Web Application Security

Dr. Digvijaysinh Rathod
Professor
School of Cyber Security and Digital Forensics
National Forensic Sciences University

digvijay.rathod@nfsu.ac.in

Web Service concepts

Introduction

- ✓ A Web Service is a secure, standardized method that allows two applications (client and server) to communicate over a network — typically the internet — using open protocols such as HTTP/HTTPS and data formats like XML or JSON.

Core Idea

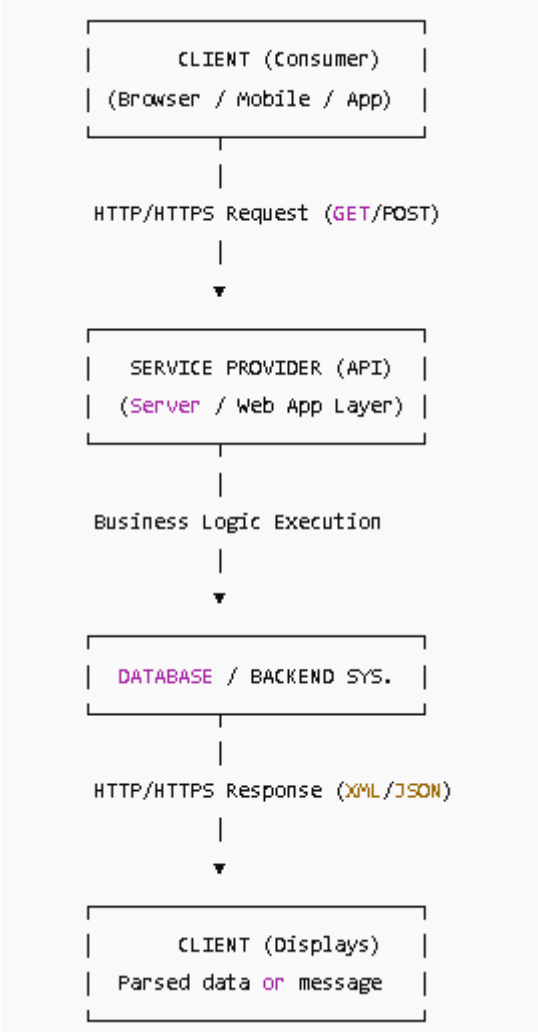
- ✓ Web Services enable machine-to-machine communication — for example, a banking app calling a payment gateway API, or a cybersecurity dashboard fetching live threat-intel feeds.They follow a request–response model, where one system requests a resource or service, and another system provides it through standardized protocols.

Types of Web Services

Type	Description	Common Security Layer
SOAP (Simple Object Access Protocol)	XML-based, uses WSDL for service definition, highly structured, enterprise-grade.	WS-Security (signing, encryption, token authentication)
REST (Representational State Transfer)	HTTP-based, lightweight, stateless, uses JSON/XML, scalable and fast.	HTTPS/TLS, OAuth 2.0, JWT tokens
GraphQL / gRPC (Modern APIs)	Efficient modern alternatives, used for microservices.	TLS, mutual authentication

Web Service Flow

Below is the logical **flow of communication** between a client and a service provider:



Summary

Concept	Key Takeaway
Web Service	Bridge between applications using HTTP/HTTPS
SOAP	Structured XML-based protocol, secure but heavy
REST	Lightweight, JSON-based, most common today
Security	Encryption, authentication, access control essential
Tools	Burp, ZAP, Postman for testing and hardening

REST Concepts

- REST (Representational State Transfer) is an architectural style used for designing scalable and lightweight web services.
- It enables communication between client and server using standard HTTP/HTTPS methods and supports simple data formats like JSON and XML.

REST is widely used in:

- Mobile applications
- Cloud platforms
- Microservices architectures
- Threat intelligence APIs
- Cybersecurity tools (SIEM, SOAR, SOC dashboards)

a) Client–Server Architecture

The client and server are independent.

