



UTM

UNIVERSITI TEKNOLOGI MALAYSIA

SESSION 2023/2024-2

SECR4483-02

SECURE PROGRAMMING

Assignment #1: Password Security

NAME	MATRIC NO.	SECTION
MUHAMAD FAIZ BIN ABDUL MUTALIB	A21EC0059	02

DUE DATE: Sunday, 28 April 2024

LECTURER: Dr. Mohd Zamri Bin Osman

Contents

1.0 Introduction	4
2.0 Security Implementation	5
Registration Page (Sign Up):	5
<i>Diagram 1: Registration Page (Sign Up)</i>	5
<i>Code 1: Sign Up page - SignIn&SignUp.html</i>	5
<i>Code 2: Check username, email and password</i>	6
<i>Code 3: Check username, email and password functions</i>	6
<i>Diagram 2: Result for invalid username</i>	7
<i>Diagram 3: Result for invalid email</i>	7
<i>Diagram 4: Result for weak password.</i>	8
<i>Code 4: Check if the email or username already exists in the database</i>	8
<i>Diagram 5: Result for duplicate accounts.</i>	9
<i>Code 5: Send an email to the user, if successfully registered.</i>	9
<i>Diagram 6: Confirmation by email to those who successfully register.</i>	10
<i>Diagram 7: Confirmation by email</i>	10
<i>Diagram 8: Database validation (Password Hashing)</i>	10
Sign In Page (Log In):	11
<i>Diagram 9: Sign In (Log In) Page</i>	11
<i>Diagram 10: Log In Profile.</i>	11
<i>Code 6: Session Storage</i>	11
Recovery Password Page	12
<i>Diagram 11: Recovery Password Page</i>	12
<i>Diagram 12: OTP Service Page</i>	12
<i>Code 7: generateOTPcode function</i>	13
<i>Code 8: Check OTP code format and Compare with User Input</i>	13
<i>Diagram 13: Result OTP Code Service</i>	14
<i>Diagram 14: Result OTP Code send via email.</i>	14
<i>Diagram 15: Reset New Password Page</i>	15
<i>Diagram 16: Results after changing the new password</i>	15
<i>Diagram 17: Results after changing the new password in Database</i>	15
3.0 List the potential vulnerabilities addressed in the future	16
Injection Attacks:	16
Authentication and Session Management:	16

Sensitive Data Handling:	16
Cross-Origin Resource Sharing (CORS):	16
Input Validation and Data Sanitization:	16
Error Handling:	17
Secure Communication:	17
4.0 Hashing algorithm (with salt)	17
<i>Code 9: CreateUser function with hashing and salt implementation</i>	17
<i>Code 10: verifyUser function with hashing and salt implementation</i>	18
<i>Diagram 18: Results password hashing</i>	18
5.0 Implementation Of Security measurements against SQL injection	19
Prepared Statements:	19
Parameterized Queries:	19
Error Handling:	19
Input Validation:	20
Session Management:	20
6.0 List of the countermeasures against SQL injection attack.....	21
Parameterized Queries:	21
Server-Side Input Validation:	21
Hashed Password Storage:	21
Session Management:	21
Error Handling:	21
Secure OTP Generation:	21
OTP Delivery:	21
OTP Validation:	21

1.0 Introduction

Introducing a Simple Password Manager System, a comprehensive tool designed to streamline password management for users across the digital landscape. Built upon a robust foundation of HTML, AngularJS, Mojolicious (Perl), and MySQL, this system offers a seamless and secure experience for users to safeguard their credentials.

Core of the system:

- **User Registration (Register):** This system allows users to create an account securely, using only username, email and password. After users register into the system they will receive an email to confirm that their registration has been successful.
- **Password Recovery (Forgot Password):** If the password is forgotten, this system offers a reliable recovery mechanism to restore access to the user's account. By leveraging security protocols like the One-Time-Password (OTP) service via the user's registered email within the system, users can reset their passwords.
- **Login to the System (Login):** Seamless access to the password manager system is facilitated through an intuitive login interface. With hash algorithm encryption protocol on user password in database.
- **OTP Service:** Enhancing security further, this system incorporates a robust OTP (One-Time Password) service. By implementing multi-factor authentication, it has an extra layer of protection to user accounts, mitigating the risk of unauthorized access.

2.0 Security Implementation

Registration Page (Sign Up):

