BROKEN AUTHENTICATION AND SESSION MANAGEMENT

BROKEN AUTHENTICATION AND SESSION MANAGEMENT

This happens to be one of the most **highly** ranked security risks as per the Open Web Application Security Project

BROKEN AUTHENTICATION AND SESSION MANAGEMENT

This covers a **wide range** of issues arising due to compromised passwords, keys, session tokens and enabling attackers to **impersonate** other users on a website

BROKEN AUTHENTICATION AND SESSION MANAGEMENT

The broad categories of issues are in

1. Credential Management
2. Session Management
3. The rest (whatever doesn't fit in the first 2 categories)

BROKEN AUTHENTICATION AND SESSION MANAGEMENT

1. Credential Management
2. Session Management
3. The rest (whatever doesn't fit in the first 2 categories)

A number of vulnerabilities exist in each of these categories and there are good practices which should be followed - we'll cover it all in detail

CREDENTIAL MANAGEMENT

CREDENTIAL MANAGEMENT

Credential management deals with passwords - the **kind** of password used, password **storage, retrieval, reset** etc

CREDENTIAL MANAGEMENT

Let's look at some of these:

Password Strength

Password Use

Password In Transit

Password Storage

Password Recovery

CREDENTIAL MANAGEMENT

Password Strength

Passwords should have a **minimum** strength and complexity requirement

lowercase + uppercase alphabets + numbers + special characters

CREDENTIAL MANAGEMENT

Password Strength

minimum strength and complexity requirement

Users should be forced to change their passwords periodically

Old passwords should not be reused

CREDENTIAL MANAGEMENT

Password Strength

minimum strength and complexity requirement

change their passwords periodically

passwords should not be reused

Simple passwords are often hacked using something called the dictionary attack

CREDENTIAL MANAGEMENT

Password Strength

minimum strength and complexity requirement

change their passwords periodically

passwords should not be **reused**

Simple passwords are often hacked using something called the **dictionary** attack

This involves trying to crack passwords by trying **millions** of words as from a dictionary

CREDENTIAL MANAGEMENT

Password Strength

minimum strength and complexity requirement

change their passwords periodically

passwords should not be **reused**

Simple passwords are often hacked using something called the **dictionary** attack

This involves trying to crack passwords by trying **millions** of words as from a dictionary

There are large scale data dumps of passwords out on the internet - these are lists of **commonly known** passwords

CREDENTIAL MANAGEMENT

minimum strength and complexity requirement

change their passwords periodically

passwords should not be reused

Password Strength

Simple passwords are often hacked using something called the dictionary attack

This involves trying to crack passwords by trying millions of words as from a dictionary

There are large scale data dumps of passwords out on the internet - these are lists of commonly known passwords

Any previously seen password is at risk!

CREDENTIAL MANAGEMENT

Password Strength

minimum strength and complexity requirement

change their passwords periodically

passwords should not be reused

CREDENTIAL MANAGEMENT

Let's look at some of these:

✓ Password Strength

Password Use

Password In Transit

Password Storage

Password Recovery

CREDENTIAL MANAGEMENT

Password Use

Specify a **limit** on the number of login attempts a user can make per unit of time

CREDENTIAL MANAGEMENT

Password Use

limit on the number of login attempts

The wrong user name and password message should be **generic** - do not give away more information than needed

CREDENTIAL MANAGEMENT

Password Use

limit on the number of login attempts

message should be generic

Never log the password or even failed password attempts

CREDENTIAL MANAGEMENT

Password Use

limit on the number of login attempts

message should be generic

Never log the password

Give the user information about last login and failed attempts

CREDENTIAL MANAGEMENT

Password Use

limit on the number of login attempts

message should be generic

Never log the password

last login and failed attempts

CREDENTIAL MANAGEMENT

Let's look at some of these:

✓ Password Strength

✓ Password Use

Password In Transit

Password Storage

Password Recovery

CREDENTIAL MANAGEMENT

Password In Transit

The entire login transaction should be **encrypted over SSL** so it cannot be intercepted or sniffed

CREDENTIAL MANAGEMENT

Let's look at some of these:

✓ Password Strength

✓ Password Use

✓ Password In Transit

Password Storage

Password Recovery

CREDENTIAL MANAGEMENT

Password Storage

Passwords for user accounts have to be **stored** in some database to allow comparisons for login

CREDENTIAL MANAGEMENT

Password Storage

Passwords for user accounts have to be stored in some database to allow comparisons for login

Never store them in plain text!

CREDENTIAL MANAGEMENT

Password Storage

Never store them in plain text!

