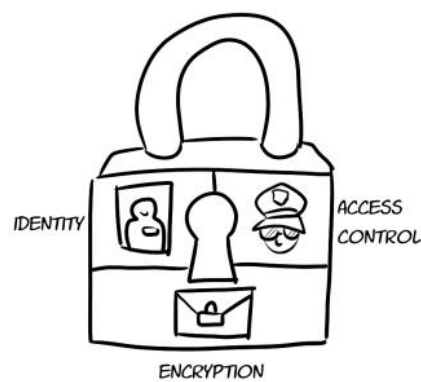


UNDERSTANDING API SECURITY

Ahmad Hamo
17-5-2025

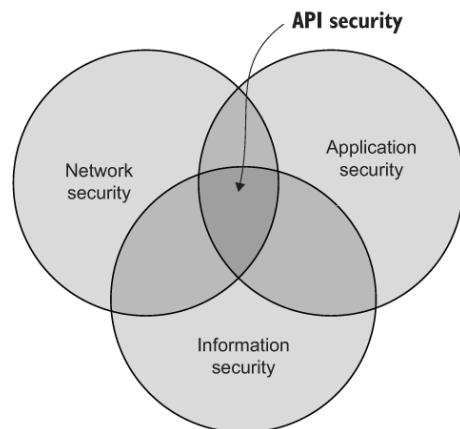


Introduction to API Security

- What is APIs Security?
- APIs Security Mechanisms
- Authentication
 - Authentication Methods
- Authorization
 - OAuth 2.0
 - Access Control Types
- API Security Best Practices
- Open Web Application Security Project (OWASP)
 - OWASP Top 10 API Security Risks

What is API Security?

- API Security refers to methods that prevent malicious attacks on application program interfaces (API).
- API security lies at the intersection of three security areas:
 - Information security,
 - Network security, and
 - Application security.



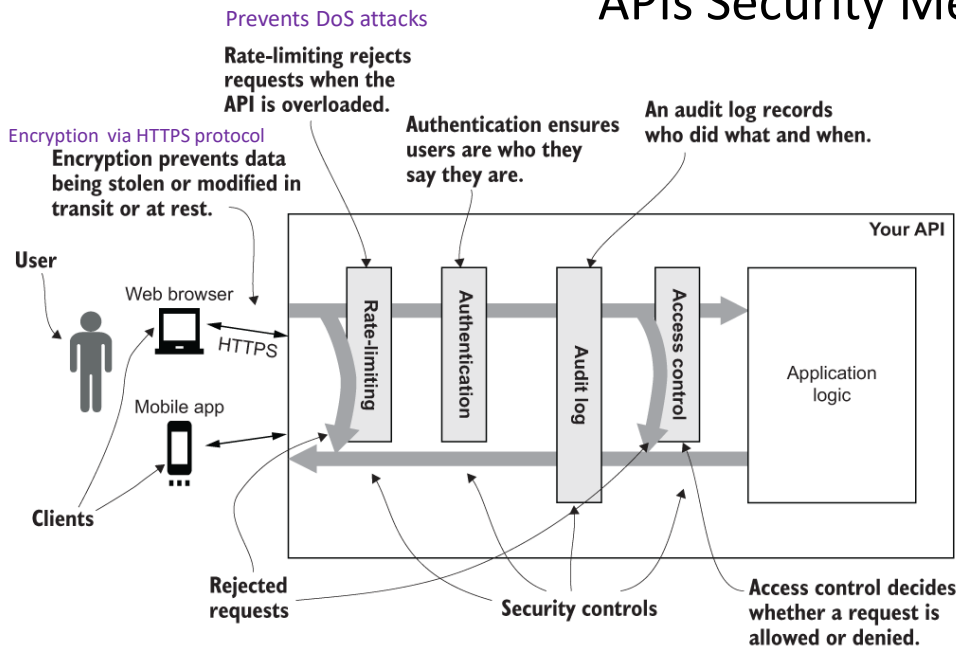
Aligning API Security with Compliance Requirements

- In today's digital environment, APIs are subject to various regulatory and compliance standards that dictate how data should be handled and protected. These standards include the **GDPR**, **HIPAA**, FedRAMP, and other frameworks.
- Compliance is not just a legal necessity; it's a crucial aspect of building trust and credibility with users and stakeholders. Compliance means ensuring that all data exchanges, storage, and processing meet the specified standards for APIs

API Security with Compliance Requirements

- **Security-First Design:** Security begins at the code level, and poor security practices at the beginning of the process cannot be solved on the backend by security band-aids.
- **Data Protection and Privacy:** Implementing measures to safeguard personal and sensitive data, as laws like **GDPR** require. This includes data encryption, secure data transfer, and ensuring data is processed for legitimate purposes.
- **Access Control:** Ensuring that only authorized individuals or systems can access or manipulate data. This is particularly relevant for APIs dealing with health-related information, where HIPAA compliance requires strict access controls.
- **Audit Trails and Record-Keeping:** Maintaining comprehensive logs of data access and transfers through APIs, a requirement under various compliance frameworks. This aids in auditability and transparency.

APIs Security Mechanisms



Security Mechanisms

- **Encryption** ensures that data can't be read by unauthorized parties, either when it is being transmitted from the API to a client or at rest in a database or filesystem.
- **Authentication** is the process of ensuring that your users and clients are who they say they are.
- Access control (also known as **authorization**) is the process of ensuring that every request made to your API is appropriately authorized.
- **Audit logging** is used to ensure that all operations are recorded to allow accountability and proper monitoring of the API.
- **Rate-limiting** is used to prevent any one user (or group of users) using all of the resources and preventing access for legitimate users.

