Identity Flows & Security

Compiled by Svet Voloshin

Diagrams by Lawrence Newcombe

What are Identity Flows?

Identity Flows are a way of automating and streamlining the process of authenticating users and granting them access to applications and resources. They are typically used in conjunction with identity and access management (IAM) solutions, such as Okta or PingFederate.

Identity Flows typically consist of a series of steps that are performed when a user attempts to access an application or resource. These steps may include:

- **Authentication**: The user is prompted to provide their credentials, such as a username and password.
- **Authorization**: The user's credentials are checked against a database to verify that they are authorized to access the application or resource.
- **Provisioning**: If the user is authorized, they are provisioned with the appropriate access rights.

Identity Flows can be used to automate a wide range of tasks, such as:

- **Single sign-on**: Users can log in to multiple applications and resources using a single set of credentials.
- **Multi-factor authentication**: Users can be prompted to provide additional factors of authentication, such as a security code or a fingerprint scan, to increase the security of their accounts.
- **User provisioning and deprovisioning**: Users can be automatically provisioned with the appropriate access rights when they are hired or terminated.

Identity Flow Types

1. **Authorization Code Flow**: This is the **most common flow**, especially for server-side applications. The application redirects the user to an authorization server which the user logs into. The server then redirects the user back to the application with an authorization code. The application exchanges this code for an access token.
2. **Implicit Flow**: This flow is used for applications that run in a browser using a scripting language such as JavaScript. In the Implicit flow, the token is returned directly without an intermediate code exchange step. This flow is **less secure and not recommended** for new applications as of the OAuth 2.1 draft specification.
3. **Resource Owner Password Credentials Flow**: This flow involves the application asking the user for their username and password and then using these to request an access token from the authorization server. This flow is **also not recommended unless the client is highly trusted**.
4. **Client Credentials Flow**: This flow is used for **server-to-server** communication where a client application needs to access a resource server **without user interaction**.
5. **Device Code Flow**: This flow is designed for clients executing on devices that **do not have an easy text input method** (e.g., game consoles, video streaming devices). The device requests a code and verifies it using another device like a smartphone or computer.
6. **Hybrid Flow**: This flow is a combination of the **Implicit and Authorization Code** flows and returns tokens from both the authorization endpoint and the token endpoint.
7. **The OAuth 2.0 JWT Bearer Flow** for **Server-to-Server** Integration is a security protocol where one server authenticates to another using a JSON Web Token (JWT) as the client's credentials, typically used to enable secure server-to-server communication without transmitting sensitive information like passwords.

Each flow has different security considerations and is appropriate for different types of applications (e.g., web applications, single-page applications, mobile applications, etc.). The selection of the flow will depend on factors like the application type, the trust level between the application and the authorization server, the presence of a backend server in the application architecture, and the requirements around tokens and user information.

TLS vs. SSL

- TLS stands for Transport Layer Security, while SSL stands for Secure Sockets Layer. They are both cryptographic protocols that secure communications over the internet.
- TLS 1.3 is the latest version of TLS, while SSL 3.0 is the most recent version of SSL. TLS 1.3 is more secure than SSL 3.0 and is considered to be the standard for secure communications.
- TLS is used to protect a wide variety of applications, including web browsing, email, and file transfers. SSL was originally developed for web browsing, but it is now used by a variety of other applications as well.
- TLS is more widely supported than SSL. Most modern browsers and web servers support TLS, while support for SSL is declining.

In general, TLS is considered to be a more secure and modern protocol than SSL. If you are concerned about the security of your communications, then you should use TLS instead of SSL.

Feature	**TLS**	**SSL**
Name	Transport Layer Security	Secure Sockets Layer
Latest version	TLS 1.3	SSL 3.0
Security	More secure than SSL	Less secure than TLS
Applications	Web browsing, email, file transfers, etc.	Web browsing, email, etc.
Support	Widely supported	Support declining

Certificate Authority (CA)

A Certificate Authority (CA) is an organization or entity responsible for issuing digital certificates, which are data files used to cryptographically link an entity with a public key. Digital certificates are commonly used for securing communications over the internet, including during SSL/TLS sessions that protect web traffic. CAs are a cornerstone of public key infrastructure (PKI), which is a framework for managing digital keys and certificates.

Functions of a Certificate Authority

1. **Issuance**: Upon receiving a certificate request from an individual or organization, the CA verifies the credentials provided. If they meet the CA's policies, a digital certificate is issued to the requester, binding a specific public key to an individual, device, or service.
2. **Verification**: The CA confirms the identity of the certificate requester before issuing a certificate. The level of verification varies depending on the type of certificate. For example, an Extended Validation (EV) certificate requires thorough verification of the requesting entity’s identity.
3. **Revocation**: CAs have the ability to revoke certificates. A Certificate Revocation List (CRL) is maintained, containing all revoked certificates that should no longer be trusted.
4. **Renewal**: Digital certificates have a set validity period. The CA is responsible for renewing certificates either upon request or automatically, depending on the policy.
5. **Record-Keeping**: The CA maintains records of the certificates it has issued, and often provides services like timestamping to help keep track of when certificates were created or revoked.

Types of Certificate Authorities

1. **Root CA**: The top-level CA whose public key serves as a trusted 'root' for an entire certificate chain. Certificates from a root CA are self-signed and are installed in the root certificate store of end-user devices and applications.
2. **Intermediate CA**: Acts under the root CA and issues certificates to end entities or even other intermediate CAs. This creates a chain of trust but adds a layer of security; compromising an intermediate CA won’t directly compromise the root CA.
3. **Issuing CA**: The CA that directly issues certificates to end users or end entities. It usually operates under an intermediate CA.

Certificate Authority (CA) - continued...