Stored passwords should always be **hashed** or **encrypted**

CREDENTIAL MANAGEMENT

Password Storage

hashed

Hashing is a type of algorithm which takes any size of data and converts it to a fixed length of data

CREDENTIAL MANAGEMENT

Password Storage

hashed

There are a few principles which should be followed for good hashes

CREDENTIAL MANAGEMENT

Password Storage

principles

hashed

It should be easy to generate a hash of a message

You cannot generate the original message from the hash

modifying the message modifies the hash

different messages have different hashes

CREDENTIAL MANAGEMENT

Password Storage

hashed or encrypted

Hashes are irreversible, you cannot get the original value back from the hash

CREDENTIAL MANAGEMENT

Password Storage

hashed or encrypted

Hashes are irreversible, you cannot get the original value back from the hash

You always go just one way - from **password to hash** to check for the right password when the user logs in

CREDENTIAL MANAGEMENT

Password Storage

hashed or **encrypted**

You typically use encryption when you want to go the other way - **get the password back** from the encrypted version

CREDENTIAL MANAGEMENT

Password Storage

hashed or **encrypted**

Say you want to be able to **retrieve** the **plaintext** password - to use it to log on to another system

SIGNING UP USERS

SIGNING UP USERS

Example10-CredentialMgmt-signup.php

SIGNING UP USERS

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

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>
<a href=“Example11-CredentialMgmt-login.php"> Already a member? Login </a>
```

SIGNING UP USERS

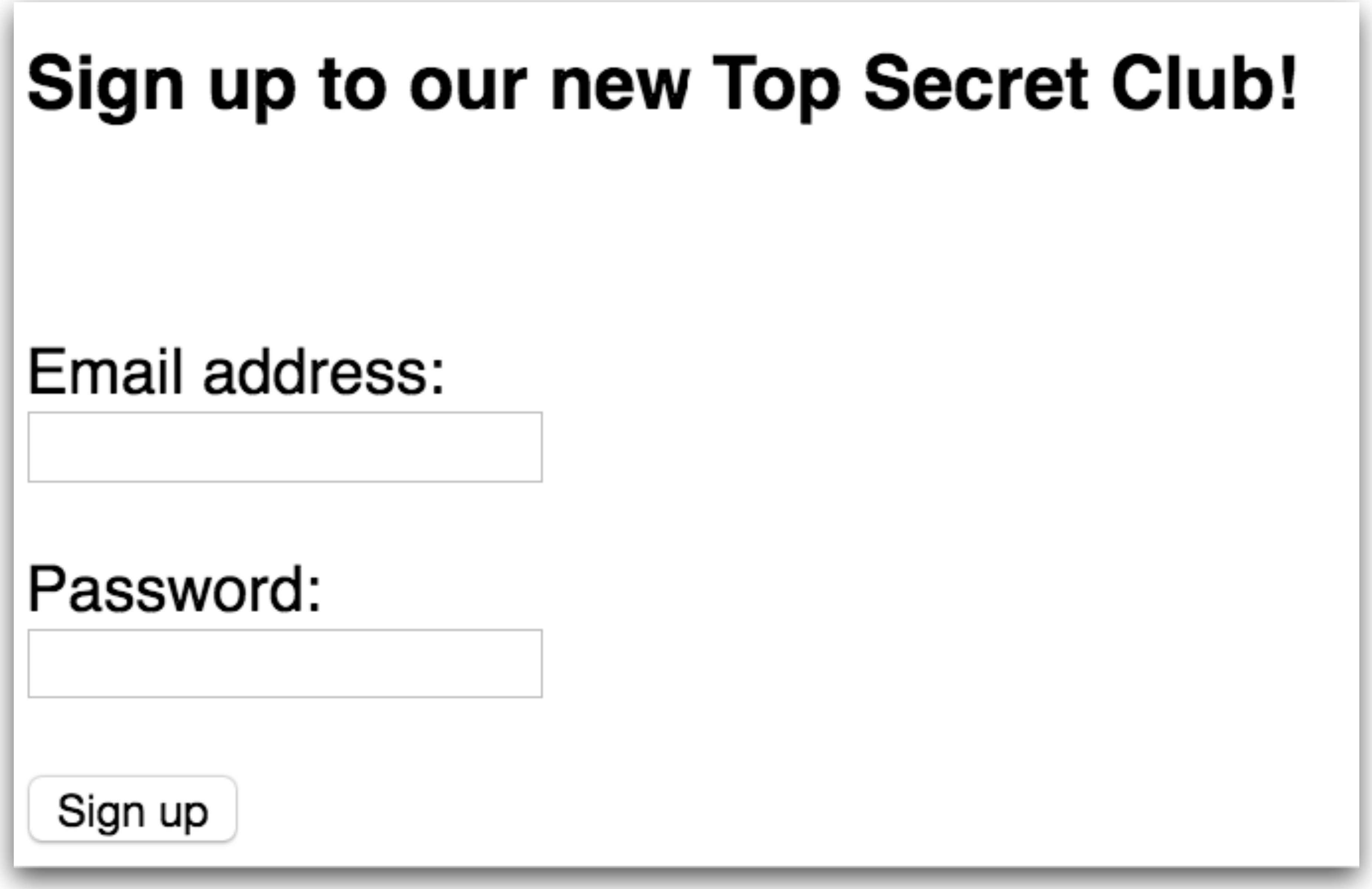
```
<h3> Sign up to our new Top Secret Club! </h3>
<form method="POST" action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]);?>">
  <span style="color: red"><?php echo $error_message;?></span>
  <br>
  <br>
  Email address:
  <br>
  <input type="text" name="user_email" maxlength="100">
  <br>
  <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>
<a href=“Example11-CredentialMgmt-login.php"> Already a member? Login </a>
```

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"]);?>">
  <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>
<a href="Example26-login.php"> Already a member? Login </a>
```



Sign up to our new Top Secret Club!

