PHP MySQL

PHP

- PHP: Hypertext Preprocessor
- PHP is a server scripting language, and a powerful tool for making dynamic and interactive Web pages.
- PHP is a widely-used, free, and efficient alternative to competitors such as Microsoft's ASP.
- PHP 7 is the latest stable release.

Reference: w3schools.com/php

EXAMPLE 1

```
<html>
<body>

<?php
$var="first";
echo "My $var PHP script!";
echo "<br>IIT Patna";
?>

</body>
</html>
```

My first PHP script!
IIT Patna

PHP Variables

- A variable starts with the $ sign, followed by the name of the variable
- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive
  ($age and $AGE are two different variables)

OUTPUT STATEMENT - ECHO

```
<html>
<body>
<?php
echo "<h2>PHP</h2>";
echo "This ", "checks", "with multiple parameters.";

$txt1 = "Learn PHP";
$txt2 = "DB LAB";
$x = 5;
$y = 4;

echo "<h3>" . $txt1 . "</h3>";
echo "Study PHP at " . $txt2 . "<br>";
echo $x + $y;
?>

</body>
</html>
```

PHP

This checkswith multiple parameters.

Learn PHP

Study PHP at DB LAB
9

OUTPUT STATEMENT- PRINT

```
<html>
<body>
<?php
print "<h2>PHP</h2>";
print "This cannot print multiple parameters.";

$txt1 = "Learn PHP";
$txt2 = "DB LAB";
$x = 5;
$y = 4;

print "<h3>" . $txt1 . "</h3>";
print "Study PHP at " . $txt2 . "<br>";
print $x + $y;
?>

</body>
</html>
```

PHP

This cannot print multiple parameters.

Learn PHP

Study PHP at DB LAB
9

Array

```
<html>
<body>
<?php
print "<h2>Testing Array</h2>";

$misc = array("Alice",3,10.6);
var_dump($misc);
echo "<p>The elements are " . $misc[0] . ", " . $misc[1] . " and " .
  $misc[2] . ".";
?>

</body>
</html>
```

Testing Array

array(3) { [0]=> string(5) "Alice" [1]=> int(3) [2]=> float(10.6) }

The elements are Alice, 3 and 10.6.

Array- loop through indexed array

```
<html>
<body>
<?php
$fruits = array("Mango", "Banana", "Orange","Apple");
$arrlength = count($fruits);

for($x = 0; $x < $arrlength; $x++) {
    echo $fruits[$x];
    echo "<br>";
}
?>

</body>
</html>
```

Mango
Banana
Orange
Apple

Loop through associative array using foreach

```
<html>
<body>
<?php
$marks = array("Alice"=>"65", "Bob"=>"77", "Carol"=>"78","Joe"=>"53");

foreach($marks as $x => $x_value) {
    echo "Key=" . $x . ", Value=" . $x_value;
    echo "<br>";
}
?>

</body>
</html>
```

Key=Alice, Value=65
Key=Bob, Value=77
Key=Carol, Value=78
Key=Joe, Value=53

Sorting indexed array

```
<html>
<body>

<?php
$friends = array("Carol", "Alice", "Bob");
 echo "<h3>Ascending Order</h3>";
sort($friends);

$clength = count($friends);
for($x = 0; $x < $clength; $x++) {
    echo $friends[$x];
    echo "<br>";
}
echo "<h3>Descending Order</h3>";
rsort($friends);

$clength = count($friends);
for($x = 0; $x < $clength; $x++) {
    echo $friends[$x];
    echo "<br>";
}
?>

</body>
</html>
```

Ascending Order

Alice
Bob
Carol

Descending Order

Carol
Bob
Alice

Sort associative array

```
<html>
<body>

<?php
$marks = array("Bob"=>"40", "Carol"=>"37", "Alice"=>"43");
echo "<h3>Sort based on value</h3>";
  asort($marks);

foreach($marks as $x => $x_value) {
    echo "Key=" . $x . ", Value=" . $x_value;
    echo "<br>";
}
  echo "<h3>Sort based on key</h3>";

ksort($marks);

foreach($marks as $x => $x_value) {
    echo "Key=" . $x . ", Value=" . $x_value;
    echo "<br>";
}
?>

</body>
</html>
```

