XSRF

XSRF

XSRF = Cross Site Request Forgery

Also called CSRF or Sea-surf

XSRF

This is a malicious exploit on a website where unauthorized commands are transmitted from a user to the web server

These commands get the server to perform actions which harm the victim and benefit the attacker

XSRF

This is a malicious exploit on a website where unauthorized commands are transmitted from a user to the web server

The website trusts the user and the attacker takes advantage of this fact

XSRF

The website trusts the user and the attacker takes advantage of this fact

This is different from a XSS attack which exploits the trust a user has for a website

XSRF

This is different from a XSS attack which exploits the trust a user has for a website

XSRF exploits the trust the website has for the user

XSRF

XSRF does not try and get data from a website

Instead it focuses on the user executing actions on a website

XSRF

Instead it focuses on the user executing actions on a website

Actions such as transferring funds, downloading malicious software, purchasing an item, changing a users password

XSRF

XSS

Get data from the web server which can be used to benefit the attacker

i.e session ids, login credentials etc

XSRF

Get user to perform actions on the server to benefit the attacker

i.e transferring funds, buying something

XSRF

`http://trustedbank.com/`

The user might be logged into his trusted bank site - he has a session open

XSRF

He's also bored and he's browsing some forums on another site

XSRF

The attacker has a post on that forum

XSRF

Isn't this a great picture of Times Square?

```
<img src="http://trustedbank.com/withdraw?amount=1000000&for=Fred">
```

XSRF

```
<img src="http://trustedbank.com/
withdraw?amount=1000000&for=Fred">
```

Browsers load the URL of images, this means that the attacker can implement actions specified by the URL!

XSRF

```
<img src="http://trustedbank.com/
withdraw?amount=1000000&for=Fred">
```

The user just needs to load the page which has the embedded image

XSRF

```
<img src="http://trustedbank.com/
withdraw?amount=1000000&for=Fred">
```

He doesn't even need to click on it!

XSRF

```
<img src="http://trustedbank.com/
withdraw?amount=1000000&for=Fred">
```

Note that the attacker was aware of the exact form of a valid withdraw request

XSRF

```
<img src="http://trustedbank.com/
withdraw?amount=1000000&for=Fred">
```

The user is obviously allowed to transfer money from his own account to any other

XSRF

```
<img src="http://trustedbank.com/
withdraw?amount=1000000&for=Fred">
```

The victim was tricked into sending a request to his bank to transfer funds

XSRF

```
<img src="http://trustedbank.com/
withdraw?amount=1000000&for=Fred">
```

The withdraw URL performs an action which benefits the attacker

XSRF

Characteristics of a XSRF attack

Involve sites which know a user's identity

The user typically has to be logged in and authenticated on a site and have the authority to perform actions

XSRF

Involve sites which know a user's identity

`http://trustedbank.com/`

The user might be logged into his trusted bank site - he has a session open

Involve sites which know a user's identity

XSRF

Characteristics of a XSRF attack

Exploit the site's trust in that identity

The server trusts and allows the authenticated and authorized user to perform actions

XSRF

Exploit the site's trust in that identity

```
<img src="http://trustedbank.com/
withdraw?amount=1000000&for=Fred">
```

The user is obviously allowed to transfer money from his own account to any other

Involve sites which know a user's identity

Exploit the site's trust in that identity

XSRF

Characteristics of a XSRF attack

Tricks the browser into sending an HTTP request to the site

The victim inadvertently performs an action on the server he is logged into

XSRF

Tricks the browser into sending an HTTP request to the site

```
<img src="http://trustedbank.com/
withdraw?amount=1000000&for=Fred">
```

The victim was tricked into sending a request to his bank to transfer funds

Involve sites which know a user's identity

Exploit the site's trust in that identity

XSRF

Tricks the browser into sending an HTTP request to the site

Characteristics of a XSRF attack

Involve HTTP requests which perform **actions**

The action will harm the victim and benefit the attacker

XSRF

Involve HTTP requests which perform actions

```
<img src="http://trustedbank.com/
withdraw?amount=1000000&for=Fred">
```

The withdraw URL performs an action which benefits the attacker

XSRF

Characteristics of a XSRF attack

Involve sites which know a user's identity

Exploit the site's trust in that identity

Tricks the browser into sending an HTTP request to the site

Involve HTTP requests which perform **actions**

XSRF

What does a successful attack need?

