

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 8, 2013

S. Farrell
Trinity College Dublin
P. Hoffman
VPN Consortium
M. Thomas
Phresheez
October 5, 2012

HTTP Origin-Bound Authentication (HOBA)
draft-farrell-httpbis-hoba-02

Abstract

HTTP Origin-Bound Authentication (HOBA) is a design for an HTTP authentication method with credentials that are not vulnerable to phishing attacks, and that does not require a server-side password database. The design can also be used in Javascript-based authentication embedded in HTML. HOBA is an alternative to HTTP authentication schemes that require passwords with all the negative attributes that come with password-based systems. HOBA can be integrated with account management and other applications running over HTTP and supports portability, so a user can associate more than one device or origin-bound key with the same service. We also describe a way in which the HOBA design can be used from a Javascript web client. When deployed, HOBA will be a drop-in replacement for password-based HTTP authentication or JavaScript authentication.

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 April 8, 2013.

Copyright Notice

Copyright (c) 2012 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.

Table of Contents

1. Introduction	4
1.1. Comparison of HOBA and Current Password Authentication . .	5
1.2. Terminology	5
2. HOBA for Both HTTP Authentication and JavaScript	6
3. HOBA HTTP Authentication Mechanism	7
4. Using HOBA-http	7
4.1. CPK Preparation Phase	8
4.2. Signing Phase	8
4.3. Authentication Phase	8
4.4. Logging in on a New User Agent	9
5. Using HOBA-js	9
5.1. Key Storage	10
5.2. User Join	10
5.3. User Login	10
5.4. Enrolling a New User Agent	11
5.5. Replay Protection	11
5.6. Signature Parameters	12
5.7. Session Management	13
5.8. Multiple Accounts on One User Agent	14
5.9. Oddities	14
6. Additional Services	14
6.1. Registration	14
6.2. Associating Additional Keys to an Existing Account . . .	15
6.3. Logging Out	16
7. Mandatory-to-Implement Algorithms	16
8. Security Considerations	16
8.1. localStorage Security for Javascript	16
9. IANA Considerations	17
9.1. HOBA Authentication Scheme	17
9.2. .well-known URLs	17
9.3. Hash names	18
10. Acknowledgements	18
11. References	18
11.1. Normative References	18
11.2. Informative References	18
Appendix A. Problems with Passwords	19

Authors' Addresses	19
------------------------------	----

1. Introduction

[[Commentary is in double-square brackets, like this. As you'll see there are a bunch of details still to be figured out. Feedback on those is very welcome. Also note that the authors fully expect that the description of HOBA-http and HOBA-js to be mostly merged in the draft; they're both here now so readers can see some alternatives and maybe support particular proposals.]]

HTTP Origin-Bound Authentication (HOBA) is a proposal for a new authentication design that can be used as an HTTP authentication scheme and for Javascript-based authentication embedded in HTML. The main goal of HOBA is to offer an easy-to-implement authentication scheme that is not based on passwords. If deployment of HOBA reduces the number of password entries in databases by any appreciable amount, then it would be worthwhile. As an HTTP authentication scheme, it would work in the current HTTP 1.0 and HTTP 1.1 authentication framework, and will very likely work with whatever changes are made to the HTTP authentication scheme in HTTP 2.0. As a JavaScript design, HOBA demonstrates a way for clients and servers to interact using the same credentials that are used by the HTTP authentication scheme.

The HTTP specification defines basic and digest authentication methods for HTTP that have been in use for many years, but which, being based on passwords, are susceptible to theft of server-side databases. (See [RFC2617] for the original specification, and [I-D.ietf-httpbis-p7-auth] for clarifications and updates to the authentication mechanism.) Even though few large web sites use basic and digest authentication, they still use username/password authentication and thus have large susceptible server-side databases of passwords.

Instead of passwords, HOBA uses digital signatures as an authentication mechanism. HOBA also adds useful features such as credential management and session logout. In HOBA, the client creates a new public-private key pair for each host ("web-origin") to which it authenticates; web-origins are defined in [RFC6454]. These keys are used in HOBA for HTTP clients to authenticate themselves to servers in the HTTP protocol or in a Javascript authentication program. HOBA keys need not be stored in public key certificates, but instead in `subjectPublicKeyInfo` structures from PKIX [RFC5280]. Because these are generally "bare keys", there is none of the semantic overhead of PKIX certificates, particularly with respect to naming and trust anchors. Thus, client public keys ("CPKs") do not have any publicly-visible identifier for the user who possesses the corresponding private key, nor the web-origin with which the client is using the CPK.

HOBA also defines some services that are required for modern HTTP authentication:

- o Servers can bind a CPK with an identifier, such as an account name. HOBA allows servers to define their own policies for binding CPKs with accounts during account registration.
- o Users are likely to use more than one device or user agent (UA) for the same HTTP based service, so HOBA gives a way to associate more than one CPK to the same account, but without having to register for each separately.
- o Users are also likely to lose a private key, or the client's memory of which key pair is associated with which origin. For example if a user loses the computer or mobile device in which state is stored. HOBA allows for clients to tell servers to delete the association between a CPK and an account.
- o Logout features can be useful for user agents, so HOBA defines a way to close a current HTTP "session", and also a way to close all current sessions, even if more than one session is currently active from different user agents for the same account.

1.1. Comparison of HOBA and Current Password Authentication

[[This will be a few paragraphs explaining how HOBA can be used as a drop-in replacement for the common form-and-cookie authentication used today. It will show how similar many of the concepts are, and also point out some of the advantages sites will get by changing to HOBA.]]

1.2. Terminology

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 RFC 2119 [RFC2119].

A client public key ("CPK") is the public key and associated cryptographic parameters needed for a server to validate a signature.

The term "account" is (loosely) used to refer to whatever data structure(s) the server maintains that are associated with an identity. That will contain of at least one CPK and a web-origin; it will also optionally include an HTTP "realm" as defined in the HTTP authentication specification. It might also involve many other non-standard pieces of data that the server accumulates as part of account creation processes. An account may have many CPKs that are considered equivalent in terms of being usable for authentication,

but the meaning of "equivalent" is really up to the server and is not defined here.

When describing something that is specific to HOBA as an HTTP authentication mechanism or HOBA as a JavaScript implementation, this document uses the terms "HOBA-http" and "HOBA-js", respectively.

Web client: the content and javascript code that run within the context of a single user agent instance (such as a tab in a web browser).

User agent (UA): typically, but not always, a web browser doing HOBA.

User: a person who is running a UA. In this document, "user" does not mean "user name" or "account name".

[[This specification may later use the Augmented Backus-Naur Form (ABNF) notation of [RFC5234]. Or maybe not.]]

2. HOBA for Both HTTP Authentication and JavaScript

A UA that implements HOBA maintains a list of web-origins and realms. The UA also maintains one or more client credentials for each web-origin/realm combination for which it has created a CPK.

[[We've discussed whether or not realms are needed. They may disappear if they're not.]]

On receipt of a challenge from a server, the client marshals a to-be-signed blob that includes the web-origin name, the realm, and the challenge string; and signs that hashed blob using the hash algorithm identified with the challenge and the private key corresponding to the CPK for that web-origin. The client concatenates the signed blob with the CPK identifier that the client and host agreed on for the client. This is called the "client response". [[Note: this is just an illustrative first cut at a challenge-response protocol, real design and analysis is needed for this, e.g. for security, algorithm agility, etc. Ideally we can just adopt something that already has some security proofs. Expect changes here.]]

HOBA will support the idea of multiple users on the same user agent. This will be useful for the problem of "can I use your browser to check my mail..." and so on. It is [[currently]] described only in the HOBA-js section, but will apply equally to HOBA-http. [[There are implications beyond the discussion in HOBA-js here in that there would only be a single CPK for a set of users for a given origin since normative HOBA-http has no clue at all about users and the

like. This needs more thought.]]

3. HOBA HTTP Authentication Mechanism

An HTTP server that supports HOBA authentication includes the "hoba" auth-scheme value in a WWW-Authenticate header field when it wants the client to authenticate with HOBA.

- o If the "hoba" scheme is listed, it MUST be followed by two or more auth-param values. The auth-param attributes defined by this specification are below. Other auth-param attributes MAY be used as well. Unknown auth-param attributes MUST be ignored by clients, if present.
- o The "challenge" attribute MUST be included. The challenge is a string of characters that the server wants the client to sign in its response. The challenge SHOULD be unique for every HTTP 401 response in order to prevent replay attacks from passive observers. [[How or if replay detection is specified is TBD.]]
- o The "hash" attribute MUST be included. This is the name of the hash algorithm that the server wants the client to use in signing challenge. The valid names for hash algorithms are listed in [[some IANA registry, maybe same as DKIM]].
- o Note that a hash here is sufficient, given the assumption that the client and server have already agreed (or are about to agree) the public key and asymmetric algorithm for the CPK.
- o A "realm" attribute MAY be included to indicate the scope of protection in the manner described in HTTP/1.1, Part 7 [I-D.ietf-httpbis-p7-auth]. The "realm" attribute MUST NOT appear more than once.

When the "client response" is created, the HOBA-http client encodes the result as a b64token and returns that b64token in the Authorization header.

The HOBA-http authentication mechanism allows for the use of cookies for preserving state between protected resources in one HTTP realm. This means that the server need only send the WWW-Authenticate header field once, and can rely on cookie management for keeping state.

4. Using HOBA-http

[[A lot of this is similar to the HOBA-js discussion below. At some

point some nuclear fusion might be nice, but for now it might be best to keep them separate until we understand better what can be merged, and what is different.]]

The interaction between an HTTP client and HTTP server using HOBA happens in three phases: the CPK preparation phase, the signing phase, and the authentication phase. The first and second phase are done in a standard fashion; the third is done using site-specific methods.

[[Need to describe what happens if the user bails half way through the flow.]]

4.1. CPK Preparation Phase

In the CPK preparation phase, the client determines if it already has a CPK for the web-origin it is going to. If the has a CPK, the client will use it; if the client does not have a CPK, it generates one in anticipation of the server asking for one.

4.2. Signing Phase

In the signing phase, the client connects to the server, the server asks for HOBA-based authentication, and the client authenticates by signing a blob of information as described in the previous sections.

The user agent tries to access a protected resource on the server. The server sends the HOBA WWW-Authenticate challenge. The user agent receives the challenge and signs the challenge using the CPK it either already had or just generated. The server validates the signature. If validation fails, the server aborts the transaction. [[Or maybe it asks again?]]

4.3. Authentication Phase

In the authentication phase, the server extracts the CPK from the signing phase and decides if it recognizes the CPK. If the server recognizes the CPK, the server may finish the client authentication process. If the process involves a second factor of authentication, such as asking the user which account it wants to use (in the case where a user agent is used for multiple accounts on a site), the server may prompt the user for the account identifying information. None of this is standardized: it all follows the server's security policy and session flow. At the end of this, the server probably assigns or updates a session cookie for the client.

If the server does not recognize the CPK the server might send the client through a either a join or login-new-user-agent (see below)

process. This process is completely up to the server, and probably entails using HTML, JavaScript and CSS to ask the user some questions in order to assess whether or not the server wants to give the client an account. Completion of the joining process might entail require confirmation by email, SMS, Captcha, and so on.

Note that there is no necessity for the server to initiate a joining or login process upon completion of the signing phase. Indeed, the server may desire to challenge the user agent even for unprotected resources and carry along the CPK in a session cookie for later use in a join or login process as it becomes necessary. For example, a server might only want to offer an account to someone who had been to a few pages on the web site; in such a case, the server could use the CPK from an associated session cookie as a way of building reputation for the user until the server wants the user to join.

After the UA is authenticated (if the user had to join, this could be the last step of joining), the server gives the UA access to the protected resource that was originally requested at the beginning of the signing phase. It is quite likely that the server would also update the UA's session cookie for the web site.

4.4. Logging in on a New User Agent

When a user wants to use a new user agent for an existing account, the flows are similar to logging in with an already-joined UA or joining for the first time. In fact, the CPK preparation phase (with the UA knowing that it needs to create a new CPK) and the signing phase are identical.

During the authentication phase, the server could use HTML, JavaScript and CSS to ask the user if they are really a new user or want to associate this new CPK with an already-joined CPK. The server can then use some out-of-band method (such as a confirmation email round trip, SMS, or an UA that is already enrolled) to verify that the "new" user is the same as the already-enrolled one.

5. Using HOBA-js

[[A description of how to use the same HOBA semantics, but doing everything in Javascript in a web page. This is more of a demonstration that you could get the similar semantics via JS rather than a normative section.]]

Web sites using javascript can also perform origin-bound authentication without needing to involve the http layer, and by inference not needing HOBA-specific support in browsers. One element

is required: `localStorage` (see <http://www.w3.org/TR/webstorage/>), and one when it is available will be highly desirable: `WebCrypto` (see <http://www.w3.org/TR/WebCryptoAPI>). In lieu of `WebCrypto`, javascript crypto libraries can be employed with the known deficiencies of PRNG, and the general immaturity of those libraries. The following section outlines a mechanism for Javascript HOBA clients to initially enroll, subsequent enrollment on new clients, login, and how HOBA-js relates to web based session management. As with HOBA-http, a pure Javascript implementation retains the property that only CPKs are stored on the server, so that server compromise doesn't suffer the multiplier affect that the various recent password exposure debacles have vividly demonstrated.

5.1. Key Storage

We use the new HTML 5 `webstorage` feature that is now widely available. Conceptually an implementation stores in the origin's `localStorage` dictionary account identifier, public key, private key tuples for subsequent authentication requests. How this is actually stored in `localStorage` is an implementation detail. We rely on the security properties of the same-origin policy that `localStorage` enforces. See the security considerations for discussion about attacks on `localStorage`.

5.2. User Join

To join a web site, the HOBA-js client generates a public/private key pair and takes as input the account identifier to which the key pair should be bound. The key pair and account identifier are stored in `localStorage` for later use. The user agent then signs the join information (see below) using the private key, and forms a message with the public key (CPK) and the signed data. The server receives the message and verifies the signed data using the supplied key. The server creates the account and adds the public key to a list of public keys associated with this account.

5.3. User Login

Each time the user needs to log in to the server, it creates a login message (see below) and signs the message using the relevant private key stored in `localStorage`. The signed login message along with the associated CPK identifier is sent to the server. The server receives the message and verifies the signed data. If the supplied public key is amongst the set of valid public keys for the supplied account, then the login proceeds. See below for a discussion about replay.

5.4. Enrolling a New User Agent

When a user wants to start using a different UA, the website has two choices: use a currently enrolled UA to permit the enrollment or use a trusted out of band mechanism (eg email, sms, etc). To enroll a new UA using an existing UA, the web site can display a one-time password on the currently enrolled UA. This password is a one-time password and expires in a fixed amount of time (say, 30 minutes). It doesn't need to be an overly fussy password since it's one-time and times out quickly. The user then inputs the one-time password and the new UA generates a new asymmetric key pair and includes the one-time password in the login message to the server (see below).

Alternatively if an enrolled UA is not available, and the site has an out of band communication mechanism (eg, sms, email, etc) a user can request that a one-time password be sent to the user. The server generates and stores the one-time password as above. The user receives the one-time password, inputs as above on the new UA, and the HOBA-js client forms the login message as above.

In both cases, when the server receives a login message with a one-time password, it checks to see if the password supplied is in a list of unexpired one-time passwords associated with that account. If the password matches, the server verifies the signature, expires or deletes the one-time password and adds the supplied public key to the list of public keys associated with the user assuming the signature verified correctly. Subsequent logins proceed as above in User Login.

5.5. Replay Protection

To guard against replay of a legitimate login/join message, we use Kerberos-like timestamps in the expectation of synchronization between the browser's and server's clocks is sufficiently reliable. This saves an HTTP round trip which is desirable, though a challenge-response mechanism as in HOBA-http could also be used. The client puts the current system time into the URL, and the server side vets it against its system time. Like Kerberos, a replay cache covering a signature timeout window is required on the server. This can be done using a database table that is keyed (in the database sense of the term) using the signature bits. If the signature is in the replay table, it ought be rejected. If the timestamp in the signature is outside the current replay cache window then it also gets rejected.

[[An addition of the ability for the server to reject a client with potential time skew and give it a nonce (as with HOBA-http) would allow the size of the replay cache to be set to just a few minutes rather than a much longer period. Or the HOBA server could always

use a nonce method. This is worthy of more discussion.]].