Client → sends request

Server → processes and returns response

b) Statelessness

Each request is independent. The server does not store client session information between requests.

c) Cacheability

Responses can be cached to improve performance.

d) Uniform Interface

- REST follows consistent interfaces such as:
- Resource identification (URLs)Standard HTTP methods
- Standard data formats (JSON/XML)

e) Layered System

- A REST system may use multiple layers like security gateways, load balancers, proxies, API gateways, etc.

REST HTTP Methods

Method	Purpose	Example
GET	Retrieve data	/users/10
POST	Create new data	/users
PUT	Update entire data	/users/10
PATCH	Update partial data	/users/10
DELETE	Remove data	/users/10

Resources are nouns, not verbs.

Example:

- GET /books
- POST /books
- GET /books/25
- DELETE /books/25

REST uses lightweight, web-friendly formats:

Example:

- JSON (most common)
- XML
- YAML
- Text

JSON Example:

JSON :

```
{
  "id": 10,
  "name": "Dr. Bhargav Patel",
  "role": "Cybersecurity Research"
}
```

REST Response Codes (Important for Cyber Security)

Code	Meaning	Security Relevance
200	Success	Normal behavior
400	Bad Request	Input validation errors
401	Unauthorized	Authentication failed
403	Forbidden	Authorization failed
404	Not Found	Can hide sensitive endpoints
429	Too Many Requests	Rate limiting during brute force
500	Server Error	Misconfiguration, exception leaks

REST vs SOAP (Quick Comparison)

Feature	REST	SOAP
Format	JSON/XML	XML only
Complexity	Simple	Heavy
Speed	Faster	Slower
Security	HTTPS, OAuth, JWT	WS-Security (Enterprise grade)
Use case	Modern apps	Banking/enterprise

REST Security Concepts (For Cyber Security Students)

a) HTTPS Everywhere

Use **TLS 1.2/1.3** to secure data in transit.

b) Authentication

- API Keys
- OAuth 2.0
- JWT Tokens
- HMAC Tokens

c) Authorization

Role-Based Access Control (RBAC):

- Admin
- Analyst
- User

d) Input Validation

Prevent:

- SQL Injection
- Command Injection
- JSON Injection

e) Rate Limiting & Throttling

Stops:

- Brute force attacks
- API abuse

f) Logging & Monitoring

Essential for:

- Incident detection
- SIEM integration
- Threat hunting

Sensitive data in GET

- In REST APIs, sending sensitive data through a GET request is unsafe because GET parameters appear directly in the URL, which is logged by browsers, servers, proxies, and monitoring tools.
- REST endpoints are often public-facing, so exposing secrets in URLs can lead to severe data leaks. For example, accessing a REST API like:

Sensitive data in GET

GET /login?username=admin&password=Secret@123

Or

GET /verify?otp=452198

Similarly, calling a REST endpoint like:

GET /user?token=ABCD1234XYZ

Sensitive data in GET

GET /login?username=admin&password=Secret@123

Or

GET /verify?otp=452198

Similarly, calling a REST endpoint like:

GET /user?token=ABCD1234XYZ

places the password and OTP in logs:

- Browser History
- Server Access Logs
- Reverse Proxies / Load Balancers (Nginx, HAProxy)
- Web Application Firewalls (WAF)
- API Gateway Logs
- Network Monitoring Tools

Anyone with access to these logs can use the token to impersonate the user.

Weak Auth Tokens & IDOR

Weak Authentication Tokens

What is it?

A weak authentication token is a session ID or access token that is **predictable, easily guessable, short, or not securely generated.**

- Tokens are supposed to uniquely identify and authenticate a user.
- When they are weak, attackers can guess or brute-force them and take over user sessions.

Suppose a REST API gives tokens like:

token=12345 token=12346 or Or token=abc001, abc002, abc003

These are sequential, so an attacker can guess the next one.