An **authenticated** session

Knowledge of URLs which perform actions

Session management which relies only on the browser

HTML tags which **access a resource** e.g. the `img` tag

XSRF

Example1 5-XSRF-embeddedImage.php

Example1 5-XSRF-transferFunds.php

XSRF

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>XSRF</title>
</head>
<body>
  <h3>Here is a pic you might like!</h3>
  <img src="http://localhost/security/Example15-XSRF-transferFunds.php/?from_id=3333&to_id=1234&amount=1000">
</body>
</html>
```

Browsers load the URL which pretends to be an image

XSRF

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>XSRF</title>
</head>
<body>
  <h3>Here is a pic you might like!</h3>
  <img src="http://localhost/security/Example16-XSRF-transferFunds.php/?from_id=3333&to_id=1234&amount=1000">
</body>
</html>
```

It's a page which transfers funds

XSRF

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>XSRF</title>
</head>
<body>
  <h3>Here is a pic you might like!</h3>
  <img src="http://localhost/security/Example16-XSRF-transferFunds.php/?from_id=3333&to_id=1234&amount=1000">
</body>
</html>
```

And here is the from account, the to account and the amount to transfer

XSRF

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>XSRF</title>
</head>
<body>
  <h3>Here is a pic you might like!</h3>
  <img src="http://localhost/security/Example16-XSRF-transferFunds.php/?from_id=3333&to_id=1234&amount=1000">
</body>
</html>
```

Obviously the pic does not load - but this is common, as users we're trained to ignore this

XSRF

http://localhost/security/Example16-XSRF-transferFunds.php/?from_id=3333&to_id=1234&amount=1000

This particular page which performs the transfer funds action gets many things wrong

XSRF

```
http://localhost/security/Example16-
XSRF-transferFunds.php/?
from_id=3333&to_id=1234&amount=1000
```

First of all why is such an important page performing this action on a GET request?

XSRF

There is of course a whole bunch of other stuff which we'll discuss when talking about XSRF mitigation

XSRF

```
$conn = getDatabaseConnection();

echo "Transferring funds <br>";
$from_id = $conn->real_escape_string($_GET['from_id']);
$to_id = $conn->real_escape_string($_GET['to_id']);
$amount = $conn->real_escape_string($_GET['amount']);
echo "From: $from_id, To: $to_id, Amount: $amount <br>";

if (!empty($from_id) && !empty($to_id) && !empty($amount)) {
  $subtract_query =
    "UPDATE BankAccounts SET account_balance = account_balance - $amount where account_id = $from_id";
  echo $subtract_query;
  $conn->query($subtract_query);

  $add_query =
    "UPDATE BankAccounts SET account_balance = account_balance + $amount where account_id = $to_id";
  echo $add_query;
  $conn->query($add_query);

  $conn->close();
}
```

XSRF

```php
$conn = getDatabaseConnection();

echo "Transferring funds <br>";
$from_id = $conn->real_escape_string($_GET['from_id']);
$to_id = $conn->real_escape_string($_GET['to_id']);
$amount = $conn->real_escape_string($_GET['amount']);
echo "From: $from_id, To: $to_id, Amount: $amount <br>";

if (!empty($from_id) && !empty($to_id) && !empty($amount)) {
  $subtract_query =
    "UPDATE BankAccounts SET account_balance = account_balance - $amount where account_id = $from_id";
  echo $subtract_query;
  $conn->query($subtract_query);

  $add_query =
    "UPDATE BankAccounts SET account_balance = account_balance + $amount where account_id = $to_id";
  echo $add_query;
  $conn->query($add_query);

  $conn->close();
}
```

