WEB SYSTEM
Name: John Euben B. Lopez

Scenario 1 — Using $_POST Instead of $_GET

✔Fixed Code

```
<?php

$conn = mysqli_connect("localhost","root","","class_db");

$id = $_GET['id'];

$sql = "SELECT * FROM students WHERE student_id = $id";
$res = mysqli_query($conn, $sql);
$r = mysqli_fetch_assoc($res);

echo $r['first_name'];
 ?>
```

The value was meant to come from the URL, so the correct superglobal is ***$_GET***. Using $_POST would not capture the parameter, causing the query to fail.

Scenario 2 — Missing Quotes in SQL

✔Fixed Code

```
<?php
$conn = mysqli_connect("localhost","root","","class_db");

$fname = $_POST['fname'];
$sql = "SELECT * FROM students WHERE first_name = '$fname'"; $res = mysqli_query($conn,
$sql);
?>
```

Without them, MySQL treats the value as invalid syntax.

Scenario 3 — SQL Injection Vulnerability

✔Fixed Code

```
<?php
$conn = mysqli_connect("localhost","root","","class_db");
$age = $_GET['age'];
$stmt = $conn->prepare("SELECT * FROM students WHERE age = ?"); $stmt->bind_param("i",
$age);
$stmt->execute();
?>
```

Prepared statements prevent attackers from injecting SQL commands into your query, making the script more secure.

Scenario 4 Validate Empty POST Field

✔Fixed Code

```
<?php
$conn = mysqli_connect("localhost","root","","class_db");
if (!empty($_POST['fname']) && !empty($_POST['lname'])) {
$first = $_POST['fname'];
$last = $_POST['lname'];
$sql = "INSERT INTO students (first_name,last_name) VALUES ('$first','$last')";
mysqli_query($conn,
$sql); echo "Inserted!"; }
else { echo "Both fields are required.";
}
?>
```

Before saving a record, it’s important to verify that all required fields contain values to avoid incomplete database entries.

Scenario 5 — Wrong POST Key Name

✔Fixed Code

```
<?php
$conn = mysqli_connect("localhost","root","","class_db");

$email = $_POST['email'];

$sql = "SELECT * FROM students WHERE email='$email'";
```

```
$res = mysqli_query($conn, $sql);
?>
```

The form key was misspelled originally. Using the wrong key results in an undefined index error and prevents the script from getting the correct input.

Scenario 6 — Unsafe DELETE Using Get

✔Fixed Code

```
<?php
$conn = mysqli_connect("localhost","root","","class_db");


$id = intval($_GET['id']);

$sql = "DELETE FROM students WHERE student_id = $id";

mysqli_query($conn, $sql);
?>
```

intval() ensures the input is strictly a number, preventing malicious values like “1 OR 1=1” from being executed.

Scenario 7 — Query Fails but Script Continues*

✔Fixed Code

```
<?php
$conn = mysqli_connect("localhost","root","","class_db");

$id = $_POST['id'];
$email = $_POST['email'];

$sql = "UPDATE students SET email='$email' WHERE
student_id=$id";
if (mysqli_query($conn, $sql)) {
echo "Updated!";
} else {
echo "Update failed."; }
?>
```

Adding a condition allows the script to notify the user if something goes wrong, rather than always showing a success message.

Scenario 8 — Missing Loop for Fetching Multiple Records

✔Fixed Code

```
<?php
$conn = mysqli_connect("localhost","root","","class_db");

$res = mysqli_query($conn, "SELECT * FROM students");

while ($row = mysqli_fetch_assoc($res)) {
echo $row['email'] . "<br>";
}
?>
```

mysqli_fetch_assoc() retrieves a single row. A loop is required to display multiple rows from the result set

Scenario 9 — Using POST but Link Sends GET

✔Fixed Code

```
<?php
$id = $_GET['id'];
?>

<a href="view.php?id=3">View Student</a>
```

Hyperlinks will always use ***GET***, so the script should expect $_GET instead of $_POST for incoming values

Scenario 10 — Wrong Variable in SQL

✔Fixed Code

```
<?php
$age = $_POST['age'];
$sql = "SELECT * FROM students WHERE age = $age";
?>
```

Using the wrong variable leads to undefined variable errors. The corrected version now references the actual POST input.

Scenario 11 —Mismatched Method

✔Fixed Code ***HTML***

```
<form method="GET" action="save.php">
<input name="email">
</form>
```

PHP

```
<?php $email = $_GET['email'];
?>
```

The PHP script must match the method defined in the form. If the form sends GET data, the script must read from $_GET.

Scenario 12 — Numeric GET Used Inside Quotes

✔Fixed Code

```
<?php
$id = $_GET['id'];
$sql = "SELECT * FROM students WHERE id = $id";
?>
```

values do not require quotes in SQL. Removing them avoids unnecessary type casting***Scenario**

13 — Missing WHERE Clause in UPDATE

✔Fixed Code

```
<?php
$conn = mysqli_connect("localhost","root","","class_db");


$newEmail = $_POST['email'];
$id = $_POST['id'];
$sql = "UPDATE students SET email='$newEmail' WHERE
student_id=$id";

mysqli_query($conn, $sql);
?>
```

Without a WHERE clause, every row in the table gets updated. Adding a condition ensures only the intended record changes

Scenario 14 — Incorrect POST Array Usage

✔️Fixed Code

```
<?php
$data = $_POST;
$sql = "INSERT INTO students (first_name, last_name, email)
  VALUES ('{$data['first_name']}', '{$data['last_name']}',
'{$data['email']}')";
?>
```

Array values must be wrapped correctly inside the SQL string. Missing quotation marks will break the query

Scenario 15 — Unsafe Page Number in LIMIT

✔️Fixed Code

```
<?php
$page = $_GET['page'];

$page = intval($page);

if ($page < 0) {
$page = 0;
}

$limit = 5;

$offset = $page * $limit;

$sql = "SELECT * FROM students LIMIT $offset, $limit";
?>
```

Sanitizing the page number prevents negative offsets and avoids invalid SQL commands during pagination.