Authentication vs. authorization

Authentication

Determines whether users are who they claim to be

Challenge the user to validate credentials (for example, through passwords, answers to security questions, or facial recognition)

Happen before authorization

Generally, transmits info through an ID Token.

ID tokens are used in token-based authentication to cache user profile information and provide it to a client application,

Generally governed by the OpenID Connect (OIDC) protocol

Example: Employees in a company are required to authenticate through the network before accessing their company email

Authorization

Determines what users can and cannot access

Verifies whether access is allowed through policies and rules

Happen after successful authentication

Generally, transmits information through an Access

Token

Generally governed by the OAuth 2.0 framework

Example: After an employee successfully authenticates, the system determines what information the employees are allowed to access

What is Authentication?

- **Definition:** Who are you?
- Authentication is something you ***know***, something you ***have***, or something you ***are***.
- **Modern Extensions:**
 - **Somewhere You Are** (Location Factor)
 - **Something You Do** (Behavioural Biometrics)
- Example: Logging in with username/password.
- Without authentication:
 - Anyone can access anything.
 - High risk of abuse and data manipulation.

Why Use Authentication?

- Prevent unauthorized access.
- Track user behavior.
- Enable personalized experiences.
- Required for secure public APIs.
- Helps in:
 - User analytics
 - Secure transactions
 - Role-based permissions

Authentication Methods Overview

- **HTTP Basic Auth**: user needs to provide user ID and password.
- **API Key**: user needs to provide a unique identifier configured for each API
- **Token**: generated by an Identity Provider (IdP)
- **Session Cookies**
- **OpenID Connect**

Basic Auth – The Double-Edged Sword

- Base64-encoded **username:password** in headers.

- Example:

`Authorization: Basic dXNlcjpwQHNzdzByZA==`

Problems:

- ✗ Credentials exposed if intercepted.
- ✗ No encryption without HTTPS.
- ✗ You have to send username: password by each request.
- **Use Case:** Legacy systems/internal tools (with HTTPS).

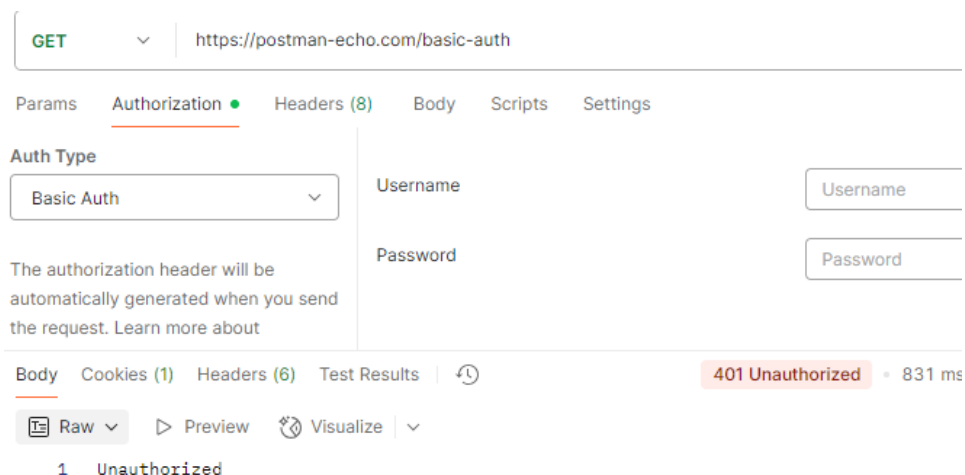
Introduction to Basic Authentication

- **Definition:**
 - **Basic:** Simple.
 - **Authentication:** Proving identity (e.g., username/password).
 - Combined: A simple way to verify identity for API access.
- **How It Works:**
 - APIs require identity verification before granting access.
 - Basic Auth sends a username and password over HTTPS (encrypted).
 - **Insecure Aspect:** Repeatedly transmitting credentials increases interception risk.
- **Historical Context:**
 - Introduced in 1999.
 - Still used today but often replaced by more secure methods (e.g., OAuth).

Using Basic Auth in Practice

- **Demo with Postman:**
 1. Access an API (e.g., Postman Echo) **without** credentials → “Unauthorized” error.
 2. Add username (postman) and password (password) → “Authenticated: true” response.
- **Technical Details:**
 - Credentials are sent in the Authorization header as a **base64-encoded string**.
 - Example: Authorization: Basic b3N0bWFuOnBhc3N3b3Jk (decodes to postman:password).
- **Security Limitation:**
 - Base64 is **not encryption**—credentials can be decoded easily.
 - Always use HTTPS to prevent interception.

Demo – postman – without cred.



POSTMAN ECHO

GET Basic Auth

https://postman-echo.com/basic-auth

This endpoint simulates a **basic-auth** returns a status code of: 200 ok | only unauthorized.