Some useless escaping here:-)

XSRF

```
$subtract_query =
  "UPDATE BankAccounts SET account_balance = account_balance - $amount where account_id = $from_id";
echo $subtract_query;
$conn->query($subtract_query);

$add_query =
  "UPDATE BankAccounts SET account_balance = account_balance + $amount where account_id = $to_id";
echo $add_query;
$conn->query($add_query);
```

This is not typically the way the SQL queries will be written

XSRF

```
$subtract_query =
  "UPDATE BankAccounts SET account_balance = account_balance - $amount where account_id = $from_id";
echo $subtract_query;
$conn->query($subtract_query);

$add_query =
  "UPDATE BankAccounts SET account_balance = account_balance + $amount where account_id = $to_id";
echo $add_query;
$conn->query($add_query);
```

This should be in a stored procedure and the execution should be atomic

XSRF

```
$conn = getDatabaseConnection();

echo "Transferring funds <br>";
$from_id = $conn->real_escape_string($_GET['from_id']);
$to_id = $conn->real_escape_string($_GET['to_id']);
$amount = $conn->real_escape_string($_GET['amount']);
echo "From: $from_id, To: $to_id, Amount: $amount <br>";

if (!empty($from_id) && !empty($to_id) && !empty($amount)) {
  $subtract_query =
    "UPDATE BankAccounts SET account_balance = account_balance - $amount where account_id = $from_id";
  echo $subtract_query;
  $conn->query($subtract_query);

  $add_query =
    "UPDATE BankAccounts SET account_balance = account_balance + $amount where account_id = $to_id";
  echo $add_query;
  $conn->query($add_query);

  $conn->close();
}
```

XSRF

Remember that this requires the user to be logged in and authenticated to his bank site

XSRF

There is an aspect of social engineering involved as well, in order to get the user to load a blog page or click on a link

XSRF

social engineering

Sending an email with a link to the user or setting up an image on pages the user visits while banking

XSRF

Let's say that the bank uses POST rather than GET, a little more secure

XSRF

Such a request cannot be delivered using <img> or <a> links

But can be submitted using a form!

XSRF

```
<form action="http://trustedbank.com/transfer.php" method="POST">
  <input type="hidden" name="from_id" value="3333"/>
  <input type="hidden" name="to_id" value="1234"/>
  <input type="hidden" name="amount" value="1000"/>
  <input type="submit" value="View my pictures"/>
</form>
```

Note that all of these are hidden fields

XSRF

```
<form action="http://trustedbank.com/transfer.php" method="POST">
  <input type="hidden" name="from_id" value="3333"/>
  <input type="hidden" name="to_id" value="1234"/>
  <input type="hidden" name="amount" value="1000"/>
  <input type="submit" value="View my pictures"/>
</form>
```

All the user sees is the "view my pictures" button

XSRF

```
<form action="http://trustedbank.com/transfer.php" method="POST">
  <input type="hidden" name="from_id" value="3333"/>
  <input type="hidden" name="to_id" value="1234"/>
  <input type="hidden" name="amount" value="1000"/>
  <input type="submit" value="View my pictures"/>
</form>
```

These become POST params to this page

XSRF

```
<form action="http://trustedbank.com/transfer.php" method="POST">
  <input type="hidden" name="from_id" value="3333"/>
  <input type="hidden" name="to_id" value="1234"/>
  <input type="hidden" name="amount" value="1000"/>
  <input type="submit" value="View my pictures"/>
</form>
```

Forms can also be submitted automatically - so even the clicking can be done away with

XSRF

```
<form action="http://trustedbank.com/transfer.php" method="POST">
  <input type="hidden" name="from_id" value="3333"/>
  <input type="hidden" name="to_id" value="1234"/>
  <input type="hidden" name="amount" value="1000"/>
  <input type="submit" value="View my pictures"/>
</form>
```

```
<body onload="document.forms[0].submit()">
```

