

Cloud Applications Architecture

...

Course 11 - Application Security

Security

Confidentiality

- only the authorized entities (people and/or systems) have access to the data

Integrity

- only the authorized entities can modify the data through well-defined procedures

Availability

- there is no point in having a well-guarded system if no one can use it.
- Also, earthquakes tend to overrule highly sophisticated security measures



Application Security

Create a secure environment for the user and his/her data.

Common processes/techniques:

- Authentication
- Authorization
- Encryption
- Input validation
- Hashing

CWE (Common Weakness Enumeration)

1. [Cross-site scripting \(XSS\)](#) (46.82)
2. Out-of-bounds write (46.17)
3. Improper input validation (33.47)
4. Out-of-bounds read (26.5)
5. Improper restriction of operations within the bounds of a memory buffer (23.73)
6. [SQL injection](#) (20.69)
7. Exposure of sensitive information to an unauthorized actor (19.16)
8. Use after free (18.87)
9. [Cross-site request forgery \(CSRF\)](#) (17.29)
10. OS command injection (16.44)

See complete list (top 25) [here](#)

CWE (Common Weakness Enumeration)

Most of these vulnerabilities are addressed by modern frameworks, libraries, and even languages.

Our duty is to keep them updated and to choose well-tested ones.

Packages managers can help - e.g. [npm audit](#).

There are also dedicated tools such as [snyk](#)

Authentication (AuthN)

Identify and ensure that the users are indeed whom they pretend to be.

Usually involves something the user:

- Has (e.g. smart card, code/token generator)
- Knows (e.g. password, the name of the first pet)
- Is (e.g. biometrics)

Multi-factor authentication = using 2 or 3 factors from above.

Flows and approaches are standardized - allows us to delegate it

Authorization (AuthZ)

Grant access to a specific **resource** based on a set of **roles, permissions or privileges**.

Takes place on each action.

Flows and processes are partially standardized. E.g. authorization on [google](#) and [github](#) works the same, but uses different **scopes** (defined by each of them).

Identity Providers (IdP)

Whenever we register on a website/platform, we create a new identity for ourselves on the internet.

Certain providers are more authoritative:

- We can use our gmail address on both Google and Facebook, but only Google can guarantee that we are indeed the owner of that address.

Facilitate **identity federation (social login)**.

E.g. Google, Facebook, Twitter, Linkedin, Microsoft, Apple, etc.

OAuth 2.0

[Link](#)





Authorization protocol based on several widely adopted flows.

Enables us to delegate authorization without providing our credentials.

Implemented by most providers and platforms.


Are your friends already on Yelp?

Many of your friends may already be here, now you can find out. Just log in and we'll display all your contacts, and you can select which ones to invite! And don't worry, we don't keep your email password or your friends' addresses. We loathe spam, too.







Your Email Service ☐  Hotmail ☐  ☐ AOL  Mail ☒ 

Your Email Address

Your Gmail Password

 75451442@qq.com (your Gmail email)

storage.com wants to

-  View and manage the files in your Google Drive 
-  View and manage Google Drive files and folders that you have opened or created with this app 
-  View the activity history of your Google apps 

Allow storage.com to do this?

By clicking Allow, you allow this app to use your information in accordance with their terms of service and privacy policies. You can remove this or any other app connected to your account in [My Account](#).