Email address:

Password:

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>
<a href="Example26-login.php"> Already a member? Login </a>
```

It's good practice to use the same names in the form as you did in the database - this really prevents errors - remembering which names go where is painful!

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>
<a href="Example26-login.php"> Already a member? Login </a>
```

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

SIGNING UP USERS

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

SIGNING UP USERS

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

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

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)) {
    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);
```

Sanitize the email and password inputs

SIGNING UP USERS

```
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) {
```

Try-catch deals with exceptions in code - **exceptions are thrown** by code when an error occurs in code

SIGNING UP USERS

```
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) {
```

Within the try block we simply insert the new user into the Users table

```
  }
}
```

SIGNING UP USERS

```
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) {
```

Do not store the password in plain-text!

SIGNING UP USERS

```
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) {

}
```

sha1 is an encryption algorithm - run this and store the encrypted form of the password!

SIGNING UP USERS

```
} 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';
  }
}
```

We enter the catch block if we encounter an error while adding the user to the table

SIGNING UP USERS

```
} 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';
   }
 }
```

If the **email already exists** in the table indicate that in the error message

SIGNING UP USERS

```
} 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';
   }
 }
```

Otherwise show a generic error message

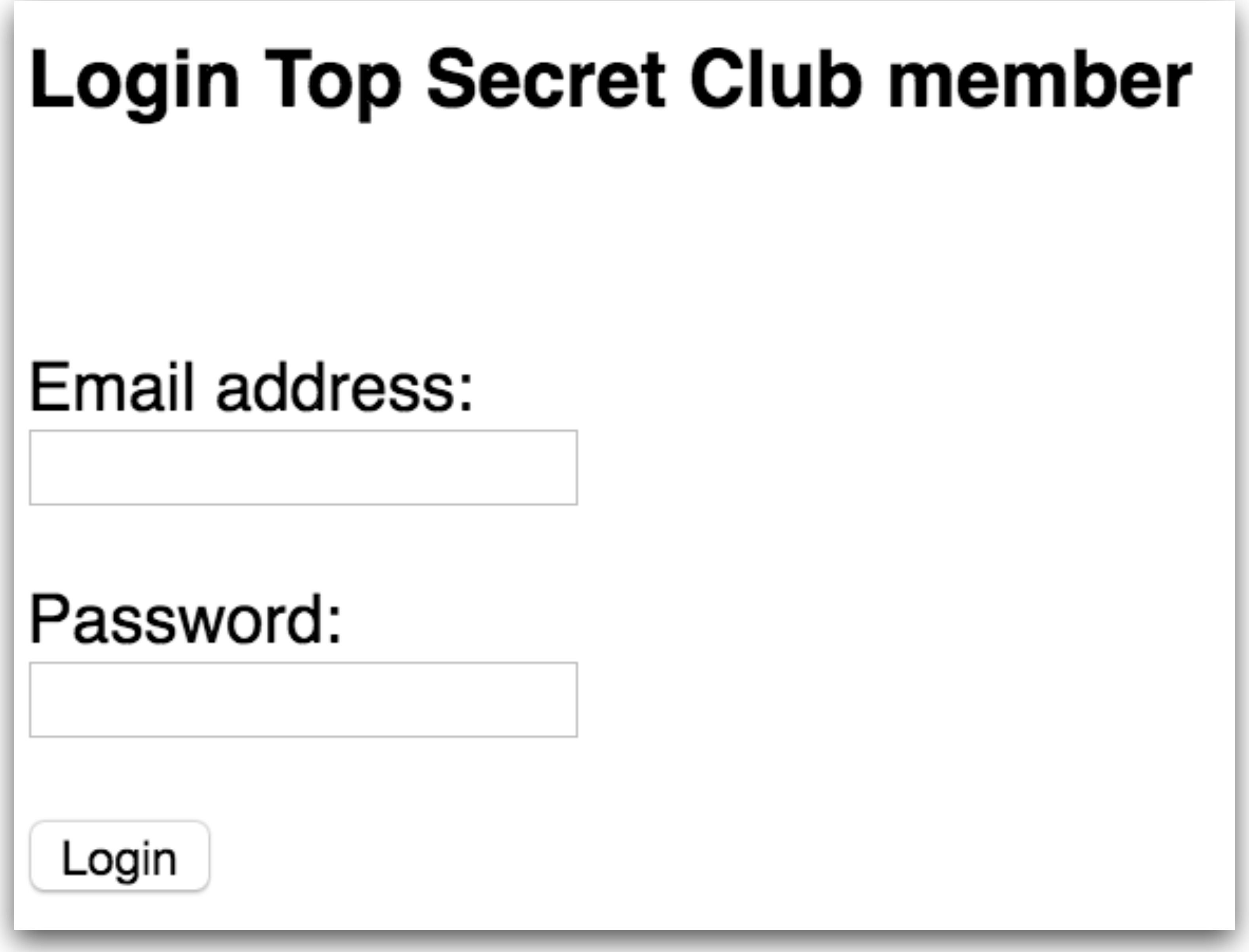
LOGIN AUTHENTICATED USERS

LOGIN AUTHENTICATED USERS

Example11-CredentialMgmt-login.php

Example11-CredentialMgmt-loginSuccess.php

LOGIN AUTHENTICATED USERS



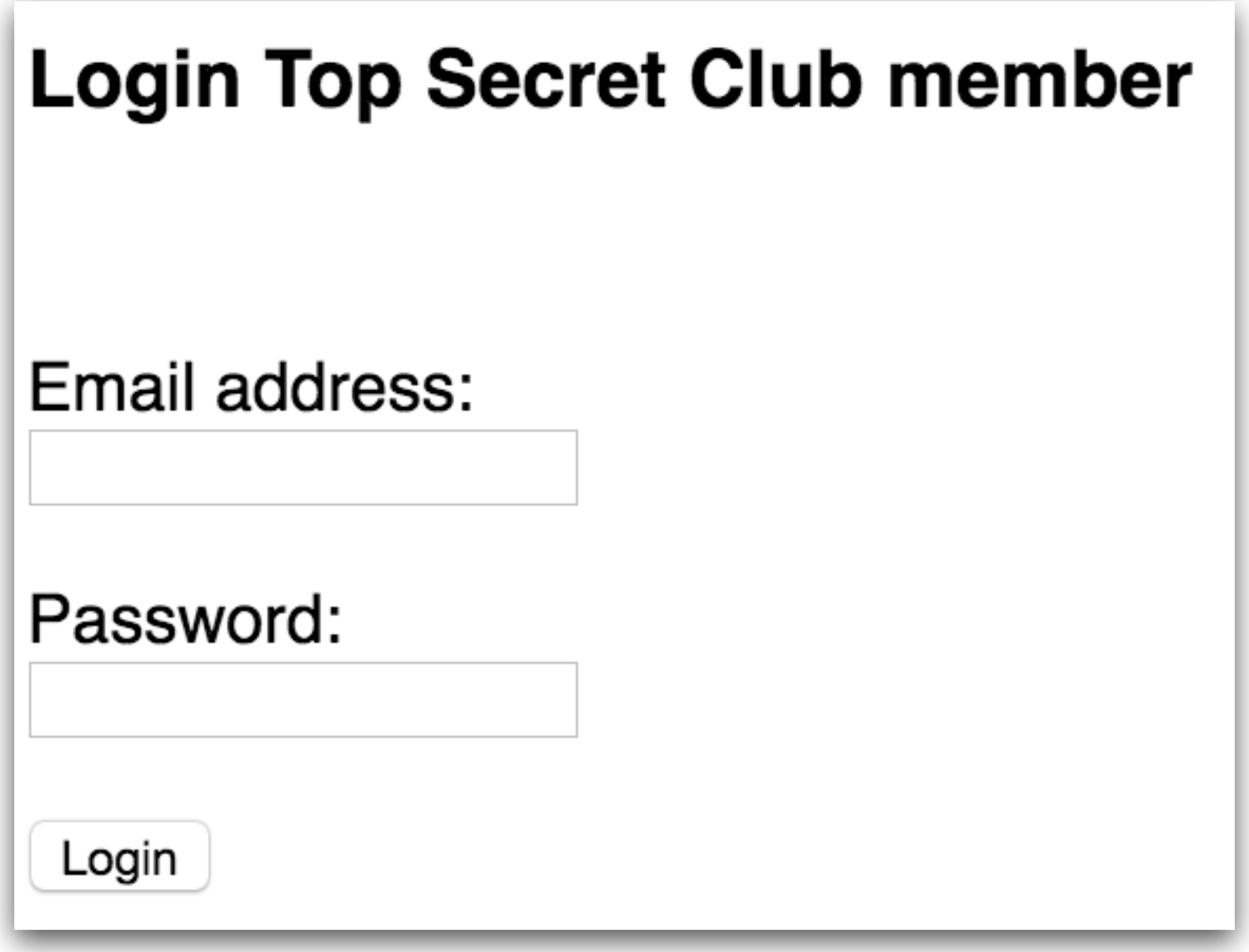
Login Top Secret Club member

Email address:

Password:

The login form looks similar to the sign up form - the button says login though

LOGIN AUTHENTICATED USERS



Login Top Secret Club member

Email address:

Password:

The form and the validation of inputs is very similar to the sign up process so we won't got over them again

LOGIN AUTHENTICATED USERS

