Network Working Group
Internet-Draft
Intended status: Experimental
Expires: July 12, 2015

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

January 8, 2015

HTTP Origin-Bound Authentication (HOBA)
draft-ietf-httpauth-hoba-10

Abstract

HTTP Origin-Bound Authentication (HOBA) is a digital signature based design for an HTTP authentication method. The design can also be used in Javascript-based authentication embedded in HTML. HOBA is an alternative to HTTP authentication schemes that require passwords and therefore avoids all problems related to passwords, such as leakage of server-side password databases.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

HTTP Origin-Bound Authentication (HOBA) is an authentication design that can be used as an HTTP authentication scheme [RFC7235] 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. Deployment of HOBA can reduce or eliminate password entries in databases, with potentially significant security benefits.

HOBA is an HTTP authentication mechanism that complies with the framework for such schemes [RFC7235]. 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.

Current username/password authentication methods such as HTTP Basic, HTTP Digest, and web forms have been in use for many years but are susceptible to theft of server-side password databases. Instead of passwords, HOBA uses digital signatures in a challenge-response scheme as it's 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" [RFC6454]) to which it authenticates. 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 session management is identical to username/password session management, with a server-side session management tool or script inserting a session cookie [RFC6265] into the output to the browser. Use of TLS for the HTTP session is still necessary to prevent session cookie hijacking.

HOBA keys are "bare keys", so there is no need for the semantic overhead of X.509 public key certificates, particularly with respect to naming and trust anchors. The client public key ("CPK") structures in HOBA 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 needed for modern HTTP authentication:

- o Servers can bind a CPK with an identifier, such as an account name. Servers using HOBA 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.

- Logout features can be useful for UAs, so HOBA defines a way to close a current HTTP "session."

- Digital signatures can be expensive to compute, so HOBA defines a way for HTTP servers to indicate how long a given challenge value is valid, and a way for UAs to fetch a fresh challenge at any time.

Users are also likely to lose a private key, or the client's memory of which key pair is associated with which origin, such as when a user loses the computer or mobile device in which state is stored. HOBA does not define a mechanism for deleting the association between an existing CPK and an account. Such a mechanism can be implemented at the application layer.

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.

1.1. Interfacing to Applications (Cookies)

HOBA can be used as a drop-in replacement for password-based user authentication schemes used in common web applications. The simplest way is to (re-)direct the UA to a HOBA "Login" URL and for the response to a successful HTTP request containing a HOBA signature to set a session cookie [RFC6265]. Further interactions with the web application will then be secured via the session cookie, as is commonly done today.

While cookies are bearer tokens, and thus weaker than HOBA signatures, they are currently ubiquitously used. If non-bearer token session continuation schemes are developed in future in the IETF or elsewhere, then those can interface to HOBA as easily as with any password based authentication scheme.

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 RFC 2119 [RFC2119].

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

Account: 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 at least one CPK and a web-origin; it will also optionally include an HTTP "realm" as defined in the HTTP authentication specification [RFC7235]. 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.

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

HOBA-http: We use this term when describing something that is specific to HOBA as an HTTP authentication mechanism.

HOBA-js: We use this term when describing something that is unrelated to HOBA-http but is relevant for HOBA as a design pattern that can be implemented in a browser in JavaScript.

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

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

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

1.3. Step-by-step Overview of HOBA-http

Step-by-step, a typical HOBA-http registration and authentication flow might look like this:

1. The client connects to the server and makes a request, and the server's response includes a WWW-Authenticate header field that contains the "HOBA" auth-scheme, along with associated parameters (see Section 3).

2. If the client was not already registered with the web-origin and realm it is trying to access, the "joining" process is invoked (see Section 6.1). This creates a key pair and makes the CPK known to the server so that the server can carry out the account creation processes required.

3. The client uses the challenge from the HOBA auth-scheme parameters, along with other information it knows about the web-origin and realm, to create and sign a HOBA-TBS string (see Section 2).

4. The client creates a HOBA client-result (HOBA-RES), using the signed HOBA-TBS for the "sig" value (see Section 2).

5. The client includes the Authorization header field in its next request, using the "HOBA" auth-scheme and putting the HOBA client-result in an auth-param named "result" (see Section 3).

