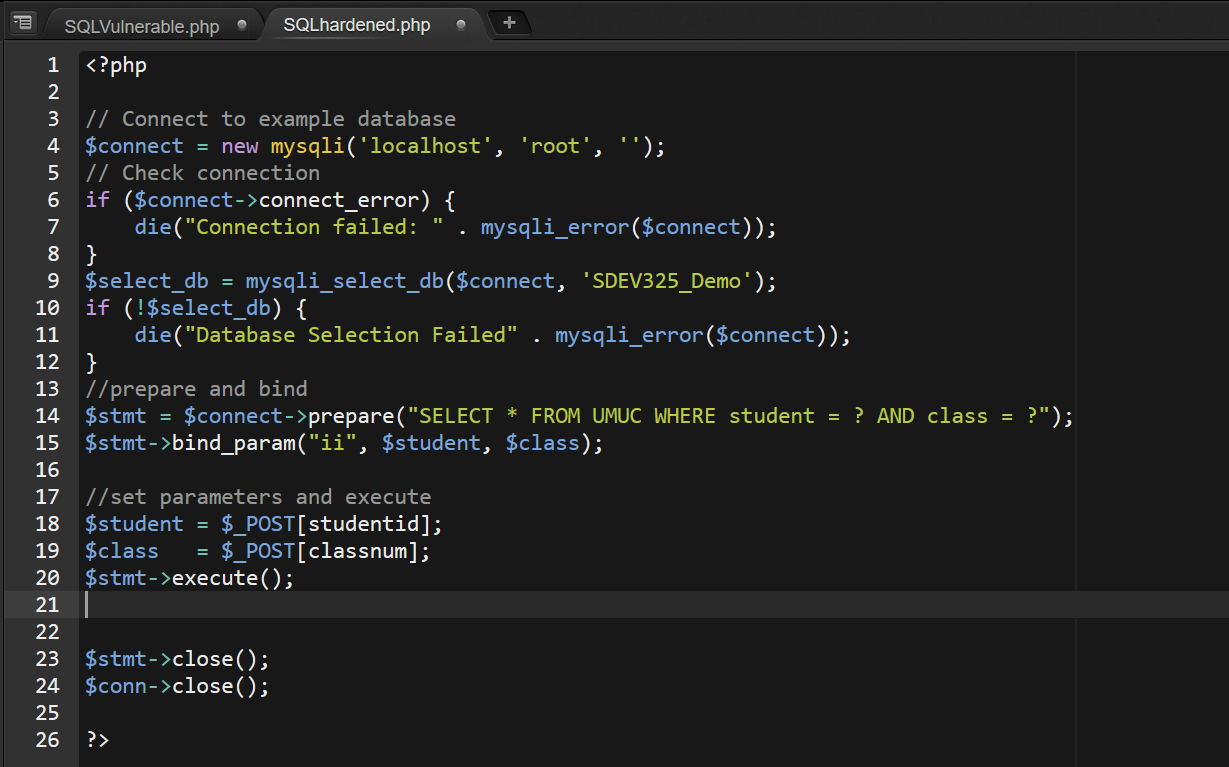
However users may also try something like: **$student = "1' OR 1 LIMIT 1; --'** and **$class = ‘any’** which would then turn the SQL statement into:

**$sql = “SELECT \* FROM SDEV325 WHERE Student = "1' OR 1 LIMIT 1; --" AND Class = ‘any’;**

The **--** input comments out the rest of the SQL statement and makes the Class input completely irrelevant. This will force the database to return a list of every student in the selected table.

Additionally, the example is vulnerable to Batched SQL statements that could include unwanted additional statements like **DROP TABLE SDEV325;**

**Hardened PHP Code**



A mitigation against the SQL Injection vulnerability is the use of the Prepared Statements with Bound Parameters. In the above example we use:

**$stmt = $connect->prepare("SELECT \* FROM UMUC WHERE student = ? AND class = ?");**

as a prepared statement where the question marks are substituted for the actual variable values.

