                          The OAuth 1.0 Protocol

Abstract

   OAuth provides a method for clients to access server resources on
   behalf of a resource owner (such as a different client or an end-
   user). It also provides a process for end-users to authorize third-
   party access to their server resources without sharing their
   credentials (typically, a username and password pair), using user-
   agent redirections.

Status of This Memo

   This document is not an Internet Standards Track specification; it is
   published for informational purposes.

   This document is a product of the Internet Engineering Task Force
   (IETF). It represents the consensus of the IETF community. It has
   received public review and has been approved for publication by the
   Internet Engineering Steering Group (IESG). Not all documents
   approved by the IESG are a candidate for any level of Internet
   Standard; see Section 2 of RFC 5741.

   Information about the current status of this document, any errata,
   and how to provide feedback on it may be obtained at
   http://www.rfc-editor.org/info/rfc5849.

Table of Contents

1. Introduction

   The OAuth protocol was originally created by a small community of web
   developers from a variety of websites and other Internet services who
   wanted to solve the common problem of enabling delegated access to
   protected resources. The resulting OAuth protocol was stabilized at
   version 1.0 in October 2007, and revised in June 2009 (Revision A) as
   published at <http://oauth.net/core/1.0a>.

   This specification provides an informational documentation of OAuth
   Core 1.0 Revision A, addresses several errata reported since that
   time, and makes numerous editorial clarifications. While this
   specification is not an item of the IETF's OAuth Working Group, which
   at the time of writing is working on an OAuth version that can be
   appropriate for publication on the standards track, it has been
   transferred to the IETF for change control by authors of the original
   work.

   In the traditional client-server authentication model, the client
   uses its credentials to access its resources hosted by the server.
   With the increasing use of distributed web services and cloud
   computing, third-party applications require access to these server-
   hosted resources.

   OAuth introduces a third role to the traditional client-server
   authentication model: the resource owner. In the OAuth model, the
   client (which is not the resource owner, but is acting on its behalf)
   requests access to resources controlled by the resource owner, but
   hosted by the server. In addition, OAuth allows the server to verify
   not only the resource owner authorization, but also the identity of
   the client making the request.

   OAuth provides a method for clients to access server resources on
   behalf of a resource owner (such as a different client or an end-
   user). It also provides a process for end-users to authorize third-
   party access to their server resources without sharing their
   credentials (typically, a username and password pair), using user-
   agent redirections.

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

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 make it unnecessary for the resource owner to share its
credentials with the client. Unlike the resource owner credentials,
tokens can be issued with a restricted scope and limited lifetime,
and revoked independently.

This specification consists of two parts. The first part 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. The second part defines a method for making
authenticated HTTP [RFC2616] 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.

The use of OAuth with any transport protocol other than [RFC2616] is
undefined.

1.1. Terminology

client
      An HTTP client (per [RFC2616]) capable of making OAuth-
      authenticated requests (Section 3).

server
      An HTTP server (per [RFC2616]) capable of accepting OAuth-
      authenticated requests (Section 3).

protected resource
      An access-restricted resource that can be obtained from the
      server using an OAuth-authenticated request (Section 3).

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

credentials
      Credentials are a pair of a unique identifier and a matching
      shared secret. OAuth defines three classes of credentials:
      client, temporary, and token, used to identify and authenticate
      the client making the request, the authorization request, and
      the access grant, respectively.

token
      A 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.

   The original community specification used a somewhat different
   terminology that maps to this specifications as follows (original
   community terms provided on left):

   Consumer: client

   Service Provider: server

   User: resource owner

   Consumer Key and Secret: client credentials

   Request Token and Secret: temporary credentials

   Access Token and Secret: token credentials

1.2. Example

   Jane (resource owner) has recently uploaded some private vacation
   photos (protected resources) to her photo sharing site
   'photos.example.net' (server). She would like to use the
   'printer.example.com' website (client) to print one of these photos.
   Typically, Jane signs into 'photos.example.net' using her username
   and password.

   However, Jane does not wish to share her username and password with
   the 'printer.example.com' website, which needs to access the photo in
   order to print it. In order to provide its users with better
   service, 'printer.example.com' has signed up for a set of
   'photos.example.net' client credentials ahead of time:

   Client Identifier
      dpf43f3p2l4k3l03

   Client Shared-Secret:
      kd94hf93k423kf44

   The 'printer.example.com' website has also configured its application
   to use the protocol endpoints listed in the 'photos.example.net' API
   documentation, which use the "HMAC-SHA1" signature method:

   Temporary Credential Request
         https://photos.example.net/initiate

   Resource Owner Authorization URI:
         https://photos.example.net/authorize

   Token Request URI:
         https://photos.example.net/token

   Before 'printer.example.com' can ask Jane to grant it access to the
   photos, it must first establish a set of temporary credentials with
   'photos.example.net' to identify the delegation request. To do so,
   the client sends the following HTTPS [RFC2818] request to the server:

     POST /initiate HTTP/1.1
     Host: photos.example.net
     Authorization: OAuth realm="Photos",
         oauth_consumer_key="dpf43f3p2l4k3l03",
         oauth_signature_method="HMAC-SHA1",
         oauth_timestamp="137131200",
         oauth_nonce="wIjqoS",
         oauth_callback="http%3A%2F%2Fprinter.example.com%2Fready",
         oauth_signature="74KNZJeDHnMBp0EMJ9ZHt%2FXKycU%3D"

   The server validates the request and replies with a set of temporary
   credentials in the body of the HTTP response (line breaks are for
   display purposes only):

     HTTP/1.1 200 OK
     Content-Type: application/x-www-form-urlencoded

     oauth_token=hh5s93j4hdidpola&oauth_token_secret=hdhd0244k9j7ao03&
     oauth_callback_confirmed=true

   The client redirects Jane's user-agent to the server's Resource Owner
   Authorization endpoint to obtain Jane's approval for accessing her
   private photos:

     https://photos.example.net/authorize?oauth_token=hh5s93j4hdidpola

   The server requests Jane to sign in using her username and password
   and if successful, asks her to approve granting 'printer.example.com'
   access to her private photos. Jane approves the request and her
   user-agent is redirected to the callback URI provided by the client
   in the previous request (line breaks are for display purposes only):

