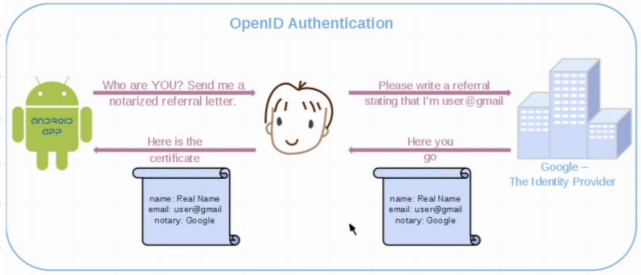
[AUTHENTICATION & AUTHORIZATION]

Authentication
verifying that someone is who they claim to be

OpenID
open std for authentication by OpenID Foundation
entities
- user
- ID provider
- web\mobile app (OpenID consumer)

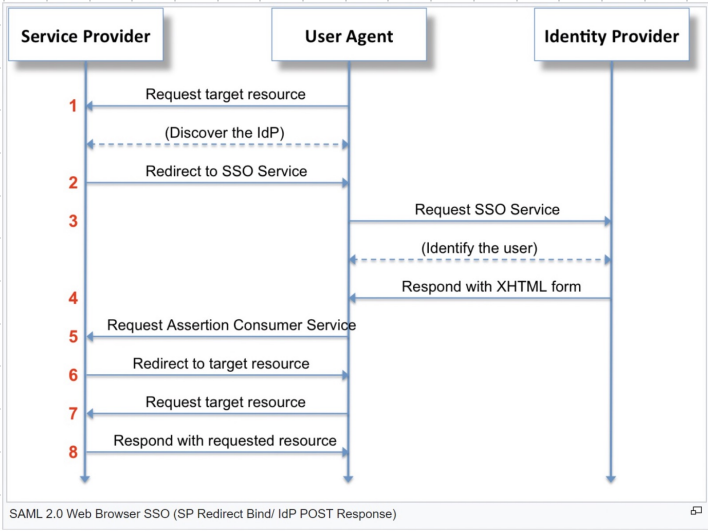


Authorization
deciding which resource a certain user should be able to access & what they should be allowed to do w those resources

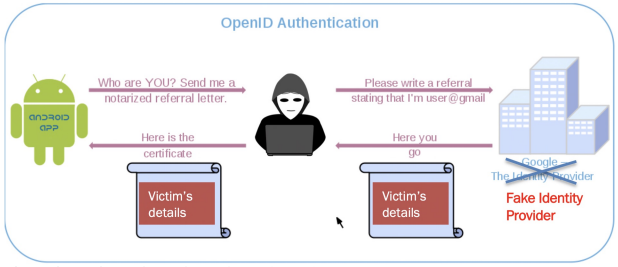
OAuth
open std. for access delegation
* way for Internet users to grant websites\apps access to their info on other websites w/out using pswds
* "secure delegated access" to server resources on behalf of resource owner
* allows issuing access tokens to 3rd party clients by authorization server w resource owner's approval
  - 3rd party uses access token to access protected resources hosted by resource server

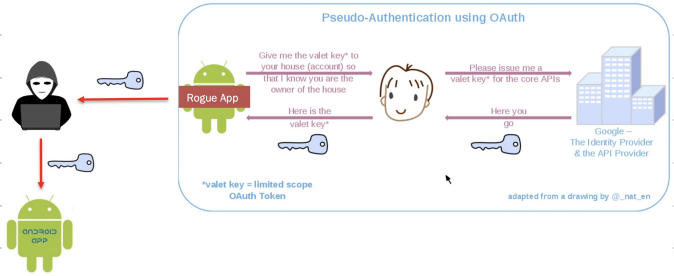
Single Sign On
allowing user to enter 1 uname & pswd to access multiple apps

Security Assertion Markup Language 2.0 (SAML 2.0)
version of SAML std for exchanging authentication & authorization identities btwn security domains
* xML-based protocol
* entities
  - user
  - ID Provider
  - service Provider

