Difference between Two legged and Three legged Authentication in OAuth

<http://stackoverflow.com/questions/5593348/difference-between-oauth-2-0-two-legged-and-three-legged-implementation>

First, the legs refer to the roles involved. A typical OAuth flow involves three parties: the end-user (or resource owner), the client (the third-party application), and the server (or authorization server). So a 3-legged flow involves all three.

The term 2-legged is used to describe an OAuth-authenticated request without the end-user involved. Basically, it is a simple client-server authenticated request in which the client credentials (identifier and secret) are used to calculate a request signature instead of sending the secret in the clear.

Implementation wise, 2-legged request are exactly the same but don't include an access token or access token secret. These two values are basically empty strings.

<http://stackoverflow.com/questions/13369516/why-is-there-3-legged-oauth2-when-2-legged-works-so-well>

Three legged does not imply a certain type of app as in "browser based". Three legged means that an application acts on the direct behalf of a user. In the three legged scenarios there is

1. an application (consumer),
2. a user (resource owner) and
3. an API (service provider).

In two legged scenarios there is no concept of a user. Typically this has to do with application-to-application solutions. There the application (consumer) acts on behalf of itself. So in two legged OAuth, there is:

1. an application (consumer),
2. an API (service provider)

The difference is simply that there is no need of a user authorisation step in the 2-legged approach.

<http://blog.nerdbank.net/2011/06/what-is-2-legged-oauth.html>

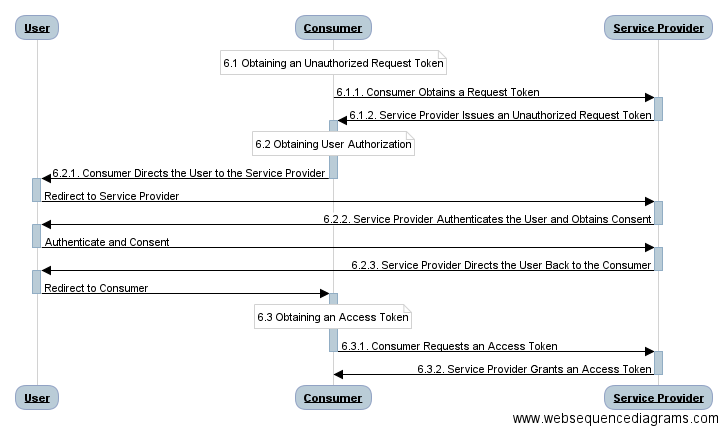
### [**What is 2-legged OAuth?**](http://blog.nerdbank.net/2011/06/what-is-2-legged-oauth.html)

Although there is an [official spec for OAuth 1.0](http://trac.tools.ietf.org/html/rfc5849), the spec only outlines what the community refers to as "3-legged OAuth".  An alternative form of OAuth is loosely referred to as "2-legged OAuth", and there are far too many variants of this and not a single finalized spec to conform to.  As a result, there are various ways and forms to achieve what people, correctly or incorrectly, refer to as 2-legged OAuth.  In this post, I will attempt to clarify what (at least in my mind) 2-legged OAuth really means.

I will not delve into the gritty details of the spec, but I will outline the flows and explain a bit.

### 3-legged OAuth

First, let's start with the spec's description of the OAuth flow (3-legged).  Here is an illustration:

[](http://bit.ly/mln2yP?r=bb)

You can clearly see the 3 "legs" of this flow:

1. The consumer (the application that wants to access data) creates an OAuth session by obtaining an *un*authorized request token from the Service Provider (the host of the data).
2. The consumer directs the user to the service provider, with the unauthorized request token in hand, so the service provider can ask the user to release the protected data to the consumer.  The user logs in as necessary, and grants the requested permission.  The service provider redirects the user back to the consumer.
3. The same request token the consumer held before is now *authorized*.  The consumer submits it to the service provider in a direct web request to exchange it for an access token.

The access token may now be used by the consumer to submit requests to the service provider that will access the user's private data.

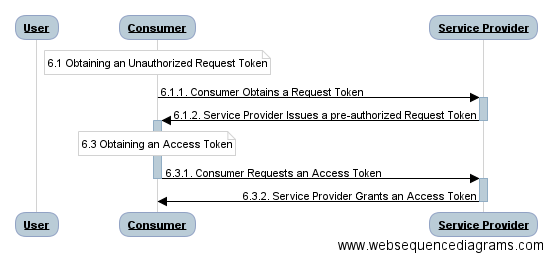
By the way: notably absent from this 3-legged flow is the consumer obtaining its own "consumer key" and "consumer secret".  The process of obtaining these is explicitly outside the scope of the OAuth spec altogether, and is therefore defined by each service provider.  This process is only completed once per application, usually by the application developer him/herself, and is *not* considered one of the legs of OAuth.

