**Paper SAS224-2017**

# Show Off Your OAuth

Joseph Henry, SAS Institute Inc.

## ABSTRACT

Applications and websites are heavily relying on web services to serve up vast amounts of data. With such a substantial reliance on the web and the rapid increase of security threats, security is a necessary concern. OAuth 2.0 has become a go-to method for websites to allow secure access to the services they provide; however, increased security inherently increases complexity. Accessing web services that use OAuth 2.0 is not entirely straightforward, and can cause many users plenty of frustration and confusion. OAuth can be simple to use and pairs well with Base SAS® to access the most popular web services available.

## INTRODUCTION

As massive amounts of data is now being hosted on the web, our communications to retrieve that data are increasingly moving from intranets to the Internet. This move to the Internet makes electronic security more critical than ever. In addition, we now see a steadily increasing number of applications that are interacting with content providers instead of the user; this requires the applications to authenticate to a content provider on the user's behalf. A user may ask, "How does all this happen? How do I keep my passwords safe?" Many protocols have been developed over the years attempting to address these concerns, but at present, OAuth is emerging as the preferred protocol for web service security. To fully comprehend OAuth, you first should understand what it is, why it is useful, and how you can use it in SAS code to communicate with certain web services.

## WHAT IS OAUTH?

Simply put, OAuth is an authentication protocol that allows an application to interact with one or more other applications on your behalf without providing your password. While this may sound simple enough, in practice, it is not quite so simple. Eran Hammer explained OAuth using the following metaphor: Some cars today come equipped with a valet key. This is a key that you would give to a parking-lot attendant that gives limited access to your car. The valet key can unlock the doors and start the car, but unlike your real key, the car will only be drivable for a few minutes, and might not allow access to the glove compartment or trunk. Basically, the valet key is a way to give access to a portion of the vehicle without having to trust a stranger with your key.

This comparison is very apt for describing how many web services work. We see many services that tie in one or more other services for convenience or efficiency, similar to the way that many web applications have a "Share on Facebook" or "Share on Twitter" button. It might be useful for the application to be able to access those other services, but if they require your user name and password, you are now essentially handing over your keys to a total stranger. OAuth provides a way for you to hold on to your keys—it allows an application to access all or part of a resource on your behalf, without needing to know your user name and password. With OAuth, you are only giving an application "a valet key" to Twitter, Facebook, or whatever web service you want to use.

## WHY OAUTH?

User names and passwords have been used for decades in web services, so why is a solution like OAuth needed now? The problem that OAuth solves is not the security of the web service itself, but the security of the additional web services that it uses on your behalf. The primary goal of OAuth is to allow a web service to access a content provider, e.g. Facebook or Twitter, without having to know your password. For instance, suppose that a web service has a feature that will automatically post to your Twitter feed. To do this without OAuth, this service would need to know your user name and password, and store it in its own database. If that service is hacked, or if your password for that service gets compromised, the attacker

now has your Twitter user name and password. The security risk increases as more services are used in this way.

With OAuth, the service does not require your user name or password. It simply submits an authorization request to the content provider, and after a series of steps, returns an Access Token to the web service. This token is your “valet key”. As a result, if the service gets hacked, the attacker might be able to post to your Twitter feed for a little while, but would not be able to access any of your settings or your profile. You can also easily revoke access to that service without the need to change passwords.

## HOW DOES OAUTH WORK?

Now that you know what OAuth is and why it is used, you need to understand how OAuth works to really be able to use it. The primary OAuth flow is known as three-legged OAuth. The “legs” refer to the roles involved, which are as follows:

1. A third-party application (Client).
   This is typically a web application in which you personally have an account, such as Strava or LinkedIn.
2. An authorization server.
   This is the authorization server owned by the content provider, that is, the web page where you log on to sites such as Twitter or Google.
3. A resource owner (User).
   This is not the user of the application (although they could be the same person), but the owner of the resource that you want to access from the content provider.

