A2 : Authentication & Authorization

Authentication (User Authe.)

Note :- InfoSec : CIA -> Parkerian Hexad (Google it)

Authe. Factors : KOI

- **K**nowledge factors : password, PIN, security que. etc.
- **O**wnership factors : wrist band, ID card, HW/SW token etc.
- **I**nherence factors : fingerprint, retina, DNA, sign., face, voice etc,

Types of Authe. : (will discuss bold one's)

- **Password based** - simple cookie based hashing user/pass etc.
- Passwordless - nothing but a hype term.
- MFA (Multifactor auth)
- Biometric, Captchas
- Certificate Based (Digital Sign.) - uses SSO
- Single Sign-On (SSO) - based on cookies
- **Token Based :** HTTP Basic, API Key, OAuth/OIDC (JWT)

Types of Authe. Protocol : Based on stateful & stateless authe. (will discuss bold ones)

- **HTTP Autho. Header (Basic Schema)** (Other Schema - Bearer, Digest, HMAC, etc.) - require no cookies, SID, login page etc.
- **OpenID/OIDC** (on top of OAuth 2.0) - will discuss in Authorization
- **SAML** (Closely used as alternate to OIDC)
- **JWT** : JWS/JWE/JWK/JWA
- Kerberos
- LDAP
- PAP
- CHAP
- EAP
- P2P
- FIDO2 - UAF, U2F
- SSL/TLS - Uses public key cryptography - digital certificates
- AAA architecture protocols :
  - TACACS, XTACACS and TACACS+
  - RADIUS (LoginRadius)
  - DIAMETER
- Others : Hawk authe, AWS Sign., NTLM, Etc... (See wiki for more)

User Authentication choice/ways :

NOTE on Session Management :

- **Client Side** - Session info. is stored in cookie (user data are openly exposed), URL querystring, hidden form field, View state
- **Server side** - Server sets SID in cookie(Only SID is exposed, rest data stored in db - widely used), App state, Session state.

- **Old/Obslete Methods (Rarely used) :**
  - **IP address** : The server knows the IP address of the computer running the browser and could theoretically link a user's session to this IP address. However, IP addresses are generally not a reliable way to track a session or identify a user.
  - URL query string : A querystring is information added to the end of a page's URL using "?" as the separator between the URL and the querystring and "&" to separate each querystring pair. Eg: http://www.examplesite.com?sid=Hu33Ymd923Js1ULsd45
    - Pros :
      - Lightweight and easy to employ.
      - Useful when client browser cookie function is disabled.
      - Transferable through a copy of the URL.
    - Cons :
      - URL length has limitations: usually 255-character
      - Visible and easy to tampered.
      - If a user visits a page by coming from a page internal to the site the first time, and then visits the same page by coming from an external search engine the second time, the query strings would likely be different. If cookies were used in this situation, the cookies would be the same.
  - **Hidden form fields** : A hidden form field is not visible in the browser, but the value can be sent to the server along with the other form fields. Example: <input type=hidden name=sid value=Hu33Ymd923Js1ULsd45>
    - GET - Similar to URL query strings (add form field to url).
    - POST - form information & hidden fields sent in HTTP request body (Neither url nor cookie)
    - Pros :
      - Not as visible as the querystring.
      - session info is not copied when the user copies the URL (to bookmark the page or send it via email, for example)
    - Cons :

