OAuth   J. Richer, Ed.
Internet-Draft   The MITRE Corporation
Intended status: Standards Track   W. Mills, Ed.
Expires: May 30, 2013   Yahoo! Inc.
                                                     H. Tschofenig, Ed.
                                                 Nokia Siemens Networks
                                                      November 28, 2012

OAuth 2.0 Message Authentication Code (MAC) Tokens
draft-ietf-oauth-v2-http-mac-02

Abstract

This document specifies the HTTP MAC access authentication scheme, an HTTP authentication method using a message authentication code (MAC) algorithm to provide cryptographic verification of portions of HTTP requests. The document also defines an OAuth 2.0 binding for use as an access token type.

NOTE: This document (and other OAuth 2.0 security documents, such as [I-D.tschofenig-oauth-hotk]) are still work in progress in the OAuth working group. As such, the content of this document may change. For a discussion about security requirements please consult [I-D .tschofenig-oauth-security]. Your input on the detailed security requirements is highly appreciated.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

This specification defines the HTTP MAC access authentication scheme, providing a method for making authenticated HTTP requests with partial cryptographic verification of the request, covering the HTTP method, request URI, and host.

Similar to the HTTP Basic access authentication scheme [RFC2617], the MAC scheme utilizes a set of client credentials which include an identifier and key. However, in contrast with the Basic scheme, the key is never included in authenticated requests but is used to calculate the request MAC value which is included instead.

The MAC scheme requires the establishment of a shared symmetric key between the client and the server. This specification offers one such method for issuing a set of MAC credentials to the client using OAuth 2.0 in the form of a MAC-type access token.

The primary design goal of this mechanism is to simplify and improve HTTP authentication for services that are unwilling or unable to employ TLS for every request. In particular, this mechanism leverage an initial TLS setup phase to establish a shared secret between the client and the server. The shared secret is then used over an insecure channel to provide protection against a passive network attacker.

In particular, when a server uses this mechanism, a passive network attacker will be unable to "steal" the user's session token, as is possible today with cookies and other bearer tokens. In addition, this mechanism helps secure the session token against leakage when sent over a secure channel to the wrong server. For example, when the client uses some form of dynamic configuration to determine where to send an authenticated request, or when the client fails to properly validate the server's identity as part of its TLS handshake.

Unlike the HTTP Digest authentication scheme, this mechanism does not require interacting with the server to prevent replay attacks. Instead, the client provides both a nonce and a timestamp, which the server can use to prevent replay attacks using a bounded amount of storage. Also unlike Digest, this mechanism is not intended to protect the user's password itself because the client and server both have access to the key material in the clear. Instead, servers should issue a short-lived derivative credential for this mechanism during the initial TLS setup phase.

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
```

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

MAC key identifier: h480djs93hd8

MAC key: 489dks293j39

MAC algorithm: hmac-sha-1

The client constructs the authentication header by calculating a timestamp (e.g. the number of seconds since January 1, 1970 00:00:00 GMT) and generating a random string used as a nonce:

Timestamp: 1336363200

Nonce: dj83hs9s

The client constructs the normalized request string (the new line separator character is represented by "\n" for display purposes only; the trailing new line separator signify that no extension value is included with the request, explained below):

```
1336363200\n
dj83hs9s\n
GET\n
/resource/1?b=1&a=2\n
example.com\n
80\n
\n
```

The request MAC is calculated using the specified MAC algorithm "hmac-sha-1" and the MAC key over the normalized request string. The result is base64-encoded to produce the request MAC:

```
bhCQXTVyfj5cmA9uKkPFx1zeOXM=
```

The client includes the MAC key identifier, nonce, and request MAC

with the request using the "Authorization" request header field:

```
  GET /resource/1?b=1&a=2 HTTP/1.1
  Host: example.com
  Authorization: MAC id="h480djs93hd8",
                     ts="1336363200",
                     nonce="dj83hs9s",
                     mac="bhCQXTVyfj5cmA9uKkPFx1zeOXM="
```

The server validates the request by calculating the request MAC again based on the request received and verifies the validity and scope of the MAC credentials. If valid, the server responds with the requested 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 specification are to be interpreted as described in [RFC2119].

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

2. Issuing MAC Credentials

This specification provides one method for issuing MAC credentials using OAuth 2.0 as described in Section 5. This specification does not mandate servers to support any particular method for issuing MAC credentials, and other methods MAY be defined and used. Whenever MAC credentials are issued, the credentials MUST include the following attributes:

MAC key identifier
A string identifying the MAC key used to calculate the request MAC. The string is usually opaque to the client. The server typically assigns a specific scope and lifetime to each set of MAC credentials. The identifier MAY denote a unique value used to retrieve the authorization information (e.g. from a database), or self-contain the authorization information in a verifiable manner (i.e. a string consisting of some data and a signature).

MAC key
A shared symmetric secret used as the MAC algorithm key. The server MUST NOT reissue a previously issued MAC key and MAC key identifier combination.

MAC algorithm

A MAC algorithm used to calculate the request MAC. Value MUST be one of "hmac-sha-1", "hmac-sha-256", or a registered extension algorithm name as described in Section 7.1. Algorithm names are case-sensitive. If the MAC algorithm is not understood by the client, the client MUST NOT use the MAC credentials and continue as if no MAC credentials were issued.

The MAC key identifier, MAC key, MAC algorithm strings MUST NOT include characters other than:

```
  %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 the possession of a valid set of MAC credentials accepted by the server. The client constructs the request by calculating a set of attributes, and adding them to the HTTP request using the "Authorization" request header field as described in Section 3.1.

3.1. The "Authorization" Request Header

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

```
  credentials    = "MAC" 1*SP #params

  params         = id / ts / nonce / ext / mac

  id             = "id" "=" string-value
  ts             = "ts" "=" ( <"> timestamp <"> ) / timestamp
  nonce          = "nonce" "=" string-value
  ext            = "ext" "=" string-value
  mac            = "mac" "=" string-value

  timestamp      = 1*DIGIT
  string-value   = ( <"> plain-string <"> ) / plain-string
  plain-string   = 1*( %x20-21 / %x23-5B / %x5D-7E )
```

The header attributes are set as follows:

id

REQUIRED. The MAC key identifier.

ts

REQUIRED. The request timestamp. The value MUST be a positive integer set by the client when making each request to the number of seconds elapsed from a fixed point in time (e.g. January 1, 1970 00:00:00 GMT). The value MUST NOT include leading zeros (e.g. "000273154346").

nonce

REQUIRED. A unique string generated by the client. The value MUST be unique across all requests with the same timestamp and MAC key identifier combination.

ext

OPTIONAL. A string used to include additional information which is covered by the request MAC. The content and format of the string is beyond the scope of this specification.

mac

REQUIRED. The HTTP request MAC as described in Section 3.2.

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. Request MAC

The client uses the MAC algorithm and the MAC key to calculate the request MAC. 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 server can both calculate the request MAC over the exact same value.

The string is constructed by concatenating together, in order, the following HTTP request elements, each followed by a new line character (%x0A):

1. The timestamp value calculated for the request.

2. The nonce value generated for the request.

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

4. The HTTP request-URI as defined by [RFC2616] section 5.1.2.

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 value of the "ext" "Authorization" request header field attribute if one was included in the request, otherwise, an empty string.

Each element is followed by a new line character (%x0A) including the last element and even when an element value is an empty string.

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 timestamp "264095:7d8f3e4a", nonce "7d8f3e4a", and extension string "a,b,c" is normalized into the following string (the new line separator character is represented by "\n" for display purposes only):

```
264095\n
7d8f3e4a\n
POST\n
/request?b5=%3D%253D&a3=a&c%40=&a2=r%20b&c2&a3=2+q\n
example.com\n
80\n
a,b,c\n
```

3.2.2. hmac-sha-1

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

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

Where:

text
      is set to the value of the normalized request string as

described in Section 3.2.1,

key

is set to the MAC key provided by the server, and

mac

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

3.2.3. hmac-sha-256

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

```
  mac = HMAC-SHA256 (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 MAC key provided by the server, and

mac

is used to set the value of the "mac" 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 MAC as described in Section 3.2 and compare the request MAC to the value received from the client via the "mac" attribute.

2. Ensure that the combination of timestamp, nonce, and MAC key identifier received from the client has not been received before in a previous request. The server MAY reject requests with stale timestamps as described in Section 4.1.

3. Verify the scope and validity of the MAC credentials.

If the request fails verification, the server SHOULD respond using the 401 (Unauthorized) HTTP status code and include the "WWW-Authenticate" response header field as described in Section 4.2.

4.1. Timestamp Verification

The timestamp, nonce, and MAC key identifier combination provide a unique identifier which enables the server to prevent replay attacks. Without replay protection, an attacker can use a compromised (but otherwise valid and authenticated) request more than once, gaining long term access to a protected resource.

Including a timestamp with the nonce removes the need to retain an infinite number of nonce values for future checks, by enabling the server to restrict the time period after which a request with an old timestamp is rejected. If such a restriction is enforced, the server MUST:

- At the time the first request is received from the client for each MAC key identifier, calculate the difference (in seconds) between the request timestamp and the server's clock. The difference - the request time delta - MUST be kept as long as the MAC key credentials are valid.
- For each subsequent client request, apply the request time delta to request timestamp to calculate the adjusted request time - the time when the request MAC has been generated by the client, adjusted to the server's clock.
- Verify that the adjusted request time is within the allowed time period defined by the server. The server SHOULD allow for a sufficiently large window to accommodate network delays (between the time the request has been generated by the client to the time it is received by the server and processed).

4.2. The "WWW-Authenticate" Response Header Field

If the protected resource request does not include authentication credentials, contains an invalid MAC key identifier, or is malformed, the server SHOULD include the HTTP "WWW-Authenticate" response header field.

For example:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: MAC
```