OAuth 2.0 Terminology

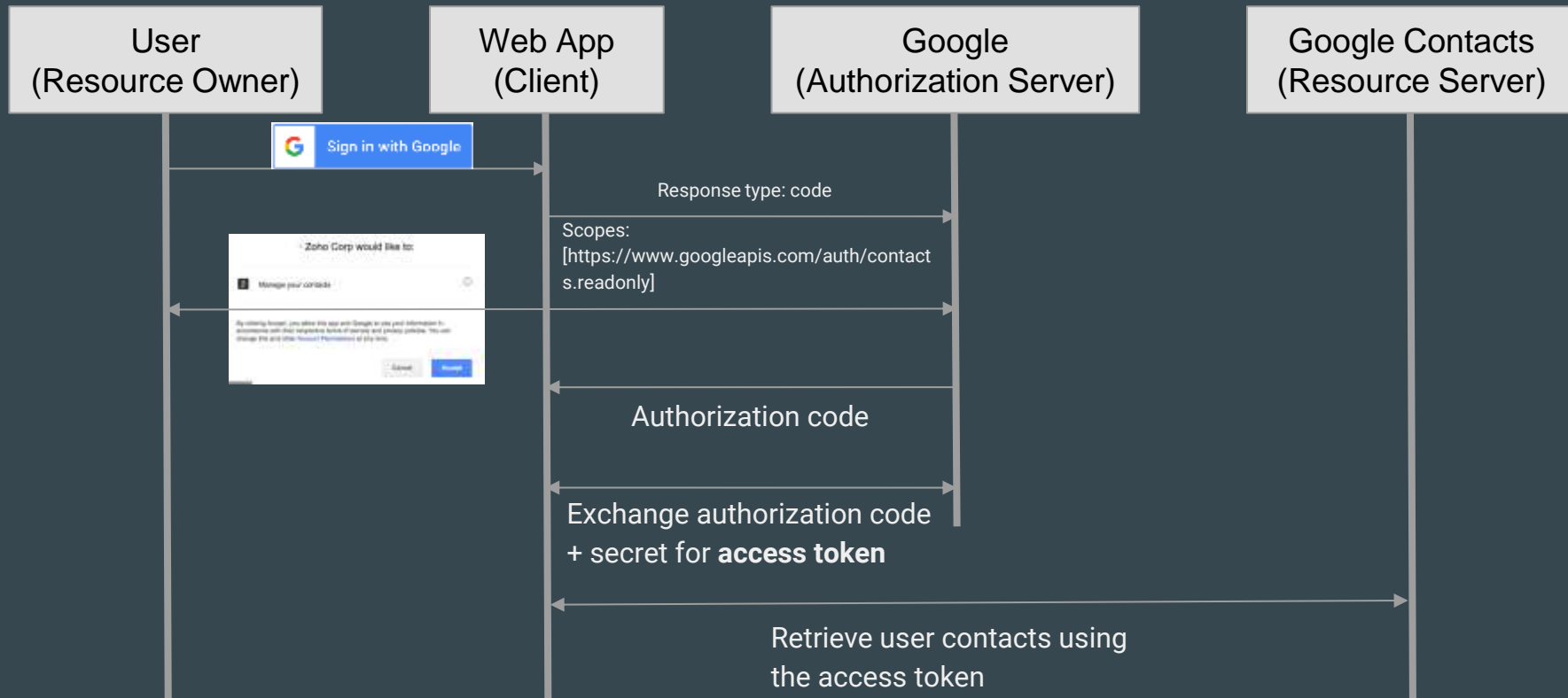
- **Resource Owner:** Entity that can grant access to a protected resource. Typically, this is the end-user.
- **Client:** Application requesting access to a protected resource on behalf of the Owner.
- **Resource Server:** The API you want to access.
- **Authorization Server:** Server that authenticates the Resource Owner and issues Access Tokens after getting proper authorization.
- **User Agent:** Agent used by the Resource Owner to interact with the Client (for example, a browser or a native application).

Source - [Auth0](#)

OAuth 2.0 - Flows/Grants

- **Authorization Code**: Used by web applications (ones built on frameworks such as Spring or Laravel). Requires a server to securely exchange information (including a **secret**) with the authorization server.
- **Implicit**: Less secure, but doesn't require a server (used for Single Page Applications and mobile apps) (obsolete).
- **Authorization Code with PKCE**: Improved version the implicit flow.
- **Resource Owner Password Credentials (ROPC)**: Involves the user providing the username and password. Intended only for special/legacy scenarios.
- **Client Credentials**: Intended for secure environments (e.g. server to server)

OAuth 2.0 - Authorization Code



OAuth 2.0 - Other Flows

With **implicit flow**, the access token is provided directly by the authorization server without requiring the authorization code and secret.

Authorization Code with PKCE is similar to simple authorization code, but doesn't involve a server to run the client (and store the secret). Instead it relies on the SPA/mobile app to generate a ***code challenge*** (sent with the initial request to the authorization server) and a ***code verifier*** (sent alongside the authorization code).

ROPC and **Client Credentials** simply involve exchanging the credentials for the access token.

OpenID Connect

Built on top of OAuth 2.0. [Link](#)

Facilitates **authentication**. (the access token contains no information about the user)

Works with [JWT tokens](#) (base64 encoded; can contain arbitrary data) - known as idToken (as opposed to access token).

To retrieve an idToken, simply add ***openid*** to the scopes and ***id_token*** to the response type. The authorization server will provide the idToken alongside the access token (and optionally ***refresh_token***).

Working with JWT Tokens

The JWT tokens can be validated **offline**

- The resource server does not have to call the authorization server in order to validate the token.
- Downside is that tokens might be out of sync/**stale** (in case any information contained in the token was changed since it was issued).
- Online validation means calling the authorization server each time.

Compared to session based authentication (usually implemented using cookies), token based authentication is stateless (since the entire info about the user can be stored in the token). This means better/easier horizontal scalability. For highly secured scenarios, we might store the tokens on the server and keep track of invalidated ones.

We can include other info about the user such as custom roles and permissions.