        - docs need to embedd data inside, waste bandwidth. have to embed the result from the previous page on the next page.
        - Everyone can see the embedded data by viewing the original source code.
        - We cannot store all kinds of objects in hidden boxes except text/string values.
        - Hidden boxes travel over the network along with the request and response, indicates more network traffic.
    - **"window.name" DOM property** : All current web browsers can store a fairly large amount of data (2–32 MB) via JavaScript using the DOM property window.name. This data can be used instead of session cookies and is also cross-domain. The technique can be coupled with JSON/JavaScript objects to store complex sets of session variables on the client side.The downside is that every separate window or tab will initially have an empty window.name property when opened. Furthermore, the property can be used for tracking visitors across different websites, making it of concern for Internet privacy. In some respects, this can be more secure than cookies due to the fact that its contents are not automatically sent to the server on every request like cookies are, so it is not vulnerable to network cookie sniffing attacks. However, if special measures are not taken to protect the data, it is vulnerable to other attacks because the data is available across different websites opened in the same window or tab.
    - **HTTP Authentication & Authorization Framework :**
        - **Basic Schema** - Oldest method, only to be used with https, not used widely. No login page required, username/password will be by default prompted to User by browser. It transmits credentials as user ID/password pairs, encoded using base64.
            - The **server** responds to client with `401` (Unauthorized) response status & provides info. on how to authorize with `www-Authenticate` **response header** containing at least one challenge. Eg. **www-Authenticate: <Basic> realm=<access to site>**
            - A **client** that wants to authenticate itself with the server can then do so by including an `Authorization` **request header** with the credentials - Key/value, JWTokens Etc. Eg. **Authorization: <Basic> <Hu33Ymd923Js1ULsd45>**
            - If All goes well, page/response is displayed. Session established.
        - Other Schemas - Bearer, Digest, HOBA, AWS3, Etc.
        - **Pros** : Implementation is simplest bcz it does not require cookies, SID/login pages, rather uses standard fields in HTTP header (Server : www-authenticate & Client : authorization).
        - **Cons :** Basic Schema has no way to logout/revoke user & is unsafe (without https).

- **Traditional Method (used presently) : Stateful session management**
  - **Cookie based authentication (server side)** - SID stored into cookies (1st party session cookies) & sent back to server on each http req. header automatically.
    - **Note** : **See Cookie topic for details (HTTP + Cookies = Stateful)**
    - Lib. : Passport-local strategy (uses express.session internally)
    - State Store : DB, Redis Cache DB, Filesystem etc.
    - Web FW : ASP.NET, django, nodejs etc. support via session management API for session based authentication.
- **Modern Method : Stateless session management (JWT) & Hybrid Approach**
  - **Glossary** : RT - refresh token | RTR - refresh token rotation | OT - Opaque token | AT - access Token | IT - identity token
  - **NOTE on modern web apps (Evolution) :** The key thing to remember is that the data, which software application is processing, is known as The Model or The Application State. Few brave souls might call it Domain Model/Business Logic of App. An App could be desktop or web app. Struggle is to understand reasonable ways of presenting this application state to user (of web front-ends).
    - Desktop MVC -> AM-MVC -> MVP -> MVVM/MVB(Base of react/angular/vue) -> Web MVC(sub-category of distbd app) -> Server Side MVC(Model 2/SSR - ASP.NET,Struts,Django,ROR,CodeIgniter), browser's internal MVC, front-end MVC(CSR) -> Microservices/Backend/Fullstack.
    - JS tech. wise : jQuery -> Backbone(MV*) -> Knockout.js (MVVM) -> Angular 1/Emberjs/Auerlia (MVVM) -> Angular 2+/React/vue/CSR->SSR (Component->class->functional/MVVM/MV*) -> svelte/Compiler/SSG/CSM/pre-rendering -> Monolith/stimulusjs/strudlejs -> Microfrontends+microservices/backend/fullstack
  - **NOTE - WHY Tokens ? :** Web applications have evolved from simple static websites (two-tiered architecture) into complex multi-layered SPA and SSR driven API first systems. CMS systems have grown into Headless content-first systems. Modern web applications are all about decoupled UI and API. Front-end community has changed rapidly in recent times. It started with DOM infused algorithms introduced by jQuery, which was quickly succeeded by MVC based Backbone.js. And, in no time, we found ourselves in the jungle of bidirectional and unidirectional data flow architecture. Most web apps are turning into SPA — Single Page Applications. Add to that is the mix of Server-side rendering, pre-rendering, etc. Core function of cookie was to preserve identity of the user between multiple page requests. With SPA, it is absolutely not required to maintain cookie to identify user. Once the application is loaded, user doesn't need to refresh the page. HTML5 provides us with new Storage API — LocalStorage and SessionStorage. All that a UI needs is a token to make API calls. They may be same domain, sub-domain or on a different domain (CORS). With fetch API, it becomes very simple to make CORS API calls. REST API have evolved into stateless API. These APIs do not need session like the way cookie provides. All it needs is valid auth token and API returns appropriate response.