### 2-legged OAuth

In 2-legged OAuth, the consumer tends to be installed on the user's machine, or is perhaps a widget embedded in a web page.  The key scenario difference as you step into 2-legged OAuth is that the consumer is*not* requesting access to any *user* data.  Instead, it is merely establishing an account with the service provider with no previous data in it at all, which it can subsequently use to store and later retrieve data.

However, since this particular installation or widget has its own "space" on the service provider to store data, the consumer can obtain user data directly from the user him/herself, and send that to the service provider, and later retrieve it. So we see that although 2-legged OAuth doesn't *start* with any user data, it may end up with user data that the consumer itself puts there.

Because no pre-existing user data is ever shared with the consumer, there is no need for the service provider to obtain authorization from the user.  Therefore the second of the three legs can be entirely skipped.  The modified flow is illustrated here:

[](http://bit.ly/lezeHC?r=bb)

Note that the "6.2" section is absent, and the User role has absolutely no interaction, leaving only two legs.  Section 6.1 is also slightly different: instead of the service provider issuing the consumer with an *un*authorized request token, this request token is *pre-*authorized.  The consumer can then immediately exchange it for an access token and discard the request token.

At this point you may be saying to yourself: "That's silly -- why doesn't the service provider just issue the access token instead of a request token, and save another leg?"  Well, it could, but then that wouldn't be OAuth any more.  You could make the case that the above 2-legged flow is still OAuth because all the existing APIs and flows work as spec'd... the skipped legs doesn't alter the remaining legs so it requires little or no changes in a standard OAuth implementation to support 2-legged as well.

### What's *not* 2-legged OAuth

I've recently seen a list of links that all claim to describe "2 legged OAuth", but are nothing like what I just described.  Instead, what they describe is what I would refer to as *0*legged OAuth.  That's right: *none* of the 3 legs are preserved in the flow they describe.  Here is what they describe, in a nutshell:

1. The consumer's developer obtains a consumer key and secret.
2. (skips the entire OAuth 3-legged flow here)
3. Consumer accesses protected resources by submitting OAuth signed requests for resources using its consumer key, an empty access token, and signs the request with the consumer secret and an empty access token secret.

Remember from the 3-legged discussion above, that #1 in this list is not considered one of the three legs: it's a one-time step taken by the app developer.  Nor is #3 in this list considered one of the 3 legs, as it is simply the actual request for the protected resource, and is alone repeated at each request.

Also consider that whereas genuine 2-legged OAuth can end up with the consumer storing *user* data in a private consumer-user data store at the service provider, this *0*-legged OAuth cannot end up with user data in its service provider account, because all instances of the consumer share exactly the same account, and that would mix all the users' data together.

This 0-legged OAuth should look remarkably similar to username/password authentication, and in flow that's exactly what it is, where the consumer app is the owner of the username/password instead of the user.  Why would someone go to the trouble of using/inventing 0-legged OAuth instead of just using username/password then?  Two reasons: 1) it may be easier to leverage existing OAuth-supporting code than adding support for username/password, and 2) OAuth provides signatures that mitigates against replay attacks that HTTP basic authentication can be vulnerable to.

### What's else is *not* 2-legged OAuth

Twitter has an interesting special case of OAuth as well, which also entirely skips the 3-legged flow, yet it is still *per user*.  The user navigates his/her browser to twitter.com, and obtains an access token directly, with no request token preliminary step.  The user copies and pastes (manually!) this access token into the consumer app, which then simply access the user's private data without ever going through any of the 3-legged flow.  This also has been (incorrectly) referred to by some as "2 legged OAuth", but I hope this discussion can help you see this is a misconception.

### Summary

Let's summarize terms and use cases then:

* 3-legged OAuth: includes user authorization through a web browser for the consumer to access the end user's previously established private user data with a service provider.
* 2-legged OAuth: skips user authorization, but still has non-empty request and access tokens.  Consumers gain access to an empty account that they may then fill with user data the consumer obtains directly from the user.
* 0-legged OAuth: skips all three legs by having consumers simply access their own protected resources at the service provider using OAuth signatures based on a private consumer key and secret.  Analogous to username/password for the consumer app.
* Twitter access token option: also skips all three legs, but still provides access to a user's pre-existing data via a non-empty, manually obtained access token.

Please comment with your thoughts, and indicate whether you agree or disagree.  With so many application-specific half-specs out there claiming to describe "2-legged OAuth", I expect some comments to this post telling me I'm wrong.  That's fine.  I'd be interested in hearing you make a logical case for how you reduce OAuth 3-legs to 2-legs then.  And to reiterate, I'm not invalidating the use cases of what these 0-legged flows are doing, I'm merely suggesting we call it like it is: "*0-legged OAuth*".