5.6. Signature Parameters

Since we only require agreement between the server and the client where the client is under the control of the server, the actual url parameter names here are only advisory. For each signed url, the client forms a url with the necessary login/join information. For example, suppose example.com has login and join scripts with various parameters:

- o `http://example.com/site/login.php?username=Mike`

- o `http://example.com/site/
join.php?username=Mike&email=mike@example.com&sms=555.1212`

The client then appends a signature parameter block to the url:

- o `curtime`: the time in milliseconds since unix epoch (ie, `new Date().getTime()`).

- o `pubkey`: the url encoded public key. See DKIM for the format of the base64 encoded PEM formatted key.

- o `temppass`: an optional url encoded one-time password for subsequent enrollment.

- o `keyalg`: currently RSA. 2048 bit keys should be use if WebCrypto is available

- o `digestalg`: currently SHA1. SHA256 should be used if WebCrypto is available.

- o `signature`: empty for signing canonicalization purposes

[[Signing the full url is problematic with PHP; we should take a clue from what OAUTH does here; we almost certainly need to add some host identifying information...]] To create the signature, the canonical text includes the path portion, the site-specific url parameters and appends a signature block onto the end of the url. The signature block consists of the parameters listed above with an empty signature parameter (ie, `signature=`), eg:

- o Login: `/site/
login.php?username=Mike&curtime=1234567890.1234&keyalg=RSA&
digestalg=SHA1&signature=`

- o Join: /site/
join.php?username=Mike&email=mike@
example.com&curtime=1234567890.1234&keyalg=RSA&digestalg=SHA1&
signature=
- o Login New User Agent: /site/
login.php?username=Mike&curtime=1234567890.1234&temppass=1239678&
keyalg=RSA&digestalg=SHA1&signature=

The canonical signature text is then signed with the private key associated with the account. The signature is then base64 encoded and appended to the full url, and sent to the server using XMLHttpRequest as usual. On receipt of the login request, the server first extracts the timestamp (curtime) and determines whether the timestamp is fresh (see above) rejecting the request if stale. The server then removes the scheme and domain:port portion of the incoming url, and removes the signature value only to create the canonical signature text. The server then extracts the public key along with the account and verifies the signature. If the signature verifies, the server then determines whether this is an enrolled public key for the user. If it is, login/join succeeds. If the key is not enrolled, the server then checks to see if a one-time password was supplied. If not, login/join fails. If a one-time password was supplied, the server checks to see if a one-time password is valid and fails if not. If valid, the server disables the one-time password (eg, deletes it from its database) and adds the new public key to the list of enrolled public keys for this user.

Once verified, the server may start up normal cookie-based session management (see below). The server should send back status to the HOBA-js client to determine whether the login/join was successful. The details are left as an implementation detail.

Note: the client SHOULD use an HTTP POST for the XMLHttpRequest as both the public key and signature blocks may exhaust the maximum size for a GET request (typically around 2KB).

5.7. Session Management

Session Management is identical to username/password session management. That is, the session management tool (such as PHP, Python CGI, and so on) inserts a session cookie into the output to the browser, and logging out simply removes the session cookie. HOBA-js does nothing to help or hurt session cookie hijacking -- TLS is still our friend.

5.8. Multiple Accounts on One User Agent

A shared UA with multiple accounts is possible if the account identifier is stored along with the asymmetric key pair binding them to one another. Multiple entries can be kept, one for each account, and selected by the current user. This, of course, is fraught with the possibility for abuse, since you're enrolling the device potentially long-term. A couple of things can possibly be done to combat that. First, the user can request that the credential be erased from keystore. Similarly, in the enrollment phase, a user could request that the key pair only be kept for a certain amount of time, or that it not be stored at all. Last, it's probably best to just not use shared devices at all since that's never especially safe.

5.9. Oddities

With the same-origin policy, subdomains do not have access to the same localStorage as parent domains do. For larger/more complex sites this could be an issue that requires enrollment into subdomains with the requisite hassle for users. One way to get around this is to use session cookies as they can be used across subdomains. That is, login using a single well-known domain, and then use session cookies to navigate around a site.

6. Additional Services

HOBA uses a well-known URL [RFC5785] "hoba" as a base URI for performing many tasks: "https://www.example.com/.well-known/hoba". These URLs are based on the name of the host that the HTTP client is accessing. There are many use cases for these URLs to redirect to other URLs: a site that does registration through a federated site, a site that only does registration under HTTPS, and so on. Like any HTTP client, HOBA clients MUST be able to handle redirection of these URLs. [[There are a bunch of security issues to consider related to cases where a re-direct brings you off-origin.]]

All additional services MUST be done in TLS-protected sessions ([RFC5246]).

6.1. Registration

Normally, a registration is expected to happen after a UA receives a WWW-Authenticate for a web-origin and realm for which it has no associated CPK. The (protocol part of the) process of registration for a HOBA account on a server is relatively light-weight. The UA generates a new key pair, and associates it with the web-origin/realm

in question. The UA sets up a TLS-protected session, goes to the registration URL ".well-known/hoba/register", and submits the CPK using a POST message. [[More description is clearly needed here.]] It is up to the server to decide what kind of user interaction is required before the account is finally set up.

If the UA has a CPK associated with the web-origin, but not for the realm concerned, then a new registration is REQUIRED. If the server did not wish for that outcome, then it ought not use a different realm.

The POST message sent to the registration URL has one parameter, called "cpksubmit", which contains the CPK that the UA will use for the origin/realm combination. The CPK MUST be sent base64 encoded. The value that is base64 encoded is the DER encoding of the `subjectPublicKeyInfo` structure that is the CPK. See [RFC5280] for details of that data structure.

6.2. Associating Additional Keys to an Existing Account

It is common for a user to have multiple UAs, and to want all those UAs to be able to authenticate to a single account. One method to allow a user who has an existing account to be able to authenticate on a second device is to securely transport the private and public keys and the origin information from the first device to the second. Previous history with such key transport has been spotty at best. As an alternative, HOBA allows associating a CPK from the second device to the account created on the first device.

Instead of registering on the new device, the UA generates a new key pair, associates it with the web-origin/realm in question, goes to the URL for starting an association, ".well-known/hoba/associate-start" in a TLS-protected session, and submits the new CPK using a POST message. [[More description is clearly needed here.]] The server's response to this request is a nonce with at least 128 bits of entropy. That nonce SHOULD be easy for the user to copy and type, such as using Base32 encoding (see [RFC4648]). The user then uses the first UA to log into the origin, goes to the URL for finishing an association, ".well-known/hoba/associate-finish", and submits the nonce using a POST message. [[More description is clearly needed here.]]. The server then knows that the authenticated user is associated with the second CPK. The server can choose to associate the two CPKs with one account. Whether to do so is entirely at the server's discretion however, but the server SHOULD make the outcome clear to the user.

6.3. Logging Out

When the user wishes to logout, the UA simply goes to ".well-known/hoba/logout". The UA MAY also delete session cookies associated with the session. [[Is that right?, maybe a SHOULD- or MUST-delete would be better]]

The server-side MUST NOT allow TLS session resumption for any logged out session and SHOULD also revoke or delete any cookies associated with the session.

7. Mandatory-to-Implement Algorithms

[[We should list two signature schemes (most likely RSA and ECDSA with P256). We should list two hash algorithms (most likely SHA-256 and SHA-384).]]

8. Security Considerations

If key binding was server-selected then a bad actor could bind different accounts belonging to the user from the network with possible bad consequences, especially if one of the private keys was compromised somehow.

Binding my CPK with someone else's account would be fun and profitable so SHOULD be appropriately hard. In particular the string generated by the server MUST be hard to guess, for whatever level of difficulty is chosen by the server. The server SHOULD NOT allow a random guess to reveal whether or not an account exists.

[[The potential impact on privacy of HOBA needs to be addressed. If a site can use a 401 and a CPK to track users without permission that would be not-so-nice so some guidance on how a UA could indicate to a user that HOBA stuff is going on might be needed.]]

[[lots more TBD, be nice to your private keys etc. etc.]]

8.1. localStorage Security for Javascript

Our use of localStorage will undoubtedly be a cause for concern. localStorage uses the same-origin model which says that the scheme, domain and port define a localStorage instance. Beyond that, any code executing will have access to private keying material. Of particular concern are XSS attacks which could conceivably take the keying material and use it to create user agents under the control of an attacker. But XSS attacks are in reality across the board

devastating since they can and do steal credit card information, passwords, perform illicit acts, etc, etc. It's not clear that we introduce unique threats from which clear text passwords don't already suffer.

Another source of concern is local access to the keys. That is, if an attacker has access to the UA itself, they could snoop on the key through a javascript console, or find the file(s) that implement localStorage on the host computer. Again it's not clear that we are worse in this regard because the same attacker could get at browser password files, etc too. One possible mitigation is to encrypt the keystore with a password/pin the user supplies. This may sound counter intuitive, but the object here is to keep passwords off of servers to mitigate the multiplier effect of a large scale compromise ala LinkedIn because of shared passwords across sites.

It's worth noting that HOBA uses asymmetric keys and not passwords when evaluating threats. As various password database leaks have shown, the real threat of a password breach is not just to the site that was breached, it's all of the sites a user used the same password on too. That is, the collateral damage is severe because password reuse is common. Storing a password in localStorage would also have a similar multiplier effect for an attacker, though perhaps on a smaller scale than a server-side compromise: one successful crack gains the attacker potential access to hundreds if not thousands of sites the user visits. HOBA does not suffer from that attack multiplier since each asymmetric key pair is unique per site/useragent/user.

9. IANA Considerations

9.1. HOBA Authentication Scheme

Authentication Scheme Name: hoba

Pointer to specification text: [[this document]]

Notes (optional): The HOBA scheme can be used with either HTTP servers or proxies. [[But we need to figure out the proxy angle;-)]]

9.2. .well-known URLs

TBD

9.3. Hash names

TBD, hopefully re-use and existing registry

We probably want a new registry for the labels beneath .well-known/hoba so that other folks can add additional features in a controlled way, e.g. for CPK/account revocation or whatever.

10. Acknowledgements

[[TBD]]

11. References

11.1. Normative References

- [I-D.ietf-httpbis-p7-auth]
Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Authentication", draft-ietf-httpbis-p7-auth-21 (work in progress), October 2012.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, April 2010.
- [RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, December 2011.

11.2. Informative References

- [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.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.

Appendix A. Problems with Passwords

By far the most common mechanism for web authentication is passwords that can be remembered by the user, called "memorizable passwords". There is plenty of good research on how users typically use memorizable passwords ([[handful of citations goes here]]), but some of the highlights are that users typically try hard to reuse passwords on as many web sites as possible, and that web sites often use either email addresses or users' names as the identifier that goes with these passwords.

If an attacker gets access to the database of memorizable passwords, that attacker can impersonate any of the users. Even if the breach is discovered, the attacker can still impersonate users until every password is changed. Even if all the passwords are changed or at least made unusable, the attacker now possesses a list of likely username/password pairs that might exist on other sites.

Using memorizable passwords on unencrypted channels also poses risks to the users. If a web site uses either the HTTP Plain authentication method, or an HTML form that does no cryptographic protection of the password in transit, a passive attacker can see the password and immediately impersonate the user. If a hash-based authentication scheme such as HTTP Digest authentication is used, a passive attacker still has a high chance of being able to determine the password using a dictionary of known passwords.

[[Say a bit about non-memorizable passwords. Still subject to database attack, although that doesn't give the attacker knowledge for other systems. Safe if digest authentication is used, but that's rare.]]

Authors' Addresses

Stephen Farrell
Trinity College Dublin
Dublin, 2
Ireland

Phone: +353-1-896-2354
Email: stephen.farrell@cs.tcd.ie

Paul Hoffman
VPN Consortium

Email: paul.hoffman@vpnc.org

Michael Thomas
Phresheez

Email: mike@phresheez.com

HTTPAUTH
Internet-Draft
Intended status: Standards Track
Expires: May 16, 2015

A. Melnikov
Isode Ltd
November 12, 2014

Salted Challenge Response (SCRAM) HTTP Authentication Mechanism
draft-melnikov-httpbis-scam-auth-01.txt

Abstract

The secure authentication mechanism most widely deployed and used by Internet application protocols is the transmission of clear-text passwords over a channel protected by Transport Layer Security (TLS). There are some significant security concerns with that mechanism, which could be addressed by the use of a challenge response authentication mechanism protected by TLS. Unfortunately, the HTTP Digest challenge response mechanism presently on the standards track failed widespread deployment, and have had success only in limited use.

This specification describes a family of HTTP authentication mechanisms called the Salted Challenge Response Authentication Mechanism (SCRAM), which addresses security concerns with HTTP Digest and meets the deployability requirements. When used in combination with TLS or an equivalent security layer, a mechanism from this family could improve the status-quo for HTTP authentication.

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 May 16, 2015.

Copyright Notice

Copyright (c) 2014 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. Conventions Used in This Document	2
1.1. Terminology	3
1.2. Notation	3
2. Introduction	5
3. SCRAM Algorithm Overview	5
4. SCRAM Mechanism Names	6
5. SCRAM Authentication Exchange	7
5.1. One round trip reauthentication	9
6. Formal Syntax	11
7. Security Considerations	12
8. IANA Considerations	13
9. Acknowledgements	14
10. Design Motivations	14
11. Open Issues	15
12. References	15
12.1. Normative References	15
12.2. Informative References	16
Author's Address	16

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 [RFC2119].

Formal syntax is defined by [RFC5234] including the core rules defined in Appendix B of [RFC5234].

Example lines prefaced by "C:" are sent by the client and ones prefaced by "S:" by the server. If a single "C:" or "S:" label applies to multiple lines, then the line breaks between those lines

are for editorial clarity only, and are not part of the actual protocol exchange.

1.1. Terminology

This document uses several terms defined in [RFC4949] ("Internet Security Glossary") including the following: authentication, authentication exchange, authentication information, brute force, challenge-response, cryptographic hash function, dictionary attack, eavesdropping, hash result, keyed hash, man-in-the-middle, nonce, one-way encryption function, password, replay attack and salt. Readers not familiar with these terms should use that glossary as a reference.

Some clarifications and additional definitions follow:

- o Authentication information: Information used to verify an identity claimed by a SCRAM client. The authentication information for a SCRAM identity consists of salt, iteration count, the "StoredKey" and "ServerKey" (as defined in the algorithm overview) for each supported cryptographic hash function.
- o Authentication database: The database used to look up the authentication information associated with a particular identity. For application protocols, LDAPv3 (see [RFC4510]) is frequently used as the authentication database. For network-level protocols such as PPP or 802.11x, the use of RADIUS [RFC2865] is more common.
- o Base64: An encoding mechanism defined in [RFC4648] which converts an octet string input to a textual output string which can be easily displayed to a human. The use of base64 in SCRAM is restricted to the canonical form with no whitespace.
- o Octet: An 8-bit byte.
- o Octet string: A sequence of 8-bit bytes.
- o Salt: A random octet string that is combined with a password before applying a one-way encryption function. This value is used to protect passwords that are stored in an authentication database.

1.2. Notation

The pseudocode description of the algorithm uses the following notations:

- o "!=": The variable on the left hand side represents the octet string resulting from the expression on the right hand side.
- o "+": Octet string concatenation.
- o "[]": A portion of an expression enclosed in "[" and "]" may not be included in the result under some circumstances. See the associated text for a description of those circumstances.
- o Normalize(str): Apply the SASLPrep profile [RFC4013] of the "stringprep" algorithm [RFC3454] as the normalization algorithm to a UTF-8 [RFC3629] encoded "str". The resulting string is also in UTF-8. When applying SASLPrep, "str" is treated as a "stored strings", which means that unassigned Unicode codepoints are prohibited (see Section 7 of [RFC3454]). Note that implementations MUST either implement SASLPrep, or disallow use of non US-ASCII Unicode codepoints in "str".
- o HMAC(key, str): Apply the HMAC keyed hash algorithm (defined in [RFC2104]) using the octet string represented by "key" as the key and the octet string "str" as the input string. The size of the result is the hash result size for the hash function in use. For example, it is 20 octets for SHA-1 (see [RFC3174]).
- o H(str): Apply the cryptographic hash function to the octet string "str", producing an octet string as a result. The size of the result depends on the hash result size for the hash function in use.
- o XOR: Apply the exclusive-or operation to combine the octet string on the left of this operator with the octet string on the right of this operator. The length of the output and each of the two inputs will be the same for this use.
- o Hi(str, salt, i):

```
U1    := HMAC(str, salt + INT(1))
U2    := HMAC(str, U1)
...
Ui-1  := HMAC(str, Ui-2)
Ui    := HMAC(str, Ui-1)

Hi := U1 XOR U2 XOR ... XOR Ui
```

where "i" is the iteration count, "+" is the string concatenation operator and INT(g) is a four-octet encoding of the integer g,

most significant octet first.