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

The callback request informs the client that Jane completed the
authorization process. The client then requests a set of token
credentials using its temporary credentials (over a secure Transport
Layer Security (TLS) channel):

```
POST /token HTTP/1.1
Host: photos.example.net
Authorization: OAuth realm="Photos",
   oauth_consumer_key="dpf43f3p2l4k3l03",
   oauth_token="hh5s93j4hdidpola",
   oauth_signature_method="HMAC-SHA1",
   oauth_timestamp="137131201",
   oauth_nonce="walatlh",
   oauth_verifier="hfdp7dh39dks9884",
   oauth_signature="gKgrFCywp7rO0OXSjdot%2FIHF7IU%3D"
```

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

```
HTTP/1.1 200 OK
Content-Type: application/x-www-form-urlencoded

oauth_token=nnch734d00sl2jdk&oauth_token_secret=pfkkdhi9sl3r4s00
```

With a set of token credentials, the client is now ready to request
the private photo:

```
GET /photos?file=vacation.jpg&size=original HTTP/1.1
Host: photos.example.net
Authorization: OAuth realm="Photos",
   oauth_consumer_key="dpf43f3p2l4k3l03",
   oauth_token="nnch734d00sl2jdk",
   oauth_signature_method="HMAC-SHA1",
   oauth_timestamp="137131202",
   oauth_nonce="chapoH",
   oauth_signature="MdpQcU8iPSUjWoN%2FUDMsK2sui9I%3D"
```

The 'photos.example.net' server validates the request and responds
with the requested photo. 'printer.example.com' is able to continue
accessing Jane's private photos using the same set of token
credentials for the duration of Jane's authorization, or until Jane
revokes access.

1.3. 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].

2. Redirection-Based Authorization

   OAuth uses tokens 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 (usually using a
   username and password).

   There are many ways in which a server 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
       (in the form of an identifier and shared-secret). The temporary
       credentials are used to identify the access request throughout
       the authorization process.

   2. The resource owner authorizes the server to grant the client's
       access request (identified by 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 server MUST revoke the temporary credentials 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 the following three endpoints:

   Temporary Credential Request
        The endpoint used by the client to obtain a set of temporary
        credentials as described in Section 2.1.

   Resource Owner Authorization
        The endpoint to which the resource owner is redirected to grant
        authorization as described in Section 2.2.

   Token Request
        The endpoint used by the client to request a set of token
        credentials using the set of temporary credentials as described
        in Section 2.3.

The three URIs advertised by the server 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, to
avoid conflicts with the protocol parameters added to the URIs when
used.

The methods in which the server advertises and documents its three
endpoints are beyond the scope of this specification. Clients should
avoid making assumptions about the size of tokens and other server-
generated values, which are left undefined by this specification. In
addition, protocol parameters MAY include values that require
encoding when transmitted. Clients and servers should not make
assumptions about the possible range of their values.

2.1. Temporary Credentials

The client obtains a set of temporary credentials from the server by
making an authenticated (Section 3) HTTP "POST" request to the
Temporary Credential Request endpoint (unless the server advertises
another HTTP request method for the client to use). The client
constructs a request URI by adding the following REQUIRED parameter
to the request (in addition to the other protocol parameters, using
the same parameter transmission method):

oauth_callback: An absolute URI back to which the server will
                 redirect the resource owner when the Resource Owner
                 Authorization step (Section 2.2) 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 MAY omit the empty "oauth_token"
protocol parameter from the request and MUST use the empty string as
the token secret value.

Since the request results in the transmission of plain text
credentials in the HTTP response, the server MUST require the use of
a transport-layer mechanisms such as TLS or Secure Socket Layer (SSL)
(or a secure channel with equivalent protections).

For example, the client makes the following HTTPS request:

```
POST /request_temp_credentials HTTP/1.1
Host: server.example.com
Authorization: OAuth realm="Example",
    oauth_consumer_key="jd83jd92dhsh93js",
    oauth_signature_method="PLAINTEXT",
    oauth_callback="http%3A%2F%2Fclient.example.net%2Fcb%3Fx%3D1",
    oauth_signature="ja893SD9%26"
```

The server MUST verify (Section 3.2) the request and if valid,
respond back to the client with a set of temporary credentials (in
the form of an identifier and shared-secret). 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] with a 200 status code (OK).

The response contains the following REQUIRED 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 parameter is used to
      differentiate from previous versions of the protocol.

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):

```
HTTP/1.1 200 OK
Content-Type: application/x-www-form-urlencoded

oauth_token=hdk48Djdsa&oauth_token_secret=xyz4992k83j47x0b&
oauth_callback_confirmed=true
```

2.2. 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 REQUIRED
query parameter to the Resource Owner Authorization endpoint URI:

oauth_token
      The temporary credentials identifier obtained in Section 2.1 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 directs 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.

For example, the client redirects the resource owner's user-agent to
make the following HTTPS request:

  GET /authorize_access?oauth_token=hdk48Djdsa HTTP/1.1
  Host: server.example.com

The way in which the server handles the authorization request,
including whether it uses a secure channel such as TLS/SSL 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 REQUIRED parameters to the callback URI query component:

oauth_token
      The temporary credentials identifier received from the client.

   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, the server redirects the resource owner's user-agent to
   make the following HTTP request:

     GET /cb?x=1&oauth_token=hdk48Djdsa&oauth_verifier=473f82d3 HTTP/1.1
     Host: client.example.net

   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.

2.3. Token Credentials

   The client obtains a set of token credentials from the server by
   making an authenticated (Section 3) HTTP "POST" request to the Token
   Request endpoint (unless the server advertises another HTTP request
   method for the client to use). The client constructs a request URI
   by adding the following REQUIRED parameter to the request (in
   addition to the other protocol parameters, using the same parameter
   transmission method):

   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 substitute for token credentials in the
   authenticated request and transmitted using the "oauth_token"
   parameter.

