Mastering OAuth 2.0 and OpenID Connect

Session 1 - Introduction to OAuth 2.0 and OpenID Connect

Q : Even though OAuth solves a lot of problems, including not needing to have access to many credentials of others, does a client ID and secret still reduce it all down to weak authN with a single point of failure?

Answer : The client ID is used in the flows and is typically public information (depends a bit on the accessibility of the application). The client secret is indeed a piece of sensitive information that needs to be handled by a confidential client. The client is however well-positioned to handle such a secret securely. One example is keeping the secret (or private key with key-based authentication) in a key vault, such as Hashicorp's vault.

This implies that an attacker needs to compromise the server-side application (the client) to obtain the secret. When the attacker is able to do that, the application is already compromised.

In essence, the client secret is not a single point of failure, but mostly a safeguard to ensure it is a legitimate confidential client interacting with the STS.

Q : Is it not that SSO can be implemented simply via cookies ? SAML etc would make more sense in Federation use cases ?

Answer : OIDC does implement SSO with cookies, since the browser maintains a cookie-based session with the STS. The reason we still need the STS is because cookies are linked to a domain. If Virtual Foodie chooses to offer "Sign in with Google", the cookie your browser has with Google will definitely be used when interacting with Google. However, the cookie for google.com will not be shared with virtualfoodie.com. That's why we have the redirect-based OIDC flow that results in an identity token being sent from Google to Virtual Foodie.

Note that SSO with cookies is often implemented on subdomains (app1.example.com and app2.example.com). However, such domain-wide cookie settings are considered to be insecure and are somewhat frowned upon today.

Q : I assume doing OAuth2 and OIDC to do yourself might be a bad thing to do yourself?

Answer : Yes. I strongly recommend to use an existing implementation, either open source or a commercial product. Writing OAuth 2.0 and OIDC server-side code is extremely tricky (even Auth0 has had major vulnerabilities in their systems).

Q : If Identity Provider is handling user credentials to do authentication then how is STS the answer for the previous question?

Answer : In our terminology, the STS is the Identity Provider/Authorization Server. It is the central service supporting OAuth 2.0 and OIDC flows.

Q : After step 12, API needs to go to STS to validate the token right?

Answer : Yes, absolutely. We talk about the details of doing that in Session 3.

Q : Is there a risk of exposing client id, etc.. in the url?

Answer : The client ID is not considered sensitive information. Without control over redirect URIs, access to a secret or access to the code verifier, the client ID is not very valuable.

Q : I assume the authorization code flow is only valid for MVC (server side rendered) apps and NOT SPAs?

Answer : The Authorization Code flow with client authentication, yes. SPAs can use the Authorization Code flow with PKCE.

Q : The token shown had time to live, but not a reference point and there was no call from the API to the STS to check validity?

Answer : The API is supposed to verify the validity of an access token. How that is done depends on the token type (reference tokens vs self-contained tokens). We discuss this in detail in Session 3.

Q : In the flow you just showed where was Authentication handled?

Answer : User authentication (e.g., username and password) is always handled by the STS. Client applications (e.g., the Virtual Foodie Backend) can consider a user authenticated after processing the identity token from an OIDC flow.

Q : Would the API not talk to the STS to ensure that the access token it received is actually valid? How would the API otherwise know?

Answer : A self-contained token is a signed JWT containing a set of claims. These can be verified independently, but there are a few drawbacks. We discuss this in detail in Session 3.

Q: So who's job is it to verify the "audience"? Is it Resource Server that checks the JWT to do that?

Answer : Yes the API (or resource server in OAuth 2.0 terminology) has to check that the audience of a token contains its own identifier. We discuss this in detail in Session 3.

Q: Was that flow simulator a client side only app (SPA) or was it hosted off of a backend as well? If so, was it just a concession for demonstration purposes to use a SPA?

Answer : It is a client-side only SPA. It keeps data in the browser and should not be used for sensitive confidential clients. There are a few limitations on using a frontend only app, but it also makes it a lot easier to build such a tool. Maybe I'll add a backend component in the future, but not yet.

