Network Working Group	E. Hammer-Lahav
Internet-Draft	Yahoo!
Intended status: Standards Track	January 22, 2011
Expires: July 26, 2011	

TOC

HTTP Authentication: MAC Authentication
draft-hammer-oauth-v2-mac-token-02

Abstract

This document specifies the HTTP MAC authentication scheme, as well as its OAuth 2.0 binding.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.
Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.
Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as “work in progress.”
This Internet-Draft will expire on July 26, 2011.

Table of Contents

1. Introduction
    1.1. Example
    1.2. Notational Conventions
2. Issuing MAC Credentials
3. Making Requests
    3.1. The "Authorization" Request Header
    3.2. Body Hash
    3.3. Signature
        3.3.1. Normalized Request String
        3.3.2. hmac-sha-1
        3.3.3. hmac-sha-256
4. Verifying Requests
    4.1. The "WWW-Authenticate" Response Header Field
5. Scheme Extensions
6. Use with OAuth 2.0
    6.1. Issuing MAC-Type Access Tokens
7. Security Considerations
    7.1. Secrets Transmission
    7.2. Confidentiality of Requests
    7.3. Spoofing by Counterfeit Servers
    7.4. Plaintext Storage of Credentials
    7.5. Entropy of Secrets
    7.6. Denial of Service / Resource Exhaustion Attacks
    7.7. Coverage Limitations
8. IANA Considerations
    8.1. OAuth Access Token Type Registration
        8.1.1. The "mac" OAuth Access Token Type
    8.2. OAuth Parameters Registration
        8.2.1. The "secret" OAuth Parameter
        8.2.2. The "algorithm" OAuth Parameter
9. Acknowledgments
Appendix A. Document History
10. References
    10.1. Normative References
    10.2. Informative References
§ Author's Address

---

1. Introduction

TOC

This specification defines the HTTP MAC authentication scheme and provides a method for making authenticated HTTP requests with partial cryptographic verification of the request - covering the HTTP method, request URI, host, and in some cases the request body.

This specification uses the terminology defined in [I-D.ietf-oauth-v2] (Hammer-Lahav, E., Recordon, D., and D. Hardt, “The OAuth 2.0 Authorization Protocol,” January 2011.).
Please discuss this draft on the oauth@ietf.org mailing list.

1.1. Example

TOC

The client attempts to access a protected resource without authentication, making the following HTTP request to the resource server:

```
GET /resource/1?b=1&a=2 HTTP/1.1
Host: example.com
```

The resource server returns the following authentication challenge:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: MAC realm="example"
Date: Thu, 02 Dec 2010 21:39:45 GMT
```

The client has previously obtained a set of token credentials for accessing resources on the http://example.com/ resource server. The MAC credentials issued to the client included the following attributes:

> **Access Token:** h480djs93hd8
>
> **Token secret:** 489dks293j39
>
> **MAC algorithm:** hmac-sha-1

The client attempts the HTTP request again, this time using the token credentials issued earlier to authenticate. To construct the authentication header, the client calculates the current timestamp and generates a nonce. The nonce is unique to the timestamp used, typically a random string:

> **Timestamp:** 137131200
>
> **Nonce:** dj83hs9s

The client normalizes the request and constructs the normalized request string (the new line separator character is represented by \n for display purposes only):

```
h480djs93hd8\n
137131200\n
dj83hs9s\n
\n
GET\n
example.com\n
80\n
/resource/1\n
a=2\n
b=1\n
```

The normalized request string is signed using the specified MAC algorithm hmac-sha-1 with the normalized request string as text and the token secret as key. The resulting digest is base64-encoded to produce the request signature:

```
YTVjyNSujYs1WsDurFnvFi4JK6o=
```

The client includes the access token, timestamp, nonce, and signature with the request using the Authorization request header field:

```
GET /resource/1 HTTP/1.1
Host: example.com
Authorization: MAC token="h480djs93hd8",
                   timestamp="137131200",
                   nonce="dj83hs9s",
                   signature="YTVjyNSujYs1WsDurFnvFi4JK6o="
```

The resource server validates the request by calculating the signature again based on the request received and verifies the validity and scope of the access token. If valid, the resource server responds with the requested protected resource representation.

1.2. Notational Conventions

TOC

The key words 'MUST', 'MUST NOT', 'REQUIRED', 'SHALL', 'SHALL NOT', 'SHOULD', 'SHOULD NOT', 'RECOMMENDED', 'MAY', and 'OPTIONAL' in this specification are to be interpreted as described in [RFC2119] (Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels," March 1997.).

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [I-D.ietf-httpbis-p1-messaging] (Fielding, R., Gettys, J., Mogul, J., Nielsen, H., Masinter, L., Leach, P., Berners-Lee, T., and J. Reschke, “HTTP/1.1, part 1: URIs, Connections, and Message Parsing,” October 2009.). Additionally, the following rules are included from [RFC2617]: realm, auth-param.

---

2. Issuing MAC Credentials

TOC

This specification does not define a general purpose method for requesting or issuing MAC credentials (an OAuth 2.0 (Hammer-Lahav, E., Recordon, D., and D. Hardt, “The OAuth 2.0 Authorization Protocol,” January 2011.) [I-D.ietf-oauth-v2] binding is provided in Section 6 (Use with OAuth 2.0)). It simply assumes that the client is in the possession of a set of MAC credentials with the following REQUIRES attributes:

access token A string representing an access authorization issued to the client. The string is usually opaque to the client. Tokens represent specific scopes and durations of access. The token may denote an identifier used to retrieve the authorization information, or self-contain the authorization information in a verifiable manner (i.e. a token string consisting of some data and a signature).

secret A shared symmetric secret used as the MAC algorithm key.

algorithm A MAC algorithm used to calculate the request signature. Value MUST be one of hmac-sha-1, hmac-sha-256, or a registered extension algorithm name as described in Section 5 (Scheme Extensions).

The access token and secret strings MUST NOT include characters other than:

```
DIGIT / ALPHA / %x20-21 / %x23-5B / %x5D-7E
; Any printable ASCII character except for <"> and <\>
```

TOC

3. Making Requests

To make authenticated requests, the client must be in possession of a valid set of MAC credentials accepted by the resource server. The client constructs the request by calculating of a set of attributes, and adding them to the HTTP request using the Authorization header field (The "Authorization" Request Header). Authenticated requests can be sent in response to an authentication challenge or directly.

3.1. The "Authorization" Request Header

TOC

The Authorization request header field uses the framework defined by [RFC2617] (Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication," June 1999.) as follows:

```
credentials    = 'MAC' [ RWS 1#param ]

param          = access-token /
                 timestamp /
                 nonce /
                 body-hash /
                 signature

access-token   = 'token' '=' <"> plain-string <">
timestamp      = 'timestamp' '=' <"> 1*DIGIT <">
nonce          = 'nonce' '=' <"> plain-string <">
body-hash      = 'bodyhash' '=' <"> plain-string <">
signature      = 'signature' '=' <"> plain-string <">

plain-string   = 1*( DIGIT / ALPHA / %x20-21 / %x23-5B / %x5D-7E )
```

The header attributes are defined as follows:

token REQUIRED. The access token string.

timestamp REQUIRED. The current time expressed in the number of seconds since January 1, 1970 00:00:00 GMT, and MUST be a positive integer.

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

combination.
To avoid the need to retain an infinite number of nonce values for future checks, resource servers MAY choose to restrict the time period after which a request with an old timestamp is rejected. Such a restriction implies a level of synchronization between the client's and server's clocks. The client MAY use the Date response header field to synchronize its clock after a failed request.

bodyhash OPTIONAL. The HTTP request entity-body hash as described in Section 3.2 (Body Hash).

signature REQUIRED. The HTTP request signature as described in Section 3.3 (Signature).

Attributes MUST NOT appear more than once. Attribute values are limited to a subset of ASCII, which does not require escaping, as defined by the plain-string ABNF.

3.2. Body Hash

TOC

The body hash is used to provide integrity verification of the HTTP request entity-body. The hash value is calculated using a hash algorithm over the entire HTTP request entity-body as included in the request.
The client MAY include the body hash with any request. The server SHOULD require the calculation and inclusion of the body hash with any request containing an entity-body, or when the presence (or lack of) of an entity-body matters.
The body hash algorithm is determined by the access token algorithm provided with the access token. The SHA-1 hash algorithm as defined by [NIST FIPS-180-3] (National Institute of Standards and Technology, “Secure Hash Standard (SHS). FIPS PUB 180-3, October 2008,” .) is used with the hmac-sha-1 access token algorithm. The SHA-256 hash algorithm as defined by [NIST FIPS-180-3] (National Institute of Standards and Technology, “Secure Hash Standard (SHS). FIPS PUB 180-3, October 2008,” .) is used with the hmac-sha-256 access token algorithm. Additional access token algorithms MUST specify the body hash algorithm.
The body hash is calculated as follows:

```
bodyhash = BASE64 ( HASH (text) )
```

Where:

HASH is the hash algorithm function,

text
: is the HTTP request entity-body,

BASE64 : is the base64-encoding function per [RFC2045] (Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies," November 1996.) section 6.8, applied to the hash digest result octet string, and

bodyhash : is the value used in the normalized request string and to set the bodyhash attribute of the Authorization header field.

The body hash is calculated before the normalized request string is constructed and the signature is calculated.
For example, the HTTP request:

```
POST /request HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded

hello=world%21
```

using access token j92fsdjf094gjfdi, timestamp 137131206, nonce f403hksd, access token algorithm hmac-sha-1, and secret 8yfrufh348h, is transmitted as (line breaks are for display purposes only):

```
POST /request HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Authorization: MAC token="h480djs93hd8",
                   timestamp="137131200",
                   nonce="dj83hs9s",
                   bodyhash="k9kbtCIy0CkI3/FEfpS/oIDjk6k=",
                   signature="FR1UCL6Ny6bsx8EkKkiveFYv5VU="

hello=world%21
```

3.3. Signature

TOC

The client uses the MAC algorithm and the token secret to calculate the request signature. This specification defines two algorithms: hmac-sha-1 and hmac-sha-256, and provides an extension registry for additional algorithms.

3.3.1. Normalized Request String

TOC

The normalized request string is a consistent, reproducible concatenation of several of the HTTP request elements into a single string. By normalizing the request into a reproducible string, the client and resource server can both sign the same string.
The string is constructed by concatenating together, in order, the following HTTP request elements, each followed by a new line character (%x0A):

1. The access token.

2. The timestamp value calculated for the request.

3. The nonce value generated for the request.

4. The request entity-body hash as described in Section 3.2 (Body Hash) if one was calculated and included in the request, otherwise, an empty string. Note that the body hash of an empty entity-body is typically not an empty string.

5. The HTTP request method in upper case. For example: HEAD, GET, POST, etc.

6. The hostname included in the HTTP request using the Host request header field in lower case.

7. The port as included in the HTTP request using the Host request header field. If the header field does not include a port, the default value for the scheme MUST be used (e.g. 80 for HTTP and 443 for HTTPS).

8. The path component of the HTTP request URI as defined by [RFC3986] (Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," January 2005.) section 3.3.

9. The query component of the HTTP request URI as defined by [RFC3986] (Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," January 2005.) section 3.4, normalized as described in Section 3.3.1.1 (Parameters Normalization).

[[TODO: I18N]]
For example, the HTTP request:

```
POST /request?b5=%3D%253D&a3=a&c%40=&a2=r%20b&c2&a3=2+q HTTP/1.1
Host: example.com

Hello World!
```

using access token kkk9d7dh3k39sjv7, timestamp 137131201, nonce 7d8f3e4a, and body hash Lve95gjOVATpfV8EL5X4nxwjKHE= is normalized into the following string (the new line separator character is represented by \n for display purposes only):

