Module1

Introduction

History of Web Applications

- **Early Web (1990s):** Initially, websites were static HTML pages with minimal interactivity.
- **Rise of Dynamic Web (2000s):** Introduction of JavaScript, PHP, ASP, and databases (MySQL, PostgreSQL) led to interactive applications.
- **Web 2.0 (Mid-2000s – Present):** social media, AJAX, and APIs created highly dynamic, user-driven web experiences.
- **Modern Web Apps (2020s):** Single-page applications (SPA), progressive web apps (PWA), microservices, and API-driven applications dominate.

Interface and Structure of Web Applications

Web applications consist of the following layers:

1. **Client-side (Frontend):**
    - Built using HTML, CSS, JavaScript.
    - Runs in a web browser.
    - Uses frameworks like React, Angular, or Vue.js.
2. **Server-side (Backend):**
    - Processes business logic and user requests.
    - Built with Node.js, Django, Flask, .NET, Ruby on Rails, etc.
3. **Database Layer:**
    - Stores data for dynamic content and authentication.
    - Uses databases like MySQL, PostgreSQL, MongoDB, or Firebase.
4. **Network Layer:**
    - Handles HTTP/HTTPS communication between the client and server.
5. **Security Layer:**
    - Includes authentication, encryption, access control, and monitoring.

Benefits of Web Applications

✅ Accessible from anywhere using a web browser.
✅ No installation required for end-users.
✅ Easier updates and maintenance compared to desktop applications.

✅ Supports multi-platform access (Windows, macOS, Linux, mobile).
✅ Scalable and cost-effective with cloud hosting options.

Drawbacks of Web Applications

❌ Security vulnerabilities such as SQL injection, XSS, CSRF.
❌ Performance can be slower due to internet dependency.
❌ Browser compatibility issues.
❌ Higher server costs for resource-intensive applications.
❌ Data privacy concerns in cloud-hosted applications.

Web Application vs. Cloud Application

Feature	**Web Application**	**Cloud Application**
Definition	Runs on a web server and accessed via a browser	Hosted on a cloud platform with flexible resources
Hosting	Hosted on a specific server	Hosted in a cloud (AWS, Azure, GCP)
Scalability	Limited based on server capacity	Highly scalable with on-demand resources
Maintenance	Requires manual updates and patches	Managed updates and automated scaling
Data Storage	Typically, on a central database	Distributed storage across multiple servers
Examples	E-commerce sites, blogs, forums	Google Drive, Dropbox, Office 365

Security Fundamentals

1. Input Validation

- Prevents malicious input (e.g., SQL injection, cross-site scripting).
- Ensures data integrity and security.
- Types of input validation:
    - **Client-side validation** (using JavaScript).
    - **Server-side validation** (more secure, using backend logic).
- Best practices:
  ✅ Use allowlists (specify allowed inputs instead of blocking bad ones).
  ✅ Sanitize and escape user inputs.
  ✅ Validate input formats (e.g., email, phone numbers).

```
javascript

function validateUsername(username)

{

const regex = /^[a-zA-Z0-9]{3,20}$/; // Only allows alphanumeric characters, 3 to 20 characters long

return regex.test(username);

}
```

2. Attack Surface Reduction

- The **attack surface** is the sum of all possible entry points an attacker can exploit.
- Reducing the attack surface minimizes vulnerabilities.
- Strategies:
  ✅ Remove unused features and APIs.
  ✅ Limit access controls to sensitive data.
  ✅ Regularly update and patch software.
  ✅ Implement the **principle of least privilege** (PoLP).

3. Rules of Thumb for Web Security

- **Confidentiality:** Ensure sensitive data is only accessible to authorized users.
- **Integrity:** Prevent data tampering and ensure authenticity.
- **Availability:** Ensure the application is resistant to attacks like DDoS.
- **Authentication & Authorization:** Implement strong user authentication mechanisms (OAuth, JWT, SAML).
- **Encryption:** Use HTTPS (TLS/SSL), encrypt sensitive data in transit and at rest.
- **Logging & Monitoring:** Detect anomalies, log security events, and analyze logs for threats.

4. Classifying and Prioritizing Threats

- **Threat Classification Methods:**
  - **STRIDE Model:**

        - **S**poofing: Impersonation attacks.
        - **T**ampering: Unauthorized data modifications.
        - **R**epudiation: Denial of user actions.
        - **I**nformation Disclosure: Data leaks.
        - **D**enial of Service (DoS): Overloading system resources.
        - **E**levation of Privilege: Gaining unauthorized access.
    - **DREAD Model:**
        - **D**amage potential: How severe is the attack?
        - **R**eproducibility: How easily can it be repeated?
        - **E**xploitability: How easy is the attack to execute?
        - **A**ffected users: How many users are impacted?
        - **D**iscoverability: How easy is the vulnerability to find?
- **Prioritizing Security Risks (OWASP Top 10):**
    - **A01 - Broken Access Control:** Unauthorized access to restricted resources.
    - **A02 - Cryptographic Failures:** Weak encryption exposing sensitive data.
    - **A03 - Injection Attacks:** SQL injection, command injection.
    - **A04 - Insecure Design:** Poor security architecture decisions.
    - **A05 - Security Misconfiguration:** Default settings, unnecessary services running.
    - **A06 - Vulnerable Components:** Using outdated libraries and software.
    - **A07 - Identification & Authentication Failures:** Weak passwords, session hijacking.
    - **A08 - Software & Data Integrity Failures:** Supply chain attacks, unverified software updates.
    - **A09 - Security Logging & Monitoring Failures:** Lack of visibility into attacks.
    - **A10 - Server-Side Request Forgery (SSRF):** Exploiting trusted network access.

