

Network Working Group
Internet-Draft
Expires: August 30, 2007

D. Cridland
A. Melnikov
Isode Limited
February 26, 2007

The Hash Exchange Authentication SASL Mechanism
draft-cridland-sasl-hexa-00

Status of this Memo

By submitting this Internet-Draft, each author represents that any applicable patent or other IPR claims of which he or she is aware have been or will be disclosed, and any of which he or she becomes aware will be disclosed, in accordance with [Section 6 of BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at <http://www.ietf.org/ietf/lid-abstracts.txt>.

The list of Internet-Draft Shadow Directories can be accessed at <http://www.ietf.org/shadow.html>.

This Internet-Draft will expire on August 30, 2007.

Copyright Notice

Copyright (C) The IETF Trust (2007).

Abstract

This memo defines and discusses a SASL mechanism that is based on the exchange of hashes. It does not require the storage of a plaintext equivalent on the server, is simple to implement, and provides a reasonable level of security.

Internet-Draft

SASL HEXA

February 2007

Table of Contents

1.	Conventions used in this document	3
2.	Introduction	3
2.1.	Rationale	3
2.1.1.	Deployability	3
2.1.2.	Hash Agility	3
2.1.3.	Ease of Implementation	4
3.	Notations and Definitions	4
3.1.	HMAC and Hash functions	4
3.1.1.	Notation	4
3.2.	Wire Message Format	5
3.3.	Prior Setup	5
3.4.	Authentication Process	5
3.4.1.	Initial client message	5
3.4.2.	Server challenge message	6
3.4.3.	Client response message	6
3.4.4.	Server Authentication and Mutual Auth	6
4.	Mandatory to Implement	7
5.	Formal Syntax	7
6.	Security Considerations	8
6.1.	Plaintext Equivalents	8
6.2.	Hash algorithm usage	8
6.3.	Resistance to attacks	9
7.	References	9
7.1.	Normative References	9
7.2.	Informative References	10
	Authors' Addresses	10
	Intellectual Property and Copyright Statements	12

Internet-Draft

SASL HEXA

February 2007

[1.](#) Conventions used in this document

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 [[KEYWORDS](#)].

[2.](#) Introduction

Although other hash-based [[SASL](#)] mechanisms exist, they are being rapidly outdated by advances in computing speed and the discovery of weaknesses in hash functions. Moreover, both [[CRAM-MD5](#)] and [[DIGEST-MD5](#)] involve the server having plaintext equivalents in the shared secret store.

This mechanism was borne out of a perceived need within the system administration community to have a mechanism which was both easier to implement and also safer than [[DIGEST-MD5](#)].

[2.1.](#) Rationale

HEXA is specifically aimed at providing three key features.

[2.1.1.](#) Deployability

HEXA is designed to be acceptable to deploy in the real world. Its authentication database is designed such that it may be used directly for local logins, effectively having the same properties as a typical /etc/shadow file on UNIX systems. This allows HEXA to co-exist with very well deployed mechanisms such as [[PLAIN](#)], freeing the need for transitioning.

[2.1.2.](#) Hash Agility

Hash algorithms have an alarming tendency to age. It is therefore beneficial to allow a server administrator to switch hash algorithms.

This is only practical, however, when it is known that the new hash algorithm can be well supported by the clients in use. Where the clients are out of the control of the administrator, for example in typical commercial settings, it is useful for the administrator to be aware of what the current deployed base is able to use.

Therefore, in HEXA, clients advertise their capability to the server, allowing a server administrator to upgrade the hash algorithm as required.

[2.1.3.](#) Ease of Implementation

In general, [[DIGEST-MD5](#)] has been found to be difficult to implement interoperably. Even well known implementations have been found to have interoperability problems under some circumstances. HEXA attempts to tackle this by using a small number of operation types, and simple parsing. This allows for simple scripting languages to implement, as well as using hash algorithms and related functions that are known to be well deployed.

Since no mechanism can be considered secure in practise if no implementations exist, this specification chooses easily available pre-existing source code above stronger, less well implemented algorithms.

[3.](#) Notations and Definitions

[3.1.](#) HMAC and Hash functions

This specification requires the use of [[HMAC](#)], based on hash functions such as [[MD5](#)], SHA1, or SHA-256.

Messages are shown in plain text, with the CR LF pair shown as a line ending.

Mandatory to implement hashes are discussed in [Section 4](#), and are considered volatile parts of this specification, very likely to change in future revisions of this specification.

[3.1.1.](#) Notation

We use a relatively simple notation to show the calculations involved:

`HASH(T)`

An agreed hash algorithm, used to produce a cryptographically secure hash of the input data T.

`HMAC(K,T)`

An [[HMAC](#)] function used with an agreed hash function, used to produce a MAC for T with a key of K.

`HMAC[n](K,T)`

Where n is an integer, expands to `HMAC(HMAC[n-1](K,T),T)`.

`HMAC[1](K,T)` is equivalent to `HMAC(K,T)`.

`Q + W`

Where Q and W are strings, represents simple concatenation.