Let’s say that someone in my home is using my TV to watch a show on Netflix. I, being the owner of the Netflix account, have authorized the TV to use the account on my behalf. In this case, the TV is the client, Netflix is the authorization server, and I am the user since I own the Netflix account. The person watching the TV is simply a consumer of the application, not necessarily the resource user.

To further explain the steps involved, I will walk you through the seven steps required with an example using the Google Drive API in SAS code.

### STEP 1. CLIENT AUTHORIZATION

The application sends the user to the authorization server to authorize the application to act on the user’s behalf. This is where you first tell the authorization server who the client is, and how you would like to communicate. With Google, you accomplish this by sending a GET request to `https://accounts.google.com/o/oauth2/v2/auth` along with the following query parameters:

- client_id – The ID of the registered third-party application.
- redirect_uri – The URI that the authorization server can call back into your application with the response. For web base clients, this is where the web server will be listening for authorization servers response.
- response_type – For this type of authorization, this value will always be "code", meaning that we want to receive an authorization code from the server.
- scope – This defines the level of access that you are requesting of the resource. Typical values are openid and email.
- state – This is an arbitrary value that allows the client to keep track of callbacks from the authorization server. **Note**: We do not need this value in the example, but it is required by the Google API, so we have included it here.

Because we are executing this call directly from SAS code, we do not have an HTTP server to receive the callback from the server. We therefore need to set the value of redirect_uri to a special value to communicate this information to the server. How this is achieved can be different with each service, but usually the service documentation will contain a session on connecting “TVs and other devices” that will

go over this type of scenario. For Google APIs, we simply present a special URI with the value urn:ietf:wg:oauth:2.0:oob. The SAS code to perform this step would look like the following example:

```
%let auth_url=https://accounts.google.com/o/oauth2/v2/auth;
%let client_id=873108665846-
l1j0nk69ip0cuu1mcgmft991ukm68803.apps.googleusercontent.com;
%let redirect_uri=urn:ietf:wg:oauth:2.0:oob;
%let drive_scope=https://www.googleapis.com/auth/drive;
%let
url=&auth_url.?client_id=&client_id.%nrstr(&redirect_uri)=&redirect_uri.%nrst
r(&response_type=code&scope=openid%20email)%20&drive_scope.&state=security_to
ken);

proc http
 url="&url"
 headerout=hdrs
 nofollow;
run;
```

**Note**: We add the “nofollow” parameter to the HTTP Procedure here because we do not actually want to follow the redirect. In a web browser, you just end up at the authentication page, but because we are doing this in SAS code, a few extra steps are required.

In this example, besides asking for the basic scope of user ID and email, you are also asking for access to Google Drive with the scope “https://www.googleapis.com/auth/drive”. Later on in the example, a call will be made to Google Drive, so that scope is important.

The response that comes back should be similar to what is shown in Output 1.

```
< HTTP/1.1 302 Found
< Content-Type: application/binary
< Location:
https://accounts.google.com/ServiceLogin?passive=1209600&continue=https://a
ccounts.google.com/o
/oauth2/v2/auth?client_id%3D873108665846-
l1j0nk69ip0cuu1mcgmft991ukm68803.apps.googleuserconten
t.com%26redirect_uri%3Durn:ietf:wg:oauth:2.0:oob%26response_type%3Dcode%26s
cope%3Dopenid%2Bemai
l%2Bhttps://www.googleapis.com/auth/drive%26state%3Dsecurity_token)%26from_
login%3D1%26as%3D-3d
8f97771013b8bf&ltmpl=nosignup&oauth=1&sarp=1&scc=1
< Content-Length: 0
<
```

**Output 1. Response from initial authorization request**

Ultimately, you want to open a web browser to the URI that is in the Location header. An easy way to get the value of the location header would be to execute the following SAS code:

```
data _null_;
  infile hdrs length=len scanover truncover;
  input @'Location: ' loc $varying1024. len;

  call symput('location',trim(loc));
run;
```

Now we have the redirected location in the macro “location”.

