Common Security Threats in Web Applications

1. Cross-Site Scripting (XSS)

1. Introduction to XSS

Cross-Site Scripting (XSS) is a web security vulnerability that allows attackers to inject malicious scripts into web applications. These scripts are executed in the context of a user's browser, potentially leading to data theft, session hijacking, or defacement of the web page.

2. Types of XSS Attacks

XSS vulnerabilities can be classified into three main types:

a) Stored XSS (Persistent XSS)

- Malicious script is permanently stored on the server (e.g., in a database, comment section, user profile, etc.).
- The script executes whenever a user visits the affected page.
- Example: A user posts a comment with a `<script>` tag that executes when another user views the comment.

b) Reflected XSS (Non-Persistent XSS)

- Malicious script is included in a URL or form input and immediately reflected back in the response.
- Common in search bars, error messages, or URL parameters.
- Example: An attacker sends a link like `https://example.com/search?q=<script>alert('Hacked')</script>`, which, if unescaped, executes JavaScript in the victim's browser.

c) DOM-Based XSS

- The attack happens entirely within the Document Object Model (DOM) on the client-side.
- JavaScript dynamically modifies the webpage based on user input without proper sanitization.
- Example: A vulnerable JavaScript function reads a URL fragment (`window.location.hash`) and inserts it directly into the page without escaping.

3. Consequences of XSS Attacks

- **Session Hijacking**: Attacker steals cookies and impersonates the user.
- **Defacement**: Malicious scripts modify the webpage content.
- **Phishing Attacks**: Injected script redirects users to fake login pages.
- **Keylogging**: Capturing user inputs (e.g., passwords, credit card details).
- **Browser Exploitation**: Injecting malicious scripts to exploit browser vulnerabilities.

4. Preventing XSS Attacks

To mitigate XSS attacks, developers should follow secure coding practices:

a) Input Validation

- Validate all user inputs before processing them.
- Allow only expected characters and reject suspicious inputs.

b) Output Encoding

- Encode dynamic content before rendering it in HTML.
- Use appropriate encoding functions:
    - **HTML Encoding**: Convert `<script>` to `&lt;script&gt;`
    - **JavaScript Encoding**: Escape characters like `"`, `'`, `<`, `>`.
    - **URL Encoding**: Encode user input in URLs to prevent script execution.

c) Content Security Policy (CSP)

- Configure CSP headers to restrict the execution of inline scripts and untrusted sources.
- Example CSP policy:

```
Content-Security-Policy: default-src 'self'; script-src 'self' https://trustedsource.com;
```

This Content-Security-Policy (CSP) restricts resources to the same origin (using default-src 'self') and allows scripts from the same origin and a trusted source (script-src 'self' https://trustedsource.com).

d) Use Security Libraries

- Implement security-focused libraries like `DOMPurify` to sanitize user input.
- Use frameworks with built-in XSS protection (e.g., React, Angular).

e) HTTP-Only Cookies

- Prevent JavaScript from accessing session cookies using the `HttpOnly` flag.
- Example:

```
Set-Cookie: sessionid=abc123; HttpOnly; Secure;
```

HttpOnly: This flag instructs the browser to prevent client-side scripts (like JavaScript) from accessing the cookie, mitigating the risk of Cross-Site Scripting (XSS) attacks.

Secure: This flag ensures that the cookie is only transmitted over secure, encrypted HTTPS connections, protecting against man-in-the-middle attacks where an attacker might intercept the cookie in transit.

f) Avoid Inline JavaScript

- Avoid using `eval()`, `document.write()`, and inline event handlers (`onclick`, `onmouseover`).
- Use event listeners instead:

```
document.getElementById("btn").addEventListener("click", function() {
  alert("Safe Click!");
});
```

5. Detecting XSS Vulnerabilities

- **Automated Scanners**: Use security tools like OWASP ZAP, Burp Suite, or Google CSP Evaluator.
- **Penetration Testing**: Manually test application input fields using XSS payloads.
- **Web Application Firewalls (WAFs)**: Deploy WAFs to detect and block malicious requests.