The **bind\_param()** function then binds the parameters to the SQL query like:

**$stmt->bind\_param("ii", $student, $class);**

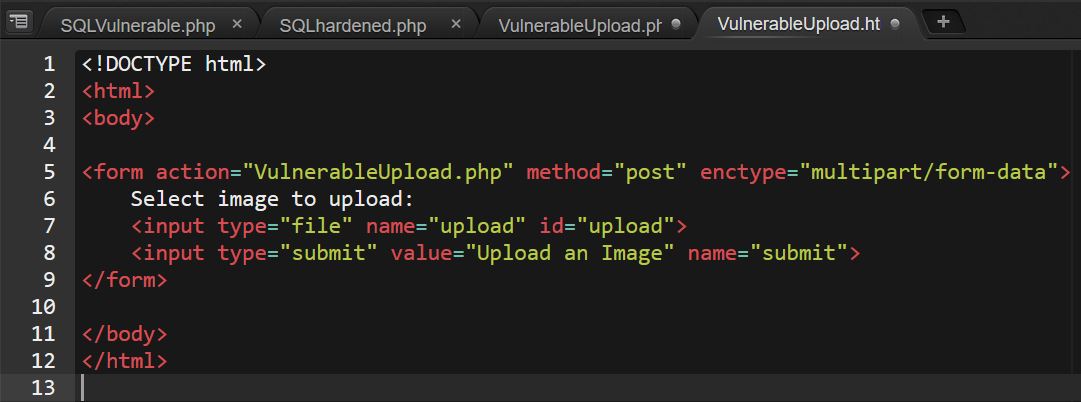
Here the **“ii”** indicates that both variable values should be integers. Once we identify the **$student** and **$class** values we then execute the statement using **$stmt->execute();**

This sanitizes user input and ensures that only variables of the integer type can be used in the SQL statement and prevents users for adding any additional input to the SQL statement.