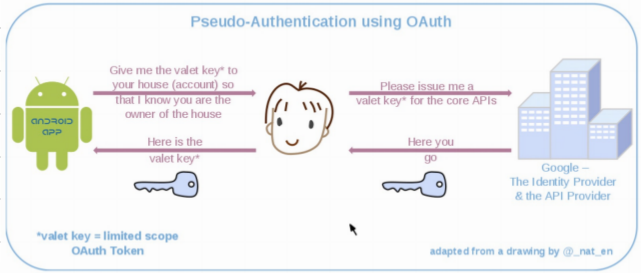


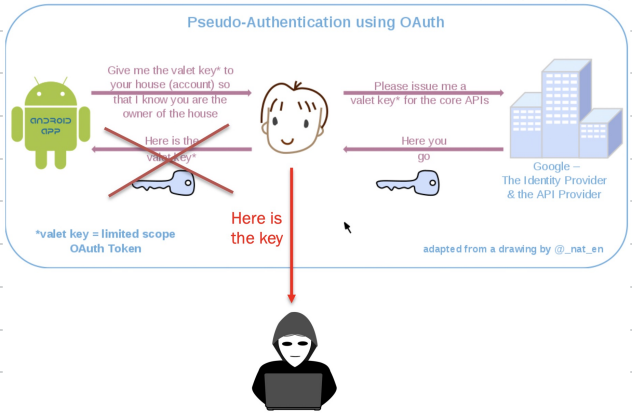
Security Problems
- msg authenticity & integrity
  * secret known → can forge msgs
- authenticity of identity provider
  * rogue identity provider can authenticate attcer as victim
- authenticity of app
  * rogue app could get access token for identity provider to authenticate to legitimate app
- covert redirect
  * redirection to attcer's website\vuln. page after authentication






```
+--------+                                  +----------------+
|        |--(A)- Authorization Request ->|  Resource      |
|        |                                  |  Owner         |
|        |<-(B)-- Authorization Grant ---|                |
|        |                                  +----------------+
|        |                                  +----------------+
|        |--(C)-- Authorization Grant -->|  Authorization |
| Client |                                  |  Server        |
|        |<-(D)----- Access Token -------|                |
|        |                                  +----------------+
|        |                                  +----------------+
|        |--(E)----- Access Token ------>|  Resource      |
|        |                                  |  Server        |
|        |<-(F)--- Protected Resource ---|                |
+--------+                                  +----------------+
```





Session-based (Stateful) Authentication

* server issues cookie containing session ID for user after login
* session value is transmitted in request's "cookie" header to server

Example:

HTTP Response	HTTP Request
Set-Cookie: PHPSESSID=298zf09hf012fh2; Secure; HttpOnly	Cookie: PHPSESSID=298zf09hf012fh2;

Cookies

* small piece of data sent from server to client browser
* can use to tell if 2 requests come from same browser
* purposes → session mgmt (ex., login, shopping carts)
  - ↳ personalization
  - ↳ tracking (recording & analyzing user behavior)

Session Management

Some Cookie Attributes

- ↳ "Secure"
  instructs web browsers to only send cookie thru encrypted HTTPS (SSL/TLS) connection } Set-Cookie: Secure
- ↳ "HttpOnly"
  prevent access to cookie vals. via JS } Set-Cookie: HttpOnly

Session Problems

* predictable sessionID
  - ↳ weak encryption
    ex., session_id(base64_encode($username));
  - ↳ insecure random
    ex., session_id(rand(time()));
  - ↳ weak secret
    ex., app->config['SECRET_KEY'] = 'changeme'
* insecure mgmt (ex., ∅ timeout)

→ detect multi-session cookies w/ CookieMonster tool

Token-based (Stateless) Authentication

* many apps use JSON Web Tokens (JWT)
  - ↳ common in REST APIs
* server creates JWT w/ secret → send JWT to client
  - → client stores JWT locally (HTML local/session storage)
* client incls. JWT in every request (normally w/ authentication header; can also be request header)
* popular for reasons:
  - ↳ scalability & performance + (∵ ∅ need to store session state on server)
  - ↳ can decouple authentication from app;
    separate servers can reuse same token for authenticating user
  - ↳ token can be cryptographically signed

JSON Web Tokens (JWT)

* open std. for creating access tokens (RFC 7519)
* base64-encoded string w/ 3 parts, separated by dots

- Header → token type ("typ")
  hashing algo ("alg") – signed using → secret (ie., HMAC algo) / public/private key pair (RSA/ECDSA)
- Payloads – contain "claims" (ie., stmts abt user)
  * client can use token to prove user's capabilities (ex., "sub", "name")
- Signature – created by applying algo specified in JWT header to encoded header & payload
  * can verify JWT sender if using asymmetric encryption

* generated & signed on server side
* ∅ put secret in payload or header unless encrypted!
* token valid forever unless signing key changes
  - ↳ best practice: define expiry attr. for validity duration (ex., 10-20 mins)