FYI, I plan on releasing the tool in the future, but I have no idea how long it will take me to bring it to an acceptable quality level to publish.

Q: Regarding the code authorization flow: In Step 12 (or 13) we receive the Refresh and Access Token. Is there anything wrong with storing these tokens in the Frontend (either encrypted or unencrypted in the cookies) so that I don't need to have a persistence for the session management in the backend?

Answer : That is a topic that is a bit out of scope for OAuth 2.0. If you consider session management with encrypted cookies acceptable, you could definitely go this way. I believe that IdentityServer uses a similar pattern to manage its sessions. However, refresh tokens are much more sensitive, so I recommend keeping these in a database. You typically do not need them that often anyway. You could even implement the client parts of the refresh token flow in a separate backend service.

Q: How does the API ensure that the access token was actually sent by the rightful client ?

Answer : Honestly, in most scenarios, there is no such check. In those cases, we call access tokens "bearer tokens" (because the holder or bearer is the one with the authority to use them). When they are stolen, trouble arises. In session 3, we will discuss a few options to lock this further down (e.g., proof of possession).

Q: Just trying to understand why refresh tokens? Why cannot we issue long term access tokens?

Answer : Because access tokens are much more vulnerable to attacks. They are sent to APIs, which exposes them to the network and external parties. Refresh tokens are a bit easier to control, as they are only exchanged between a client and an STS. The semantics are debatable here, but a best practice is to use short-lived access tokens in combination with a refresh token.

Additionally, if an access token is a self-contained token, the STS has no control over that token. That would mean that a user loses the option to revoke an existing token for a long period.

Q: Is the primary reason for the refresh token flow to allow the user to revoke access to their information?

Answer : Yes, that is definitely part of the idea behind refresh tokens. Additionally, refresh tokens are less exposed, as they are only exchanged between the client and the STS. Access tokens have much more exposure, since they are also sent to APIs.

Q: How is the refresh token protected in the backend for this long time?

Answer : Backend applications can encrypt the refresh token when storing it. ADditionally, using the refresh token requires client authentication with either a secret or a private key.

Q: Does a refresh token ever expire?

Answer : That is up to the STS to decide. An STS may give it an expiration date, or an eternal lifetime. To give you a concrete example, I use Buffer to schedule social media posts on Twitter and LinkedIn. The TWitter connection has been working for a couple of years now, while the LinkedIn connection fails every x number of weeks. That's when Buffer tells me to log in and reconnect (which re-executes a flow with the user involved).

Q: With Long-term access, this could be quite long if no real expiry is set on the refresh token, how long is typically found acceptable in a high risk scenario for the frontend channel for native applications?

Answer : It all depends on how often you want the user involved. When a refresh token expires, there is no recovery path that does not involve the user running the flow again.

In session 2, we discuss refresh token rotation in detail, which helps detect potential abuse of refresh tokens.

Q: In this scenario, does the backend store access and refresh tokens in a database somewhere? If so, do you have reference material or quick guidelines as the best way to do that?

Answer : Yes, a backend client will typically keep track of tokens. Refresh tokens are long-lived, so they belong in a database. Encrypt them before storing them. Preferably, use a separate service to encapsulate the handling of refresh tokens and isolate that service from the main application.

For access tokens, it depends on their lifetime. If they are short-lived, you can keep them in the session object. Otherwise, store them in the DB, but again, encrypt them to properly protect them against heft.

This story from Buffer explains what happens if you neglect that:
https://open.buffer.com/buffer-has-been-hacked-here-is-whats-going-on/

Q: Also, can these access and refresh tokens be invalidated on demand if they are controlled by STS? Can these tokens be protected by more secure means?

Answer : Yes, if an STS controls the tokens, the user can typically revoke them. For refresh tokens, this is always possible. For access tokens, it depends on the token type. We discuss access token types in detail in Session 3.
In session 3, we also discuss strategies to tie tokens to specific clients, which prevents abuse when such a token is stolen.

Q: You mentioned earlier that the "state" parameter is less relevant nowadays, can you go into more detail why that is? Are there scenarios where it is still needed? Or can we generally ignore it?

