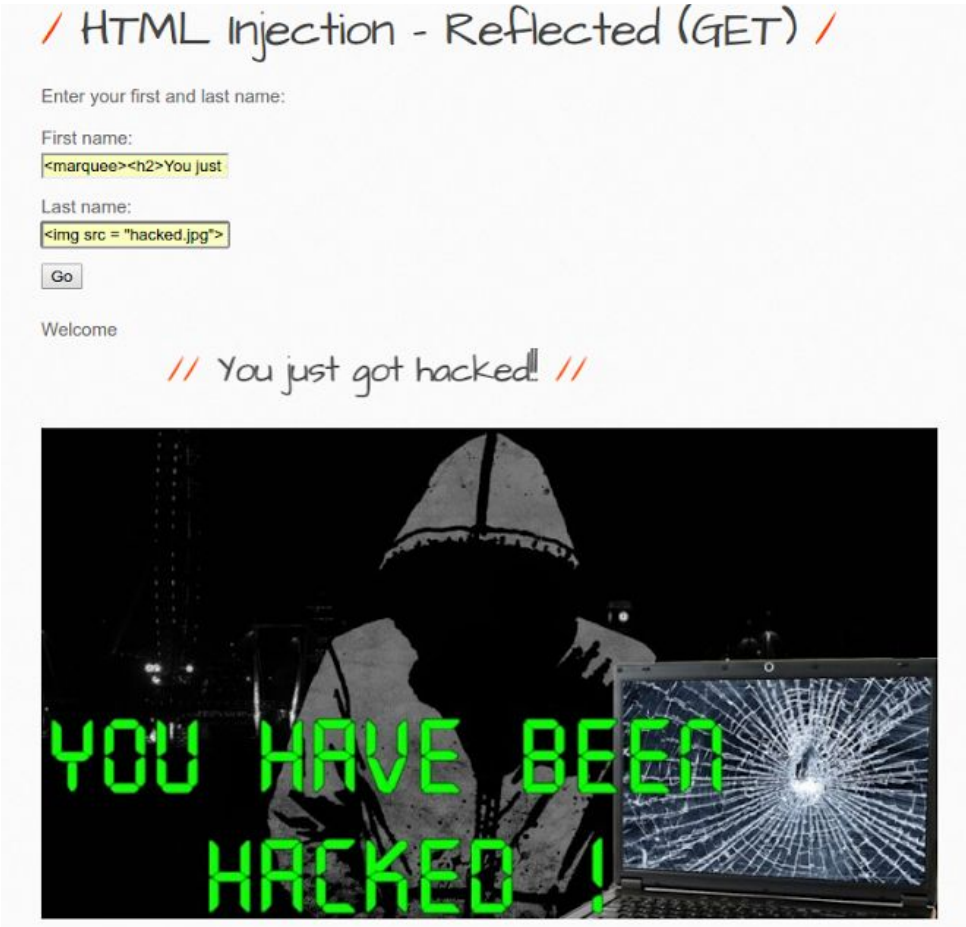
Systems and Network Security (Assignment 3)

by Nagaraj Poti (20162010)

Vulnerability 1:

- **Vulnerability type** : Injection
- **Vulnerability scope** : Main site
- **Title** : HTML INJECTION - REFLECTED GET
- **URL** : http://localhost/sns_demoapp/htmli_get.php?firstname=%3Cmarquee%3E%3Ch1%3EYou+just+got+hacked%21%21%3C%2Fh1%3E%3C%2Fmarquee%3E&lastname=%3Ch2%3E+Fix+it+soon+%3C%2Fh2%3E&form=submit
- **Description** : The flaw here is that HTML text that is being passed as GET parameters to the server is being stored in its raw format, without applying HTML encoding to prevent HTML special characters from being interpreted by the browser. This encoding must be done at the server side while storing the parameters.
- **Impact** : Severe
- **Proof of concept** :
  i. Navigate to the URL : http://localhost/sns_demoapp/htmli_get.php
  ii. At the First Name and Last Name input form boxes, enter some html code such as <marquee><h2>You just got hacked!!</h2></marquee> and <img src="hacked.jpg">
  iii. Submit the form
  iv. The resulting screen will have html text embedded unintentionally at the bottom of the page

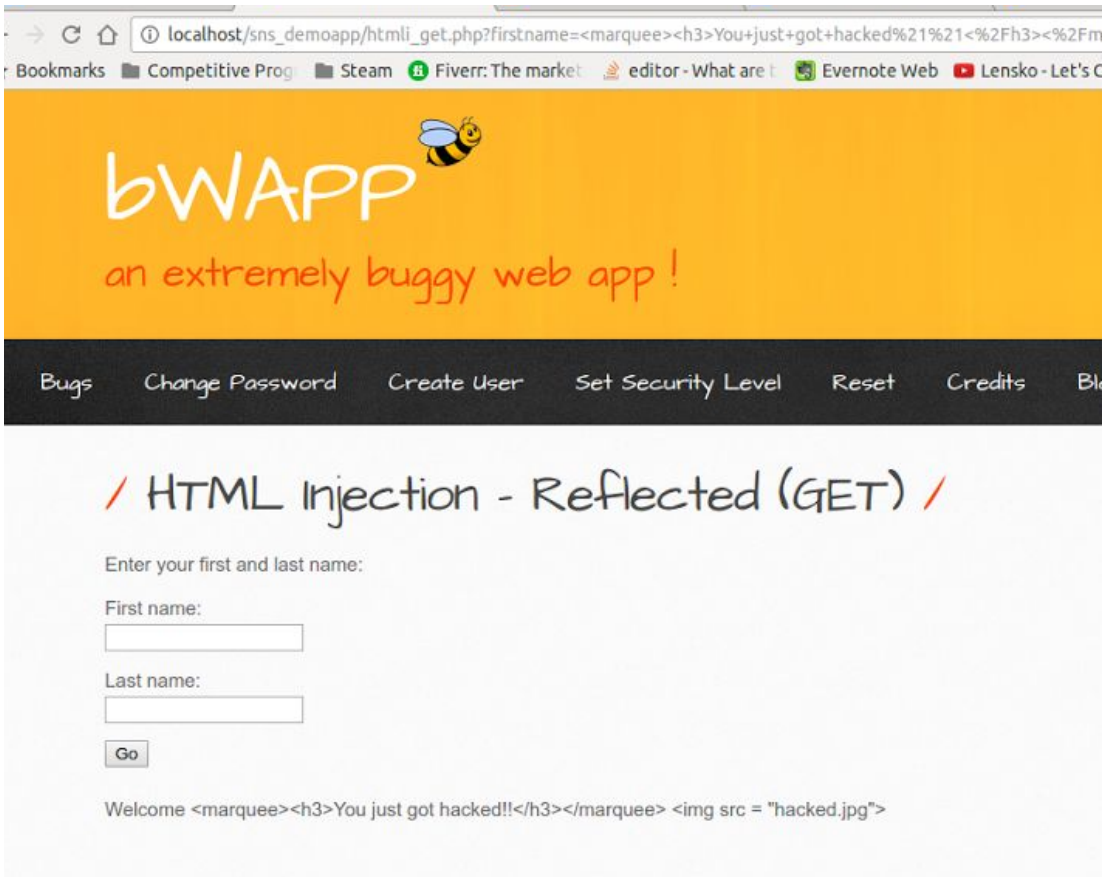


- **Plausible solution** :

i. Identify the server side script file responsible for handling form parameters (**htmli_get.php**)

ii. Identify the server side script file responsible for handling form parameters (htmli_get.php)

iii. Edit the above shown section of code to pass parameter to htmlspecialchars() function in PHP to encode html text. Save the file and reset the webpage. Now the bug is no longer present.

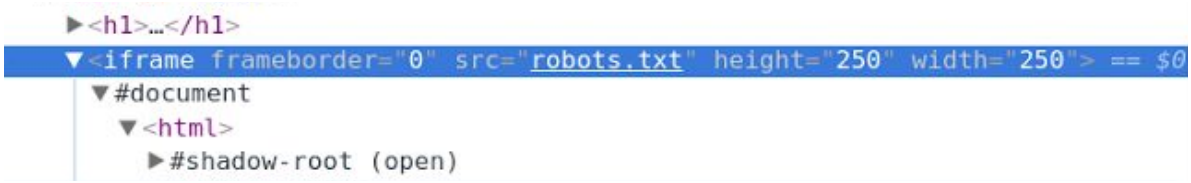


Vulnerability 2:

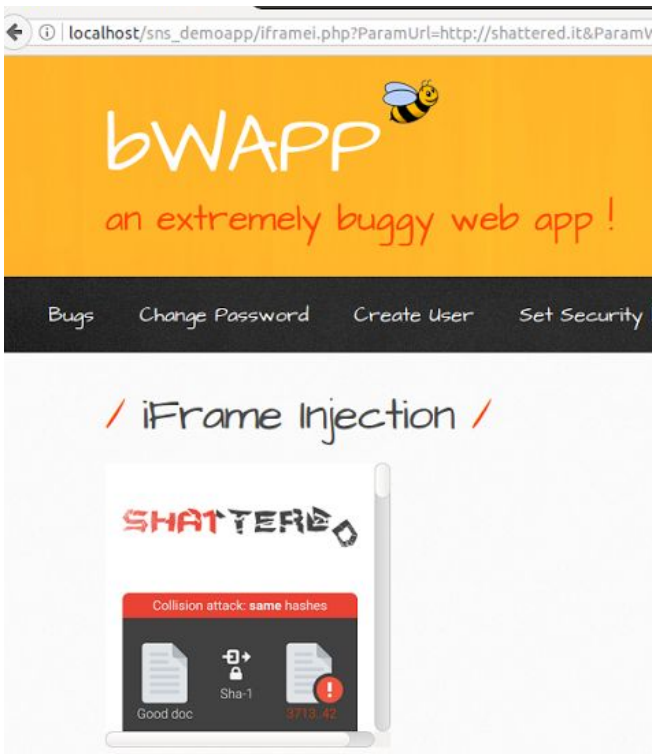
- **Vulnerability type** : Cross Site Scripting (XSS)
- **Vulnerability scope** : Main site
- **Title** : IFRAME INJECTION
- **URL** : http://localhost/sns_demoapp/iframei.php?ParamUrl=http://shattered.it&ParamWidth=250&ParamHeight=250
- **Description** : The inclusion of an iFrame in the webpage and the exposure of the URL parameter for the iFrame render the webpage vulnerable to the attack. This enables attackers to inject malware containing web links using cross site scripting in popular websites. Injected content may range from less harmful advertisements only to more harmful keystroke logging or malware downloading websites.
- **Impact** : Moderate to Severe
- **Proof of concept** :

i. Navigate to the URL : http://localhost/sns_demoapp/iframei.php?ParamUrl=robots.txt&ParamWidth=250&ParamHeight=250

ii. The iFrame source URL is robots.txt and is exposed as a parameter in the URL

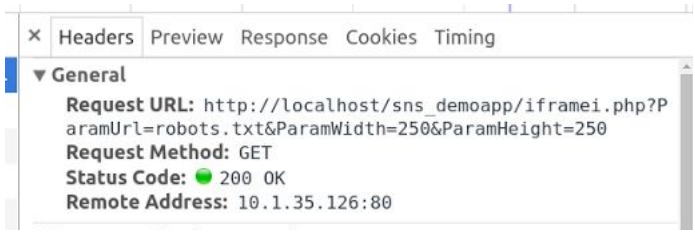


iii. Replacing the ParamUrl parameter in the URL with a link to another website, we are able to display another page.



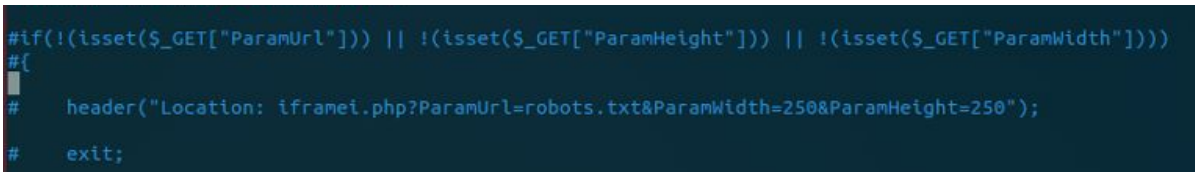
- **Plausible solution** :

i. Identify the server side script file responsible for handling the request (**iframei.php**)



ii. It is observed that the iFrame URL is passed back as a raw HTTP header. header() function is used to send a raw HTTP header.

iii. The line must be omitted as it is unsafe to send back content in the raw header. For current purposes, the line is simply commented out. The iFrame tag is modified to get parameters directly. As a result there is no longer an entry point for the attacker to inject a URL via parameters.



Vulnerability 3:

- **Vulnerability type** : Injection
- **Vulnerability scope** : Main site
- **Title** : OS COMMAND INJECTION
- **URL** : http://localhost/sns_demoapp/commandi.php
- **Description** : The flaw here is that shell commands that are concatenated with the use of ';' or '|' or '&&' operators are not checked, and are thus executed. This opens up all possibilities for execution of shell commands, some particularly malicious ones such as enabling reverse shells listening on a particular port or fork bombs that can hang a system.
- **Impact** : Severe
- **Proof of concept** :

i. Navigate to the URL : http://localhost/sns_demoapp/commandi.php

ii. Running a normal dns query would return an output as shown below

iii. Replace the input textbox contents with the string "www.google.com ; cat /etc/passwd"

iv. The output contains the output of shell commands that were never intended to be executed from the developer's perspective.

- **Plausible solution** : Injection

i. Identify the server side script file responsible for handling form parameters (**commandi.php**)

```
▶<h1>…</h1>
▶<form action="/sns_demoapp/commandi.php" method="POST">…</form>
▶<p align="left">…</p> == $0
</div>
▶<div id="side">…</div>
▶<div id="disclaimer">…</div>
```

ii. We observe that the input form element received at the server is not validated before passing it for execution

```
switch($_COOKIE["security_level"])
{

    case "0" :

        $data = no_check($data);
        break;
```

```
else
{

    echo "<p align=\"left\">" . shell_exec("nslookup  " . commandi($target)) . "</p>";
```

iii. To fix this vulnerability, escapeshellarg() adds single quotes around a string and quotes/escapes any existing single quotes allowing you to pass a string directly to a shell function and having it be treated as a single safe argument. This function should be used to escape individual arguments to shell functions coming from user input. The shell functions include exec(), system() and the backtick operator.

```
switch($_COOKIE["security_level"])
{

    case "0" :

        $data = escapeshellarg($data);
        break;
```

No result returned

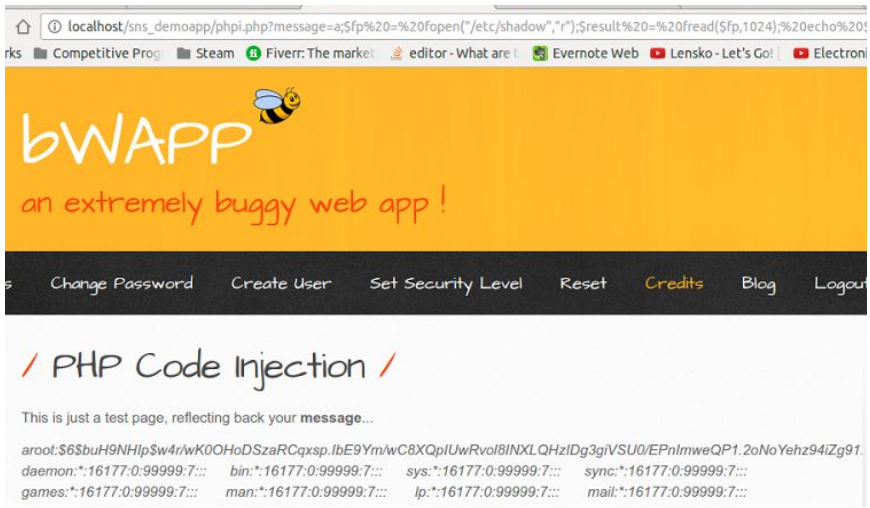
Vulnerability 4:

- **Vulnerability type** : Code Injection
- **Vulnerability scope** : Main site
- **Title** : PHP CODE INJECTION
- **URL** : http://localhost/sns_demoapp/phpi.php?message=a;echo%20%22what%22;%20$fp%20=%20fopen(%22/etc/shadow%22,%22r%22);$result%20=%20fread($fp,1024);%20echo%20$result
- **Description** : Code Injection is the general term for attack types which consist of injecting code that is then interpreted/executed by the application. This type of attack exploits poor handling of untrusted data and a lack of proper input/output data validation. If an attacker is able to inject PHP code into an application and have it executed, he is only limited by what PHP is capable of. Here, it is possible to append the page request URL with php code and have it executed.
- **Impact** : Severe
- **Proof of concept** :

i. Navigate to the URL : http://localhost/sns_demoapp/phpi.php

ii. The page request URL allows you to append a parameter to the URL called “message”. This can be exploited as it turns out it is not validated properly.

iii. In this example, we set the message parameter as [message=a;$fp = fopen("/etc/shadow","r") ;$result = fread($fp,1024); echo $result]. This lists out the shadow file contents on the page.