A little bit of Javascript to submit the first form of the page

XSRF
Mitigation

XSRF
Mitigation

1. Referer header
2. origin header
3. Challenge response
4. Synchronizer token

XSRF

Mitigation - Referer header

This is an HTTP header which identifies the URL of the webpage which linked to the current webpage

XSRF

Mitigation – Referer header

This allows a page to identify where the request to this page came from

XSRF

Mitigation – Referer header

A page can check the referer and only allow those requests which come from pages it trusts

My website trusts links from the www.nyt.com but not from www.evil.com

XSRF

Mitigation – Referer header

A page can check the referer and
only allow those requests which
come from pages it trusts

e.g. allow referers only from the
same domain

XSRF

Mitigation - Referer header

This is a reasonable way to protect against XSRF but not foolproof

It's considered one of the weaker forms of protection

XSRF

Mitigation - Referer header

If an XSRF attack originates from an HTTPS domain then the referer will be omitted

XSRF
Mitigation

1. Referer header
2. origin header
3. Challenge response
4. Synchronizer token

XSRF

Mitigation - Origin header

This is a special header the browser adds when the request is made to a domain which is different from the page making the request

www.evil.com makes a request to www.trustedbank.com - this will have the origin header saying the request came from evil.com

XSRF

Mitigation - Origin header

The header will indicate the protocol, domain and port of the page from which the request was made

i.e. evil.com. The trusted bank should be able to test for this!

XSRF
Mitigation

1. Referer header
2. origin header
3. Challenge response
4. Synchronizer token

XSRF

Mitigation – Challenge response

If a request performs any state changing operation challenge it to ensure that it's doesn't go through automatically!

XSRF

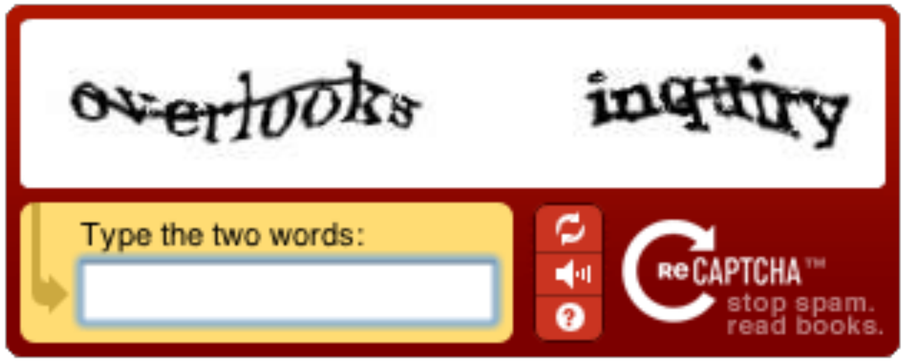
Mitigation – Challenge response

This could involve the use of a CAPTCHA code to ensure a human is behind the keyboard

XSRF

Mitigation – Challenge response

This could involve the use of a CAPTCHA code to ensure a human is behind the keyboard



XSRF

Mitigation – Challenge response

The user could be asked to re-authenticate before any critical action is performed

XSRF

Mitigation – Challenge response

The user could be asked to re-authenticate before any critical action is performed

Re-entering passwords is painful but much much safer

XSRF

Mitigation - Challenge response

Or the user could be prompted to enter a OTP (One Time Password)

XSRF

Mitigation – Challenge response

One Time Password

This is a one time token sent to the user's mobile phone or secure device

XSRF

Mitigation – Challenge response

One Time Password

It's often used to confirm sensitive and state-changing operations like transfer of funds

XSRF
Mitigation

1. Referer header
2. origin header
3. Challenge response
4. Synchronizer token

XSRF

Mitigation - Synchronizer token

This method of XSRF protection involves using a secure random token to ensure that the request comes from the trusted website

XSRF

Mitigation - Synchronizer token

secure random token

On sensitive operations this challenge token should be sent along with the request

XSRF

Mitigation - Synchronizer token

secure random token

The server then verifies that this token exists and is correct before performing the action

SIGNING UP USERS

Example16-signupWithToken.php

SIGNING UP USERS

Users sign up using a simple form which has a user name and password field

We know how to do this!