Importance of a Certificate Authority

1. **Trust**: In the digital world, CAs serve as a third-party authority that both parties (usually a client and a server) can trust. When a CA signs a certificate, it's confirming that the individual or entity associated with the certificate is indeed who they claim to be.
2. **Security**: CAs enable secure communications and transactions over the internet by issuing certificates required for SSL/TLS encryption.
3. **Identity Verification**: The process of getting a certificate often involves some level of organizational or individual identity verification, lending credibility to websites and systems.
4. **Data Integrity**: Certificates also help in ensuring that the data has not been tampered with during transmission.
5. **Legal and Compliance**: Many industry regulations require the use of digital certificates to meet security and privacy standards.

Risks

1. **Compromise**: If a CA is compromised, the attacker could issue fraudulent certificates, which could be used to impersonate websites or services.
2. **Weak Practices**: If a CA does not rigorously verify the identity of the certificate requester, it could issue certificates to malicious actors.
3. **Cost**: High-assurance certificates, especially those requiring extensive verification like EV certificates, can be costly to obtain.

Thus, a Certificate Authority plays a **critical role** in internet security. It is the trusted third party that allows you to verify the identity of a certificate owner and the validity of the certificate itself.

DigiCert: www.digicert.com

Comodo: www.comodo.com

Symantec: www.symantec.com

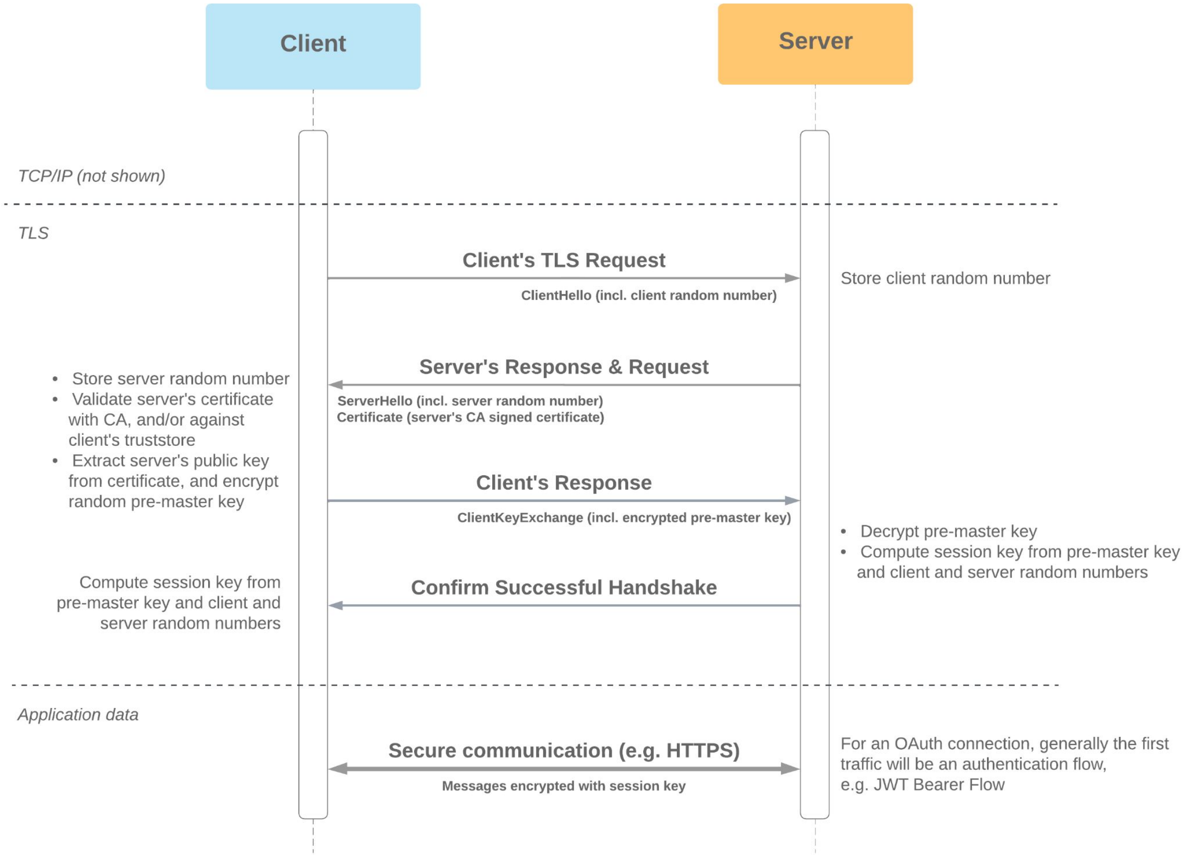
Let's Encrypt: letsencrypt.org

GlobalSign: www.globalsign.com

STANDARD (ONE-WAY) TLS

Keystore: A keystore is a repository for storing cryptographic keys and certificates. It is typically used to store the private key of a server or client, as well as the certificates that are trusted by the server or client.

Truststore: A truststore is a repository for storing certificates that are trusted by a server or client. It is typically used to store the certificates of certificate authorities (CAs) that are trusted by the server or client.