6. The server authenticates the HOBA client-result (see Section 5.1).

7. Typically, the server's response includes a session cookie that allows the client to indicate its authentication state in future requests (see Section 1.1).

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 a HOBA to-be-signed (TBS) blob that includes a client generated nonce, the web-origin, the realm, an identifier for the CPK and the challenge string; and signs that blob with 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.

Figure 1 specifies the ABNF for the signature input. The term "unreserved" means that the field does not have a specific format defined and allows the characters specified in Section 2.3 of [RFC3986].

```
      HOBA-TBS = len ":" nonce
              len ":" alg
              len ":" origin
              len ":" [ realm  ]
              len ":" kid
              len ":" challenge
      len = 1*DIGIT
      nonce = 1*base64urlchars
```

```
      alg = 1*2DIGIT
      origin = scheme "://" authority ":" port
      ; scheme etc are from RFC 3986
      realm = unreserved
      ; realm is to be treated as in Section 2.2 of RFC 7235
      kid = 1*base64urlchars
      challenge = 1*base64urlchars
      ; Characters for Base64URL encoding from Table 2 of RFC 4648
      ; all of which are US-ASCII (see RFC 20)
      base64urlchars = %x30-39             ; Digits
                     / %x41-5A           ; Uppercase letters
                     / %x61-7A           ; Lowercase letters
                     / "-" / "_" / "="   ; Special characters
```

Figure 1: To-be-signed data for HOBA

The fields above contain the following:

- len: Each field is preceeded by the number of octets of the following field, expressed as a decimal number in ASCII [RFC0020]. Lengths are separated from field values by a colon character. So if a nonce with the value "ABCD" were used then that would be preceeded by "4:" (see the example in Appendix B for detail).

- nonce: is a random value chosen by the UA and MUST be base64url encoded before being included in the HOBA-TBS value. (base64url encoding is defined in [RFC4648], guidelines for randomness are give in [RFC4086].) UAs MUST be able to use at least 32 bits of randomness in generating a nonce. UAs SHOULD be able to use 64 or more bits of randomness for nonces.

- alg: specifies the signature algorithm being used. See Section 7 for details of algorithm support requirements.

- The IANA registered algorithm values (see Section 9.3) are encoded as one- or two-digit ASCII numbers. For example, RSA-SHA256 (number 0) is encoded as the ASCII character "0" (0x30), while a future algorithm registered as number 17 would be encoded as the ASCII characters "17" (0x3137).

- origin: is the web origin expressed as the concatenation of the scheme, authority and port from [RFC3986]. These are not base64 encoded as they will be most readily available to the server in plain text. For example, if accessing the URL "https:// www.example.com:8080/foo" then the bytes input to the signature process will be "https://www.example.com:8080". There is no default for the port number, and the port number MUST be present.

- realm: is similarly just a string with the syntactic restrictions defined in [RFC7235]. If no realm is specified for this authentication then this is absent, but is preceeded by a length of zero ("0:"). Recall both sides know when this needs to be there, independent of the encoding via a zero length.

- 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 challenge MUST be chosen so that it is infeasible to guess, and SHOULD be indistinguishable from (the base64url encoding of) an at least 128-bit long random string.

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 reduce the amount of state that needs to be maintained by servers. (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 by the UA 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 Authorizaion 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.

```
HOBA-RES = kid "." challenge "." nonce "." sig
sig = 1*base64urlchars
```

Figure 2: HOBA Client Result value

If a malformed message of any kind is received by a server, the server MUST fail authenticaiton. If a malformed message of any kind is received by a client, the client MUST abandon that authentication attempt. (The client is of course free to start another authentication attempt if it desires.)

3. Introduction to the HOBA-http 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. Note that the HOBA auth-scheme might not be the only one that the server includes in a WWW-Authenticate header.

The HOBA scheme has two REQUIRED attributes (challenge and max-age) and one OPTIONAL attribute (realm):

o The "challenge" attribute MUST be included. The challenge is the string made up of the base64url encoded octets that the server wants the client to sign in its response. The challenge MUST be unique for every HTTP 401 response in order to prevent replay attacks from passive observers.

o A "max-age" attribute MUST be included that specifies the number of seconds from the time the HTTP response is emitted for which responses to this challenge can be accepted for example "max-age: 10" would indicate ten seconds. If max-age is set to zero, then that means that only one signature will be accepted for this challenge.

o A "realm" attribute MAY be included to indicate the scope of protection in the manner described in HTTP/1.1, Part 7 [RFC7235]. The "realm" attribute MUST NOT appear more than once.

