My understanding is that the use of a **refresh token** enable short lived access token and therefore limits the vulnerability of those access tokens. Great so far. Once an access token expires, you somehow use the refresh token to get a new access token.

The OAuth 2.0 Authorization Framework

Access tokens CANNOT be invalidated. Once issued they are forever valid until the expiration is reached or the token is tampered with. Revoking refresh tokens does not influence access tokens.

Access token are ***bearer* tokens**. Meaning **no other identification is required, and the access token is all that is needed to impersonate you**. Because of this, they should always remain short lived. On the other hand **refresh tokens are not *bearer* tokens**. When you send a refresh token to YouTube to get a new access token, you also have to send a **client\_id** and **client\_secret**. Because of this, refresh token can remain longer lived because it is much less likely that both the refresh token and the client\_secret would be compromised.

The idea of refresh tokens is that if an access token is compromised, because it is short-lived, the attacker has a limited window in which to abuse it.

**Refresh tokens**, if compromised, are useless because the attacker requires the client id and secret in addition to the refresh token in order to gain an access token.

**Having said that**, because every call to both the authorization server and the resource server is done over **SSL** - including the original **client id and secret** when they request the access/refresh tokens - **I am unsure as to how the access token is any more "compromisable" than the long-lived refresh token and clientid/secret combination.**

This of course is different to implementations where you don't control both the authorization and resource servers.

Refresh tokens... mitigates the risk of a long-lived access\_token leaking (query param in a log file on an insecure resource server, beta or poorly coded resource server app, JS SDK client on a non https site that puts the access\_token in a cookie, etc)

the **client ID** and **secret** are credentials for the **OAuth client**, not the user. When talking about **OAuth the "client" is usually a server** (for example the stackoverflow web server) which interfaces with an **authorization** or **resource API server** (for example the facebook auth provider). The **user's credentials** are only passed between the user and the OAuth API server, and never known to the client. The **client secret** is only passed from the client to the OAuth API server, and is never known to the user.

**How the system with long-lived access tokens should work**

The server allows the Client to get access to User's data within a pre-defined set of scopes by issuing a token. As we want to keep the token revocable, we must store in the database the token along with the flag "revoked" being set or unset (otherwise, how would you do that with self-contained token?) Database can contain as much as len(users) x len(registered clients) x len(scopes combination) records. Every API request then must hit the database. Although it's quite trivial to make queries to such database performing O(1), the single point of failure itself can have negative impact on the scalability and performance of the system.

**How the system with long-lived refresh token and short-lived access token should work**

Here we issue two keys: random refresh token with the corresponding record in the database, and signed self-contained access token, containing among others the expiration timestamp field.

**As the access token is self-contained, we don't have to hit the database at all to check its validity. All we have to do is to decode the token and to validate the signature and the timestamp.**

Nonetheless, we still have to keep the database of refresh tokens, but the number of requests to this database is generally defined by the lifespan of the access token (the longer the lifespan, the lower the access rate).

In order to revoke the access of Client from a particular User, we should mark the corresponding refresh token as "revoked" (or remove it completely) and stop issuing new access tokens. It's obvious though that there is a window during which the refresh token has been revoked, but its access token may still be valid.

**Tradeoffs**

Refresh tokens partially eliminate the SPoF (Single Point of Failure) of Access Token database, yet they have some obvious drawbacks.

1. The "window". A timeframe between events "user revokes the access" and "access is guaranteed to be revoked".
2. The complication of the Client logic.

**without** refresh token

* + send API request with access token
  + if access token is invalid, fail and ask user to re-authenticate

**with** refresh token

* + send API request with access token
  + If access token is invalid, try to update it using refresh token
  + if refresh request passes, update the access token and re-send the initial API request
  + If refresh request fails, ask user to re-authenticate

I hope this answer does make sense and helps somebody to make more thoughtful decision. I'd like to note also that some well-known OAuth2 providers, including github and foursquare adopt protocol without refresh tokens, and seem happy with that.

 the short version of my post is, if you save the access token in the database, you hit the database on every request to your API (which may or may not be a problem in your particular case). If you save refresh tokens and keep access tokens "self-contained", you hit the database only when the client decides to refresh the access token.

To clear up some confusion you have to understand the roles of the [*client secret*](https://salesforce.stackexchange.com/questions/14009/whats-the-benefit-of-the-client-secret-in-oauth2) and the *user password*, which are very different.

**The *client*** is an app/website/program/..., backed by a server, that wants **to *authenticate* a *user*** by using a third-party authentication service. The **client secret** is a (random) string that is known to **both this client and the authentication server**. Using this secret the client can identify itself with the authentication server, receiving *authorization* to request access tokens.

**To get the initial access token and refresh token**, what is required is:

* **The user ID**
* **The user password**
* **The client ID**
* **The client secret**

To get a **refreshed access token** however the *client* uses the following information:

* **The client ID**
* **The client secret**
* **The refresh token**

This clearly shows the difference: when refreshing, the client receives authorization to refresh access tokens by using its client secret, and can thus re-authenticate the user using the refresh token *instead* of the user ID + password. **This effectively prevents the user from having to re-enter his/her password.**

This also shows that losing a refresh token is no problem because the client ID and secret are not known. It also shows that keeping the client ID and client secret secret is *vital*.

A server would verify an access token based on credentials and signing of (typically) a JWT.

An access token leaking is bad, but once it expires it is no longer useful to an attacker. A refresh token leaking is far worse, but presumably it is less likely. (I think there is room to question whether the likelihood of a refresh token leaking is much lower than that of an access token leaking, but that’s the idea.)

Point is that the access token is added to every request you make, whereas a refresh token is only used during the refresh flow So less chance of a MITM seeing the token

Frequency helps an attacker. [Heartbleed](https://en.wikipedia.org/wiki/Heartbleed)-like potential security flaws in SSL, potential security flaws in the client, and potential security flaws in the server all make leaking possible.

In addition, if the authorization server is separate from the application server processing other client requests then that application server will never see refresh tokens. It will only see access tokens that will not live for much longer.

JSON Web Token (JWT)

JWT defines a way in which certain common information pertaining to the process of authentication/authorization may be **represented**. As the name implies, the data format is **JSON**. JWTs carry certain **common fields** such as **subject**, **issuer**, **expiration time**, etc. JWTs become really useful when combined with other specs such as [JSON Web Signature (JWS)](https://tools.ietf.org/html/rfc7515) and [JSON Web Encryption (JWE)](https://tools.ietf.org/html/rfc7516). Together these specs provide not only all the information usually needed for an authorization token, but also a means to **validate the content** of the token so that it cannot be tampered with (JWS) and a way to **encrypt information** so that it remains **opaque** to the client (JWE). The simplicity of the data format (and its other virtues) have helped JWTs become one of the most common types of tokens.