Authentication on the WEB

(Sessions , Cookies, JWT, Local Storage ….)

Sessions

Flow

- user submits login *credentials*, e.g. email & password
- server verifies the credentials against the DB
- server creates a temporary user **session**
- sever issues a cookie with a **session ID**
- user sends the cookie with each request
- server validates it against the session store & grants access
- when user logs out, server destroys the sess. & clears the cookie

Features

- every user session is stored server-side (**stateful**)
  - memory (e.g. file system)
  - cache (e.g. `Redis` or `Memcached`), or
  - DB (e.g. `Postgres`, `MongoDB`)
- each user is identified by a session ID
  - **opaque** ref.
    - no 3rd party can extract data out
    - only issuer (server) can map back to data

- each user is identified by a session ID
  - **opaque** ref.
    - no 3rd party can extract data out
    - only issuer (server) can map back to data
  - stored in a cookie
    - signed with a secret
    - protected with flags
- SSR web apps, frameworks (`Spring` , `Rails`), scripting langs (`PHP`)

Cookies

- `Cookie` header, just like `Authorization` or `Content-Type`
- used in session management, personalization, tracking
- consists of *name*, *value*, and (optional) *attributes / flags*
- set with `Set-Cookie` by server, appended with `Cookie` by browser

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: SESS_ID=9vKnWqiZvuvVsIV1zmzJQeYUgINqXYeS; Domain=example.com;
Path=/
```

Security

- signed (`HMAC`) with a secret to mitigate tampering
- *rarely* encrypted (`AES`) to protected from being read
  - no security concern if read by 3rd party
  - carries no meaningful data (random string)
  - even if encrypted, still a 1-1 match
- encoded (`URL`) - not for security, but compat

Attributes

- `Domain` and `` `Path` `` (can only be used on a given site & route)
- `Expiration` (can only be used until expiry)
  - when omitted, becomes a *session cookie*
  - gets deleted when browser is closed

Flags

- `HttpOnly` (cannot be read with JS on the client-side)
- `Secure` (can only sent over encrypted `HTTPS` channel), and
- `SameSite` (can only be sent from the same domain, i.e. no CORS sharing)

CSRF

- unauthorized actions on behalf of the authenticated user
- mitigated with a CSRF token (e.g. sent in a separate `X-CSRF-TOKEN` cookie)

Tokens

Flow

- user submits login *credentials*, e.g. email & password
- server verifies the credentials against the DB
- sever generates a temporary **token** and embeds user data into it
- server responds back with the token (in body or header)
- user stores the token in client storage
- user sends the token along with each request
- server verifies the token & grants access
- when user logs out, token is cleared from client storage

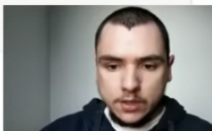
Features

- tokens are *not* stored server-side, only on the client (**stateless**)
- *signed* with a secret against tampering
  - verified and can be trusted by the server
- tokens can be *opaque* or *self-contained*
  - carries all required user data in its payload
  - reduces database lookups, but exposes data to XSS
- typically sent in `Authorization` header
- when a token is about to expire, it can be *refreshed*
  - client is issued both access & refresh tokens
- used in SPA web apps, web APIs, mobile apps

JWT (JSON Web Tokens)

- open standard for authorization & info exchange
- *compact*, *self-contained*, *URL-safe* tokens
- signed with *symmetric* (secret) or *asymmetric* (public/private) key

```
HTTP/1.1 200 OK
Content-type: application/json
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI1YmQ2MWFhMWJiNDNmNzI0M2EyOTM
xNmQiLCJuYW1lIjoiSm9obiBTbWl0aCIsImlhdCI6MTU0MTI3NjA2MH0.WDKey8WGO6LENkHWJRy
8S0QOCbdGwFFoH5XCAR49g4k
```



- contains **header** (meta), **payload** (claims), and **signature** delimited by `.`

```js
atob('eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9')
// "{"alg":"HS256","typ":"JWT"}"
//     ↑ algorithm   ↑ type