Client

Server

TCP/IP (not shown)

TLS

Client's TLS Request

ClientHello (incl. client random number)

Store client random number

Server's Response & Request

ServerHello (incl. server random number)
Certificate (server's CA signed certificate)

- Store server random number
- Validate server's certificate with CA, and/or against client's truststore
- Extract server's public key from certificate, and encrypt random pre-master key

Client's Response

ClientKeyExchange (incl. encrypted pre-master key)

- Decrypt pre-master key
- Compute session key from pre-master key and client and server random numbers

Confirm Successful Handshake

Compute session key from pre-master key and client and server random numbers

Application data

Secure communication (e.g. HTTPS)

Messages encrypted with session key

For an OAuth connection, generally the first traffic will be an authentication flow, e.g. JWT Bearer Flow

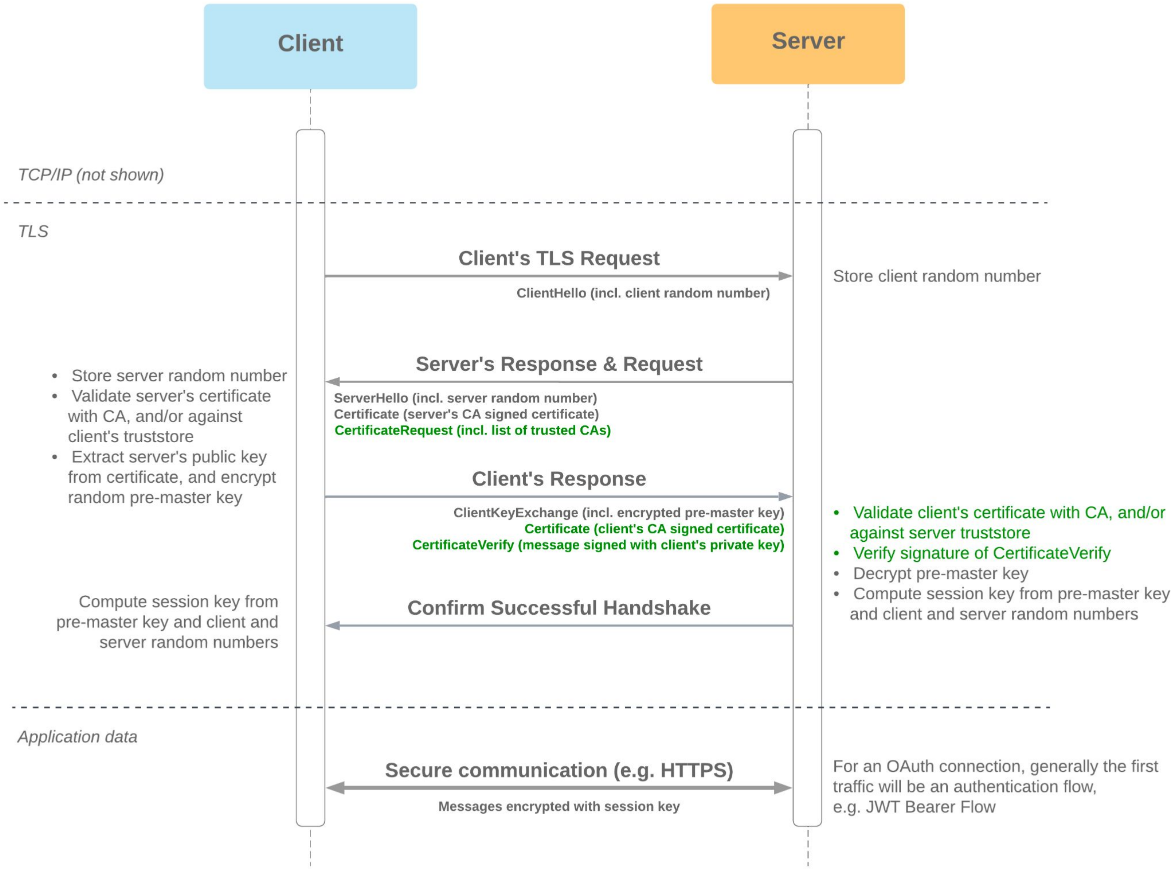
Diagram illustrates key concepts of TLS 1.2 but omits some messages and abstracts some details - see links below for a more in depth oveview

MUTUAL TLS

Keystores and truststores are typically used in conjunction with Transport Layer Security (TLS) and Secure Sockets Layer (SSL). TLS and SSL are cryptographic protocols that are used to secure communications between two parties.

Keystores and truststores are typically stored in a secure location, such as an encrypted file or a hardware security module (HSM).

The contents of keystores and truststores should be kept confidential.



Client

Server

TCP/IP (not shown)

TLS

Client's TLS Request

ClientHello (incl. client random number)

Store client random number

Server's Response & Request

ServerHello (incl. server random number)
Certificate (server's CA signed certificate)
CertificateRequest (incl. list of trusted CAs)

- Store server random number
- Validate server's certificate with CA, and/or against client's truststore
- Extract server's public key from certificate, and encrypt random pre-master key

Client's Response

ClientKeyExchange (incl. encrypted pre-master key)
Certificate (client's CA signed certificate)
CertificateVerify (message signed with client's private key)

- Validate client's certificate with CA, and/or against server truststore
- Verify signature of CertificateVerify
- Decrypt pre-master key
- Compute session key from pre-master key and client and server random numbers

Confirm Successful Handshake

Compute session key from pre-master key and client and server random numbers

Application data

Secure communication (e.g. HTTPS)

Messages encrypted with session key

For an OAuth connection, generally the first traffic will be an authentication flow, e.g. JWT Bearer Flow

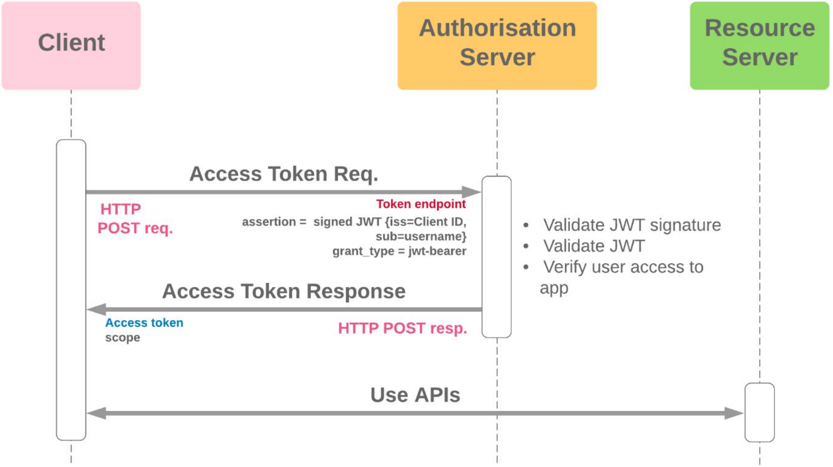
Diagram illustrates key concepts of Mutual TLS 1.2 but omits some messages and abstracts some details - see links below for a more in depth oveview

Additional elements for Mutual TLS are shown in **green**

JWT Bearer Flow

OAuth 2.0 JWT Bearer Flow for Server-to-Server Integration

Sometimes you want to authorize servers to access data without interactively logging in each time the servers exchange information. For these cases, you can use the OAuth 2.0 JSON Web Token (JWT) bearer flow. This flow uses a certificate to sign the JWT request and doesn't require explicit user interaction. However, this flow does require prior approval of the client app.