SIGNING UP USERS

```
<h3> Sign up to our new Top Secret Club! </h3>
<form method="POST" action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>">
  <span style="color: red"><?php echo $error_message;?></span>
  <br>
  <br>
  Email address:
  <br>
  <input type="text" name="user_email" maxlength="100">
  <br>
  <br>
  Password:
  <br>
  <input type="text" name="user_password" maxlength="20">
  <br>
  <br>
  <input type="submit" value="Sign up">

  <input type="hidden" name="form_token" value="<?php echo $form_token; ?>" />
</form>
<br>
```

SIGNING UP USERS

```
<h3> Sign up to our new Top Secret Club! </h3>
<form method="POST" action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>">
  <span style="color: red"><?php echo $error_message;?></span>
  <br>
  <br>
  Email address:
  <br>
  <input type="text" name="user_email" maxlength="40">
  <br>
  <br>
  Password:
  <br>
  <input type="text" name="user_password" maxlength="20">
  <br>
  <br>
  <input type="submit" value="Sign up">

  <input type="hidden" name="form_token" value="<?php echo $form_token; ?>" />
</form>
<br>
```

Users who sign up to our Top Secret Club need to fill up this form!

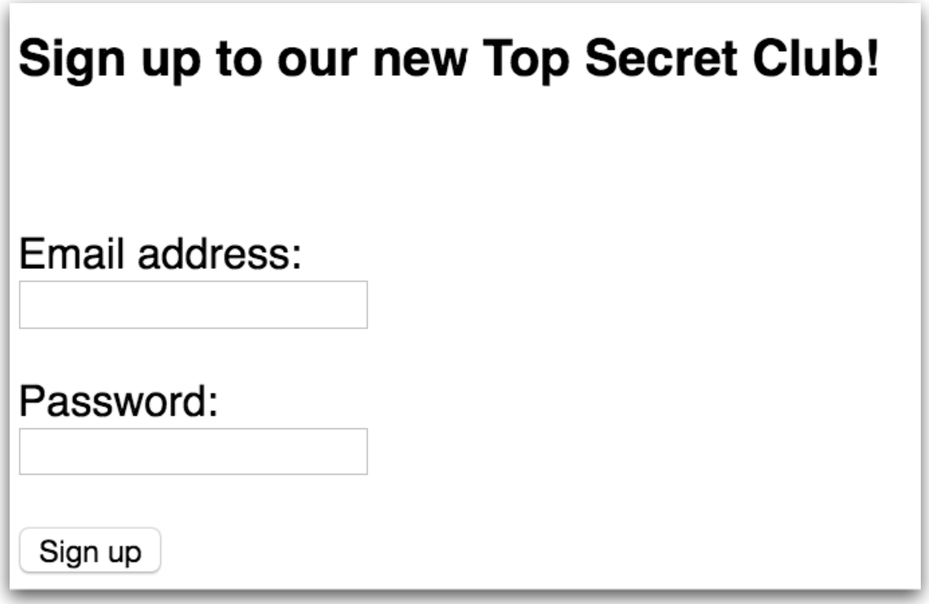
SIGNING UP USERS

```
<h3> Sign up to our new Top Secret Club! </h3>
<form method="POST" action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>">
</form>
<br>
```

The data is POST'ed to the server and processed in this same PHP file!