When the "client response" is created, the UA encodes the HOBA client-result and returns that in the Authorization header. The client-result is a string matching the HOBA-RES production in Figure 2 as an auth-param with the name "result".

The server MUST check the cryptographic correctness of the signature based on a public key it knows for the kid in the signatures, and if the server cannot do that, or if the signature fails cryptographic checks, then validation has failed. The server can use any additional mechanisms to validate the signature. If the validation fails, or if the server chooses reject the signature for any reason whatsoever, the server fails the request with a 401 Unauthorized HTTP response.

The server MUST check that the same web origin is used in all of the server's TLS server certificates, the URL being accessed and the HOBA signature. If any of those checks fail, the server treats the signature as being cryptographically incorrect.

Note that a HOBA signature is good for however long a non-zero max-age parameter allows. This means that replay is possible within the time window specified by the "max-age" value chosen by the server.

Servers can attempt to detect any such replay (via caching if they so choose) and MAY react to such replays by responding with a second (or subsequent) 401-status HTTP response containing a new challenge.

To optimise their use of challenges, UAs MAY pre-fetch a challenge value, for example after (max-age)/2 seconds have elapsed, using the ".well-known/hoba/getchal" scheme described later in this document. This also allows for pre-calculation of HOBA signatures, if that is required in order to produce a responsive user interface.

4. Introduction to the HOBA-js Mechanism

Web sites using JavaScript can also perform origin-bound authentication without needing to involve the HTTP layer, and by inference not needing HOBA-http support in browsers. HOBA-js is not an on-the-wire protocol like HOBA-http is: instead, it is a design pattern that can be realized completely in JavaScript served in normal HTML pages.

One thing that is highly desirable for HOBA-js is WebCrypto (see http://www.w3.org/TR/WebCryptoAPI) which is (at the time of writing) starting to see deployment. In lieu of WebCrypto, JavaScript crypto libraries can be employed with the known deficiencies of their pseudo-random number generators and the general immaturity of those libraries.

Without Webcrypto, one element is required for HOBA-js: localStorage (see http://www.w3.org/TR/webstorage/) from HTML5 can be used for persistent key storage. For example, an implementation would store a dictionary account identifier, public key and private key tuples in the origin's localStorage for subsequent authentication requests. How this information is actually stored in localStorage is an implementation detail. This type of key storage relies on the security properties of the same-origin policy that localStorage enforces. See the security considerations for discussion about attacks on localStorage. Note that IndexedDB (See http://www.w3.org/TR/IndexedDB/) is an alternative to localStorage that can also be used here and that is used by WebCrypto.

Because of JavaScript's same-origin policy, scripts from subdomains do not have access to the same localStorage that scripts in their parent domains do. For larger or more complex sites, this could be an issue that requires enrollment into subdomains, which could be difficult for users. One way to get around this is to use session cookies because they can be used across subdomains. That is, with HOBA-js, the user might log in using a single well-known domain, and then session cookies are used whilst the user navigates around the site.

5. HOBA's Authentication Process

This section describes how clients and servers use HOBA for authentication. 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. This section also covers the actions that give HOBA user features similar to today's password based schemes.

5.1. CPK Preparation Phase

In the CPK preparation phase, the client determines if it already has a CPK for the web-origin with which it needs to authenticate. If the client 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.

5.2. 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.

5.3. Authentication Phase

The authentication phase is completely dependent on the policies and practices of the server. That is, this phase involves no standardized protocol in HOBA-http; in HOBA-js, there is no suggested interaction template.

In the authentication phase, the server uses the key identifier (kid) to determine 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 this stage of the process involves additional information for authentication, such as asking the user which account she wants to use (in the case where a UA is used for multiple accounts on a site), the server can prompt the user for account identifying information or the user could choose based on HTML offered by the server before the 401 response is triggered. 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.

During the authentication phase, if the server cannot determine the correct CPK, it could use HTML and JavaScript to ask the user if they are really a new user or want to associate this new CPK with another 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. Thus, logging in on a new user agent is identical to logging in with an existing account.