Hi() is, essentially, PBKDF2 [RFC2898] with HMAC() as the PRF and with dkLen == output length of HMAC() == output length of H().

2. Introduction

This specification describes a family of authentication mechanisms called the Salted Challenge Response Authentication Mechanism (SCRAM) which addresses the requirements necessary to deploy a challenge-response mechanism more widely than past attempts (see [RFC5802]). When used in combination with Transport Layer Security (TLS, see [RFC5246]) or an equivalent security layer, a mechanism from this family could improve the status-quo for HTTP authentication.

HTTP SCRAM is adoption of [RFC5802] for use in HTTP. (SCRAM data exchanged is identical to what is defined in [RFC5802].) It also adds 1 round trip reauthentication mode.

HTTP SCRAM provides the following protocol features:

- o The authentication information stored in the authentication database is not sufficient by itself (without a dictionary attack) to impersonate the client. The information is salted to prevent a pre-stored dictionary attack if the database is stolen.
- o The server does not gain the ability to impersonate the client to other servers (with an exception for server-authorized proxies).
- o The mechanism permits the use of a server-authorized proxy without requiring that proxy to have super-user rights with the back-end server.
- o Mutual authentication is supported, but only the client is named (i.e., the server has no name).

3. SCRAM Algorithm Overview

The following is a description of a full HTTP SCRAM authentication exchange. Note that this section omits some details, such as client and server nonces. See Section 5 for more details.

To begin with, the SCRAM client is in possession of a username and password (*) (or a ClientKey/ServerKey, or SaltedPassword). It sends the username to the server, which retrieves the corresponding authentication information, i.e. a salt, StoredKey, ServerKey and the iteration count i. (Note that a server implementation may choose to use the same iteration count for all accounts.) The server sends the

salt and the iteration count to the client, which then computes the following values and sends a ClientProof to the server:

(*) - Note that both the username and the password MUST be encoded in UTF-8 [RFC3629].

Informative Note: Implementors are encouraged to create test cases that use both username passwords with non-ASCII codepoints. In particular, it's useful to test codepoints whose "Unicode Normalization Form C" and "Unicode Normalization Form KC" are different. Some examples of such codepoints include Vulgar Fraction One Half (U+00BD) and Acute Accent (U+00B4).

```
SaltedPassword := Hi(Normalize(password), salt, i)
ClientKey      := HMAC(SaltedPassword, "Client Key")
StoredKey      := H(ClientKey)
AuthMessage    := client-first-message-bare + "," +
                  server-first-message + "," +
                  client-final-message-without-proof
ClientSignature := HMAC(StoredKey, AuthMessage)
ClientProof     := ClientKey XOR ClientSignature
ServerKey       := HMAC(SaltedPassword, "Server Key")
ServerSignature := HMAC(ServerKey, AuthMessage)
```

The server authenticates the client by computing the ClientSignature, exclusive-ORing that with the ClientProof to recover the ClientKey and verifying the correctness of the ClientKey by applying the hash function and comparing the result to the StoredKey. If the ClientKey is correct, this proves that the client has access to the user's password.

Similarly, the client authenticates the server by computing the ServerSignature and comparing it to the value sent by the server. If the two are equal, it proves that the server had access to the user's ServerKey.

For initial authentication the AuthMessage is computed by concatenating decoded "data" attribute values from the authentication exchange. The format of these messages is defined in [RFC5802].

4. SCRAM Mechanism Names

A SCRAM mechanism name (authentication scheme) is a string "SCRAM-" followed by the uppercased name of the underlying hash function taken from the IANA "Hash Function Textual Names" registry (see <http://www.iana.org>) .

For interoperability, all HTTP clients and servers supporting SCRAM MUST implement the SCRAM-SHA-1 authentication mechanism, [[CREF1: OPEN ISSUE: Possibly switch to SHA-256 as the mandatory-to-implement.]] i.e. an authentication mechanism from the SCRAM family that uses the SHA-1 hash function as defined in [RFC3174].

5. SCRAM Authentication Exchange

HTTP SCRAM is a HTTP Authentication mechanism whose client response (<credentials-scam>) and server challenge (<challenge-scam>) messages are text-based messages containing one or more attribute-value pairs separated by commas. The messages and their attributes are described below and defined in Section 6.

```
challenge-scam    = scam-name [1*SP 1#auth-param]
                   ; Complies with <challenge> ABNF from RFC 7235.
                   ; Included in the WWW-Authenticate header field.

credentials-scam  = scam-name [1*SP 1#auth-param]
                   ; Complies with <credentials> from RFC 7235.
                   ; Included in the Authorization header field.

scam-name = "SCRAM-SHA-1" / other-scam-name
           ; SCRAM-SHA-1 is registered by this RFC
other-scam-name = "SCRAM-" hash-name
                 ; hash-name is a capitalized form of names from IANA
                 ; "Hash Function Textual Names" registry.
                 ; Additional SCRAM names must be registered in both
                 ; the IANA "SASL mechanisms" registry
                 ; and the IANA "authentication scheme" registry.
```

This is a simple example of a SCRAM-SHA-1 authentication exchange when the client doesn't support channel bindings (username 'user' and password 'pencil' are used):

```
C: GET /resource HTTP/1.1
C: Host: server.example.com
C: [...]

S: HTTP/1.1 401 Unauthorized
S: WWW-Authenticate: Digest realm="realm1@host.com",
    Digest realm="realm2@host.com",
    Digest realm="realm3@host.com",
    SCRAM-SHA-1 realm="realm3@host.com",
    SCRAM-SHA-1 realm="testrealm@host.com"
S: [...]

C: GET /resource HTTP/1.1
C: Host: server.example.com
C: Authorization: SCRAM-SHA-1 realm="testrealm@host.com",
    data=base64(n,n=user,r=fyko+d2lbbFgONRv9qkxdawL)
C: [...]

S: HTTP/1.1 401 Unauthorized
S: WWW-Authenticate: SCRAM-SHA-1
    sid=AAAABBBBCCCCDDDD,
    data=base64(r=fyko+d2lbbFgONRv9qkxdawL3rfcNHYJY1ZVvWVs7j,
    s=QSXCR+Q6sek8bf92,i=4096)
S: [...]

C: GET /resource HTTP/1.1
C: Host: server.example.com
C: Authorization: SCRAM-SHA-1 sid=AAAABBBBCCCCDDDD,
    data=base64(c=biws,r=fyko+d2lbbFgONRv9qkxdawL3rfcNHYJY1ZVvWVs7j,
    p=v0X8v3Bz2T0CJGbJQyF0X+HI4Ts=)
C: [...]

S: HTTP/1.1 200 Ok
S: Authentication-Info: SCRAM-SHA-1
    sid=AAAABBBBCCCCDDDD,
    data=base64(v=rmF9pqV8S7suAoZWja4dJRkFsKQ=)
S: [...Other header fields and resource body...]
```

Note that in the example above the client can also initiate SCRAM authentication without first being prompted by the server.

Initial "SCRAM-SHA-1" authentication starts with sending the "Authorization" request header field defined by HTTP/1.1, Part 7 [RFC7235] containing "SCRAM-SHA-1" authentication scheme and the following attributes:

- o A "realm" attribute MAY be included to indicate the scope of

protection in the manner described in HTTP/1.1, Part 7 [RFC7235]. As specified in [RFC7235], the "realm" attribute MUST NOT appear more than once. The realm attribute only appears in the first SCRAM message to the server and in the first SCRAM response from the server.

- o The client also includes the data attribute that contains base64 encoded "client-first-message" [RFC5802] containing:
 - * a header consisting of a flag indicating whether channel binding is supported-but-not-used, not supported, or used . Note that the header always starts with "n", "y" or "p", otherwise the message is invalid and authentication MUST fail.
 - * SCRAM username and a random, unique nonce attributes.

In HTTP response, the server sends WWW-Authenticate header field containing: a unique session identifier (the "sid" attribute) plus the "data" attribute containing base64-encoded "server-first-message" [RFC5802]. The "server-first-message" contains the user's iteration count *i*, the user's salt, and the nonce with a concatenation of the client-specified one with a server nonce. [[CREF2: OPEN ISSUE: Alternatively, the "sid" attribute can be another header field.]]

The client then responds with another HTTP request with the Authorization header field, which includes the "sid" attribute received in the previous server response, together with the "data" attribute containing base64-encoded "client-final-message" data. The latter has the same nonce and a ClientProof computed using the selected hash function (e.g. SHA-1) as explained earlier.

The server verifies the nonce and the proof, and, finally, it responds with a 200 HTTP response with the Authentication-Info header field containing the "data" attribute containing base64-encoded "server-final-message", concluding the authentication exchange.

The client then authenticates the server by computing the ServerSignature and comparing it to the value sent by the server. If the two are different, the client MUST consider the authentication exchange to be unsuccessful and it might have to drop the connection.

5.1. One round trip reauthentication

If the server supports SCRAM reauthentication, the server sends in its initial HTTP response a WWW-Authenticate header field containing: the "realm" attribute (as defined earlier), the "sr" attribute that contains the server part of the "r" attribute (see [RFC5802] and optional "ttl" attribute (which contains the "sr" value validity in

seconds).

If the client has authenticated to the same realm before (i.e. it remembers "i" and "s" attributes for the user from earlier authentication exchanges with the server), it can respond to that with "client-final-message".

If the server considers the server part of the nonce (the "r" attribute) to be still valid, it will provide access to the requested resource (assuming the client hash verifies correctly, of course). However if the server considers that the server part of the nonce is stale (for example if the "sr" value is used after the "ttl" seconds), the server returns "401 Unauthorized" containing the SCRAM mechanism name with a new "sr" and optional "ttl" attributes.

When constructing AuthMessage Section 3 to be used for calculating client and server proofs, "client-first-message-bare" and "server-first-message" are reconstructed from data known to the client and the server.

Reauthentication can look like this:

```
C: GET /resource HTTP/1.1
C: Host: server.example.com
C: [...]

S: HTTP/1.1 401 Unauthorized
S: WWW-Authenticate: Digest realm="realm1@host.com",
    Digest realm="realm2@host.com",
    Digest realm="realm3@host.com",
    SCRAM-SHA-1 realm="realm3@host.com",
    SCRAM-SHA-1 realm="testrealm@host.com", sr=3rfcNHYJY1ZVvWVs7j
    SCRAM-SHA-1 realm="testrealm2@host.com", sr=AAABBBCCCCDDD, ttl=120
S: [...]
```

[Client authenticates as usual to realm "testrealm@host.com"]

[Some time later client decides to reauthenticate.
It will use the cached "i" and "s" from earlies exchanges.
It will use the server advertised "sr" value as the server part of the "r".
Should some counter be added to make "sr" unique for each reauth???]

```
C: GET /resource HTTP/1.1
C: Host: server.example.com
C: Authorization: SCRAM-SHA-1 realm="testrealm@host.com",
    data=base64(c=biws,r=fyko+d2lbbFgONRv9qkxdawL3rfcNHYJY1ZVvWVs7j,
    p=v0X8v3Bz2T0CJGbJQyF0X+HI4Ts=)
C: [...]

S: HTTP/1.1 200 Ok
S: Authentication-Info: SCRAM-SHA-1
    sid=AAAABBBBCCCCDDD,
    data=base64(v=rmF9pqV8S7suAoZWja4dJRkFsKQ=)
S: [...Other header fields and resource body...]
```

6. Formal Syntax

The following syntax specification uses the Augmented Backus-Naur Form (ABNF) notation as specified in [RFC5234]. "UTF8-2", "UTF8-3" and "UTF8-4" non-terminal are defined in [RFC3629].

ALPHA = <as defined in RFC 5234 appendix B.1>

DIGIT = <as defined in RFC 5234 appendix B.1>

base64-char = ALPHA / DIGIT / "/" / "+"

base64-4 = 4base64-char

base64-3 = 3base64-char "="

base64-2 = 2base64-char "=="

base64 = *base64-4 [base64-3 / base64-2]

sr = "sr=" s-nonce
;; s-nonce is defined in RFC 5802.

data = "data=" base64
;; The data attribute value is base-64 encoded
;; SCRAM challenge or response defined in
;; RFC 5802.

t1 = "t1" = 1*DIGIT
;; "sr" value validity in seconds.
;; No leading 0s.

sid = "sid=" <...>

realm = "realm=" <...as defined in HTTP Authentication...>

7. Security Considerations

If the authentication exchange is performed without a strong security layer (such as TLS with data confidentiality), then a passive eavesdropper can gain sufficient information to mount an offline dictionary or brute-force attack which can be used to recover the user's password. The amount of time necessary for this attack depends on the cryptographic hash function selected, the strength of the password and the iteration count supplied by the server. An external security layer with strong encryption will prevent this attack.

If the external security layer used to protect the SCRAM exchange uses an anonymous key exchange, then the SCRAM channel binding mechanism can be used to detect a man-in-the-middle attack on the security layer and cause the authentication to fail as a result. However, the man-in-the-middle attacker will have gained sufficient information to mount an offline dictionary or brute-force attack.

For this reason, SCRAM allows to increase the iteration count over time. (Note that a server that is only in possession of "StoredKey" and "ServerKey" can't automatic increase the iteration count upon successful authentication. Such increase would require resetting user's password.)

If the authentication information is stolen from the authentication database, then an offline dictionary or brute-force attack can be used to recover the user's password. The use of salt mitigates this attack somewhat by requiring a separate attack on each password. Authentication mechanisms which protect against this attack are available (e.g., the EKE class of mechanisms). RFC 2945 [RFC2945] is an example of such technology.

If an attacker obtains the authentication information from the authentication repository and either eavesdrops on one authentication exchange or impersonates a server, the attacker gains the ability to impersonate that user to all servers providing SCRAM access using the same hash function, password, iteration count and salt. For this reason, it is important to use randomly-generated salt values.

SCRAM does not negotiate a hash function to use. Hash function negotiation is left to the HTTP authentication mechanism negotiation. It is important that clients be able to sort a locally available list of mechanisms by preference so that the client may pick the most preferred of a server's advertised mechanism list. This preference order is not specified here as it is a local matter. The preference order should include objective and subjective notions of mechanism cryptographic strength (e.g., SCRAM with a successor to SHA-1 may be preferred over SCRAM with SHA-1).

SCRAM does not protect against downgrade attacks of channel binding types. The complexities of negotiation a channel binding type, and handling down-grade attacks in that negotiation, was intentionally left out of scope for this document.

A hostile server can perform a computational denial-of-service attack on clients by sending a big iteration count value.

See [RFC4086] for more information about generating randomness.

8. IANA Considerations

New mechanisms in the SCRAM- family are registered according to the IANA procedure specified in [RFC5802].

Note to future SCRAM- mechanism designers: each new SCRAM- HTTP authentication mechanism MUST be explicitly registered with IANA and

MUST comply with SCRAM- mechanism naming convention defined in Section 4 of this document.

IANA is requested to add the following entry to the Authentication Scheme Registry defined in HTTP/1.1, Part 7 [RFC7235]:

Authentication Scheme Name: SCRAM-SHA-1
Pointer to specification text: [[this document]]
Notes (optional): (none)

9. Acknowledgements

This document benefited from discussions on the HTTPAuth, SASL and Kitten WG mailing lists. The authors would like to specially thank co-authors of [RFC5802] from which lots of text was copied.

Thank you to Martin Thomson for the idea of adding "ttl" attribute.

Special thank you to Tony Hansen for doing an early implementation and providing extensive comments on the draft.

10. Design Motivations

The following design goals shaped this document. Note that some of the goals have changed since the initial version of the document.

- o The HTTP authentication mechanism has all modern features: support for internationalized usernames and passwords, support for channel bindings.
- o The protocol supports mutual authentication.
- o The authentication information stored in the authentication database is not sufficient by itself to impersonate the client.
- o The server does not gain the ability to impersonate the client to other servers (with an exception for server-authorized proxies), unless such other servers allow SCRAM authentication and use the same salt and iteration count for the user.
- o The mechanism is extensible, but [hopefully] not overengineered in this respect.
- o Easier to implement than HTTP Digest in both clients and servers.

