Module 2 –WEB APPLICATION SECURITY PRINCIPLES

Authentication and Authorization Overview

Authentication and Authorization are crucial components of any security framework. They are often used together but serve distinct purposes:

- Authentication is the process of verifying the identity of a user, system, or application. It ensures that the entity attempting to access a system is who they claim to be.
- Authorization is the process of granting access rights and permissions to authenticated users or systems. It defines what an authenticated user can do once their identity has been confirmed.

Authentication: Access Control Overview

Access Control is a security mechanism that defines who can access or use resources in a computing environment. It is often divided into three main categories:

1. Authentication - Verifying the identity of the user/system.
2. Authorization - Granting the right to perform actions on resources after authentication.
3. Auditing - Tracking activities for compliance and security monitoring.

Authentication typically involves verifying a user's identity through a variety of means such as passwords, biometric scans, or digital certificates.

Authentication Fundamentals

Authentication is the process of verifying the identity of a user or system. It is typically accomplished using one of the following factors:

1. Something You Know (Knowledge Factor):
    - Passwords or PINs.
    - Security questions.
2. Something You Have (Possession Factor):
    - Hardware tokens (e.g., USB security keys).
    - One-time passcodes (OTPs) sent via SMS or email.

3. Something You Are (Inherence Factor):
    - Biometric factors like fingerprints, facial recognition, retina scans, or voice recognition.
4. Somewhere You Are (Location Factor) (Optional):
    - IP address or GPS coordinates.
5. Something You Do (Behavioral Factor) (Optional):
    - Behavioral biometrics, such as how a user types on a keyboard or how they interact with a device.

Two-Factor and Three-Factor Authentication

1. Two-Factor Authentication (2FA):
    - Combines two different factors for authentication.
    - Example: Password (something you know) and an OTP (something you have).
    - Benefit: It enhances security by requiring both a piece of knowledge (e.g., password) and a possession (e.g., phone for OTP).
    - Common Methods: SMS codes, email codes, authenticator apps (Google Authenticator, Authy).

2. Three-Factor Authentication (3FA):

- Uses three different factors for authentication.
- Example: Password (something you know), OTP (something you have), and fingerprint recognition (something you are).
- Benefit: Even higher security by requiring three independent forms of verification.
- Common Methods: Biometrics, hardware tokens, and PINs.

Web Application Authentication

In the context of web applications, authentication often involves:

1. Login Forms:

- Users provide their credentials (username and password) to authenticate.

2. Session Management:
    - After authentication, the user is granted a session, often through a session ID stored in a cookie.
    - A session represents the state of the user's authentication and allows continued access without needing to re-authenticate.
3. Token-Based Authentication:
    - Common in modern web applications and APIs (e.g., JSON Web Tokens (JWT)).
    - The server issues a token upon successful authentication, which the client stores and sends with subsequent requests.
4. OAuth2:
    - A popular protocol for third-party authentication. For example, using your Google or Facebook login to authenticate with another service.
5. Single Sign-On (SSO):
    - Enables users to authenticate once and gain access to multiple applications or services without having to re-enter credentials.

Securing Password-Based Authentication

Password Security Measures:

1. Strong Password Policies:
    - Require passwords to be complex (e.g., a mix of uppercase, lowercase, numbers, and special characters).
    - Enforce length requirements (e.g., 12+ characters).
2. Password Hashing:
    - Never store passwords in plain text. Always hash passwords using algorithms like bcrypt, Argon2, or PBKDF2.
    - Add a salt to the hash to make it resistant to rainbow table attacks.

3. Rate Limiting and Account Lockout:
    - Limit login attempts to mitigate brute-force attacks.
    - Implement account lockout after several failed attempts, or introduce time delays between failed login attempts.
4. Password Storage Best Practices:
    - Store hashed passwords with a salt using strong algorithms (e.g., bcrypt).
    - Use pepper for additional security—this is a secret key added to the password before hashing, known only to the server.
5. Password Recovery:
    - Use secure methods for password reset (e.g., sending a secure link to the registered email address, with time-limited validity).

Authorization: Access Control Continued

Authorization manages what users can do once authenticated. Common access control models include:

1. Role-Based Access Control (RBAC):
    - Users are assigned roles (e.g., admin, user, guest) that grant access to different levels of resources.
    - Roles determine what permissions the user has, such as read, write, delete, or execute.
2. Attribute-Based Access Control (ABAC):
    - Permissions are granted based on attributes such as the user's department, role, or other contextual factors (e.g., time of day, location).
3. Access Control Lists (ACL):
    - Explicitly defines what users can access specific resources and what actions they can perform.
    - Often used in file systems and network resources.

Session Management Fundamentals

Session management controls the state of user interactions within a web application after the user has authenticated. It is key to maintaining security and a smooth user experience.