```
$stmt = $conn->prepare(
  "SELECT user_id, user_email, user_password FROM Users WHERE user_email = ?"
);
$stmt->bind_param("s", $user_email);
$stmt->execute();

$stmt->bind_result($user_id_db, $user_email_db, $user_password_db);

$user_valid = false;
while ($stmt->fetch()) {
  if ($user_id_db) {
    // Check if the password hashes are the same.
    if (sha1($user_password) == $user_password_db) {
      $_SESSION['logged_in_user'] = $user_id_db;

      // clear out the output buffer
      while (ob_get_status()) {
        ob_end_clean();
      }

      header("Location: Example11-CredentialMgmt-loginSuccess.php");
   } else {
      $error_message = 'Wrong user name or password provided!';
    }
  } else {
    $error_message = 'Wrong user name or password provided';
  }
}
```

LOGIN AUTHENTICATED USERS

```
$stmt = $conn->prepare(
  "SELECT user_id, user_email, user_password FROM Users WHERE user_email = ?"
);
$stmt->bind_param("s", $user_email);
$stmt->execute();

$stmt->bind_result($user_id_db, $user_email_db, $user_password_db);
```

Select the row from the users table where the user email matches

LOGIN AUTHENTICATED USERS

```
$stmt = $conn->prepare(
  "SELECT user_id, user_email, user_password FROM Users WHERE user_email = ?"
);
$stmt->bind_param("s", $user_email);
$stmt->execute();

$stmt->bind_result($user_id_db, $user_email_db, $user_password_db);
```

Bind the result of the SELECT statement to variables

LOGIN AUTHENTICATED USERS

```
$stmt = $conn->prepare(
  "SELECT user_id, user_email, user_password FROM Users WHERE user_email = ?"
);
$stmt->bind_param("s", $user_email);
$stmt->execute();

$stmt->bind_result($user_id_db, $user_email_db, $user_password_db);
```

Multiple rows may be selected (not here but in general), each row’s information will be available in these variables

LOGIN AUTHENTICATED USERS

```
$user_valid = false;
while ($stmt->fetch()) {
  if ($user_id_db) {
    // Check if the password hashes are the same.
    if (sha1($user_password) == $user_password_db) {
```

Fetch the results of select and compare the passwords

LOGIN AUTHENTICATED USERS

```
$user_valid = false;
while ($stmt->fetch()) {
  if ($user_id_db) {
    // Check if the password hashes are the same.
    if (sha1($user_password) == $user_password_db) {
```

Encrypt the user specified password before you **compare** them - remember the password stored in the database is encrypted!

LOGIN AUTHENTICATED USERS

```
$user_valid = false;
while ($stmt->fetch()) {
  if ($user_id_db) {
    // Check if the password hashes are the same.
    if (sha1($user_password) == $user_password_db) {
```

If the passwords match the encrypted strings will also match!

LOGIN AUTHENTICATED USERS

```
$user_valid = false;
while ($stmt->fetch()) {
  if ($user_id_db) {
    // Check if the password hashes are the same.
    if (sha1($user_password) == $user_password_db) {
```

If the user does not exist or the passwords don't match specify an error message and don't allow the user access to other pages

LOGIN AUTHENTICATED USERS

```
header("Location: Example11-CredentialMgmt-loginSuccess.php");
```

This gets interesting - **this redirects the browser to the login success page** - however there is a whole bunch of stuff going on behind the scenes for this to work

LOGIN AUTHENTICATED USERS

```
header("Location: Example11-CredentialMgmt-loginSuccess.php");
```

The header() method emits a header to the browser from the server - it can be any header

LOGIN AUTHENTICATED USERS

```
header("Location: Example11-CredentialMgmt-loginSuccess.php");
```

The “**Location: path**” in the header is what tells the browser that it should go to the page specified in the header

LOGIN AUTHENTICATED USERS

```
header("Location: Example11-CredentialMgmt-loginSuccess.php");
```

There is one issue though - headers() have to be the very first thing sent from the server

LOGIN AUTHENTICATED USERS

```
header("Location: Example11-CredentialMgmt-loginSuccess.php");
```

There is one issue though - headers() have to be the very first thing sent from the server

No HTML, nothing can be rendered on the browser before the header

LOGIN AUTHENTICATED USERS

```
header("Location: Example11-CredentialMgmt-loginSuccess.php");
```

There is one issue though - headers() have to be the very first thing sent from the server

No HTML, nothing can be rendered on the browser before the header

But our current page has the form and whole bunch of stuff already rendered!

LOGIN AUTHENTICATED USERS

That was a lot to take in - how do I get this to work?

Buffer all the output you render in this page

Flush the buffer before emitting the header

LOGIN AUTHENTICATED USERS