## STEP 2. USER AUTHENTICATION

If the application is validated, the authorization server requests the user to log on. This is where the user authenticates to the service and where user interaction is required. You need to get the authentication URI that is stored in the location macro into a web browser. You can just print it out and copy and paste it into a browser, or if you are running SAS locally and have x commands enabled, you can do something like this:

```
options noxsync noxwait;
x "start """" ""&location.""";
```

On Microsoft Windows systems, this code will open up your default browser to the URL. Either way, you will be presented with a logon page similar to what is shown in Image 1.

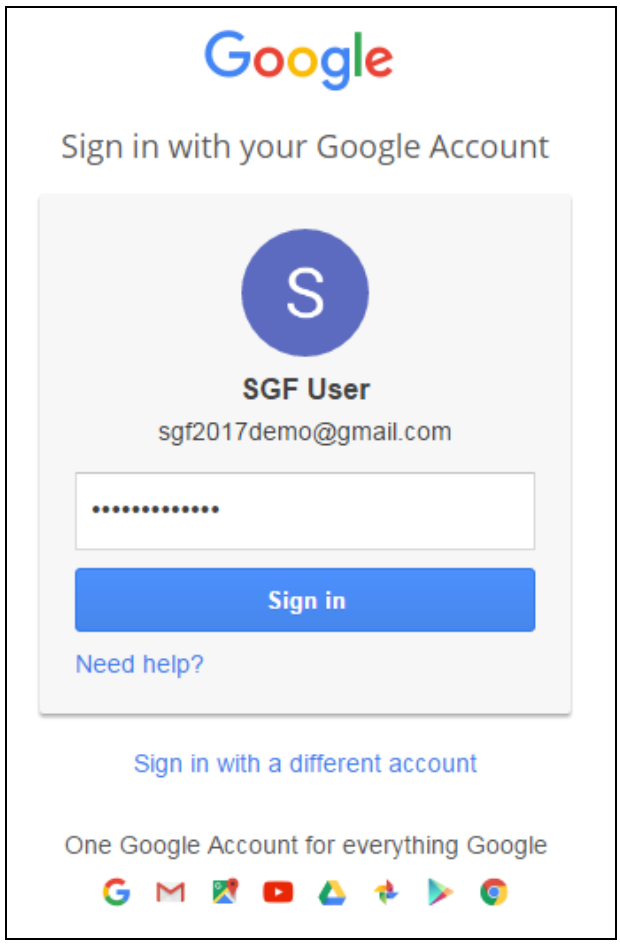


**Image 1. Sample user authentication page.**

Now you enter the credentials of the resource owner.

## STEP 3. USER AUTHORIZATION

The user then explicitly authorizes the application scope, that is, the parts of the user's account that the application is allowed to access. Do not confuse authorization with authentication. Authentication is what happened in step 2, where you logged in with the user's credentials. *Authorization* refers to the permissions that the application is allowed to have to the user's resource, like what is shown in Image 2.