6. Conclusion

Cross-Site Scripting (XSS) is a serious vulnerability that can compromise user security. By following secure coding practices, input validation, output encoding, and implementing Content Security Policies, developers can effectively prevent XSS attacks and ensure a safer web application environment.

2. Cross-Site Request Forgery (CSRF)

1. Introduction to CSRF

Cross-Site Request Forgery (CSRF) is a type of web security vulnerability that allows an attacker to trick a user into performing unintended actions on a trusted website where they are authenticated. CSRF exploits the trust that a web application has in a user's browser.

2. How CSRF Works

Step 1: User Authentication

- The user logs into a website (e.g., a banking site) and receives an authentication token (usually a session cookie).

Step 2: Attacker Prepares a Malicious Request

- The attacker creates a malicious website or email with a hidden request targeting the authenticated website.

Step 3: User Interaction

- The user unknowingly interacts with the malicious website, triggering the request.

Step 4: Request Execution

- The user's browser sends the request, including authentication cookies, to the trusted website.
- The server processes the request as if it came from the legitimate user.

3. Example of a CSRF Attack

Scenario: Changing a User's Email

An attacker crafts a malicious HTML form:

```
<form action="https://bank.com/change-email" method="POST">
  <input type="hidden" name="email" value="attacker@example.com">
  <input type="submit" value="Submit">
```

```
</form>
```

The input type="hidden" is a fundamental part of HTML forms that allows developers to include data in a form submission without making it visible or directly interactable by the user.

- The attacker embeds this form in a webpage.
- When the user visits the page, the form is submitted automatically.
- If the user is logged into their banking account, their email gets changed to `attacker@example.com`.

4. Preventing CSRF Attacks

4.1 CSRF Tokens

- Generate a unique CSRF token for each session.
- Require the token in every state-changing request.
- Verify the token on the server before processing the request.

Example (Express.js Middleware for CSRF Protection)

```
const csrf = require('csurf');

const csrfProtection = csrf({ cookie: true }); //instructing to store the CSRF token in a cookie instead of the session.



app.use(csrfProtection);

app.post('/update-profile', (req, res) => {

   res.send('Profile updated successfully');

});
```

4.2 SameSite Cookie Attribute

- Set `SameSite` attribute on cookies to prevent them from being sent with cross-site requests.

Example:

```
app.use(session({
```

```
  secret: 'secret',
  resave: false,
  saveUninitialized: true,
  cookie: { sameSite: 'Strict' }
}));
```

Instructs the browser to only send the specified cookie in a first-party context, meaning it's only sent when the user is directly interacting with the website that set the cookie, and not when navigating to a different website.

4.3 Referer and Origin Header Validation

- Validate the `Referer` or `Origin` header to ensure the request is coming from a trusted domain.

4.4 User Authentication with Multi-Factor Authentication (MFA)

- Even if an attacker successfully executes a CSRF attack, MFA can prevent unauthorized actions.

4.5 Implementing CORS Policy

- Restrict allowed origins using Cross-Origin Resource Sharing (CORS) headers.

Example:

```
const cors = require('cors');

app.use(cors({ origin: 'https://trusted-site.com' }));
```

5. Real-World CSRF Attacks

Case Study: Gmail CSRF Vulnerability (2007)

- Attackers exploited CSRF to auto-forward emails from victim accounts to an attacker's address.
- Google mitigated this by implementing CSRF tokens.

6. Conclusion

- CSRF attacks exploit users' trust in a website.
- Implementing CSRF tokens, `SameSite` cookies, and request validation techniques can mitigate risks.
- Security should be continuously updated to handle new attack vectors.

By implementing proper CSRF protection techniques, web applications can safeguard users from unauthorized actions and improve overall security.

3. SQL Injection

1. Introduction to SQL Injection

SQL Injection (SQLi) is a code injection technique that exploits vulnerabilities in an application's database layer. It occurs when an attacker inserts or "injects" malicious SQL queries into input fields, manipulating the database to gain unauthorized access, retrieve data, or even destroy database records.

2. How SQL Injection Works

SQL Injection occurs when user inputs are not properly sanitized before being used in SQL queries. If an application dynamically constructs SQL queries using user input without validation, an attacker can modify the query structure.