11. Open Issues

Mandatory to implement SCRAM mechanism? Probably will switch to SHA-256

Should "sid" directive be an attribute or a new HTTP header field shared with other HTTP authentication mechanisms?

Username/password normalization algorithm needs to be picked.

12. References

12.1. Normative References

- [RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC3174] Eastlake, D. and P. Jones, "US Secure Hash Algorithm 1 (SHA1)", RFC 3174, September 2001.
- [RFC3454] Hoffman, P. and M. Blanchet, "Preparation of Internationalized Strings ("stringprep")", RFC 3454, December 2002.
- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.
- [RFC4013] Zeilenga, K., "SASLprep: Stringprep Profile for User Names and Passwords", RFC 4013, February 2005.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.
- [RFC5056] Williams, N., "On the Use of Channel Bindings to Secure Channels", RFC 5056, November 2007.
- [RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.
- [RFC5802] Newman, C., Menon-Sen, A., Melnikov, A., and N. Williams, "Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms", RFC 5802, July 2010.

- [RFC5929] Altman, J., Williams, N., and L. Zhu, "Channel Bindings for TLS", RFC 5929, July 2010.
- [RFC7235] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, June 2014.

12.2. Informative References

- [RFC2865] Rigney, C., Willens, S., Rubens, A., and W. Simpson, "Remote Authentication Dial In User Service (RADIUS)", RFC 2865, June 2000.
- [RFC2898] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, September 2000.
- [RFC2945] Wu, T., "The SRP Authentication and Key Exchange System", RFC 2945, September 2000.
- [RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.
- [RFC4510] Zeilenga, K., "Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map", RFC 4510, June 2006.
- [RFC4616] Zeilenga, K., "The PLAIN Simple Authentication and Security Layer (SASL) Mechanism", RFC 4616, August 2006.
- [RFC4949] Shirey, R., "Internet Security Glossary, Version 2", RFC 4949, August 2007.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [tls-server-end-point] Zhu, L., , "Registration of TLS server end-point channel bindings", IANA <http://www.iana.org/assignments/channel-binding-types/tls-server-end-point>, July 2008.

Author's Address

Alexey Melnikov
Isode Ltd

Email: Alexey.Melnikov@isode.com

Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: December 6, 2012

Y. Oiwa
H. Watanabe
H. Takagi
RISEC, AIST
B. Kihara
T. Hayashi
Lepidum
Y. Ioku
Yahoo! Japan
June 4, 2012

Mutual Authentication Protocol for HTTP
draft-oiwa-http-mutualauth-12

Abstract

This document specifies a mutual authentication method for the Hypertext Transport Protocol (HTTP). This method provides a true mutual authentication between an HTTP client and an HTTP server using password-based authentication. Unlike the Basic and Digest authentication methods, the Mutual authentication method specified in this document assures the user that the server truly knows the user's encrypted password. This prevents common phishing attacks: a phishing attacker controlling a fake website cannot convince a user that he authenticated to the genuine website. Furthermore, even when a user authenticates to an illegitimate server, the server cannot gain any information about the user's password. The Mutual authentication method is designed as an extension to the HTTP protocol, and is intended to replace the existing authentication methods used in HTTP (the Basic method, Digest method, and authentication using HTML forms).

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 December 6, 2012.

Copyright Notice

Copyright (c) 2012 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	5
1.1. Relations to other technologies	6
1.1.1. Technologies updated or superceded by this proposal	6
1.1.1.1. HTTP Basic and Digest authentication	6
1.1.1.2. HTML Form authentication	6
1.1.2. Technologies not updated by this proposal	7
1.1.2.1. Federated identity/authorization management	7
1.1.2.2. HTTPS and HTTPS client-certificate authentication	8
1.1.2.3. Protocols for local identity-management frameworks	8
1.1.2.4. HTTP and HTTP authentication architecture	8
1.2. Terminology	9
1.3. Document Structure and Related Documents	9
2. Protocol Overview	10
2.1. Messages Overview	10
2.2. Typical Flows of the Protocol	11
2.3. Alternative Flows	14
3. Message Syntax	15
3.1. Values	16
3.1.1. Tokens	16
3.1.2. Strings	17
3.1.3. Numbers	17
4. Messages	18
4.1. 401-INIT and 401-STALE	19
4.2. req-KEX-C1	22
4.3. 401-KEX-S1	22
4.4. req-VFY-C	23
4.5. 200-VFY-S	24
5. Authentication Realms	25
5.1. Resolving Ambiguities	26
6. Session Management	27
7. Validation Methods	29
8. Authentication Extensions	30
9. Decision Procedure for Clients	31
10. Decision Procedure for Servers	36
11. Authentication Algorithms	38
11.1. Support Functions and Notations	39
11.2. Default Functions for Algorithms	40
12. Application Channel Binding	41
13. Application for Proxy Authentication	41
14. Methods to Extend This Protocol	42
15. IANA Considerations	43
16. Security Considerations	43
16.1. Security Properties	43
16.2. Denial-of-service Attacks to Servers	44

16.3. Implementation Considerations	44
16.4. Usage Considerations	45
17. Notice on Intellectual Properties	45
18. References	46
18.1. Normative References	46
18.2. Informative References	47
Appendix A. (Informative) Draft Remarks from Authors	48
Appendix B. (Informative) Draft Change Log	49
B.1. Changes in Revision 12	49
B.2. Changes in Revision 11	49
B.3. Changes in Revision 10	49
B.4. Changes in Revision 09	50
B.5. Changes in Revision 08	50
B.6. Changes in Revision 07	51
B.7. Changes in Revision 06	51
B.8. Changes in Revision 05	51
B.9. Changes in Revision 04	51
B.10. Changes in Revision 03	52
B.11. Changes in Revision 02	52
B.12. Changes in Revision 01	52
Authors' Addresses	52

1. Introduction

This document specifies a mutual authentication method for Hyper-Text Transport Protocol (HTTP). The method, called "Mutual Authentication Protocol" in this document, provides a true mutual authentication between an HTTP client and an HTTP server, using just a simple password as a credential.

The currently available methods for authentication in HTTP and Web systems have several deficiencies. The Basic authentication method [RFC2617] sends a plaintext password to a server without any protection; the Digest method uses a hash function that suffers from simple dictionary-based off-line attacks, and people have begun to think it is obsolete.

The authentication method proposed in this document solves these problems, substitutes for these existing methods, and serves as a long-term solution to Web authentication security. It has the following main characteristics:

- o It provides "true" mutual authentication: in addition to assuring the server that the user knows the password, it also assures the user that the server truly knows the user's encrypted password at the same time. This makes it impossible for fake website owners to persuade users that they have authenticated with the original websites.
- o It uses only passwords as the user's credential: unlike public-key-based security algorithms, the method does not rely on secret keys or other cryptographic data that have to be stored inside the users' computers. The proposed method can be used as a drop-in replacement to the current authentication methods like Basic or Digest, while ensuring a much stronger level of security.
- o It is secure: when the server fails to authenticate with a user, the protocol will not reveal any bit of the user's password.

Users can discriminate between true and fake Web servers using their own passwords by using the proposed method. Even when a user inputs his/her password to a fake website owned by illegitimate phishers, the user will certainly notice that the authentication has failed. Phishers will not be successful in their authentication attempts, even if they forward the received data from a user to a legitimate server or vice versa. Users can input sensitive data to the web forms after confirming that the mutual authentication has succeeded, without fear of phishing attacks.

The document, along with [I-D.oiwa-http-auth-extension], also

proposes several extensions to the current HTTP authentication framework, to replace current widely-used form-based Web authentication. The extensions provided include:

- o Multi-host single authentication within an Internet domain (Section 5),
- o non-mandatory, optional authentication on HTTP (Section 8),
- o log out from both server and client side (Section 8), and
- o finer control for redirection depending on authentication status (Section 8).

1.1. Relations to other technologies

1.1.1. Technologies updated or superseded by this proposal

1.1.1.1. HTTP Basic and Digest authentication

The main purpose of this proposal is obviously providing an upgrade for the two existing HTTP authentication methods, Basic and Digest [RFC2617].

HTTP Basic authentication, as its name suggests, provides very simple authentication mechanism using plain-text password directly upon the HTTP transport. HTTP Digest authentication focuses on mitigating the fundamental weakness of Basic authentication by using MD5-based hashing to the authentication, but that has almost failed to deploy due to improper implementations, interoperability problems, and missing feature implementations before MD5 has deprecated by its cryptographic weakness. Digest also has a fundamental problem that the server-side must possess a password-equivalent to perform authentication, which increases risks of server-side data leakage.

1.1.1.2. HTML Form authentication

Another aim of this protocol is (at least) partially replacing the HTML form authentication. Because of inflexibility of the HTTP Basic authentication, recent Web applications tend to use application-level implementations for user authentication using HTML Forms and Web browser rendering engines. However, that method has many potential security weaknesses as same as the HTTP Basic authentication as it uses plaintext. Considering server-impersonations and existence of human-forging rogue servers (i.e. phishing), script-based implementations of hash-based authentication does not help, because its behavior is completely controlled by the web-page content itself, which is possibly provided by such a rogue server. This also closes

any possibilities for extending HTML forms to implement cryptography, as its user-interface could not be prevented from being imitated using plain-text forms. Using HTTP-level authentication is better in this field, because it is under the control of the client software (Web browsers), which can enforce security checks regardless of server-provided contents.

Of course, we could not ignore the strong reasons of favoring Form authentication over Basic authentication: its flexibility. HTTP authentication framework lacks many features for recent Web applications, mainly for interactions between HTTP-level authentications and application-level management of "authentication sessions". As long as current HTTP-layer (and lower-layer) authentication are used, the new method would share the same problem. To solve this problem, this protocol has a companion mechanism for application-level control of authentication behaviors as a separate draft [I-D.oiwa-http-auth-extension]. By using this additional mechanism, Web applications can implement most of these required features as easy as just calling an already-provided API for them.

1.1.2. Technologies not updated by this proposal

1.1.2.1. Federated identity/authorization management

There are several technologies (protocols, frameworks, or systems) for managing authentications/authorizations involving multiple-parties: some of those examples are OAuth [I-D.ietf-oauth-v2], OpenID Connect [OIDF.Connect.Standard], SAML [OASIS.saml-core-2.0-os] etc. These technologies can be further divided to two categories: federated authentication and authorization delegation, although some of these technologies cover both.

Federated authentication provides so-called "three-legged authentication": provided the result of user authentication to a single entity (identity provider) and the user's consent, the mechanism can provide other entities assertion of the user's identity without performing a separate identity management by every entity. Authorization delegation gives a mechanism for transferring a part of the user's privilege on an entity (resource owners) to another entity without requiring users give away the full credential for the authentication.

Essentially, both of those technologies are transforming a result of conventional, one-by-one (two-legged) authentication into a multi-party privilege management. The purpose of this protocol is to secure the very part of the two-legged authentication, and so it can be naturally combined with existing federated management frameworks for increasing security of the entire system.

Additionally, this protocol can provide a secure peer-to-peer shared key generated during authentication to the higher-layer applications Section 12. These keys can be possibly used by such federating mechanisms in future for simplifying/securing the framework.

1.1.2.2. HTTPS and HTTPS client-certificate authentication

This protocol will not replace the wide-spread and widely-accepted technology of SSL/TLS and HTTPS [RFC2818]. This protocol will be still relying on the HTTPS for the integrity and secrecy of the HTTP payload. This protocol ensures users the integrity and secrecy of the authentication credentials, and authenticity of the talking peer server.

Client certificate (and other public-key-based) authentications have a fair-amount of applications (mainly for high-assurance applications), and there are possible needs for redesigning/updating the whole framework. However, currently public-key-based user-authentication and connection-based user identification is out-of-scope of this proposal.

1.1.2.3. Protocols for local identity-management frameworks

There are several existing frameworks for managing user identity of tightly-managed, closed group of users, such as Kerberos [RFC3961]/GSS-API [RFC2743] etc. Some of these have defined a bridging protocol for HTTP authentication. This protocol does not currently aim to replace such existing frameworks.

More precisely, requirements for those framework and usual Web user authentication differ fundamentally. In such framework, user authentication is performed first, and the result of the authentication tends to be shared in all applications, sometimes even shared regardless of the underlying protocols. In those systems, it is almost never likely to use a multiple identity to be used inside a single server and inside a single client machine at the same time. In such applications, connection-based or even a machine-based authentication can be used without a trouble. This is not a case for the general Web authentication applications.

1.1.2.4. HTTP and HTTP authentication architecture

Although HTTP and generic HTTP authentication architecture lacks some required features (see above), the whole structure of per-request, per-resource authentication is well-suited for general Web applications compared with connection-based or machine-based authentication/authorization framework (those which tie user identity to either connections or machines). The whole protocol in this

specification is designed on top of the framework of [I-D.ietf-httpbis-p7-auth]. Small extensions to the framework in this specification and [I-D.oiwa-http-auth-extension], which are designed for filling the missing features, are carefully designed so that it can be implemented easily only by the client-side without changing the whole framework.

1.2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

The terms "encouraged" and "advised" are used for suggestions that do not constitute "SHOULD"-level requirements. People MAY freely choose not to include the suggested items regarding [RFC2119], but complying with those suggestions would be a best practice; it will improve security, interoperability, and/or operational performance.

This document distinguishes the terms "client" and "user" in the following way: A "client" is an entity understanding and talking HTTP and the specified authentication protocol, usually computer software; a "user" is a (usually natural) person who wants to access data resources using "a client".

The term "natural numbers" refers to the non-negative integers (including zero) throughout this document.

This document treats target (codomain) of hash functions to be natural numbers. The notation OCTETS(H(s)) gives a usual octet-string output of hash function H applied to string s.

1.3. Document Structure and Related Documents

The entire document is organized as follows:

- o Section 2 presents an overview of the protocol design.
- o Sections 3 to 10 define a general framework of the Mutual authentication protocol. This framework is independent of specific cryptographic primitives.
- o Section 11 describes properties needed for cryptographic algorithms used with this protocol framework, and defines a few functions which will be shared among such cryptographic algorithms.

- o The sections after that contain general normative and informative information about the protocol.
- o The appendices contain some information that may help developers to implement the protocol.

In addition, there are two companion documents which are referred from/related to this specification:

- o [I-D.oiiwa-http-mutualauth-algo]: defines a cryptographic primitives which can be used with this protocol framework. [draft note: it is separated so that it may be replaced with another crypto in future. We need at least one example for testing/ implementing this protocol, so here it is.]
- o [I-D.oiiwa-http-auth-extension]: defines a small but useful extensions to the current HTTP authentication framework so that it can support application-level semantics of existing Web systems.

2. Protocol Overview

The protocol, as a whole, is designed as a natural extension to the HTTP protocol [I-D.ietf-httpbis-pl-messaging] using a framework defined in [I-D.ietf-httpbis-p7-auth]. Internally, the server and the client will first perform a cryptographic key exchange, using the secret password as a "tweak" to the exchange. The key-exchange will only succeed when the secrets used by the both peers are correctly related (i.e. generated from the same password). Then, both peers will verify the authentication results by confirming the sharing of the exchanged key. This section describes a brief image of the protocol and the exchanged messages.

2.1. Messages Overview

The authentication protocol uses seven kinds of messages to perform mutual authentication. These messages have specific names within this specification.

- o Authentication request messages: used by the servers to request clients to start mutual authentication.
 - * 401-INIT message: a general message to start the authentication protocol. It is also used as a message indicating an authentication failure.
 - * 200-Optional-INIT message: a variant of the 401-INIT message indicating that an authentication is not mandatory.

- * 401-STALE message: a message indicating that it has to start a new authentication trial.
- o Authenticated key exchange messages: used by both peers to perform authentication and the sharing of a cryptographic secret.
- * req-KEX-C1 message: a message sent from the client.
- * 401-KEX-S1 message: a message sent from the server as a response to a req-KEX-C1 message.
- o Authentication verification messages: used by both peers to verify the authentication results.
- * req-VFY-C message: a message used by the client, requesting that the server authenticates and authorizes the client.
- * 200-VFY-S message: a successful response used by the server, and also asserting that the server is authentic to the client simultaneously.

In addition to the above, either a request or a response without any HTTP headers related to this specification will be hereafter called a "normal request" or a "normal response", respectively.

2.2. Typical Flows of the Protocol

In typical cases, the client access to a resource protected by the Mutual authentication will follow the following protocol sequence.