1. Session Creation:
    - After successful authentication, the application generates a unique session ID and stores it on the server. This session ID is sent to the client, usually as a cookie.
2. Session Persistence:
    - Sessions persist across multiple requests (i.e., users don’t need to authenticate repeatedly within a session).
    - Sessions are often expired after a certain period of inactivity, or they can be manually terminated by the user (logout).
3. Session Hijacking Prevention:
    - Use Secure Cookies: Set the "Secure" flag to ensure cookies are transmitted over HTTPS only.
    - SameSite Cookie Attribute: Prevents cross-site request forgery (CSRF) attacks by restricting how cookies are sent.
    - Tokenization: Use tokens (like JWT) for stateless authentication, reducing the need for server-side session storage.
4. Session Expiration and Logout:
    - Implement session timeouts or idle timeouts. This ensures that inactive sessions are automatically terminated after a predefined period.
    - Always provide a secure logout feature that invalidates the session or token.
5. Multi-Device Sessions:
    - Allow users to manage sessions across different devices by providing options to view and log out from other devices.

Securing Web Application Session Management

1. Use HTTPS:
    - Always use HTTPS to encrypt the communication between the client and server, protecting sensitive session data, including session IDs.
2. Session ID Management:
    - Use strong, random session IDs.
    - Regenerate session IDs after login to prevent session fixation attacks.
3. Secure Cookies:
    - Use HttpOnly flag to prevent JavaScript from accessing session cookies, reducing the risk of XSS attacks.
    - Use the Secure flag to ensure cookies are only sent over HTTPS.
4. Prevent Cross-Site Request Forgery (CSRF):
    - Implement CSRF tokens to verify that requests are coming from an authenticated source, especially for state-changing actions.
5. Regular Session Expiry:
    - Set reasonable session expiration times to limit the potential impact of session hijacking.
    - Expired sessions should automatically require re-authentication.
6. Monitor Session Activity:
    - Track unusual session activities, such as rapid login attempts, geolocation changes, or simultaneous access from different IP addresses.
7. Use Web Application Firewalls (WAF):
    - Employ a WAF to filter and monitor HTTP traffic, providing an additional layer of protection against malicious session attacks.

1. Authentication: Access Control Overview

Q1:Create an HTML login form that accepts a username and password. Implement basic client-side validation using JavaScript to check if both fields are filled before submission.

Solution Code (HTML + JavaScript):

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Login Form</title>
    <script>
        function validateForm() {
            const username = document.getElementById("username").value;
            const password = document.getElementById("password").value;

            if (username === "" || password === "") {
                alert("Both fields are required!");
                return false; // Prevent form submission
            }
            return true; // Allow form submission
        }
    </script>
</head>
```

```
<body>

    <h2>Login</h2>
    <form onsubmit="return validateForm()">
        <label for="username">Username:</label>
        <input type="text" id="username" name="username" required>
        <br><br>
        <label for="password">Password:</label>
        <input type="password" id="password" name="password" required>
        <br><br>
        <input type="submit" value="Login">
    </form>

</body>
</html>
```

Explanation:

This HTML form includes client-side validation with JavaScript. The form will not submit if either the username or password field is empty, ensuring that both are filled.

2. Authentication Fundamentals

Q2:Create a web page that simulates the authentication process by prompting the user for a username and password. Display a message indicating whether the authentication is successful or failed based on hardcoded credentials.

Solution Code (HTML + JavaScript):

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Authentication Simulation</title>
    <script>
        function authenticate() {
            const username = document.getElementById("username").value;
            const password = document.getElementById("password").value;

            // Hardcoded credentials for simulation
            const validUsername = "admin";
            const validPassword = "password123";

            if (username === validUsername && password === validPassword)
{
                alert("Authentication successful!");
            } else {
                alert("Authentication failed. Please check your credentials.");
            }
        }
    </script>
</head>
<body>

    <h2>Authentication</h2>
    <form onsubmit="event.preventDefault(); authenticate()">
```

```
        <label for="username">Username:</label>
        <input type="text" id="username" required>
        <br><br>
        <label for="password">Password:</label>
        <input type="password" id="password" required>
        <br><br>
        <input type="submit" value="Authenticate">
    </form>

</body>
</html>
```

Explanation:

This simple authentication simulation checks the entered username and password against hardcoded credentials. If they match, a success message is shown; otherwise, a failure message is displayed.

3. Two-Factor Authentication (2FA)

Q3: Create an HTML form for Two-Factor Authentication (2FA). In this form, the user enters their username and password. After a successful login, a simulated OTP (One-Time Password) is sent to the user for verification.