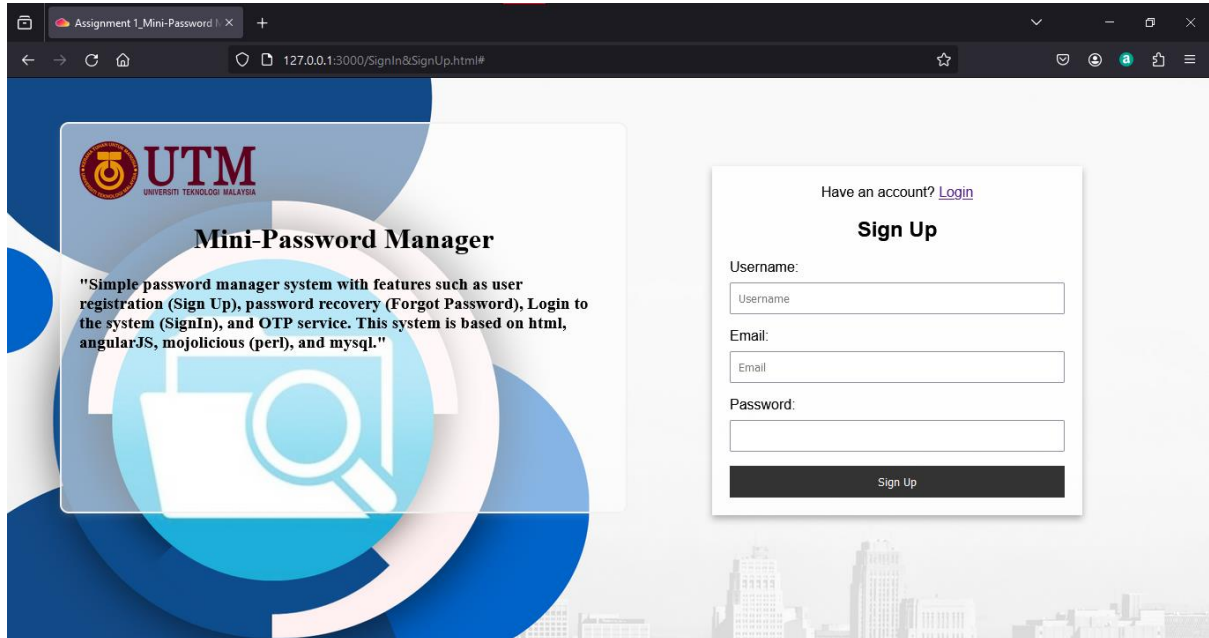


Diagram 1: Registration Page (Sign Up)

```
<!-- Sign Up page -->
<div id="signup" class="container" ng-show="oprSignUp"
  style="margin-left: 800px;margin-top: 100px;box-shadow: 0 4px 8px rgba(0, 0, 0, 0.3);">
  <span>Have an account? <a href="#" ng-click="oprSignUp = false; oprSignIn = true">Login</a></span>
  <h2 style="align-items: center">Sign Up</h2>
  <form>
    <label>Username:</label>
    <input type="text" placeholder="Username" size="50" ng-model="user.username" required />

    <label>Email:</label>
    <input type="text" placeholder="Email" ng-model="user.email" required />

    <label>Password:</label>
    <input type="password" id="newPassword" name="newPassword" ng-model="user.password" required />

    <button type="button" ng-click="CheckUsers()">Sign Up</button>
  </form>
</div>
```

Code 1: Sign Up page - SignIn&SignUp.html

Input Validation: Implement a basic input validation by marking the username, email, and password fields as required (**required** attribute). This ensures that these fields must be filled out before the form can be submitted, which helps prevent incomplete submissions

Cross-Site Scripting (XSS) Prevention: Use AngularJS (**ng-click**, **ng-model**) for client-side functionality. AngularJS has built-in protections against XSS attacks, such as data binding and context-aware escaping, which help prevent malicious script injection into the page.

```
$scope.CheckUsers = function () {  
    if (!$scope.isValidUsername($scope.user.username)) {  
        alert("Please enter a username with only characters.");  
        return;  
    } else if (!$scope.isValidEmail($scope.user.email)) {  
        alert("Please enter a valid email address.");  
        return;  
    } else if (!$scope.isStrongPassword($scope.user.password)) {  
        alert("Please enter a strong password.");  
        return;  
    } else {  
        $scope.CheckUsersz();  
    }  
};
```

Code 2: Check username, email and password

```
// Function to check if the password is strong  
$scope.isStrongPassword = function (password) {  
    // Example: Password should be at least 8 characters long, with at least one uppercase letter, one lowercase letter, one digit, and one special character  
    var passwordRegex = /^(?=.*[a-z])(?=.*[A-Z])(?=.*[@!#$%^&*()_+]{1};?/?>.<,'\"=-\\[\\]\\\\/]{8,})$/;  
    return passwordRegex.test(password);  
};  
  
// Function to check if the email is valid  
$scope.isValidEmail = function (email) {  
    // Example: Using a simple regex for email validation  
    var emailRegex = /^[^\\s@]+@[^\\s@]+\\.([^\\s@]+)$/;  
    return emailRegex.test(email);  
};  
  
// Function to check if the username contains only characters  
$scope.isValidUsername = function (username) {  
    // Allow only alphabetical characters  
    var usernameRegex = /^[A-Za-z]+$/;  
    return usernameRegex.test(username);  
};
```

Code 3: Check username, email and password functions

Input Validation: The **CheckUsers** function checks the validity of the username, email, and password entered by the user before proceeding. It ensures that the data entered by the user meets certain criteria, such as having a valid email format, containing only alphabetic characters for the username, and meeting the criteria for a strong password.

Password Strength Checking: The **isStrongPassword** function checks if the password meets certain criteria for strength. It ensures that the password is at least 8 characters long

and contains at least one uppercase letter, one lowercase letter, one digit, and one special character. This helps enforce stronger password policies, making it more difficult for attackers to guess or brute-force passwords.

Email Validation: The `isValidEmail` function validates the format of the email address entered by the user using a simple regular expression. This helps ensure that only properly formatted email addresses are accepted, reducing the risk of invalid or malicious email addresses being submitted.

Username Validation: The `isValidUsername` function ensures that the username entered by the user contains only alphabetical characters. This helps prevent the submission of usernames containing special characters or numbers, which could potentially be used for malicious purposes or to exploit vulnerabilities.

Client-Side Feedback: The code provides feedback to the user through alert messages in case the entered data doesn't meet the specified criteria. This helps improve user experience by guiding them on how to correct their input.