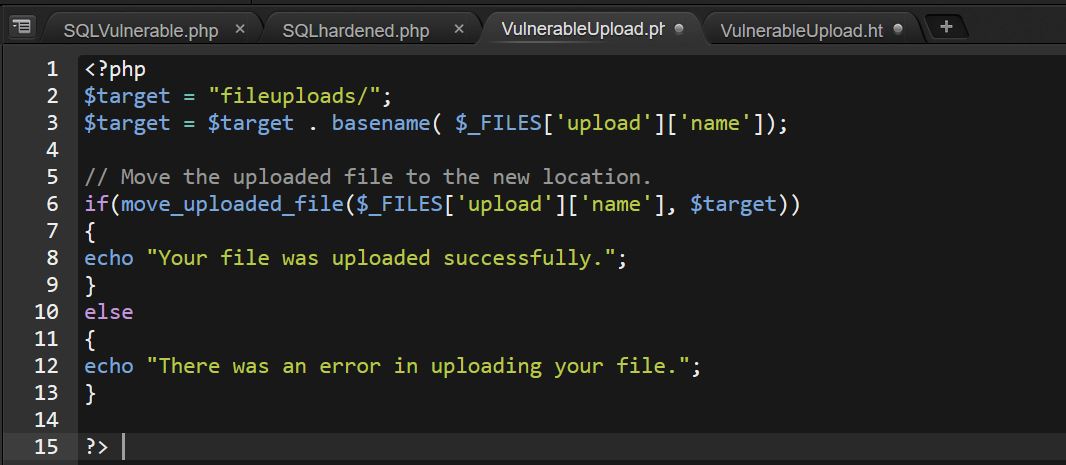
**Example #2**

**CWE-434: Unrestricted Upload of File with Dangerous Type**

**HTML Form for Uploading Files**

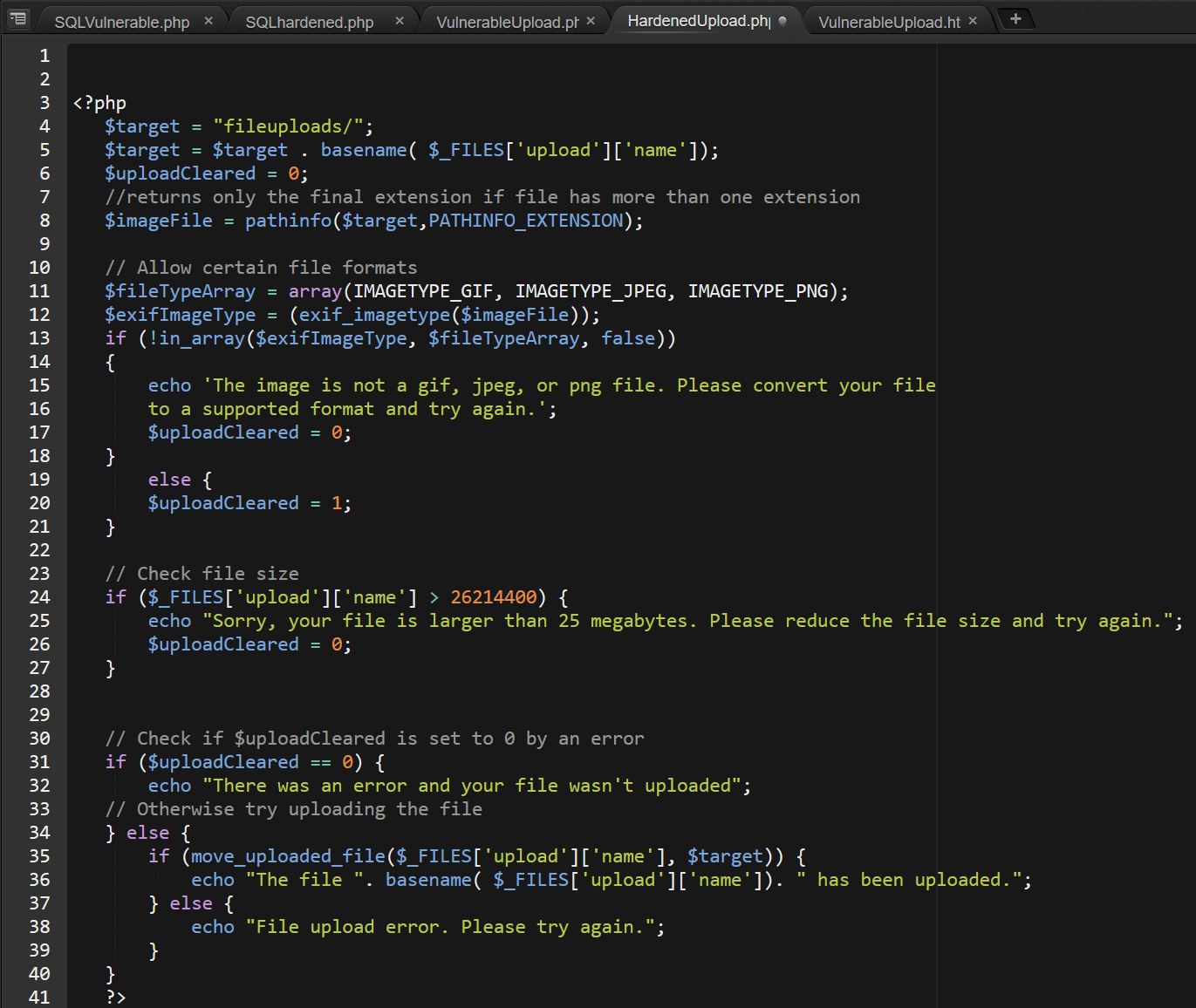
****

**Vulnerable PHP Code for Uploading Files**



The above example is vulnerable because there is no system in place to check the type of file being uploaded. Despite the HTML form indicating that users should upload pictures, it is possible to upload any type of file and have that file be of any size. This leaves the site vulnerable to users uploading malicious files that may execute unwanted code. For example: If the fileuploads/ directory is available in the web document root, a user could upload a php file which could include malicious code that gives access to terminal commands. From that upload users could gain access to files and directories they were not intended to have access to.

**Hardened PHP Code for Uploading Files**



Several steps were taken to help mitigate vulnerabilities.

**$imageFile = pathinfo($target,PATHINFO\_EXTENSION);**

Strips away any additional extensions in a file name and returns only the final extension. This can help mitigate against file uploads with multiple extensions.

**$fileTypeArray = array(IMAGETYPE\_GIF, IMAGETYPE\_JPEG, IMAGETYPE\_PNG);**

Stores the image types we want to accept to compare against when uploading files.

**$exifImageType = (exif\_imagetype($imageFile));**

Returns a one of several different image types to help guarantee that the file being uploaded is in fact an image we want to store and not a malicious file.

**if (!in\_array($exifImageType, $fileTypeArray, false))**

**{**

**echo 'The image is not a gif, jpeg, or png file. Please convert your file**

**to a supported format and try again.';**

**$uploadCleared = 0;**

Compares the values of **$exifImageType** and **$FileTypeArray** to see if the uploaded file type is one of the white listed file types. If the types do not match the upload is canceled and an error message is given.

**if ($\_FILES['upload']['name'] > 26214400) {**

**echo "Sorry, your file is larger than 25 megabytes. Please reduce the file size and try again.";**

**$uploadCleared = 0;**

Checks the size of the file being uploaded and ensures it’s less thant 25 megabytes. If the file is larger than the maximum the value of **$uploadCleared** is set to 0 which will stop the file upload.

**if ($uploadCleared == 0) {**

**echo "There was an error and your file wasn't uploaded";**

Finally, the above code checks **$uploadCleared** variable to confirm there was nothing preventing the upload.