Username: postman
Password: password

Auth Type
Basic Auth

The authorization header will be automatically generated when you send the request. Learn more about

Body Cookies (1) Headers (7) Test Results 200 OK • 698 ms

{ } JSON Preview Visualize

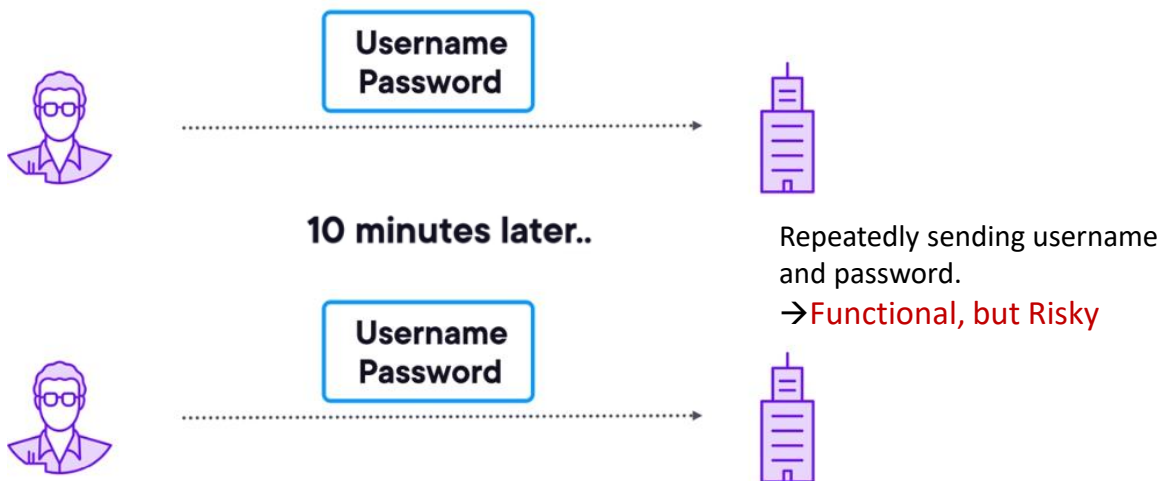
```
1 {
2   |   "authenticated": true
3   }
```

Params **Authorization** **Headers (8)** **Body** **Scripts** **Settings**

	Key	Value
☒	Authorization	Basic cG9zdG1hbJpwYXNzd29yZA==

Demo – Postman with auth.

Why not use HTTP Basic Authentication?



API Keys – Convenience at a Cost

How They're Used:

- Query strings: `?api_key=123` (✗ Risky).
- Headers: `X-API-Key: 9038-20380-9398` (✓ Better).

Security Flaws:

-  Keys leak in logs/URLs.
-  No built-in expiration.

Mitigation: (1) Rotate keys frequently; (2) use IP whitelisting.

API Keys vs. HTTP Basic Authentication

<u>Feature</u>	<u>API Key</u>	<u>HTTP Basic Authentication</u>
Authentication Method	A static, predefined key sent in headers/URL.	A username & password encoded in Base64.
How It's Sent	Usually in headers (<code>x-api-key: abc123</code>) or URLs (<code>?api_key=value</code>).	In the Authorization header (<code>Basic base64(username:password)</code>).
Security Level	Low (if leaked, full access is granted).	Very low (Base64 is easily decoded; should always use HTTPS).
Usage	Common for machine-to-machine APIs (e.g., weather APIs, payment gateways).	Rarely used today; replaced by tokens (OAuth, JWT) for user auth.
Example Request	<code>http GET /data HTTP/1.1 Host: api.example.com x-api-key: abc123</code>	<code>http GET /data HTTP/1.1 Host: api.example.com Authorization: Basic dXNlcjoxMjM=</code>
Best For	Server-to-server communication where simplicity is key.	Legacy systems; rarely recommended for modern APIs.

Bearer Tokens – The Modern Standard

- Stateless JWT sent in headers:

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

Advantages:

-  Scalable (no server-side sessions).
-  Fine-grained expiration/scopes.

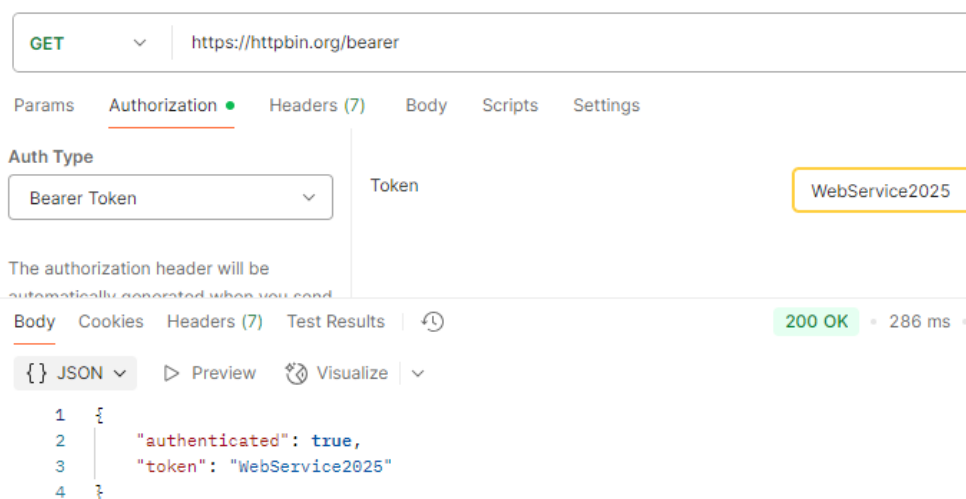
Introduction to Bearer Tokens

- **Definition:**
 - A bearer token is a credential that grants access to an API.
 - Split into two parts:
 - **Bearer:** The holder of the token (e.g., “I am a bearer of a pen”).
 - **Token:** A tool to gain access (e.g., casino chips or subway tokens).
- **Key Characteristics:**
 - Originated in 2012 as part of OAuth 2.0 (RFC 6750).
 - No authentication or identification—anyone with the token can use it.
 - Must be protected from unauthorized access (use HTTPS for encryption).
- **Example:**
 - Sending a bearer token to an API grants access, regardless of user identity.

