Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 7, 2010

E. Hammer-Lahav, Ed.
Yahoo!

July 6, 2009

The OAuth Protocol: Web Delegation
draft-ietf-oauth-web-delegation-01

Status of this Memo

Copyright Notice

Abstract

This document specifies the OAuth protocol web delegation method. OAuth allows clients to access server resources on behalf of another party (such a different client or an end user). This document

defines a redirection-based user-agent process for end users to authorize access to clients by substituting their credentials (typically, a username and password pair) with a different set of delegation-specific credentials.

Table of Contents

1. Introduction

The OAuth protocol provides a method for servers to allow third-party access to protected resources, without forcing their end users to share their credentials. This pattern is common among services that allow third-party developers to extend the service functionality, by building applications using an open API.

For example, a web user (resource owner) can grant a printing service (client) access to its private photos stored at a photo sharing service (server), without sharing its credentials with the printing service. Instead, the user authenticates directly with the photo sharing service and issue the printing service delegation-specific credentials.

OAuth introduces a third role to the traditional client-server authentication model: the resource owner. In the OAuth model, the client requests access to resources hosted by the server but not controlled by the client, but by the resource owner. In addition, OAuth allows the server to verify not only the resource owner's credentials, but also those of the client making the request.

In order for the client to access resources, it first has to obtain permission from the resource owner. This permission is expressed in the form of a token and matching shared-secret. The purpose of the token is to substitute the need for the resource owner to share its server credentials (usually a username and password pair) with the client. Unlike server credentials, tokens can be issued with a restricted scope and limited lifetime.

This specification consists of two parts. [draft-ietf-oauth-authentication] defines a method for making authenticated HTTP requests using two sets of credentials, one identifying the client making the request, and a second identifying the resource owner on whose behalf the request is being made.

This document defines a redirection-based user agent process for end users to authorize client access to their resources, by authenticating directly with the server and provisioning tokens to the client for use with the authentication method.

1.1. Terminology

client
: An HTTP client (per [RFC2616]) capable of making OAuth-authenticated requests per [draft-ietf-oauth-authentication].

server
: An HTTP server (per [RFC2616]) capable of accepting OAuth-authenticated requests per [draft-ietf-oauth-authentication].

protected resource
: An access-restricted resource (per [RFC2616]) which can be obtained from the server using an OAuth-authenticated request per [draft-ietf-oauth-authentication].

resource owner
: An entity capable of accessing and controlling protected resources by using credentials to authenticate with the server.

token
: An unique identifier issued by the server and used by the client to associate authenticated requests with the resource owner whose authorization is requested or has been obtained by the client. Tokens have a matching shared-secret that is used by the client to establish its ownership of the token, and its authority to represent the resource owner.

2. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. Redirection-Based Authorization

OAuth uses a set of token credentials to represent the authorization granted to the client by the resource owner. Typically, token credentials are issued by the server at the resource owner's request, after authenticating the resource owner's identity using its server credentials (usually a username and password pair).

There are many ways in which a resource owner can facilitate the provisioning of token credentials. This section defines one such way, using HTTP redirections and the resource owner's user agent. This redirection-based authorization method includes three steps:

1. The client obtains a set of temporary credentials from the server.

2. The resource owner authorizes the server to issue token credentials to the client using the temporary credentials.

3. The client uses the temporary credentials to request a set of token credentials from the server, which will enable it to access the resource owner's protected resources. The temporary credentials discarded.

The temporary credentials MUST be revoked after being used once to obtain the token credentials. It is RECOMMENDED that the temporary credentials have a limited lifetime. Servers SHOULD enable resource owners to revoke token credentials after they have been issued to clients.

In order for the client to perform these steps, the server needs to advertise the URIs of these three endpoints, as well as the HTTP method (GET, POST, etc.) used to make each requests. To assist in communicating these endpoint, each is given a name:

Temporary Credential Request
: The endpoint used by the client to obtain temporary credentials as described in Section 4.

Resource Owner Authorization
: The endpoint to which the resource owner is redirected to grant

authorization as described in Section 5.

Token Request
: The endpoint used by the client to request a set of token credentials using the temporary credentials as described in Section 6.

The three URIs MAY include a query component as defined by [RFC3986] section 3, but if present, the query MUST NOT contain any parameters beginning with the "oauth_" prefix.

The method in which the server advertises its three endpoint is beyond the scope of this specification.