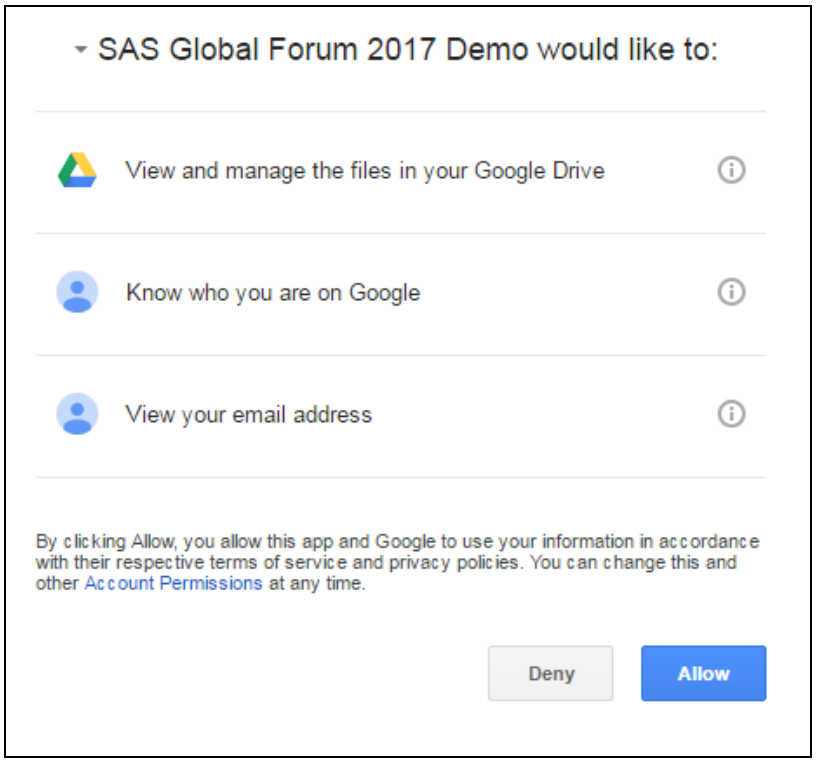


**Image 2. Sample scope validation page**

Here the user can either explicitly allow or deny access to the given areas.

## STEP 4. RESPONSE CODE

At this point, the user is redirected back to the application with an authorization code. In web-based applications, the authorization server sends a response back to the URI that was set in the redirect_uri parameter with an authorization code. The client will then verify that the state matches what was originally sent to the authorization server and retrieve the code. Because SAS is not a web application, and we set the redirect URI to the special value, Google will display the authorization code on the screen, similar to what is shown in Image 3. Other web services might not have a special redirect_uri value, but will instead just show the authorization code as part of the URL that would be displayed in your browser.

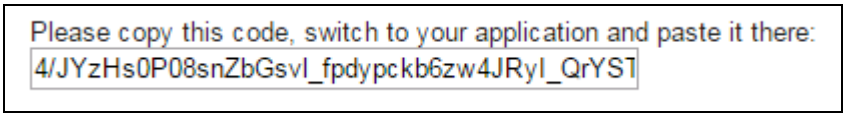


**Image 3. Sample authorization code display**

Copy the authorization code and paste it into your SAS program for later use, as in the following example:

```
%let code=4/K1Idr1Bp9a1EGUdty22xk4WSch9a4JW1HhJ63qmH3tI;
```

## STEP 5. REQUEST ACCESS TOKEN

An out-of-band request is now made to the authorization server, exchanging the code and application credentials for an access token. To request an access token, we send a POST to `https://www.googleapis.com/oauth2/v4/token` with the following form encoded parameters:

- client_id – The same client ID that was used on the initial call.
- client_secret – This is essentially the client's password. You should have had one generated when you registered your application.

- redirect_uri – This would typically be the URI of the location where you want the application user to end up after all the authentication is complete. In this case, you just use the same redirect_uri that you used for the initial call to the authorization server.
- grant_type – Because we are exchanging an authorization code for an access token, this value needs to be authorization_code.
- code – This is the authorization code that was returned to the application.

**Note**: This is the step where application-only authentication begins, but instead of grant_type=authorization_code, you would use grant_type=client_credentials.

In this step, you are sending your client secret to the authentication server. You need to take special care not to expose this information, so instead of having the secret directly set in your SAS code, you could store the secret in a file for which only you have read permissions and read it into a variable when needed, as in the following example:

```
filename sec "secret.dat";
data _null_;
 length str $1024;
   fid = fopen("sec");
   rc = fread(fid);
   rc = fget(fid, str, 256);

   call symput("client_secret",trim(str));

   rc = fclose(fid);
run;
```

Now you can build a PROC HTTP request as follows:

```
filename resp TEMP;
proc http url="https://www.googleapis.com/oauth2/v4/token"
  method="POST"
  out=resp
  headerout=hdrs
  ct="application/x-www-form-urlencoded"

in="code=&code.%nrstr(&client_id)=&client_id.%nrstr(&client_secret)=&client_s
ecret.%nrstr(&redirect_uri)=&redirect_uri.&grant_type=authorization_code";
  run;

%let client_secret=;
```