   Since the request results in the transmission of plain text
   credentials in the HTTP response, the server MUST require the use of
   a transport-layer mechanism such as TLS or SSL (or a secure channel
   with equivalent protections).

For example, the client makes the following HTTPS request:

```
POST /request_token HTTP/1.1
Host: server.example.com
Authorization: OAuth realm="Example",
    oauth_consumer_key="jd83jd92dhsh93js",
    oauth_token="hdk48Djdsa",
    oauth_signature_method="PLAINTEXT",
    oauth_verifier="473f82d3",
    oauth_signature="ja893SD9%26xyz4992k83j47x0b"
```

The server MUST verify (Section 3.2) the validity of the request,
ensure that the resource owner has authorized the provisioning of
token credentials to the client, and ensure that the temporary
credentials have not expired or been 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] with a 200 status code (OK).

The response contains the following REQUIRED parameters:

oauth_token
      The token identifier.

oauth_token_secret
      The token shared-secret.

For example:

```
HTTP/1.1 200 OK
Content-Type: application/x-www-form-urlencoded

oauth_token=j49ddk933skd9dks&oauth_token_secret=ll399dj47dskfjdk
```

The server must retain the scope, duration, and other attributes
approved by the resource owner, and enforce these restrictions when
receiving a client request made with the token credentials issued.

Once the client receives and stores the token credentials, it can
proceed to access protected resources on behalf of the resource owner
by making authenticated requests (Section 3) using the client
credentials together with the token credentials received.

3. Authenticated Requests

   The HTTP authentication methods defined by [RFC2617] enable clients
   to make authenticated HTTP requests. Clients using these methods
   gain access to protected resources by using their credentials
   (typically, a username and password pair), which allow the server to
   verify their authenticity. Using these methods for delegation
   requires the client to assume the role of the resource owner.

   OAuth provides a method designed to include two sets of credentials
   with each request, one to identify the client, and another to
   identify the resource owner. Before a client can make authenticated
   requests on behalf of the resource owner, it must obtain a token
   authorized by the resource owner. Section 2 provides one such method
   through which the client can obtain a token authorized by the
   resource owner.

   The client credentials take the form of a unique identifier and an
   associated shared-secret or RSA key pair. Prior to making
   authenticated requests, the client establishes a set of credentials
   with the server. The process and requirements for provisioning these
   are outside the scope of this specification. Implementers are urged
   to consider the security ramifications of using client credentials,
   some of which are described in Section 4.6.

   Making authenticated requests requires prior knowledge of the
   server's configuration. OAuth includes multiple methods for
   transmitting protocol parameters with requests (Section 3.5), as well
   as multiple methods for the client to prove its rightful ownership of
   the credentials used (Section 3.4). The way in which clients
   discover the required configuration is outside the scope of this
   specification.

3.1. Making Requests

   An authenticated request includes several protocol parameters. Each
   parameter name begins with the "oauth_" prefix, and the parameter
   names and values are case sensitive. Clients make authenticated
   requests by calculating the values of a set of protocol parameters
   and adding them to the HTTP request as follows:

   1. The client assigns value to each of these REQUIRED (unless
       specified otherwise) protocol parameters:

oauth_consumer_key
   The identifier portion of the client credentials (equivalent to
   a username). The parameter name reflects a deprecated term
   (Consumer Key) used in previous revisions of the specification,
   and has been retained to maintain backward compatibility.

oauth_token
   The token value used to associate the request with the resource
   owner. If the request is not associated with a resource owner
   (no token available), clients MAY omit the parameter.

oauth_signature_method
   The name of the signature method used by the client to sign the
   request, as defined in Section 3.4.

oauth_timestamp
   The timestamp value as defined in Section 3.3. The parameter
   MAY be omitted when using the "PLAINTEXT" signature method.

oauth_nonce
   The nonce value as defined in Section 3.3. The parameter MAY
   be omitted when using the "PLAINTEXT" signature method.

oauth_version
   OPTIONAL. If present, MUST be set to "1.0". Provides the
   version of the authentication process as defined in this
   specification.

2. The protocol parameters are added to the request using one of the
    transmission methods listed in Section 3.5. Each parameter MUST
    NOT appear more than once per request.

3. The client calculates and assigns the value of the
    "oauth_signature" parameter as described in Section 3.4 and adds
    the parameter to the request using the same method as in the
    previous step.

4. The client sends the authenticated HTTP request to the server.

For example, to make the following HTTP request authenticated (the
"c2&a3=2+q" string in the following examples is used to illustrate
the impact of a form-encoded entity-body):

```
POST /request?b5=%3D%253D&a3=a&c%40=&a2=r%20b HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded

c2&a3=2+q
```

   The client assigns values to the following protocol parameters using
   its client credentials, token credentials, the current timestamp, a
   uniquely generated nonce, and indicates that it will use the
   "HMAC-SHA1" signature method:

     oauth_consumer_key: 9djdj82h48djs9d2
     oauth_token: kkk9d7dh3k39sjv7
     oauth_signature_method: HMAC-SHA1
     oauth_timestamp: 137131201
     oauth_nonce: 7d8f3e4a

   The client adds the protocol parameters to the request using the
   OAuth HTTP "Authorization" header field:

     Authorization: OAuth realm="Example",
                    oauth_consumer_key="9djdj82h48djs9d2",
                    oauth_token="kkk9d7dh3k39sjv7",
                    oauth_signature_method="HMAC-SHA1",
                    oauth_timestamp="137131201",
                    oauth_nonce="7d8f3e4a"

   Then, it calculates the value of the "oauth_signature" parameter
   (using client secret "j49sk3j29djd" and token secret "dh893hdasih9"),
   adds it to the request, and sends the HTTP request to the server:

     POST /request?b5=%3D%253D&a3=a&c%40=&a2=r%20b HTTP/1.1
     Host: example.com
     Content-Type: application/x-www-form-urlencoded
     Authorization: OAuth realm="Example",
                    oauth_consumer_key="9djdj82h48djs9d2",
                    oauth_token="kkk9d7dh3k39sjv7",
                    oauth_signature_method="HMAC-SHA1",
                    oauth_timestamp="137131201",
                    oauth_nonce="7d8f3e4a",
                    oauth_signature="bYT5CMsGcbgUdFHObYMEfcx6bsw%3D"

     c2&a3=2+q

3.2. Verifying Requests

   Servers receiving an authenticated request MUST validate it by:

   o Recalculating the request signature independently as described in
      Section 3.4 and comparing it to the value received from the client
      via the "oauth_signature" parameter.

o If using the "HMAC-SHA1" or "RSA-SHA1" signature methods, ensuring
   that the combination of nonce/timestamp/token (if present)
   received from the client has not been used before in a previous
   request (the server MAY reject requests with stale timestamps as
   described in Section 3.3).

o If a token is present, verifying the scope and status of the
   client authorization as represented by the token (the server MAY
   choose to restrict token usage to the client to which it was
   issued).

o If the "oauth_version" parameter is present, ensuring its value is
   "1.0".

If the request fails verification, the server SHOULD respond with the
appropriate HTTP response status code. The server MAY include
further details about why the request was rejected in the response
body.

The server SHOULD return a 400 (Bad Request) status code when
receiving a request with unsupported parameters, an unsupported
signature method, missing parameters, or duplicated protocol
parameters. The server SHOULD return a 401 (Unauthorized) status
code when receiving a request with invalid client credentials, an
invalid or expired token, an invalid signature, or an invalid or used
nonce.

3.3. Nonce and Timestamp

The timestamp value MUST be a positive integer. Unless otherwise
specified by the server's documentation, the timestamp is expressed
in the number of seconds since January 1, 1970 00:00:00 GMT.

A nonce is a random string, uniquely generated by the client to allow
the server to verify that a request has never been made before and
helps prevent replay attacks when requests are made over a non-secure
channel. The nonce value MUST be unique across all requests with the
same timestamp, client credentials, and token combinations.

To avoid the need to retain an infinite number of nonce values for
future checks, servers MAY choose to restrict the time period after
which a request with an old timestamp is rejected. Note that this
restriction implies a level of synchronization between the client's
and server's clocks. Servers applying such a restriction MAY provide
a way for the client to sync with the server's clock; alternatively,
both systems could synchronize with a trusted time service. Details
of clock synchronization strategies are beyond the scope of this
specification.

3.4. Signature

   OAuth-authenticated requests can have two sets of credentials: those
   passed via the "oauth_consumer_key" parameter and those in the
   "oauth_token" parameter. In order for the server to verify the
   authenticity of the request and prevent unauthorized access, the
   client needs to prove that it is the rightful owner of the
   credentials. This is accomplished using the shared-secret (or RSA
   key) part of each set of credentials.

   OAuth provides three methods for the client to prove its rightful
   ownership of the credentials: "HMAC-SHA1", "RSA-SHA1", and
   "PLAINTEXT". These methods are generally referred to as signature
   methods, even though "PLAINTEXT" does not involve a signature. In
   addition, "RSA-SHA1" utilizes an RSA key instead of the shared-
   secrets associated with the client credentials.

   OAuth does not mandate a particular signature method, as each
   implementation can have its own unique requirements. Servers are
   free to implement and document their own custom methods.
   Recommending any particular method is beyond the scope of this
   specification. Implementers should review the Security
   Considerations section (Section 4) before deciding on which method to
   support.

   The client declares which signature method is used via the
   "oauth_signature_method" parameter. It then generates a signature
   (or a string of an equivalent value) and includes it in the
   "oauth_signature" parameter. The server verifies the signature as
   specified for each method.

   The signature process does not change the request or its parameters,
   with the exception of the "oauth_signature" parameter.

3.4.1. Signature Base String

   The signature base string is a consistent, reproducible concatenation
   of several of the HTTP request elements into a single string. The
   string is used as an input to the "HMAC-SHA1" and "RSA-SHA1"
   signature methods.

   The signature base string includes the following components of the
   HTTP request:

   o The HTTP request method (e.g., "GET", "POST", etc.).

   o The authority as declared by the HTTP "Host" request header field.

   o The path and query components of the request resource URI.

   o The protocol parameters excluding the "oauth_signature".

   o Parameters included in the request entity-body if they comply with
      the strict restrictions defined in Section 3.4.1.3.

   The signature base string does not cover the entire HTTP request.
   Most notably, it does not include the entity-body in most requests,
   nor does it include most HTTP entity-headers. It is important to
   note that the server cannot verify the authenticity of the excluded
   request components without using additional protections such as SSL/
   TLS or other methods.

3.4.1.1. String Construction

   The signature base string is constructed by concatenating together,
   in order, the following HTTP request elements:

   1. The HTTP request method in uppercase. For example: "HEAD",
       "GET", "POST", etc. If the request uses a custom HTTP method, it
       MUST be encoded (Section 3.6).

   2. An "&" character (ASCII code 38).

   3. The base string URI from Section 3.4.1.2, after being encoded
       (Section 3.6).

   4. An "&" character (ASCII code 38).

   5. The request parameters as normalized in Section 3.4.1.3.2, after
       being encoded (Section 3.6).

   For example, the HTTP request:

     POST /request?b5=%3D%253D&a3=a&c%40=&a2=r%20b HTTP/1.1
     Host: example.com
     Content-Type: application/x-www-form-urlencoded
     Authorization: OAuth realm="Example",
                    oauth_consumer_key="9djdj82h48djs9d2",
                    oauth_token="kkk9d7dh3k39sjv7",
                    oauth_signature_method="HMAC-SHA1",
                    oauth_timestamp="137131201",
                    oauth_nonce="7d8f3e4a",
                    oauth_signature="bYT5CMsGcbgUdFHObYMEfcx6bsw%3D"

     c2&a3=2+q

   is represented by the following signature base string (line breaks
   are for display purposes only):