4. Temporary Credentials

The client obtains a set of temporary credentials from the server by making an authenticated request per [draft-ietf-oauth-authentication]. The client MUST use the HTTP method advertised by the server. The HTTP POST method is RECOMMENDED. The client constructs a request URI by adding the following parameter to the Temporary Credential Request endpoint URI:

oauth_callback: An absolute URL to which the server will redirect the resource owner back when the Resource Owner Authorization step (Section 5) is completed. If the client is unable to receive callbacks or a callback URI has been established via other means, the parameter value MUST be set to "oob" (case sensitive), to indicate an out-of-band configuration.

Servers MAY specify additional parameters.

When making the request, the client authenticates using only the client credentials. The client MUST omit the "oauth_token" protocol parameter from the request and use an empty string as the token secret value.

The server MUST verify that the request is valid per

[draft-ietf-oauth-authentication] and respond back to the client with a set of temporary credentials. The temporary credentials are included in the HTTP response body using the "application/x-www-form-urlencoded" content type as defined by [W3C.REC-html40-19980424].

The response contains the following parameters:

oauth_token
: The temporary credentials identifier.

oauth_token_secret
: The temporary credentials shared-secret.

oauth_callback_confirmed: MUST be present and set to "true". The client MAY use this to confirm that the server received the callback value.

Note that even though the parameter names include the term 'token', these credentials are not token credentials, but are used in the next two steps in a similar manner to token credentials.

For example (line breaks are for display purposes only):

```
oauth_token=ab3cd9j4ks73hf7g&oauth_token_secret=xyz4992k83j47x0b&
oauth_callback_confirmed=true
```

5. Resource Owner Authorization

Before the client requests a set of token credentials from the server, it MUST send the user to the server to authorize the request. The client constructs a request URI by adding the following

parameters to the Resource Owner Authorization endpoint URI:

oauth_token
: REQUIRED. The temporary credentials identifier obtained in Section 4 in the "oauth_token" parameter. Servers MAY declare this parameter as OPTIONAL, in which case they MUST provide a way for the resource owner to indicate the identifier through other means.

Servers MAY specify additional parameters.

The client redirects the resource owner to the constructed URI using an HTTP redirection response, or by other means available to it via the resource owner's user agent. The request MUST use the HTTP GET method.

The way in which the server handles the authorization request is beyond the scope of this specification. However, the server MUST first verify the identity of the resource owner.

When asking the resource owner to authorize the requested access, the server SHOULD present to the resource owner information about the client requesting access based on the association of the temporary credentials with the client identity. When displaying any such information, the server SHOULD indicate if the information has been verified.

After receiving an authorization decision from the resource owner, the server redirects the resource owner to the callback URI if one was provided in the "oauth_callback" parameter or by other means.

To make sure that the resource owner granting access is the same resource owner returning back to the client to complete the process, the server MUST generate a verification code: an unguessable value passed to the client via the resource owner and REQUIRED to complete the process. The server constructs the request URI by adding the following parameter to the callback URI query component:

oauth_token
: The temporary credentials identifier the resource owner authorized or denied access to.

oauth_verifier: The verification code.

If the callback URI already includes a query component, the server MUST append the OAuth parameters to the end of the existing query.

For example (line breaks are for display purposes only):

```
http://client.example.net/cb?state=1&oauth_token=ab3cd9j4ks73hf7g&
oauth_verifier=473829k9302sa
```

If the client did not provide a callback URI, the server SHOULD display the value of the verification code, and instruct the resource owner to manually inform the client that authorization is completed. If the server knows a client to be running on a limited device it SHOULD ensure that the verifier value is suitable for manual entry.

6. Token Credentials

The client obtains a set of token credentials from the server by making an authenticated request per [draft-ietf-oauth-authentication]. The client MUST use the HTTP method advertised by the server. The HTTP POST method is RECOMMENDED. The client constructs a request URI by adding the following parameter to the Token Request endpoint URI:

oauth_verifier: The verification code received from the server in the previous step.

When making the request, the client authenticates using the client credentials as well as the temporary credentials. The temporary credentials are used as a substitution for token credentials in the authenticated request.

The server MUST verify the validity of the request per [draft-ietf-oauth-authentication], ensure that the resource owner has authorized the provisioning of token credentials to the client, and that the temporary credentials have not expired or used before. The server MUST also verify the verification code received from the client. If the request is valid and authorized, the token credentials are included in the HTTP response body using the "application/x-www-form-urlencoded" content type as defined by [W3C.REC-html40-19980424].