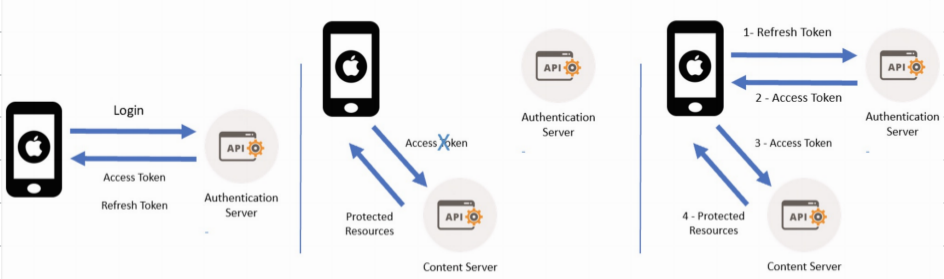
	OAuth2	OpenId	SAML
Token (or assertion) format	JSON or SAML2	JSON	XML
Authorization?	Yes	No	Yes
Authentication?	Pseudo-authentication	Yes	Yes
Year created	2005	2006	2001
Current version	OAuth2	OpenID Connect	SAML 2.0
Transport	HTTP	HTTP GET and HTTP POST	HTTP Redirect (GET) binding, SAML SOAP binding, HTTP POST binding, and others
Security Risks	Phishing OAuth 2.0 does not support signature, encryption, channel binding, or client verification. Instead, it relies completely on TLS for confidentiality.	Phishing Identity providers have a log of OpenID logins, making a compromised account a bigger privacy breach	XML Signature Wrapping to impersonate any user
Best suited for	API authorization	Single sign-on for consumer apps	Single sign-on for enterprise Note: not well suited for mobile

Two-factor Authentication (2FA)

Authentication that requires 2 distinct forms of identification:
- ↳ what you know (ex., pswd, email)
- ↳ what you have (ex., mobile device, physical key, cert)
- ↳ what you are (biometric attributes)

Access & Refresh Token

* access token (short-lived; minutes)
  ↳ carry necessary info to access resource directly
* refresh token (long-lived; weeks\months)
  ↳ carry necessary info to get new access token



Problems w Tokens

* JWT supports "None" algo for signatures
  ↳ server assumes token already validated
  ⇒ attcr can change algo to "None" & tamper payload
  ⇒ effectively ✗ integrity!

* RS256\HS256 confusion Atk
  ↳ HMAC (HS256) — symmetric encryption
  RSA (RS256) — asymmetric encryption
  ↳ attcr changes RS256 → HS256 signature
  → public key used as expected "symmetric key"
  ⇒ attcr can forge msgs

* using weak secrets
  ↳ secret for HS256 signature can be brute-forced offline
  ⇒ attcr can use obtained secret to forge new tokens
* token info. disclosure: devs still keep sensitive info in JWT
  ⇒ attcr base64 decode JWT to gain access to sensitive info
* disclosing secret on server used to sign tokens

Access Control

determines whether user allowed to carry out attempted action

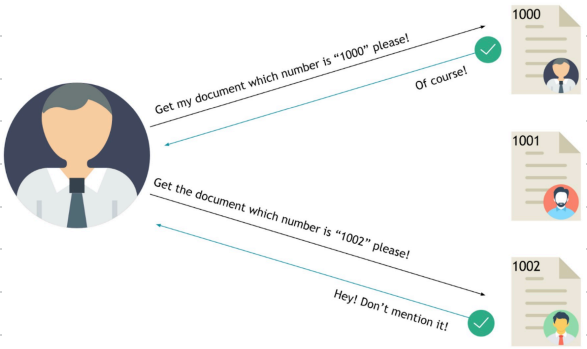
* broken access ctrl (OWASP Top Ten: A01: 2021)
  ↳ commonly encountered & often critical security vuln.
* types:
  ↳ vertical access ctrl
  mechanisms that restrict access to sensitive fnality ✗ available to other types of users
  ↳ horizontal access ctrl
  mechanisms that restrict access to resources to the users who are specifically allowed to access those resources

user groups →
sensitivity ↓

Insecure Direct Object References (IDOR)

arises when app uses user-supplied input to access objects directly

* attcr can mod input to obtain unauthorized access



Mass Assignment Vulnerability

* mass assign\autobinding\object injection
  ↳ allows automatically binding HTTP request params into prog code vars\objs.
  ⇒ framework easier to use
* attcr uses this methodology to create new params ✗ intended by dev
  ⇒ create or overwrite new var.\obj. in program code that was ✗ intended

ex., Egor Homakov's exploit against GitHub

Exploited by Egor Homakov against Github (a Rails app) to gain commit access to the Rails project

```
POST /users/42 HTTP/1.1
Host: vulnerable.com

id=1&user[email]=john@new.com
```

→

```
POST /users/42 HTTP/1.1
Host: vulnerable.com

id=1&user[email]=john@new.com
&user[admin]=true
```

```
def update
  user = User.find(params[:id])
  user.update_attributes(params[:user])
end
```