- **Token Based Authentication (General/Stateless/cookie-less)** :
  - Library : jwtokens (jwt.io), passport-jwt strategy
  - JWT are stateless bcz server doesn't need to maintain state as JWT uses RSA. Token itself is all that is needed to verify a token bearer's/Client authorization. Hence no need to store user info at server side. Authorization is also based on token which we will discuss in OAuth/OIDC.
  - **JWT Token Storage (Client-side) :** Tokens sent to server via HTTP Authorization Request Header for every subsequent request.
    - **Cookies** (Using JWT as sessions - **not recommended**) [discussed in stateful JWT]
    - **SessionStorage, LocalStorage** (mostly LS is used, although both SS/LS are **not recommended** due to security)
    - **RT is recommended over cookie, SS, LS.**
  - **JWT Problem (Token revocation) :** Since JWT is stateless & comes with their own expiry, there is no way to revoke (Logout) the user's session once the server signs a valid token. i.e Server can't control tokens after sending.
    - **Solution :**
      - Maintaing token revocation/blacklist **(recommended here as temp. solution - better approach is JWT+RT/RTR)**
      - Change Private Key/Digital Sign. (**not recommended** as it will invalidate across all devices)
      - using jwt with http cookies (**not recommended** - discussed in stateful JWT)
      - **Recommended** to use JWT with RT to keep user logged in (OT is session cookie) [See - Stateless JWT with RT]
        - Short-lived AT (JWT, OT) + long-lived RT (OT)
        - Short-lived AT (JWT, OT) + short-lived RTR (OT)
  - **Algorithm (JWT - Stateless - General) :**
    - User navigates to web app at https://ui.example.com
    - Initial API call is made. If it returns status as 401 means user doesn’t have token and a login view is shown.
    - User enter creds. Server verifies by hashing password match in DB.
    - After verification, server send back token by adding into HTTP Bearer Authentication response header (could also send tokens via - URL query string/post req body/etc.) with expiry which is stored client side - Session/localStorage. If Client is :
      - **REST-API/curl/postman** : token should be send/stored via the HTTP Auth header makes sense. (**recommended**)
      - **Browser** : token (store : local/session) & send token via HTTP Auth header (**both below approach not recommended**)

        - If token need for **tabs/window** - **sessionstorage**, if req. to persist across **multiple tabs/window** then **localstorage**
        - SPA : token (Store : memory/server side/DB) i.e. stateful JWT approach with cookies.
    - Unlike Cookies, for every subsequent req. token needs to be passed manually via authorization header (alongwith any API public key if any). Server match public_api_key/token again private key/DB. If token has expired new token is attached.
    - For logout/revoke after expiry token is deleted from server. **token revocation problem is solved by maintaing** `revoked token blacklist`**.** user is shown login view again.
- **Stateful JWT (JWT + session cookie)** - Using JWT's as Session **(Not recommended)**
    - **NOTE** : we get statelessness benefit of jwt (scalibility) & token revocation problems is solved either with token revocation list or using cookies. Both comes with its pros/cons. A thrid approach (stateless JWT + RTR/RT) is recommended. However, we'll discuss the cookie approach here for the sake of knowing.
    - **Token Storage (Cookies)** : Use cookies with (Httonly, secure, samesite) flags. Following are the **Drawbacks** :
        - JWT token size should not exceed than that of cookie's allowed size.
        - samesite = strict (Protects from CSRF, but invalidate the whole point of using tokens as it can't be used for cross origin eg. OAuth). Temporary workaround is samesite=lax.
        - samesite = lax (Allows - GET, HEAD, OPTION, TRACE | Disallow - POST) : Could use Authorization Bearer Header though.
    - **Algorithm (Stateful JWT : JWT + session cookie) :**
        - First time, User enter creds, Server verifies it, create a token/SID/session_cookie & maps it to user DB ID and store in redis cache db.
        - Server sends token/SID/session_cookie (http-only) as response to browser.
        - Now for every subsequent req. token gets send to server, which it verifies and allows access if valid. Token will be valid until expiry, however server can revoke/logout on demand (delete the SID/session_cookie from server side, hence invalidating). Also it could update the token revocation blacklist etc.
        - **NOTE** : **This approach is similar to JWT stateless, however only difference is use of cookies to make it stateful. Still not recommended, better use RT approach.**