- **Plausible solution** :

i. Identify the server side script file responsible for handling the request (**phpi.php**)

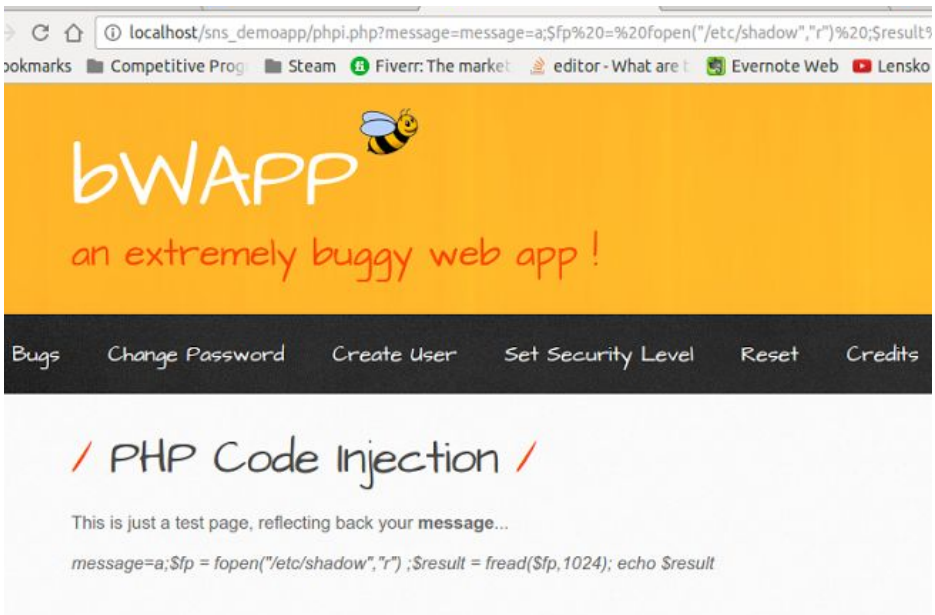
ii. The “message” parameter is passed to the eval function without any check on the contents of the message. The eval() function is used to run arbitrary PHP code

```
// If the security level is not MEDIUM or HIGH
if($_COOKIE["security_level"] != "1" && $_COOKIE["security_level"] != "2")
{
?>
<p><i><?php @eval ("echo " . $_REQUEST["message"] . ";");?></i></p>
```

iii. In order to disallow the execution of the message contents, htmlentities() function can be used to convert potential interpretable characters to no longer be interpreted. Moreover, we remove the eval() function as it is dangerous and interprets the string given to it as php code.

```
if($_COOKIE["security_level"] != "1" && $_COOKIE["security_level"] != "2")
{

<p><i><?php echo htmlspecialchars($_REQUEST["message"], ENT_QUOTES, "UTF-8");?></i></p>
```

iv. As a result, code strings inserted by the attacker are no longer executed. Thus the vulnerability is patched.



Vulnerability 5:

- **Vulnerability type** : SQL Injection
- **Vulnerability scope** : Main site
- **Title** : SQL INJECTION - GET/SEARCH AND GET/SELECT
- **Description** : SQL Injection (SQLi) refers to an injection attack wherein an attacker can execute malicious SQL statements that control a web application's database server. A variety of sql injection commands varying in structure can be tested to check for successful recovery of unintended results.
- **Impact** : Severe
- **Proof of concept** :

i. Navigate to the URL : http://localhost/sns_demoapp/sqli_1.php

ii. Several types of commands listed in the sql injection commands cheat sheet were run to obtain some pretty interesting results.

SQL Injection Payload - **title=hulk' or 1=1#**

SQL Injection Payload - **title=hulk' union select 1,2,3,4,5,6,7 #** (Capture exploitable column information)

SQL Injection Payload - **title=hulk' union select 1,user(),@@version,4,5,6,7 #** (user information)

- **Plausible solution** :

i. Identify server side script file responsible for handling request (**sqli_1.php**)

ii. The vulnerability is caused by the lack of input validation before passing the data string to the mysql query. The section of code responsible for validation is not doing anything in this case.

```
switch($_COOKIE["security_level"])
{

    case "0" :

        $data = no_check($data);
        break;
```

iii. The fix involves applying validation to ensure that interpretable characters such as quotation marks, backslashes, etc. are not inadvertently parsed. PHP function **mysqli_real_escape_string()** escapes special characters in the unescaped_string, taking into account the current character set of the connection so that it is safe to place it in a mysqli_query(). mysqli_real_escape_string() prepends backslashes to the following characters: \x00, \n, \r, \, ', " and \x1a.

```
switch($_COOKIE["security_level"])
{

    case "0" :

        $data = mysqli_real_escape_string($link, $data);
        break;
```

iv. After applying the code update, we see that none of the sql injection commands operate anymore. Thus, we have patched the vulnerability.

Vulnerability 6:

- **Vulnerability type** : Injection
- **Vulnerability scope** : Main site
- **Title** : XML/XPATH INJECTION - LOGIN FORM
- **URL** : http://localhost/sns_demoapp/xmli_1.php?login=%27or+%271%27%3D1&password=sdfsd&form=submit
- **Description** : XPath Injection is an attack technique used to exploit applications that construct XPath (XML Path Language) queries from user-supplied input to query or navigate XML documents. The syntax of XPath bears some resemblance to an SQL query, and indeed, it is possible to form SQL-like queries on an XML document using XPath. Here, the application is vulnerable to this attack as the username and password fields can be both set to 1' or '1' = '1 to gain further access.
- **Impact** : Severe
- **Proof of concept** :

i. Navigate to the URL : http://localhost/sns_demoapp/xmli_1.php
ii. Set the username and password fields as 1' or '1' = '1
iii. The result is that access to the account is gained

- **Plausible solution** :

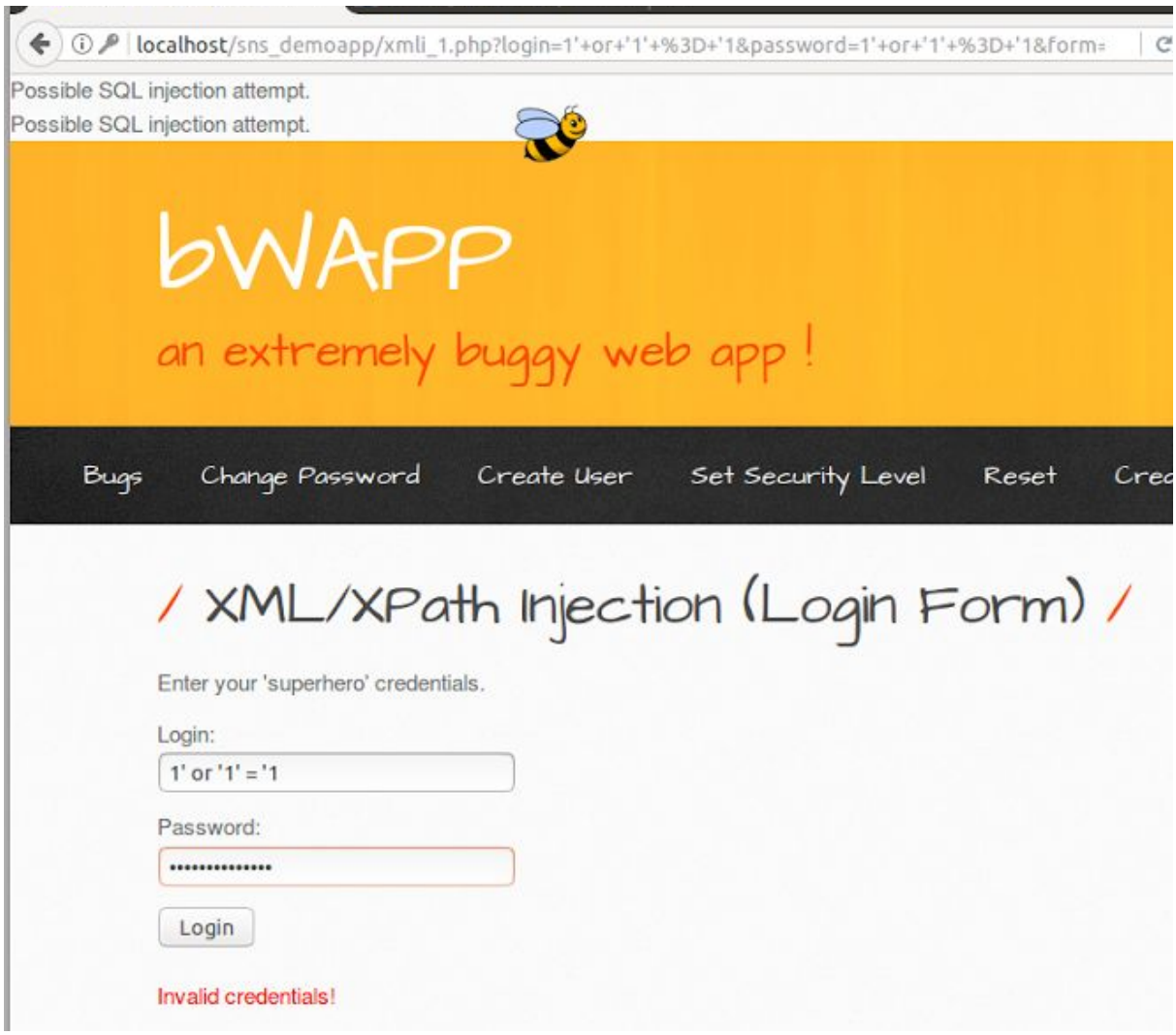
i. Identify server side script file responsible for handling the request (**xmli_1.php**)

```
<form action="/sns_demoapp/xmli_1.php" method="GET"> == $0
```

ii. The data is being passed to the xpath function without any validation. We correct it by checking the string for the presence of non alphanumeric characters (assumption is that alphanumeric characters constitute the username and password valid formats). We do a simple preg_match() on the string. Any suspicious string is returned as empty string, thus failing to pass any database query.

```
case "0" :

    $data = no_check($data);
    if (!preg_match('/^[A-Za-z][A-Za-z0-9_]*$/', $data)) {
            echo "<h1>Possible SQL injection attempt.</h1>";
            return "";
    }
    break;
```

iii. The webpage is thus safeguarded from traditional xpath injection attacks. Results are shown below.