Examples

History of Web Applications – Real-World Examples

- **Static Websites (1990s)**
  - Example: **Yahoo (1995)** – Early static HTML-based search engine.
- **Dynamic Websites (2000s)**
  - Example: **Amazon (2000s)** – Introduced personalized recommendations and real-time transactions.
- **Web 2.0 & Social Media (Mid-2000s – Present)**
  - Example: **Facebook (2004)** – Interactive UI, user-generated content, and API-based architecture.
- **Modern Web Apps & Cloud-Based Services (2020s)**
  - Example: **Google Docs (Cloud-based, 2020s)** – Multi-user collaboration with real-time updates.

Interface and Structure – Example of a Web Application

Example: E-commerce Website (Amazon, eBay, Flipkart)

- **Client-side (Frontend):** Built with React, Vue.js, or Angular.
- **Backend (Server-side):** Uses Node.js, Django, or .NET for order processing and inventory management.
- **Database Layer:** Uses MySQL or MongoDB to store product and customer data.
- **Security Layer:** Implements HTTPS, authentication (OAuth), and role-based access.

Benefits & Drawbacks – Example of Online Banking Web Apps

Example: PayPal & Online Banking (HDFC, ICICI, SBI Net Banking)

- **Benefits:**
  ✅ Secure transactions using encryption.
  ✅ Accessible 24/7 from any device.
  ✅ Real-time fraud detection mechanisms.
- **Drawbacks:**
  ❌ Susceptible to phishing attacks.
  ❌ Requires a stable internet connection for transactions.

Web Application vs. Cloud Application – Example: Google Drive vs. Dropbox

- **Google Drive (Cloud Application):**
  - Stored in Google Cloud servers.
  - Real-time collaboration and auto-sync.
  - Scalable and accessible from any device.
- **Self-Hosted File Sharing Web App (OwnCloud, NextCloud):**
  - Hosted on private servers.
  - Requires manual maintenance.
  - Less scalable compared to Google Drive.

---

2. Security Fundamentals – Application Examples

1. Input Validation – Example: Login Form Security

Scenario: A user submits a login form with SQL Injection (admin' OR 1=1 --)

- **Vulnerable Web App (No Input Validation):**
  - The database runs the injected query, granting unauthorized access.
- **Secure Web App (Proper Input Validation):**
  - Uses prepared statements to sanitize inputs and prevent SQL injection.
  - Example: **LinkedIn uses input validation to prevent unauthorized logins.**

2. Attack Surface Reduction – Example: API Security in Social Media Platforms

Scenario: A social media app (Twitter) exposes unnecessary API endpoints.

- **Vulnerable App:**
  - Allows public access to internal admin APIs, leading to data leaks.
- **Secure App:**
  - Disables unused APIs, restricts sensitive API access via authentication tokens (OAuth).
  - Example: **Facebook restricts Graph API access based on user permissions.**

3. Rules of Thumb for Web Security – Example: Banking Apps

Example: SBI Net Banking (sbi.co.in), HDFC Bank Web App

- Uses **Multi-Factor Authentication (MFA)** to prevent unauthorized logins.
- Encrypts **financial transactions** with TLS/SSL.
- Logs and monitors **suspicious activities** to detect fraud.

4. Classifying and Prioritizing Threats – Example: E-commerce Website (Amazon, Flipkart)

Threat Modeling using STRIDE & OWASP Top 10

- **Threat:** SQL Injection (A03 - Injection Attacks)
  - **Example:** Attackers manipulate an Amazon search query to access backend data.
  - **Solution:** Implement **parameterized queries** to sanitize inputs.
- **Threat:** Broken Authentication (A07 - Authentication Failures)
  - **Example:** A weak password allows unauthorized access to an Amazon seller account.
  - **Solution:** Enforce **strong passwords** and **2FA authentication**.
- **Threat:** Denial of Service (DoS) Attack
  - **Example:** A botnet floods Flipkart's servers during a **Big Billion Days sale**.
  - **Solution:** Use **rate limiting, firewalls, and CDNs** to mitigate attacks.

Input Validation Questions with HTML Solution Code

Q1:

Write an HTML form with JavaScript to ensure that a user’s email input is in a valid email format before submitting.