     POST&http%3A%2F%2Fexample.com%2Frequest&a2%3Dr%2520b%26a3%3D2%2520q
     %26a3%3Da%26b5%3D%253D%25253D%26c%2540%3D%26c2%3D%26oauth_consumer_
     key%3D9djdj82h48djs9d2%26oauth_nonce%3D7d8f3e4a%26oauth_signature_m
     ethod%3DHMAC-SHA1%26oauth_timestamp%3D137131201%26oauth_token%3Dkkk
     9d7dh3k39sjv7

3.4.1.2. Base String URI

   The scheme, authority, and path of the request resource URI [RFC3986]
   are included by constructing an "http" or "https" URI representing
   the request resource (without the query or fragment) as follows:

   1. The scheme and host MUST be in lowercase.

   2. The host and port values MUST match the content of the HTTP
       request "Host" header field.

   3. The port MUST be included if it is not the default port for the
       scheme, and MUST be excluded if it is the default. Specifically,
       the port MUST be excluded when making an HTTP request [RFC2616]
       to port 80 or when making an HTTPS request [RFC2818] to port 443.
       All other non-default port numbers MUST be included.

   For example, the HTTP request:

     GET /r%20v/X?id=123 HTTP/1.1
     Host: EXAMPLE.COM:80

   is represented by the base string URI: "http://example.com/r%20v/X".

   In another example, the HTTPS request:

     GET /?q=1 HTTP/1.1
     Host: www.example.net:8080

   is represented by the base string URI:
   "https://www.example.net:8080/".

3.4.1.3. Request Parameters

   In order to guarantee a consistent and reproducible representation of
   the request parameters, the parameters are collected and decoded to
   their original decoded form. They are then sorted and encoded in a
   particular manner that is often different from their original
   encoding scheme, and concatenated into a single string.

3.4.1.3.1. Parameter Sources

   The parameters from the following sources are collected into a single
   list of name/value pairs:

   o The query component of the HTTP request URI as defined by
      [RFC3986], Section 3.4. The query component is parsed into a list
      of name/value pairs by treating it as an
      "application/x-www-form-urlencoded" string, separating the names
      and values and decoding them as defined by
      [W3C.REC-html40-19980424], Section 17.13.4.

   o The OAuth HTTP "Authorization" header field (Section 3.5.1) if
      present. The header's content is parsed into a list of name/value
      pairs excluding the "realm" parameter if present. The parameter
      values are decoded as defined by Section 3.5.1.

   o The HTTP request entity-body, but only if all of the following
      conditions are met:

      * The entity-body is single-part.

      * The entity-body follows the encoding requirements of the
         "application/x-www-form-urlencoded" content-type as defined by
         [W3C.REC-html40-19980424].

      * The HTTP request entity-header includes the "Content-Type"
         header field set to "application/x-www-form-urlencoded".

      The entity-body is parsed into a list of decoded name/value pairs
      as described in [W3C.REC-html40-19980424], Section 17.13.4.

   The "oauth_signature" parameter MUST be excluded from the signature
   base string if present. Parameters not explicitly included in the
   request MUST be excluded from the signature base string (e.g., the
   "oauth_version" parameter when omitted).

For example, the HTTP request:

```
POST /request?b5=%3D%253D&a3=a&c%40=&a2=r%20b HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Authorization: OAuth realm="Example",
               oauth_consumer_key="9djdj82h48djs9d2",
               oauth_token="kkk9d7dh3k39sjv7",
               oauth_signature_method="HMAC-SHA1",
               oauth_timestamp="137131201",
               oauth_nonce="7d8f3e4a",
               oauth_signature="djosJKDKJSD8743243%2Fjdk33klY%3D"

c2&a3=2+q
```

contains the following (fully decoded) parameters used in the
signature base sting:

Name	Value
b5	=%3D
a3	a
c@	
a2	r b
oauth_consumer_key	9djdj82h48djs9d2
oauth_token	kkk9d7dh3k39sjv7
oauth_signature_method	HMAC-SHA1
oauth_timestamp	137131201
oauth_nonce	7d8f3e4a
c2	
a3	2 q

Note that the value of "b5" is "=%3D" and not "==". Both "c@" and
"c2" have empty values. While the encoding rules specified in this
specification for the purpose of constructing the signature base
string exclude the use of a "+" character (ASCII code 43) to
represent an encoded space character (ASCII code 32), this practice
is widely used in "application/x-www-form-urlencoded" encoded values,
and MUST be properly decoded, as demonstrated by one of the "a3"
parameter instances (the "a3" parameter is used twice in this
request).

3.4.1.3.2. Parameters Normalization

   The parameters collected in Section 3.4.1.3 are normalized into a
   single string as follows:

   1. First, the name and value of each parameter are encoded
       (Section 3.6).

   2. The parameters are sorted by name, using ascending byte value
       ordering. If two or more parameters share the same name, they
       are sorted by their value.

   3. The name of each parameter is concatenated to its corresponding
       value using an "=" character (ASCII code 61) as a separator, even
       if the value is empty.

   4. The sorted name/value pairs are concatenated together into a
       single string by using an "&" character (ASCII code 38) as
       separator.