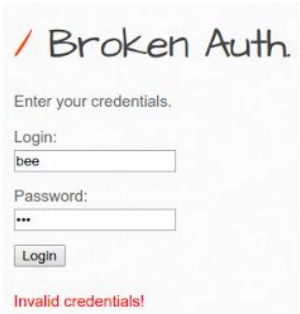


Vulnerability 7:

- **Vulnerability type** : Broken Authentication
- **Vulnerability scope** : Main site
- **Title** : BROKEN AUTHENTICATION - INSECURE LOGIN FORMS
- **URL** : http://localhost/sns_demoapp/ba_insecure_login_1.php
- **Description** : The username and password are present in the HTML source file itself! Hiding is attempted by making the font white but is not effective at all.
- **Impact** : Low (Would require extreme levels of incapability by the developer)
- **Proof of concept** :

i. Navigate to the URL : http://localhost/sns_demoapp/ba_insecure_login_1.php

ii. Inspect source HTML file




```
▼<form action="/sns_demoapp/ba_insecure_login_1.php" method="POST">
  ▼<p>
      <label for="login">Login:</label>
      <font color="white">tonystark</font>
      <br> == $0
      <input type="text" id="login" name="login" size="20">
    </p>
  ▼<p>
      <label for="password">Password:</label>
      <font color="white">I am Iron Man</font>
```

iii. Retrieve username and password and use it!

- **Plausible solution** :

i. Identify the server side script file responsible for handling the request (**ba_insecure_login_1.php**)

ii. Remove the username and password text labels from the source HTML file. The bug is fixed as a result.

```
<form action="<?php echo($_SERVER["SCRIPT_NAME"]);?>" method="POST">

    <p><label for="login">Login:</label><font color="white">tonystark</font><br />
    <input type="text" id="login" name="login" size="20" /></p>

    <p><label for="password">Password:</label><font color="white">I am Iron Man</font><br />
    <input type="password" id="password" name="password" size="20" /></p>
```

```
<form action="/sns_demoapp/ba_insecure_login_1.php" method="POST">
  <input type="text" id="login" name="login" size="20">
  <p></p>
  <input type="password" id="password" name="password" size="20">
```

Vulnerability 8:

- **Vulnerability type** : Session Management
- **Vulnerability scope** : Main site
- **Title** : SESSION MANAGEMENT - ADMINISTRATIVE PORTALS
- **URL** : http://localhost/sns_demoapp/smgmt_admin_portal.php?admin=1
- **Description** : The admin portal is by default locked. However, with a simple change in the URL request parameters (setting the value of admin to 1), we are able to gain access to the admin portal. This is a serious flaw of sending critical parameters in a GET request.
- **Impact** : Low (Would require extreme levels of incapability by the developer)
- **Proof of concept** :

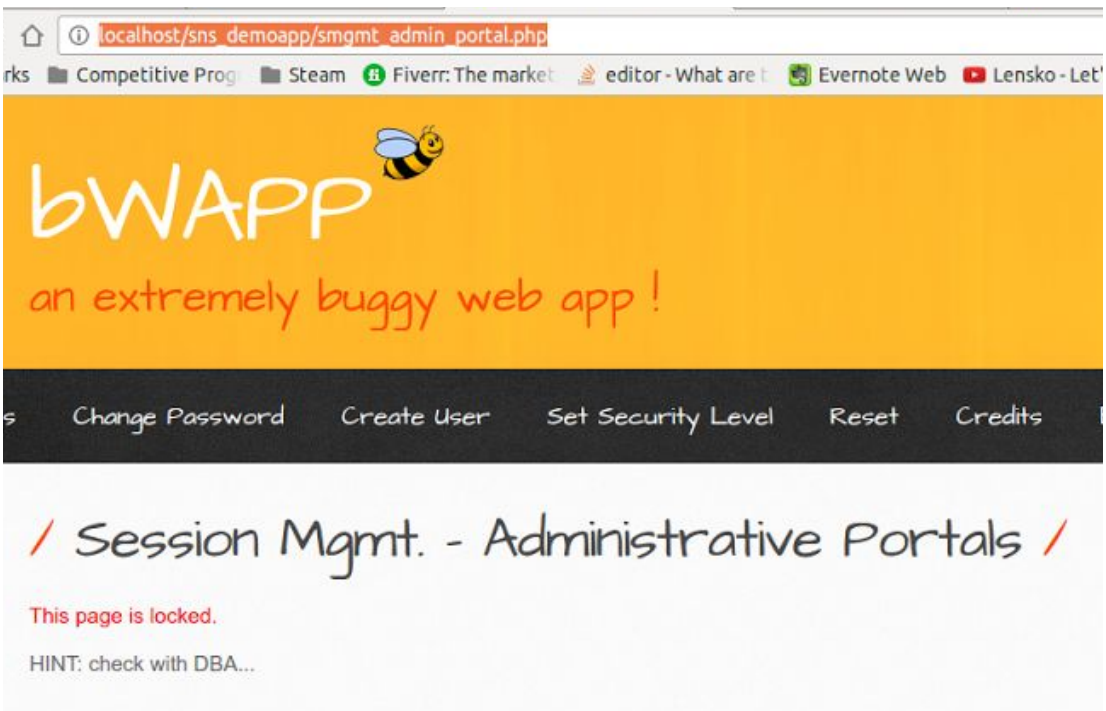
i. Navigate to the URL : http://localhost/sns_demoapp/smgmt_admin_portal.php?admin=0

ii. Change value of admin parameter to 1 and send the page request. Admin portal is now unlocked.

- **Plausible solution** :
  i. Identify server side script file responsible for handling the request (**smgmt_admin_portal.php**)
  ii. We change the GET method of accepting parameters to SESSION way as neither POST nor COOKIE methods are good enough. SESSION variables are entirely under the control of the server, hence can be controlled. COOKIE, POST and GET requests can be modified on the other hand.

```
$message = "";
$_SESSION["admin"] = 0;
switch($_COOKIE["security_level"])
{

    case "0" :

        if(isset($_SESSION["admin"]))
        {

            if($_SESSION["admin"] == "1")
            {

                $message = "Cowabunga...<p><font color=\"green\">You unlocked this page using an URL ma
nipulation.</font></p>";

            }

            else
            {

                $message="<font color=\"red\">This page is locked.</font><p>HINT: check with DBA...</p
>";

            }

        }

        else
        {

            $message="<font color=\"red\">This page is locked.</font><p>HINT: contact your dba...</p>";

        }
```

  iii. Consequently, the admin parameter is no longer exposed for change at the user end. The server will check for the admin status of the current user, before granting admin access.

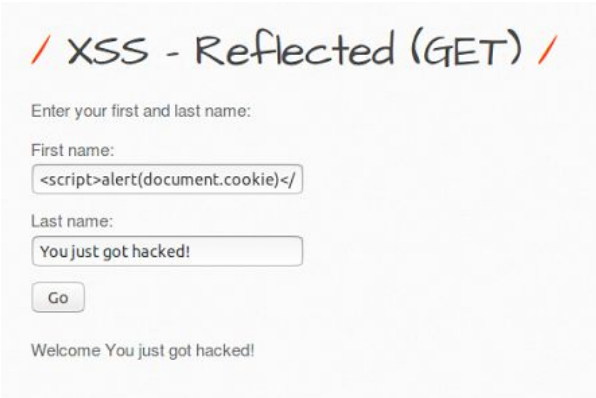


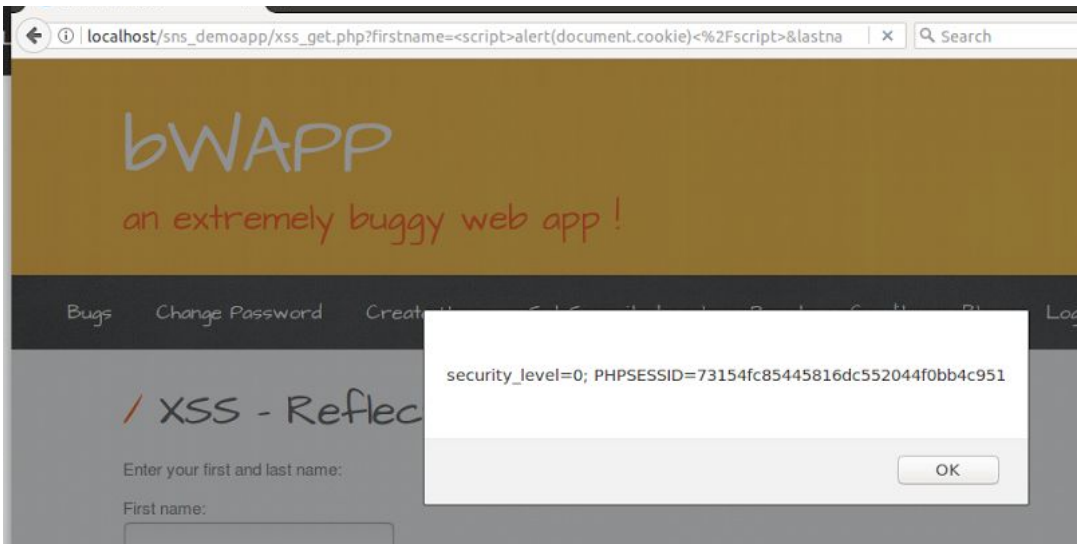
Vulnerability 9:

- **Vulnerability type** : Cross Site Scripting
- **Vulnerability scope** : Main site
- **Title** : XSS - REFLECTED GET
- **URL** : http://localhost/sns_demoapp/xss_get.php?firstname=%3Cscript%3Ealert%28document.cookie%29%3C%2Fscript%3E&lastname=%3Cscript%3Ealert%28document.cookie%29%3C%2Fscript%3E&form=submit
- **Description** : The danger behind XSS is that it allows an attacker to inject content into a website that forcefully executes on the victim's browser while loading the page. Here, due to lack of user input validation of parameters passed from the client at the server side, it is possible to embed javascript code that can execute on the page once it is served back to the client.
- **Impact** : Severe
- **Proof of concept** :

i. Navigate to the URL : http://localhost/sns_demoapp/xss_get.php

ii. Enter the following phrases in the First Name and Last Name fields respectively : **<script>alert(document.cookie)</script>** and **Warning! Hacking in progress!**





iii. The javascript code that is executed returns the cookie of the current page. Further code could easily be added to send back the cookie to the malicious attacker.

- **Plausible solution** :

I. Identify server side script file responsible for handling the request (**xss_get.php**)

```
<p>Enter your first and last name:</p>
▶<form action="/sns_demoapp/xss_get.php" method="GET">…</form> == $0
<br>
```

II. Convert user input by passing it through htmlentities() function to convert interpretable characters to non-interpretable versions of themselves.

```
if(isset($_GET["firstname"]) && isset($_GET["lastname"]))
{

    $firstname = htmlentities($_GET["firstname"]);
    $lastname = htmlentities($_GET["lastname"]);
```

III. The vulnerability is thus patched.

Vulnerability 10:

- **Vulnerability type** : Cross Site Scripting (XSS)
- **Vulnerability scope** : Main site
- **Title** : XSS Reflected JSON
- **URL** : http://localhost/sns_demoapp/xss_json.php?title=watchmen%22%7D%5D%7D%27%3Bprompt%28%22Enter+password+to+continue+%3A+%22%29%3B+alert%28%22Password+received+with+thanks%22%29%3B%2F%2F&action=search
- **Description** : The input value in the “search for a movie” field is submitted to the server. The server does not validate the contents of the field. This leads to the serious flaw when the field is returned back to the client browser as a JSON object. To parse the JSON object for appropriate display of the contents, javascript code is executed. If the contents of the original field itself contains javascript code written in a clever fashion, it is possible to execute javascript code.
- **Impact** : Severe
- **Proof of concept** :

i. Navigate to the URL : http://localhost/sns_demoapp/xss_json.php

ii. It is observed upon inspection of the source code that the text box field contents returned from the server is a JSON object that is parsed using javascript.

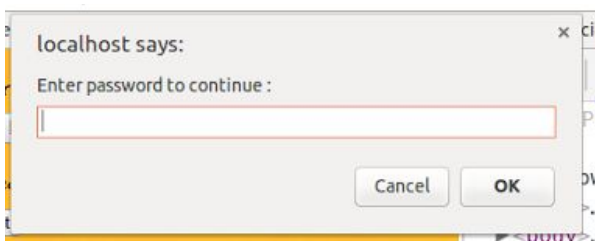

```
<script> == $0

        var JSONResponseString = '{"movies":[{"response":"watchmen???
Sorry, we don&#039;t have that movie :("}]}';

        // var JSONResponse = eval ("(" + JSONResponseString + ")");
        var JSONResponse = JSON.parse(JSONResponseString);


document.getElementById("result").innerHTML=JSONResponse.movies[0].res
ponse;
```

iii. The JSON string can be closed early by introducing "}]}' right after the name of the movie and then appending javascript code after it. Finally the string is appended with // symbol to comment out rest of the string that is part of the JSON object response. The field input is as follows : **watchmen"}]}';*prompt("Enter password to continue:"); alert("Password received with thanks");//***

iv. The result of passing the above mentioned input to the movies field results in execution of external javascript code.

```
▼<script> == $0

        var JSONResponseString = '{"movies":
[{"response":"watchmen"}]}';prompt("Enter password to continue : ");
alert("Password received with thanks");//??? Sorry, we don&#039;t have
that movie :("}]}';

        // var JSONResponse = eval ("(" + JSONResponseString + ")");
        var JSONResponse = JSON.parse(JSONResponseString);


document.getElementById("result").innerHTML=JSONResponse.movies[0].res
```

v. For example, we can steal user information without the user’s knowledge



- **Plausible solution** :

i. Identify the server side script file responsible for handling the request (**xss_json.php**)

```
<form action="/sns_demoapp/xss_json.php" method="GET">
```

ii. In the file, the title of the movie that is obtained from the user is stored as it is without any validation or format checking. However, this very unsafe as we have seen.

```
if($_COOKIE["security_level"] != "1" && $_COOKIE["security_level"] != "2")
{

    // Retrieves the movie title
    $title = $_GET["title"];
```

iii. To fix the vulnerability, we would need to disallow special characters that are normally interpreted by the browser from being interpreted. Thus we use the htmlspecialchars() function to convert special characters such as single quotes, double quotes, etc. to their respective replacement codes.

```
if($_COOKIE["security_level"] != "1" && $_COOKIE["security_level"] != "2")
{

    // Retrieves the movie title
    $title = htmlspecialchars($_GET["title"], ENT_QUOTES, "UTF-8");
```

iv. After making the change at the server script file, when the same request is sent again, we observe that the attack no longer works as the special characters have been converted to html printing format. Thus the vulnerability is patched.

```
▼<script> == $0

        var JSONResponseString = '{"movies":
[{"response":"watchmen&quot;}]}&#039;;prompt(&quot;Enter password to
continue : &quot;); alert(&quot;Password received with
thanks&quot;);//??? Sorry, we don&#039;t have that movie :("}]}';

        // var JSONResponse = eval ("(" + JSONResponseString + ")");
        var JSONResponse = JSON.parse(JSONResponseString);


document.getElementById("result").innerHTML=JSONResponse.movies[0].res
```

/ XSS - Reflected (JSON) /

Search for a movie: Search

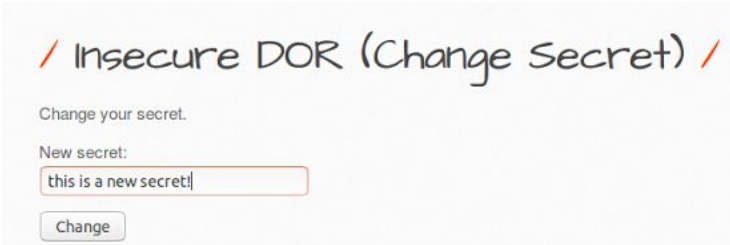
watchmen"}]}';prompt("Enter password to continue : "); alert("Password received with thanks");//??? Sorry, we don't have that movie :(

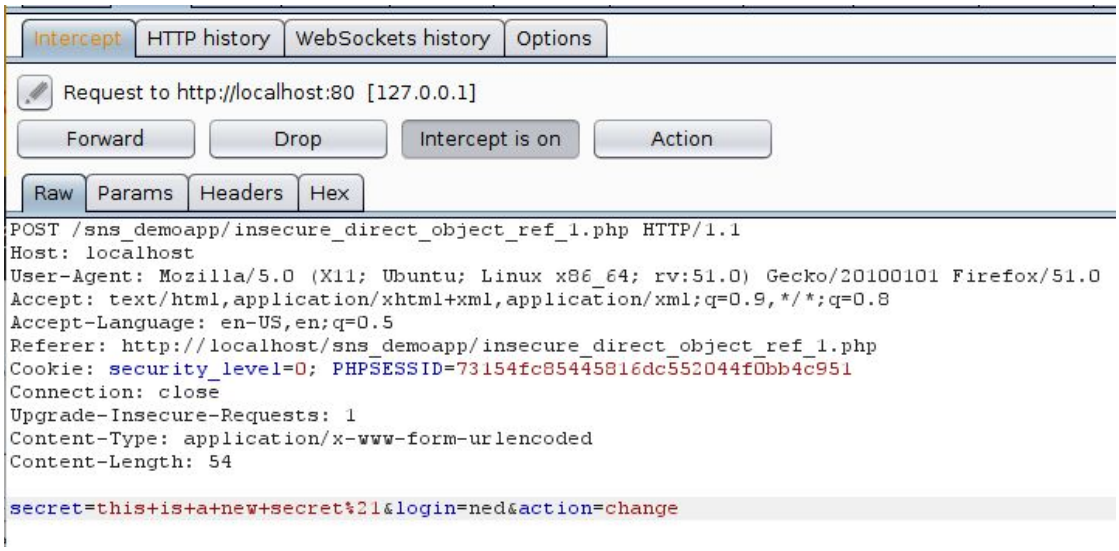
Vulnerability 11:

- **Vulnerability type** : Insecure Direct Object Reference
- **Vulnerability scope** : Main site
- **Title** : INSECURE DOR - CHANGE SECRET
- **URL** : http://localhost/sns_demoapp/insecure_direct_object_ref_1.php
- **Description** : Insecure Direct Object References occur when an application provides direct access to objects based on user-supplied input. As a result of this vulnerability attackers can bypass authorization and access resources in the system directly, for example database records or files. In this case, user supplied login ID is being directly used to access and update the database at the back end without checking it with the current session login ID and whether it matches or not.
- **Impact** : Severe
- **Proof of concept** :

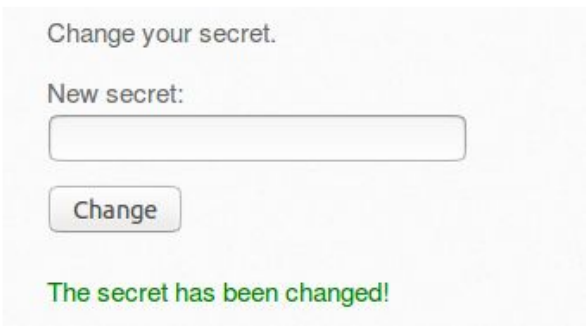
i. Navigate to the URL : http://localhost/sns_demoapp/insecure_direct_object_ref_1.php

ii. Burp suite is used to intercept the message between the browser and server code. After intercepting the message, we modify the body of the POST message to change the login ID to something else such as “ned”.





iii. Although the message shows success, the user is not aware that the secret hasn’t changed.



- **Plausible solution** :

i. Identify server side script file responsible for handling request (**insecure_direct_object_ref_1.php**)

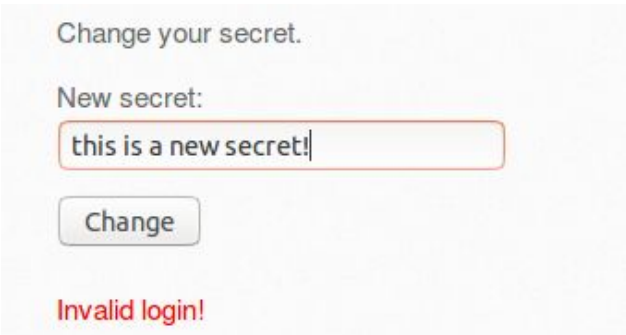
ii. The script code accepts the login ID from the user and does not check to see whether it truly is the currently logged in user (i.e logged in user in session variable).

```
// If the security level is not MEDIUM or HIGH
if($_COOKIE["security_level"] != "1" && $_COOKIE["security_level"] != "2")
{

    if(isset($_REQUEST["login"]) && $_REQUEST["login"])
    {

        $login = $_REQUEST["login"];
        $login = mysqli_real_escape_string($link, $login);

        $secret = mysqli_real_escape_string($link, $secret);
        $secret = htmlspecialchars($secret, ENT_QUOTES, "UTF-8");

        $sql = "UPDATE users SET secret = '" . $secret . "' WHERE login = '" . $login . "'"
```

iii. To fix the vulnerability, we would have to check the user supplied login ID with that of the session stored login ID. Only if they match, can we proceed forward with the database query.

```
// If the security level is not MEDIUM or HIGH
if($_COOKIE["security_level"] != "1" && $_COOKIE["security_level"] != "2")
{

    if(isset($_REQUEST["login"]) && $_REQUEST["login"] == $_SESSION["login"])
    {

        $login = $_REQUEST["login"];
        $login = mysqli_real_escape_string($link, $login);

        $secret = mysqli_real_escape_string($link, $secret);
        $secret = htmlspecialchars($secret, ENT_QUOTES, "UTF-8");
        $sql = "UPDATE users SET secret = '" . $secret . "' WHERE login = '" . $login . "'"
```

iv. Now, if the attacker changes the login ID, it is caught by the server and an appropriate error message is displayed



v. Note that however, this does not prevent the secret message from being changed. For that, some sort of replay attack addressing mechanism must be implemented, beyond the scope of this implementation.

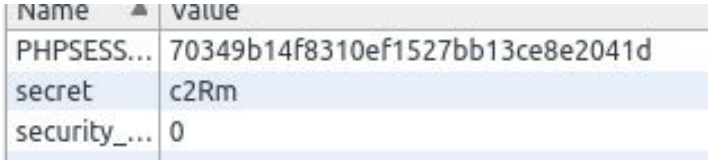
Vulnerability 12:

- **Vulnerability type** : Sensitive Data Exposure
- **Vulnerability scope** : Main site
- **Vulnerability level solved** : HIGH
- **Title** : BASE-64 ENCODING SECRET
- **URL** : http://localhost/sns_demoapp/insecure_crypt_storage_3.php
- **Description** : The encoded / encrypted cookie that is stored at the client side is simply not strong enough to prevent decoding/decryption. SHA1 and Base64 schemes are very vulnerable to cracking.
- **Impact** : Moderate (Not used commonly)
- **Proof of concept** :

i. Navigate to the URL : http://localhost/sns_demoapp/insecure_crypt_storage_3.php

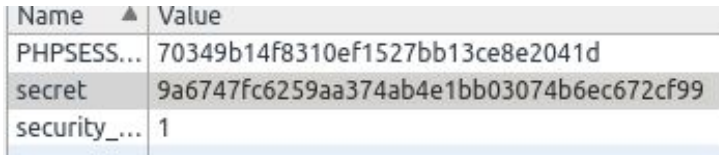
ii. Retrieve the cookies set for the page by inspecting the cookies via the console (F12) -> application -> cookies

Base64 encoding cookie (Low security level)



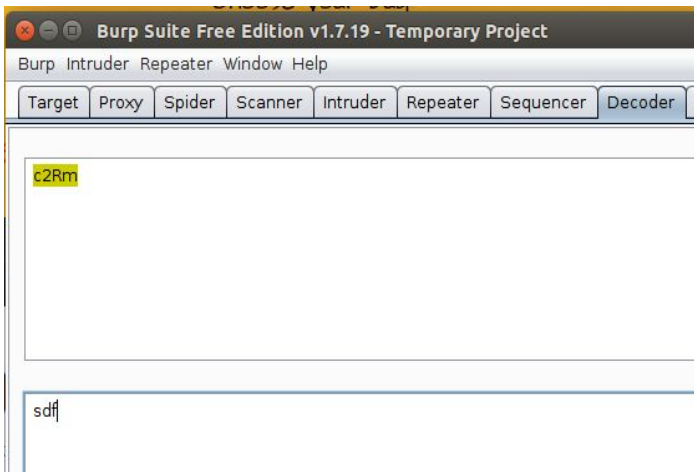
Name	Value
PHPSESS...	70349b14f8310ef1527bb13ce8e2041d
secret	c2Rm
security_...	0

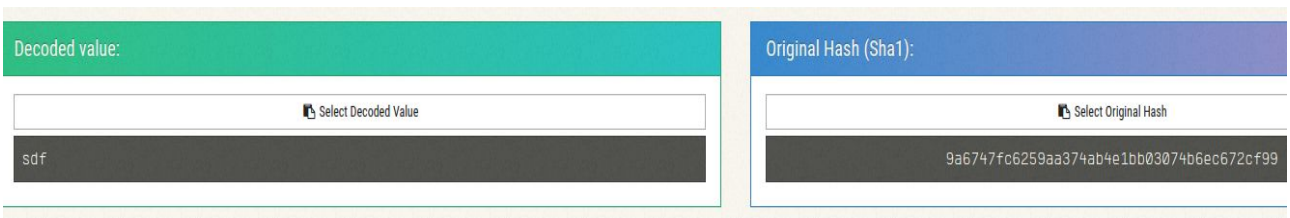
SHA 1 encrypted cookie (High security level)



Name	Value
PHPSESS...	70349b14f8310ef1527bb13ce8e2041d
secret	9a6747fc6259aa374ab4e1bb03074b6ec672cf99
security_...	1

iii. Run the encrypted cookies under Base64 decoder for the low security case and SHA1 cracker for the high security case to obtain the decoded/decrypted text as “sdf”. Thus the cookie is vulnerable to cracking.