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>Email Validation</title>
    <script>
        function validateEmail() {
            const email = document.getElementById("email").value;
            const regex = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]
{2,6}$/;

            if (!regex.test(email)) {
                alert("Please enter a valid email address.");
                return false; // Prevent form submission
            }
            return true; // Allow form submission
        }
    </script>
</head>
<body>

    <h2>Sign Up</h2>
    <form onsubmit="return validateEmail()">
        <label for="email">Email:</label>
        <input type="text" id="email" name="email" required>
        <input type="submit" value="Submit">
    </form>

</body>
</html>
```

Explanation:

This form uses JavaScript to validate the email input using a regular expression before submission. If the email is invalid, an alert will be displayed, and the form won't be submitted.

Q2:

Create an HTML form with JavaScript to prevent Cross-Site Scripting (XSS) by sanitizing the user’s input.

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>XSS Prevention</title>
    <script>
        function sanitizeInput() {
            let userInput = document.getElementById("comment").value;
            const textarea = document.createElement("textarea");
            textarea.textContent = userInput;
            userInput = textarea.innerHTML; // Escapes any HTML/JS characters

            document.getElementById("sanitizedComment").innerText = userInput;
            return false; // Prevent form submission for demonstration
        }
    </script>
</head>
```

```
<body>

    <h2>Leave a Comment</h2>
    <form onsubmit="return sanitizeInput()">
        <label for="comment">Comment:</label>
        <input type="text" id="comment" name="comment" required>
        <input type="submit" value="Submit">
    </form>

    <h3>Sanitized Comment:</h3>
    <p id="sanitizedComment"></p>

</body>
</html>
```

Explanation:

This HTML form takes user input for a comment, then sanitizes it by escaping any special HTML/JS characters before displaying it. This prevents malicious scripts from being executed (XSS attacks).

Attack Surface Reduction Rules of Thumb with HTML Solution Code

Q3:

Write an HTML form with JavaScript that prevents SQL Injection by properly sanitizing and escaping user input in a web application.

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>SQL Injection Prevention</title>
    <script>
        function sanitizeSQLInput() {
            let userInput = document.getElementById("username").value;

            // Simple sanitization: remove single quotes and semicolons
            userInput = userInput.replace(/['";]/g, "");

            // Display sanitized input
            document.getElementById("sanitizedInput").innerText = userInput;

            return false; // Prevent form submission for demonstration
        }
    </script>
</head>
<body>

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

    <h3>Sanitized Username:</h3>
    <p id="sanitizedInput"></p>
```

```
</body>

</html>
```

Explanation:

This form sanitizes the user input to prevent SQL Injection by removing characters like single quotes ('), semicolons (;), or other dangerous characters. This is a basic example; ideally, use parameterized queries on the server side for stronger protection.

Q4:

Create an HTML form that implements basic access control by limiting access to the form based on the user's IP address using JavaScript.

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>IP-Based Access Control</title>

    <script>

        function checkAccess() {

            // Simulating user's IP (In reality, you'd fetch this server-side)

            const userIP = "192.168.1.100";

            const allowedIP = "192.168.1.100"; // Define the allowed IP


            if (userIP !== allowedIP) {

                alert("Access denied: Unauthorized IP address.");
```

```
            return false; // Prevent access
        }
        return true; // Allow access
    }
  </script>
</head>
<body>

  <h2>Restricted Form</h2>
  <form onsubmit="return checkAccess()">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name" required>
    <input type="submit" value="Submit">
  </form>

</body>
</html>
```

Explanation:

This HTML form simulates a basic IP-based access control. If the user's IP doesn't match the allowed IP address (192.168.1.100), the form won't be accessible, and an alert message is displayed.

Classifying and Prioritizing Threats with HTML Solution Code

Q5:

Create an HTML page that classifies different types of security threats based on their impact and likelihood.

Solution Code (HTML + JavaScript):

html

Copy

Edit

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Classify Threats</title>
    <script>
        function classifyThreat() {
            const threat = document.getElementById("threat").value.toLowerCase();
            let classification = "";
            let priority = "";

            switch (threat) {
                case "sql injection":
                    classification = "Impact: High, Likelihood: Medium";
                    priority = "High Priority";
                    break;
                case "xss":
                    classification = "Impact: Medium, Likelihood: High";
                    priority = "Medium Priority";
                    break;
                case "ddos":
                    classification = "Impact: High, Likelihood: High";
                    priority = "High Priority";
                    break;
                case "phishing":
```

```
                classification = "Impact: High, Likelihood: Medium";
                priority = "High Priority";
                break;
            default:
                classification = "Threat not recognized.";
                priority = "Low Priority";
        }

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

  <h2>Classify a Security Threat</h2>
  <label for="threat">Enter a threat (e.g., SQL Injection, XSS, DDoS, Phishing):</label>
  <input type="text" id="threat" name="threat" required>
  <button onclick="classifyThreat()">Classify</button>

  <h3>Classification:</h3>
  <p id="classification"></p>

  <h3>Priority:</h3>
  <p id="priority"></p>

</body>
</html>
```

Explanation:

This page allows a user to input a security threat (e.g., SQL Injection, XSS, DDoS, Phishing), then it classifies the threat based on its potential impact and likelihood. The classification and priority are displayed based on predefined criteria.