How Bearer Tokens Work

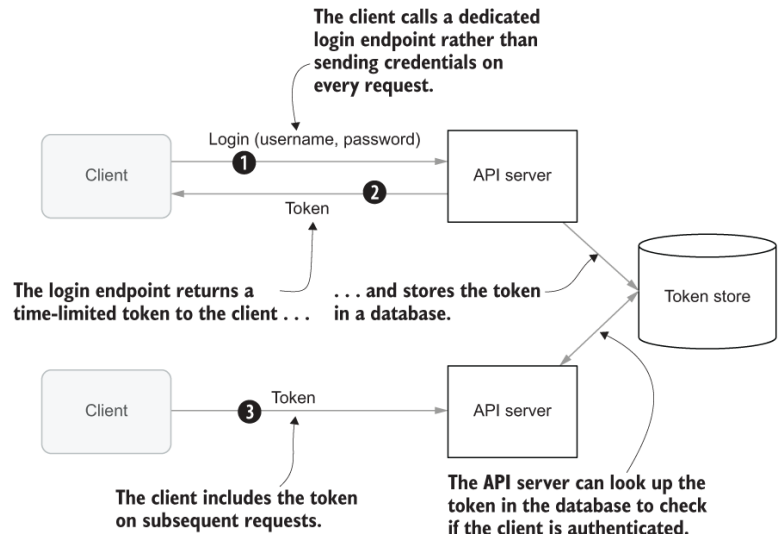
- **Usage in APIs:**
 - Sent in the Authorization header with the format: `Bearer <token_value>`.
 - Example: `Authorization: Bearer shy%^bgfdd`.
- **Demo with httpbin.org:**
 1. Call `/bearer` without a token → Unauthorized.
 2. Add token in Postman (e.g., "somepassword") → Access granted.
- **RFC 6749 Specification:**
 - Mandates the `Bearer` prefix in the header.
 - Tokens must be protected in storage and transport to prevent misuse.
- **Access Token vs. Bearer Token:**
 - Access tokens have an expiration; bearer tokens do not (unless revoked).

Bearer - Demo



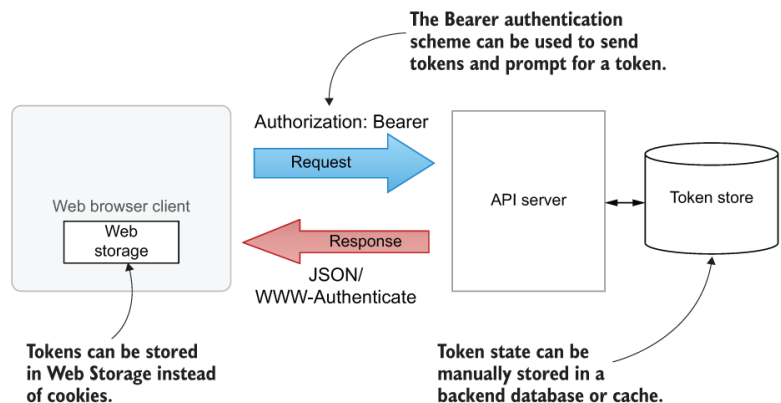
Token-Based Authentication

1. Client makes request to a dedicated login endpoint with the user's credentials.
2. In response, the login endpoint returns a time-limited token.
3. The client then sends that token on requests to other API endpoints that use it to authenticate the user.
4. API endpoints can validate the token by looking it up in the token database.



Web- and Token Storage

- **Web Storage** for storing tokens on the client.
- The **Bearer authentication** scheme provides a standard way to communicate tokens from the client to the API, and
- a **Token Store** can be manually implemented on the backend.

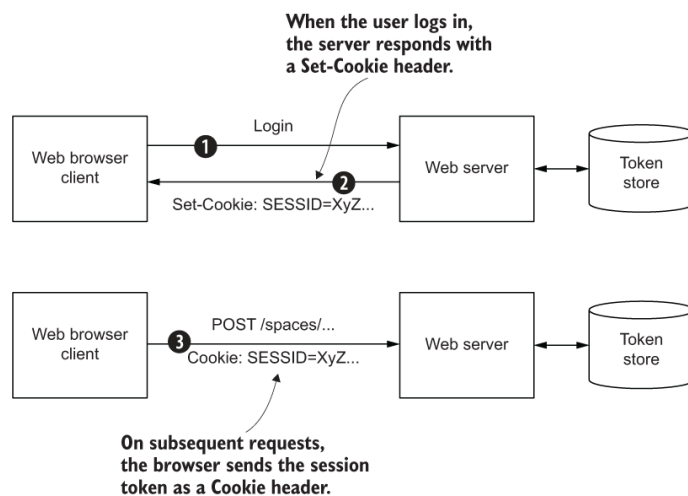


API Keys vs. Tokens

<u>Feature</u>	<u>API Key</u>	<u>Token (e.g., JWT, OAuth)</u>
Purpose	Simple authentication, often used for API access.	Authentication & authorization (may include user permissions).
Structure	Usually a long, random string (e.g., x-api-key: abc123...).	Can be structured (e.g., JSON Web Token—JWT) with encoded data.
Security	Less secure—sent as-is; if leaked, it grants full access.	More secure—can expire, be scoped, or include signatures.
Usage	Typically added in headers or URLs (e.g., ?api_key=xyz).	Sent in headers (e.g., Authorization: Bearer <token>).
Lifespan	Often long-lived or manually rotated.	Can be short-lived (e.g., expiring in hours) or refreshable.
Example	x-api-key: 123abc...	Authorization: Bearer eyJhbGciOi...