- **Plausible solution** :

i. Identify server side script file responsible for handling request (**insecure_crypt_storage_3.php**)

```
if($_COOKIE["security_level"] == "1" or $_COOKIE["security_level"] == "2")
{

    $secret = sha1($secret);

}
else
{

    $secret = base64_encode($secret);
```

ii. Change the encryption algorithm to something much more secure such as “sha512”.

```
if($_COOKIE["security_level"] == "1" or $_COOKIE["security_level"] == "2")
{

    $secret = hash('sha512',$secret);

}

else
{

    $secret = hash('sha512',$secret);

}
```

iii. The resulting encrypted cookie is much tougher to crack :

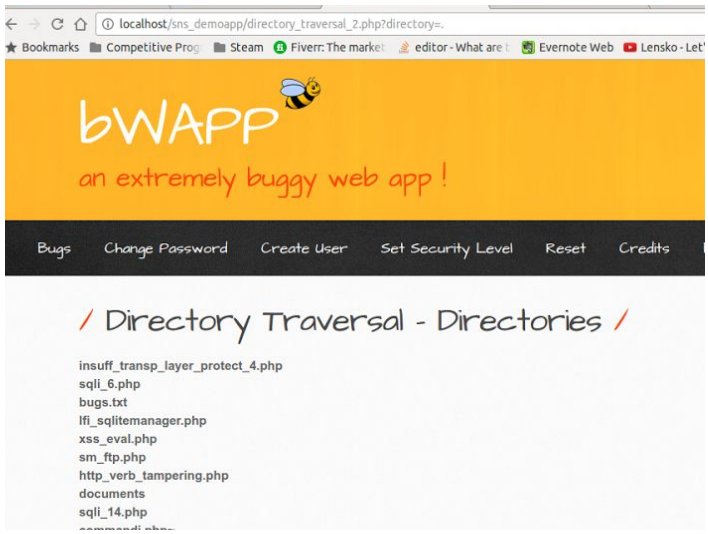
50fde99373b04363727473d00ae938a4f4debfd0afb1d428337d81905f6863b3cc303bb331ffb3361085c3a6a2ef4589ff9cd2014c90ce90010cd3805fa5fbc6

Vulnerability 13:

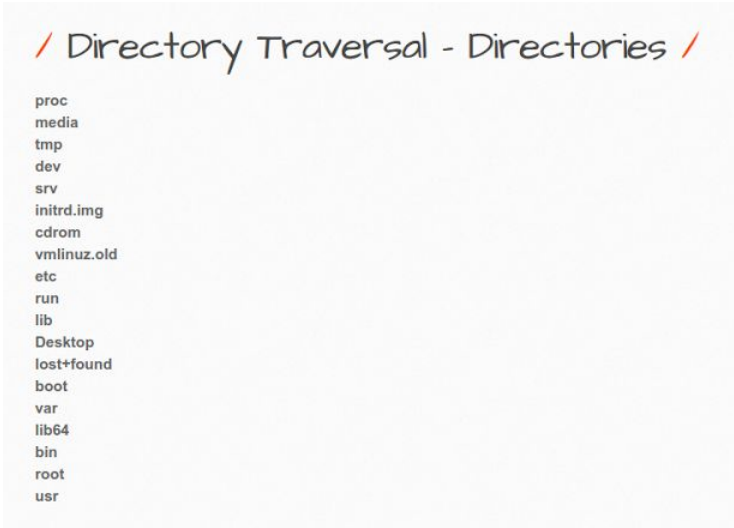
- **Vulnerability type** : Missing Functional Level Access Control
- **Vulnerability scope** : Main site
- **Title** : DIRECTORY TRAVERSAL - DIRECTORIES
- **URL** : http://localhost/sns_demoapp/directory_traversal_2.php?directory=.
- **Description** : The directory listing which is to be made available to the end user is publicly passed as a parameter in the GET request. An attacker would be able to modify the directory field to list the contents of other directories if the directory name is not validated.
- **Impact** : Moderate
- **Proof of concept** :

i. Navigate to the URL : http://localhost/sns_demoapp/directory_traversal_2.php?directory=documents

ii. Change the directory parameter from “documents” to “.” to list files in the current directory



iii. You could in a similar manner using relative paths gain access to more secure directories. Knowing the directory structure gives a lot of information to the attacker for planning future attacks that are truly malicious. For example, **directory=../../../../** leads us right to the root directory of the server.



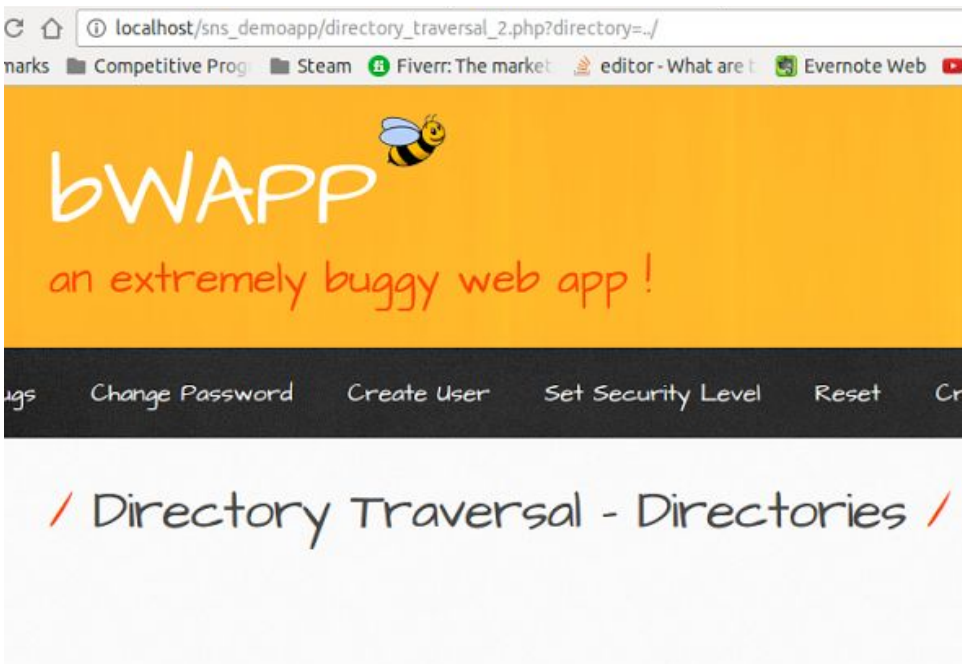
- **Plausible solution** :
  i. Identify server side script file responsible for handling the request (**directory_traversal_2.php**)
  ii. It is observed that the directory that is being taken as an input from the user is being passed to the show_directory function without validating it for being a valid request. As a result, relative path requests go through as well, thus allowing the attacker to traverse through directories that are restricted.

```
switch($_COOKIE["security_level"])
    {

    case "0" :

        show_directory($directory);

        // echo "<br />" . $_GET['page'];

        break;
```

  iii. To resolve the vulnerability, we have to ensure checks to ensure that the string "../" is not present anywhere in the directory string. Using strpos() function ensures that the check is ensured before serving the request.

```
if(isset($_GET["directory"]))
{

    $directory = $_GET["directory"];

    switch($_COOKIE["security_level"])
        {

        case "0" :
            if (strpos($directory, "../") === false) {
                    show_directory($directory);
            }

            // echo "<br />" . $_GET['page'];

            break;
```

  iv. The result is that no directory beyond the directories listing in the current directory can be traversed.



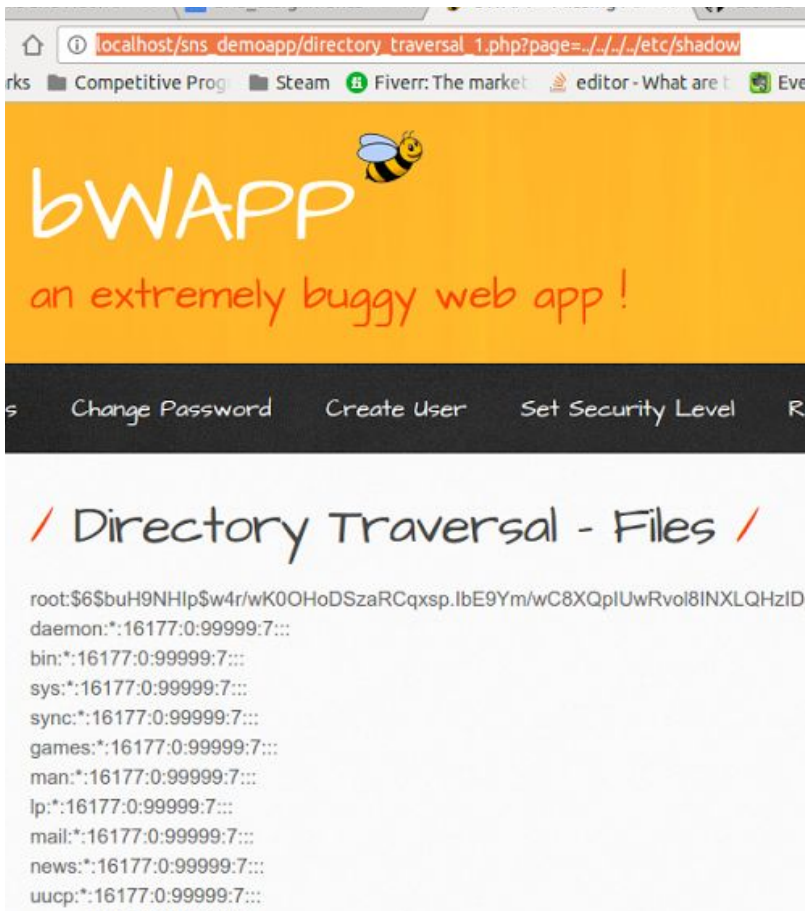
Vulnerability 14:

- **Vulnerability type** : Missing Functional Level Access Control
- **Vulnerability scope** : Main site
- **Title** : DIRECTORY TRAVERSAL - FILES
- **URL** : http://localhost/sns_demoapp/directory_traversal_1.php?page=../../../../etc/shadow
- **Description** : The page parameter which is to be made available to the end user is publicly passed as a parameter in the GET request. An attacker would be able to modify the page parameter to other files not in the current directory. This is because relative paths are not being checked. Thus the attacker could get access to hidden, protected files that are not supposed to be revealed.
- **Impact** : Severe
- **Proof of concept** :

i. Navigate to the URL : http://localhost/sns_demoapp/directory_traversal_1.php?page=message.txt

ii. The page parameter is exposed for manipulation by the attacker. The attacker after a small trial can find out that relative paths are unchecked.

iii. Change the page parameter from “message.txt” to **page=../../../../etc/shadow**. The directory path formulation can be found by trial and error as you can freely traverse the directories in recursive order. You can eventually analyze the directory structure (for example by exploiting some directory traversal vulnerability involving directory listings) and find out the relative path of the required directory. Once the directory is found, contents of the file can be directly printed by providing the path.



