



Implementing 2-Legged OAuth in Javascript (and CloudTest)



Introduction

If you're reading this you are probably looking for information on how to implement 2-Legged OAuth in Javascript. I recently had to implement 2-legged OAuth into a CloudTest performance test for one of our customers. Because 2-legged OAuth is not part of the official OAuth spec yet (as of 6/15/2011) there is relatively little information out there about how to make this all work. Where there is information unfortunately it doesn't universally work for all implementations since there isn't a specification for it. I hope this saves you some time... it definitely would have helped me out. You will need a working knowledge of Javascript to find the implementation details in this article useful. Without an understanding of Javascript you may find just the general OAuth overview interesting.

High Level OAuth Overview

OAuth is a way for applications to authenticate with one-another. In essence a client application encrypts a string of values and passes that encrypted string, along with the values it used to encrypt it (except one, your secret key), to the server. The server then uses the values you sent across to look up your secret key and attempt to generate the same encrypted string you did. The server then compares the two encrypted strings together. If they match, it's a success. If not, it's a failure.

The difference between 3-Legged OAuth and 2-Legged OAuth is that in the 3-Legged variant, the client first passes some credentials to the server and gets an access token back if authentication is successful. Then this token is passed along in subsequent requests. This is commonly called 'the dance' in OAuth developer circles. When you authenticate with Netflix through various platforms (AppleTV, iPhone, Netflix.com), you do a 3-Legged OAuth dance. This allows for users, applications, and authentication to be abstracted out into separate tiers.

Some other types of applications may be better suited for the authentication and message passing to happen in 1 request and 1 request only. This is where 2-legged comes in. In 2-legged OAuth you pass the encrypted string, the values used for encryption, and the message payload in 1 GET or POST. If it is rejected, the message fails. If it's accepted then the message is processed. This particular app that I was working on testing was a central logging system. Every message was a log event. There was no time (or functional need) for a three-way handshake in this app and also no notion of a maintained state. 2-Legged OAuth cuts out the middleman. If authentication is successful the message is processed, no dancing around.

Diving Deeper

OAuth messaging can be done in two ways: in an authorization header or right on the query string. I'm going to focus on the query string method here because I think it's easier to debug by swapping values around in a URL instead of trying to mess with request headers. That's just my preference though.

The process for encrypting your signature string sounds easy but in reality it's much more challenging to get right. In short, the process goes like this:

- 1) Build a long string called a 'signature blob' made up of sets of key/value pairs
- 2) Run that signature blob through a standard encryption function, using your 'consumer secret key' and end up with a 'signature'
- 3) URLEncode the signature (it's getting passed on the query string so it needs encoded)
- 4) Pass all of the key/value pairs and the signature along in the query string (in alphabetical order... very important)
- 5) Pass or fail ;)

The challenge is that if the signature blob you build is off by even one character before you encrypt it then it's not going to match what the server generates when it creates a signature to compare to yours. All that matters is what the server gets when it creates it's sigblob and signature. Your sigblob and signature HAS to match what is generated on the server side.

A HUGE help is if you can have a developer on the server side give you a sample sigblob string and the signature that was created from encrypting that sigblob, this will help you immensely. If you have this, you can construct your sigblob to look exactly like what the server side will process. You can also compare the signature you get from encrypting it and make sure those match.

Implementation

First off – here is the official OAuth documentation: <http://oauth.net/core/1.0/>

There are really great explanations and definitions in there for the things I'm about to cover if you still have questions.

Here is an example of a 2 legged OAuth request using the query string instead of an auth header (against a fictitious site... obviously):

```
http://www.blahblahblah.com/made-up-uri/RESTFUL-WEBSERVICE-  
CALL?oauth_consumer_key=905324be-d7e0-4f1f-bff7-9892c6c37a73&oauth_nonce=  
rMeXln&oauth_signature=hR8gU95DBqHAn4v0qqPGrn%2B%2BJAQ%3D&oauth_sig  
nature_method=HMAC-SHA1&oauth_timestamp=1302308307&oauth_token=&oauth_  
version=1.0
```

Note that all of the query string parameters are in alphabetical order. This is important because doing this is mandatory... it's required as part of the OAuth specification.