The response contains the following parameters:

oauth_token
: The token identifier.

oauth_token_secret
: The token shared-secret.

For example:

oauth_token=j49ddk933skd9dks&oauth_token_secret=ll399dj47dskfjdk

The token credentials issued by the server MUST reflect the exact scope, duration, and other attributes approved by the resource owner.

Once the client receives the token credentials, it can proceed to access protected resources on behalf of the resource owner by making an authenticated request per [draft-ietf-oauth-authentication] using the client credentials and the token credentials received.

7. IANA Considerations

This memo includes no request to IANA.

8. Security Considerations

As stated in [RFC2617], the greatest sources of risks are usually found not in the core protocol itself but in policies and procedures surrounding its use. Implementers are strongly encouraged to assess how this protocol addresses their security requirements.

8.1. Credentials Transmission

The OAuth specification does not describe any mechanism for protecting tokens and shared-secrets from eavesdroppers when they are transmitted from the server to the client during the authorization phase. Servers should ensure that these transmissions are protected using transport-layer mechanisms such as TLS or SSL.

8.2. Phishing Attacks

Wide deployment of OAuth and similar protocols may cause resource owners to become inured to the practice of being redirected to websites where they are asked to enter their passwords. If resource owners are not careful to verify the authenticity of these websites before entering their credentials, it will be possible for attackers to exploit this practice to steal resource owners' passwords.

Servers should attempt to educate resource owners about the risks phishing attacks pose, and should provide mechanisms that make it easy for resource owners to confirm the authenticity of their sites.

8.3. Scoping of Access Requests

By itself, OAuth does not provide any method for scoping the access rights granted to a client. However, most applications do require greater granularity of access rights. For example, servers may wish to make it possible to grant access to some protected resources but not others, or to grant only limited access (such as read-only access) to those protected resources.

When implementing OAuth, servers should consider the types of access resource owners may wish to grant clients, and should provide mechanisms to do so. Servers should also take care to ensure that resource owners understand the access they are granting, as well as any risks that may be involved.

8.4. Entropy of Secrets

Unless a transport-layer security protocol is used, eavesdroppers will have full access to OAuth requests and signatures, and will thus be able to mount offline brute-force attacks to recover the credentials used. Servers should be careful to assign shared-secrets which are long enough, and random enough, to resist such attacks for at least the length of time that the shared-secrets are valid.

For example, if shared-secrets are valid for two weeks, servers should ensure that it is not possible to mount a brute force attack that recovers the shared-secret in less than two weeks. Of course, servers are urged to err on the side of caution, and use the longest secrets reasonable.

It is equally important that the pseudo-random number generator (PRNG) used to generate these secrets be of sufficiently high quality. Many PRNG implementations generate number sequences that may appear to be random, but which nevertheless exhibit patterns or other weaknesses which make cryptanalysis or brute force attacks easier. Implementers should be careful to use cryptographically secure PRNGs to avoid these problems.

8.5. Denial of Service / Resource Exhaustion Attacks

The OAuth protocol has a number of features which may make resource exhaustion attacks against servers possible. For example, if a server includes a nontrivial amount of entropy in token shared-secrets as recommended above, then an attacker may be able to exhaust the server's entropy pool very quickly by repeatedly obtaining temporary credentials from the server.

Similarly, OAuth requires servers to track used nonces. If an

attacker is able to use many nonces quickly, the resources required to track them may exhaust available capacity. And again, OAuth can require servers to perform potentially expensive computations in order to verify the signature on incoming requests. An attacker may exploit this to perform a denial of service attack by sending a large number of invalid requests to the server.

Resource Exhaustion attacks are by no means specific to OAuth. However, OAuth implementers should be careful to consider the additional avenues of attack that OAuth exposes, and design their implementations accordingly. For example, entropy starvation typically results in either a complete denial of service while the system waits for new entropy or else in weak (easily guessable) secrets. When implementing OAuth, servers should consider which of these presents a more serious risk for their application and design accordingly.

8.6. Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is a web-based attack whereby HTTP requests are transmitted from a user that the website trusts or has authenticated. CSRF attacks on OAuth approvals can allow an attacker to obtain authorization to protected resources without the consent of the User. Servers SHOULD strongly consider best practices in CSRF prevention at all OAuth endpoints.

CSRF attacks on OAuth callback URIs hosted by client are also possible. Clients should prevent CSRF attacks on OAuth callback URIs by verifying that the resource owner at the client site intended to complete the OAuth negotiation with the server.