A JSON Web Token (JWT) is a compact and self-contained way for securely transmitting information between parties as a JSON object. JWTs can be signed or encrypted, and they can be used for authentication, authorization, information exchange, and more.

JWTs are made up of three parts, separated by dots:

- **Header**: The header contains information about the token, such as the type of token, the hashing algorithm used to sign or encrypt the token, and the token's expiration time.
- **Payload**: The payload contains the claims, which are the actual data that is being transmitted. Claims can be used to represent anything, such as a user's identity, their permissions, or the time that they last logged in.
- **Signature**: The signature is used to verify the authenticity and integrity of the token. It is created using a secret key that is shared between the parties that are exchanging the token.

SERVICE PROVIDER (SP) INITIATED SSO

Use Case:
A user wants to access Salesforce from their web browser. They go to the Salesforce login page and click on the "Sign in with SAML" button.

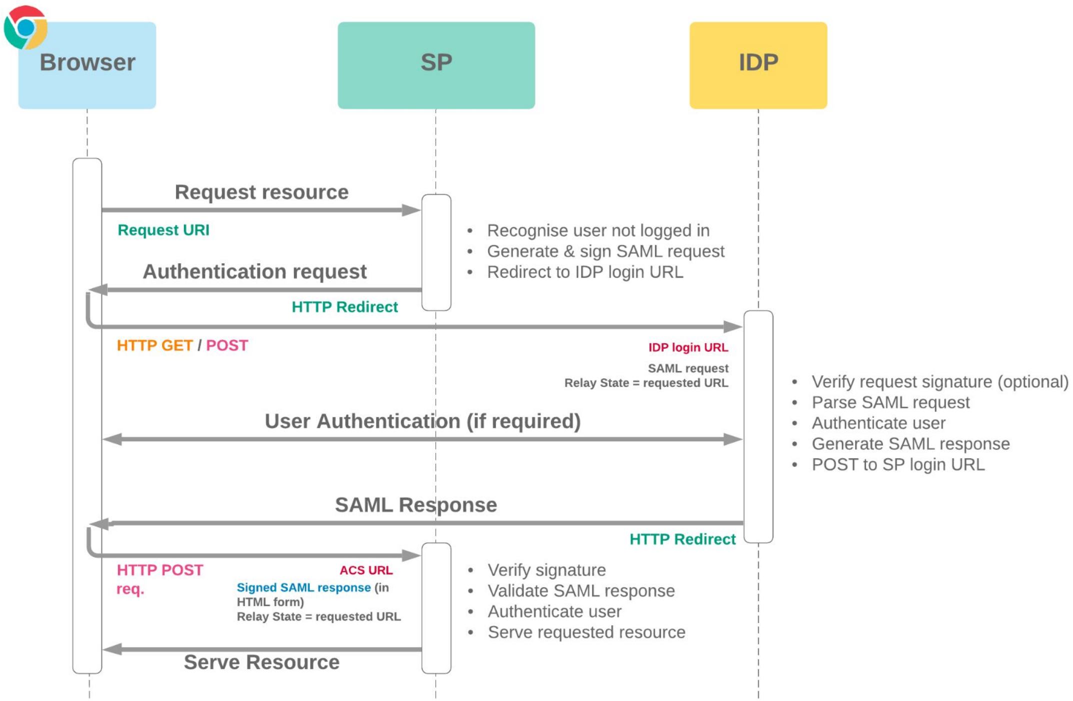
The Salesforce SP initiates an SSO request to the identity provider (IdP). The IdP authenticates the user and returns an authentication assertion to the SP.

The SP validates the authentication assertion and creates a session for the user. The user is then redirected to the Salesforce home page.

Here are the steps involved in SP initiated SSO with Salesforce:

1. The user clicks on the "Sign in with SAML" button on the Salesforce login page.
2. The SP sends an SSO request to the IdP.
3. The IdP authenticates the user and returns an authentication assertion to the SP.
4. The SP validates the authentication assertion and creates a session for the user.
5. The user is redirected to the Salesforce home page.

SP initiated SSO is a secure and convenient way for users to access Salesforce. It eliminates the need for users to remember and enter their Salesforce credentials every time they want to access Salesforce.



Browser

SP

IDP

Request resource

Request URI

- Recognise user not logged in
- Generate & sign SAML request
- Redirect to IDP login URL

Authentication request

HTTP Redirect

HTTP GET / POST

IDP login URL

SAML request
Relay State = requested URL

- Verify request signature (optional)
- Parse SAML request
- Authenticate user
- Generate SAML response
- POST to SP login URL

User Authentication (if required)

SAML Response

HTTP Redirect

HTTP POST req.