Example of Vulnerable Code:

```
const express = require('express');

const mysql = require('mysql');

const app = express();

const connection = mysql.createConnection({

  host: 'localhost',

  user: 'root',
```

```
  password: '',
  database: 'testdb'
});

app.get('/user', (req, res) => {
  let username = req.query.username;
  let query = `SELECT * FROM users WHERE username = '${username}'`;
  connection.query(query, (err, results) => {
    if (err) throw err;
    res.json(results);
  });
});
```

Exploiting the Vulnerability:

If a user enters:

```
' OR '1'='1
```

The resulting query becomes:

```
SELECT * FROM users WHERE username = '' OR '1'='1';
```

Since '1'='1' is always true, this retrieves all user records, potentially exposing sensitive data.

3. Types of SQL Injection

3.1. Classic SQL Injection

Directly injecting SQL commands into a vulnerable input field to manipulate the query.

3.2. Blind SQL Injection

Occurs when an attacker does not directly see the database output but determines vulnerabilities based on application responses.

3.3. Time-Based SQL Injection

Uses SQL commands that cause a delay in database response to infer information.

3.4. Out-of-Band SQL Injection

Involves sending queries that cause the database to send data to an external server controlled by the attacker.

4. Preventing SQL Injection

4.1. Use Prepared Statements and Parameterized Queries

Prepared statements ensure user input is treated as data, not executable SQL.

Example of Safe Code (Using MySQL and Node.js):

```
app.get('/user', (req, res) => {
  let username = req.query.username;
  let query = 'SELECT * FROM users WHERE username = ?';
  connection.query(query, [username], (err, results) => {
    if (err) throw err;
    res.json(results);
  });
});
```

4.2. Use Object-Relational Mapping (ORM) Libraries

ORMs like Sequelize, Mongoose, and Hibernate prevent SQL injection by handling database queries securely.

4.3. Input Validation

- Allow only expected input formats (e.g., usernames should be alphanumeric).
- Use regular expressions to filter input.

4.4. Escaping User Inputs

Ensure that special SQL characters like `;`, `'`, and `--` are properly escaped.

4.5. Least Privilege Principle

Grant database accounts only the necessary permissions to minimize damage in case of an attack.

4.6. Web Application Firewalls (WAFs)

Use WAFs to detect and block malicious SQL injection attempts in real-time.

5. Detecting SQL Injection Vulnerabilities

- **Manual Testing**: Enter SQL payloads in input fields and observe behavior.
- **Automated Scanners**: Tools like SQLMap, Acunetix, and Burp Suite can scan for SQLi vulnerabilities.
- **Database Logs**: Monitor logs for suspicious queries or abnormal access patterns.

6. Real-World SQL Injection Attacks

6.1. Sony PlayStation Network Hack (2011)

Attackers exploited an SQLi vulnerability, leaking personal details of over 77 million users.

6.2. Heartland Payment Systems Breach (2008)

Hackers injected SQL commands to access sensitive financial data, leading to a massive data breach.

7. Conclusion

SQL Injection is one of the most dangerous web application vulnerabilities. Developers must adopt secure coding practices, including input validation, parameterized queries, and least privilege access, to protect applications from SQLi attacks. Regular security testing and monitoring can also help mitigate risks.

Conclusion

- **XSS** compromises user security by executing malicious scripts in browsers.
- **CSRF** tricks users into unintended actions using authenticated sessions.
- **SQL Injection** compromises databases by executing unauthorized queries.

Best Security Practices

✔ Validate and sanitize user input.
✔ Implement strong authentication and authorization.
✔ Use security headers and encryption.
✔ Keep software and dependencies updated.
✔ Monitor and log security incidents.

By implementing these best practices, developers can mitigate these common threats and ensure a more secure web application environment.

Best Practices for Secure Web Development

1. 🔐Secure Authentication

1. What is Authentication?

Authentication is the process of verifying the identity of a user or system.

Example: A user enters a username and password to log into a system.

2. Key Components of Secure Authentication

Component	Description
User Credentials	Username/email and password
Identity Verification	Checking the validity of credentials
Token Generation	JWT or session cookies post-login
Storage & Security	How passwords and tokens are managed

3. Password Security

➤ Hashing Passwords

- Never store plain-text passwords.
- Use strong one-way hashing algorithms like:
    - **bcrypt**
    - **argon2**

➤ Salting

- Add randomness to each password hash to prevent rainbow table attacks.