```
kkk9d7dh3k39sjv7\n
137131201\n
7d8f3e4a\n
Lve95gjOVATpfV8EL5X4nxwjKHE=\n
POST\n
example.com\n
80\n
/request\n
a2=r%20b\n
a3=2%20q\n
a3=a\n
b5=%3D%253D\n
c%40=\n
c2=\n
```

3.3.1.1. Parameters Normalization

TOC

The query component is parsed into a list of name/value parameter 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-html401-19991224] (Hors, A., Raggett, D., and I. Jacobs, "HTML 4.01 Specification," December 1999.) section 17.13.4. Form-encoded parameters present in the entity-body are not included.
Once separated and decoded, the parameters are concatenated back together as follows:

1. First, the name and value of each parameter are escaped using the [RFC3986] (Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," January 2005.) percent-encoding (%XX) mechanism. Characters in the unreserved character set as defined by [RFC3986] (Berners-

Lee, T., Fielding, R., and L. Masinter, “Uniform Resource Identifier (URI): Generic Syntax,” January 2005.) 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 upper case.

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

3. The name/value parameter pairs are sorted using ascending byte value ordering.

4. The sorted parameters are concatenated together into a single string by using an new line character (ASCII code 10) as separator.

Note that the percent-encoding method described is different from the encoding scheme used by the application/x-www-form-urlencoded content-type (for example, it encodes space characters as %20 instead of the + character). It MAY be different from the percent-encoding functions provided by web development frameworks (e.g. encode different characters, use lower case hexadecimal characters).
For example, the HTTP request URI:

```
/request?b5=%3D%253D&a3=a&c%40=&a2=r%20b&c2&a3=2+q
```

Contains the following (fully decoded) parameters used in the normalized request sting:

Name	Value
b5	=%3D
a3	a
c@	
a2	r b
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 normalized request 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).
The parsed parameters are normalized as follows:

Escaped:

Name	Value
b5	%3D%253D
a3	a
c%40	
a2	r%20b
c2	
a3	2%20q

Concatenated Pairs:

Name=Value
b5=%3D%253D
a3=a
c%40=
a2=r%20b
c2=
a3=2%20q

Sorted:

Name=Value
a2=r%20b
a3=2%20q
a3=a
b5=%3D%253D
c%40=
c2=

And concatenated together into a single string (the new line separator character is represented by \n for display purposes only):

```
a2=r%20b\n
a3=2%20q\n
a3=a\n
b5=%3D%253D\n
c%40=\n
c2=\n
```

3.3.2. hmac-sha-1

TOC

hmac-sha-1 uses the HMAC-SHA1 algorithm as defined in [RFC2104] (Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," February 1997.):

```
digest = HMAC-SHA1 (key, text)
```

Where:

text is set to the value of the normalize request string as described in Section 3.3.1 (Normalized Request String),

key is set to the access token shared-secret provided by the authorization server, and

digest is used to set the value of the signature attribute, after the result octet string is base64-encoded per [RFC2045] (Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies," November 1996.) section 6.8.

3.3.3. hmac-sha-256

TOC

hmac-sha-1 uses the HMAC algorithm as defined in [RFC2104] (Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," February 1997.) together with the SHA-256 hash function defined in [NIST FIPS-180-3] (National Institute of Standards and Technology, "Secure Hash Standard (SHS). FIPS PUB 180-3, October 2008," .):

```
digest = HMAC-SHA256 (key, text)
```

Where:

text is set to the value of the normalize request string as described in Section 3.3.1 (Normalized Request String),

key is set to the access token shared-secret provided by the authorization server, and

digest is used to set the value of the signature attribute, after the result octet string is base64-encoded per [RFC2045] (Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies," November 1996.) section 6.8.

---

4. Verifying Requests

TOC

A servers receiving an authenticated request validates it by performing the following REQUIRED steps:

1. Recalculate the request body hash (if included in the request) as described in Section 3.2 (Body Hash) and signature as described in Section 3.3 (Signature) and compare the signature to the value received from the client via the signature attribute.

2. Ensure that the combination of nonce, timestamp, and access token received from the client has not been used before in a previous request (the server MAY reject requests with stale timestamps; the determination of staleness is left up to the server to define).

3. Verify the scope and status of the access token.

If the request fails verification, the server SHOULD respond with an HTTP 401 (unauthorized) status code, and SHOULD include a token scheme authentication challenge using the WWW-Authenticate header field. The server MAY include further details about why the request was rejected using the error attribute.

TOC

4.1. The "WWW-Authenticate" Response Header Field

If the protected resource request does not include authentication credentials, contains an invalid access token, or is malformed, the resource server MUST include the HTTP WWW-Authenticate response header field. The WWW-Authenticate header field uses the framework defined by [RFC2617] (Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication," June 1999.) as follows:

```
challenge   = "MAC" [ RWS 1#param ]

param       = realm / error / auth-param

error       = "error" "=" quoted-string
```

Each attribute MUST NOT appear more than once.
If the protected resource request included a MAC Authorization header field and failed authentication, the resource server MAY include the error attribute to provide the client with a human-readable explanation why the access request was declined.
For example, in response to a protected resource request without authentication:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: MAC realm="example"
```

And in response to a protected resource request with an authentication attempt using an expired access token:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: MAC realm="example"
                      error="The access token expired"
```

The resource server response SHOULD use the appropriate HTTP status code as follows:

> **400 (Bad Request)** The request is missing a required parameter, includes an unsupported parameter or parameter value, repeats the

same parameter, uses more than one method for including an access token, or is otherwise malformed.

401 (Unauthorized) The access token provided is expired, revoked, malformed, or invalid. The body hash or signature provided do not match the values calculated by the server.

403 (Forbidden) The request requires higher privileges than provided by the access token.

---

5. Scheme Extensions

TOC

[[TBD]]

---

6. Use with OAuth 2.0

TOC

OAuth 2.0 ([I-D.ietf-oauth-v2] (Hammer-Lahav, E., Recordon, D., and D. Hardt, “The OAuth 2.0 Authorization Protocol,” January 2011.)) defines a token-based authentication framework in which third-party applications (clients) access protected resources using access tokens. Access tokens are obtained via the resource owners' authorization from an authorization server. This specification defines the OAuth 2.0 MAC token type, as well as type-specific token attributes.
This specification does not define methods for the client to specifically request a MAC-type token from the authorization server. Additionally, it does not include any discovery facilities for identifying which HMAC algorithms are supported by a resource server, or how the client may go about obtaining MAC access tokens.

6.1. Issuing MAC-Type Access Tokens

TOC

Authorization servers issuing MAC-type access tokens MUST include the following parameters whenever a response includes the access_token parameter:

secret REQUIRED. The token shared secret used as the MAC algorithm key.

algorithm REQUIRED. The MAC algorithm used to calculate the request signature. Value MUST be one of hmac-sha-1, hmac-sha-256, or a

> registered extension algorithm name as described in Section 5 (Scheme Extensions).

---

7. Security Considerations

TOC

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

7.1. Secrets Transmission

TOC

This specification does not describe any mechanism for obtaining or transmitting access token secrets. Methods used to obtain tokens should ensure that these transmissions are protected using transport-layer mechanisms such as TLS or SSL.

7.2. Confidentiality of Requests

TOC

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.

7.3. Spoofing by Counterfeit Servers

TOC

This protocol makes no attempt to verify the authenticity of the resource 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 resource server or of request responses is an issue.

7.4. Plaintext Storage of Credentials

TOC

The access token shared-secret functions the same way passwords do in traditional authentication systems. In order to compute the signature, the server must have access to the secret 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.

7.5. Entropy of Secrets

TOC

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 secret used. Authorization 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, authorization 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, authorization 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.

7.6. Denial of Service / Resource Exhaustion Attacks

TOC

This specification includes a number of features which 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.

7.7. Coverage Limitations

TOC