- **Stateless JWT with RT** : **Recommended** & Widely used
  - 3rd party lib - AWS cognito, Firebase etc.
  - **NOTE** : OT is like session cookie (Httponly, secure, samesite). Remeber there are only two type of tokens : JWT & OT. Refresh token and access token use either or any of them based upon which protocol is used (OAuth/OIDC).
  - **Algorithm** (Used in OAuth2.0 -> OIDC) : mentioned in detail in authorization section.
    - **Short-lived AT (JWT, OT) + long-lived RT (OT)**
    - **Short-lived AT (JWT, OT) + short-lived RT/RTR (OT)**

Authorization (User Autho./Access Control)

NOTE : There is no specific type/protocol for authorization. Some are HTTP Autho. & OAuth 2.0/OIDC. We will discuss here only **OAuth 2.0/OIDC**. To delegate Autho. use - Authorization as a Service (Eg. Cerbos)

Access Control System (Autho.) Factors :

- AAA : Authentication, Authorization, Accountability/Auditing
  - Identity & Access Mgmt. (IAM)
- Other keywords :
  - Access control List, Access control matrix
  - Authorization OSID (Similar to Authentication OSID)

Authorization/Access Control Model :

NOTE : As per OWASP, prefer ABAC & ReBAC over RBAC.

- **RBAC** (Role-Based-Access-Control) : RBAC is a model of access control in which access is granted or denied based upon the roles assigned to a user.A role is nothing but a collection of permissions. Permissions are not directly assigned to an entity; rather, permissions are associated with a role and the entity inherits the permissions of any roles assigned to it. Generally, the relationship between roles and users can be many-to-many, and roles may be hierarchical in nature. RBAC treats authorization as permissions associated with roles and not directly with users attribute. For example, imagine that you work as a department manager in an organization. In this situation, you should have permissions that reflect your role, for example, the ability to approve vacation requests and expense requests, assign tasks, and so on. To grant these permissions, a system manager would first create a role called "Manager" (or similar). Then, they would assign these permissions to this role and would associate you with the "Manager" role. Of course, other users that need the same set of permissions can be associated with that role.
  - **Pros** : Managing authorization privileges becomes easier because sysadmin can deal with users and permissions in bulk instead of having to deal with them one by one. consider RBAC if :
    - You’re in a small- to medium-sized enterprise
    - Your access control policies are broad
    - You have few external users, and your organization roles are clearly defined.
  - **Cons** : RBAC simplicity is also a big disadvantage in systems as they scale. You can’t assign granular permissions like project and team-level administrative access without

expanding your roles table. This makes maintaining RBAC a headache in a SaaS application with lots of users and permission levels.

- **ABAC** (Attribute-Based-Access-Control) : In ABAC "subject requests to perform operations on objects are granted or denied based on assigned attributes of the subject, assigned attributes of the object, environment conditions, and a set of policies that are specified in terms of those attributes and conditions". Attributes are simply characteristics that be represented as name-value pairs and assigned to a subject, object, or the environment. When using ABAC, a computer system defines whether a user has sufficient access privileges to execute an action based on a trait (attribute or claim) associated with that user.
  - Eg. online store that sells alcoholic beverages provided proof of their age : Alcoholic beverage = resource | Online store = resource owner | Age of consumer validated during registration process is claim, that is proof of user's age attribute. Presenting the age claim allows the store to process access requests to buy alcohol. So, in this case, the decision to grant access to the resource is made upon the user attribute.
  - **Pros** : Flexibility - ABAC allows admins to implement granular, policy-based access control, using different combinations of attributes to create conditions of access that are as specific or broad as the situation calls for. In short, choose ABAC if:
    - You're in a large organization with many users
    - You want deep, specific access control capabilities
    - You have time to invest in a model that goes the distance
    - You need to ensure privacy and security compliance
  - **Cons** : Implementation complexity - Admins need to manually define attributes, assign them to every component, and create a central policy engine that determines what attributes are allowed to do, based on various conditions ("if X, then Y").
- **ReBAC** (Relationship Based Access Control) : "Does this user have a sufficient relationship to this object or action such that they can access it?" ReBAC grants access based on the relationships between resources. For instance, allowing only the user who created a post to edit it. This is especially necessary in social network applications, like Twitter or Facebook, where users want to limit access to their data (tweets or posts) to people they choose (friends, family, followers).
  - **Pros** : Supports Multi-Tenancy and Cross-Organizational Requests, Ease of Management. Support fine-grained, complex Boolean logic, Robustness, Speed.
- Others :
  - Graph-Based Access Control (GBAC)
  - Discretionary Access Control (DAC)
  - History-Based Access Control (HBAC)
  - History-of-Presence Based Access Control (HPBAC)
  - Identity-Based Access Control (IBAC)
  - Lattice-Based Access Control (LBAC)
  - Mandatory Access Control (MAC)