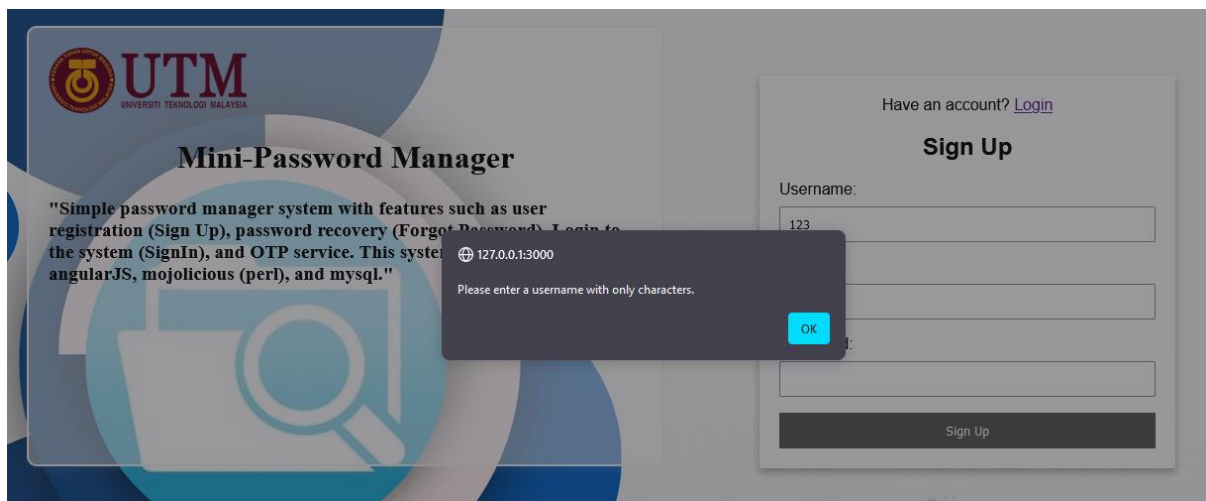


Diagram 2: Result for invalid username

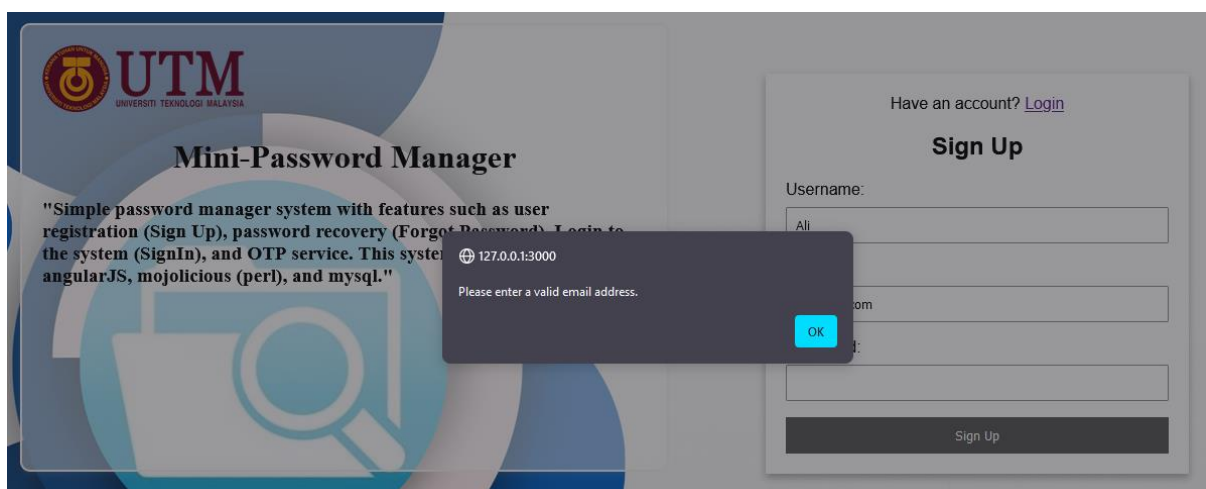


Diagram 3: Result for invalid email

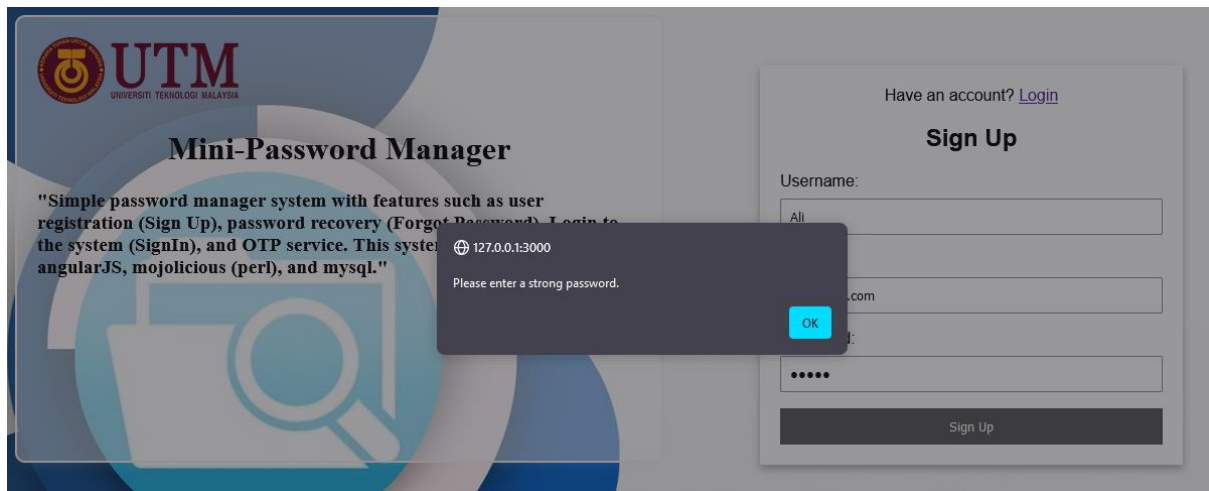


Diagram 4: Result for weak password.

```
$scope.CheckUsersz = function () {
  let Dats = $httpParamSerializer({
    username: $scope.user.username,
    email: $scope.user.email,
  });

  $http.get("http://localhost:3000/checkUser?" + Dats).then(
    function (response) {
      $scope.mew = response.data;
      // console.log("How many? " + $scope.mew);
      if ($scope.mew >= 1) {
        alert("Username or Email already exists");
      } else {
        $scope.createUser();
      }
    },

    function (response) {
      alert("Invalid User");
    }
  );
};
```

Code 4: Check if the email or username already exists in the database

Client-Side Communication: The AngularJS function **CheckUsersz** sends a GET request to the server to check if the entered username or email already exists in the database before

creating a new user. This helps prevent duplicate accounts and enhances the overall security of the system.

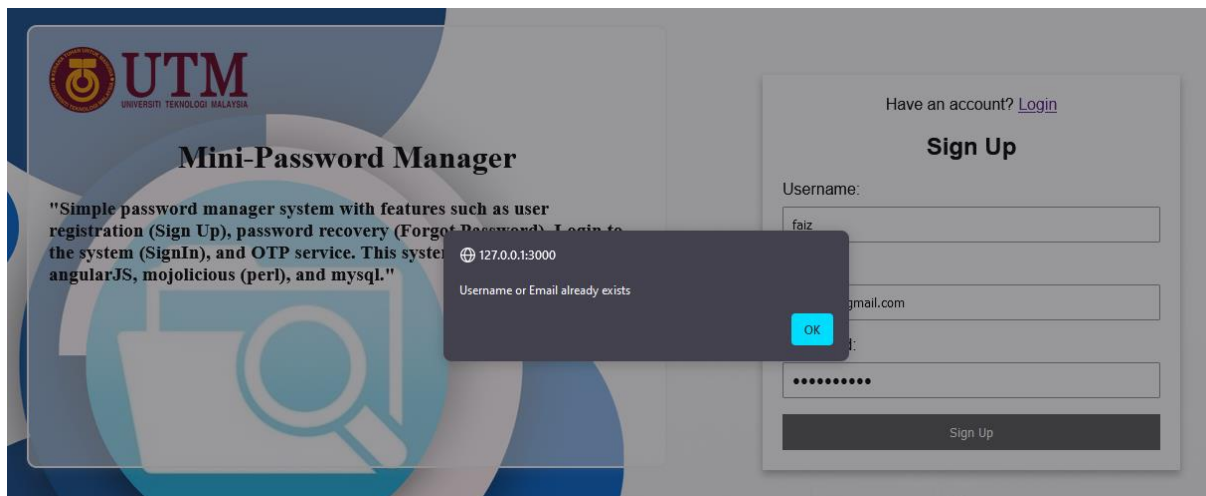


Diagram 5: Result for duplicate accounts.

```
$scope.NewRegister = function () {
  var emailData = {
    email: $scope.user.email,
    subject: "Registration",
    text: "Congratulations, Registration Successful",
  };

  $http
    .post("http://localhost:3001/send-email", emailData)
    .then(function (response) {
      alert("Check Your Email");
    })
    .catch(function (error) {
      alert("Error registration");
      console.error(error);
    });
};
```

Code 5: Send an email to the user, if successfully registered.