`Q ^ W`

Where Q and W are strings, represents an octet by octet XOR.

`SASLprep(X)`

Where X is a string, represents the application of the [[SASLPREP](#)] algorithm to the string.

[3.2.](#) Wire Message Format

HEXA uses a wire format based on a simplified variant of email message header formats, and only transfers text. No folding or encoding is required or allowed.

The message contains keys and values, where each key appears a maximum of once. Keys consist of ASCII letters, numbers, and the hyphen character. Values contain UTF-8, and begin with a non-space character. They never contain NUL, CR, or LF. Each Key Value pair ends with a CR LF pair.

Keys appear first, followed by a single colon (":"), followed by the value. Any surrounding spaces are considered part of the value.

[3.3.](#) Prior Setup

The client is assumed to have an Authcid, an optional Authzid, and a Password.

The server has a Realm, and a database keyed against Authcid containing a Salt, and hash output known as the Verifier.

```
Verifier := HMAC[n](Intermediate, Salt)
Intermediate := HMAC[n](Realm + SASLprep(Authcid)
    + SASLprep>Password), Salt)
```

The server's database is initially populated before authentication by temporarily calculating the Intermediate from the supplied Password, and choosing a new Salt. A new Salt SHOULD be used whenever the Password is changed. The hash algorithm used is also remembered by the server - servers MAY use multiple hash algorithms.

After calculating and storing the Verifier, the Intermediate MUST be discarded.

[3.4.](#) Authentication Process

[3.4.1.](#) Initial client message

Initially, the client sends a message containing Authcid, optionally Authzid, a list of hash algorithm names it supports, and some random

data to use as a client nonce, this message is ClientMessage:

```
Authcid:mel
Hashes:MD5 SHA1 SHA-256
Client-Nonce:laksjdoijcosijdv
Channel-Bindings:TLS
```

[3.4.2.](#) Server challenge message

The server looks up Authcid in its database, selects the strongest hash algorithm mutually supported, and returns the hash algorithm, the number of cycles it uses, and the value of Salt. It also creates some random data for use as a server nonce. Because this MUST be textual, servers MAY base64 encode this data, however, this is an implementation detail. The server sends the server nonce, Salt, and

Realm to the client, along with an indication of which channel binding the server will use, if any:

```
Realm:example.net
Salt:aajvskjhvslkjdnvcn
Hash:MD5
Hash-Cycles:5
Server-Nonce:ksjdnclksdhufdh
Channel-Binding:TLS
```

[3.4.3.](#) Client response message

The client stores this message precisely as received, as `ServerMessage`.

The client now calculates `Intermediate` and `Verifier` as above, and in addition a hash `Key`, and a value `Exchange`, such that:

```
Key := HMAC[n](Verifier, ClientMessage
+ ServerMessage + ChannelBinding)
Exchange := Key ^ Intermediate
```

If there is no channel binding available that the server supports, then `ChannelBinding` will be an empty string. `Exchange` is represented as hex, and the result is sent to the server:

```
Hash-Exchange:1f2d...
```

[3.4.4.](#) Server Authentication and Mutual Auth

The server can now construct `Key`, extract `Intermediate` from `Exchange`, and verify `Intermediate` against the stored hash output `Verifier`. In order to prove to the client that it can do so, it sends the client a

final message containing a hash output `Authentication`, such that:

```
Intermediate := Exchange ^ Key
Authentication := HMAC[n](Intermediate, ChannelBinding
+ ServerMessage + Salt + ClientMessage)
```

This has is sent to the client as:

Server-Auth:3f4d5a...

4. Mandatory to Implement

The rationale behind this mechanism is ease of deployment and implementation, thus implementations MUST provide a configuration which supports the [MD5] hash algorithm using a minimal number of cycles of 16. Implementations SHOULD also support SHA-256.

This is because both [MD5], and [HMAC] implementations which are hardcoded to use [MD5], are easily available in many languages and environments.

5. Formal Syntax

Insert boilerplate about [ABNF] here.

```
wire-message = *key-value
key-value = key ":" value CRLF
key = ALPHA *( ALPHA / "-" / DIGIT )
value = utf8-text
utf8-text = 1*( VCHAR / SP / UTF-2 / UTF8-3 / UTF8-4 )
; visible UTF-8 or space.
hash-output = 1*( HEXDIG )
; Output of hash algorithm, generally 32 or more has digits.

; Following productions all conform to wire-message:

client-init-message = authcid [ authzid ] hashes client-nonce
    [ channel-bindings ] *( extension )
server-challenge-message = realm hash hash-cycles server-nonce
    [ channel-binding ] *( extension )
client-response-message = hash-exchange *( extension )
server-auth-message = server-auth *( extension )

; Following productions all conform to key-value:

; client-init-message:
```