```
ob_start();
```

This **starts** an output buffer which stores whatever is rendered on the browser

LOGIN AUTHENTICATED USERS

```
ob_start();
```

This **starts** an output buffer which stores whatever is rendered on the browser

Just before emitting the redirect header flush the buffer using:

```
while (ob_get_status()) {
  ob_end_clean();
}
```

LOGIN AUTHENTICATED USERS

```
while (ob_get_status()) {
  ob_end_clean();
}
```

Emitting the header **after** flushing the output buffer allows us to redirect to the login success page!

LOGIN AUTHENTICATED USERS

```
// clear out the output buffer
while (ob_get_status()) {
  ob_end_clean();
}

if (isset($continue_url)) {
  header("Location: $continue_url");
} else {
  header("Location: Example11-CredentialMgmt-loginSuccess.php");
}
```

Flush the buffer if it has and then emit the header

LOGIN AUTHENTICATED USERS

```
    } else {
      $error_message = 'Wrong user name or password provided!';
    }
  } else {
    $error_message = 'Wrong user name or password provided';
  }
}
```

If the user does not exist or the password does not match show an error message and stay on this page

LOGIN AUTHENTICATED USERS

All other code in this file is very similar to the sign up page

A LITTLE ABOUT HASHING

A LITTLE ABOUT HASHING

Hashing is a type of algorithm which takes **any size** of data and converts it to a **fixed length** of data

A LITTLE ABOUT HASHING

It should be easy to generate a hash of a message

You cannot generate the original message from the hash

modifying the message modifies the hash

different messages have different hashes

A LITTLE ABOUT HASHING

Hashes are **irreversible**, you cannot get the original value back from the hash

A LITTLE ABOUT HASHING

The hash function should be resistant to:

Collisions: Two messages should not generate the **same** hash

A LITTLE ABOUT HASHING

The hash function should be resistant to:

Collisions: Two messages should not generate the **same** hash

Pre-image resistance: Given a hash it should be near impossible to find a message which results in that hash

A LITTLE ABOUT HASHING

The hash function should be resistant to:

Collisions: Two messages should not generate the **same** hash

Pre-image resistance: Given a hash it should be near impossible to find a message which results in that hash

Second pre-image resistance: It should be infeasible to have two messages have the same hash

A LITTLE ABOUT HASHING

The hash function should be resistant to:

Collisions: Two messages should not generate the **same** hash

Pre-image resistance: Given a hash it should be near impossible to find a message which results in that hash

Second pre-image resistance: It should be infeasible to have two messages have the same hash

A LITTLE ABOUT HASHING

Common hashing algorithms:

MD-5

SHA-1

SHA-2

SHA-3

SHA-1
SHA-2
SHA-3

A LITTLE ABOUT HASHING

MD-5

This is widely used but cryptographically flawed as it's prone to collisions i.e. two messages result in the same hash

MD-5

A LITTLE ABOUT HASHING

SHA-1 SHA-2 SHA-3

A family of algorithms published by the National Institute of Standards and Technology

MD-5

A LITTLE ABOUT HASHING

SHA-1 SHA-2 SHA-3

Of these, SHA-1 is considered cryptographically broken however it still has all the properties we need in a **password hashing** algorithm

A LITTLE ABOUT HASHING

Attacking hashed passwords can be done using:

Dictionary Attacks

Brute Force

Rainbow Tables

Brute Force

Rainbow Tables

A LITTLE ABOUT HASHING

Dictionary Attacks

This involves using a bank of previously seen passwords and trying each to see if there is a match

Brute Force

Rainbow Tables

A LITTLE ABOUT HASHING

Dictionary Attacks

Thanks to password breaches huge loads of **real passwords** are available to hackers

Dictionary Attacks

Rainbow Tables

A LITTLE ABOUT HASHING

Brute Force

Brute force refers to trying **every combination** of alphabets, numbers, special characters to try and **guess** the password

Dictionary Attacks

Rainbow Tables

A LITTLE ABOUT HASHING

Brute Force

If your password is 8 characters long and you're choosing from the ASCII set of 128 characters then there are 128^8 possibilities

Dictionary Attacks

Rainbow Tables

A LITTLE ABOUT HASHING

Brute Force

Addition of every character to your password makes the brute force method **exponentially** tougher

Dictionary Attacks

Brute Force

A LITTLE ABOUT HASHING

Rainbow Tables

This involves setting a precomputed table for reversing the hash functions

Dictionary Attacks

Brute Force

A LITTLE ABOUT HASHING

Rainbow Tables

The precomputed table should ideally have the hashes of all passwords which a hacker plans to check as a part of the attack

Dictionary Attacks

Brute Force

A LITTLE ABOUT HASHING

Rainbow Tables

That can get **prohibitively** large!

Dictionary Attacks

Brute Force

A LITTLE ABOUT HASHING

Rainbow Tables

Rainbow tables involve storing a **subset** of the hashes which can be used to **trace** the original hash

Dictionary Attacks

Brute Force

A LITTLE ABOUT HASHING

Rainbow Tables

A complete discussion of Rainbow Tables is **beyond** the scope of this lecture but it's useful to know that this is an important technique to hack hashed passwords

A LITTLE ABOUT HASHING

SALT

Salting is a technique where a value is **appended** to the password before it is hashed and stored

The hash after the password is salted is called the **salted hash**

A LITTLE ABOUT HASHING

SALT