If the server does not recognize the CPK the server might send the client through a either a join or login-new-UA (see below) process. This process is completely up to the server, and probably entails using HTML and JavaScript 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 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 UA even for unprotected resources and set 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.

6. Other Parts of the HOBA Process

The authentication process is more than just the act of authentication. In password-based authentication and HOBA, there are other processes that are needed both before and after an authentication step. This section covers those processes. Where possible, it combines practices of HOBA-http and HOBA-js; where that is not possible, the differences are called out.

All HOBA interactions other than those defined in Section 5 MUST be performed in TLS-protected sessions ([RFC5246]). If the current HTTP traffic is not running under TLS, a new session is started before any of the actions described here are performed.

HOBA-http 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-http clients have to be able to handle redirection of these requests. However, as that would potentially cause security issues when a re-direct brings the client to a different web origin, servers

implementing HOBA-http SHOULD NOT re-direct to a different web origin from below .well-known/hoba URLs. The above is considered sufficient to allow experimentation with HOBA, but if at some point HOBA is placed on the standards track then a full analysis of off-origin re-directions would need to be documented.

6.1. Registration

Normally, a registration (also called "joining") is expected to happen after a UA receives a 401 response for a web-origin and realm (for HOBA-http) or on demand (for HOBA-js) for which it has no associated CPK. 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.

Note that 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 to use the same or no realm.

The registration message for HOBA-http is sent as a POST message to the URL ".well-known/hoba/register" with an HTML form (x-www-form-encoded) (see http://www.w3.org/TR/2014/REC-html5-20141028/forms.html #url-encoded-form-data) described below; The registration message for HOBA-js can be in any format specified by the server, but it could be the same as the one described here for HOBA-http. It is up to the server to decide what kind of user interaction is required before the account is finally set up. When the server's chosen registration flow is completed successfully the server MUST add a Hobareg HTTP header (see Section 6.1.1) to the HTTP response message that completes the registration flow.

The registration message sent to server has 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.

o 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.

o 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 hashed public key option MUST be used.

o 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 defined in Section 9.5 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."

Note that replay of registration (and other HOBA) messages is quite possible. That however can be counteracted if challenge freshness is ensured. See Section 2 for details. Note also that with HOBA-http the HOBA signature does not cover the POST message body. If that is required then HOBA-JS may be a better fit for registration and other account management actions.

6.1.1. Hobareg Definition

Since registration can often be a multi-step process, e.g. requiring a user to fill in contact details, the initial response to the HTTP POST message defined above may not be the end of the registration process even though the HTTP response has a 200 OK status. This creates an issue for the UA since, during the registration process (e.g., while dealing with interstitial pages), the UA doesn't yet know whether the CPK is good for that web origin or not.

For this reason the server MUST add a header field to the response message when the registration has succeded to indicate the new state. The header to be used is "Hobareg" and the value when registration has succeeded is to be "regok". When registration is in an intermediate state (e.g. on an HTTP response for an interstitial page) the server MAY add this header with a value of "reginwork". See Section 9.6 for the relevant IANA registration of this header field.

For interstitial pages, the client MAY include a HOBA Authorization header. This is not considered a MUST as that might needlessly complicate client implementations but is noted here in case a server implementer assumes that all registration messages contain a HOBA Authorization header.

```
      Hobareg-val = "regok" / "reginwork"
```

Figure 3: Hobareg Header Field Definition

Figure 3 provides an ABNF definition for the values allowed in the Hobareg header field. Note that these (and the header field name) are case insensitive. Section 8.3.1 of [RFC7231] calls for documenting the following details for this new header field:

- Only one single value is allowed in a Hobareg header field. Should more than one (a list) be encountered or any other ANBF-invalid value, that SHOULD be interpreted as being the same as "reginwork."

- The Hobareg header field can only be used in HTTP responses.

- Since Hobareg is only meant for responses it ought not appear in requests.

- The HTTP response code does affect the interpretation of Hobareg. Registration is only considered to have succeeded if the regok value is seen in a 2xx response. 4xx and other errors means that registration has failed regardless of the value of Hobareg seen. The request method has no influence on the interpretation of Hobareg.