authcid = "Authcid" ":" utf8text CRLF


```

authzid = "Authzid" ":" utf8text CRLF
; SASLPrepped authcid/authzid.
hashes = "Hashes" ":" hash-name *( SP hash-name ) CRLF
client-nonce = "Client-Nonce" ":" utf8text CRLF
; MUST be generated afresh with reasonable entropy.
channel-bindings = "Channel-Bindings" ":" channel-binding-name
    *( SP channel-binding-name ) CRLF

; server-challenge-message:
realm = "Realm" ":" utf8text CRLF
hash = "Hash" ":" hash-name CRLF
hash-cycles = "Cycles" ":" 1*( DIGIT ) CRLF
server-nonce = "Server-Nonce" ":" utf8text CRLF
; MUST be generated afresh with reasonable entropy.
channel-binding = "Channel-Binding:" ":" channel-binding-name CRLF

; client-response-message:
hash-exchange = "Hash-Exchange" ":" hash-output CRLF

; server-auth-message:
server-auth = "Server-Auth" ":" hash-output CRLF

; Values:
channel-binding-name = ALPHA *( ALPHA / DIGIT / "-" )
hash-name = ALPHA *( ALPHA / DIGIT / "-" )

; Extensions:
extension = key-value

```

[6. Security Considerations](#)

[6.1. Plaintext Equivalents](#)

The intermediate hash B is a plaintext equivalent. Clients SHOULD NOT store this, and MUST NOT store the original plaintext password. Servers MUST NOT store B.

[6.2. Hash algorithm usage](#)

In general, it is thought that the recursive application of hash functions increases the strength of a hash. In particular, if the hash has no weaknesses at all, merely doubling the number of iterations will cause an offline dictionary attack to take twice as much CPU resource. Making this a variable, negotiated, factor allows very simple increases in security, as long as the hash algorithm itself is not compromised sufficiently that a non-brute-force attack

becomes practical.

The exact hash algorithm used may be changed by live deployments. HEXA provides a simple method for server administrators to discover actual availability of new hash algorithms in clients, simplifying a hash algorithm change.

[6.3.](#) Resistance to attacks

HEXA is thought to be resistant to slightly more attacks than [\[DIGEST-MD5\]](#):

Downgrade

Assuming that HEXA can be negotiated at all, a downgrade attack inside HEXA cannot be mounted, as complete messages are used as input to the hashing functions - a man-in-the-middle attack will cause the authentication to fail.

Server based attack

Merely obtaining the authentication database will not directly allow an attacker to authenticate masquerading as a legitimate user without substantial offline dictionary attacks. However, if an attacker can both obtain the authentication database and observe traffic on the wire, then the attacker can obtain B. As with [\[DIGEST-MD5\]](#), this will not yield the password without an expensive offline-dictionary attack.

Client based attack

Clients typically store sufficient data to reauthenticate later without interactively requesting passwords from the user. Like [\[DIGEST-MD5\]](#), clients need not store the actual password, but can merely store B for this purpose. This practise is not recommended, as an attacker obtaining B can authenticate as the user.

[7.](#) References

[7.1.](#) Normative References

- [ABNF] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", [RFC 4234](#), October 2005.
- [HMAC] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", [RFC 2104](#), February 1997.
- [KEYWORDS] Bradner, S., "Key words for use in RFCs to Indicate

Requirement Levels", [BCP 14](#), [RFC 2119](#), March 1997.

Internet-Draft

SASL HEXA

February 2007

- [MD5] Rivest, R., "The MD5 Message-Digest Algorithm", [RFC 1321](#), April 1992.
- [SASL] Melnikov, A. and K. Zeilenga, "Simple Authentication and Security Layer (SASL)", [RFC 4422](#), June 2006.
- [SASLPREP] Zeilenga, K., "SASLprep: Stringprep Profile for User Names and Passwords", [RFC 4013](#), February 2005.
- [UTF-8] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, [RFC 3629](#), November 2003.

[7.2](#). Informative References

- [CRAM-MD5] Klensin, J., Catoe, R., and P. Krumviede, "IMAP/POP AUTHorize Extension for Simple Challenge/Response", [RFC 2195](#), September 1997.
- [DIGEST-MD5] Leach, P. and C. Newman, "Using Digest Authentication as a SASL Mechanism", [RFC 2831](#), May 2000.
- [PLAIN] Zeilenga, K., "The PLAIN Simple Authentication and Security Layer (SASL) Mechanism", [RFC 4616](#), August 2006.

Authors' Addresses

Dave Cridland
Isode Limited
5 Castle Business Village
36, Station Road
Hampton, Middlesex TW12 2BX
GB

Email: dave.cridland@isode.com

Internet-Draft

SASL HEXA

February 2007

Alexey Melnikov
Isode Limited
5 Castle Business Village
36, Station Road
Hampton, Middlesex TW12 2BX
GB

Email: alexey.melnikov@isode.com

Internet-Draft

SASL HEXA

February 2007

Full Copyright Statement

Copyright (C) The IETF Trust (2007).

This document is subject to the rights, licenses and restrictions contained in [BCP 78](#), and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY, THE IETF TRUST AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Intellectual Property

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information

on the procedures with respect to rights in RFC documents can be found in [BCP 78](#) and [BCP 79](#).

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at <http://www.ietf.org/ipr>.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Acknowledgment

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).