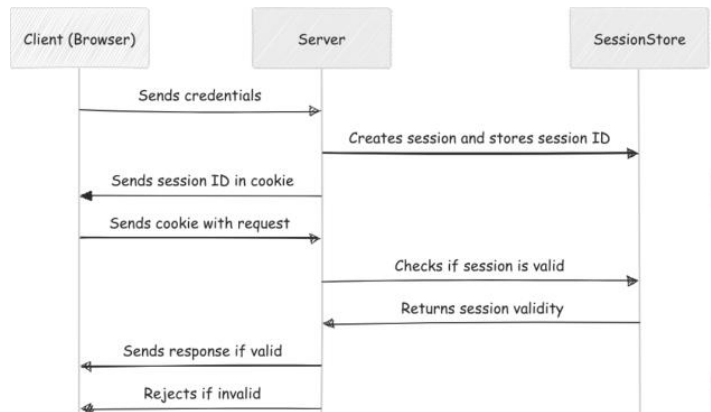
Session Cookie Authentication

1. User logs in
2. Server sends a **Set-Cookie header** on the response with a session token.
3. On subsequent requests to the same server, the **browser** will **send** the session token **back** in a **Cookie header**,
4. which the server can then look up in the **token store** to access the **session state**.



Authentication using Session Management

- OG Method (Old Generation)
- Sessions are Stateful
- **Cons.:** scalability problems with large number of users



JWT is preferred by modern apps
, because it scales better.

OpenID Connect – Identity + OAuth

- **OpenID Connect** is an identity layer built on top of **OAuth 2.0** that enables secure user authentication and single sign-on (SSO).
- It allows applications to verify a user's identity using an **Identity Provider (IdP)** (e.g., Google, Microsoft, Facebook, ..)
- **Features:**
 - Authentication (Not Just Authorization)
 - Standardized Identity Data (ID Token + additional user details)
 - Single Sign-On (SSO)
 - Users log in once with an IdP (e.g., Google) and access multiple apps without re-entering credentials.

OpenID Connect – Identity + OAuth

What It Adds:

- Standardized **ID Token** (JWT) with user claims:

```
{ "name": "Jane Doe", "email": "jane@example.com" }
```

- **Discovery Endpoint:**

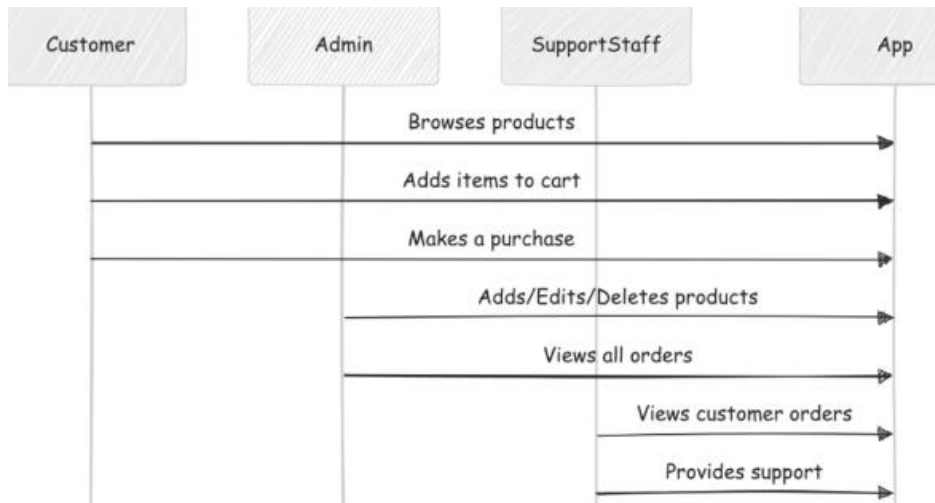
</.well-known/openid-configuration> for metadata.

Use Case: Single Sign-On (SSO) across applications.

What is Authorization?

- **Definition:** What do you want to do?
- Determines what actions a user can perform.
- Example:
 - Read-only users vs. Admins
- Often follows authentication.

Example Authorization



Introducing OAuth2.0

- Open standard for authorization.
- Allows **third-party** services to access resources on behalf of a user.
- OAuth 2.0 defines multiple **grant types** (flows) for different use cases.
- Each **flow** determines how an application obtains an **access token**:
 - Authorization Code : for Web apps with a backend
 - Implicit: Old SPAs (avoid)
 - Password: Trusted first-party apps.
 - Client Credentials: Server-to-server APIs (no user).