- Intermediaries never insert, delete or modify a Hobareg header field.

- As a response-only header field, it is not appropriate to list a Hobareg in a Vary response header field.

- Hobareg is allowed in trailers.

- As a response-only header field, Hobareg will not be preserved across re-directs.

- Hobareg itself discloses little security or privacy sensitive information. If an attacker can somehow detect that a Hobareg header field is being added, then that attacker would know that the UA is in the process of registration which could be significant. However, it is likely that the set of messages between the UA and server would expose this information in many cases, regardless of whether or not TLS is used. Using TLS is still however a good plan.

6.2. Associating Additional Keys to an Existing Account

From the user perspective, the UA having a CPK for a web origin will often appear to be the same as having a way to sign in to an account at that web site. Since users often have more than one UA, and since the CPKs are, in general, UA-specific, that raises the question of how the user can sign in to that account from different UAs. And from the server perspective that turns into the question of how to safely bind different CPKs to one account. In this section, we describe some ways in which this can be done, as well as one way in which this ought not be done.

Note that the context here is usually that the user has succeeded in registering with one or more UAs (for the purposes of this section we call this "the first UA" below) and can use HOBA with those, and the user is now adding another UA. The newest UA might or might not have a CPK for the site in question. Since it is in fact trivial, we assume that the site is able to put in place some appropriate quicker, easier registration for a CPK for the newest UA. The issue then becomes one of binding the CPK from the newest UA with those of other UAs bound to the account.

6.2.1. Moving private keys

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. If this approach is taken, then there is no impact on the HOBA-http or HOBA-js so this is a pure UA implementation issue and not discussed further.

6.2.2. Human memorable one time password (don't do this one)

It will be tempting for implementers to use a human-memorable one-time password (OTP) in order to "authenticate" binding CPKs to the same account. The workflow here would likely be something along the lines of some server administrative utility generating a human memorable OTP such as "1234" and sending that to the user out of band for the user to enter at two web pages each authenticated via the relevant CPK. While this seems obvious enough and could even be secure enough in some limited cases, we consider that this is too risky to use in the Internet and so servers SHOULD NOT provide such a mechanism. The reason this is so dangerous is that it would be trivial for an automated client to guess such tokens and "steal" the binding intended for some other user. At any scale, there would always be some in-process bindings so that even with only a trickle

of guesses (and hence not being detectable via message volume) an attacker would have a high probability of succeeding in registering a binding with the attacker's CPK.

This method of binding CPKs together is therefore NOT RECOMMENDED.

6.2.3. Out of band URL

One easy binding method is to simply provide a web page where, using the first UA, the user can generate a URL (containing some "unguessable" cryptographically generated value) that the user then later de-references on the newest UA. The user could e-mail that URL to herself for example, of the web server accessed at the first UA could automatically do that.

Such a URL SHOULD contain at least the equivalent of 128 bits of randomness.

6.3. Logging Out

The user can tell the server it wishes to log out. With HOBA-http, this is done by sending a HOBA-authenticated POST message to the URL ".well-known/hoba/logout" on the site in question. The UA SHOULD also delete session cookies associated with the session so that the user's state is no longer "logged in."

The server MUST NOT allow TLS session resumption for any logged out session.

The server SHOULD also revoke or delete any cookies associated with the session.

6.4. Getting a Fresh Challenge

The UA can get a "fresh" challenge from the server. In HOBA-http, it sends a POST message to ".well-known/hoba/getchal". If successful, the response MUST contain a fresh (base64url encoded) HOBA challenge for this origin in the body of the response. Whitespace in the response MUST be ignored.

7. Mandatory-to-Implement Algorithms

RSA-SHA256 MUST be supported. HOBA implementations MUST use RSA-SHA256 if it is provided by the underlying cryptographic libraries. RSA-SHA1 MAY be used. RSA modulus lengths of at least 2048 bits SHOULD be used. RSA indicates the RSASSA-PKCS1-v1_5 algorithm defined in Section 8.2 of [RFC3447], and SHA-1 and SHA-256 are defined in [SHS]. Keys with moduli shorter than 2048 bits SHOULD

only be used in cases where generating 2048-bit (or longer) keys is impractical, e.g. on very constrained or old devices.