```
saltedhash = hash(password + salt)
```

The salt can be stored somewhere in the database and can be unique for a **database**, for a **table** or for each **password**

A LITTLE ABOUT HASHING

SALT

```
saltedhash = hash(password + salt)
```

The basis of a Rainbow Tables attack is that the **same password** produces the **same hash**

A LITTLE ABOUT HASHING

SALT

```
saltedhash = hash(password + salt)
```

The addition of a salt makes this assumption false!

A LITTLE ABOUT HASHING

SALT

```
saltedhash = hash(password + salt)
```

Rainbow Table attacks are made nearly impossible by the use of salts!

CREDENTIAL MANAGEMENT

Let's look at some of these:

✓ Password Strength

✓ Password Use

✓ Password In Transit

✓ Password Storage

Password Recovery

CREDENTIAL MANAGEMENT

Password Recovery

Password change, password recovery have their own unique issues and caveats

CREDENTIAL MANAGEMENT

Password Recovery

1. Initialize and Notify
2. Protect the current account
3. Validation using tokens
4. User Verification
5. Destroy tokens and notify
7. Login
8. Audit trail using logging

CREDENTIAL MANAGEMENT

Initialize and Notify

When a user initiates a **password recovery** only ask for an email address

CREDENTIAL MANAGEMENT

Initialize and Notify

When a user initiates a password recovery only ask for an email address

Do not provide feedback on whether the email address was valid in your system!

CREDENTIAL MANAGEMENT

Initialize and Notify

When a user initiates a password recovery only ask for an email address

Do not provide feedback on whether the email address was valid in your system!

Attackers can use this to harvest users in your system!

CREDENTIAL MANAGEMENT

Initialize and Notify

When a user initiates a password recovery only ask for an email address

Notify the user using that email address that a password recovery request has been initiated

Do not provide feedback on whether the email address was valid in your system!

Attackers can use this to harvest users in your system!

CREDENTIAL MANAGEMENT

Password Recovery

1. Initialize and Notify ✓
2. Protect the current account
3. Validation using tokens
4. User verification
5. Destroy tokens and notify
7. Login
8. Audit trail using logging

CREDENTIAL MANAGEMENT

Protect the current account

If a password recovery request has been initialized:

do not **lock** the user out of the account

do not **de-activate** the old password

CREDENTIAL MANAGEMENT

Protect the current account

If a password recovery request has been initialized:

This is a classic Denial Of Service (DOS) attack!

do not lock the user out of the account

do not de-activate the old password

CREDENTIAL MANAGEMENT

Protect the current account

If a password recovery request has been initialized:

do not lock the user out of the account

do not de-activate the old password

This behavior allows hackers to block legitimate users from accessing their accounts

CREDENTIAL MANAGEMENT

Password Recovery

1. Initialize and Notify ✓
2. Protect the current account ✓
3. Validation using tokens
4. User Verification
5. Destroy tokens and notify
7. Login
8. Audit trail using logging

CREDENTIAL MANAGEMENT

Validation using tokens

Use a token to identify a specific password recovery request

CREDENTIAL MANAGEMENT

Validation using tokens

Generate a **secure token** for password reset and store this along with a timestamp

Have this token be valid only for a **limited** period of time i.e. 1 hour, 1 day

CREDENTIAL MANAGEMENT

Validation using tokens

The token should not represent any sensitive data associated with the user!

Generate a secure token for password [illegible] along with a timestamp

Have this token be valid only for a limited period of time i.e. 1 hour, 1 day

CREDENTIAL MANAGEMENT

Validation using tokens

Even better store a hash of the token in your database and also store it in a different table from other user credential data

Generate a secure token for password reset and store this along with a timestamp

Have this token be valid only for a limited period of time i.e. 1 hour, 1 day

CREDENTIAL MANAGEMENT

Validation using tokens

store a hash of the token in your database

All the password protection reasons apply to reset tokens as well!

CREDENTIAL MANAGEMENT

Validation using tokens

Database information is also prone to vulnerabilities using hacks such as SQL injection

store it in a different table from other user credential data

CREDENTIAL MANAGEMENT

Validation using tokens

Even better store a hash of the token in your database and also store it in a different table from other user credential data

Generate a secure token for password reset and store this along with a timestamp

Have this token be valid only for a limited period of time i.e. 1 hour, 1 day

CREDENTIAL MANAGEMENT

Validation using tokens

Generate a **secure token** for password reset and store this along with a timestamp