Solution Code (HTML + JavaScript):

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Two-Factor Authentication</title>
    <script>
        function authenticate() {
            const username = document.getElementById("username").value;
            const password = document.getElementById("password").value;

            // Hardcoded credentials
            const validUsername = "user";
            const validPassword = "password123";

            if (username === validUsername && password === validPassword)
{
                alert("Login successful! Please enter the OTP sent to your
phone.");
                document.getElementById("otpSection").style.display = "block";
            } else {
                alert("Login failed. Invalid credentials.");
            }
        }

        function verifyOTP() {
            const enteredOTP = document.getElementById("otp").value;
            const validOTP = "123456"; // Simulated OTP for demonstration

            if (enteredOTP === validOTP) {
                alert("Two-Factor Authentication successful!");
```

```
            } else {
                alert("Invalid OTP. Please try again.");
            }
        }
    </script>
</head>
<body>

    <h2>Login</h2>
    <form onsubmit="event.preventDefault(); authenticate()">
        <label for="username">Username:</label>
        <input type="text" id="username" required>
        <br><br>
        <label for="password">Password:</label>
        <input type="password" id="password" required>
        <br><br>
        <input type="submit" value="Login">
    </form>

    <!-- OTP Section -->
    <div id="otpSection" style="display:none;">
        <h3>Enter OTP</h3>
        <label for="otp">OTP:</label>
        <input type="text" id="otp" required>
        <br><br>
        <button onclick="verifyOTP()">Verify OTP</button>
    </div>
```

```
</body>

</html>
```

Explanation:

After the user enters their username and password, they are prompted to enter an OTP. If the credentials are valid, an OTP prompt appears. The user must enter the correct OTP (simulated here as 123456) to complete the authentication.

4. Securing Password-Based Authentication

Q4: Create an HTML form that implements a basic password strength checker. The password field should be checked for minimum length, presence of special characters, and a mix of uppercase and lowercase letters.

Solution Code (HTML + JavaScript):

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Password Strength Checker</title>

    <script>

        function checkPasswordStrength() {

            const password = document.getElementById("password").value;

            let strengthMessage = "";

            let strength = 0;
```

```
            // Check length
            if (password.length >= 8) strength++;

            // Check for uppercase letter
            if (/[A-Z]/.test(password)) strength++;

            // Check for lowercase letter
            if (/[a-z]/.test(password)) strength++;

            // Check for special characters
            if (/[!@#$%^&*(),.?":{}|<>]/.test(password)) strength++;

            // Determine strength
            if (strength === 4) {
                strengthMessage = "Password is Strong.";
            } else if (strength === 3) {
                strengthMessage = "Password is Medium.";
            } else {
                strengthMessage = "Password is Weak.";
            }

            document.getElementById("strengthMessage").innerText =
strengthMessage;
        }
    </script>
</head>
<body>
```

```
    <h2>Password Strength Checker</h2>
    <form>
        <label for="password">Password:</label>
        <input type="password" id="password"
onkeyup="checkPasswordStrength()" required>
        <br><br>
        <p id="strengthMessage"></p>
    </form>


</body>
</html>
```

Explanation:

This form checks the strength of the password by verifying its length, the presence of uppercase and lowercase letters, and the inclusion of special characters. It provides feedback on whether the password is weak, medium, or strong.

5. Session Management Fundamentals

Q5:Create a simple login/logout simulation where the session is managed using JavaScript. When the user logs in, a "session" is created, and a "logout" button will allow them to end the session.

Solution Code (HTML + JavaScript):

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Session Management</title>
    <script>
        function login() {
            const username = document.getElementById("username").value;
            const password = document.getElementById("password").value;

            // Simulate authentication
            if (username === "user" && password === "password123") {
                localStorage.setItem("session", "active");
                alert("Login successful. Session started.");
                document.getElementById("loginForm").style.display = "none";
                document.getElementById("logoutButton").style.display =
"block";
            } else {
                alert("Invalid credentials. Try again.");
            }
        }

        function logout() {
            localStorage.removeItem("session");
            alert("Session ended.");
            document.getElementById("loginForm").style.display = "block";
            document.getElementById("logoutButton").style.display = "none";
        }
```

```
        window.onload = function() {
            if (localStorage.getItem("session") === "active") {
                document.getElementById("loginForm").style.display = "none";
                document.getElementById("logoutButton").style.display =
"block";
            }
        }
    </script>
</head>
<body>

    <h2>Login</h2>
    <div id="loginForm">
        <form onsubmit="event.preventDefault(); login()">
            <label for="username">Username:</label>
            <input type="text" id="username" required>
            <br><br>
            <label for="password">Password:</label>
            <input type="password" id="password" required>
            <br><br>
            <input type="submit" value="Login">
        </form>
    </div>

    <button id="logoutButton" onclick="logout()"
style="display:none;">Logout</button>

</body>
```

</html>

Explanation:

This simulation uses the localStorage API to mimic session management. After a successful login, the session is stored in localStorage, and the user is presented with a "logout" button. When the user clicks "logout", the session is cleared, and the login form is displayed again.