OAuth 2.0 – The Gold Standard

Core Components:

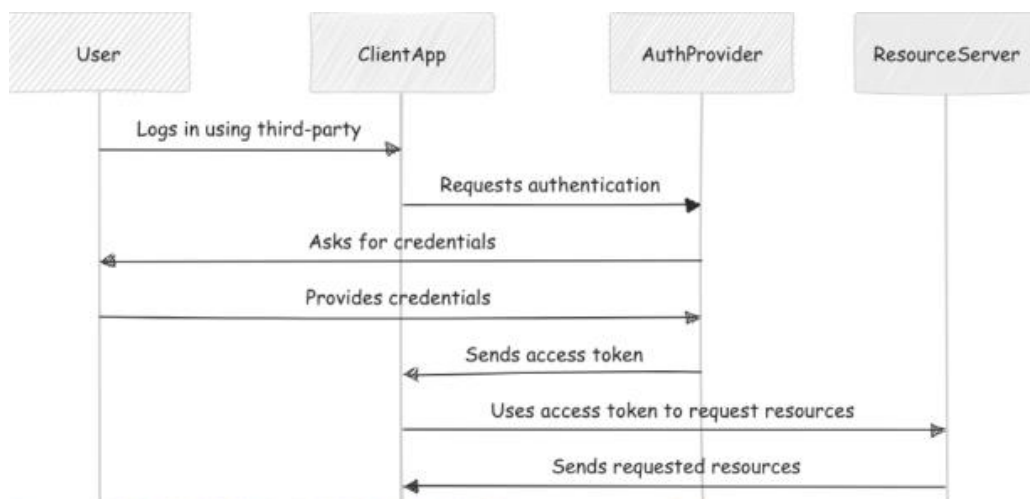
1. **Access Token:** Short-lived (minutes/hours).
2. **Refresh Token:** Long-lived (days/months).

Why It's Secure:

- 🔑 Tokens are **scope-limited** (e.g., `read:contacts`).
- ⌚ Tokens **expire**, reducing breach impact.

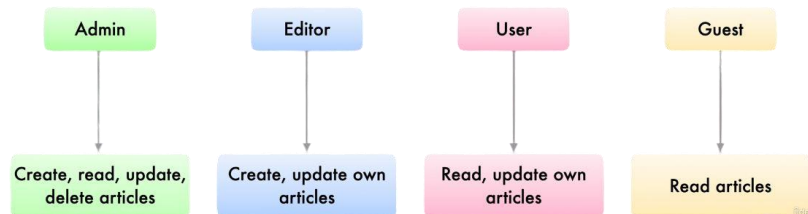
OAuth flow: (User → Auth Server → Client).

Authentication using OAuth2



Role Based Access Control - RBAC

- Away for Defining Authorization
- Assigning Roles and Actions:
 - Define Roles (admin, editor, guest, ...)
 - Define Permissions to each Role (CRUD, ...)
 - Assign Roles to the users
 - A user can have one or more roles
 - Example

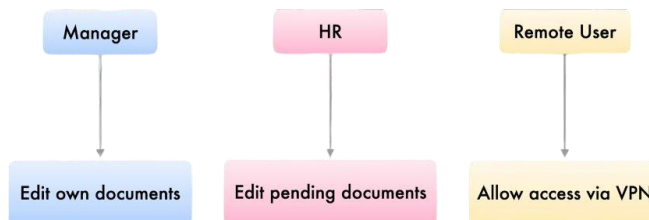


Attribute Based Access Control - ABAC

- Away for **granular** defining Authorization
- Flexible: Access based on Attributes of the user, resource or environment
- User Attributes:
- Resource Attributes:
- Environmental Attributes:



- Example:



Steps for Individual Auth

1. Collect username/password via form or header.
2. Hash password using `bcrypt`.
3. Compare hash with stored value in DB.
4. If match → create JWT token.
5. Return token to client.
6. Client uses token in future requests.

Encryption & Hashing

- **Encryption** scrambles data that can be decoded with a key. The intent is to pass the information to another party, and the recipient will use keys to decipher the data.
- The main example of this is [HTTPS with TLS](#), where a certificate is used and installed on a server to allow a client and the server to generate a symmetric keypair for communication.
- **Hashing** also scrambles data, but the intent is to prove its authenticity. Administrators can run a check on hashed data to determine the contents haven't been touched or altered while in storage. No deciphering key exists.

API Security Best Practices

(1) Use Encryption

Organisations using APIs which routinely exchange sensitive data (such as login credentials, credit card, social security, banking information, health information), TLS encryption should be considered essential.

(2) Use Strong Authentication and Authorisation

Poor authentication or non-existent authorisation are major issues with many publicly available APIs. They provide an entry point to an organisation's databases, it's critical that the organisation strictly controls access to them.

(3) Prioritise API Security

- One of the key battles with API security is that it is sometimes considered when it is too late.
- Another issue is that API security is someone else's issue.

(4) Be Aware of all APIs

Whether an organisation has one or hundreds of publicly available APIs, building an inventory of all the APIs so that they can be secured and managed using perimeter scans, along with insight from developers, to create an inventory of all APIs is the best place to begin.

(5) Practice the Principle of Least Privilege

- It is a security principle that effectively holds that the subjects/entities (users, processes, programs, systems, devices) are only ever granted the minimum necessary access rights to complete a required function.
- This is something that applies more broadly to all IT systems, and it should be applied equally to APIs.