Verification: Email Verification Code ensures the security of the email address provided during registration is valid and belongs to the user.

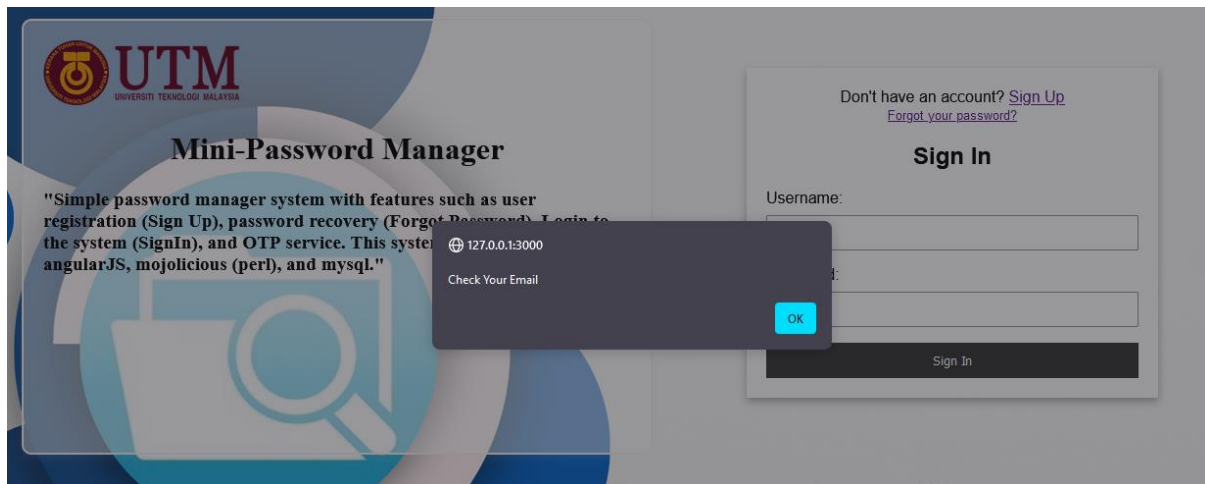


Diagram 6: Confirmation by email to those who successfully register.

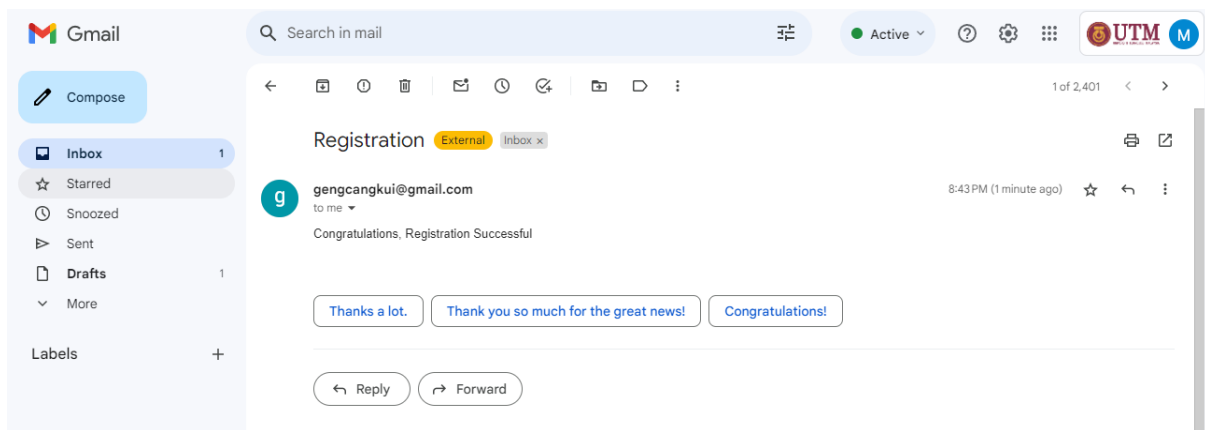


Diagram 7: Confirmation by email

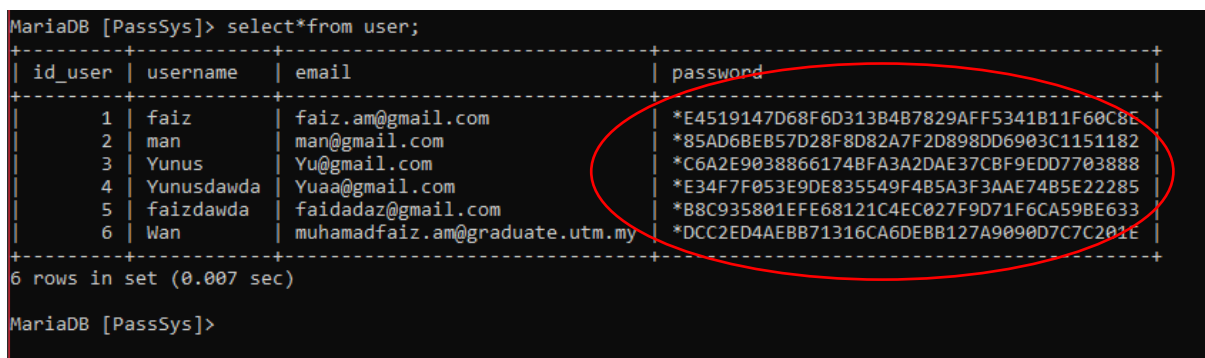


Diagram 8: Database validation (Password Hashing)

Password Hashing: The passwords stored in the database are hashed. Hashing passwords is a crucial security measure because it protects user passwords in case of a data breach. Hashing irreversibly transforms the passwords into a fixed-length string of characters, making it computationally infeasible for attackers to retrieve the original passwords from the hash values

Sign In Page (Log In):