atob('eyJzdWIiOiI1YmQ2MWFhMWJiNDNmNzI0M2EyOTMxNmQiLCJuYW1lIjoiSm9obiBTbWl0aC
IsImlhdCI6MTU0MTI3NjA2MH0')
// "{"sub":"5bd61aa1bb43f7243a29316d","name":"John Smith","iat":1541276060}"
//     ↑ subject (e.g. user ID)         ↑ claim(s)          ↑ issued at (in
seconds)
```

js

Security

Security

- signed (`HMAC`) with a secret
  - guarantees that token was not tampered
  - any manipulation (e.g. exp. time) invalidates token
- *rarely* encrypted (`JWE`)
  - (web) clients need to read token payload
  - can't store the secret in client storage securely
- encoded (`Base64Url`) - not for security, but transport
  - payload can be decoded and read
  - no sensitive/private info should be stored
  - access tokens should be short-lived

XSS

- client-side script injections
- malicious code can access client storage to
  - steal user data from the token
  - initiate AJAX requests on behalf of user
- mitigated by sanitizing & escaping user input

Client Storage

- JWT can be stored in client storage, `localStorage` or `sessionStorage`
  - `localStorage` has no expiration time
  - `sessionStorage` gets cleared when page is closed

`localStorage`

Browser key-value store with a simple JS API

Pros

- domain-specific, each site has its own, other sites can't read/write
- max size higher than cookie (`5 MB` / domain vs. `4 KB` / cookie)

Cons

- plaintext, hence not secure by design
- limited to string data, hence need to serialize
- can't be used by web workers
- stored permanently, unless removed explicitly
- accessible to any JS code running on the page (incl. XSS)
  - scripts can steal tokens or impersonate users

Best for

- public, non-sensitive, string data

Worst for

- private sensitive data
- non-string data
- offline capabilities

Sessions vs. JWT

Sessions + Cookies

Pros

- session IDs are opaque and carry no meaningful data
- cookies can be secured with flags (same origin, HTTP-only, HTTPS, etc.)
- HTTP-only cookies can't be compromised with XSS exploits
- battle-tested 20+ years in many langs & frameworks

Cons

- server must store each user session in memory
- session auth must be secured against CSRF
- horizontal scaling is more challenging
  - risk of single point of failure
  - need sticky sessions with load balancing

JWT Auth

Pros

- server does not need to keep track of user sessions
- horizontal scaling is easier (any server can verify the token)
- CORS is not an issue if `Authorization` header is used instead of `Cookie`
- FE and BE architecture is decoupled, can be used with mobile apps
- operational even if cookies are disabled

