A Primer to the OAuth Protocol

Adrian Hannah

Abstract

OAuth uses digital signatures rather than the “Basic” authentication method used by the HTTP protocol.

During the past several decades, Web pages have changed from being static, mostly informational tools to full-blown applications. Coinciding with this development, Web developers have created interfaces to their Web applications so that other developers could develop applications to work with the Web application. For instance, think of any application on your phone for a Web service. This is possible only because of the application programming interface (API) constructed by the Web service's developers.

An API allows developers to give others access to certain functionality of their service without losing control of their service or how it behaves. *With the development of these APIs arose the issue of user authentication and security.* Every time you want to do something with the service, you have to send your user credentials (typically a user ID and password). This exposes the user to interested parties and makes the authentication untrustworthy. The application used by the user also could store the password and allow another application or person access to the user's account.

Enter OAuth.

OAuth is intended to be a simple, secure way to authenticate users without exposing their secret credentials to anyone who shouldn't have access to them. It was started in November 2006 by Blaine Cook, who was working on an OpenID implementation for Twitter. While working on it, Blaine realized that the Twitter API couldn't deal with a user who had authenticated with an OpenID. He got in touch with Chris Messina in order to find a way to use OpenID with the Twitter API. After several conversations with a few other people later, OAuth was born. In December of that year, OAuth Core 1.0 was finalized.

10965s1.qrk

Since August 31, 2010, all third-party Twitter applications are required to use

OAuth.

You can think of OAuth like an ATM card. Your bank account (the Web service) has a load of services associated with it, and you can use all of them, provided you put your card in the ATM and enter your PIN. Ultimately, anyone who has your card and PIN has full access to your account and can use all those neat services to do whatever he or she wants to your account. However, you can use your card as a credit card as well, and in that case, replace your knowledge of the PIN with a signature. In this capacity, the cardholder can do only very limited transactions, namely make charges against the balance of the account.

If someone were to try to use your signature to charge something to your account without your card, it wouldn't work. If you had the card but not the signature, the same result would occur (theoretically). OAuth works in a similar manner. If an application has your signature, it can make API calls on your behalf to the Web service, but that signature works only with that application. Allowing one party to access someone else's resources on his or her behalf is the core of the OAuth protocol.

An Example of OAuth

Consider user Jane, a member of a photo-sharing site, photosharingexample.com (Service Provider), where she keeps all her pictures. For Christmas, she decides to give her mother some nice prints of her family, so she signs up for an account with another site called photoprintingexample.com (Consumer). The new site, photoprintingexample.com, has a feature that allows Jane to select pictures stored in her photosharingexample.com account and transfer them to her photoprintingexample.com account to be printed.

Photoprintingexample.com already has registered for a Consumer Key and Consumer Secret from photosharingexample.com:

```
Consumer Key:  dpf43f3p2l4k3l03
Consumer Secret:  kd94hf93k423kf44
```

Jane elects to use this service. When photoprintingexample.com tries to retrieve Jane's pictures from photosharingexample.com, it receives an HTTP 401 Unauthorized error, indicating those photos are private. This is expected, because Jane hasn't authorized photoprintingexample.com access to her photosharingexample.com account yet. The Consumer sends the following request to the Service Provider: *Garrick, shrink below.*

```
https://photosharingexample.com/request_token?
```

```
↪oauth_consumer_key=dpf43f3p2l4k3l03&oauth_
↪signature_method=PLAINTEXT&oauth_signature=
↪kd94hf93k423kf44%26&oauth_timestamp=
↪1191242090&oauth_nonce=hsu94j3884jdopsl&oauth_version=1.0
```

Using nonces can be very costly for Service Providers, as they demand persistent storage of all nonce values ever received. To make server implementations less costly, OAuth adds a timestamp value to each request, which allows the Service Provider to keep nonce values only for a limited time. When a request comes in with a timestamp that is older than the retained time frame, it is rejected, because the Service Provider no longer has nonces from that time period.

Nonce*10965s2.qrk*

The term nonce means “number used once” and is a unique and usually random string that is meant to identify each signed request uniquely.

The Service Provider checks the signature of the request and replies with an unauthorized request token: *Garrick, one line below.*

```
oauth_token=hh5s93j4hdidpola&oauth_token_secret=hdhd0244k9j7ao03
```

The Consumer redirects Jane's browser to the Service Provider User Authorization URL: *Garrick, shrink below.*

```
http://photosharingexample.com/authorize?oauth_token=
↪hh5s93j4hdidpola&oauth_callback=
↪http%3A%2F%2Fphotoprintingexample.com%2Frequest_token_ready
```

If Jane is logged in to photosharingexample.com, this page will ask her whether she authorizes photoprintingexample.com to have access to her account. If Jane authorizes the request, her browser will be redirected back to http://photoprintingexample.com /request_token_ready?oauth_token=hh5s93j4hdidpola, telling the consumer that the request token has been authorized. The Consumer then will exchange the Request Token for an Access Token using the following address: *Garrick, shrink below.*

```
https://photosharingexample.com/access_token?
↪oauth_consumer_key=dpf43f3p2l4k3l03&oauth_token=
↪hh5s93j4hdidpola&oauth_signature_method=PLAINTEXT&
↪oauth_signature=kd94hf93k423kf44%26hdhd0244k9j7ao03&
↪oauth_timestamp=1191242092&oauth_nonce=
↪dji430splmx33448&oauth_version=1.0
```

which will return the Access Token in the response: *Garrick, one line below.*

```
oauth_token=nnch734d00sl2jdk&oauth_token_secret=pfkkdhi9sl3r4s00
```

This exchange will happen only the first time Jane tries to access her photosharingexample.com photos from photoprintingexample.com. Any time afterward, only the following will happen.