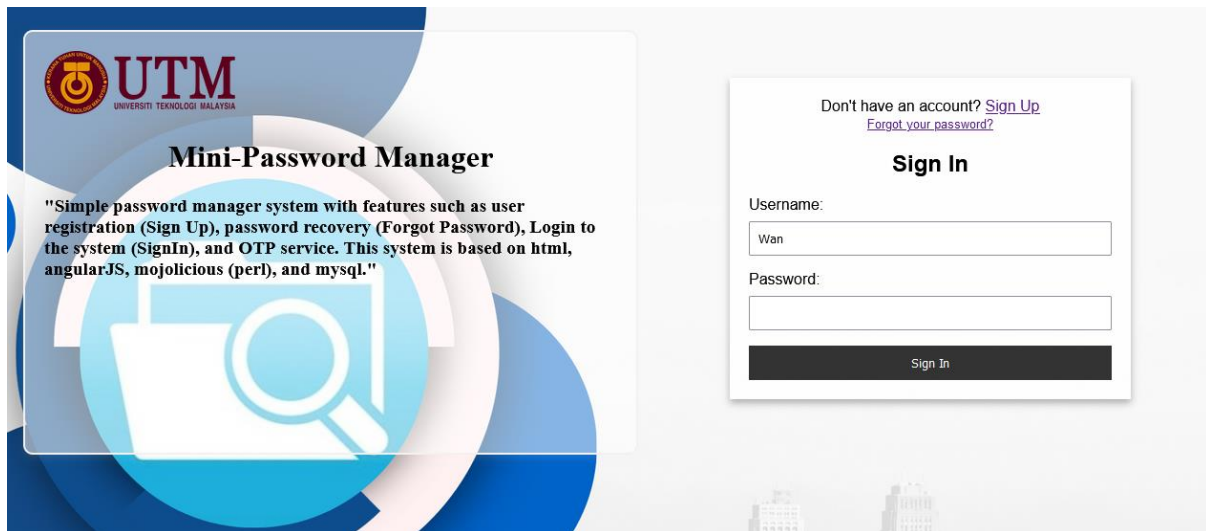


Diagram 9: Sign In (Log In) Page



Diagram 10: Log In Profile.

```
var app = angular.module("LoginPage", []);

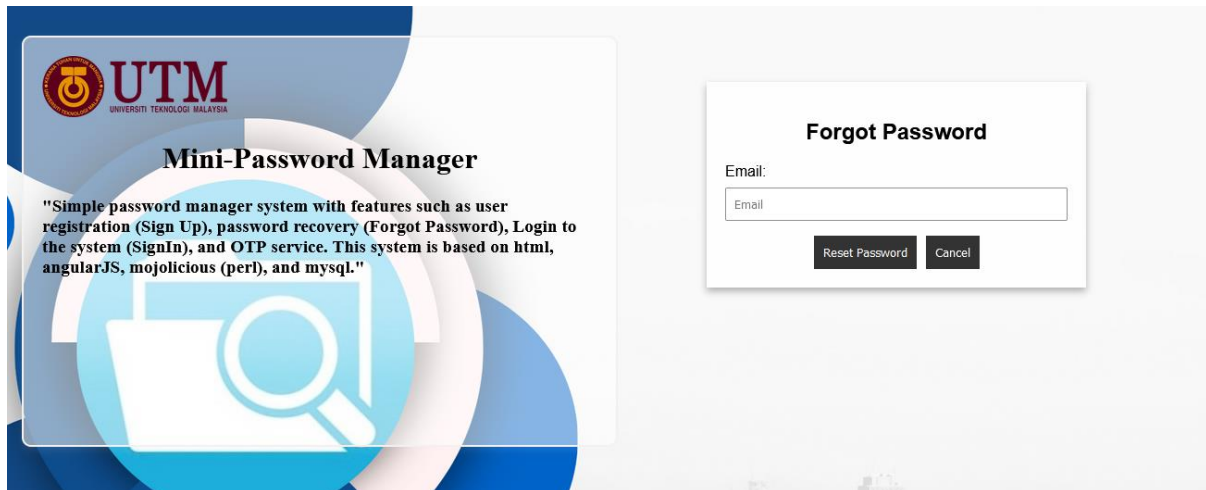
app.controller(
  "loginSys",
  function ($scope, $httpParamSerializer, $http, $window) {
    var storedData = sessionStorage.getItem("myData");
    $scope.studs = JSON.parse(storedData);
    console.log("Data in sessionStorage:", $scope.studs);

    $scope.logout = function () {
      // Clear session storage or perform logout actions
      sessionStorage.removeItem("myData"); // Remove the stored user data
      // Optionally, redirect the user to the login page or any other page
      $window.location.href = "/SignIn&SignUp.html"; // Redirect to the login page
    };
  }
);
```

Code 6: Session Storage

Session Storage: The code uses sessionStorage to store user data ("**myData**") retrieved from the session. sessionStorage is a client-side storage mechanism that persists data only for the duration of the page session. Storing sensitive user data in sessionStorage rather than localStorage helps mitigate the risk of data theft through cross-site scripting (XSS) attacks.

Recovery Password Page



Mini-Password Manager

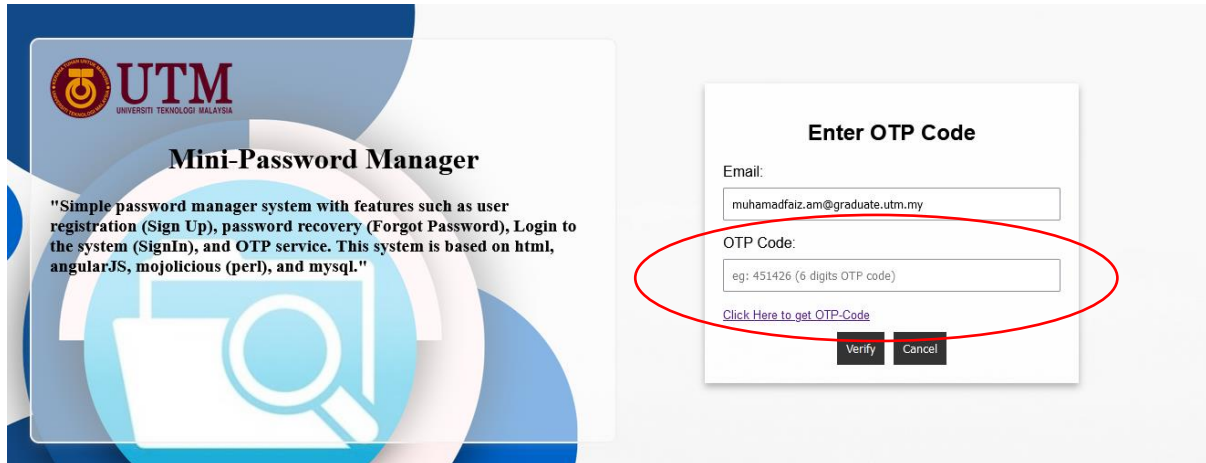
"Simple password manager system with features such as user registration (Sign Up), password recovery (Forgot Password), Login to the system (SignIn), and OTP service. This system is based on html, angularJS, mojolicious (perl), and mysql."