Answer : The state parameter was relevant to protect against CSRF in OAuth 2.0 flows. The details are a bit technical, but such an attack basically transfers a half-completed flow from one browser to another. The state parameter allowed a client to detect that.

Today, the recommendation is to use PKCE. Because of PKCE, this CSRF attack is no longer possible (there is already state involved, which is the code verifier now). As a result, the state parameter can be used for its original purpose again, which is to track client-specific state during the redirects.

More details can be found here:
https://tools.ietf.org/html/draft-ietf-oauth-security-topics-15#page-25

Q: Can we use sub claims in access token as a user identity and use for permission or authorize?

Answer : Yes, the sub claim in an access token can be used for user-specific authorization decisions. More details on making such decisions in Session 3.

Q: Does the client check if JWT is signed by the right auth server? In that case it would need the keys to verify, correct?

Answer : Yes and no. When the information is received from the STS in the body of the code exchange response, it already comes from a trusted source. There is no explicit need to verify the signature again. In other flows (e.g., an Implicit or Hybrid flow) this is still required.

Note that the identity token can also be sent back to the STS in a logout flow, which would require the STS to validate its validity. That's when the signature is checked.

To verify a signature, a client would indeed need the public keys. These are published by the STS and are part of the discovery file
(e.g. https://sts.restograde.com/.well-known/openid-configuration)

Q: Can I use OIDC with the password grant type to just offload authentication, but with my app's own login screen?

Answer : Yes and no. Yes you can do that, but it is typically not recommended. The only use case I see for such a flow is when you have 1 application using an STS as a "user database". The moment you have multiple apps, you should use the redirect-based flow.

Q: How can an API forward the access token it gets to contact other APIs transitively in order to fulfil a request?

Answer : That's a bit of a challenge. In principle, access tokens are not supposed to be used in that fashion. Access tokens are used by a client and consumed by an API. Forwarding an access token is also known as "poor man's delegation". The proper solution would be to implement a Token Exchange mechanism (https://tools.ietf.org/html/rfc8693).

Note that implementing a Token Exchange is an advanced use case. In-between alternatives can be deploying mTLS between different APIs to restrict communication.

Q: What do you mean by Evil Chef?

Answer : The evil chef is our attacker. In this case, the attacker could trick the user into installing a malicious app, which receives the redirect URI. The threat model is discussed here (https://tools.ietf.org/html/rfc8252), but the practicalities on how such an attack is conducted in practice are somewhat unclear. Nonetheless, PKCE is a recommended security feature (it also prevents code injection attacks).

Q: Is it okay to use password grant type or is Authorization code flow with PKCE still necessary?

Answer : If you want to use OAuth 2.0/OIDC, a password grant is probably not OK. I know that this is a common pattern, but the recommendation today is to use a redirect-based flow. The only scenario where I see the possibility of using a password grant is when you use the STS as a "user management service". If you have one app with one STS, there is no point in using OAuth 2.0 or OIDC.

Q: Why would you ever want to use the Authorization Code flow without PKCE? It seems as if it supports the MVC backend app as well (is a client secret)?
Why is PKCE not the default way, instead of client secret?
Why not always use PKCE for the authorization code flow, even if we are not in a mobile client environment, and never use client credentials (which need to be stored securely)?
Can one use PKCE on a non-mobile?
Answer : We talked about PKCE a lot during this session. In general, the Authorization Code flow without PKCE is the original flow. With the addition of PKCE, there is little reason to not use it. The recommendation is to always use the Authorization Code flow with PKCE for redirect-based flows that involve the user.

Moving away from client secrets would not make much sense. They are well supported and offer an additional layer of security for confidential clients.

Q: Why is the code challenge and code verifier better than the “state” cookie for web applications?

Answer : First, state is not a cookie, but a parameter in the URIs of the flows. Second, state was not intended to be used as a CSRF defense, but assumed that role in order to fix an unpredicted problem with the flows. Furthermore, the state needs to be checked by the client, which is often neglected in actual implementations. PKCE solves this by design (along with other issues, such as code injection) and is enforced by the STS.