- **Plausible solution** :

i. Identify server side script file responsible for handling the request (**directory_traversal_1.php**)

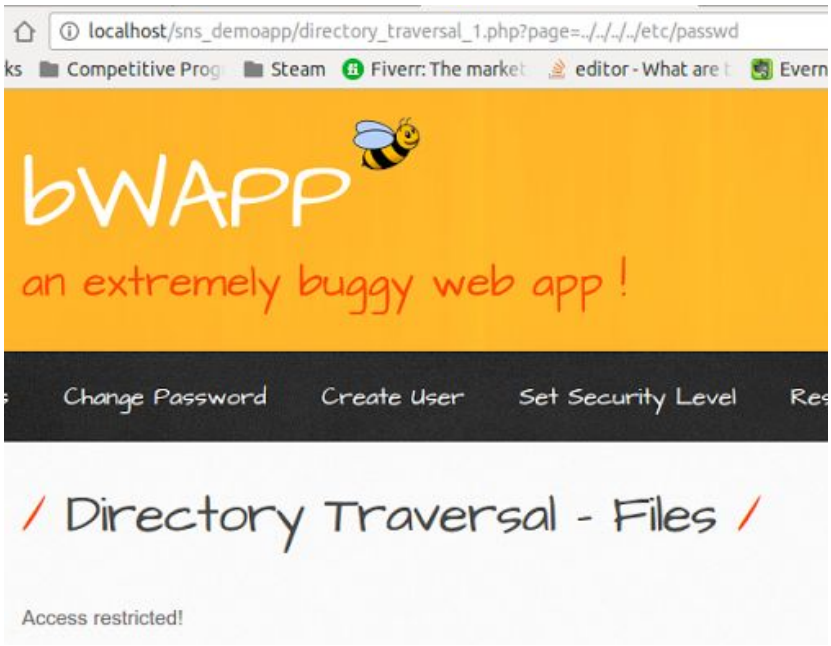
ii. Any file that is passed by the user is directly passed to the show_file function without checking it for relative path constructs. Therein lies the vulnerability.

```
if(isset($_GET["page"]))
{

    $file = $_GET["page"];

    switch($_COOKIE["security_level"])
        {

        case "0" :

            show_file($file);
```

iii. To patch the bug, we must make a check on the $file variable before passing it to the function. We can check the file name in a straightforward manner for relative path constructs.

```
switch($_COOKIE["security_level"])
    {

    case "0" :
        if (strpos($file, '../') === false) {
                show_file($file);
        }
        else {
                echo "<br /> Access restricted! ";
        }
```

iv. Refreshing the web page after making the appropriate changes in the server script file, the attack is no longer functional.

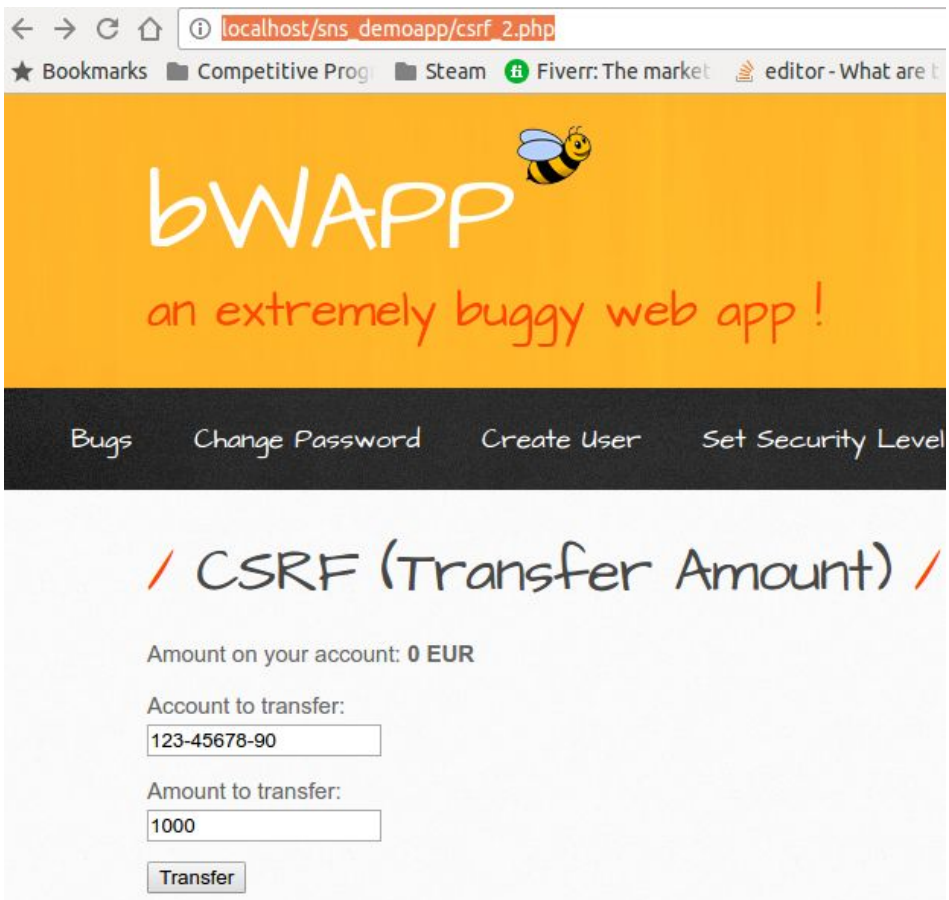


Vulnerability 15:

- **Vulnerability type** : Cross-Site Request Forgery
- **Vulnerability scope** : Main site
- **Title** : CROSS-SITE REQUEST FORGERY (TRANSFER AMOUNT)
- **URL** : http://localhost/sns_demoapp/csrf_2.php?account=Nagaraj_Account&amount=1000&action=transfer
- **Description** : Cross-Site Request Forgery (CSRF) is a type of attack that occurs when a malicious website, email, blog, instant message, or program causes a user's web browser to perform an unwanted action on a trusted site for which the user is currently authenticated.
- **Impact** : Severe
- **Proof of concept** :

i. Navigate to the URL : http://localhost/sns_demoapp/csrf_2.php

ii. Being a fund transfer portal, passing parameters via the URL is a bad idea. In this instance, the very same mistake is being made. The link can be copied and modified to point to the attacker's bank account and the amount can also be modified.



iii. A modified link that points to the attacker's account is :

csrf_2.php?account=Nagaraj_Account&amount=1000&action=transfer

You can embed this link as a hyperlink in an image and use social engineering skills to coerce a person to click on the link. Clicking on the link will result in the amount transfer being made. or example, the image below contains a hyperlink to the above url. Thus the user is fooled into clicking the link below.

****
**<img src = "lottery.jpg"> **

- **Plausible solution** :
  i. Identify server side script file responsible for handling request (**csrf_2.php**)
  ii. The first defense against such an attack is by converting the form submission type from "GET" to "POST". This ensures that the URL alone cannot be used to perform the transaction. You **cannot** embed a POST message in an object.
  iii. The figures below show the code that was in the original file, that needs to be modified.

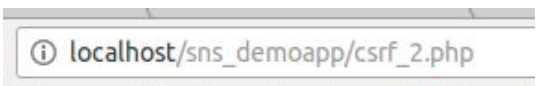
```
<h1>CSRF (Transfer Amount)</h1>

<p>Amount on your account: <b> <?php echo $_SESSION["amount"] ?> EUR</b></p>

<form action="<?php echo($_SERVER["SCRIPT_NAME"]);?>" method="GET">
```

```
if(isset($_GET["amount"]))
{

    if(($_COOKIE["security_level"] != "1" && $_COOKIE["security_level"] != "2"))
    {

        $_SESSION["amount"] = $_SESSION["amount"] - $_GET["amount"];

    }
```

  iv. After making the change, it is no longer possible to use URL embedding to carry out CSRF.

```
<h1>CSRF (Transfer Amount)</h1>

<p>Amount on your account: <b> <?php echo $_SESSION["amount"] ?> EUR</b></p>

<form action="<?php echo($_SERVER["SCRIPT_NAME"]);?>" method="POST">
```

```
if(isset($_POST["amount"]))
{

    if(($_COOKIE["security_level"] != "1" && $_COOKIE["security_level"] != "2"))
    {

        $_SESSION["amount"] = $_SESSION["amount"] - $_POST["amount"];

    }
```



  v. Thus the vulnerability can be patched.