Forgot Password

Email:

Reset Password Cancel

Diagram 11: Recovery Password Page



Mini-Password Manager

"Simple password manager system with features such as user registration (Sign Up), password recovery (Forgot Password), Login to the system (SignIn), and OTP service. This system is based on html, angularJS, mojolicious (perl), and mysql."

Enter OTP Code

Email:

OTP Code:

[Click Here to get OTP-Code](#)

Verify Cancel

Diagram 12: OTP Service Page

```

$scope.generateOTPcode = function () {
    const digits = "0123456789";
    let OTP = "";
    for (let i = 0; i < 6; i++) {
        OTP += digits[Math.floor(Math.random() * 10)];
    }
    return OTP;
};

$scope.isValidOTP = function (otp) {
    var otpRegex = /^\d{6}$/;
    return otpRegex.test(otp);
};

```

Code 7: generateOTPcode function

Secure Random Number Generation: The `generateOTPcode` function generates a random 6-digit OTP code using JavaScript's built-in `Math.random()` function. While this method is suitable for generating random numbers, it's essential to ensure that the generated OTP code is sufficiently random and unpredictable to prevent guessing or brute-force attacks.

```

$scope.isValidOTP = function (otp) {
    var otpRegex = /^\d{6}$/;
    return otpRegex.test(otp);
};

$scope.CheckOTPcode = function () {
    // console.log("OTPP? " + $scope.otppcode);
    if (!$scope.isValidOTP($scope.otppcode)) {
        alert("Please enter a valid OTP-CODE (6 Digits OTP-Code)");
        return;
    } else if ($scope.otppcode == $scope.generatedOTP) {
        alert("OTP-Code is valid");
        $scope.ResetPassword = true;
        $scope.SendOTPcode = false;
    } else {
        alert("Invalid OTP-Code, Enter again!");
    }
}

```

Code 8: Check OTP code format and Compare with User Input

OTP Code Validation: The `isValidOTP` function validates whether the entered OTP code matches the expected format (exactly 6 digits). This helps prevent users from entering invalid OTP codes and ensures that only correctly formatted OTP codes are accepted.

Input Sanitization: The code checks whether the entered OTP code matches the expected format using a regular expression (`otpRegex`). Regular expressions are a common tool for validating input format and preventing injection attacks. By enforcing a strict format for OTP codes (6 digits), the code helps mitigate the risk of injection attacks or manipulation of input data.

Client-Side Feedback: The code provides feedback to the user through alert messages in case of invalid or valid OTP codes. Providing clear and informative feedback to users helps improve user experience and ensures that users understand the reason for any errors or validation failures.

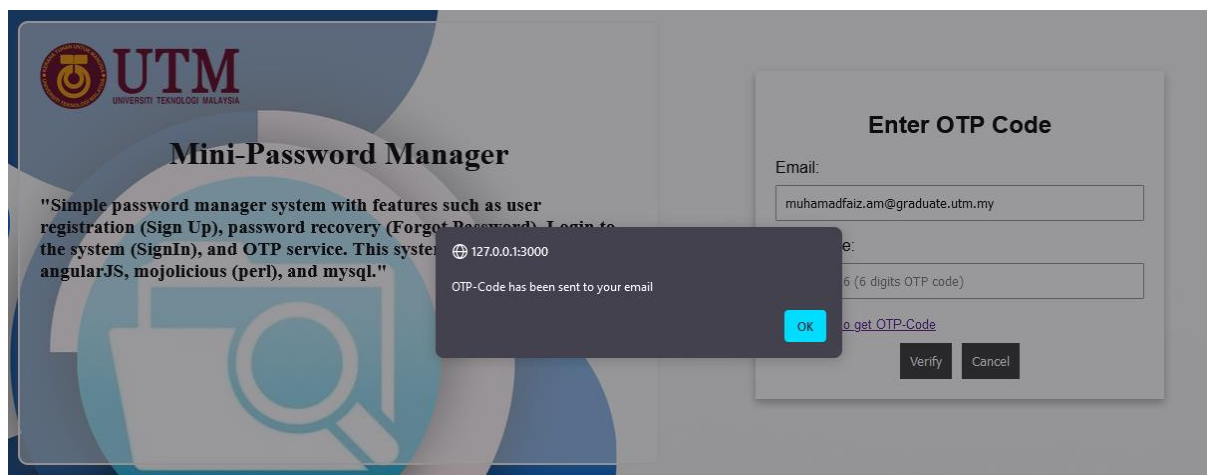


Diagram 13: Result OTP Code Service

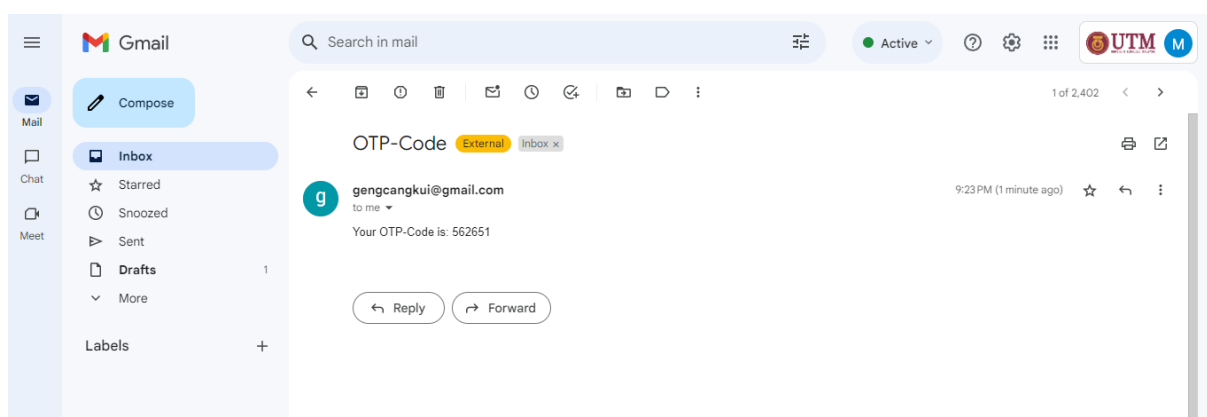


Diagram 14: Result OTP Code send via email

Mini-Password Manager

"Simple password manager system with features such as user registration (Sign Up), password recovery (Forgot Password), Login to the system (SignIn), and OTP service. This system is based on html, angularJS, mojolicious (perl), and mysql."