ACS URL

Signed SAML response (in HTML form)
Relay State = requested URL

- Verify signature
- Validate SAML response
- Authenticate user
- Serve requested resource

Serve Resource

IDENTITY PROVIDER (IDP) INITIATED SSO

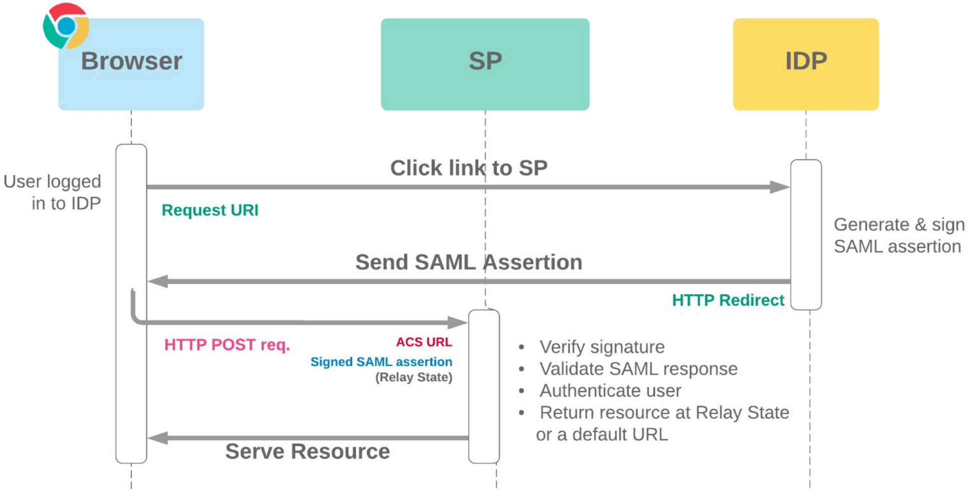
Use Case:

A company called Acme Corporation uses Salesforce as their customer relationship management (CRM) system. They have a large number of employees who need to access Salesforce from a variety of devices, including laptops, tablets, and smartphones.

Acme Corporation decides to implement IdP Initiated SSO with Okta. This means that when an employee tries to access Salesforce, they will first be redirected to Okta, where they will authenticate using their Okta credentials. Once they have been authenticated, Okta will redirect them back to Salesforce, where they will be able to access their data.

Here are some additional details about how IdP Initiated SSO with Okta works:

- When an employee tries to access Salesforce, they will be redirected to the Okta login page.
- The employee will enter their Okta credentials and click the "Login" button.
- Okta will authenticate the employee and redirect them back to Salesforce.
- Salesforce will verify the employee's identity and grant them access to the application.



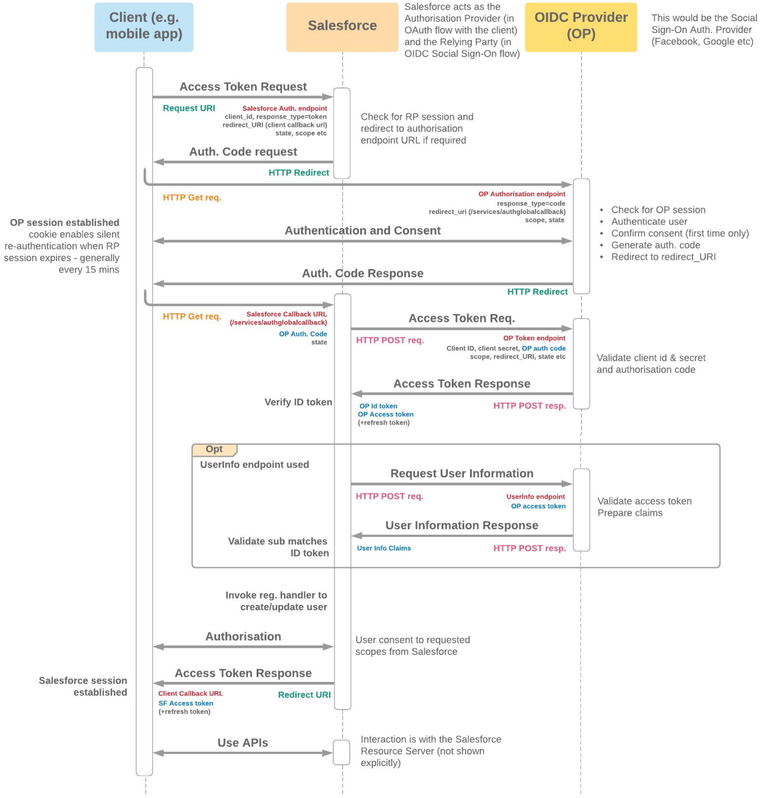
OAUTH 2.0 USER-AGENT WITH SOCIAL SIGN ON

Use Case:

- A company called Acme Corporation uses Salesforce as their customer relationship management (CRM) system. They want to allow their customers to sign in to Salesforce using their social media accounts, such as Facebook or Google.
- Acme Corporation decides to implement OAuth 2.0 User-Agent with Social Sign On Salesforce. This means that when a customer tries to sign in to Salesforce, they will first be redirected to the Salesforce login page. They will then be given the option to sign in using their social media account.
- If the customer chooses to sign in using their social media account, they will be redirected to the social media provider's website. They will then be prompted to enter their social media credentials and grant Salesforce permission to access their account.
- Once the customer has authenticated with the social media provider, they will be redirected back to Salesforce. Salesforce will then verify the customer's identity and grant them access to the application.

Here are some additional details about how OAuth 2.0 User-Agent with Social Sign On Salesforce works:

- When a customer tries to sign in to Salesforce, they will be redirected to the Salesforce login page.
- The customer will see a list of social media providers that they can use to sign in.
- If the customer chooses to sign in using a social media provider, they will be redirected to the social media provider's website.
- The social media provider will authenticate the customer and redirect them back to Salesforce.
- Salesforce will verify the customer's identity and grant them access to the application.