Now, the Consumer is equipped properly to access Jane's photos. First, the Consumer needs to generate the request signature. The initial step is to create the Signature Base String. This is a combination of the following elements:

```
oauth_consumer_key: dpf43f3p2l4k3l03
oauth_token: nnch734d00sl2jdk
oauth_signature_method: HMAC-SHA1
oauth_timestamp: 1191242096
oauth_nonce: kllo9940pd9333jh
oauth_version: 1.0
file: family.jpg
size: original
```

Ultimately, you end up with the string: *Garrick, shrink below.*

```
GET&http%3A%2F%2Fphotosharingexample.com%2Fphotos&
↪file%3Dfamily.jpg%26oauth_consumer_key%
↪3Ddpf43f3p2l4k3l03%26oauth_nonce%3Dkllo9940pd9333jh%
↪26oauth_signature_method%3DHMAC-SHA1%26oauth_timestamp%
↪3D1191242096%26oauth_token%3Dnnch734d00sl2jdk%
↪26oauth_version%3D1.0%26size%3Doriginal"
```

If your request is being transmitted through SSL, the request can be in plain text. However, a vast majority of Web sites do not use SSL, so the signature string must be encoded.

Traditionally, the HTTP protocol uses an authentication method it calls “Basic” in which users provide their user names and passwords in order to gain access to the protected resource. The major flaw in that procedure is that those credentials are passed in plain text, clear for any people listening to read and store as they wish. *In order to protect users' credentials, OAuth uses digital signatures instead of sending credentials with each request.*

This digital signature is used to verify that the request being made is legitimate and hasn't been tampered with. A hashing algorithm is used to make that work. In order to allow the recipient to verify that the request came from the claimed sender, the hash algorithm is combined with a shared secret. If both sides agree on a secret known only to both parties, they can add it to the content being hashed. This can be done by simply appending the secret to the content, or by using a more sophisticated algorithm with a built-in mechanism for secrets, such as HMAC.

Note:*10965s3.qrk*

OAuth defines three signature methods used to sign and verify requests: PLAINTEXT, HMAC-SHA1 and RSA-SHA1.

Hash Algorithm*10965s4.qrk*

The process of taking data (of any size) and condensing it to a much smaller value (digest) in a fully reproducible (one-way) manner. Using the same hash algorithm on the same data always will produce the same smaller value.

For this example, let's say the Service Provider allows HMAC-SHA1 signatures. Thus, the encoded signature string becomes:

```
tR3+Ty81lMeYAr/Fid0kMTYa/WM=
```

All together, the Consumer request for the photo is: *Garrick, shrink below.*

```
http://photosharingexample.com/photos?file=vacation.jpg&size=
↪original&oauth_consumer_key=dpf43f3p2l4k3l03&
↪oauth_token=nnch734d00sl2jdk&oauth_signature_method=
↪HMAC-SHA1&oauth_signature=tR3%2BTy81lMeYAr%2FFid0kMTYa%
↪2FWM%3D&oauth_timestamp=1191242096&oauth_nonce=
↪kllo9940pd9333jh&oauth_version=1.0
```

The Service Provider performs the same work flow to calculate the signature of the request that the Consumer performs. It then compares its calculated signature to the provided signature. If the two match, the recipient can be confident that the request has not been modified in transit. The Service Provider then responds with the requested pictures.

This process can be daunting to deal with programmatically. There are a number of libraries written for both the OAuth server and client for quite a few programming languages.

Security Issues

The shared secret used to verify the request signature is called the Consumer Secret. Because it is vital to the integrity of this transaction, it is imperative that this piece of data be kept secret. In the case of a Web-based Consumer, such as a Web service, it is easy to keep the Consumer Secret safe. If the Consumer is a client-side application, the Consumer Secret must be hard-coded in each copy of the application. This means the Consumer Secret potentially is discoverable, which compromises the integrity of any desktop application.

There is a known session fixation attack vulnerability found in the OAuth 1.0 protocol that allows an attacker to gain access to a target's account. The attacker logs in to the Consumer site and initiates the OAuth authorization process. The attacker saves the authorization request page instead of clicking submit. This stores the request token and secret. The attacker sends a link to a victim, which, if clicked, will continue the authorization process as started by the attacker. Once completed, the attacker will have access to the victim's protected resources via the Consumer used.