   For example, the list of parameters from the previous section would
   be normalized as follows:

                             Encoded:

         +------------------------+------------------+
         | Name | Value |
         +------------------------+------------------+
         | b5 | %3D%253D |
         | a3 | a |
         | c%40 | |
         | a2 | r%20b |
         | oauth_consumer_key | 9djdj82h48djs9d2 |
         | oauth_token | kkk9d7dh3k39sjv7 |
         | oauth_signature_method| HMAC-SHA1 |
         | oauth_timestamp | 137131201 |
         | oauth_nonce | 7d8f3e4a |
         | c2 | |
         | a3 | 2%20q |
         +------------------------+------------------+

Sorted:

```
+------------------------+------------------+
|          Name          |      Value       |
+------------------------+------------------+
|           a2           |      r%20b       |
|           a3           |      2%20q       |
|           a3           |        a         |
|           b5           |     %3D%253D     |
|          c%40          |                  |
|           c2           |                  |
|    oauth_consumer_key  |  9djdj82h48djs9d2 |
|       oauth_nonce      |     7d8f3e4a     |
|  oauth_signature_method |    HMAC-SHA1     |
|     oauth_timestamp    |     137131201    |
|       oauth_token      | kkk9d7dh3k39sjv7 |
+------------------------+------------------+
```

Concatenated Pairs:

```
+-----------------------------------+
|             Name=Value            |
+-----------------------------------+
|             a2=r%20b              |
|             a3=2%20q              |
|               a3=a                |
|            b5=%3D%253D            |
|              c%40=                |
|               c2=                 |
|  oauth_consumer_key=9djdj82h48djs9d2 |
|        oauth_nonce=7d8f3e4a       |
|   oauth_signature_method=HMAC-SHA1 |
|     oauth_timestamp=137131201     |
|    oauth_token=kkk9d7dh3k39sjv7   |
+-----------------------------------+
```

and concatenated together into a single string (line breaks are for
display purposes only):

```
a2=r%20b&a3=2%20q&a3=a&b5=%3D%253D&c%40=&c2=&oauth_consumer_key=9dj
dj82h48djs9d2&oauth_nonce=7d8f3e4a&oauth_signature_method=HMAC-SHA1
&oauth_timestamp=137131201&oauth_token=kkk9d7dh3k39sjv7
```

3.4.2. HMAC-SHA1

   The "HMAC-SHA1" signature method uses the HMAC-SHA1 signature
   algorithm as defined in [RFC2104]:

     digest = HMAC-SHA1 (key, text)

   The HMAC-SHA1 function variables are used in following way:

   text   is set to the value of the signature base string from
           Section 3.4.1.1.

   key   is set to the concatenated values of:

           1.   The client shared-secret, after being encoded
                (Section 3.6).

           2.   An "&" character (ASCII code 38), which MUST be included
                even when either secret is empty.

           3.   The token shared-secret, after being encoded
                (Section 3.6).

   digest is used to set the value of the "oauth_signature" protocol
           parameter, after the result octet string is base64-encoded
           per [RFC2045], Section 6.8.

3.4.3. RSA-SHA1

   The "RSA-SHA1" signature method uses the RSASSA-PKCS1-v1_5 signature
   algorithm as defined in [RFC3447], Section 8.2 (also known as
   PKCS#1), using SHA-1 as the hash function for EMSA-PKCS1-v1_5. To
   use this method, the client MUST have established client credentials
   with the server that included its RSA public key (in a manner that is
   beyond the scope of this specification).

   The signature base string is signed using the client's RSA private
   key per [RFC3447], Section 8.2.1:

     S = RSASSA-PKCS1-V1_5-SIGN (K, M)

   Where:

   K   is set to the client's RSA private key,

   M   is set to the value of the signature base string from
           Section 3.4.1.1, and

   S   is the result signature used to set the value of the
          "oauth_signature" protocol parameter, after the result octet
          string is base64-encoded per [RFC2045] section 6.8.

   The server verifies the signature per [RFC3447] section 8.2.2:

     RSASSA-PKCS1-V1_5-VERIFY ((n, e), M, S)

   Where:

   (n, e) is set to the client's RSA public key,

   M   is set to the value of the signature base string from
          Section 3.4.1.1, and

   S   is set to the octet string value of the "oauth_signature"
          protocol parameter received from the client.

3.4.4. PLAINTEXT

   The "PLAINTEXT" method does not employ a signature algorithm. It
   MUST be used with a transport-layer mechanism such as TLS or SSL (or
   sent over a secure channel with equivalent protections). It does not
   utilize the signature base string or the "oauth_timestamp" and
   "oauth_nonce" parameters.

   The "oauth_signature" protocol parameter is set to the concatenated
   value of:

   1. The client shared-secret, after being encoded (Section 3.6).

   2. An "&" character (ASCII code 38), which MUST be included even
       when either secret is empty.

   3. The token shared-secret, after being encoded (Section 3.6).

3.5. Parameter Transmission

   When making an OAuth-authenticated request, protocol parameters as
   well as any other parameter using the "oauth_" prefix SHALL be
   included in the request using one and only one of the following
   locations, listed in order of decreasing preference:

   1. The HTTP "Authorization" header field as described in
       Section 3.5.1.

   2. The HTTP request entity-body as described in Section 3.5.2.

   3. The HTTP request URI query as described in Section 3.5.3.

   In addition to these three methods, future extensions MAY define
   other methods for including protocol parameters in the request.

3.5.1. Authorization Header

   Protocol parameters can be transmitted using the HTTP "Authorization"
   header field as defined by [RFC2617] with the auth-scheme name set to
   "OAuth" (case insensitive).

   For example:

     Authorization: OAuth realm="Example",
        oauth_consumer_key="0685bd9184jfhq22",
        oauth_token="ad180jjd733klru7",
        oauth_signature_method="HMAC-SHA1",
        oauth_signature="wOJIO9A2W5mFwDgiDvZbTSMK%2FPY%3D",
        oauth_timestamp="137131200",
        oauth_nonce="4572616e48616d6d65724c61686176",
        oauth_version="1.0"

   Protocol parameters SHALL be included in the "Authorization" header
   field as follows:

   1. Parameter names and values are encoded per Parameter Encoding
       (Section 3.6).

   2. Each parameter's name is immediately followed by an "=" character
       (ASCII code 61), a """ character (ASCII code 34), the parameter
       value (MAY be empty), and another """ character (ASCII code 34).

   3. Parameters are separated by a "," character (ASCII code 44) and
       OPTIONAL linear whitespace per [RFC2617].

   4. The OPTIONAL "realm" parameter MAY be added and interpreted per
       [RFC2617] section 1.2.

   Servers MAY indicate their support for the "OAuth" auth-scheme by
   returning the HTTP "WWW-Authenticate" response header field upon
   client requests for protected resources. As per [RFC2617], such a
   response MAY include additional HTTP "WWW-Authenticate" header
   fields:

   For example:

     WWW-Authenticate: OAuth realm="http://server.example.com/"

The realm parameter defines a protection realm per [RFC2617], Section 1.2.