Reset Password

Email:

New Password:

Diagram 15: Reset New Password Page

(If the OTP Code is entered by the user correctly, the Reset Password page will appear)

Mini-Password Manager

"Simple password manager system with features such as user registration (Sign Up), password recovery (Forgot Password), Login to the system (SignIn), and OTP service. This system is based on html, angularJS, mojolicious (perl), and mysql."

Reset Password

Email:

New Password:

127.0.0.1:3000

Your Password has been updated, Try to Login again!

Diagram 16: Results after changing the new password

```
MariaDB [PassSys]> select*from user;
```

id_user	username	email	password
1	faiz	faiz.am@gmail.com	*E4519147D68F6D313B4B7829AFF5341B11F60C8E
2	man	man@gmail.com	*85AD68EB57D28F8D82A7F2D898DD6903C1151182
3	Yunus	Yu@gmail.com	*C6A2E9038866174BFA3A2DAE37CBF9EDD7703888
4	Yunusdawda	Yuaa@gmail.com	*E34F7F053E9DE835549F4B5A3F3AAE74B5E22285
5	faizdawda	faidadz@gmail.com	*B8C935801EFF68121C4FC027F9D71F6CA59BE633
6	Wan	muhamadfaiz.am@graduate.utm.my	*DCC2ED4AEBB71316CA6DEBB127A9090D7C7C201E

6 rows in set (0.007 sec)

```
MariaDB [PassSys]> select*from user;
```

id_user	username	email	password
1	faiz	faiz.am@gmail.com	*E4519147D68F6D313B4B7829AFF5341B11F60C8E
2	man	man@gmail.com	*85AD68EB57D28F8D82A7F2D898DD6903C1151182
3	Yunus	Yu@gmail.com	*C6A2E9038866174BFA3A2DAE37CBF9EDD7703888
4	Yunusdawda	Yuaa@gmail.com	*E34F7F053E9DE835549F4B5A3F3AAE74B5E22285
5	faizdawda	faidadz@gmail.com	*B8C935801EFF68121C4FC027F9D71F6CA59BE633
6	Wan	muhamadfaiz.am@graduate.utm.my	*0F38374B1715A6C22589B809F5CE4204A8EAS79

6 rows in set (0.006 sec)

Diagram 17: Results after changing the new password in Database

3.0 List the potential vulnerabilities addressed in the future

Injection Attacks:

- **SQL Injection:** The Perl code interacts with the database using SQL queries without proper parameterization or input validation, making it vulnerable to SQL injection attacks.
- **Client-Side Script Injection:** The AngularJS code renders dynamic content without sanitization, potentially exposing the application to cross-site scripting (XSS) attacks if user-supplied data is not properly escaped.

Authentication and Session Management:

- **Lack of Proper Authentication:** The code does not demonstrate a comprehensive authentication mechanism. Without proper authentication, unauthorized users may access restricted functionalities or data.
- **Session Management:** The AngularJS code uses sessionStorage for storing user data, which may be vulnerable to session fixation or session hijacking attacks if not implemented securely.

Sensitive Data Handling:

- **Email Transmission:** The code sends sensitive information (such as OTP codes) over HTTP, which is insecure. Using HTTPS for secure communication is essential to protect sensitive data during transmission.

Cross-Origin Resource Sharing (CORS):

- **Lack of Proper CORS Configuration:** The Perl code includes an Access-Control-Allow-Origin header with a wildcard value (*), allowing cross-origin requests from any domain. This may expose the application to Cross-Origin Resource Sharing (CORS) vulnerabilities if not properly configured.

Input Validation and Data Sanitization:

- **Insufficient Input Validation:** The code lacks comprehensive input validation and data sanitization mechanisms, making it vulnerable to various input-based attacks such as XSS, CSRF, and command injection.
- **Lack of Output Encoding:** User-supplied data is not properly encoded before being rendered in the HTML response, which increases the risk of XSS vulnerabilities.

Error Handling:

- Lack of Robust Error Handling: While the code includes basic error handling, it may not adequately handle all possible error scenarios, leading to potential information leakage or denial-of-service vulnerabilities.

Secure Communication:

- Insecure Communication Protocols: The code communicates over HTTP instead of HTTPS, potentially exposing sensitive data to interception or tampering by attackers.

4.0 Hashing algorithm (with salt)

```
use strict;
use warnings;
use Digest::SHA qw(sha256_hex);
use Data::Entropy::Algorithms qw(rand_bits);
use MIME::Base64 qw(encode_base64url);

sub createUser {
    my ($this, $dbh) = @_;

    # Generate a random salt
    my $salt = encode_base64url(rand_bits(128));

    # Hash the password with the salt using SHA-256
    my $hashed_password = sha256_hex($this->{'password'} . $salt);

    my $sth = undef;

    if ($sth = $dbh->prepare('INSERT INTO user(username, email, password, salt) values (?, ?, ?, ?)')) {
        if ($sth->execute($this->{'username'}, $this->{'email'}, $hashed_password, $salt)) {
            print "Success create new user...\n";
        } else {
            print "Error: " . $dbh->errstr() . "\n";
        }
    } else {
        print "Error: " . $dbh->errstr() . "\n";
    }
}
```

Code 9: CreateUser function with hashing and salt implementation

```

sub verifyUser {
    my ($username, $password, $dbh) = @_;

    my $sth = $dbh->prepare('SELECT password, salt FROM user WHERE username = ?');
    $sth->execute($username);
    my ($stored_password, $salt) = $sth->fetchrow_array();

    if ($stored_password && $salt) {
        # Hash the input password with the retrieved salt
        my $hashed_input_password = sha256_hex($password . $salt);

        # Compare the hashed input password with the stored hashed password
        if ($stored_password eq $hashed_input_password) {
            return 1; # Password is verified
        } else {
            return 0; # Password verification failed
        }
    } else {
        return 0; # User not found or password not stored
    }
}

```

Code 10: verifyUser function with hashing and salt implementation

The **createUser** function generates a random salt using **rand_bits**, then hashes the password concatenated with the salt using SHA-256 before storing it in the database.