Sort based on value

Key=Carol, Value=37
Key=Bob, Value=40
Key=Alice, Value=43

Sort based on key

Key=Alice, Value=43
Key=Bob, Value=40
Key=Carol, Value=37

Sort associative array descending

```
<html>
<body>

<?php
$marks = array("Bob"=>"40", "Carol"=>"37", "Alice"=>"43");
echo "<h3>Sort based on value descending</h3>";
  arsort($marks);

foreach($marks as $x => $x_value) {
    echo "Key=" . $x . ", Value=" . $x_value;
    echo "<br>";
}
  echo "<h3>Sort based on key descending</h3>";

krsort($marks);

foreach($marks as $x => $x_value) {
    echo "Key=" . $x . ", Value=" . $x_value;
    echo "<br>";
}
?>

</body>
</html>
```

Sort based on value descending

Key=Alice, Value=43
Key=Bob, Value=40
Key=Carol, Value=37

Sort based on key descending

Key=Carol, Value=37
Key=Bob, Value=40
Key=Alice, Value=43

CONSTANTS

```
<html>
<body>

<?php
define("PI", 3.124);
echo PI;

define("MSG", "Welcome to the DB Class", false);
echo "<br>" . MSG;

?>

</body>
</html>
```

3.124
Welcome to the DB Class

If statement

```
<html>
<body>

<?php
$t = date("H");
echo "<p>The hour (of the server) is " . $t;

if ($t < "10") {
    echo "<p>Have a good morning!";
} elseif ($t < "20") {
    echo "<p>Have a good day!";
} else {
    echo "<p>Have a good night!";
}
?>

</body>
</html>
```

The hour (of the server) is 00

Have a good morning!

FUNCTION

```
<html>
<body>

<?php
function familyName($fname) {
    echo "$fname Kumar.<br>";
}

familyName("Sumit");
familyName("Rahul");
familyName("Manish");

?>

</body>
</html>
```

Sumit Kumar.
Rahul Kumar.
Manish Kumar.

SUPERGLOBAL

- Several predefined variables in PHP 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.
- The PHP superglobal variables are:
  - $GLOBALS
  - $_SERVER
  - $_REQUEST
  - $_POST
  - $_GET
  - $_FILES
  - $_ENV
  - $_COOKIE
  - $_SESSION

HTML FORM: POST METHOD

```
<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>
```

Name:
E-mail:
Submit

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.

A SAMPLE WELCOME.PHP FILE USING POST METHOD

```
<html>
<body>

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

</body>
</html>
```

HTML FORM: GET METHOD

```html
<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>
```

Name:
E-mail:
Submit

A SAMPLE WELCOME.PHP FILE USING GET METHOD

```
<html>
<body>

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

</body>
</html>
```

GET VS POST

- Both GET and POST create an array (e.g. array(key1 => value1, 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

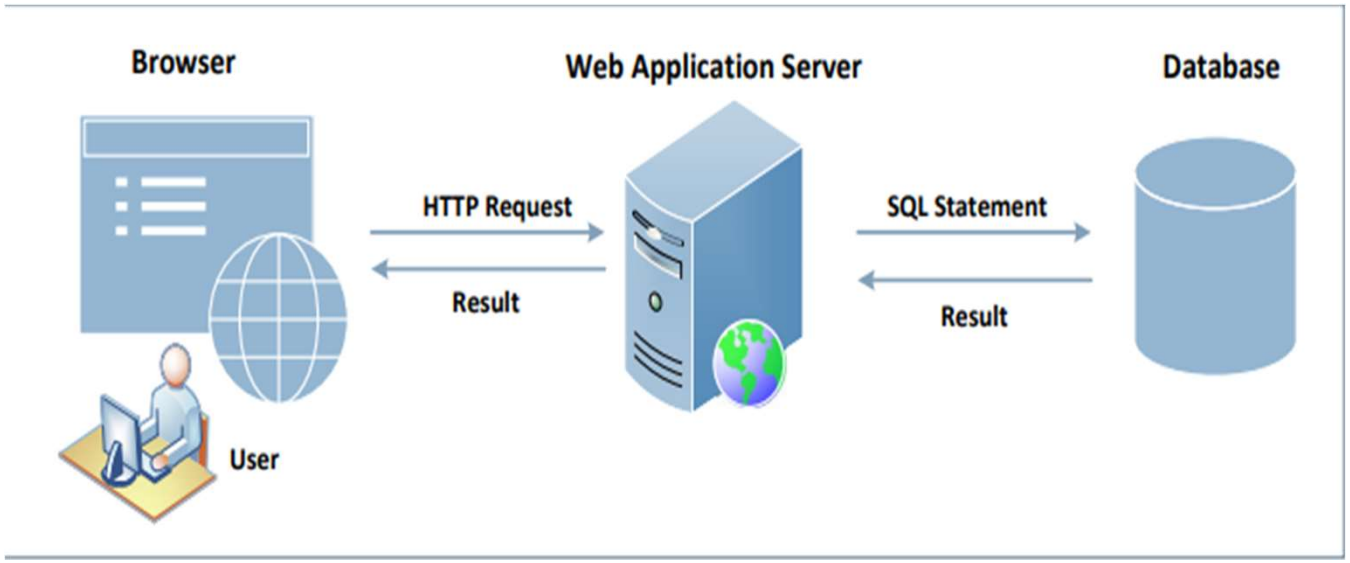
- Information sent from a form with the GET method is **visible to everyone** (all variable names and values are displayed in the URL).
- GET also has limits on the amount of information to send. The limitation is about 2000 characters. However, because the variables are displayed in the URL, it is possible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