Cons

- server still has to maintain a blacklist of revoked tokens
  - defeats the purpose of stateless tokens
  - a whitelist of active user sessions is more secure
- when scaling, the secret must be shared between servers
- data stored in token is "cached" and can go *stale* (out of sync)
- tokens stored in client storage are vulnerable to XSS
  - if JWT token is compromised, attacker can
    - steal user info, permissions, metadata, etc.
    - access website resources on user's behalf
- requires JavaScript to be enabled

Options for Auth in SPAs / APIs

1. Sessions
2. Stateless JWT
3. Stateful JWT

Stateless JWT

- user payload embedded in the token
- token is signed & `base64url` encoded
  - sent via `Authorization` header
  - stored in `localStorage` / `sessionStorage` (in plaintext)
- server retrieves user info from the token
  - no user sessions are stored server side
  - only revoked tokens are persisted
  - refresh token sent to renew the access token

Stateful JWT

- only user ref (e.g. ID) embedded in the token
- token is signed & `base64url` encoded
  - sent as an HTTP-only cookie (`Set-Cookie` header)
  - sent along with non-HTTP `X-CSRF-TOKEN` cookie
- server uses ref. (ID) in the token to retrieve user from the DB
- no user sessions stored on the server either
- revoked tokens still have to be persisted

Sessions

- sessions are persisted server-side and linked by sess. ID
- session ID is signed and stored in a cookie
  - sent via `Set-Cookie` header
  - `HttpOnly`, `Secure`, & `SameSite` flags
  - scoped to the origin with `Domain` & `Path` attrs
- another cookie can hold CSRF token

Verdict

Sessions are (probably) better suited for web apps and websites.

Why not JWT?

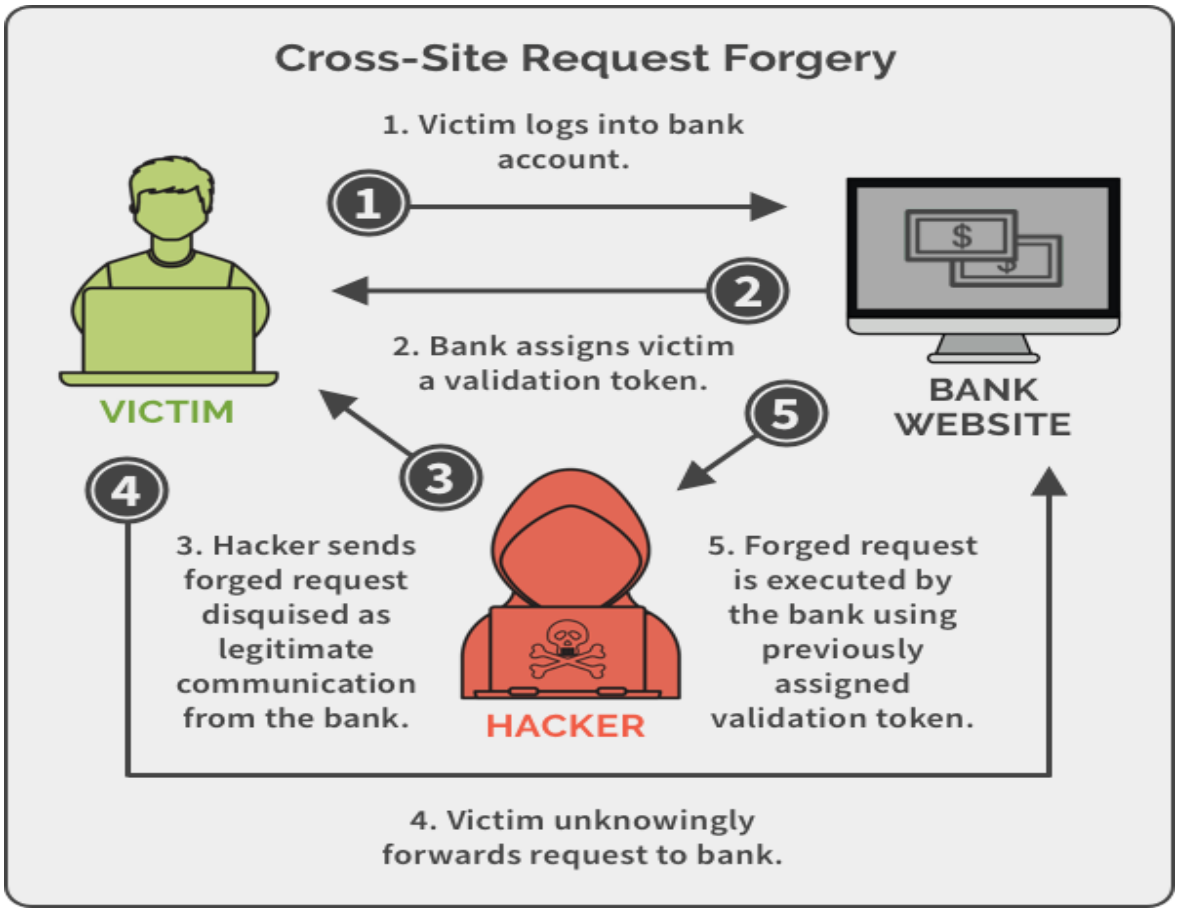
- server state needs to be maintained either way
- sessions are easily extended or invalidated
- data is secured server side & doesn't leak through XSS
- CSRF is easier to mitigate than XSS (still a concern)
- data never goes stale (always in sync with DB)
- sessions are generally easier to set up & manage
- most apps/sites don't require enterprise scaling

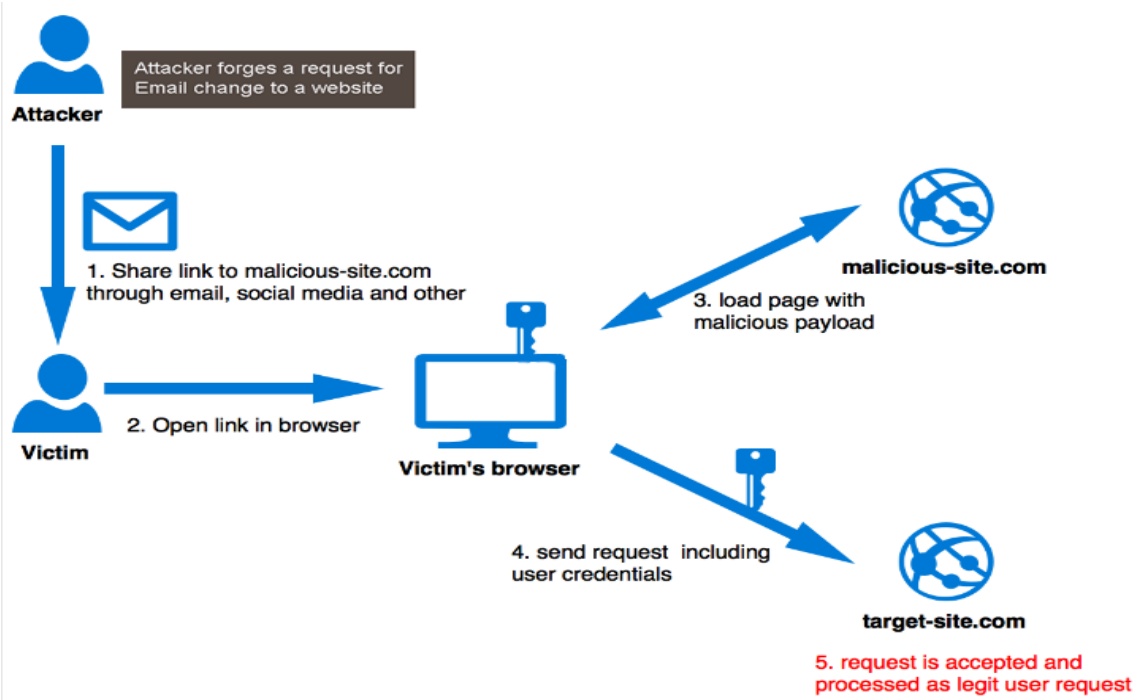
Important

Regardless of auth mechanism

- XSS can compromise user accounts
  - by leaking tokens from `localStorage`
  - via AJAX requests with user token in `Authorization`
  - via AJAX requests with `HttpOnly` cookies
- SSL/HTTPS must be configured
- security headers must be set

CSRF





Auxiliary measures

- IP verification
- user agent verification
- two-factor auth
- API throttling