SIGNING UP USERS

```
<span style="color: red"><?php echo $error_message;?></span>
<br>
<br>
Email address:
<br>
<input type="text" name="user_email" maxlength="100">
<br>
<br>
Password:
<br>
<input type="text" name="user_password" maxlength="20">
<br>
<br>
<input type="submit" value="Sign up">
```



SIGNING UP USERS

```
<span style="color: red"><?php echo $error_message;?></span>
<br>
<br>
Email address:
<br>
<input type="text" name="user_email" maxlength="100">
<br>
<br>
Password:
<br>
<input type="text" name="user_password" maxlength="20">
<br>
<br>
<input type="submit" value="Sign up">
```

Limit the character lengths of the field to what you expect in the database

SIGNING UP USERS

Hmm... everything looks familiar except for this one little thing here...

```
<input type="hidden" name="form_token" value="<?php echo $form_token; ?>" />
```

SIGNING UP USERS

A hidden input field which sends to the server a $form_token

```
<input type="hidden" name="form_token" value="<?php echo $form_token; ?>" />
```

SIGNING UP USERS

This token protects our site from Cross Site Request Forgery or CSRF attacks!

SIGNING UP USERS

A malicious site can cause the browser to send requests to perform unwanted actions on a trusted site

SIGNING UP USERS

In our case we only want our own site to fill out the form to access the Top Secret Club

No other site should be able to embed that form and sign up new users

SIGNING UP USERS

In our case we only want our own site to fill out the form to access the Top Secret Club

No other site should be able to embed that form and sign up new users

We should be able to uniquely identify requests from our own site

SIGNING UP USERS

In our case we only want our own site to fill out the form to access the Top Secret Club

No other site should be able to embed that form and sign up new users

We should be able to uniquely identify requests from our own site

Create a random per-session identifier on the server and store it in a session variable

SIGNING UP USERS

We should be able to uniquely identify requests from our own site

Create a random per-session identifier on the server and store it in a session variable

Send this random identifier to the server along with every form submit

SIGNING UP USERS

We should be able to **uniquely identify** requests from our own site

Create a **random per-session identifier** on the server and store it in a session variable

Send this random identifier to the server along with **every form submit**

The server **compares** this random identifier with the one stored in the session to **validate** that the submit is from a trusted site!

SIGNING UP USERS

We should be able to uniquely identify requests from our own site

Create a random per-session identifier on the server and store it in session variable

Send this identifier to the server along with every form submit

The server compares this random identifier with the one stored in the session to validate that the submit is from a trusted site!

This token need be generated only **once** per session

SIGNING UP USERS

We should be able to uniquely identify requests from our own site

Create a random per-session identifier on the server and store it in a cookie

Send this identifier to the server along with every form submit

The server compares this random identifier with the one stored in the session to validate that the submit is from a trusted site!

The token can be **reused** for all requests across that session

SIGNING UP USERS

```
<?php
  session_start();

  $form_token = md5(uniqid('auth', true));
  $_SESSION['form_token'] = $form_token;
?>
```

Generate a random number which will serve as a token for this session

SIGNING UP USERS

```
<?php
  session_start();

  $form_token = md5(uniqid('auth', true));
  $_SESSION['form_token'] = $form_token;
?>
```

This generates a unique id based on the current time - the prefix for this id will be "auth"

SIGNING UP USERS

```
<?php
  session_start();

  $form_token = md5(uniqid('auth', true));
  $_SESSION['form_token'] = $form_token;
?>
```

The more_entropy field is true which means the id will be 23 characters long rather than 13

SIGNING UP USERS

```
<?php
  session_start();

  $form_token = md5(uniqid('auth', true));
  $_SESSION['form_token'] = $form_token;
?>
```

The unique id is not secure by itself - the **md5** hash is a **cryptographically secure** hash

SIGNING UP USERS

```
<?php
  session_start();

  $form_token = md5(uniqid('auth', true));
  $_SESSION['form_token'] = $form_token;
?>
```

Store it in the current session so it's accessible to all pages

SIGNING UP USERS

Make sure you always send along the token at the time of form submit

```
<input type="hidden" name="form_token" value="<?php echo $form_token; ?>" />
```