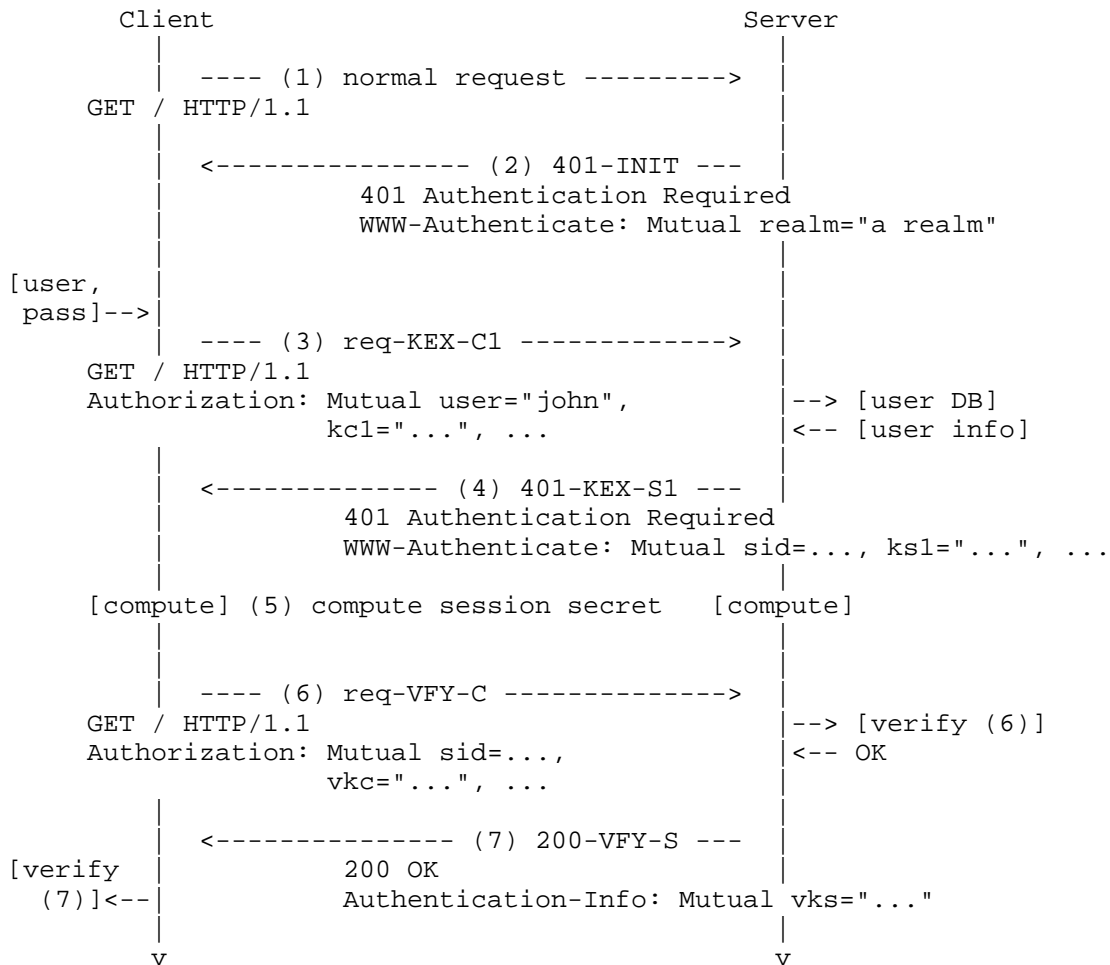


Figure 1: Typical communication flow for first access to resource

- o As usual in general HTTP protocol designs, a client will at first request a resource without any authentication attempt (1). If the requested resource is protected by the Mutual authentication, the server will respond with a message requesting authentication (401-INIT) (2).
- o The client processes the body of the message, and waits for the user to input the user name and a password. If the user name and the password are available, the client will send a message with the authenticated key exchange (req-KEX-C1) to start the authentication (3).

- o If the server has received a req-KEX-C1 message, the server looks up the user's authentication information within its user database. Then the server creates a new session identifier (sid) that will be used to identify sets of the messages that follow it, and responds back with a message containing a server-side authenticated key exchange value (401-KEX-S1) (4).
- o At this point (5), both peers calculate a shared "session secret" using the exchanged values in the key exchange messages. Only when both the server and the client have used secret credentials generated from the same password will the session secret values match. This session secret will be used for the actual access authentication after this point.
- o The client will send a request with a client-side authentication verification value (req-VFY-C) (6), generated from the client-owned session secret. The server will check the validity of the verification value using its own session secret.
- o If the authentication verification value from the client was correct, it means that the client definitely owns the credential based on the expected password (i.e. the client authentication succeeded.) The server will respond with a successful message (200-VFY-S) (7). Contrary to the usual one-way authentication (e.g. HTTP Basic authentication or POP APOP authentication), this message also contains a server-side authentication verification value.

When the client's verification value is incorrect (e.g. because the user-supplied password was incorrect), the server will respond with the 401-INIT message (the same one as used in (2)) instead.

- o The client MUST first check the validity of the server-side authentication verification value contained in the message (7). If the value was equal to the expected one, the server authentication succeeded.

If it is not the value expected, or if the message does not contain the authentication verification value, it means that the mutual authentication has been broken for some unexpected reason. The client MUST NOT process any body or header values contained in this case. (Note: This case should not happen between a correctly-implemented server and a client.)

2.3. Alternative Flows

As shown above, the typical flow for a first authenticated request requires three request-response pairs. To reduce the protocol overhead, the protocol enables several short-cut flows which require fewer messages.

- o (case A) If the client knows that the resource is likely to require the authentication, the client MAY omit the first unauthenticated request (1) and immediately send a key exchange (req-KEX-C1 message). This will reduce one round-trip of messages.
- o (case B) If both the client and the server previously shared a session secret associated with a valid session identifier (sid), the client MAY directly send a req-VFY-C message using the existing session identifier and corresponding session secret. This will further reduce one round-trip of messages.

In such cases, the server MAY have thrown out the corresponding sessions from the session table. In this case, the server will respond with a 401-STALE message, indicating a new key exchange is required. The client SHOULD retry constructing a req-KEX-C1 message in this case.

Figure 2 depicts the shortcut flows described above. Under the appropriate settings and implementations, most of the requests to resources are expected to meet both the criteria, and thus only one round-trip of request/responses will be required in most cases.

(A) omit first request
(2 round trips)

Client	Server
--- req-KEX-C1 ---->	
<---- 401-KEX-S1 ---	
---- req-VFY-C ---->	
<----- 200-VFY-S ---	

(B) reusing session secret

(B-1) key available
(1 round trip)

Client	Server
---- req-VFY-C ---->	
<----- 200-VFY-S ---	

(B-2) key expired
(3 round trips)

Client	Server
--- req-VFY-C ----->	
<----- 401-STALE ---	
--- req-KEX-C1 ----->	
<----- 401-KEX-S1 ---	
--- req-VFY-C ----->	
<----- 200-VFY-S ---	

Figure 2: Several alternative flows on protocol

For more details, see Sections 9 and 10.

3. Message Syntax

Throughout this specification, The syntax is denoted in the extended augmented BNF syntax defined in [I-D.ietf-httpbis-p1-messaging] and [RFC5234]. The following elements are quoted from [RFC5234], [I-D.ietf-httpbis-p1-messaging] and [I-D.ietf-httpbis-p7-auth]: DIGIT, ALPHA, SP, auth-scheme, quoted-string, auth-param, header-field, token, challenge, and credential.

The Mutual authentication protocol uses three headers: WWW-Authenticate (in responses with status code 401), Authorization (in requests), and Authentication-Info (in responses other than 401 status). These headers follow a common framework described in [I-D.ietf-httpbis-p7-auth]. The detailed meanings for these headers are contained in Section 4.

The framework in [I-D.ietf-httpbis-p7-auth] defines the syntax for the headers WWW-Authenticate and Authorization as the syntax elements "challenge" and "credentials", respectively. The "auth-scheme" contained in those headers MUST be "Mutual" throughout this protocol specification. The syntax for "challenge" and "credentials" to be used with the "Mutual" auth-scheme SHALL be name-value pairs (#auth-param), not the "b64token" defined in [I-D.ietf-httpbis-p7-auth].

The Authentication-Info: header used in this protocol SHALL contain the value in same syntax as those the "WWW-Authenticate" header, i.e. the "challenge" syntax element.

In HTTP, the WWW-Authenticate header may contain more than one challenges. Client implementations SHOULD be aware of and be capable of handle those cases correctly.

3.1. Values

The parameter values contained in challenge/credentials MUST be parsed strictly conforming to the HTTP semantics (especially un-quoting of the string parameter values). In this protocol, those values are further categorized into the following value types: tokens (bare-token and extensive-token), string, integer, hex-fixed-number, and base64-fixed-number.

For clarity, implementations are encouraged to use the canonical representations specified in the following subsections for sending values. Recipients SHOULD accept both quoted and unquoted representations interchangeably as specified in HTTP.

3.1.1. Tokens

For sustaining both security and extensibility at the same time, this protocol defines a stricter sub-syntax for the "token" to be used. The extensive-token values SHOULD follow the following syntax (after HTTP value parsing):

```
bare-token      = 1*(DIGIT / ALPHA / "-" / "_")
extension-token = "-" bare-token 1*("." bare-token)
extensive-token = bare-token / extension-token
```

Figure 3: BNF syntax for token values

The tokens (bare-token and extension-token) are case insensitive; Senders SHOULD send these in lower-case, and receivers MUST accept both upper- and lower-cases. When tokens are used as (partial) inputs to any hash or other mathematical functions, it MUST always be used in lower-case.

Extensive-tokens are used in this protocol where the set of acceptable tokens may include non-standard extensions. Any non-standard extensions of this protocol SHOULD use the extension-tokens with format "-<bare-token>.<domain-name>", where <domain-name> is a validly registered (sub-)domain name on the Internet owned by the party who defines the extensions.

Bare-tokens and extensive-tokens are also used for parameter names (of course in the unquoted form). Requirements for using the extension-token for the parameter names are the same as the above.

The canonical format for bare-tokens and tokens are unquoted tokens.

3.1.2. Strings

All character strings outside ASCII character sets MUST be encoded using the UTF-8 encoding [RFC3629] for the ISO 10646-1 character set [ISO.10646-1.1993], without any leading BOM characters. Both peers are RECOMMENDED to reject any invalid UTF-8 sequences that might cause decoding ambiguities (e.g., containing "<" in the second or later byte of the UTF-8 encoded characters).

If strings are representing a domain name or URI that contains non-ASCII characters, the host parts SHOULD be encoded as it is used in the HTTP protocol layer (e.g. in a Host: header); under current standards it will be the one defined in [RFC5890]. It SHOULD use lower-case ASCII characters.

The canonical format for strings are quoted-string.

3.1.3. Numbers

The following syntax definitions gives a syntax for number-type values:

```
integer           = "0" / (%x31-39 *DIGIT)           ; no leading zeros
hex-fixed-number = 1*(2(DIGIT / %x41-46 / %x61-66))
base64-fixed-number = 1*( ALPHA / DIGIT /
                        "-" / "." / "_" / "~" / "+" / "/" ) *"="
```

Figure 4: BNF syntax for number types

The syntax definition of the integers only allows representations that do not contain extra leading zeros.

The numbers represented as a hex-fixed-number MUST include an even number of characters (i.e. multiples of eight bits). Those values are case-insensitive, and SHOULD be sent in lower-case. When these values are generated from any cryptographic values, they SHOULD have their "natural length": if these are generated from a hash function, these lengths SHOULD correspond to the hash size; if these are representing elements of a mathematical set (or group), its lengths SHOULD be the shortest for representing all the elements in the set. For example, any results of SHA-256 hash function will be represented by 64 characters, and any elements in 2048-bit prime field (modulo a 2048-bit integer) will be represented by 512 characters, regardless of how much 0's will be appear in front of such representations. Session-identifiers and other non-cryptographically generated values are represented in any (even) length determined by the side who generates it first, and the same length SHALL be used throughout the all communications by both peers.

The numbers represented as base64-fixed-number SHALL be generated as follows: first, the number is converted to a big-endian radix-256 binary representation as an octet string. The length of the representation is determined in the same way as mentioned above. Then, the string is encoded using the Base 64 encoding [RFC4648] without any spaces and newlines. Implementations decoding base64-fixed-number SHOULD reject any input data with invalid characters, excess/insufficient paddings, or non-canonical pad bits (See Sections 3.1 to 3.5 of [RFC4648]).

The canonical format for integer and hex-fixed-number are unquoted tokens, and that for base64-fixed-number is quoted-string.

4. Messages

In this section we define the seven kinds of messages used in the authentication protocol along with the formats and requirements of the headers for each message.

To determine which message are expected to be sent, see Sections 9 and 10.

In the descriptions below, the type of allowable values for each header parameter is shown in parenthesis after each parameter name. The "algorithm-determined" type means that the acceptable value for

the parameter is one of the types defined in Section 3, and is determined by the value of the "algorithm" parameter. The parameters marked "mandatory" SHALL be contained in the message. The parameters marked "non-mandatory" MAY either be contained or omitted in the message. Each parameter SHALL appear in each headers exactly once at most.

All credentials and challenges MAY contain any parameters not explicitly specified in the following sections. Recipients who do not understand such parameters MUST silently ignore those. However, all credentials and challenges MUST meet the following criteria:

- o For responses, the parameters "reason", any "ks*" (where * stands for any decimal integers), and "vks" are mutually exclusive: any challenge MUST NOT contain two or more parameters among them. They MUST NOT contain any "kc*" and "vkc" parameters.
- o For requests, the parameters "kc*" (where * stands for any decimal integers), and "vks" are mutually exclusive and any challenge MUST NOT contain two or more parameters among them. They MUST NOT contain any "ks*" and "vks" parameters.

4.1. 401-INIT and 401-STALE

Every 401-INIT or 401-STALE message SHALL be a valid HTTP 401-status (Authentication Required) message containing one (and only one: hereafter not explicitly noticed) "WWW-Authenticate" header containing a "reason" parameter in the challenge. The challenge SHALL contain all of the parameters marked "mandatory" below, and MAY contain those marked "non-mandatory".

version: (mandatory extensive-token) should be the token "-draft11" in this specification. The behavior is undefined when other values are specified.

algorithm: (mandatory extensive-token) specifies the authentication algorithm to be used. The value MUST be one of the tokens specified in [I-D.oiwa-http-mutualauth-algo] or other supplemental specification documentation.

validation: (mandatory extensive-token) specifies the method of host validation. The value MUST be one of the tokens described in Section 7, or the tokens specified in other supplemental specification documentation.

auth-domain: (non-mandatory string) specifies the authentication domain, the set of hosts for which the authentication credentials are valid. It MUST be one of the strings described in Section 5. If the value is omitted, it is assumed to be the "single-port" type domain in Section 5.

realm: (mandatory string) is a UTF-8 encoded string representing the name of the authentication realm inside the authentication domain. As specified in [I-D.ietf-httpbis-p7-auth], this value MUST always be sent in the quoted-string form.

pwd-hash: (non-mandatory extensive-token) specifies the hash algorithm (hereafter referred to by ph) used for additionally hashing the password. The valid tokens are

- * none: $ph(p) = p$
- * md5: $ph(p) = MD5(p)$
- * digest-md5: $ph(p) = MD5(\text{username} \mid ":" \mid \text{realm} \mid ":" \mid p)$, the same value as MD5(A1) for "MD5" algorithm in [RFC2617].
- * sha1: $ph(p) = SHA1(p)$

If omitted, the value "none" is assumed. The use of "none" is recommended.

reason: (mandatory extensive-token) SHALL be an extensive-token which describes the possible reason of the failed authentication/authorization. Both servers and clients SHALL understand and support the following three tokens:

- * initial: authentication was not tried because there was no Authorization header in the corresponding request.
- * stale-session: the provided sid; in the request was either unknown to or expired in the server.
- * auth-failed: authentication trial was failed by some reasons, possibly with a bad authentication credentials.

Implementations MAY support the following tokens or any extensive-tokens defined outside this specification. If clients has received any unknown tokens, these SHOULD treat these as if it were "auth-failed" or "initial".