Have this token be valid only for a **limited** period of time i.e. 1 hour, 1 day limited period

CREDENTIAL MANAGEMENT

Validation using tokens

Legitimate email addresses should receive a link with the **token** which takes them to a **password reset page**

Ensure the password reset link is over **https**

limited period
https
password reset page

CREDENTIAL MANAGEMENT

Validation using tokens

Avoid specifying the **current password** or even the **current user name** in the mail

Give only the information required - the message and the reset link!

CREDENTIAL MANAGEMENT

Validation using tokens

limited period

https

password reset page

no current password, current user name

CREDENTIAL MANAGEMENT

Password Recovery

1. Initialize and Notify ✓
2. Protect the current account ✓
3. Validation using tokens ✓
4. User verification
5. Destroy tokens and notify
7. Login
8. Audit trail using logging

CREDENTIAL MANAGEMENT

User verification

The user needs to access and click on the **reset link** in the email to continue with the process - this should take him to a secure page for password reset

Remember, the link is valid only for a **limited** period since it was generated

CREDENTIAL MANAGEMENT

click on the reset link

User verification

If the link has **expired** notify the user and start the process all over again

CREDENTIAL MANAGEMENT

click on the reset link

User verification

If the link is valid at this point:

1. The user is a legit user

2. Or a malicious who has access to a legit user's email

How do we differentiate?

CREDENTIAL MANAGEMENT

click on the reset link

User verification

1. The user is a legit user
2. Or a malicious who has access to a legit user's email

How do we differentiate?

2 FACTOR AUTHENTICATION OR SECRET QUESTION AND ANSWER

CREDENTIAL MANAGEMENT

User verification

2 FACTOR AUTHENTICATION

A security process which requires 2 means of identification before users are allowed to access secure data

Usually one physical (card, numeric code) and one memorized (password)

CREDENTIAL MANAGEMENT

click on the reset link

User verification

SECRET QUESTION AND ANSWER

1. The user is a legit user

2. Or a malicious who has access to a legit user's email

How do we differentiate?

Once again treat these as you would passwords! They should be cryptographically secured

2 FACTOR AUTHENTICATION OR SECRET QUESTION AND ANSWER

CREDENTIAL MANAGEMENT

User verification

click on the reset link

2 factor authentication
or secret questions

Once the user has been verified **only then** allow them provide a new password

CREDENTIAL MANAGEMENT

User verification

click on the reset link

2 factor authentication
or secret questions

In the case of password **RESET** rather than recovery **ask for the old password** before allowing the user to change the password

CREDENTIAL MANAGEMENT

User verification

click on the reset link

2 factor authentication
or secret questions

RESET - ask for the old
password

The password change is now
successful!

CREDENTIAL MANAGEMENT

User verification

click on the reset link

2 factor authentication
or secret questions

RESET - ask for the old
password

Do not automatically login the user!

CREDENTIAL MANAGEMENT

User verification

click on the reset link

2 factor authentication
or secret questions

RESET - ask for the old
password

Just take them to the login page

CREDENTIAL MANAGEMENT

Password Recovery

1. Initialize and Notify ✓
2. Protect the current account ✓
3. Validation using tokens ✓
4. User verification ✓
5. Destroy tokens and notify
7. Login
8. Audit trail using logging

CREDENTIAL MANAGEMENT

Destroy tokens and notify

Once the password change was successful **destroy** the secure token associated with this request

Once again **notify** the user - just in case

CREDENTIAL MANAGEMENT

Password Recovery

1. Initialize and Notify ✓
2. Protect the current account ✓
3. Validation using tokens ✓
4. User verification ✓
5. Destroy tokens and notify ✓
7. Login
8. Audit trail using logging

CREDENTIAL MANAGEMENT

Login

Re-logging in forces the creation of **new** sessions

Any **existing** session with the old password should be logged out and the session destroyed

CREDENTIAL MANAGEMENT

Password Recovery

1. Initialize and Notify ✓
2. Protect the current account ✓
3. Validation using tokens ✓
4. User verification ✓
5. Destroy tokens and notify ✓
7. Login ✓
8. Audit trail using logging

CREDENTIAL MANAGEMENT

Audit trail using logging

Log every step of the password change/recovery process - remember do not log passwords!

CREDENTIAL MANAGEMENT

Audit trail using logging

Consider throttling or legitimizing password change requests using **CAPTCHA** - a great way to differentiate between bots and users

CREDENTIAL MANAGEMENT

Password Recovery

1. Initialize and Notify ✓
2. Protect the current account ✓
3. Validation using tokens ✓
4. User verification ✓
5. Destroy tokens and notify ✓
7. Login ✓
8. Audit trail using logging ✓