Managed Authentication Services

[Firebase Authentication](#)

[AWS Cognito](#)

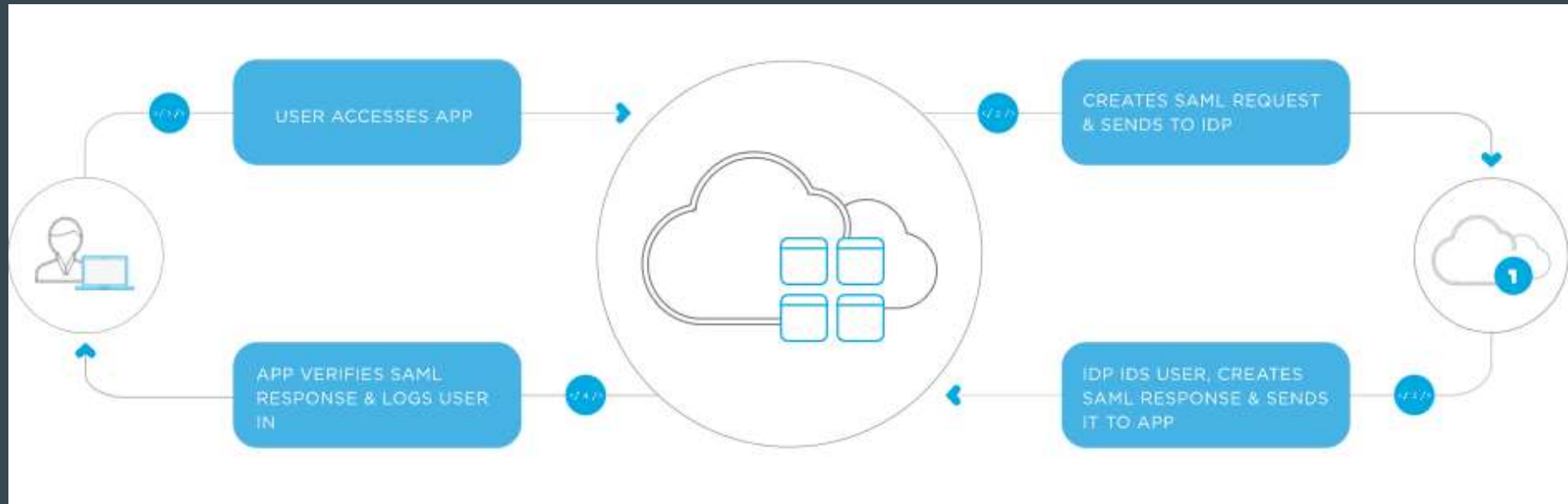
Azure Active Directory [B2B](#) (standard) and [B2C](#)

[Auth0](#)

[Okta](#)

[OneLogin](#)

SSO and SAML



Encoding, Hashing, Encryption

Encoding/decoding is used to facilitate communication

- E.g. in order to store some data in an URL, it has to be URL encoded (i.e. escape certain characters such as spaces and ?).
- Can be reversed (decoding) - **provides no security**.

Hashing is used to verify the integrity of data and to store passwords.

- Integrity is verified by comparing a given checksum to the one generated by ourselves.
E.g. to ensure that a file was completely received and authentic
- Cannot be reversed.

Encryption is used to safely send and store data.

- Can be reversed (decrypted) if both the algorithm (cypher) and the key are known.

Encryption (Cryptography)

One of the pillars that enable us to do more online (work, make purchases, etc).

- Enables secure **transfer** of data (TLS, IPsec).
- Enables secure **storage** of data (data at rest).
- Enables **digital signatures**.

2 types of encryption:

- **Symmetric**: one key, works with large data (e.g. [AES](#))
- **Asymmetric** or **public key**: 2 keys - **public** (encryption) and **private** (decryption) (e.g. [RSA](#)). Cannot handle large data.

There is also **envelope** encryption which combines both (e.g. AES for the actual data and RSA for the AES key).

Encryption in Cloud

Offered by all providers for most storage services.

Most of the times enabled by default and transparent to developers and users.

Dedicated services (main role is to store encryption keys) such as [AWS KMS](#), [Google KMS](#) or [Azure Key Vault](#).

Can be entirely handled by the provider (generating and managing the keys and data encryption/decryption) or we can choose to **supply/provide** our own keys.
E.g. [AWS S3 encryption](#), [Google storage encryption](#).

HTTPS, SSL, TLS

SSL (secure sockets layer) - Developed by Netscape in 1995.

TLS (transport layer security) - the successor of SSL standardized by IETF (i.e. SSL v4).

The SSL term is still used today especially for **SSL Certificates**.

A website/server that supports TLS is HTTPS enabled.

Based on RSA - allows browsers to verify the authenticity of the origin/server.

Secure connection established through the **SSL/TLS handshake**.

Secrets Management

Similar to cryptographic key management.

Where do we store API keys, client secrets (e.g. for OAuth), and passwords?

Active Directory

Users, groups, domains, forests

Azure Active Directory

Hybrid Identity

AAD B2B vs B2C

Resources

[Authentication on the Web](#) (video)

[OAuth 2.0 and OpenID Connect](#) (video)

[Transport Layer Security \(TLS\) - Computerphile](#)