- * reauth-needed: server-side application requires a new authentication trial, regardless of the current status.
- * invalid-parameters: authentication was not even tried in the server-side because some parameters are not acceptable.
- * internal-error: authentication was not even tried in the server-side because there is some troubles on the server-side.
- * user-unknown: a special case of auth-failed, suggesting that the provided user-name is invalid. The use of this parameter is NOT RECOMMENDED for security implications, except for special-purpose applications which makes this value sense.
- * invalid-credential: ditto, suggesting that the provided user-name was valid but authentication was failed. The use of this parameter is NOT RECOMMENDED as the same as the above.
- * authz-failed: authentication was successful, but access to the specified resource is not authorized to the specific authenticated user. (It is different from 403 responses which suggest that the reason of inaccessibility is other than authentication.)

The algorithm specified in this header will determine the types (among those defined in Section 3) and the values for K_cl, K_sl, VK_c and VK_s.

Among these messages, those with the reason parameter of value "stale-session" will be called "401-STALE" messages hereafter, because these have a special meaning in the protocol flow. Messages with any other reason parameters will be called "401-INIT" messages.

4.2. req-KEX-C1

Every req-KEX-C1 message SHALL be a valid HTTP request message containing an "Authorization" header with a credential containing a "kcl" parameter.

The credential SHALL contain the parameters with the following names:

version: (mandatory, extensive-token) should be the token "-draft11" in this specification. The behavior is undefined when other values are specified.

algorithm, validation, auth-domain, realm: MUST be the same value as it is when received from the server.

user: (mandatory, string) is the UTF-8 encoded name of the user. If this name comes from a user input, client software SHOULD prepare the string using SASLprep [RFC4013] before encoding it to UTF-8.

kcl: (mandatory, algorithm-determined) is the client-side key exchange value K_cl, which is specified by the algorithm that is used.

rekey-sid: (non-mandatory, hex-fixed-number): reserved for future extensions (see rekey-method in "200-VFY-S" message).

4.3. 401-KEX-S1

Every 401-KEX-S1 message SHALL be a valid HTTP 401-status (Authentication Required) response message containing a "WWW-Authenticate" header with a challenge containing a "ks1" parameter.

The challenge SHALL contain the parameters with the following names:

version: (mandatory, extensive-token) should be the token "-draft11" in this specification. The behavior is undefined when other values are specified.

algorithm, validation, auth-domain, realm: MUST be the same value as it is when received from the client.

sid: (mandatory, hex-fixed-number) MUST be a session identifier, which is a random integer. The sid SHOULD have uniqueness of at least 80 bits or the square of the maximal estimated transactions concurrently available in the session table, whichever is larger.

See Section 6 for more details.

- ks1:** (mandatory, algorithm-determined) is the server-side key exchange value `Ks1`, which is specified by the algorithm.
- nc-max:** (mandatory, integer) is the maximal value of nonce counts that the server accepts.
- nc-window:** (mandatory, integer) the number of available nonce slots that the server will accept. The value of the `nc-window` parameter is RECOMMENDED to be 32 or more.
- time:** (mandatory, integer) represents the suggested time (in seconds) that the client can reuse the session represented by the `sid`. It is RECOMMENDED to be at least 60. The value of this parameter is not directly linked to the duration that the server keeps track of the session represented by the `sid`.
- path:** (non-mandatory, string) specifies which path in the URI space the same authentication is expected to be applied. The value is a space-separated list of URIs, in the same format as it was specified in domain parameter [RFC2617] for the Digest authentications, and clients are RECOMMENDED to recognize it. The all path elements contained in the parameter MUST be inside the specified `auth-domain`: if not, clients SHOULD ignore such elements.

4.4. req-VFY-C

Every `req-VFY-C` message SHALL be a valid HTTP request message containing an "Authorization" header with a credential containing a "vkc" parameter.

The parameters contained in the header are as follows:

- version:** (mandatory, extensive-token) should be the token `"-draft11"` in this specification. The behavior is undefined when other values are specified.
- algorithm, validation, auth-domain, realm:** MUST be the same value as it is when received from the server for the session.

sid: (mandatory, hex-fixed-number) MUST be one of the sid values that was received from the server for the same authentication realm.

nc: (mandatory, integer) is a nonce value that is unique among the requests sharing the same sid. The values of the nonces SHOULD satisfy the properties outlined in Section 6.

vk: (mandatory, algorithm-determined) is the client-side authentication verification value VK_c, which is specified by the algorithm.

4.5. 200-VFY-S

Every 200-VFY-S message SHALL be a valid HTTP message that is not of the 401 (Authentication Required) status, containing an "Authentication-Info" header with a "vks" parameter.

The parameters contained in the header are as follows:

version: (mandatory, extensive-token) should be the token "-draft11" in this specification. The behavior is undefined when other values are specified.

sid: (mandatory, hex-fixed-number) MUST be the value received from the client.

vks: (mandatory, algorithm-determined) is the server-side authentication verification value VK_s, which is specified by the algorithm.

logout-timeout: (non-mandatory, integer) is the number of seconds after which the client should re-validate the user's password for the current authentication realm. The value 0 means that the client SHOULD automatically forget the user-inputted password for the current authentication realm and revert to the unauthenticated state (i.e. server-initiated logout). This does not, however, mean that the long-term memories for the passwords (such as the password reminders and auto fill-ins) should be removed. If a new timeout value is received for the same authentication realm, it overrides the previous timeout. If logout-timeout parameters are specified both in an Authentication-Info header and an Authentication-Control header ([I-D.oiwa-http-auth-extension]), the client SHOULD

respect the smaller one of those and ignore the other.

rekey-method: (non-mandatory, extensive-token): defining a credential used for reestablishing a new session with a new sid. It must be either omitted or the token "passwords" at the current specification. The bare-tokens "refresh-key" and "refresh-key-global" are reserved for future extensions.

The header MUST be sent before the content body: it MUST NOT be sent in the trailer of a chunked-encoded response. If a "100 Continue" response is sent from the server, the Authentication-Info header SHOULD be included in that response, instead of the final response.

5. Authentication Realms

In this protocol, an "authentication realm" is defined as a set of resources (URIs) for which the same set of user names and passwords is valid for. If the server requests authentication for an authentication realm that the client is already authenticated for, the client will automatically perform the authentication using the already-known secrets. However, for the different authentication realms, the clients SHOULD NOT automatically reuse the usernames and passwords for another realm.

Just like in Basic and Digest access authentication protocols, Mutual authentication protocol supports multiple, separate protection spaces to be set up inside each host. Furthermore, the protocol supports that a single authentication realm spans over several hosts within the same Internet domain.

Each authentication realm is defined and distinguished by the triple of an "authentication algorithm", an "authentication domain", and a "realm" parameter. However, server operators are NOT RECOMMENDED to use the same pair of an authentication domain and a realm for different authentication algorithms.

The realm parameter is a string as defined in Section 4. Authentication domains are described in the remainder of this section.

An authentication domain specifies the range of hosts that the authentication realm spans over. In this protocol, it MUST be one of the following strings.

- o Single-server type: The string in format "`<scheme>://<host>:<port>`", where `<scheme>`, `<host>`, and `<port>` are

the corresponding URI parts of the request URI. Even if the request-URI does not have a port part, the string will include one (i.e. 80 for http and 443 for https). The port part MUST NOT contain leading zeros. Use this when authentication is only valid for specific protocol (such as https).

- o Single-host type: The "host" part of the requested URI. This is the default value. Authentication realms within this kind of authentication domain will span over several protocols (i.e. http and https) and ports, but not over different hosts.
- o Wildcard-domain type: The string in format "*.<domain-postfix>", where <domain-postfix> is either the host part of the requested URI or any domain in which the requested host is included (this means that the specification "*.example.com" is valid for all of hosts "www.example.com", "web.example.com", "www.sales.example.com" and "example.com"). The domain-postfix sent from the servers MUST be equal to or included in a valid Internet domain assigned to a specific organization: if clients know, by some means such as a blacklist for HTTP cookies [RFC6265], that the specified domain is not to be assigned to any specific organization (e.g. "*.com" or "*.jp"), the clients are RECOMMENDED to reject the authentication request.

In the above specifications, every "scheme", "host", and "domain" MUST be in lower-case, and any internationalized domain names beyond the ASCII character set SHALL be represented in the way they are sent in the underlying HTTP protocol, represented in lower-case characters; i.e. these SHALL be in the form of the LDH labels in IDNA [RFC5890]. All "port"s MUST be in the shortest, unsigned, decimal number notation. Not obeying these requirements will cause failure of valid authentication attempts.

5.1. Resolving Ambiguities

In the above definitions of authentication domains, several domains will overlap each other. Depending on the "path" parameters given in the "401-KEX-S1" message (see Section 4), there may be several candidates when the client is going to send a request including an authentication credential (Steps 3 and 4 of the decision procedure presented in Section 9).

If such choices are required, the following procedure SHOULD be followed.

- o If the client has previously sent a request to the same URI, and if it remembers the authentication realm requested by 401-INIT messages at that time, use that realm.

- o In other cases, use one of authentication realms representing the most-specific authentication domains. From the list of possible domain specifications shown above, each one earlier has priority over ones described after that.

If there are several choices with different domain-postfix specifications, the one that has the longest domain-postfix has priority over ones with a shorter domain-postfix.

- o If there are realms with the same authentication domain, there is no defined priority: the client MAY choose any one of the possible choices.

If possible, server operators are encouraged to avoid such ambiguities by properly setting the "path" parameters.

6. Session Management

In the Mutual authentication protocol, a session represented by an sid is set up using first four messages (first request, 401-INIT, req-KEX-C1 and 401-KEX-S1), and a "session secret" (z) associated with the session is established. After sharing a session secret, this session, along with the secret, can be used for one or more requests for resources protected by the same realm in the same server. Note that session management is only an inside detail of the protocol and usually not visible to normal users. If a session expires, the client and server SHOULD automatically reestablish another session without informing the users.

Sessions and session identifiers are local to each server (defined by scheme, host and port) inside an authentication domain; the clients MUST establish separate sessions for each port of a host to be accessed. Furthermore, sessions and identifiers are also local to each authentication realm, even if these are provided from the same servers. The same session identifiers provided either from different servers or for different realms SHOULD be treated as independent ones.

The server SHOULD accept at least one req-VFY-C request for each session, given that the request reaches the server in a time window specified by the timeout parameter in the 401-KEX-S1 message, and that there are no emergent reasons (such as flooding attacks) to forget the sessions. After that, the server MAY discard any session at any time and MAY send 401-STALE messages for any req-VFY-C requests.

The client MAY send two or more requests using a single session

specified by the sid. However, for all such requests, each value of the nonce (in the nc parameter) MUST satisfy the following conditions:

- o It is a natural number.
- o The same nonce was not sent within the same session.
- o It is not larger than the nc-max value that was sent from the server in the session represented by the sid.
- o It is larger than (largest-nc - nc-window), where largest-nc is the maximal value of nc which was previously sent in the session, and nc-window is the value of the nc-window parameter which was received from the server in the session.

The last condition allows servers to reject any nonce values that are "significantly" smaller than the "current" value (defined by the value of nc-window) of the nonce used in the session involved. In other words, servers MAY treat such nonces as "already received". This restriction enables servers to implement duplicated nonce detection in a constant amount of memory (for each session).

Servers MUST check for duplication of the received nonces, and if any duplication is detected, the server MUST discard the session and respond with a 401-STALE message, as outlined in Section 10. The server MAY also reject other invalid nonce values (such as ones above the nc-max limit) by sending a 401-STALE message.

For example, assume the nc-window value of the current session is 32, nc-max is 100, and that the client has already used the following nonce values: {1-20, 22, 24, 30-38, 45-60, 63-72}. Then the nonce values that can be used for next request is one of the following set: {41-44, 61-62, 73-100}. The values {0, 21, 23, 25-29, 39-40} MAY be rejected by the server because they are not above the current "window limit" ($40 = 72 - 32$).

Typically, clients can ensure the above property by using a monotonically-increasing integer counter that counts from zero upto the value of nc-max.

The values of the nonces and any nonce-related values MUST always be treated as natural numbers within an infinite range. Implementations using fixed-width integers or fixed-precision floating numbers MUST correctly and carefully handle integer overflows. Such implementations are RECOMMENDED to accept any larger values that cannot be represented in the fixed-width integer representations, as long as other limits such as internal header-length restrictions are

not involved. The protocol is designed carefully so that both the clients and servers can implement the protocol using only fixed-width integers, by rounding any overflowed values to the maximum possible value.

7. Validation Methods

The "validation method" specifies a method to "relate" the mutual authentication processed by this protocol with other authentications already performed in the underlying layers and to prevent man-in-the-middle attacks. It decides the value *v* that is an input to the authentication protocols.

The valid tokens for the validation parameter and corresponding values of *v* are as follows:

- host: hostname validation: The value *v* will be the ASCII string in the following format:
 "<scheme>://<host>:<port>", where <scheme>, <host>, and <port> are the URI components corresponding to the currently accessing resource. The scheme and host are in lower-case, and the port is in a shortest decimal representation. Even if the request-URI does not have a port part, *v* will include one.
- tls-cert: TLS certificate validation: The value *v* will be the octet string of the hash value of the public key certificate used in the underlying TLS [RFC5246] (or SSL) connection. The hash value is defined as the value of the entire signed certificate (specified as "Certificate" in [RFC5280]), hashed by the hash algorithm specified by the authentication algorithm used.
- tls-key: TLS shared-key validation: The value *v* will be the octet string of the shared master secret negotiated in the underlying TLS (or SSL) connection.

If the HTTP protocol is used on a non-encrypted channel (TCP and SCTP, for example), the validation type MUST be "host". If HTTP/TLS [RFC2818] (HTTPS) protocol is used with the server certificates, the validation type MUST be either "tls-cert" or "tls-key". If HTTP/TLS protocol is used with an anonymous Diffie-Hellman key exchange, the validation type MUST be "tls-key" (see the note below).

If the validation type "tls-cert" is used, the server certificate provided on TLS connection MUST be verified to make sure that the

server actually owns the corresponding secret key.

Clients MUST validate this parameter upon reception of the 401-INIT messages.

However, when the client is a Web browser with any scripting capabilities, the underlying TLS channel used with HTTP/TLS MUST provide server identity verification. This means (1) the anonymous Diffie-Hellman key exchange ciphersuite MUST NOT be used, and (2) the verification of the server certificate provided from the server MUST be performed.

For other systems, when the underlying TLS channel used with HTTP/TLS does not perform server identity verification, the client SHOULD ensure that all the responses are validated using the Mutual authentication protocol, regardless of the existence of the 401-INIT responses.

Note: The protocol defines two variants for validation on the TLS connections. The "tls-key" method is more secure. However, there are some situations where tls-cert is more preferable.

- o When TLS accelerating proxies are used, it is difficult for the authenticating server to acquire the TLS key information that is used between the client and the proxy. This is not the case for client-side "tunneling" proxies using a CONNECT method extension of HTTP.
- o When a black-box implementation of the TLS protocol is used on either peer.

Implementations supporting a Mutual authentication over the HTTPS protocol SHOULD support the "tls-cert" validation. Support for "tls-key" validation is OPTIONAL for both the servers and clients.

8. Authentication Extensions

The HTTP authentication extensions described in [I-D.oiswa-http-auth-extension] is a definitive part of this protocol. Interactive clients (e.g. Web browsers) supporting this protocol are RECOMMENDED to support non-mandatory authentication and the Authentication-Control header defined there, except the "auth-style" parameter. This specification also proposes (however, not mandates) default "auth-style" to be "non-modal". Web applications SHOULD however consider the security impacts of the behaviors of clients that do not support these headers.

Authentication-initializing messages with the Optional-WWW-Authenticate header are used where 401-INIT response is valid. Such a message is called a 200-Optional-INIT message in this document. (It will not replace other 401-type messages such as 401-STALE and 401-KEX-S1.)

9. Decision Procedure for Clients

To securely implement the protocol, the user client must be careful about accepting the authenticated responses from the server. This also holds true for the reception of "normal responses" (responses which do not contain Mutual-related headers) from HTTP servers.

Clients SHOULD implement a decision procedure equivalent to the one shown below. (Unless implementers understand what is required for the security, they should not alter this.) In particular, clients SHOULD NOT accept "normal responses" unless explicitly allowed below. The labels on the steps are for informational purposes only. Action entries within each step are checked in top-to-bottom order, and the first clause satisfied SHOULD be taken.