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

Interacting with Database in Web Application

- A typical web application consists of three major components:



- SQL Injection attack is a potential attack in such context and can cause damage to the database. As we notice in the figure, the users do not directly interact with the database but through a web server. If this channel is not implemented properly, malicious users can attack the database.

Getting Data from User

- This example shows a form where users can type their data. Once the submit button is clicked, an HTTP request will be sent out with the data attached

EID EID5000
Password paswd123
Submit

- The HTML source of the above form is given below:

```
<form action="getdata.php" method="get">
  EID:      <input type="text" name="EID"><br>
  Password: <input type="text" name="Password"><br>
            <input type="submit" value="Submit">
</form>
```

- Request generated is:

```
http://www.example.com/getdata.php?EID=EID5000&Password=paswd123
```

Getting Data from User

- The request shown is an HTTP GET request, because the method field in the HTML code specified the get type
- In GET requests, parameters are attached after the question mark in the URL
- Each parameter has a name=value pair and are separated by "&"
- In the case of HTTPS, the format would be similar but the data will be encrypted
- Once this request reached the target PHP script the parameters inside the HTTP request will be saved to an array $_GET or $_POST. The following example shows a PHP script getting data from a GET request

```
<?php
   $eid = $_GET['EID'];
   $pwd = $_GET['Password'];
   echo "EID: $eid --- Password: $pwd\n";
?>
```

Connecting to MySQL Database

- PHP program connects to the database server before conducting query on database.
- The code shown below uses new mysqli(…) along with its 4 arguments to create the database connection.

```
function getDB() {
   $dbhost="localhost";
   $dbuser="root";
   $dbpass="seedubuntu";
   $dbname="dbtest";

   // Create a DB connection
   $conn = new mysqli($dbhost, $dbuser, $dbpass, $dbname);
   if ($conn->connect_error) {
         die("Connection failed: " . $conn->connect_error . "\n");
   }
   return $conn;
}
```

How Web Applications Interact with Database

- Construct the query string and then send it to the database for execution.
- The channel between user and database creates a new attack surface for the database.

```
/* getdata.php */
<?php
   $eid = $_GET['EID'];
   $pwd = $_GET['Password'];

   $conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");
   $sql = "SELECT Name, Salary, SSN
           FROM employee
           WHERE eid= '$eid' and password='$pwd'";

   $result = $conn->query($sql);
   if ($result) {
      // Print out the result
      while ($row = $result->fetch_assoc()) {
        printf ("Name: %s -- Salary: %s -- SSN: %s\n",
                $row["Name"], $row["Salary"], $row['SSN']);
      }
      $result->free();
   }
   $conn->close();
?>
```

Constructing SQL statement

MySQL: Comments

MySQL supports three comment styles

- Text from the # character to the end of line is treated as a comment
- Text from the "--" to the end of line is treated as a comment.
- Similar to C language, text between /* and */ is treated as a comment

```
mysql> SELECT * FROM employee;    # Comment to the end of line
mysql> SELECT * FROM employee;    -- Comment to the end of line
mysql> SELECT * FROM /* In-line comment */ employee;
```

Launching SQL Injection Attacks

- Everything provided by user will become part of the SQL statement. Is it possible for a user to change the meaning of the SQL statement?
- The intention of the web app developer by the following is for the user to provide some data for the blank areas.

```
SELECT Name, Salary, SSN
FROM employee
WHERE eid=' [                ] ' and password=' [                ] '
```

- Assume that a user inputs a random string in the password entry and types "EID5002'#" in the eid entry. The SQL statement will become the following

```
SELECT Name, Salary, SSN
FROM employee
WHERE eid= 'EID5002' #' and password='xyz'
```