- Organization-Based Access Control (OrBAC)
- Rule-Based Access Control (RAC)/Rule-Based Role-Based Access Control (RB-RBAC)
- Responsibility Based Access Control (RespBAC)

User Authorization Choice [OAuth/OAuth 2.0 -> OpenID/OIDC]

NOTE : Library - passport.js OAuth strategy | Identity providers - Auth0, okta, etc.
Keywords :

- Permissions - declaration of action that can be executed on resource.
- Privileges - privileges are assigned permissions to user.
- Scopes - Scopes enable a mechanism to define what an application can do on behalf of the user.
- Attributes - characteristic of users.
- ACL - access control list. kind of matrix of what scope is given to service.
- Groups - collection of user.
- Role - Eg. group 1 has role of manager. so all user in that group are managers.
- Consent - Consent is the act of letting the user participate in deciding what data to share with a third party.
- Claims - consent screen presented to the user contains all the Claim Names with corresponding descriptions for each claim.

- **OAuth/OAuth 2.0 :** Authorization Protocol (Psuedo Authentication)
  Definition : Facilitates the authorization of one site to access and use information related to the user's account on another site. Although OAuth is not an authentication protocol, it can be used as part of one. Eg. Google, FB logins. OAuth 2.0 is used to grant authorization. An example is a to-do application, that lets you log in using your Google account, and can push your to-do items as calendar entries, at your Google Calendar. The part where you authenticate your identity is implemented via OpenID Connect, while the part where you authorize the to-do application to modify your calendar by adding entries, is implemented via OAuth 2.0. It enables you to authorize the Web App A to access your information from Web App B, without sharing your credentials. It was built with only authorization in mind and doesn't include any authentication mechanisms (in other words, it doesn't give the Authorization Server any way of verifying who the user is), but people started using it as authentication protcol hence it was called pseudo authentication and thus OpendID connect was created for authentication.
  Generally, OAuth provides clients a "secure delegated access" to server resources on behalf of a resource owner. It specifies a process for resource owners to authorize third-party access to their server resources without providing credentials. Designed specifically to work with Hypertext Transfer Protocol (HTTP), OAuth essentially allows access tokens to be issued to third-party clients by an authorization server, with the approval of the resource owner. The third party then uses the access token to access the protected resources hosted by the resource server. In particular, OAuth 2.0 provides specific authorization flows for web applications, desktop applications, mobile phones, and smart devices.

- **Tokens :**
  - Part 1
  - Part 2
  - **Access Tokens (OT - Session_Cookies) - Authorization** - Could be used to send API Calls for Resource Access.
    - Access Tokens should be treated as opaque strings by clients. They are only meant for the API. Your client should not attempt to decode them or depend on a particular access_token format. An access_token cannot be used for authentication. It holds no information about the user. It cannot tell us if the user has authenticated and when.
  - **Refresh Token (OT) - Authentication** -> Access + ID Token (Use to gen. new access, id token once expired)
- OAuth2 grant types/flows - To request & get an access token
  - Authorization Code
  - PKCE
  - Client Credentials
  - Device Code
  - Refresh Token