3.5.2. Form-Encoded Body

   Protocol parameters can be transmitted in the HTTP request entity-
   body, but only if the following REQUIRED conditions are met:

   o The entity-body is single-part.

   o The entity-body follows the encoding requirements of the
      "application/x-www-form-urlencoded" content-type as defined by
      [W3C.REC-html40-19980424].

   o The HTTP request entity-header includes the "Content-Type" header
      field set to "application/x-www-form-urlencoded".

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

     oauth_consumer_key=0685bd9184jfhq22&oauth_token=ad180jjd733klr
     u7&oauth_signature_method=HMAC-SHA1&oauth_signature=wOJIO9A2W5
     mFwDgiDvZbTSMK%2FPY%3D&oauth_timestamp=137131200&oauth_nonce=4
     572616e48616d6d65724c61686176&oauth_version=1.0

   The entity-body MAY include other request-specific parameters, in
   which case, the protocol parameters SHOULD be appended following the
   request-specific parameters, properly separated by an "&" character
   (ASCII code 38).

3.5.3. Request URI Query

   Protocol parameters can be transmitted by being added to the HTTP
   request URI as a query parameter as defined by [RFC3986], Section 3.

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

     GET /example/path?oauth_consumer_key=0685bd9184jfhq22&
     oauth_token=ad180jjd733klru7&oauth_signature_method=HM
     AC-SHA1&oauth_signature=wOJIO9A2W5mFwDgiDvZbTSMK%2FPY%
     3D&oauth_timestamp=137131200&oauth_nonce=4572616e48616
     d6d65724c61686176&oauth_version=1.0 HTTP/1.1

   The request URI MAY include other request-specific query parameters,
   in which case, the protocol parameters SHOULD be appended following
   the request-specific parameters, properly separated by an "&"
   character (ASCII code 38).

3.6. Percent Encoding

   Existing percent-encoding methods do not guarantee a consistent
   construction of the signature base string. The following percent-
   encoding method is not defined to replace the existing encoding
   methods defined by [RFC3986] and [W3C.REC-html40-19980424]. It is
   used only in the construction of the signature base string and the
   "Authorization" header field.

   This specification defines the following method for percent-encoding
   strings:

   1. Text values are first encoded as UTF-8 octets per [RFC3629] if
       they are not already. This does not include binary values that
       are not intended for human consumption.

   2. The values are then escaped using the [RFC3986] percent-encoding
       (%XX) mechanism as follows:

       * Characters in the unreserved character set as defined by
          [RFC3986], Section 2.3 (ALPHA, DIGIT, "-", ".", "_", "~") MUST
          NOT be encoded.

       * All other characters MUST be encoded.

       * The two hexadecimal characters used to represent encoded
          characters MUST be uppercase.

   This method is different from the encoding scheme used by the
   "application/x-www-form-urlencoded" content-type (for example, it
   encodes space characters as "%20" and not using the "+" character).
   It MAY be different from the percent-encoding functions provided by
   web-development frameworks (e.g., encode different characters, use
   lowercase hexadecimal characters).

4. 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.

4.1. RSA-SHA1 Signature Method

   Authenticated requests made with "RSA-SHA1" signatures do not use the
   token shared-secret, or any provisioned client shared-secret. This
   means the request relies completely on the secrecy of the private key
   used by the client to sign requests.

4.2. Confidentiality of Requests

   While this protocol provides a mechanism for verifying the integrity
   of requests, it provides no guarantee of request confidentiality.
   Unless further precautions are taken, eavesdroppers will have full
   access to request content. Servers should carefully consider the
   kinds of data likely to be sent as part of such requests, and should
   employ transport-layer security mechanisms to protect sensitive
   resources.

4.3. Spoofing by Counterfeit Servers

   This protocol makes no attempt to verify the authenticity of the
   server. A hostile party could take advantage of this by intercepting
   the client's requests and returning misleading or otherwise incorrect
   responses. Service providers should consider such attacks when
   developing services using this protocol, and should require
   transport-layer security for any requests where the authenticity of
   the server or of request responses is an issue.

4.4. Proxying and Caching of Authenticated Content

   The HTTP Authorization scheme (Section 3.5.1) is optional. However,
   [RFC2616] relies on the "Authorization" and "WWW-Authenticate" header
   fields to distinguish authenticated content so that it can be
   protected. Proxies and caches, in particular, may fail to adequately
   protect requests not using these header fields.

   For example, private authenticated content may be stored in (and thus
   retrievable from) publicly accessible caches. Servers not using the
   HTTP "Authorization" header field should take care to use other
   mechanisms, such as the "Cache-Control" header field, to ensure that
   authenticated content is protected.

4.5. Plaintext Storage of Credentials

   The client shared-secret and token shared-secret function the same
   way passwords do in traditional authentication systems. In order to
   compute the signatures used in methods other than "RSA-SHA1", the
   server must have access to these secrets in plaintext form. This is
   in contrast, for example, to modern operating systems, which store
   only a one-way hash of user credentials.

   If an attacker were to gain access to these secrets -- or worse, to
   the server's database of all such secrets -- he or she would be able
   to perform any action on behalf of any resource owner. Accordingly,
   it is critical that servers protect these secrets from unauthorized
   access.

4.6. Secrecy of the Client Credentials

   In many cases, the client application will be under the control of
   potentially untrusted parties. For example, if the client is a
   desktop application with freely available source code or an
   executable binary, an attacker may be able to download a copy for
   analysis. In such cases, attackers will be able to recover the
   client credentials.

   Accordingly, servers should not use the client credentials alone to
   verify the identity of the client. Where possible, other factors
   such as IP address should be used as well.

4.7. Phishing Attacks

   Wide deployment of this 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.
   Client developers should consider the security implications of how
   they interact with a user-agent (e.g., separate window, embedded),
   and the ability of the end-user to verify the authenticity of the
   server website.