8.7. User Interface Redress

Servers should protect the authorization process against UI Redress attacks (also known as "clickjacking"). As of the time of this writing, no complete defenses against UI redress are available. Servers can mitigate the risk of UI redress attacks through the following techniques:

- Javascript frame busting.
- Javascript frame busting, and requiring that browsers have javascript enabled on the authorization page.
- Browser-specific anti-framing techniques.
- Requiring password reentry before issuing OAuth tokens.

8.8. Automatic Processing of Repeat Authorizations

Servers may wish to automatically process authorization requests (Section 5) from clients which have been previously authorized by the resource owner. When the resource owner is redirected to the server to grant access, the server detects that the resource owner has already granted access to that particular client. Instead of prompting the resource owner for approval, the server automatically redirects the resource owner back to the client.

If the client credentials are compromised, automatic processing creates additional security risks. An attacker can use the stolen client credentials to redirect the resource owner to the server with an authorization request. The server will then grant access to the resource owner's data without the resource owner's explicit approval, or even awareness of an attack. If no automatic approval is implemented, an attacker must use social engineering to convince the resource owner to approve access.

Servers can mitigate the risks associated with automatic processing by limiting the scope of token credentials obtained through automated approvals. Tokens credentials obtained through explicit resource owner consent can remain unaffected. clients can mitigate the risks associated with automatic processing by protecting their client credentials.

Appendix A. Examples

In this example, photos.example.net is a photo sharing website (server), and printer.example.com is a photo printing service (client). Jane (resource owner) would like printer.example.com to print a private photo stored at photos.example.net.

When Jane signs-into photos.example.net using her username and password, she can access the photo by requesting the URI "http://photos.example.net/photo?file=vacation.jpg" (which also supports the optional "size" parameter). Jane does not want to share her username and password with printer.example.com, but would like it to access the photo and print it.

The server documentation advertises support for the "HMAC-SHA1" and "PLAINTEXT" methods, with "PLAINTEXT" restricted to secure (HTTPS) requests. It also advertises the following endpoint URIs:

Temporary Credential Request
https://photos.example.net/initiate, using HTTP POST

Resource Owner Authorization URI:
http://photos.example.net/authorize, using HTTP GET

Token Request URI:
https://photos.example.net/token, using HTTP POST

The printer.example.com has already established client credentials with photos.example.net:

Client Identifier
"dpf43f3p2l4k3l03"

Client Shared-Secret:
"kd94hf93k423kf44"

When printer.example.com attempts to print the request photo, it receives an HTTP response with a 401 (Unauthorized) status code, and a challenge to use OAuth:

```
WWW-Authenticate: OAuth realm="http://photos.example.net/"
```

Appendix A.1. Obtaining Temporary Credentials

The client sends the following HTTPS POST request to the server:

```
POST /initiate HTTP/1.1
Host: photos.example.net
Authorization: OAuth realm="http://photos.example.com/",
   oauth_consumer_key="dpf43f3p2l4k3l03",
   oauth_signature_method="PLAINTEXT",
   oauth_signature="kd94hf93k423kf44%26",
   oauth_timestamp="1191242090",
   oauth_nonce="hsu94j3884jdopsl",
   oauth_version="1.0",
   oauth_callback="http%3A%2F%2Fprinter.example.com%2Fready"
```

The server validates the request and replies with a set of temporary credentials in the body of the HTTP response:

```
oauth_token=hh5s93j4hdidpola&oauth_token_secret=hdhd0244k9j7ao03&
oauth_callback_confirmed=true
```


Appendix A.2. Requesting Resource Owner Authorization

The client redirects Jane's browser to the server's Resource Owner Authorization endpoint URI to obtain Jane's approval for accessing her private photos.

```
http://photos.example.net/authorize?oauth_token=hh5s93j4hdidpola
```

The server asks Jane to sign-in using her username and password and if successful, asks her if she approves granting printer.example.com access to her private photos. Jane approves the request and is

redirects her back to the client's callback URI:

```
http://printer.example.com/ready?
oauth_token=hh5s93j4hdidpola&oauth_verifier=hfdp7dh39dks9884
```

Appendix A.3. Obtaining Token Credentials

After being informed by the callback request that Jane approved authorized access, printer.example.com requests a set of token credentials using its temporary credentials:

```
POST /token HTTP/1.1
Host: photos.example.net
Authorization: OAuth realm="http://photos.example.com/",
   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"
   oauth_verifier="hfdp7dh39dks9884"
```

The server validates the request and replies with a set of token credentials in the body of the HTTP response:

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

Appendix A.4. Accessing protected resources

The printer is now ready to request the private photo. Since the photo URI does not use HTTPS, the "HMAC-SHA1" method is required.