If everything works properly, you should receive a response similar to Output 2:

```
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=UTF-8
< Transfer-Encoding: chunked
<
{
 "access_token":
"ya29.GlvgA7GhQK0W1TMWo2VpKbbLaypiqRy_l8v5TpWpp0qCh-t2Pf1RuEwDMCVuWOEW-
xnFeXVLIH0l6UBG7iHTV8bBW
LEvQ7s-TJhBQWW6TfGLcB4dGk39g47OyNn0",
 "token_type": "Bearer",
 "expires_in": 3600,

"refresh_token": "1/Ac6MhfdTHEE8ARQbcDJ4EtVEeo7W3m76B3RloXtLsgc"
}
```

**Output 2. Sample output for access token request**

What you want from this response is the access_token. You can easily parse the response and retrieve the access token as follows:

```
data _null_;
 infile resp truncover scanover length=len;
 input @'"access_token": ' t $varying1024. len;
 token = dequote(t);
 call symput("access_token",trim(token));
run;
```

Now you have been authenticated successfully. You can now use the access token for all subsequent calls to the service. The token will eventually expire, and at that point, you can either refresh the access token or generate a new one. Refresh tokens are not discussed in this paper, but the process would be very similar to step 4.

## STEP 6. CLIENT ACCESS

The application can now make requests to the content provider on behalf of the user by passing the access token with each request.

Now that you have an access token, you can use it to make calls to the service as if you were the user. In PROC HTTP, you can easily use the headers statement to add the access token to all of your subsequent service calls, as in the following call to the Google Drive API:

```
filename sample "Getting Started.pdf";
 proc http
url="https://www.googleapis.com/drive/v3/files/0BwfmKHYUomArc3RhcnRlcl9maWxl?
alt=media"
 out=sample;
 headers "Authorization" = "Bearer &access_token.";
run;
```

You can now access the authenticated user's Google Drive account (or whatever resource scope was specified). Note: The above example assumes that you have a file in Google Drive with the ID of 0BwfmKHYUomArc3RhcnRlcl9maWxl. The ID of any file will be unique to you.

## STEP 7. ACCESS TOKEN VALIDATION

Each time a request is made to the content provider with the access token, the content provider talks to the authorization server to validate the user, scope, and so on.

This step is ongoing. Every time you use the access token, it is validated.

## TWO-LEGGED OAUTH

A lot of content providers also implement two-legged OAuth, or Application-Only OAuth. It is referred to as “application-only” because there is no user context. Two-legged OAuth works pretty much the same as three-legged OAuth, except that you can skip steps 1 through 3 because a user is not needed. You would instead start at step 4, but rather than provide a code and a grant_type of authorization_code, you would instead use a grant_type of client_credentials. You would then be returned an access_token that could be used with certain APIs that do not require a user context. For example in Twitter (which lets you can use Application-Only authentication to read from a Twitter feed) you can get an access token using client_credentials like:

```
proc http
url="https://api.twitter.com/oauth2/token?grant_type=client_credentials"
 in=''
 out=out
 headerout=hdrs
 webusername="&client_id"
 webpassword="&client_secret"
 AUTH_BASIC;
run;
```

From here you can proceed to step 5 to parse the access token.

## CONCLUSION

OAuth provides a secure way for web services to interact with each other without compromising the security of the user. Although OAuth is not as straightforward as basic authentication with usernames and passwords, the benefit of not having to expose your password definitely compensates for any loss of simplicity. With a bit of extra understanding of how OAuth works, you can fairly easily write SAS applications that interact with OAuth-secured web services and feel good about the fact that the security of your application is stronger than ever.

## REFERENCES

Hammer, Eran. "Explaining OAuth." Hueniverse. N.p., 05 Sept. 2007. Accessed 01 Jan. 2017. https://hueniverse.com/2007/09/05/explaining-oauth/

## CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Joseph Henry
100 SAS Campus Drive
Cary, NC 27513
SAS Institute, Inc.
Joseph.Henry@sas.com

http://www.sas.com