Let's break down this long URL:

1) <http://www.blahblahblah.com/made-up-uri/RESTFUL-WEBSERVICE-CALL>

This is your standard run of the mill RESTful web service URI. You would be posting something to this URI most likely. OAuth will be on the query string, and the payload will be in the POST body.

2) ?

A URI is always separated from the query string by a question mark... this is standard HTTP formatting and you probably already know this.

3) `oauth_consumer_key=905324be-d7e0-4f1f-bff7-9892c6c37a73`

The `consumer_key` is given to you by the service provider. This is basically your 'user ID' for making authentication requests to the server. You pass it along with your request and the server will use it to look up your `consumer_secret_key` and use that for encrypting the whole sigblob. You will also be given the `consumer_secret_key` by the service provider because you have to use it when encrypting your signature string. You don't, however, pass this secret key along in the URL. That would be very insecure and defeat the purpose ;) Instead you pass your `consumer_key` along and the server will look you up and use your secret encryption key from the database.

4) `oauth_nonce=rMeXIn`

A nonce is basically a random alphanumeric string. It's an id tag for the request. It's required but there's not much to it... so I won't spend much time on it. The OAuth class provided a bit later has a function for randomly generating a nonce value.

5) `oauth_signature=hR8gU95DBqHAN4v0qqPGrn%2B%2BJAQ%3D`

This is the result of encrypting the sigblob (signature blob.. all of the values here formed into a string and then encrypted generate the signature) with your `consumer_secret_key`. Notice some values are URLEncoded (%2B, %3D, etc.)

6) `oauth_signature_method=HMAC-SHA1`

This is the encryption method used in the particular OAuth implementation your server is using. The most common method is HMAC-SHA1. Make sure you get the right signature method from the service endpoint you're trying to call.

7) `oauth_timestamp=1302308307`

Timestamp from when the signature was encrypted. Another security measure that ensures stale requests cannot be processed.

8) `oauth_token=`

This is required on the query string but since it isn't used in 2-legged OAuth it is left blank. If this were 3-legged OAuth, there would have been a preceding request that would have resulted in you getting an `oauth_token` to use in future calls.

9) `oauth_version=1.0`

The version of oauth being used on the server side. As of the time this post was written 1.0 is the only version.

Resources You'll Need First

Let's get down to building some sigblobs (I love this word). There are three Javascript snippets that you will need to grab first.

1) This is a javascript implementation of the sha1 hash algorithm. You'll need it for encrypting the sigblob.

<http://pajhome.org.uk/crypt/md5/sha1.js>

2) This is an Oauth class written in Javascript that provides various utilities like creating a nonce value, etc.

<http://code.google.com/p/oauth/source/browse/code/javascript/oauth.js?r=1136>

3) This is a nice function called makeSignedRequest() that implements the Oauth class above. I found it useful for piecing everything together.

<http://paul.donnelly.org/2008/10/31/2-legged-oauth-javascript-function-for-yql/>