➢ **Password Policies**

- Minimum 8–12 characters
- Mix of uppercase, lowercase, digits, and special characters
- Prevent common or leaked passwords

4. Token-Based Authentication

After login, generate a token (e.g., JWT) that the client can send in future requests.

➢ **JSON Web Token (JWT)**

- Contains payload with user ID and expiry.
- Signed with a secret key.

5. Multi-Factor Authentication (MFA)

Adds an additional layer:

- Something you know (password)
- Something you have (OTP, mobile app)
- Something you are (biometrics)

6. Secure Session Management

- Regenerate session ID after login
- Set expiration time for sessions
- Use `HttpOnly` and `Secure` flags on cookies

7. Common Authentication Vulnerabilities

Vulnerability	Risk
Brute-force attacks	Trying multiple passwords
Credential stuffing	Using leaked credentials
Insecure password storage	Plain-text passwords
Token leakage	Exposed JWTs or cookies
Session fixation	Reusing session ID

Best Practices Summary

- Always **hash passwords** before storing
- Use **token-based authentication** (JWT) securely
- Implement **rate limiting** on login endpoints
- Use **HTTPS** for encrypted communication
- Consider **MFA** for added security
- Never log credentials or tokens

2. Authorization

1. What is Authorization?

Authorization is the process of **determining what an authenticated user is allowed to do.**

> Example: Alice is logged in (authenticated), but can only edit her own blog posts (authorized).

2. Authentication vs. Authorization

Feature	Authentication	Authorization
Definition	Verifies **who you are**	Verifies **what you can do**
Occurs	First	After authentication
Examples	Login with email/password	Access control (admin/user privileges)
Technique	Passwords, biometrics, tokens	Roles, ACLs, ownership checks

3. Types of Authorization

A. Role-Based Access Control (RBAC)

- Users are assigned roles
- Roles define permissions

B. Attribute-Based Access Control (ABAC)

- Permissions based on user, resource, environment attributes

  Example: Allow access only if the department is "HR" and time is within office hours.

C. Access Control Lists (ACLs)

- Lists define what users can do on specific resources

  Example: File access rights like read, write, execute.

4. JWT Payloads with Roles or Permissions

JWTs can include user roles:

```
const token = jwt.sign({ id: user._id, role: user.role },
'secretKey');
```

Middleware can use this info:

```
const authorizeAdmin = (req, res, next) => {

  if (req.user.role !== 'admin') return res.status(403).send('Admins
only');

  next();

};
```

5. Resource Ownership Authorization

Users may be allowed access to **only their own** data:

```
if (req.user.id !== post.ownerId) {

  return res.status(403).send("You can't edit this post");

}
```

6. Example: Middleware for Authorization

```
const authorize = (roles) => {

  return (req, res, next) => {

    if (!roles.includes(req.user.role)) {

      return res.status(403).send('Forbidden');

    }

    next();

  };

};
```

```
// Usage

app.get('/admin', authenticateToken, authorize(['admin']),
adminHandler);
```

7. Common Authorization Vulnerabilities

Vulnerability	Impact
Insecure direct object reference (IDOR)	Users can access others' data by modifying URLs
Missing authorization checks	Anyone can perform restricted actions
Over-permissioned roles	Users get more access than needed

8. Best Practices for Secure Authorization

Principle of Least Privilege
Always perform **server-side checks**
Validate resource ownership
Separate concerns: Authentication vs Authorization
Include role/permission info in secure tokens
Log access control violations

Libraries and Tools

Use Case	Tools
Role-based auth	Custom middleware, `casl`, `accesscontrol`
Token auth	`jsonwebtoken`
Fine-grained access	`opa`, `auth0`, or `keycloak`

3. Data Validation and Sanitization

1. What is Data Validation?

Data Validation ensures that incoming data meets expected **formats, types, and constraints**.

> Example: Checking if an email is in a valid format, or if a password is at least 8 characters long.