Launching SQL Injection Attacks

- Everything from the # sign to the end of line is considered as comment. The SQL statement will be equivalent to the following:

```
SELECT Name, Salary, SSN
FROM employee
WHERE eid= 'EID5002'
```

- The above statement will return the name, salary and SSN of the employee whose EID is EID5002 even though the user doesn't know the employee's password. This is security breach.
- Let's see if a user can get all the records from the database assuming that we don't know all the EID's in the database.
- We need to create a predicate for WHERE clause so that it is true for all records

```
SELECT Name, Salary, SSN
FROM employee
WHERE eid= 'a' OR 1=1
```

Launching SQL Injection Attacks using cURL

- More convenient to use a command-line tool to launch attacks.
- Easier to automate attacks without a graphic user interface.
- Using cURL, we can send out a form from a command-line, instead of

```
% curl 'www.example.com/getdata.php?EID=a' OR 1=1 #&Password='
```

- The above command will not work. In an HTTP request, special characters are in the attached data needs to be encoded or they maybe mis-interpreted.
- In the above URL we need to encode the apostrophe (%27), whitespace (%20) and the # sign (%23) and the resulting cURL command is as shown below:

```
% curl 'www.example.com/getdata.php?EID=a%27%20
                                    OR%201=1%20%23&Password='
Name: Alice -- Salary: 80000 -- SSN: 555-55-5555<br>
Name: Bob -- Salary: 82000 -- SSN: 555-66-5555<br>
Name: Charlie -- Salary: 80000 -- SSN: 555-77-5555<br>
Name: David -- Salary: 80000 -- SSN: 555-88-5555<br>
```

Modify Database

- If the statement is UPDATE or INSERT INTO, we will have chance to change the database.
- Consider the form created for changing passwords. It asks users to fill in three pieces of information- EID, old password and new password.
- When Submit button is clicked, an HTTP POST request will be sent to the server-side script `changepassword.php`, which uses an UPDATE statement to change the user's password.

EID	EID5000
Old Password	paswd123
New Password	paswd456

Submit

```
/* changepasswd.php */
<?php
   $eid = $_POST['EID'];
   $oldpwd = $_POST['OldPassword'];
   $newpwd = $_POST['NewPassword'];

   $conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");
   $sql = "UPDATE employee
           SET password='$newpwd'
           WHERE eid= '$eid' and password='$oldpwd'";

   $result = $conn->query($sql);
   $conn->close();
?>
```

Modify Database

- Let us assume that Alice (EID5000) is not satisfied with the salary she gets. She would like to increase her own salary using the SQL injection vulnerability. She would type her own EID and old password. The following will be typed into the "New Password" box :

New Password	paswd456', salary=100000 #

- By typing the above string in "New Password" box, we get the UPDATE statement to set one more attribute for us, the salary attribute. The

```
UPDATE employee
SET password='paswd456', salary=100000 #'
WHERE eid= 'EID5000' and password='paswd123'";
```

- What if Alice doesn't like Bob and would like to reduce Bob's salary to 0, but she only knows Bob's EID (eid5001), not his password. How can she execute the attack?

EID	EID5001' #
Old Password	anything
New Password	paswd456', salary=0 #

Multiple SQL Statements

- Damages that can be caused are bounded because we cannot change everything in the existing SQL statement.
- It will be more dangerous if we can cause the database to execute an arbitrary SQL statement.
- To append a new SQL statement "DROP DATABASE dbtest" to the existing SQL statement to delete the entire dbtest database, we can type the following in the EID box

EID | a'; DROP DATABASE dbtest; #

- The resulting SQL statement is equivalent to the following, where we have successfully appended a new SQL statement to the existing SQL statement string.

```
SELECT Name, Salary, SSN
FROM employee
WHERE eid= 'a'; DROP DATABASE dbtest;
```

- The above attack doesn't work against MySQL, because in PHP's mysqli extension, the mysqli::query() API doesn't allow multiple queries to run in the database server.