- **OpenID/OIDC & SAML :** Identity Authentication Protocol build on top of OAuth [Used for SSO - IPC]
  NOTE : 2 main protocols for authentication over networks - OIDC (OpenID Connect) and SAML (Security Assertion Markup Language)
  OIDC Definitions : An authentication method for the web (Token based - JWT). OpenID is a way to use a single set of user credentials to access multiple sites. OAuth was originally dreamt up as a solution to something known as “delegated access,” or allowing applications to talk to one another and share information on the user’s behalf. OIDC adds the authentication feature to OAuth’s powerful authorization utility.
  Tokens (OIDC) : ID Token (JWT) - Authentication
  - An id_token cannot be used for API access. Each token contains information on the intended audience (recipient). According to the OpenID Connect specification, the audience (claim aud) of each id_token must be the client_id of the client making the authentication request. If it isn't you shouldn't trust the token. An API, on the other hand, expects a token with the audience set to the API's unique identifier. So unless you are in control of both the client and the API, sending an id_token to an API will not work. Furthermore, the id_token is signed with a secret that is known to the client (since it is issued to a particular client). This means that if an API were to accept such token, it would have no way of knowing if the client has modified the token (to add more scopes) and then signed it again.

- In the example we used earlier, when you authenticate using Google, an id_token is sent from Google to the to-do application, that says who you are. The to-do application can parse the token's contents and use this information, like your name and your profile picture, to customize the user experience.The access_token can be any type of token (not necessarily a JWT) and is meant for the API. Its purpose is to inform the API that the bearer of this token has been authorized to access the API and perform specific actions (as specified by the scope that has been granted). In the example we used earlier, after you authenticate, and provide your consent that the to-do application can have read/write access to your calendar, an access_token is sent from Google to the to-do application. Each time the to-do application wants to access your Google Calendar it will make a request to the Google Calendar API, using this access_token in an HTTP Authorization header.

SAML - Authentication protcol just like OIDC :

- **Federated Identity :** We are searching for a concept called "federated identity." This is the ability to link a user's digital identity across separate security domains. In other words, when two applications are "federated," a user can use one application by authenticating their identity with the other, without needing to create separate usernames/passwords for both. The application responsible for performing this authentication is known as the identity provider (IDP). If you work for enterprise companies, you are likely familiar with IDPs like Okta, Google OAuth, Microsoft Active Directory, or Auth0. Common OSS alternatives include Keycloak and Gluu. When you sign in with your Google account (IDP), Firebase does not create a new account using the traditional username/password authentication pattern. In fact, if you create an account through Google and separately attempt to log in using that very same Gmail address, Firebase will not be able to locate your account. This is because Firebase and Google are federated - to gain access to Firebase, you have to authenticate your identity through Google, not directly with Firebase. To discuss federated identity without mentioning SAML would be disingenuous. Recall that federated identity is the ability to use the same digital identity across multiple domains, meaning we are primarily concerned with the act of authentication, which involves proving who you are.
- **OIDC Vs. SAML :**
  - OIDC is an authentication layer built on top of an authorization protocol, whereas SAML is its own self-contained authentication and authorization protocol.
  - SAML uses XML-based federation while OIDC uses JSON/REST.
  - SAML requires individual actors within the SAML architecture to be pre-configured. By keeping a tight leash on who is part of the system, SAML identity providers can tightly control who accesses data. OIDC work with any identity provider that supports the protocol. This form of authentication is more popular with consumer and native mobile applications, like gaming or productivity apps. Exporting identity management to companies like Google, Amazon, and Microsoft, these app developers can significantly reduce the friction of user sign-up. It is important to note that OIDC is extensible and can be configured to meet the security demands that enterprises require.

- **Identity Providers :** Inhouse (Google, MS etc), SaaS (Auth0, Okta), Others/Adhoc (Custom 3rd party login pages)

Conclusion : Authentication & Authorization

- Authentication : Excluding obsolete methods
  - Cookie based/Session based
  - Token based (JWT)
    - Normal Stateless JWT (Local/SessionStorage)
    - Stateful JWT (JWT with Cookies)
    - Stateless JWT + Refresh Token [Mostly recommended] => Used in OAuth2.0/OIDC