Simple IDOR Example - Imagine this REST endpoint:

GET /user/profile?id=101

GET /user/profile?id=102

And the system returns another user's data, this is IDOR.

How Attackers Exploit Weak Tokens

- Brute-force the next token
- Predict tokens based on patterns (timestamp-based, incremental)
- Replay stolen tokens
- Hijack active sessions

Impact

- Complete account takeover
- Privilege escalation
- Unauthorized transactions
-

Summary

Concept	Meaning	Risk
Weak Auth Token	Predictable/guessable session or API token	Session hijacking
IDOR	Unauthorized access by changing object identifiers	Data leakage, privilege escalation

Leaky APIs and Insecure Data Storage

What Are Leaky APIs?

- A Leaky API is a REST endpoint that exposes more data than necessary or reveals internal system details due to weak access control, poor filtering, or misconfigurations.
- In a REST-based system, APIs often return structured data (usually JSON).
- When the API does not validate who is requesting and what they should receive, attackers can extract confidential information.

Exposing entire user data

GET /api/user/101

Response:


```
{
  "id": 101,
  "name": "Rohit",
  "email": "rohit@example.com",
  "password": "hashed_value",
  "aadhar": "1234-5678-9123",
  "account_balance": 27000
}
```

Exposing entire user data

GET /api/user/101

Response:


```
{
  "id": 101,
  "name": "Rohit",
  "email": "rohit@example.com",
  "password": "hashed_value",
  "aadhar": "1234-5678-9123",
  "account_balance": 27000
}
```


A normal user should never receive another user's private data.

Debug info exposure

GET /api/login?user=admin

Response:

{

"error": "SQL error: column 'password' does not exist in table users_temp"

}

This reveals database structure → helpful for attackers.

Impact of Leaky APIs

- Personal data disclosure (PII leak)
- Account takeover
- Business logic abuse
- IDOR exploitation
- Recon phase jumps straight to exploitation

Insecure Data Storage in REST APIs

- REST APIs often work with mobile apps, web apps, and IoT devices.
- If sensitive data is stored without encryption or in predictable locations, attackers can retrieve it.
- **Where Insecure Data Storage Occurs**
    - Local storage of mobile apps (Android/iOS)
    - Browser localStorage/sessionStorage
    - Unencrypted server-side database tables
    - Log files storing sensitive values
    - Caches or backups exposed publicly
    - Hardcoded secrets in API code

Examples of Insecure Data Storage

1. Storing API tokens in localStorage

```
localStorage.setItem("authToken", "ABCD1234XYZ");
```

If XSS occurs → attacker steals token.

2. Storing plaintext credentials in database

email: user@example.com

password: admin123

If DB leaks → credentials are exposed instantly.

3. Logging sensitive data

Payment failed for card=4580-****-****-1234 cvv=123

Logs become an attack surface.

4. Mobile app storing tokens in files

/Android/data/app/cache/token.txt

Any malicious app can read it.

Impact of Insecure Data Storage

- Token theft → session hijacking
- Identity theft (PII exposure)
- Financial fraud
- Compromised devices or apps
- Attackers chaining multiple vulnerabilities (XSS → token theft → API misuse)

Prevent Leaky APIs

- Enforce strict authentication and authorization
- Validate user roles before returning data
- Implement data filtering (send only required fields)
- Disable debug messages in production
- Perform proper input validation
- Use rate limiting to prevent mass data harvesting

Prevent Insecure Storage

- Encrypt data at rest (AES-256)
- Encrypt tokens and credentials
- Never store sensitive data in local Storage if avoidable
- Use secure server-side storage
- Mask or hash sensitive fields
- Secure logs and avoid logging credentials
- Rotate keys and tokens regularly

Mobile Phone Security

Dr. Digvijaysinh Rathod
Professor
School of Cyber Security and Digital Forensics
National Forensic Sciences University

digvijay.rathod@nfsu.ac.in