4.8. Scoping of Access Requests

   By itself, this protocol 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 this protocol, 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.

4.9. Entropy of Secrets

   Unless a transport-layer security protocol is used, eavesdroppers
   will have full access to authenticated 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
   that 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 that nevertheless exhibit patterns or
   other weaknesses that make cryptanalysis or brute force attacks
   easier. Implementers should be careful to use cryptographically
   secure PRNGs to avoid these problems.

4.10. Denial-of-Service / Resource-Exhaustion Attacks

   This specification includes a number of features that may make
   resource exhaustion attacks against servers possible. For example,
   this protocol 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, this protocol 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 this
   specification. However, implementers should be careful to consider
   the additional avenues of attack that this protocol 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 this protocol, servers should
   consider which of these presents a more serious risk for their
   application and design accordingly.

4.11. SHA-1 Cryptographic Attacks

   SHA-1, the hash algorithm used in "HMAC-SHA1" and "RSA-SHA1"
   signature methods, has been shown to have a number of cryptographic
   weaknesses that significantly reduce its resistance to collision
   attacks. While these weaknesses do not seem to affect the use of
   SHA-1 with the Hash-based Message Authentication Code (HMAC) and
   should not affect the "HMAC-SHA1" signature method, it may affect the
   use of the "RSA-SHA1" signature method. NIST has announced that it
   will phase out use of SHA-1 in digital signatures by 2010
   [NIST_SHA-1Comments].

   Practically speaking, these weaknesses are difficult to exploit, and
   by themselves do not pose a significant risk to users of this
   protocol. They may, however, make more efficient attacks possible,
   and servers should take this into account when considering whether
   SHA-1 provides an adequate level of security for their applications.

4.12. Signature Base String Limitations

   The signature base string has been designed to support the signature
   methods defined in this specification. Those designing additional
   signature methods, should evaluated the compatibility of the
   signature base string with their security requirements.

   Since the signature base string does not cover the entire HTTP
   request, such as most request entity-body, most entity-headers, and
   the order in which parameters are sent, servers should employ
   additional mechanisms to protect such elements.

4.13. 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 authorization 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 the protocol authorization endpoints.

   CSRF attacks on OAuth callback URIs hosted by clients 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. The methods for
   preventing such CSRF attacks are beyond the scope of this
   specification.

4.14. User Interface Redress

   Servers should protect the authorization process against user
   interface (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 using
   the following techniques:

   o JavaScript frame busting.

   o JavaScript frame busting, and requiring that browsers have
      JavaScript enabled on the authorization page.

   o Browser-specific anti-framing techniques.

   o Requiring password reentry before issuing OAuth tokens.

4.15. Automatic Processing of Repeat Authorizations

   Servers may wish to automatically process authorization requests
   (Section 2.2) from clients that 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.

5. Acknowledgments

   This specification is directly based on the OAuth Core 1.0 Revision A
   community specification, which in turn was modeled after existing
   proprietary protocols and best practices that have been independently
   implemented by various companies.

   The community specification was edited by Eran Hammer-Lahav and
   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.

   The editor would like to thank the following individuals for their
   invaluable contribution to the publication of this edition of the
   protocol: Lisa Dusseault, Justin Hart, Avshalom Houri, Chris Messina,
   Mark Nottingham, Tim Polk, Peter Saint-Andre, Joseph Smarr, and Paul
   Walker.

Appendix A. Differences from the Community Edition

   This specification includes the following changes made to the
   original community document [OAuthCore1.0_RevisionA] in order to
   correct mistakes and omissions identified since the document was
   originally published at <http://oauth.net>.

   o Changed using TLS/SSL when sending or requesting plain text
      credentials from SHOULD to MUST. This change affects any use of
      the "PLAINTEXT" signature method, as well as requesting temporary
      credentials (Section 2.1) and obtaining token credentials
      (Section 2.3).

   o Adjusted nonce language to indicate it is unique per token/
      timestamp/client combination.

   o Removed the requirement for timestamps to be equal to or greater
      than the timestamp used in the previous request.

   o Changed the nonce and timestamp parameters to OPTIONAL when using
      the "PLAINTEXT" signature method.

   o Extended signature base string coverage that includes
      "application/x-www-form-urlencoded" entity-body parameters when
      the HTTP method used is other than "POST" and URI query parameters
      when the HTTP method used is other than "GET".

   o Incorporated corrections to the instructions in each signature
      method to encode the signature value before inserting it into the
      "oauth_signature" parameter, removing errors that would have
      caused double-encoded values.

   o Allowed omitting the "oauth_token" parameter when empty.

   o Permitted sending requests for temporary credentials with an empty
      "oauth_token" parameter.

   o Removed the restrictions from defining additional "oauth_"
      parameters.

6. References

6.1. Normative References

   [RFC2045]   Freed, N. and N. Borenstein, "Multipurpose Internet Mail
               Extensions (MIME) Part One: Format of Internet Message
               Bodies", RFC 2045, November 1996.

   [RFC2104]   Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-
               Hashing for Message Authentication", RFC 2104,
               February 1997.

   [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.

   [RFC2818]   Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

   [RFC3447]   Jonsson, J. and B. Kaliski, "Public-Key Cryptography
               Standards (PKCS) #1: RSA Cryptography Specifications
               Version 2.1", RFC 3447, February 2003.

   [RFC3629]   Yergeau, F., "UTF-8, a transformation format of ISO
               10646", STD 63, RFC 3629, November 2003.

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

6.2. Informative References

   [NIST_SHA-1Comments]
             Burr, W., "NIST Comments on Cryptanalytic Attacks on
             SHA-1",
             <http://csrc.nist.gov/groups/ST/hash/statement.html>.

   [OAuthCore1.0_RevisionA]
             OAuth Community, "OAuth Core 1.0 Revision A",
             <http://oauth.net/core/1.0a>.

Author's Address

   Eran Hammer-Lahav (editor)

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