Step 1 (step_new_request):

If the client software needs to access a new Web resource, check whether the resource is expected to be inside some authentication realm for which the user has already been authenticated by the Mutual authentication scheme. If yes, go to Step 2. Otherwise, go to Step 5.

Step 2:

Check whether there is an available sid for the authentication realm you expect. If there is one, go to Step 3. Otherwise, go to Step 4.

Step 3 (step_send_vfy_1):

Send a req-VFY-C request.

- * If you receive a 401-INIT message with a different authentication realm than expected, go to Step 6.
- * If you receive a 200-Optional-INIT message with a different authentication realm than expected, go to Step 6.
- * If you receive a 401-STALE message, go to Step 9.
- * If you receive a 401-INIT message, go to Step 13.

- * If you receive a 200-VFY-S message, go to Step 14.

- * If you receive a normal response, go to Step 11.

Step 4 (step_send_kex1_1):

Send a req-KEX-C1 request.

- * If you receive a 401-INIT message with a different authentication realm than expected, go to Step 6.

- * If you receive a 200-Optional-INIT message with a different authentication realm than expected, go to Step 6.

- * If you receive a 401-KEX-S1 message, go to Step 10.

- * If you receive a 401-INIT message with the same authentication realm, go to Step 13 (see Note 1).

- * If you receive a normal response, go to Step 11.

Step 5 (step_send_normal_1):

Send a request without any Mutual authentication headers.

- * If you receive a 401-INIT message, go to Step 6.

- * If you receive a 200-Optional-INIT message, go to Step 6.

- * If you receive a normal response, go to Step 11.

Step 6 (step_rcvd_init):

Check whether you know the user's password for the requested authentication realm. If yes, go to Step 7. Otherwise, go to Step 12.

Step 7:

Check whether there is an available sid for the authentication realm you expect. If there is one, go to Step 8. Otherwise, go to Step 9.

Step 8 (step_send_vfy):

Send a req-VFY-C request.

- * If you receive a 401-STALE message, go to Step 9.

- * If you receive a 401-INIT message, go to Step 13.

- * If you receive a 200-VFY-S message, go to Step 14.

Step 9 (step_send_kex1):

Send a req-KEX-C1 request.

- * If you receive a 401-KEX-S1 message, go to Step 10.
- * If you receive a 401-INIT message, go to Step 13 (See Note 1).

Step 10 (step_rcvd_kex1):

Send a req-VFY-C request.

- * If you receive a 401-INIT message, go to Step 13.
- * If you receive a 200-VFY-S message, go to Step 14.

Step 11 (step_rcvd_normal):

The requested resource is out of the authenticated area. The client will be in the "UNAUTHENTICATED" status. If the response contains a request for authentications other than Mutual, it MAY be handled normally.

Step 12 (step_rcvd_init_unknown):

The requested resource requires a Mutual authentication, and the user is not yet authenticated. The client will be in the "AUTH-REQUESTED" status, and is RECOMMENDED to process the content sent from the server, and to ask user for a user name and a password. When those are supplied from the user, proceed to Step 9.

Step 13 (step_rcvd_init_failed):

For some reason the authentication failed: possibly the password or the username is invalid for the authenticated resource. Forget the password for the authentication realm and go to Step 12.

Step 14 (step_rcvd_vfy):

Check the validity of the received VK_s value. If it is equal to the expected value, it means that the mutual authentication has succeeded. The client will be in the "AUTH-SUCCEEDED" status.

If the value is unexpected, it is a fatal communication error.

If a user explicitly requests to log out (via user interfaces), the client MUST forget the user's password, go to step 5 and reload the current resource without an authentication header.

Note 1: These transitions MAY be accepted by clients, but NOT RECOMMENDED for servers to initiate.

Any kind of response (including a normal response) other than those

shown in the above procedure SHOULD be interpreted as a fatal communication error, and in such cases the clients MUST NOT process any data (response body and other content-related headers) sent from the server. However, to handle exceptional error cases, clients MAY accept a message without an Authentication-Info header, if it is a Server-Error (5xx) status. The client will be in the "UNAUTHENTICATED" status in these cases.

The client software SHOULD display the three client status to the end-user. For an interactive client, however, if a request is a sub-request for a resource included in another page (e.g., embedded images, style sheets, frames etc.), its status MAY be omitted from being shown, and any "AUTH-REQUESTED" statuses MAY be treated in the same way as an "UNAUTHENTICATED" status.

Figure 5 shows a diagram of the client-side state.

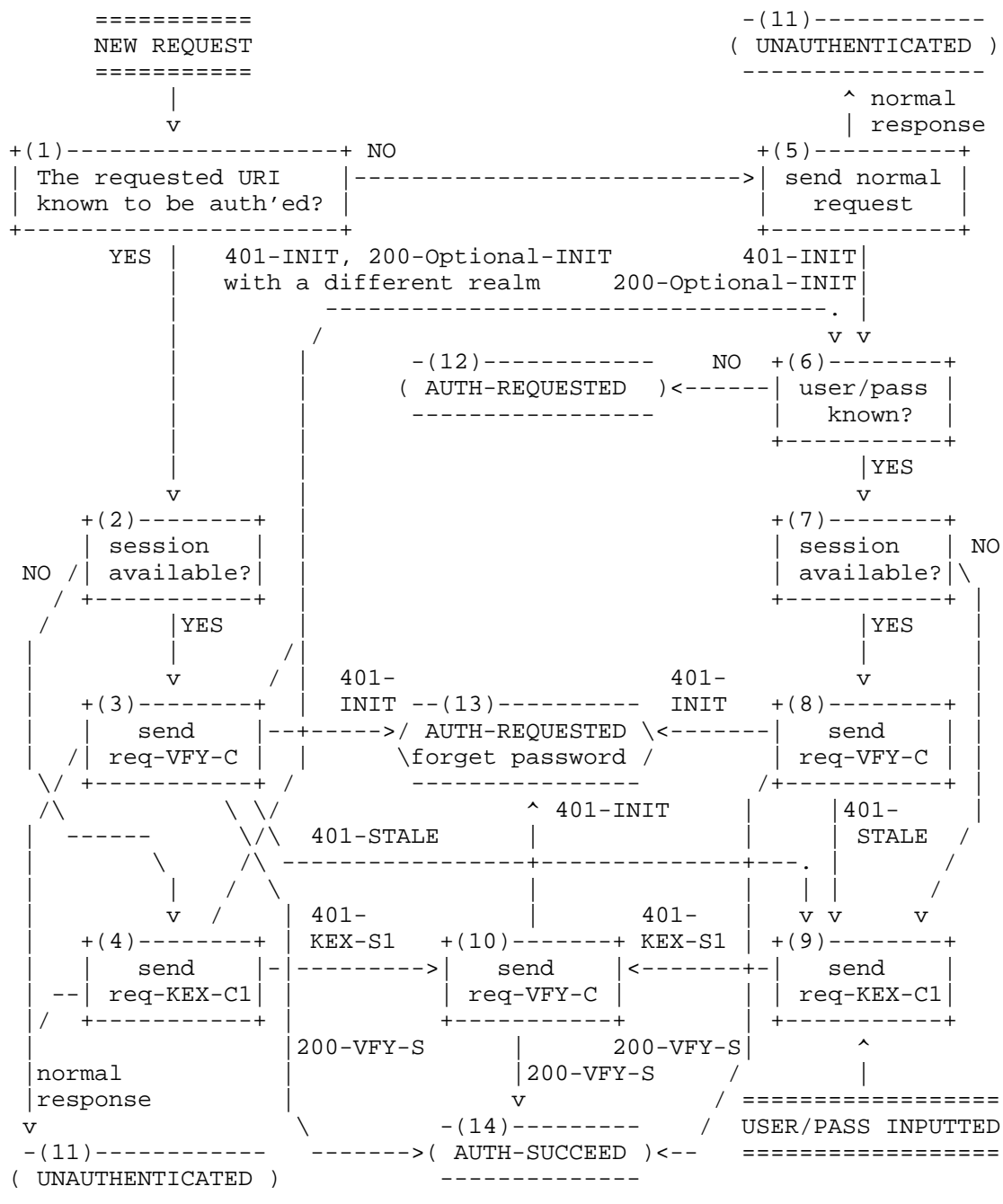


Figure 5: State diagram for clients

10. Decision Procedure for Servers

Each server SHOULD have a table of session states. This table need not be persistent over a long term; it MAY be cleared upon server restart, reboot, or others. Each entry in the table SHOULD contain at least the following information:

- o The session identifier, the value of the sid parameter.
- o The algorithm used.
- o The authentication realm.
- o The state of the protocol: one of "key exchanging", "authenticated", "rejected", or "inactive".
- o The user name received from the client
- o The boolean flag noting whether or not the session is fake.
- o When the state is "key exchanging", the values of K_cl and S_sl.
- o When the state is "authenticated", the following information:
 - * The value of the session secret z
 - * The largest nc received from the client (largest-nc)
 - * For each possible nc values between (largest-nc - nc-window + 1) and max_nc, a flag whether or not a request with the corresponding nc has been received.

The table MAY contain other information.

Servers SHOULD respond to the client requests according to the following procedure:

- o When the server receives a normal request:
 - * If the requested resource is not protected by the Mutual Authentication, send a normal response.
 - * If the resource is protected by the Mutual Authentication, send a 401-INIT response.
 - * If the resource is protected by the optional Mutual Authentication, send a 200-Optional-INIT response.

- o When the server receives a req-KEX-C1 request:
 - * If the requested resource is not protected by the Mutual Authentication, send a normal response.
 - * If the authentication realm specified in the req-KEX-C1 request is not the expected one, send either a 401-INIT or a 200-Optional-INIT response.
 - * If the server cannot validate the parameter kcl, send a 401-INIT response.
 - * If the received user name is either invalid, unknown or unacceptable, create a new session, mark it a "fake" session, compute a random value as K_s1, and send a fake 401-KEX-S1 response. (Note: the server SHOULD NOT send a 401-INIT response in this case, because it will leak the information to the client that the specified user will not be accepted. Instead, postpone it to the response for the next req-VFY-C request.)
 - * Otherwise, create a new session, compute K_s1 and send a 401-KEX-S1 response.

The created session has the "key exchanging" state.

- o When the server receives a req-VFY-C request:
 - * If the requested resource is not protected by the Mutual Authentication, send a normal response.
 - * If the authentication realm specified in the req-VFY-C request is not the expected one, send either a 401-INIT or a 200-Optional-INIT response.

If none of above holds true, the server will lookup the session corresponding to the received sid and the authentication realm.

- * If the session corresponding to the received sid could not be found, or it is in the "inactive" state, send a 401-STALE response.
- * If the session is in the "rejected" state, send either a 401-INIT or a 401-STALE message.
- * If the session is in the "authenticated" state, and the request has an nc value that was previously received from the client, send a 401-STALE message. The session SHOULD be changed to the

"inactive" status.

- * If the nc value in the request is larger than the nc-max parameter sent from the server, or if it is not larger than (largest-nc - nc-window) (when in "authenticated" status), the server MAY (but not REQUIRED to) send a 401-STALE message. The session SHOULD be changed to the "inactive" status if so.
- * If the session is a "fake" session, or if the received vkc is incorrect, then send a 401-INIT response. If the session is in the "key exchanging" state, it SHOULD be changed to the "rejected" state; otherwise, it MAY either be changed to the "rejected" status or kept in the previous state.
- * Otherwise, send a 200-VFY-S response. If the session was in the "key exchanging" state, the session SHOULD be changed to an "authenticated" state. The maximum nc and nc flags of the state SHOULD be updated properly.

At any time, the server MAY change any state entries with both the "rejected" and "authenticated" statuses to the "inactive" status, and MAY discard any "inactive" states from the table. The entries with the "key exchanging" status SHOULD be kept unless there is an emergency situation such as a server reboot or a table capacity overflow.

11. Authentication Algorithms

Cryptographic authentication algorithms which are used with this protocol will be defined separately. The algorithm definition MUST at least provide a definitions for the following functions:

- o The server-side authentication credential J, derived from user-side authentication credential pi.
- o Key exchange values K_cl, K_sl (exchanged on wire) and S_cl, S_sl (kept secret in each peer).
- o Shared secret z, to be computed in both server-side and client side.
- o A hash function H to be used with the protocol.

All algorithm used with this protocol SHOULD provide secure mutual authentication between client and servers, and generate a cryptographically strong shared secret value z, equivalently strong to or stronger than the hash function H. If any passwords (or pass-

phrases or any equivalents, i.e. weak secrets) are involved, these SHOULD NOT be guessable from any data transmitted in the protocol, even if an attacker (either an eavesdropper or an active server) knows the possible thoroughly-searchable candidate list of the passwords. Furthermore, if possible, the function for deriving server-side authentication credential J is RECOMMENDED to be one-way so that pi should not be easily computed from J(pi).

11.1. Support Functions and Notations

In this section we define several support functions and notations to be shared by several algorithm definitions:

The integers in the specification are in decimal, or in hexadecimal when prefixed with "0x".

The function `octet(c)` generates a single octet string whose code value is equal to `c`. The operator `|`, when applied to octet strings, denotes the concatenation of two operands.

The function `VI` encodes natural numbers into octet strings in the following manner: numbers are represented in big-endian radix-128 string, where each digit is represented by a octet within 0x80-0xff except the last digit is represented by a octet within 0x00-0x7f. The first octet MUST NOT be 0x80. For example, `VI(i) = octet(i)` for `i < 128`, and `VI(i) = octet(0x80 + (i >> 7)) | octet(i & 127)` for `128 <= i < 16384`. This encoding is the same as the one used for the subcomponents of object identifiers in the ASN.1 encoding [ITU.X690.1994], and available as a "w" conversion in the `pack` function of several scripting languages.

The function `VS` encodes a variable-length octet string into a uniquely-decoded, self-delimited octet string, as in the following manner:

`VS(s) = VI(length(s)) | s`

where `length(s)` is a number of octets (not characters) in `s`.

Some examples:

`VI(0) = "\000"` (in C string notation)

`VI(100) = "d"`

`VI(10000) = "\316\020"`

VI(1000000) = "\275\204@"

VS("") = "\000"

VS("Tea") = "\003Tea"

VS("Caf<e acute" [in UTF-8]) = "\005Caf\303\251"

VS([10000 "a"s]) = "\316\020aaaaa..." (10002 octets)

[Editorial note: Unlike the colon-separated notion used in the Basic/Digest HTTP authentication scheme, the string generated by a concatenation of the VS-encoded strings will be unique, regardless of the characters included in the strings to be encoded.]

The function OCTETS converts an integer into the corresponding radix-256 big-endian octet string having its natural length: See Section 3.1.3 for the definition of "natural length".

11.2. Default Functions for Algorithms

The functions defined in this section are common default functions among authentication algorithms.

The client-side password-based string pi used by this authentication is derived in the following manner:

pi = H(VS(algorithm) | VS(auth-domain) | VS(realm) | VS(username) | VS(ph(password))).

The values of algorithm, realm, and auth-domain are taken from the values contained in the 401-INIT (or 200-Optional-INIT, hereafter implied) message. When pi is used in the context of an octet string, it SHALL have the natural length derived from the size of the output of function H (e.g. 32 octets for SHA-256). The function ph is determined by the value of the pwd-hash parameter given in a 401-INIT message. If the password comes from a user input, it SHOULD first be prepared using SASLprep [RFC4013]. Then, the password SHALL be encoded as a UTF-8 string before passed to ph.

The values VK_c and VK_s are derived by the following equation.

VK_c = H(octet(4) | OCTETS(K_cl) | OCTETS(K_sl) | OCTETS(z) | VI(nc) | VS(v))

VK_s = H(octet(3) | OCTETS(K_cl) | OCTETS(K_sl) | OCTETS(z) | VI(nc) | VS(v))

Specifications for cryptographic algorithms used with this framework MAY override the functions `pi`, `VK_c`, and `VK_s` defined above. In such cases implementations MUST use the ones defined with such algorithm specifications.

12. Application Channel Binding

Applications and upper-layer communication protocols may need authentication binding to the HTTP-layer authenticated user. Such applications MAY use the following values as a standard shared secret.

These values are parameterized with an optional octet string (`t`) which may be arbitrarily chosen by each applications or protocols. If there is no appropriate value to be specified, use a null string for `t`.