Why it's important:

- Prevents invalid or malicious input
- Protects against common attacks like **SQL Injection**, **XSS**, etc.
- Improves application stability and user experience

2. What is Data Sanitization?

Data Sanitization is the process of **cleaning input** to remove or encode harmful parts.

> Example: Escaping `<script>` tags to prevent XSS.

Why it's important:

- Prevents code injection
- Protects users from seeing or executing unsafe content
- Ensures data is stored safely in databases

3. Validation vs. Sanitization

Feature	Validation	Sanitization
Purpose	Verifies correctness of data	Removes/escapes unwanted characters
Example	`email.includes("@")`	`escapeHtml(userInput)`
Focus	Accuracy	Safety

4. Tools for Validation and Sanitization

Built-in JavaScript

```
if (typeof name !== 'string' || name.length < 2) {
  return res.status(400).send("Invalid name");
}
```

Joi (Powerful schema validation)

```
npm install joi
```

```
const Joi = require('joi');

const schema = Joi.object({
  name: Joi.string().min(2).required(),
  email: Joi.string().email().required()
});

const { error } = schema.validate(req.body);
if (error) return res.status(400).send(error.details[0].message);
```

express-validator

```
npm install express-validator
```

```
const { check, validationResult } = require('express-validator');

app.post('/user',
  [
    check('email').isEmail().normalizeEmail(),
```

```
    check('name').not().isEmpty().trim().escape()
  ],
  (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) return res.status(400).json({ errors:
errors.array() });

    res.send("Validated and sanitized");
  }
);
```

5. Common Validation Rules

Field	Rule Example
Name	Must be non-empty, letters only
Email	Must be valid email format
Password	At least 8 characters, 1 special char
Age	Must be a number between 18–100

6. Common Sanitization Techniques

Goal	Method
Remove HTML tags	`stripHtml()`
Escape HTML entities	`escape()`
Trim whitespace	`trim()`

Normalize email	`normalizeEmail()`

7. Backend Protection Example

Before saving user input:

```
const name = req.body.name.trim();

const email = req.body.email.toLowerCase().trim();
```

Preventing XSS:

```
const escapeHtml = (unsafe) =>

  unsafe.replace(/</g, "&lt;").replace(/>/g, "&gt;");
```

8. Best Practices

Validate **all** user inputs (even hidden fields)
Sanitize data before displaying it
Use libraries instead of writing regex manually
Use whitelists (e.g., allow only expected characters)
Always validate both on frontend **and** backend
Log rejected inputs for monitoring suspicious activity

Libraries Summary

Library	Use Case
`joi`	Schema-based validation
`express-validator`	Middleware-style validation & sanitation
`validator.js`	Lightweight string validation
`sanitize-html`	Remove/escape dangerous HTML

4. Use Parameterized Queries

- Prevent **SQL Injection** by never directly concatenating user inputs into SQL queries.

```
// Sequelize (Node.js)

User.findOne({ where: { email: req.body.email } });
```

5. Avoid Sensitive Data Exposure

- Store API keys, DB passwords in **environment variables** (`.env`)
- Never push `.env` files to GitHub
- Use HTTPS to encrypt data in transit
- Implement secure headers using libraries like `helmet` in Express:

```
const helmet = require('helmet');

app.use(helmet());
```

6. Error Handling

- Avoid sending stack traces or system info to clients.
- Customize error responses:

```
app.use((err, req, res, next) => {

  res.status(500).json({ error: "Internal Server Error" });

});
```

7. Regular Security Testing

- Run vulnerability scanners: **OWASP ZAP**, **Nessus**

- Perform **penetration testing**
- Keep dependencies updated (use `npm audit`)

8. Protect Against Common Web Attacks

Attack Type	Prevention
XSS	Escape output, use CSP headers
CSRF	Use anti-CSRF tokens
Clickjacking	Use `X-Frame-Options` header
Directory Traversal	Sanitize file paths
Rate Limiting	Use libraries like `express-rate-limit`

9. Use Secure Deployment Practices

- Configure firewalls and security groups
- Use HTTPS (SSL/TLS) via Let's Encrypt
- Auto-deploy only from secure repositories
- Rotate credentials regularly

10. Follow OWASP Guidelines

- Familiarize with the **OWASP Top 10** vulnerabilities
- Stay updated with their annual reports
- Integrate OWASP cheat sheets into development

HTTPS and SSL Certificates

1. Introduction

HTTP vs HTTPS

- **HTTP (HyperText Transfer Protocol):** Standard protocol for communication between client and server.
- **HTTPS (HTTP Secure):** Secure version of HTTP using encryption (via SSL/TLS).
  Why HTTPS?
  To ensure **confidentiality**, **integrity**, and **authentication** of data transmitted over the web.