Client (e.g. mobile app)
Salesforce
Salesforce acts as the Authorisation Provider (in OAuth flow with the client) and the Relying Party (in OIDC Social Sign-On flow)
OIDC Provider (OP)
This would be the Social Sign-On Auth. Provider (Facebook, Google etc)
Access Token Request
Request URI
Salesforce Auth. endpoint
client_id, response_type=token
redirect_URI (client callback url)
state, scope etc
Check for RP session and redirect to authorisation endpoint URL if required
Auth. Code request
HTTP Redirect
HTTP Get req.
OP Authorisation endpoint
response_type=code
redirect_uri (/services/authglobalcallback)
scope, state
OP session established
cookie enables silent re-authentication when RP session expires - generally every 15 mins
Authentication and Consent
• Check for OP session
• Authenticate user
• Confirm consent (first time only)
• Generate auth. code
• Redirect to redirect_URI
Auth. Code Response
HTTP Redirect
HTTP Get req.
Salesforce Callback URL (/services/authglobalcallback)
OP Auth. Code
state
Access Token Req.
HTTP POST req.
OP Token endpoint
Client ID, client secret, OP auth code
scope, redirect_URI, state etc
Validate client id & secret and authorisation code
Access Token Response
Verify ID token
OP Id token
OP Access token
(+refresh token)
HTTP POST resp.
Opt
UserInfo endpoint used
Request User Information
HTTP POST req.
UserInfo endpoint
OP access token
Validate access token
Prepare claims
User Information Response
Validate sub matches ID token
User Info Claims
HTTP POST resp.
Invoke reg. handler to create/update user
Authorisation
User consent to requested scopes from Salesforce
Access Token Response
Salesforce session established
Client Callback URL
SF Access token
(+refresh token)
Redirect URI
Use APIs
Interaction is with the Salesforce Resource Server (not shown explicitly)

OPENID CONNECT (OIDC)

Use Case:

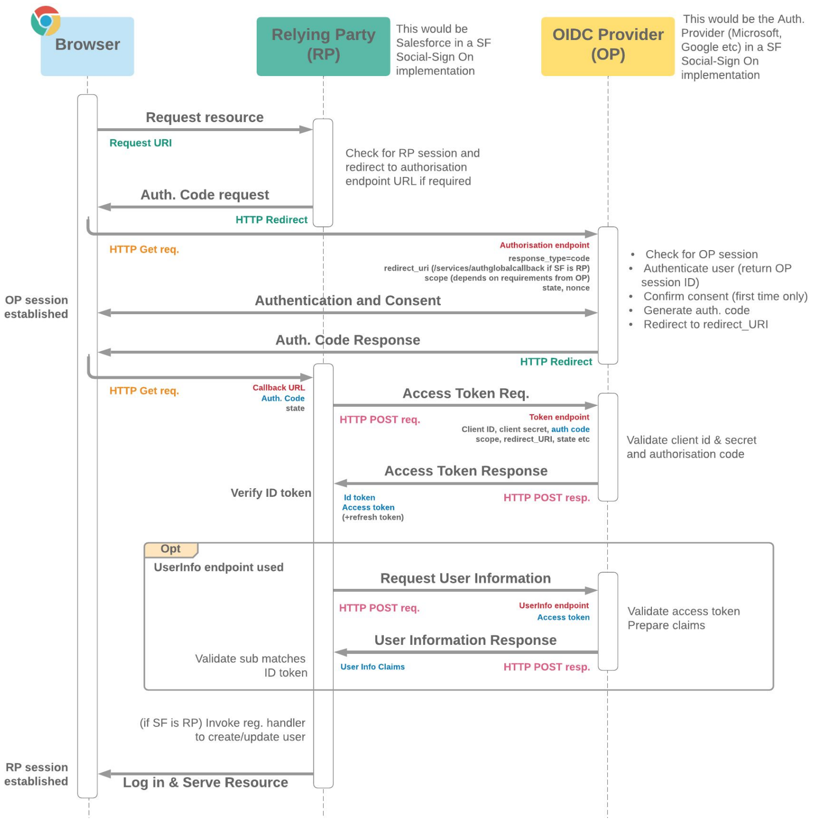
A company called Universal Containers uses Salesforce as their customer relationship management (CRM) system. They have a large number of customers who are active in their Social Salesforce Customer Community. Universal Containers wants to make it easier for their customers to participate in the community by allowing them to sign in using OpenID Connect.

Universal Containers decides to implement OIDC flow with Social Salesforce Customer Community. This means that when a **customer tries to sign in to the community**, they will first be redirected to the OIDC provider's website. They will then be prompted to enter their OIDC credentials and grant Salesforce permission to access their account.

Once the customer has authenticated with the OIDC provider, they will be redirected back to the community. The community will then verify the customer's identity and grant them access.

Here are some additional details about how OIDC flow with Social Salesforce Customer Community works:

- When a customer tries to sign in to the community, they will be redirected to the OIDC provider's website.
- The customer will see a list of OIDC providers that they can use to sign in.
- If the customer chooses to sign in using an OIDC provider, they will be redirected to the OIDC provider's website.
- The OIDC provider will authenticate the customer and redirect them back to the community.
- The community will verify the customer's identity and grant them access.



Browser

Relying Party (RP)

This would be Salesforce in a SF Social-Sign On implementation

OIDC Provider (OP)

This would be the Auth. Provider (Microsoft, Google etc) in a SF Social-Sign On implementation

Request resource

Request URI

Check for RP session and redirect to authorisation endpoint URL if required

Auth. Code request

HTTP Redirect

HTTP Get req.

Authorisation endpoint

response_type=code
redirect_uri (/services/authglobalcallback if SF is RP)
scope (depends on requirements from OP)
state, nonce

- Check for OP session
- Authenticate user (return OP session ID)
- Confirm consent (first time only)
- Generate auth. code
- Redirect to redirect_URI

OP session established

Authentication and Consent

Auth. Code Response

HTTP Redirect

HTTP Get req.

Callback URL
Auth. Code
state

Access Token Req.

HTTP POST req.