SIGNING UP USERS

Now at the server end we want to add the newly signed up user to the database

SIGNING UP USERS

```php
$error_message = "";
if ($_SERVER["REQUEST_METHOD"] == "POST") {
  $error_message = validate_inputs($_POST['user_email'], $_POST['user_password'], $form_token);
  // If no errors then add the user to the database.
  if (empty($error_message)) {
    mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT);

    $user_email = filter_var($_POST['user_email'], FILTER_SANITIZE_STRING);
    $user_password = filter_var($_POST['user_password'], FILTER_SANITIZE_STRING);

    try {
      $conn = getDatabaseConnection();

      $stmt = $conn->prepare(
        "INSERT INTO `Users` (user_email, user_password) VALUES (?, ?)"
      );
      $stmt->bind_param("ss", $user_email, sha1($user_password));
      $stmt->execute();

      $stmt->close();
      $conn->close();
    } catch (Exception $e) {
      // Duplicate entry for key is error 1062
      if($e->getCode() == 1062) {
        $error_message =
          'Username already exists, please sign in or choose a different user name';
      }
      else {
        $error_message =
          'We are unable to process your request. Please try again later';
      }
    }
  }
}
```

SIGNING UP USERS

```
if ($_SERVER["REQUEST_METHOD"] == "POST") {
  $error_message = validate_inputs($_POST['user_email'], $_POST['user_password'], $form_token);
  // If no errors then add the user to the database.
  if (empty($error_message)) {
```

Make sure the form inputs are valid by calling the validate_inputs() function from the included file

SIGNING UP USERS

```php
if ($_SERVER["REQUEST_METHOD"] == "POST") {
  $error_message = validate_inputs($_POST['user_email'], $_POST['user_password'], $form_token);
  // If no errors then add the user to the database.
  if (empty($error_message)) {
```

Checking whether the token received is the same as the current session token is part of the validation

SIGNING UP USERS

```
$conn = getDatabaseConnection();
```

```
$stmt = $conn->prepare(
  "INSERT INTO `Users` (user_email, user_password) VALUES (?, ?)"
);
$stmt->bind_param("ss", $user_email, sha1($user_password));
$stmt->execute();
```

We simply insert the new user into the Users table

SIGNING UP USERS

The validation of inputs now needs to check the form token as well

SIGNING UP USERS

```php
function validate_inputs($user_email, $user_password, $form_token) {
  $error = "";

  if (!isset($user_email, $user_password, $form_token)) {
    $error = 'Please enter a valid username and password';
  } elseif ($form_token != $_SESSION['form_token']) {
    $error = 'The form submission is invalid';
  } elseif (strlen($user_password) < 6 || strlen($user_password) > 20) {
    $error = 'The password length should be between 6 and 20 characters';
  } elseif (!filter_var($user_email, FILTER_VALIDATE_EMAIL)) {
    $error = 'The user name should be a valid email address';
  } elseif (!ctype_alnum($user_password)) {
    $error = 'The password should only have alphabets or numbers';
  }

  return $error;
}
```

SIGNING UP USERS

```
function validate_inputs($user_email, $user_password, $form_token) {
  $error = "";

  if (!isset($user_email, $user_password, $form_token)) {
    $error = 'The form submission is invalid';
  } elseif ($form_token != $_SESSION['form_token']) {
    $error = 'The form submission is invalid';
  } elseif (strlen($user_password) < 6 || strlen($user_password) > 20) {
    $error = 'Password must be between 6 and 20 characters';
  } elseif (!filter_var($user_email, FILTER_VALIDATE_EMAIL)) {
    $error = 'The user name should be a valid email address';
  } elseif (!ctype_alnum($user_password)) {
    $error = 'The password should only have alphabets or numbers';
  }

  return $error;
}
```

This is pretty self-explanatory except for one part

SIGNING UP USERS

```
function validate_inputs($user_email, $user_password, $form_token) {
```

```
  } elseif ($form_token != $_SESSION['form_token']) {
    $error = 'The form submission is invalid';
  }
```

Check that the form token is the **same token** that was generated in this session

XSRF
Mitigation

1. Referer header
2. origin header
3. Challenge response
4. Synchronizer token