The normalized request string has been designed to support the authentication methods defined in this specification. Those designing additional methods, should evaluated the compatibility of the normalized request string with their security requirements. Since the normalized request string does not cover the entire HTTP request, servers should employ additional mechanisms to protect such elements. The signature does not cover entity-header fields which can often affect how the request body is interpreted by the server (i.e. Content-Type). If the server behavior is influenced by the presence or value of such header fields, an attacker can manipulate the request header without being detected. This will alter the request even when using the body hash attribute.

---

8. IANA Considerations

TOC

8.1. OAuth Access Token Type Registration

TOC

This specification registers the following access token type in the OAuth Access Token Type Registry.

8.1.1. The "mac" OAuth Access Token Type

TOC

> **Type name:** mac
>
> **Additional Token Endpoint Response Parameters:** secret, algorithm
>
> **HTTP Authentication Scheme(s):** MAC
>
> **Change controller:** IETF
>
> **Specification document(s):** [[this document]]

8.2. OAuth Parameters Registration

TOC

This specification registers the following parameters in the OAuth Parameters Registry established by [I-D.ietf-oauth-v2] (Hammer-Lahav, E., Recordon, D., and D. Hardt, "The OAuth 2.0 Authorization Protocol," January 2011.).

8.2.1. The "secret" OAuth Parameter

TOC

> **Parameter name:** secret
>
> **Parameter usage location:** authorization response, token response
>
> **Change controller:** IETF
>
> **Specification document(s):** [[this document]]
>
> **Related information:** None

8.2.2. The "algorithm" OAuth Parameter

TOC

> **Parameter name:** algorithm
>
> **Parameter usage location:** authorization response, token response
>
> **Change controller:** IETF
>
> **Specification document(s):** [[this document]]

Related information:

None

---

9. Acknowledgments

TOC

The author would like to thank James Manger for his suggestions, feedback, and continued support.

Appendix A. Document History

TOC

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

-02

- *Added body-hash support.
- *Updated OAuth 2.0 reference and added token type registration template.
- *Removed error codes and error URI.

-01

- *Changed parameters sorting to come after name=value string construction.
- *Added new line at the end of the normalized request string.
- *Moved OAuth2 references to separate section.
- *Added 'WWW-Authenticate' header definition.
- *Fixed example header use of single quote.
- *Restricted strings to ASCII subset (printable, no double-quotes or back-slash).

-00

- *Initial draft.

TOC

10. References

10.1. Normative References

TOC

[I-D.ietf-httpbis-p1-messaging]	Fielding, R., Gettys, J., Mogul, J., Nielsen, H., Masinter, L., Leach, P., Berners-Lee, T., and J. Reschke, “HTTP/1.1, part 1: URIs, Connections, and Message Parsing,” draft-ietf-httpbis-p1-messaging-08 (work in progress), October 2009 (TXT).
[I-D.ietf-oauth-v2]	Hammer-Lahav, E., Recordon, D., and D. Hardt, “The OAuth 2.0 Authorization Protocol,” draft-ietf-oauth-v2-12 (work in progress), January 2011 (TXT).
[NIST FIPS-180-3]	National Institute of Standards and Technology, “Secure Hash Standard (SHS). FIPS PUB 180-3, October 2008.”
[RFC2045]	Freed, N. and N. Borenstein, “Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies,” RFC 2045, November 1996 (TXT).
[RFC2104]	Krawczyk, H., Bellare, M., and R. Canetti, “HMAC: Keyed-Hashing for Message Authentication,” RFC 2104, February 1997 (TXT).
[RFC2119]	Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels,” BCP 14, RFC 2119, March 1997 (TXT, HTML, XML).
[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 (TXT, HTML, XML).
[RFC3986]	Berners-Lee, T., Fielding, R., and L. Masinter, “Uniform Resource Identifier (URI): Generic Syntax,” STD 66, RFC 3986, January 2005 (TXT, HTML, XML).
[W3C.REC-html401-19991224]	Hors, A., Raggett, D., and I. Jacobs, “HTML 4.01 Specification,” World Wide Web Consortium Recommendation REC-html401-19991224, December 1999 (HTML).

10.2. Informative References

TOC

[RFC5849]	

	Hammer-Lahav, E., “The OAuth 1.0 Protocol,” RFC 5849, April 2010 (TXT).

Author's Address

TOC

	Eran Hammer-Lahav
	Yahoo!
Email:	eran@hueniverse.com
URI:	http://hueniverse.com