Making it All Work

Below is the code that I used to implement everything we've talked about. Extra comments abound so that it's pretty clear what's going on. In order for this to work you need to have included the 3 blocks of code referenced in the previous section.

```
/*
   Implementation of the SHA-1 encryption code and the
   OAuth code snippets referenced in the blog post at
   www.SOASTA.com

   The CK, CKS, and URL in the makeSignedRequest method
   call below are fictitious.
*/

var signedURL =
makeSignedRequest("consumer_key_123456789", "consumerkeys
ecret/987654321", "http://www.blahblahblah.com/made-up-uri/RESTFUL-WEBSERVICE-CALL

// Slice the query string off of the end of the returned
URL

var fullLocation = signedURL;
var beginning = fullLocation.indexOf("?");
var queryString = fullLocation.substring(beginning +
1, fullLocation.length);

/***/ Begin splitting out each OAuth param out of the
```

```

query string that was generated above   ***

//Consumer Key

var beginning =
fullLocation.indexOf("oauth_consumer_key=");
var consumer_key = fullLocation.substring(beginning,
fullLocation.length);

//Nonce

var beginning = fullLocation.indexOf("oauth_nonce=");
var end = fullLocation.indexOf("&oauth_signature");
var nonce = fullLocation.substring(beginning + 12, end);

//Sig method

var beginning =
fullLocation.indexOf("oauth_signature_method=");
var end = fullLocation.indexOf("&oauth_timestamp");
var signature_method = fullLocation.substring(beginning,
end);

//Timestamp

var beginning = fullLocation.indexOf("oauth_timestamp=");
var end = fullLocation.indexOf("&oauth_version");
var timestamp = fullLocation.substring(beginning + 16,
end);

//Version

var beginning = fullLocation.indexOf("oauth_version=");
var end = fullLocation.indexOf("&oauth_signature=");
var version = fullLocation.substring(beginning, end);

// Generate the signature string and encrypt it to be
passed along in the final URL

/* The signature blob

    Things to note:
    1) There is a required & between the URL and the oauth
parameters
    2) The ordering is significant - OAuth spec requires
that the oauth parameters are in alphabetical order
    3) Consumer key is hard coded should probably
parameterized later
    4) oauth_token is on the URL but there is no value

    The real secret to making this work is to know exactly
what parameters and order of parameters that the server
expects. This could be different depending on
implementation but I've found that it's usually close
to this.
*/
    
```

```
var sigBlob =
"POST&http%3A%2F%2Fwww.blahblahblah.com%2Fmade-up-uri%2FRESTFUL-WEBSERVICE-CALL&oauth_consumer_key%
consumer_key_123456789%26oauth_nonce%3D" + nonce +
"%26oauth_signature_method%3DHMAC-SHA1%26oauth_timestamp%
3D" + timestamp +
"%26oauth_token%3D%26oauth_version%3D1.0";

// The consumer key secret, parameter 1 in the function
call below, is URLEncoded. Should use a URLEncode
function on the real key but it was hard coded it in a
time crunch (note the %2F which is a '/').
// Use the b64 sha1 function from the code included above
to sign the sigBlob and generate oauth_signature

signature =
b64_hmac_sha1("consumerkeysecret%2F987654321", sigBlob);

//URLEncode the resulting signature - since it's passed
along on the querystring if it's not URLEncoded then the
signatures wont match between client/server

encodedSignature = encodeURIComponent(signature);

//Generate the final query string to send along in the
message request

var reorderedOAuthQueryString = consumer_key +
"&oauth_nonce=" + nonce + "&oauth_signature=" +
encodedSignature + "&" + signature_method + "&" +
"oauth_timestamp=" + timestamp + "&" + "oauth_token=&" +
version;
```

More Resources

Official OAuth spec.
<http://oauth.net/core/1.0/>

For more information about
SOASTA, please visit
<http://www.soasta.com>.

©SOASTA, Inc. 2010. SOASTA,
SOASTA CloudTest, and SOASTA
CloudTest On-Demand, are
trademarks or registered trademarks of
SOASTA, Inc. All other trademarks are
the property of their respective owners.

About SOASTA

SOASTA's CloudTest products and services leverage the power of cloud computing to quickly and affordably test consumer-facing Web and mobile applications at scale, providing customers with the confidence that their business-critical applications will be reliable and scalable, even under extreme loads. SOASTA's customers include many of today's most successful brands, including American Girl, Chegg, Gilt Groupe, Hallmark, Intuit, Microsoft and Netflix.