2. What is SSL/TLS?

SSL (Secure Sockets Layer) and **TLS (Transport Layer Security)** are cryptographic protocols that provide **secure communication over a network**, especially for websites using HTTPS.

- **SSL:** Developed by Netscape in the 1990s.
- **TLS:** The modern, more secure replacement of SSL.
- Though the term **"SSL"** is still widely used, websites today use **TLS**.

Think of TLS as "SSL version 3 and beyond."

2.1. Why SSL/TLS is Needed

Without encryption:

- Data like passwords, card numbers, etc., can be **intercepted**.
- Attackers can perform **man-in-the-middle (MITM)** attacks.

SSL/TLS provides:

- **Confidentiality** – Prevents unauthorized reading.
- **Integrity** – Detects tampering.
- **Authentication** – Confirms server identity.

2.2. Core Concepts

Concept	Explanation
Encryption	Transforms readable data into a scrambled format.
Symmetric Key	Same key used to encrypt and decrypt data.
Asymmetric Key	Uses public and private key pairs (e.g., RSA).
Handshake	The process of initiating a secure connection.
Certificate	Proves the server's identity.

2.3. TLS/SSL Handshake (Simplified Steps)

1. **Client Hello:** Client sends supported TLS versions and a random string.
2. **Server Hello:** Server responds with certificate, chosen cipher suite, and another random string.
3. **Certificate Validation:** Client verifies the certificate with a trusted Certificate Authority (CA).
4. **Key Exchange:** A shared session key is established (via RSA, Diffie-Hellman, etc.).
5. **Secure Communication:** Data is encrypted using the session key.

2.4. SSL Certificate

An SSL certificate is a digital file containing:

- The **public key**
- The **domain name**
- **Certificate Authority's** digital signature
- **Validity period**

Types of Certificates:

Type	Validates	Use Case
DV (Domain Validated)	Domain ownership	Basic sites
OV (Org Validated)	Domain + org details	Business sites

EV (Extended Validation)	Most strict	Banks, high-security apps

2.5. Certificate Authorities (CAs)

- Trusted bodies that issue SSL certificates.
- Examples:
    - Let's Encrypt (Free)
    - DigiCert
    - Sectigo
    - GoDaddy

2.6. Cipher Suites

A **cipher suite** is a group of cryptographic algorithms used in TLS.

Example:
`TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`

It defines:

1. Key exchange algorithm (Elliptic Curve Diffie-Hellman Ephemeral-ECDHE)
2. Authentication algorithm (Rivest-Shamir-Adleman-RSA)
3. Encryption algorithm (Advanced Encryption Standard with Galois Counter Mode-AES 256 GCM)
4. Hashing algorithm (Secure Hash Algorithm-SHA384)

2.7. Vulnerabilities in SSL/TLS

Vulnerability	Description
POODLE	Exploits SSL 3.0 fallback.
Heartbleed	Bug in OpenSSL allowing memory leakage.
BEAST/CRIME	Exploit outdated cipher suites.

Best practice: Always use the latest TLS version and disable SSL 2.0/3.0.

2.8. TLS Versions

Version	Status
SSL 2.0/3.0	Deprecated
TLS 1.0/1.1	Deprecated (as of 2020)
TLS 1.2	Widely used
TLS 1.3	Fast, secure, modern standard

2.9. Tools for Testing SSL/TLS

- SSL Labs
- `openssl` CLI
- Chrome DevTools → Security tab

3. Implementing HTTPS in Web Applications

For Express.js (Node.js backend):

```
const https = require('https');

const fs = require('fs');

const express = require('express');



const app = express();



const options = {

  key: fs.readFileSync('key.pem'),//contains the private key for your SSL/TLS certificate

  cert: fs.readFileSync('cert.pem')//contains the SSL/TLS certificate

};
```

```
https.createServer(options, app).listen(443, () => {
  console.log('Secure server running on https://localhost');
});
```

In production:

- Use **NGINX**, **Apache**, or **Heroku** to manage SSL. NGINX is a powerful and versatile open-source software that is widely used as a web server, reverse proxy, load balancer, HTTP cache, and mail proxy.