Token endpoint
Client ID, client secret, auth code
scope, redirect_URI, state etc

Validate client id & secret and authorisation code

Access Token Response

Verify ID token

Id token
Access token
(+refresh token)

HTTP POST resp.

Opt

UserInfo endpoint used

Request User Information

HTTP POST req.

UserInfo endpoint
Access token

Validate access token
Prepare claims

User Information Response

Validate sub matches ID token

User Info Claims

HTTP POST resp.

(if SF is RP) Invoke reg. handler to create/update user

RP session established

Log in & Serve Resource

AUTHORISATION CODE WITH PKCE

Use Case: Secure Mobile Application Access to Salesforce Data using OAuth 2.0 Authorization Code Flow with PKCE

Scenario

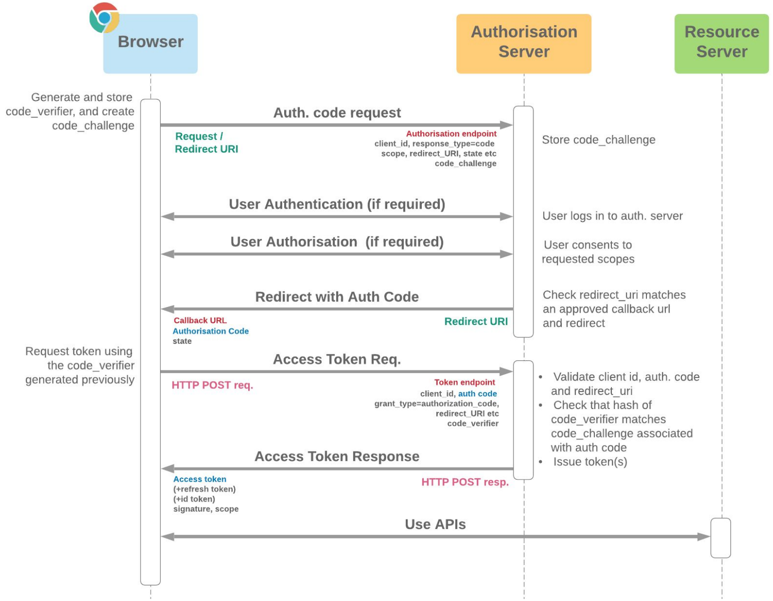
A financial services company has built a mobile app using Salesforce Mobile Publisher to give their agents real-time access to customer data stored in Salesforce. The mobile application needs to ensure that the authorization process is highly secure, as sensitive financial data is involved. To increase the security of the OAuth 2.0 authorization code flow, the application is designed to use PKCE (Proof Key for Code Exchange).

Requirement

- The mobile application must authenticate agents securely against the Salesforce instance.
- The authorization process should not expose any tokens that can be intercepted and misused.
- Agents should have access to specific data based on their roles and permissions.

Steps

1. **Initialization**: The mobile app generates a unique and random string called the **code_verifier**. It also hashes this string to create the code_challenge.
2. **Authorization Request**: When an agent starts the application and clicks the "Login via Salesforce" button, the app initiates an OAuth 2.0 Authorization Code Flow. It sends an authorization request to the Salesforce authorization endpoint and includes the **code_challenge** and a method parameter that specifies how the challenge was derived (e.g., S256 for SHA-256).
3. **Salesforce Authentication**: Salesforce's authorization server authenticates the agent via usual means (username/password, multi-factor authentication, etc.).
4. **Authorization Grant**: Upon successful authentication, Salesforce sends an authorization code back to the mobile app via a redirect to the app's registered URI.
5. **Token Request**: The mobile app sends this authorization code along with the original code_verifier to Salesforce's token endpoint. The server uses the code_verifier to generate its own version of the code_challenge.
6. **Token Verification and Issue**: Salesforce verifies that the regenerated code_challenge matches the original code_challenge sent during the authorization request. If they match, it ensures that the same client application that initiated the authorization request is also the one exchanging the authorization code for the access token. Salesforce then issues an access token and optional refresh token.
7. **Access Salesforce Data**: The mobile application uses the received access token to securely query Salesforce and retrieve customer data as per the permissions set in Salesforce for that agent.
8. **Role-based Data Access**: Salesforce roles and permissions determine which records and fields the agent can view or edit, thus ensuring compliance with data access policies.


Browser
Authorisation Server
Resource Server
Generate and store code_verifier, and create code_challenge
Auth. code request
Request / Redirect URI
Authorisation endpoint
client_id, response_type=code
scope, redirect_URI, state etc
code_challenge
Store code_challenge
User Authentication (if required)
User logs in to auth. server
User Authorisation (if required)
User consents to requested scopes
Redirect with Auth Code
Callback URL
Authorisation Code
state
Redirect URI
Check redirect_uri matches an approved callback url and redirect
Request token using the code_verifier generated previously
Access Token Req.
HTTP POST req.
Token endpoint
client_id, auth code
grant_type=authorization_code,
redirect_URI etc
code_verifier
• Validate client id, auth. code and redirect_uri
• Check that hash of code_verifier matches code_challenge associated with auth code
• Issue token(s)
Access Token Response
Access token
(+refresh token)
(+id token)
signature, scope
HTTP POST resp.
Use APIs

Benefits

- **Enhanced Security**: PKCE prevents an attacker from intercepting the authorization code and using it to gain a token, as they would also need the unique **code_verifier** generated by the client.
- **Better Compliance**: Using PKCE with Salesforce Mobile Publisher ensures that the application adheres to strong security protocols, making it easier to comply with financial services regulations.
- **User Experience**: Agents benefit from secure, seamless access to essential data through their mobile devices.

SHA-256

- SHA-256 stands for Secure Hash Algorithm 256. It is a cryptographic hash function that takes an input of any size and produces an output of 256 bits, or 32 bytes.
- SHA-256 is a one-way function, meaning that it is impossible to reverse the process and get the original input from the output. This makes it ideal for use in digital signatures and other applications where it is important to be able to verify the authenticity of data.
- SHA-256 is considered to be a secure hash function and is widely used in applications such as blockchain, file verification, and password hashing.