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

```
challenge   = "MAC" [ 1*SP #param ]
param       = error / auth-param
```

```
     error       = "error" "=" ( token / quoted-string)
```

Each attribute MUST NOT appear more than once.

If the protected resource request included a MAC "Authorization" request header field and failed authentication, the server MAY include the "error" attribute to provide the client with a human-readable explanation why the access request was declined to assist the client developer in identifying the problem.

For example:

```
  HTTP/1.1 401 Unauthorized
  WWW-Authenticate: MAC error="The MAC credentials expired"
```

5. Use with OAuth 2.0

OAuth 2.0 ([RFC6749]) defines an extensible token-based authentication framework. The MAC authentication scheme can be used to make OAuth-based requests by issuing MAC-type access tokens.

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 for any given protected resource.

5.1. 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:

access_token
REQUIRED. The MAC key identifier.

mac_key
REQUIRED. The MAC key.

mac_algorithm
REQUIRED. The MAC algorithm used to calculate the request MAC. Value MUST be one of "hmac-sha-1", "hmac-sha-256", or a registered extension algorithm name as described in Section 7.1.

For example:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
  "access_token":"SlAV32hkKG",
  "token_type":"mac",
  "expires_in":3600,
  "refresh_token":"8xLOxBtZp8",
  "mac_key":"adijq39jdlaska9asud",
  "mac_algorithm":"hmac-sha-256"
}
```

6. 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.

6.1. MAC Keys Transmission

This specification describes two mechanism for obtaining or transmitting MAC keys, both require the use of a transport-layer security mechanism when sending MAC keys to the client. Additional methods used to obtain MAC credentials must ensure that these transmissions are protected using transport-layer mechanisms such as TLS or SSL.

6.2. 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.

6.3. 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 resource server or of request responses is an issue.

6.4. Plaintext Storage of Credentials

The MAC key functions the same way passwords do in traditional authentication systems. In order to compute the request MAC, the server must have access to the MAC key 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 MAC keys - or worse, to the server's database of all such MAC keys - he or she would be able to perform any action on behalf of any resource owner. Accordingly, it is critical that servers protect these MAC keys from unauthorized access.

6.5. Entropy of MAC Keys

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

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

It is equally important that the pseudo-random number generator (PRNG) used to generate these MAC keys 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.

6.6. 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 request MAC 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) MAC keys. When implementing this protocol, servers should consider which of these presents a more serious risk for their application and design accordingly.

6.7. Timing Attacks

This specification makes use of HMACs, for which a signature verification involves comparing the received MAC string to the expected one. If the string comparison operator operates in observably different times depending on inputs, e.g. because it compares the strings character by character and returns a negative result as soon as two characters fail to match, then it may be possible to use this timing information to determine the expected MAC, character by character.

Service implementers are encouraged to use fixed-time string comparators for MAC verification.

6.8. CSRF Attacks

A Cross-Site Request Forgery attack occurs when a site, evil.com, initiates within the victim's browser the loading of a URL from or the posting of a form to a web site where a side-effect will occur, e.g. transfer of money, change of status message, etc. To prevent this kind of attack, web sites may use various techniques to determine that the originator of the request is indeed the site itself, rather than a third party. The classic approach is to include, in the set of URL parameters or form content, a nonce generated by the server and tied to the user's session, which indicates that only the server could have triggered the action.

Recently, the Origin HTTP header has been proposed and deployed in some browsers. This header indicates the scheme, host, and port of the originator of a request. Some web applications may use this Origin header as a defense against CSRF.

To keep this specification simple, HTTP headers are not part of the string to be MAC'ed. As a result, MAC authentication cannot defend against header spoofing, and a web site that uses the Host header to defend against CSRF attacks cannot use MAC authentication to defend against active network attackers. Sites that want the full protection of MAC Authentication should use traditional, cookie-tied

CSRF defenses.

6.9. Coverage Limitations

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 request MAC 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.

7. IANA Considerations

7.1. The HTTP MAC Authentication Scheme Algorithm Registry

