HTTP Authentication: MAC Authentication
draft-hammer-oauth-v2-mac-token-01

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 22, 2011.

Copyright Notice

Copyright (c) 2011 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.
# Table of Contents

1. Introduction ............................................. 3  
   1.1. Example ............................................. 3  
   1.2. Notational Conventions ............................... 5  
2. Issuing MAC Credentials ................................. 5  
3. Making Requests ......................................... 6  
   3.1. The Authorization Request Header .................... 6  
   3.2. Signature ........................................... 7  
      3.2.1. Normalized Request String ....................... 7  
      3.2.2. hmac-sha-1 ..................................... 11  
      3.2.3. hmac-sha-256 .................................. 11  
4. Verifying Requests ....................................... 12  
   4.1. The WWW-Authenticate Response Header Field .......... 12  
   4.1.1. Error Codes ..................................... 13  
5. Scheme Extensions ........................................ 14  
6. Use with OAuth 2.0 ....................................... 14  
   6.1. Issuing MAC-Type Access Tokens ....................... 15  
7. Security Considerations ................................ 15  
   7.1. Secrets Transmission ................................. 15  
   7.2. Confidentiality of Requests .......................... 15  
   7.3. Spoofing by Counterfeit Servers ..................... 15  
   7.4. Plaintext Storage of Credentials .................... 16  
   7.5. Entropy of Secrets .................................. 16  
   7.6. Denial of Service / Resource Exhaustion Attacks .... 16  
   7.7. Coverage Limitations ................................. 17  
8. IANA Considerations ..................................... 17  
   8.1. The "secret" OAuth Parameter ........................ 17  
   8.2. The "algorithm" OAuth Parameter ..................... 18  
9. Acknowledgments ......................................... 18  

Appendix A. Document History ............................... 18  
10. References ............................................. 18  
   10.1. Normative References ............................... 18  
   10.2. Informative References ............................. 19

Author’s Address ............................................ 20
1. Introduction

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].

Please discuss this draft on the oauth@ietf.org [1] mailing list.

1.1. Example

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 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\n137131200\ndj83hs9s\nGET\ne\nexample.com\n80\n/resource/1\na=2\nb=1
```

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:

```
kDZvddkndxvhGRXZhvuDjEWhGeE=
```

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="kDZvddkndxvhGRXZhvuDjEWhGeE="
```

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

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].

This document uses the Augmented Backus-Naur Form (ABNF) notation of [I-D.ietf-httpbis-p1-messaging]. Additionally, the following rules are included from [RFC2617]: realm, auth-param.

2. Issuing MAC Credentials

This specification does not define a general purpose method for requesting or issuing MAC credentials (an OAuth 2.0 [I-D.ietf-oauth-v2] binding is provided in Section 6). 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.

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 <\>
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 a set of attributes, and adding them to the HTTP request using the Authorization header field (Section 3.1). Authenticated request can be sent in response to an authentication challenge or directly.

3.1. The Authorization Request Header

The "Authorization" request header field uses the framework defined by [RFC2617] as follows:

```
credentials = 'MAC' [ RWS 1#param ]
param       = access-token /
              timestamp /
              nonce /
              signature

access-token = 'token' '=' > plain-string <
timestamp    = 'timestamp' '=' > 1*DIGIT <
nonce        = 'nonce'  '=' > plain-string <
signature    = 'signature' '=' > plain-string <
plain-string = 1*( DIGIT / ALPHA / %x20-21 / %x23-5B / %x5D-7E )
```

The "token" attribute value is set to the access token string.

The "timestamp" attribute value is set to the current time expressed in the number of seconds since January 1, 1970 00:00:00 GMT, and MUST be a positive integer.

The "nonce" attribute value is set to 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.

The "signature" attribute value is set as described in Section 3.2.

Each of the four attributes MUST appear once, and only once. Attribute values are limited to a subset of ASCII, which does not require escaping.

3.2. Signature

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.

3.2.1. Normalized Request String

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 HTTP request method in upper case. For example: "HEAD", "GET", "POST", etc.
5. The hostname included in the HTTP request using the "Host" request header field in lower case.
6. 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).
7. The path component of the HTTP request URI as defined by [RFC3986] section 3.3.
8. The query component of the HTTP request URI as defined by [RFC3986] section 3.4, normalized as described in
Section 3.2.1.1.