(6) Don't Expose More than is Necessary

One of the biggest issues with APIs is that they are designed to allow for the exchange of data. In short, APIs can reveal more information than necessary.

(7) Always Validate Input

- All input data must be validated, and anything that is too big or doesn't comply with required checks must be rejected.
- Ensure injection exploits are prevented.

(8) Ensure all OWASP Vulnerabilities are Secured

- The OWASP (Open Web Application Security Project) Top 10 is a list of the ten worst vulnerabilities.
- It should be ensured that systems have been secured for these vulnerabilities.

(9) Implement an API Management Solution

Good API management solution will help make sense of API data, and establish secure API practices.

(10) Get Help from Security Experts

The world of cybersecurity is enormously challenging and continuously evolving with new threats appearing daily. Partnering with experienced cybersecurity experts who know exactly what's needed to secure enterprise level API based systems, is often the best API security solution.

OWASP Overview

Open Web Application Security Project (OWASP)

- Non-profit organization
- Vision: "No more insecure software"
- Mission: Promote secure software through education and collaboration
- Known for Top 10 lists (Web & API security risks)

What is OWASP?

- Community-led open source project
- 250+ local chapters globally
- Provides standards, tools, and documentation
- Resources available at owasp.org

Unsafe Consumption of APIs

Risk #10

- Developers trust 3rd-party API data without validation

Attack Vector: Compromised 3rd-party APIs

Weakness: 3rd-party APIs are trusted and not verified

Impact:

- Sensitive data exposure
- Injection attacks
- Data leakage

Vulnerabilities (Unsafe Consumption)

- Unencrypted API interactions
- No data validation/sanitization
- Blindly following API redirects
- No timeouts for 3rd-party service interactions

Mitigation Strategies

- Use encryption (SSL/TLS)
- Validate & sanitize all data
- Implement allow lists for input validation
- Use strict data type constraints
- Deploy API gateways for filtering/monitoring

Improper Inventory Management

Risk #9

- Poor tracking of API versions/endpoints

Attack Vector: Old/deprecated APIs with weak security

Impact:

- Sensitive data exposure
- Remote server takeover

Vulnerabilities (Inventory)

Documentation Blind Spots:

- Missing environment details
- Outdated documentation
- No retirement strategy

Data Flow Blind Spots:

- Unjustified 3rd-party data sharing
- No visibility into data flows

Mitigation Strategies

- Maintain up-to-date API inventory
- Document all API aspects comprehensively
- Limit access to documentation
- Decommission unused APIs
- Avoid production data in non-production environments

Security Misconfiguration

Risk #8 APIs not securely configured

Attack Vector: Unpatched flaws, unsecured endpoints

Impact:

- Sensitive data exposure
- Full system compromise

Vulnerabilities (Misconfiguration):

- Missing security hardening
- Outdated security patches
- No TLS/CORS policies
- Overly verbose error messages

Mitigation Strategies

- Secure APIs early in development
- Review/update security configurations
- Use automated tools for detection/remediation

Server-Side Request Forgery (SSRF)

Risk #7 Attacker manipulates server-side requests

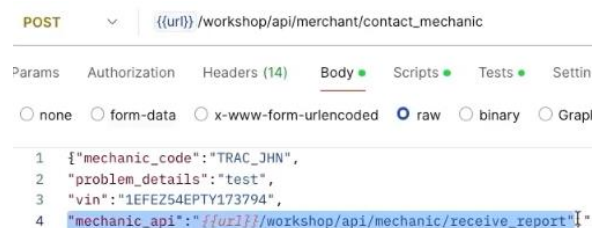
Attack Vector: API that accesses client-provided URIs

Impact:

- Sensitive data exposure
- Remote code execution

Vulnerabilities (SSRF):

- No validation of user-supplied URLs
- Common in modern features (webhooks, URL previews)
- Difficult to limit outbound traffic



Mitigation Strategies

- Isolate resource fetching
- Use allow lists for URLs
- Disable HTTP redirections
- Validate/sanitize all inputs
- Never send raw responses

Unrestricted Access to Sensitive Business Flows

Risk #6 APIs expose critical business processes

Attack Vector: Exploiting business model APIs

Impact: Business harm (e.g., inventory hoarding)

Vulnerabilities (Business Flows): Examples:-

- Product purchasing flows
- Comment/post creation
- Reservation systems

Mitigation Strategies

- Identify abusable business flows
- Implement protection mechanisms:
 - Device fingerprinting
 - CAPTCHA/biometrics
 - Least privilege principle

Broken Function Level Authorization

Risk #5 Users access privileged functions

Attack Vector: Non-privileged user accessing admin endpoints

Impact:

- Data loss/corruption
- Service disruption

Vulnerabilities (Authorization):

- Access to admin API endpoints
- Unauthorized CRUD operations

POST {{uri}}/workshop/api/shop/orders

```
{ "product_id": 1, "quantity": -3 }
```

```
{
  "id": 7,
  "message": "Order sent successfully.",
  "credit": 60.0
}
```

```
{ "product_id": 1, "quantity": -11 }
```

```
{
  "id": 9,
  "message": "Order sent successfully.",
  "credit": 200.0
}
```

Unrestricted Resource Consumption

Risk #4 - API resource starvation

Attack Vector: DDoS attacks

Impact:

- Service disruption
- Increased cloud costs

Vulnerabilities (Resource): Missing limits on:-

- Execution time
- Memory allocation
- File descriptors
- Upload sizes

Denial of Service

- **DoS:** (*Denial of Service*)
 - A **single** source (e.g., one attacker)
 - Easier to execute but simpler to block
- **DDoS:** (*Distributed Denial of Service*)
 - Attacks from **many** sources simultaneously, often via a botnet
 - Very difficult to stop and extremely damaging

Mitigation Strategies

- Set resource limits (CPU/memory)
- Implement rate limiting
- Use server-side validation
- Set cloud spending limits
- Implement caching

Broken Object Property Level Authorization

Risk #3 Unauthorized access to object properties

Attack Vector: Inspecting API responses

Impact:

- Data disclosure
- Account takeover

Vulnerabilities (Object Property):

- Excessive data exposure
- Unauthorized CRUD on sensitive properties

GET /api/users/456

Authorization: Bearer <valid_token>

GET /api/users/123

Authorization: Bearer <valid_token>

```
{
  "userId": 123,
  "name": "Alice",
  "email": "alice@example.com",
  "accountBalance": 1000 // Only admins
should see this!
}
```

```
{
  "userId": 456,
  "name": "Alice",
  "email": "alice@example.com",
  "accountBalance": 5000 // Only admins
should see this!
}
```

Mitigation Strategies

- **Implement property-level access checks**

```
# ABAC policy example: Grant read access to 'email' only if user is an admin
if user.role == "admin" and requested_property == "email":
    grant_access()
```

- **Validate user permissions**
- **Cherry-pick returned properties:** selectively extracting specific fields or attributes from an object, API response, or dataset while ignoring the rest.
- **Use Attribute-Based Access Control (ABAC):** grants or denies access to resources based on attributes of the user, resource, action, and environment.

Broken Authentication

Risk #2 Weak authentication mechanisms

Attack Vector: Brute force attacks

Impact:

- Full account compromise
- Private data access

Vulnerabilities (Authentication):

- No CAPTCHA/account lockouts
- Weak passwords/JWT tokens
- Sensitive data in URLs
- No token validation

Mitigation Strategies

- Strong password policies
- Secure password storage (Bcrypt/Argon2)
- Multi-Factor Authentication (MFA)
- Anti-brute force mechanisms

Broken Object-Level Authorization

Risk #1 (BOLA) - Access to other users' data

Attack Vector: Manipulating object IDs

`http://localhost:8888/identity/api/v2/vehicle/4093f2c0-c142-4da8-b40a-8b463110b939/location`

Impact:

- Data disclosure/manipulation

Vulnerabilities (BOLA):

- Missing object-level authorization checks
- Insecure coding practices

```
{
  "carId": "4093f2c0-c142-4da8-b40a-8b463110b939",
  "vehicleLocation": {
    "id": 5,
    "latitude": "37.406769",
    "longitude": "-94.705528"
  },
  "fullName": "Martina Kraus",
}
```

GET `http://localhost:8888/identity/api/v2/vehicle/4bae9968-ec7f-4de3-a3a0-ba1b2ab5e5e5/location`

```
{
  "carId": "4bae9968-ec7f-4de3-a3a0-ba1b2ab5e5e5",
  "vehicleLocation": {
    "id": 3,
    "latitude": "37.746880",
    "longitude": "-84.301460"
  },
  "fullName": "Robot",
}
```

Mitigation Strategies

- Implement strong authorization
- Use UUIDs instead of sequential IDs
- Adopt zero trust security
- Test authorization mechanisms

Other API Security Best Practices ...

1. Rate limiting/throttling
2. API gateways
3. Versioning

Security Design Principles (1/3)

- **Least Privilege:** An entity should only have the required set of permissions to perform the actions for which they are authorized, and no more. Permissions can be added as needed and should be revoked when no longer in use.
- **Fail-Safe Defaults:** A user's default access level to any resource in the system should be "denied" unless they've been granted a "permit" explicitly. Base access decisions on permission rather than exclusion.
- **The economy of Mechanism:** The design should be as simple as possible. All the component interfaces and the interactions between them should be simple enough to understand. Keep the design as *simple* and *small* as possible

REST Security Design Principles (2/3)

- **Complete Mediation:** A system should validate access rights to all its resources to ensure that they're allowed and should not rely on the cached permission matrix. If the access level to a given resource is being revoked, but that isn't reflected in the permission matrix, it would violate the security.
- **Open Design:** The Principle of Open Design says that "*your system security shouldn't rely on the secrecy of your implementation*". This is a particularly important principle for security concepts like cryptographic implementations. Well-designed cryptography implementations are published publicly. *The design should not be secret.*

REST Security Design Principles (3/3)

- **Separation of Privilege:** Granting permissions to an entity should not be purely based on a single condition, a combination of conditions based on the type of resource is a better idea. Such as Separating system functions like read, edit, write, execute, etc.
- **Least Common Mechanism:** It concerns the risk of sharing state among different components. If one can corrupt the shared state, it can then corrupt all the other components that depend on it.
 - we don't reuse passwords from service accounts and other subjects.
- **Psychological Acceptability:** It states that security mechanisms should not make the resource more difficult to access than if the security mechanisms were not present. In short, security should not make worse the user experience.

PCI Compliance for APIs

Core Principles:

1. Secure network/systems
2. Protect cardholder data
3. Vulnerability management
4. Strong access controls
5. Regular monitoring
6. Security policies