This specification establishes the HTTP MAC authentication scheme algorithm registry.

Additional MAC algorithms are registered on the advice of one or more Designated Experts (appointed by the IESG or their delegate), with a Specification Required (using terminology from [RFC5226]). However, to allow for the allocation of values prior to publication, the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published.

Registration requests should be sent to the [TBD]@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request for MAC Algorithm: example"). [[Note to RFC-EDITOR: The name of the mailing list should be determined in consultation with the IESG and IANA. Suggested name: http-mac-ext-review.]]

Within at most 14 days of the request, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

Decisions (or lack thereof) made by the Designated Expert can be first appealed to Application Area Directors (contactable using app-ads@tools.ietf.org email address or directly by looking up their email addresses on http://www.iesg.org/ website) and, if the appellant is not satisfied with the response, to the full IESG (using the iesg@iesg.org mailing list).

IANA should only accept registry updates from the Designated Expert(s), and should direct all requests for registration to the review mailing list.

7.1.1. Registration Template

Algorithm name:
The name requested (e.g., "example").

Change controller:
For standards-track RFCs, state "IETF". For others, give the name of the responsible party. Other details (e.g., postal address, e-mail address, home page URI) may also be included.

Specification document(s):
Reference to document that specifies the algorithm, preferably including a URI that can be used to retrieve a copy of the document. An indication of the relevant sections may also be included, but is not required.

7.1.2. Initial Registry Contents

The HTTP MAC authentication scheme algorithm registry's initial contents are:

- Algorithm name: hmac-sha-1

- Change controller: IETF

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

- Algorithm name: hmac-sha-256

- Change controller: IETF

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

7.2. OAuth Access Token Type Registration

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

7.2.1. The "mac" OAuth Access Token Type

Type name:
mac

Additional Token Endpoint Response Parameters:
secret, algorithm

HTTP Authentication Scheme(s):
MAC

Change controller:
IETF

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

7.3. OAuth Parameters Registration

This specification registers the following parameters in the OAuth Parameters Registry established by [RFC6749].

7.3.1. The "mac_key" OAuth Parameter

Parameter name: mac_key

Parameter usage location: authorization response, token response

Change controller: IETF

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

Related information: None

7.3.2. The "mac_algorithm" OAuth Parameter

Parameter name: mac_algorithm

Parameter usage location: authorization response, token response

Change controller: IETF

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

Related information: None

8. Contributors

This document is based on OAuth 1.0 and we would like to thank Eran Hammer-Lahav for his work on incorporating the ideas into OAuth 2.0.

9. Acknowledgments

The author would like to thank Ben Adida, Adam Barth, Phil Hunt, Rasmus Lerdorf, James Manger, William Mills, Scott Renfro, Justin Richer, Toby White, Peter Wolanin, and Skylar Woodward for their contributions, suggestions, and feedback.

10. References

10.1. Normative References

[10] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", Internet-Draft draft-ietf-httpbis-p1-messaging-21, October 2012.

[11] Hardt, D., "The OAuth 2.0 Authorization Framework", RFC

6749, October 2012.

[12] Hors, A., Raggett, D. and I. Jacobs, "HTML 4.01 Specification", World Wide Web Consortium Recommendation REC-html401-19991224, December 1999, <http://www.w3.org/TR/1999/REC-html401-19991224>.

[13] National Institute of Standards and Technology, "Secure Hash Standard (SHS). FIPS PUB 180-3, October 2008", .

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

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

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

[4] 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.

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

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

[7] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[8] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[9] Barth, A., "HTTP State Management Mechanism", RFC 6265, April 2011.

10.2. Informative References

[1] Tschofenig, H. and P. Hunt, "OAuth 2.0 Security: Going Beyond Bearer Tokens", Internet-Draft draft-tschofenig-oauth-security-00, September 2012.

[2] Bradley, J., Hunt, P., Nadalin, A. and H. Tschofenig, "The

OAuth 2.0 Authorization Framework: Holder-of-the-Key Token
Usage", Internet-Draft draft-tschofenig-oauth-hotk-01,
July 2012.

[3] Hammer-Lahav, E., "The OAuth 1.0 Protocol", RFC 5849, April 2010.

Authors' Addresses

Justin Richer, editor
The MITRE Corporation

Email: jricher@mitre.org

William Mills, editor
Yahoo! Inc.

Email: wmills@yahoo-inc.com

Hannes Tschofenig, editor
Nokia Siemens Networks
Linnoitustie 6
Espoo, 02600
Finland

Phone: +358 (50) 4871445
Email: Hannes.Tschofenig@gmx.net
URI: http://www.tschofenig.priv.at