For example, the HTTP request:

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

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

kkk9d7dh3k39sjv7
137131201
7d8f3e4a
GET\nexample.com\n80\n/request\n/a2=r%20b\n/a3=2%20q\n/a3=a\nb5=%3D%253D\nc%40=\nc2=\n
3.2.1.1. Parameters Normalization

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

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] percent-encoding (%XX) mechanism. 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 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>b5</td>
<td>%=3D</td>
</tr>
<tr>
<td>a3</td>
<td>a</td>
</tr>
<tr>
<td>c@</td>
<td></td>
</tr>
<tr>
<td>a2</td>
<td>r b</td>
</tr>
<tr>
<td>c2</td>
<td></td>
</tr>
<tr>
<td>a3</td>
<td>2 q</td>
</tr>
</tbody>
</table>

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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>b5</td>
<td>%3D%253D</td>
</tr>
<tr>
<td>a3</td>
<td>a</td>
</tr>
<tr>
<td>c%40</td>
<td></td>
</tr>
<tr>
<td>a2</td>
<td>r%20b</td>
</tr>
<tr>
<td>c2</td>
<td></td>
</tr>
<tr>
<td>a3</td>
<td>2%20q</td>
</tr>
</tbody>
</table>

**Concatenated Pairs:**

<table>
<thead>
<tr>
<th>Name=Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>b5=%3D%253D</td>
</tr>
<tr>
<td>a3=a</td>
</tr>
<tr>
<td>c%40=</td>
</tr>
<tr>
<td>a2=r%20b</td>
</tr>
<tr>
<td>c2=</td>
</tr>
<tr>
<td>a3=2%20q</td>
</tr>
</tbody>
</table>

**Sorted:**

<table>
<thead>
<tr>
<th>Name=Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a2=r%20b</td>
</tr>
<tr>
<td>a3=2%20q</td>
</tr>
<tr>
<td>a3=a</td>
</tr>
<tr>
<td>b5=%3D%253D</td>
</tr>
<tr>
<td>c%40=</td>
</tr>
<tr>
<td>c2=</td>
</tr>
</tbody>
</table>
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```

### 3.2.2. hmac-sha-1

"hmac-sha-1" uses the HMAC-SHA1 algorithm as defined in [RFC2104]:

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

Where:

- **text** is set to the value of the normalize request string as described in Section 3.2.1.
- **key** is set to the access token shared-secret provided by the authorization server.
- **digest** is used to set the value of the "signature" attribute, after the result octet string is base64-encoded per [RFC2045] section 6.8.

### 3.2.3. hmac-sha-256

"hmac-sha-1" uses the HMAC algorithm as defined in [RFC2104] together with the SHA-256 hash function defined in [NIST FIPS-180-3]:

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

Where:
is set to the value of the normalize request string as described in Section 3.2.1.

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

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

4. Verifying Requests

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

1. Recalculate the request signature as described in Section 3.2 and compare it 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.

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] as follows:
Each attribute MUST NOT appear more than once.

If the protected resource request included an access token and failed authentication, the resource server SHOULD include the "error" attribute to provide the client with the reason why the access request was declined. The parameter value is described in Section 4.1.1. In addition, the resource server MAY include the "error_description" attribute to provide a human-readable explanation, and the "error-uri" attribute with an absolute URI identifying a human-readable web page explaining the error.

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="invalid_token",
    error_description="The access token expired"

4.1.1. Error Codes

When a request fails, the resource server responds using the appropriate HTTP status code (typically, 400, 401, or 403), and includes one of the following error codes in the response:

invalid_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. The resource server SHOULD respond with the HTTP 400 (Bad Request) status code.

invalid_token
The access token provided is expired, revoked, malformed, or invalid for other reasons. The resource SHOULD respond with the HTTP 401 (Unauthorized) status code. The client MAY request a new access token and retry the protected resource request.

insufficient_scope
The request requires higher privileges than provided by the access token. The resource server SHOULD respond with the HTTP 403 (Forbidden) status code and MAY include the "scope" attribute with the scope necessary to access the protected resource.

If the request lacks any authentication information (i.e. the client was unaware authentication is necessary or attempted using an unsupported authentication method), the resource server SHOULD not include an error code or other error information.

For example:

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

5. Scheme Extensions
[[ TBD ]]

6. Use with OAuth 2.0

OAuth 2.0 ([I-D.ietf-oauth-v2]) 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

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.

7. 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.

7.1. Secrets Transmission

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

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

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

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

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

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

The normalized request string has been designed to support the authentication methods defined in this specification. Those designing additional methods, should evaluate 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.

8. IANA Considerations

8.1. The "secret" OAuth Parameter

Parameter name: secret

Parameter usage location: The end-user authorization endpoint response and the token endpoint response.

Change controller: IETF

Specification document(s): [[ this document ]]

Related information: None
8.2. The "algorithm" OAuth Parameter

Parameter name:  algorithm

Parameter usage location: The end-user authorization endpoint response and the token endpoint response.

Change controller: IETF

Specification document(s): [[ this document ]]

Related information: None

9. Acknowledgments

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

Appendix A. Document History

-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.

-00
- Initial draft.

10. References

10.1. Normative References

[I-D.ietf-httpbis-p1-messaging]
  Fielding, R., Gettys, J., Mogul, J., Nielsen, H.,
10.2. Informative References


URIs

Author’s Address

Eran Hammer-Lahav
Yahoo!

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