Here are some additional details about SHA-256:

- SHA-256 is a member of the SHA-2 family of hash functions, which also includes SHA-512 and SHA-384. SHA-256 is generally considered to be more efficient than SHA-512 and SHA-384, while still providing a comparable level of security.
- SHA-256 is based on the Merkle–Damgård construction, which is a method for constructing secure hash functions. The Merkle–Damgård construction involves repeatedly applying a compression function to the input data, until the output reaches the desired length.
- The compression function used in SHA-256 is a modified version of the Davies–Meyer hash function. The Davies–Meyer hash function is a simple and efficient hash function that is known to be secure.
- SHA-256 has been subjected to extensive cryptanalysis and no significant weaknesses have been found. This makes SHA-256 a widely trusted and secure hash function.

SALT

- Salt is a random string that is added to a password before it is hashed. This makes it more difficult for an attacker to crack the password, even if they have access to the hashed password database.
- Salt can be any length, but it is typically 8-128 characters long. The longer the salt, the more difficult it is for an attacker to crack the password.
- Salt is typically stored in the same database as the hashed password. This allows the server to verify the password without having to store the original password in plaintext.
- Salt can be used with any hashing algorithm. However, it is most effective when used with a secure hashing algorithm, such as bcrypt or PBKDF2.

Here are some additional details about salt in cryptography:

- Salt helps to protect passwords from dictionary attacks and rainbow table attacks. Dictionary attacks involve trying every word in a dictionary as a password. Rainbow table attacks involve pre-computing hashes of common passwords, so that they can be quickly checked against a hashed password database.
- Salt makes it more difficult for an attacker to crack a password, even if they have access to the hashed password database. This is because the attacker would need to know both the password and the salt in order to recreate the original password.
- Salt is a simple and effective way to improve the security of passwords. It is a widely used security practice and is recommended by most password security guidelines.

Base64

- Base64 is a way to encode binary data into a text format that can be easily transmitted over a network or stored in a file.
- Base64 uses a 64-character alphabet that includes 26 letters, 26 numbers, and 10 special characters.
- To encode binary data, each 8-bit byte of data is converted into 6 bits, which are then represented by one character from the Base64 alphabet.
- To decode Base64 data, each character from the Base64 alphabet is converted into 6 bits, which are then reassembled into 8-bit bytes.
- Base64 is a popular choice for encoding binary data because it is easy to implement and it is relatively secure.

Here are some additional details about Base64:

- Base64 is a binary-to-text encoding scheme that represents binary data in a textual form by translating it into a base64 alphabet.
- The Base64 alphabet is composed of 64 characters: 26 uppercase letters, 26 lowercase letters, 10 numerals, and '+', '/', '='.
- Base64 is often used to encode data that needs to be transferred over a network, such as email attachments or files uploaded to a website.
- Base64 is also often used to encode data that needs to be stored in a file, such as configuration files or passwords.
- Base64 is a relatively secure encoding scheme, but it is not unbreakable.

NONCE

In cryptography, a nonce is an arbitrary number that can be used just once in a cryptographic communication. It is often a random or pseudo-random number issued in an authentication protocol to ensure that old communications cannot be reused in replay attacks.

Nonces are used to prevent replay attacks, which are a type of attack where an attacker sends a previously captured message to a server in an attempt to gain unauthorized access. By using a nonce, the server can verify that the message has not been previously sent and that it is coming from a legitimate source.

Nonces are also used in cryptographic hash functions and initialization vectors. In a cryptographic hash function, a nonce is used to ensure that the hash of a message is unique. In an initialization vector, a nonce is used to ensure that the encryption of a message is unique.

Here are some of the uses of nonces:

- Preventing replay attacks: Nonces can be used to prevent replay attacks by ensuring that each message is unique.
- Ensuring the uniqueness of hashes: Nonces can be used to ensure the uniqueness of hashes by ensuring that each message is hashed with a unique nonce.
- Ensuring the uniqueness of encryption: Nonces can be used to ensure the uniqueness of encryption by ensuring that each message is encrypted with a unique nonce.

Nonces are an important part of many cryptographic protocols and algorithms. They help to prevent replay attacks and ensure the security of communications.

URI

A URI, or Uniform Resource Identifier, is a string of characters used to identify a name or a resource on the Internet. URIs provide a simple and extensible means for identifying a resource, which could be anything like a document, image, downloadable file, or service, among others.

URIs are a broader category that includes Uniform Resource Locators (URLs) and Uniform Resource Names (URNs):

- URL (Uniform Resource Locator): Specifies where a particular resource is accessible and how to retrieve it. A URL contains information like the protocol to use (HTTP, HTTPS, FTP, etc.), the domain name, the path to the resource, and optionally a query string and fragment.
    - Example: https://www.example.com/page?name=value#section
- URN (Uniform Resource Name): Provides a way of identifying a resource by name in a particular namespace. Unlike URLs, URNs don't have to be updated if the resource moves to a different location.
    - Example: urn:isbn:0451450523 (for identifying a book by its ISBN number)

URI Elements

- **Scheme**: Specifies the protocol or method used to access the resource (e.g., http, https, ftp, mailto).
- **Authority**: Specifies the domain that holds the resource, often including subdomain and port number (e.g., www.example.com:8080).
- Path: Specifies the specific resource within the domain or a hierarchy leading to it (e.g., /folder/file.html).
- **Query**: Provides additional parameters that the server can use to identify specific resources or return a certain type of response (e.g., ?key1=value1&key2=value2).
- **Fragment**: Specifies a specific section within the resource (e.g., #section-1).

https://www.example.com:8080/path/to/resource?query=value#fragment