Network Working Group
Internet-Draft
Intended status: Experimental
Expires: November 2, 2013

S. Farrell
Trinity College Dublin
P. Hoffman
VPN Consortium
M. Thomas
Phresheez
may 2013

HTTP Origin-Bound Authentication (HOBA) draft-ietf-httpauth-hoba-00

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 November 2, 2013.

Table of Contents

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 an 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, but that can easily replace HTTP or HTML forms-based password authentication. 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 use 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:

- 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.

- 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.

- 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.

- 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 uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234]

2. The HOBA Authentication Scheme

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.

On receipt of a challenge (and optional realm) from a server, the client marshals an HOBA to-be-signed (TBS) blob that includes the a client generated nonce, the web-origin, the realm, an identifier for the CPK 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 formatting chosen for this TBS blob is chosen so as to make server-side signature verification as simple as possible for a wide-range of current server tooling, e.g. including relatviely simple PHP scripts. [[ref for PHP?]]

[[Note - we do support different forms of key identifier, but currently only implicitly. The idea is that when you register a CPK then you get to say what type of key identifier you're using (if that is needed or even makes sense). When authenticating, the site doesn't need to be told that again since it can find the CPK or not based on the key identifier value. One could argue that the type of key indentifier here would allow a site to support two octet-wise identical key identifier values but with different types. Not sure how useful that is, migth be for those who want keys to be 3rd party

validated.]]

Figure 1 specifies the abnf for the signature input. [[This definition is core to the security of HOBA and should ideally be based on some cryptographic protocol that's been well known and studied for ages. We do plan to go looking for such a protocol, (maybe ideally more than 20 years old;-) but have yet to do that.]]

```
HOBA-TBS = nonce alg origin realm kid challenge
nonce = unreserved
alg = 1*2DIGIT
origin = scheme authority port
realm = unreserved
kid = unreserved
challenge = unreserved
```

Figure 1: To-be-signed data for HOBA

The fields above contain the following:

- nonce: is a random value chosen by the UA and MUST be base64url encoded. UAs MUST be able to use at least 32 bits of randomness in generating a nonce. UAs SHOULD be able to use up to 64 bits of randomness for nonces. [[Not sure what's right here. Also - might be better to add a "nonce-type" as well, so we could have e.g. TLS channel bindings as an option.]]

- alg: specifies the signature algorithm being used encoded as an ASCII charecter as defined in Section 9.3. RSA-SHA256 MUST be supported, RSA-SHA1 MAY be supported.

- origin: is the web origin expressed as the catentation of the scheme, authority and port are from [RFC3986]. These are not base64 encoded as they will be most readily available to the server in plain text.

- realm: is similarly just a string with the syntactic restrictions defined in [I-D.ietf-httpbis-p7-auth]. If no realm is specified for this authentication then this is absent. (A missing field here is no problem since both sides know when it needs to be there.)

- kid: is a key identifier - this MUST be a base64url encoded value that is presented to the server in the HOBA client result (see below).

- challenge: MUST be a base64url encoded challenge value that the server chose to send to the client

The HOBA-TBS string is the input to the client's signing process, but is not itself sent over the network since some fields are already inherent in the HTTP exchange. The challenge however is sent over the network so as to make it simpler for a server to be stateless. (One form of stateless challenge might be a ciphertext that the server decrypts and checks, but that is an implementation detail.) The value that is sent over the network is the HOBA "client result" which we now define.

The HOBA "client result" is a dot-separated string that includes the signature and is sent in the HTTP Authorized header field value using the value syntax defined in Figure 2. The "sig" value is the base64url encoded version of the binary output of the signing process. The kid, challenge and nonce are as defined above and are also base64url encoded. [[Expect more changes here. This is very like JOSE's compact form and maybe ought be an instance of that.]]

```
HOBA-RES = kid "." challenge "." nonce "." sig
sig = unreserved
```

Figure 2: HOBA Client Result value

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.]]

The HOBA scheme is far from new, for example, the basic idea is pretty much identical to the first two messages from "Mechanism R" on page 6 of [MI93] which predates HOBA by 20 years.

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.

- 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.

- The "challenge" attribute MUST be included. The challenge is a string of base64url encoded octets 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.

- 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 HOBA client-result (a string matching the HOBA-RES production in Figure 2 as an auth-param with the name "result" and returns that 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:

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

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

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

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

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

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

- 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:

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

- Join: /site/
  join.php?username=Mike&email=mike@
  example.com&curtime=1234567890.1234&keyalg=RSA&digestalg=SHA1&
  signature=

- 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 as described below. 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 contains an HTML form (x-www-form-encoded) with one mandatory field (pub) and some optional fields that allow the UA to specify the type and value of key and device identifiers that the UA wishes to use. [[The device stuff is just a thought.]]

- pub: is a mandatory field containing the PEM formatted public key of the client. See Appendix C of [RFC6376] for an example of how to generate this key format.

- kidtype: contains the type of key identifier, this is a numeric value intended to contain one of the values from Section 9.4. If this is not present then the mandatory to implement "DANE-hash" option MUST be used.

- kid: contains the key identifier as a base64url encoded string that is of the type indicated in the kidtype. If the kid is a hash of a public key then the correct (base64url encoded) hash value MUST be provided and the server SHOULD check that and refuse the registration if an incorrect value was supplied.

- didtype: specifies a kind of device identifier intended to contain one of the values from Section 9.5, if absent then the "string" form of device identifier MUST be used.

- did: A UTF8 string that specifies the device identifier. This can be used to help a user be confident that authentication has worked, e.g., following authentication some web content might say "You last logged in from device 'did' at time T."

6.2. Associating Additional Keys to an Exiting 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

RSA-SHA256 MUST be supported. RSA-SHA1 MAY be used. RSA modulus lengths of at least 2048 bits SHOULD be used.

[[Maybe we should add ECDSA with P256 for shorter signatures.]]

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

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.

9.3. Algorithm Names

TBD, hopefully re-use and existing registry

"0" means RSA-SHA256

"1" means RSA-SHA1

9.4. Key Identifier Types

"0" means a hashed public key, as done in DANE. [RFC6698]

"1" means a URI, such as a mailto: or acct: URI, but anything conforming to [RFC3986] is ok.i

"2" means an unformatted string, at the user's/UA's whim

9.5. Device Identifier Types

"0" means an unformatted nickname, at the user's/UA's whim

10. Acknowledgements

Thanks to the following for good comments received during the prepartion of this specification: Julian Reschke [[and many more to be added]. All errors and stupidities are of course the editors' fault.

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-22 (work in progress), February 2013.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[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.

[RFC6698] Hoffman, P. and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA", RFC 6698, August 2012.

11.2. Informative References

[MI93] Mitchell and Thomas, "Standardising Authentication Protocols Based on Public-Key Techniques.", Journal of Computer Security 2 (1993): 23-36. , 1993.

[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.

[RFC6376] Crocker, D., Hansen, T., and M. Kucherawy, "DomainKeys Identified Mail (DKIM) Signatures", RFC 6376, September 2011.

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.]]

Appendix B. Examples

TBD, but will add next time.

Appendix C. Changes

[[Note to RFC editor - please delete this section before publication.]]

C.1. WG-00

- First WG draft, replacing draft-farrell-httpbis-hoba-02
- Fleshed out HTTP scheme some more.
- Fleshed out registration form more.

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