The **verifyUser** function retrieves the stored salt along with the hashed password from the database based on the provided username. Then it hashes the input password with the retrieved salt and compares it with the stored hashed password to verify the user's credentials.

```

MariaDB [PassSys]> select*from user;
+-----+-----+-----+-----+
| id_user | username | email | password |
+-----+-----+-----+-----+
| 1 | faiz | faiz.am@gmail.com | *E4519147D68F6D313B4B7829AFF5341B11F60C8E |
| 2 | man | man@gmail.com | *85AD68EB57D28F8D82A7F2D898DD6903C1151182 |
| 3 | Yunus | Yu@gmail.com | *C6A2E9038866174BFA3A2DAE37CBF9EDD7703888 |
| 4 | Yunusdawda | Yuaa@gmail.com | *E34F7F053E9DE835549F4B5A3F3AAE74B5E22285 |
| 5 | faizdawda | faidadaz@gmail.com | *B8C935801EFE68121C4EC027F9D71F6CA59BE633 |
| 6 | Wan | muhamadfaiz.am@graduate.utm.my | *0F38374B1715A6C22589B809F5CE4204A8EA5C79 |
+-----+-----+-----+-----+
6 rows in set (0.006 sec)

MariaDB [PassSys]>

```

Diagram 18: Results password hashing

5.0 Implementation of Security measurements against SQL injection

Prepared Statements:

Prepared statements are used throughout the code, which is a good practice for preventing SQL injection attacks. Prepared statements separate SQL logic from data, allowing parameters to be securely bound to the query without the risk of SQL injection.

Parameterized Queries:

The code utilizes parameterized queries where user inputs are passed as placeholders in the SQL query and later bound to the actual values. This prevents direct insertion of user input into the SQL query string, reducing the risk of SQL injection.

Example:

```
sub createUser {  
  
    my $this = shift @_;  
    my $dbh = shift @_;  
    my $sth = undef;  
  
    if ($sth = $dbh->prepare('INSERT INTO user(username, email, password) values (?, ?, password(?))')) {  
        if ($sth->execute($this->{'username'}, $this->{'email'}, $this->{'password'})) {  
            print "Success create new user...\n";  
        } else {  
            print "Error: $dbh->errstr()\n";  
        }  
    } else {  
        print "Error: $dbh->errstr()\n";  
    }  
}
```

Error Handling:

Proper error handling is crucial to detect and handle any potential SQL execution errors gracefully. Handling errors securely prevents detailed error messages from being exposed to users, which could inadvertently leak sensitive information or aid attackers in crafting SQL injection payloads.

Example:

```
if ($sth = $dbh->prepare('SELECT COUNT(*) FROM user WHERE email = ? OR username = ?')) {  
  
    if ($sth->execute($email, $username)) {  
        my ($count) = $sth->fetchrow_array;  
        $sth->finish();  
        return $count;  
    } else {  
        # SQL execution error  
        return JSON->new->pretty->encode({ error => $dbh->errstr() });  
    }  
}  
else {  
    print "Error: $dbh->errstr()\n";  
}
```

Input Validation:

Validate user input to ensure that it meets expected criteria before using it in SQL queries. This helps prevent injection attacks by rejecting invalid or malicious input.

Example:

```
get '/createUser' => sub ($c) {  
  my $username = $c->param('username');  
  my $email = $c->param('email');  
  my $password = $c->param('password');  
  
  my $s = new ModelUser();  
  $s->setUser($username, $email, $password);  
  $s->createUser($dbh);  
  
  $c->res->headers->header( 'Access-Control-Allow-Origin' => '*' );  
  
  $c->render(text => "Create row for $username - $email - $password");  
};
```

Session Management:

Ensure that session data, such as usernames, is properly sanitized and validated before using it in SQL queries. Avoid directly using session data in SQL queries without validation to prevent injection attacks.

Example:

```
var app = angular.module("LoginPage", []);  
  
app.controller(  
  "loginSys",  
  function ($scope, $httpParamSerializer, $http, $window) {  
    var storedData = sessionStorage.getItem("myData");  
    $scope.studs = JSON.parse(storedData);  
    console.log("Data in sessionStorage:", $scope.studs);  
  
    $scope.logout = function () {  
      // Clear session storage or perform logout actions  
      sessionStorage.removeItem("myData"); // Remove the stored user data  
      // Optionally, redirect the user to the login page or any other page  
      $window.location.href = "/SignIn&SignUp.html"; // Redirect to the login page  
    };  
  }  
);
```

6.0 List of the countermeasures against SQL injection attack

Parameterized Queries:

In the Perl code, parameterized queries are used to interact with the database. Instead of directly embedding user inputs into SQL queries, placeholders are used, and user inputs are passed as parameters. This prevents SQL injection attacks by ensuring that user inputs are treated as data rather than executable SQL code.

Server-Side Input Validation:

Input validation is performed on the server-side using regular expressions to ensure that user inputs adhere to expected formats. For example, in the Perl code, regex patterns are used to validate email addresses and usernames before executing database queries.

Hashed Password Storage:

Passwords are stored securely using a hashing algorithm. Instead of storing plaintext passwords, passwords are hashed before being stored in the database. This adds an additional layer of security and prevents attackers from obtaining plaintext passwords even if they manage to access the database.

Session Management:

Session management is implemented to authenticate users and prevent unauthorized access to protected resources. User sessions are managed securely, and session tokens are validated on each request to ensure that authenticated users are authorized to access the requested resources.

Error Handling:

Proper error handling mechanisms are implemented to handle unexpected errors gracefully. Detailed error messages are avoided to prevent leakage of sensitive information about the system or database structure, which could be exploited by attackers.

Secure OTP Generation:

OTPs are generated securely using cryptographically strong random number generators. In the provided code, a secure OTP generation function is implemented using a combination of digits and randomness to generate a unique one-time password for each authentication attempt.

OTP Delivery:

OTPs are delivered securely to the user via a trusted communication channel, such as email. In the provided code, OTPs are sent to the user's email address using a server-side email service. It's essential to ensure that the OTP delivery mechanism is secure and resistant to interception or tampering by unauthorized parties.

OTP Validation:

OTPs are validated securely on the server-side to ensure that the OTP entered by the user matches the expected OTP generated by the system. In the provided code, the entered OTP is compared with the generated OTP to authenticate the user. It's crucial to validate OTPs securely to prevent brute-force attacks or replay attacks.