For applications requiring binding to either an authenticated user or a shared-key session (to ensure that the requesting client is certainly authenticated), the following value `b_1` MAY be used.

```
b_1 = OCTETS(H(OCTETS(H(octet(6) | OCTETS(K_cl) | OCTETS(K_sl) |  
OCTETS(z) | VI(0) | VS(v))) | VS(t)))).
```

For applications requiring binding to a specific request (to ensure that the payload data is generated for the exact HTTP request), the following value `b_2` MAY be used.

```
b_2 = OCTETS(H(OCTETS(H(octet(7) | OCTETS(K_cl) | OCTETS(K_sl) |  
OCTETS(z) | VI(nc) | VS(v))) | VS(t)))).
```

Note: Channel bindings to lower-layer transports (TCP and TLS) are defined in Section 7.

13. Application for Proxy Authentication

The authentication scheme defined by the previous sections can be applied m.m. for proxy authentications. In such cases, the following alterations MUST be applied:

- o The 407 status is to be sent and recognized for places where the 401 status is used,
- o Proxy-Authenticate: header is to be used for places where WWW-Authenticate: is used,

- o Proxy-Authorization: header is to be used for places where Authorization: is used,
- o Proxy-Authentication-Info: header is to be used for places where Authentication-Info: is used,
- o The auth-domain parameter is fixed to the host-name of the proxy, which means to cover all requests processed through the specific proxy,
- o The limitation for the paths contained in the path parameter of 401-KEX-S1 messages is disregarded,
- o The omission of the path parameter of 401-KEX-S1 messages means that the authentication realm will potentially cover all requests processed by the proxy,
- o The scheme, host name and the port of the proxy is used for validation tokens, and
- o Authentication extension in [I-D.oiwa-http-auth-extension] is not applicable.

The requirements for client software to display the authentication status to the end-user is also not applicable for proxy authentication. If the client software supports both end-to-end and proxy authentication using this protocol, it SHOULD be careful that the authentication status of the proxy communication will never be confused by users with authentication statuses of the end-to-end resource authentications.

14. Methods to Extend This Protocol

If a private extension to this protocol is implemented, it MUST use the extension-tokens defined in Section 3 to avoid conflicts with this protocol and other extensions. (standardized or being-standardizing extensions MAY use either bare-tokens or extension-tokens.)

Specifications defining authentication algorithms MAY use other representations for the parameters "kc1", "ks1", "vkc", and "vks", replace those parameter names, and/or add parameters to the messages containing those parameters in supplemental specifications, provided that syntactic and semantic requirements in Section 3, [I-D.ietf-httpbis-pl-messaging] and [I-D.ietf-httpbis-p7-auth] are satisfied. Any parameters starting with "kc", "ks", "vkc" or "vks" and followed by decimal natural numbers (e.g. kc2, ks0, vkc1, vks3

etc.) are reserved for this purpose. If those specifications use names other than those mentioned above, it is RECOMMENDED to use extension-tokens to avoid any parameter name conflict with the future extension of this protocol.

Extension-tokens MAY be freely used for any non-standard, private, and/or experimental uses for those parameters provided that the domain part in the token is appropriately used.

15. IANA Considerations

When bare-tokens are used for the authentication-algorithm, pwd-hash, and validation parameters MUST be allocated by IANA. To acquire registered tokens, a specification for the use of such tokens MUST be available as an RFC, as outlined in [RFC5226].

Note: More formal declarations will be added in the future drafts to meet the RFC 5226 requirements.

16. Security Considerations

16.1. Security Properties

- o The protocol is secure against passive eavesdropping and replay attacks. However, the protocol relies on transport security including DNS integrity for data secrecy and integrity. HTTP/TLS SHOULD be used where transport security is not assured and/or data secrecy is important.
- o When used with HTTP/TLS, if TLS server certificates are reliably verified, the protocol provides true protection against active man-in-the-middle attacks.
- o Even if the server certificate is not used or is unreliable, the protocol provides protection against active man-in-the-middle attacks for each HTTP request/response pair. However, in such cases, JavaScript or similar scripting facilities can be used to affect the Mutually-authenticated contents from other contents not protected by this authentication mechanism. This is the reason why this protocol requires that valid TLS server certificates MUST be presented (Section 7).

16.2. Denial-of-service Attacks to Servers

The protocol requires a server-side table of active sessions, which may become a critical point of the server resource consumptions. For proper operation, the protocol requires that at least one key verification request is processed for each session identifier. After that, servers MAY discard sessions internally at any time, without causing any operational problems to clients. Clients will silently reestablishes a new session then.

However, if a malicious client sends too many requests of key exchanges (req-KEX-C1 messages) only, resource starvation might occur. In such critical situations, servers MAY discard any kind of existing sessions regardless of these statuses. One way to mitigate such attacks are that servers MAY have a number and a time limits for unverified pending key exchange requests (in the "wa received" status).

This is a common weakness of authentication protocols with almost any kind of negotiations or states, including Digest authentication method and most Cookie-based authentication implementations. However, regarding the resource consumption, a situation of the mutual authentication method is a slightly better than the Digest, because HTTP requests without any kind of authentication requests will not generate any kind of sessions. Session identifiers are only generated after a client starts a key negotiation. It means that simple clients such as web crawlers will not accidentally consume server-side resources for session managements.

16.3. Implementation Considerations

- o To securely implement the protocol, the Authentication-Info headers in the 200-VFY-S messages MUST always be validated by the client. If the validation fails, the client MUST NOT process any content sent with the message, including other headers and the body part. Non-compliance to this requirement will allow phishing attacks.
- o The authentication status on the client-side SHOULD be visible to the users of the client. In addition, the method for asking for the user's name and passwords SHOULD be carefully designed so that (1) the user can easily distinguish the request from this authentication method from any other authentication methods such as Basic and Digest methods, and (2) the Web contents cannot imitate the user-interfaces for this protocol.

An informational memo regarding user-interface considerations and recommendations for implementing this protocol will be separately

published.

- o For HTTP/TLS communications, when a web form is submitted from Mutually-authenticated pages with the "tls-cert" validation method to a URI that is protected by the same realm (so indicated by the path parameter), if the server certificate has been changed since the pages were received, the peer is RECOMMENDED to be revalidated using a req-KEX-C1 message with an "Expect: 100-continue" header. The same applies when the page is received with the "tls-key" validation method, and when the TLS session has expired.
- o Server-side storages of user passwords are advised to contain the values encrypted by one-way function J(pi), instead of the real passwords, those hashed by ph, or pi.

16.4. Usage Considerations

- o The user-names inputted by a user may be sent automatically to any servers sharing the same auth-domain. This means that when host-type auth-domain is used for authentication on an HTTPS site, and when an HTTP server on the same host requests Mutual authentication within the same realm, the client will send the user-name in a clear text. If user-names have to be kept secret against eavesdropping, the server must use full-scheme-type auth-domain parameter. Contrarily, passwords are not exposed to eavesdroppers even on HTTP requests.
- o The "pwd-hash" parameter is only provided for backward compatibility of password databases. The use of "none" function is the most secure choice and is RECOMMENDED. If values other than "none" are used, you MUST ensure that the hash values of the passwords were not exposed to the public. Note that hashed password databases for plain-text authentications are usually not considered secret.
- o If the server provides several ways for storing server-side password secrets into the password database, it is advised to store the values encrypted by using the one-way function J(pi), instead of the real passwords, those hashed by ph, or pi.

17. Notice on Intellectual Properties

The National Institute of Advanced Industrial Science and Technology (AIST) and Yahoo! Japan, Inc. has jointly submitted a patent application on the protocol proposed in this documentation to the Patent Office of Japan. The patent is intended to be open to any implementors of this protocol and its variants under non-exclusive

royalty-free manner. For the details of the patent application and its status, please contact the author of this document.

The elliptic-curve based authentication algorithms might involve several existing third-party patents. The authors of the document take no position regarding the validity or scope of such patents, and other patents as well.

18. References

18.1. Normative References

- [I-D.ietf-httpbis-pl-messaging]
Fielding, R., Lafon, Y., and J. Reschke, "HTTP/1.1, part 1: URIs, Connections, and Message Parsing", draft-ietf-httpbis-pl-messaging-19 (work in progress), March 2012.
- [I-D.ietf-httpbis-p7-auth]
Fielding, R., Lafon, Y., and J. Reschke, "HTTP/1.1, part 7: Authentication", draft-ietf-httpbis-p7-auth-19 (work in progress), March 2012.
- [I-D.oowa-http-auth-extension]
Oiwa, Y., Watanabe, H., Takagi, H., Kihara, B., Hayashi, T., and Y. Ioku, "HTTP Authentication Extensions for Interactive Clients", draft-oowa-http-auth-extension-02 (work in progress), June 2012.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.
- [RFC4013] Zeilenga, K., "SASLprep: Stringprep Profile for User Names and Passwords", RFC 4013, February 2005.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.
- [RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

18.2. Informative References

- [I-D.ietf-oauth-v2]
Hammer-Lahav, E., Recordon, D., and D. Hardt, "The OAuth 2.0 Authorization Framework", draft-ietf-oauth-v2-26 (work in progress), May 2012.
- [I-D.ietf-precis-framework]
Saint-Andre, P. and M. Blanchet, "PRECIS Framework: Preparation and Comparison of Internationalized Strings in Application Protocols", draft-ietf-precis-framework-03 (work in progress), May 2012.
- [I-D.oiwa-http-mutualauth-algo]
Oiwa, Y., Watanabe, H., Takagi, H., Kihara, B., Hayashi, T., and Y. Ioku, "Mutual Authentication Protocol for HTTP: KAM3-based Cryptographic Algorithms", draft-oiwa-http-mutualauth-algo-02 (work in progress), May 2012.
- [ISO.10646-1.1993]
International Organization for Standardization, "Information Technology - Universal Multiple-octet coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane", ISO Standard 10646-1, May 1993.
- [ITU.X690.1994]
International Telecommunications Union, "Information Technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ITU-T Recommendation X.690, 1994.
- [OASIS.saml-core-2.0-os]
Cantor, S., Kemp, J., Philpott, R., and E. Maler, "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard saml-core-2.0-os, March 2005.
- [OIDF.Connect.Standard]
Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., Mortimore, C., and E. Jay, "OpenID Connect Standard 1.0 - draft 10", May 2012.
- [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.

- [RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, January 2000.
- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.
- [RFC3961] Raeburn, K., "Encryption and Checksum Specifications for Kerberos 5", RFC 3961, February 2005.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.
- [RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, August 2010.
- [RFC5929] Altman, J., Williams, N., and L. Zhu, "Channel Bindings for TLS", RFC 5929, July 2010.
- [RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, April 2011.

Appendix A. (Informative) Draft Remarks from Authors

The following items are currently under consideration for future revisions by the authors.

- o Whether to keep TLS-key validation or not.
- o When keeping tls-key validation, whether to use "TLS channel binding" [RFC5929] for "tls-key" verification (Section 7). Note that existing TLS implementations should be considered to determine this.
- o Adopt [I-D.ietf-precis-framework] for replacing SASLprep reference. Especially, use NFC canonicalization instead of NFKC.
- o Adding test vectors for ensuring implementation correctness.
- o Possibly adding a method for servers to detect availability of Mutual authentication on client-side.

- o Possible support for optional key renewal and cross-site federated authentication.

Appendix B. (Informative) Draft Change Log

B.1. Changes in Revision 12

Note: the token for the header parameter "version" is NOT changed from "-draft11", because the protocol semantics has not been changed in this revision.

- o Added a reason "authz-failed".

B.2. Changes in Revision 11

- o Message syntax definition reverted to pre-07 style as httpbis-p1 and p7 now defines a precise rule for parameter value parsing.
- o Replaced "stale" parameter with more infomative/extensive "reason" parameter in 401-INIT and 401-STALE.
- o Reserved "rekey-sid" and "rekey-method" parameters for future extensions.
- o Added descriptions for replacing/non-replacing existing technologies.

B.3. Changes in Revision 10

- o The authentication extension parts (non-mandatory authentication and authentication controls) are separated to yet another draft.
- o The default auth-domain parameter is changed to the full scheme-host-port syntax, which is consistent with usual HTTP authentication framework behavior.
- o Provision for application channel binding is added.
- o Provision for proxy access authentication is added.
- o Bug fix: syntax specification of sid parameter was wrong: it was inconsistent with the type specified in the main text (the bug introduced in -07 draft).
- o Terminologies for headers are changed to be in harmony with httpbis drafts (e.g. field to parameter).

- o Syntax definitions are changed to use HTTP-extended ABNF syntax, and only the header values are shown for header syntax, in harmony with httpbis drafts.
- o Names of parameters and corresponding mathematical values are now renamed to more informative ones. The following list shows correspondence between the new and the old names.

new name	old name	description
S_cl, S_sl	s_a, s_b	client/server-side secret randoms
K_cl, K_sl	w_a, w_b	client/server-side exchanged key components
kc1, ks1	wa, wb	parameter names for those
VK_c, VK_s	o_a, o_b	client/server-side key verifiers
vk_c, vk_s	oa, ob	parameter names for those
z	z	session secrets

B.4. Changes in Revision 09

- o The (default) cryptographic algorithms are separated to another draft.
- o Names of the messages are changed to more informative ones than before. The following is the correspondence table of those names:

new name	old name	description
401-INIT	401-B0	initial response
401-STALE	401-B0-stale	session key expired
req-KEX-C1	req-A1	client->server key exchange
401-KEX-S1	401-B1	server->client key exchange
req-VFY-C	req-A3	client->server auth. verification
200-VFY-S	200-B4	server->client auth. verification
200-Optional-INIT	200-Optional-B0	initial with non-mandatory authentication

B.5. Changes in Revision 08

- o The English text has been revised.

B.6. Changes in Revision 07

- o Adapt to httpbis HTTP/1.1 drafts:
 - * Changed definition of extensive-token.
 - * LWSP continuation-line (%0D.0A.20) deprecated.
- o To simplify the whole spec, the type of nonce-counter related parameters are change from hex-integer to integer.
- o Algorithm tokens are renamed to include names of hash algorithms.
- o Clarified the session management, added details of server-side protocol decisions.
- o The whole draft was reorganized; introduction and overview has been rewritten.

B.7. Changes in Revision 06

- o Integrated Optional Mutual Authentication to the main part.
- o Clarified the decision procedure for message recognitions.
- o Clarified that a new authentication request for any sub-requests in interactive clients may be silently discarded.
- o Typos and confusing phrases are fixed.
- o Several "future considerations" are added.

B.8. Changes in Revision 05

- o A new parameter called "version" is added for supporting future incompatible changes with a single implementation. In the (first) final specification its value will be changed to 1.
- o A new header "Authentication-Control" is added for precise control of application-level authentication behavior.

B.9. Changes in Revision 04

- o Changed text of patent licenses: the phrase "once the protocol is accepted as an Internet standard" is removed so that the sentence also covers the draft versions of this protocol.

- o The "tls-key" verification is now OPTIONAL.
- o Several description fixes and clarifications.

B.10. Changes in Revision 03

- o Wildcard domain specifications (e.g. "*.example.com") are allowed for auth-domain parameters (Section 4.1).
- o Specification of the "tls-cert" verification is updated (incompatible change).
- o State transitions fixed.
- o Requirements for servers concerning w_a values are clarified.
- o RFC references are updated.

B.11. Changes in Revision 02

- o Auth-realm is extended to allow full-scheme type.
- o A decision diagram for clients and decision procedures for servers are added.
- o 401-B1 and req-A3 messages are changed to contain authentication realm information.
- o Bugs on equations for o_A and o_B are fixed.
- o Detailed equations for the entire algorithm are included.
- o Elliptic-curve algorithms are updated.
- o Several clarifications and other minor updates.

B.12. Changes in Revision 01

- o Several texts are rewritten for clarification.
- o Added several security consideration clauses.

Authors' Addresses

Yutaka Oiwa
National Institute of Advanced Industrial Science and Technology
Research Institute for Secure Systems
Tsukuba Central 2
1-1-1 Umezono
Tsukuba-shi, Ibaraki
JP

Email: mutual-auth-contact-ml@aist.go.jp

Hajime Watanabe
National Institute of Advanced Industrial Science and Technology

Hiromitsu Takagi
National Institute of Advanced Industrial Science and Technology

Boku Kihara
Lepidum Co. Ltd.
#602, Village Sasazuka 3
1-30-3 Sasazuka
Shibuya-ku, Tokyo
JP

Tatsuya Hayashi
Lepidum Co. Ltd.

Yuichi Ioku
Yahoo! Japan, Inc.
Midtown Tower
9-7-1 Akasaka
Minato-ku, Tokyo
JP