Multiple SQL Statements

- The code below tries to execute two SQL statements using the $mysqli-

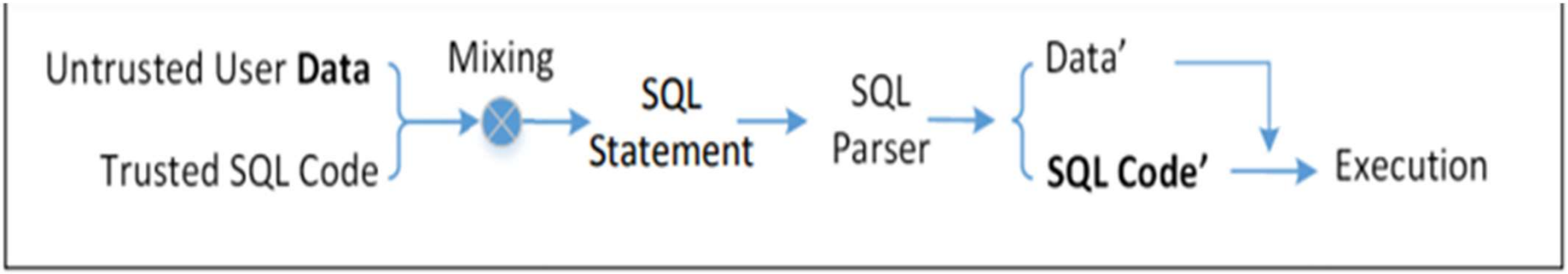
```
/* testmulti_sql.php */
<?php
$mysqli = new mysqli("localhost", "root", "seedubuntu", "dbtest");
$res    = $mysqli->query("SELECT 1; DROP DATABASE dbtest");
if (!$res) {
  echo "Error executing query: (" .
        $mysqli->errno . ") " . $mysqli->error;
}
?>
```

- When we run the code, we get the following error message:

```
$ php testmulti_sql.php
Error executing query: (1064) You have an error in your SQL syntax;
    check the manual that corresponds to your MySQL server version
    for the right syntax to use near 'DROP DATABASE dbtest' at line 1
```

- If we do want to run multiple SQL statements, we can use $mysqli -> multi_query(). [not recommended]

The Fundamental Cause



Mixing data and code together is the primary cause of SQL Injection attack

Countermeasures: Filtering and Encoding Data

- Before mixing user-provided data with code, inspect the data. Filter out any character that may be interpreted as code.
- Special characters are commonly used in SQL Injection attacks. To get rid of them, encode them.
- Encoding a special character tells parser to treat the encoded character as data and not as code. This can be seen in the following example

```
Before encoding:   aaa' OR 1=1 #
After encoding:    aaa\' OR 1=1 #
```

- PHP's mysqli extension has a built-in method called mysqli::real_escape_string(). It can be used to encode the characters that have special meanings in SQL. The following code snippet shows how to use this API.

```
/* getdata_encoding.php */
<?php
   $conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");
   $eid = $mysqli->real_escape_string($_GET['EID']);          ①
   $pwd = $mysqli->real_escape_string($_GET['Password'];      ②
   $sql = "SELECT Name, Salary, SSN
           FROM employee
           WHERE eid= '$eid' and password='$pwd'";
?>
```

Countermeasures: Prepared Statement

- **Fundament cause** of SQL injection: mixing data and code
- **Fundament solution**: separate data and code.
- **Main Idea:** Sending code and data in separate channels to the database server. This way the database server knows not to retrieve any code from the data channel.
- **How:** using **prepared statement**
- **Prepared Statement:** It is an optimized feature that provides improved performance if the same or similar SQL statement needs to be executed repeatedly. Using prepared statements, we send an SQL statement template to the database, with certain values called parameters left unspecified. The database parses, compiles and performs query optimization on the SQL statement template and stores the result without executing it. We later bind data to the prepared statement

Countermeasures: Prepared Statement

```
$conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");
$sql = "SELECT Name, Salary, SSN
        FROM employee
        WHERE eid= '$eid' and password='$pwd'";
$result = $conn->query($sql);
```

The vulnerable version: code and data are mixed together.

Using prepared statements, we separate code and data.

```
$conn = new mysqli("localhost", "root", "seedubuntu", "dbtest");
$sql = "SELECT Name, Salary, SSN
        FROM employee
        WHERE eid= ? and password=?";                ①

if ($stmt = $conn->prepare($sql)) {                  ②
   $stmt->bind_param("ss", $eid, $pwd);              ③
   $stmt->execute();                                 ④

   $stmt->bind_result($name, $salary, $ssn);         ⑤
   while ($stmt->fetch()) {                          ⑥
      printf ("%s %s %s\n", $name, $salary, $ssn);
   }
}
```

Send code

Send data

Start execution

Why Are Prepared Statements Secure?

- Trusted code is sent via a code channel.
- Untrusted user-provided data is sent via data channel.
- Database clearly knows the boundary between code and data.
- Data received from the data channel is not parsed.
- Attacker can hide code in data, but the code will never be treated as code, so it will never be attacked.

Account No

Password

Type

Balance

Submit

Account No

Old Password

New Password

Submit