Appendix A.4.1. Generating Signature Base String

To generate the signature, it first needs to generate the signature base string. The request contains the following parameters

("oauth_signature" excluded) which need to be ordered and concatenated into a normalized string:

oauth_consumer_key

```
      "dpf43f3p2l4k3l03"

   oauth_token
      "nnch734d00sl2jdk"

   oauth_signature_method
      "HMAC-SHA1"

   oauth_timestamp
      "1191242096"

   oauth_nonce
      "kllo9940pd9333jh"

   oauth_version
      "1.0"

   file
      "vacation.jpg"

   size
      "original"
```

The following inputs are used to generate the signature base string:

1. The HTTP request method: "GET"

2. The request URI: "http://photos.example.net/photos"

3. The encoded normalized request parameters string: "file=vacation.jpg&oauth_consumer_key=dpf43f3p2l4k3l03&oauth_nonce=kllo9940pd9333jh&oauth_signature_method=HMAC-SHA1&oauth_timestamp=1191242096&oauth_token=nnch734d00sl2jdk&oauth_version=1.0&size=original"

The signature base string is (line breaks are for display purposes only):

```
  GET&http%3A%2F%2Fphotos.example.net%2Fphotos&file%3Dvacation.jpg%26
  oauth_consumer_key%3Ddpf43f3p2l4k3l03%26oauth_nonce%3Dkllo9940pd933
  3jh%26oauth_signature_method%3DHMAC-SHA1%26oauth_timestamp%3D119124
  2096%26oauth_token%3Dnnch734d00sl2jdk%26oauth_version%3D1.0%26size%
  3Doriginal
```

Appendix A.4.2. Calculating Signature Value

HMAC-SHA1 produces the following "digest" value as a base64-encoded string (using the signature base string as "text" and "kd94hf93k423kf44&pfkkdhi9sl3r4s00" as "key"):

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

Appendix A.4.3. Requesting protected resource

All together, the client request for the photo is:

```
GET /photos?file=vacation.jpg&size=original HTTP/1.1
Host: photos.example.com
Authorization: OAuth realm="http://photos.example.net/",
   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 photos.example.net sever validates the request and responds with the requested photo.

Appendix B. Acknowledgments

This specification is directly based on the [OAuth Core 1.0 Revision A] community specification which was the product of the OAuth community. OAuth was modeled after existing proprietary protocols and best practices that have been independently implemented by various web sites. This specification was orignially authored by: Mark Atwood, Dirk Balfanz, Darren Bounds, Richard M. Conlan, Blaine Cook, Leah Culver, Breno de Medeiros, Brian Eaton, Kellan Elliott-McCrea, Larry Halff, Eran Hammer-Lahav, Ben Laurie, Chris Messina, John Panzer, Sam Quigley, David Recordon, Eran Sandler, Jonathan Sergent, Todd Sieling, Brian Slesinsky, and Andy Smith.

Appendix C. Document History

[[To be removed by the RFC editor before publication as an RFC.]]

-01

- Moved all subsection from section 3 to the document root.

-00

- Transitioned from the individual submission draft-hammer-oauth-02 to working group draft.

- Split draft-hammer-oauth-02 into two drafts, one dealing with web delegation (this draft) and another dealing with authentication draft-ietf-oauth-web-authentication.

- Updated draft with changes from OAuth Core 1.0 Revision A to fix a session fixation exploit.

9. References

9.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[W3C.REC-html40-19980424] Hors, A., Jacobs, I., and D. Raggett, "HTML 4.0 Specification", World Wide Web Consortium Recommendation REC-html40-19980424, April 1998, <http://www.w3.org/TR/1998/REC-html40-19980424>.

[draft-ietf-oauth-authentication]
Hammer-Lahav, E., Ed., "The OAuth Protocol: Authentication".

9.2. Informative References

[OAuth Core 1.0 Revision A]
OAuth, OCW., "OAuth Core 1.0".

Author's Address

Eran Hammer-Lahav (editor)
Yahoo!

Email: eran@hueniverse.com
URI: http://hueniverse.com