- Use **Let's Encrypt** + Certbot for free SSL.

4. Importance of HTTPS

Concern	Without HTTPS	With HTTPS
Data Privacy	Data sent in plain text	Encrypted
Authentication	Easy to spoof sites	Server identity is verified
SEO	Lower ranking	Google favors HTTPS
Browser Warnings	Shown	None

5. Risks Without HTTPS

- **Man-in-the-Middle Attacks (MITM):** Attackers intercept data.
- **Phishing:** Unsecured sites easier to impersonate.
- **Data Leakage:** Credentials or PII can be stolen.

Role-Based Access Control (RBAC)

1. Introduction

Role-Based Access Control (RBAC) is a method for **restricting system access** to **authorized users** based on their **roles** within an organization or application.

> "Don't assign permissions to users directly; assign roles to users and permissions to roles."

2. Key Concepts in RBAC

Term	Description
User	An individual who interacts with the system.
Role	A named job function (e.g., admin, editor, viewer).
Permission	Approval to perform certain operations (e.g., read, write, delete).
Resource	An object the permissions act upon (e.g., file, API endpoint).

3. How RBAC Works (Flow)

1. Users are assigned **one or more roles**.
2. Each role has a **set of permissions**.
3. When a user attempts to perform an action, the system checks: Does the user's role have the required permission?

Example:

- **Admin** → Can read, write, delete users
- **Editor** → Can read and write content
- **Viewer** → Can only read content

4. Benefits of RBAC

- **Scalability:** Easily manage large numbers of users.
- **Security:** Principle of least privilege.

- **Simplicity:** Manage access through roles rather than individuals.
- **Compliance:** Helps with GDPR, HIPAA, etc.

5. RBAC vs ABAC vs ACL

Feature	RBAC	ABAC (Attribute-Based)	ACL (Access Control List)
Basis	Role	User + resource attributes	User-specific permissions
Example	"Editor can write"	"User in HR can edit salary data"	"User X can read file Y"
Flexibility	Moderate	High	Low

6. Components of RBAC

Component	Function
Role	A named collection of permissions.
Permission	Specific access rights (e.g., read, write, delete).
Session	Represents a user's active connection with assigned roles.
Constraints	Additional restrictions (e.g., role cannot be held by same person).

7. Implementing RBAC in Web Applications

Step-by-Step:

1. Define roles and permissions in your database.
2. Assign roles to users.
3. Use middleware to authorize based on roles.
4. Protect routes or actions accordingly.

8. Example: Node.js + Express + JWT

```
const authorizeRoles = (...roles) => {
```

```
  return (req, res, next) => {
    if (!roles.includes(req.user.role)) {
      return res.status(403).send('Access Denied: Insufficient Role');
    }
    next();
  };
};


// Usage
app.get('/admin', authenticateToken, authorizeRoles('admin'), (req,
res) => {
  res.send('Welcome Admin');
});
```

Here:

- `authenticateToken` ensures user is logged in.
- `authorizeRoles` checks if the user's role is allowed.
- The ... syntax allows this function to accept any number of arguments, and these arguments will be collected into an array named roles.

9. Sample Database Structure

Users Table:

id	username	role
1	alice	admin
2	bob	editor

Roles Table:

role	permissions
admin	[read, write, delete]
editor	[read, write]
viewer	[read]

10. Best Practices

- Use least privilege principle.
- Avoid hardcoding roles in code — use DB or config files.
- Regularly audit roles and permissions.
- Log access and permission changes.
- Use middleware for reusable role checks.

11. Use Cases

- **Enterprise systems**: Employees with different access levels.
- **CMS platforms**: Admins, editors, and readers.
- **E-commerce**: Customers vs. vendors vs. staff.
- **Education portals**: Students, faculty, admins.