Q: I'm curious about the mobile flow (native app) that is made by the same company as the backend. Like Restograde creates a backend and it makes an Android app. In the Android app, to login/signup it doesn't use a webview. It just uses the native components to hit restograde.com/api/v1/login How would that work?

Answer : That would be a password flow, which is not recommended. The problem quickly becomes how users should know when to enter their Restograde credentials. What if the "MaliciousFood app" asks for Restograde credentials. Is that legitimate, or not? With a redirect flow, credentials always go to the STS authentication page. In my case, my password manager would handle that out of the box.

You can use a password flow when there is only one app using the STS. In that case, the STS becomes a "user management service". I believe that Amazon's Cognito supports various of these use cases.

Q: Is the code challenge used several times, e.g. in refresh scenarios?

Answer : No. The code challenge & verifier are only used for the code exchange (hence the name).

The refresh token flow has no redirects, so no need to add a new challenge and verifier. Re-using the old verifier also would not make much sense. The purpose of the code verifier is to protect against code theft (among other things). If you would require the same for refresh tokens, it would be to protect against refresh token theft. But to use it later, the app needs to store the code verifier. As a result, when the attacker can steal the refresh token, they can also steal the code verifier.

Q: What about hybrid apps and webviews?

Answer : Hybrid apps have access to secure storage (through the native APIs). Because of that, they follow the rules for native applications. They should still use a system browser to run the redirects, instead of running them in their own webview.

Q: What encryption techniques do you recommend for storing tokens in a secure storage on a mobile device or on a backend?

In general, I recommend either the use of an encrypted DB service (https://www.cossacklabs.com/acra/), or the use of a high-level crypto library (libSodium, Tink). Make sure you know how to handle encryption properly, as there are some pitfalls.

Answer : On backend systems, you can also offload this responsibility to a vault system (e.g., Hashicorp Vault). On mobile, you would generate application-specific keys and store them in the OS's keychain or keystore. For example, the Android Cipher API makes a full abstraction of using the keystore for encryption and decryption.

Q: When should we use the “bearer” token? Which use cases? What other types are there and when should they be used?

Answer : This question is a bit open ended to answer here. Consider joining us in session 3 for more details on protecting access tokens.

Q:When do you recommend using an access_token vs and id_token in a frontend web SPA scenario?

Answer : The identity token is intended for the client and contains information about the user's authentication. The access token is intended for the API and contains information about the client's authority on behalf of the user. They serve two distinct purposes.

Q: For security purposes, do you recommend having the backend app playing the “client” role and use a cookie-based session between SPA and backend, to avoid having the access token in the browser

Answer : Yes, that is a valid pattern. We call that a Backend-for-Frontend (BFF). I will discuss this pattern in detail in Session 2.

Q: What alternative do I have when I don't have a user interaction but need to authenticate their username + password?

Answer : Usually, if you have such a need, there is a problem with the application architecture. Clients should not have access to credentials. You can use a password flow when there is only one app using the STS. In that case, the STS becomes a "user management service". I believe that Amazon's Cognito supports various of these use cases.

Q: IIRC, token binding specs were drafted and expired.

Answer : Token binding in the sense of this spec is indeed not supported by most systems. However, many STS support binding tokens to the TLS certificate used by the client application. This allows binding tokens to a certificate for confidential clients and native clients. The correct term in the specs would be "sender-constrained tokens".

Browsers are a bit more tricky though (no binding available), as we will discuss in Session 2.

Q: What SDK's did you use in your flow simulation app?

Answer : None, as I need access to the dirty details of the URI and the redirect. It's custom code that I will publish once it reaches a level of quality that I deem acceptable.

Q: Is session 3 independent of session 2? Can we take session 3, if interested?

Answer : Yes, you can buy a ticket for a single session, or purchase the session 2 + 3 bundle. Up to you. For both sessions, recordings will be made available to registered users.

Q: Will you cover token binding and JWE in any of the sessions?

Answer : I will discuss sender-constrained tokens (binding to a TLS certificate). JWE is probably out of scope.