8. Security Considerations

Binding my CPK with someone else's account would be fun and profitable so SHOULD be appropriately hard. In particular URLs or other values generated by the server as part of any CPK binding process 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.

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.

When the max-age parameter is not zero, then a HOBA signature has a property that is like a bearer token for the relevant number of seconds: it can be replayed for a server-selected duration. Similarly, for HOBA-js, signatures might be replayable depending on the specific implementation. The security considerations of [RFC6750] therefore apply in any case where the HOBA signature can be replayed. Server administrators can set the max-age to the minimum acceptable value in such cases, which would often be expected to be just a few seconds. There seems to be no reason to ever set the max-age more than a few minutes; the value ought also decrease over time as device capabilities improve. The administrator will most likely want to set the max-age to something that is not too short for the slowest signing device that is significant for that site.

8.1. Privacy considerations

HOBA does impact to some extent on privacy and could be considered to represent a super-cookie to the server, or to any entity on the path from UA to HTTP server that can see the HOBA signature. This is because we need to send a key identifier as part of the signature and that will not vary for a given key. For this reason, and others, it is strongly RECOMMENDED to only use HOBA over server-authenticated TLS and to migrate web sites using HOBA to only use "https" URLs.

UAs SHOULD provide users a way to manage their CPKs. Ideally, there would be a way for a user to maintain their HOBA details for a site while at the same time deleting other site information such as cookies or non-HOBA HTML5 LocalStorage. However, as this is likely to be complex and appropriate user interfaces counter intutitive, we expect that UAs that implement HOBA will likely treat HOBA information as just some more site data, that would disappear should the user choose to "forget" that site.

Device identifiers are intended to specify classes of device in a way that can assist with registration and with presentation to the user of information about previous sessions, e.g. last login time. Device identifier types MUST NOT be privacy sensitive, with values that would allow tracking a user in unexpected ways. In particular, using an device identifier type that is analogous to the International Mobile Equipment Identifier (IMEI) would be a really bad idea and is the reason for the MUST NOT above. In that case "mobile phone" could be an acceptable choice.

If possible, implementations ought encourage use of device identifier values that are not personally identifying except for the user concerned, for example "Alice's mobile" is likely to be chosen and is somewhat identifying but "Alice's phone: UUID 1234-5567-89abc-def0" would be a very bad choice.

8.2. localStorage Security for Javascript

The use of localStorage (likely with a non-WebCrypto implementation of HOBA-js) 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 UAs 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 [bland] 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/UA/user.

8.3. 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 a server is potentially enrolling the device for a long period and the user may not want to have to be responsible for the credential for that long. To alleviate this problem, the user could request that the credential be erased from the browser. Similarly, during 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 beyond the current browser session. However, all such features really ought be part of the operating system or platform and not part of a HOBA implementation so those are not discussed further.

8.4. Injective Mapping for HOBA-TBS

The repeated length fields in the HOBA-TBS structure are present in order to ensure that there is no possibility that the catenation of different input values can cause confusion that might lead to an attack, either against HOBA as specified here, or else an attack against some other protocol that re-used this to-be-signed structure.

Those fields ensure that the mapping from input fields to the HOBA-TBS string is an injective mapping.

9. IANA Considerations

IANA is requested to make registrations and create new registries as described below.

For all new registries requested by this document, please place those beneath a new "HTTP Origin-Bound Authentication (HOBA) Parameters" category.

9.1. HOBA Authentication Scheme

Please register a new scheme in the HTTP Authentication Scheme Registry registry as follows:

Authentication Scheme Name: HOBA

Pointer to specification text: Section 3 of [[this document]]

Notes (optional): The HOBA scheme can be used with either HTTP servers or proxies. When used in response to a 407 Proxy Authentication Required indication, the appropriate proxy authentication header fields are used instead, as with any other HTTP authentication scheme.

9.2. .well-known URI

Please register a new .well-known URI in the Well-Known URIs registry as described below.

URI suffix: hoba

Change controller: IETF

Specification document(s): Section 6 of [[this document]]

Related information: N/A

9.3. Algorithm Names

Please create a new HOBA signature algorithms registry as follows, with the specification required rule for updates. New HOBA signature algorithms SHOULD be in use with other IETF standards track protocols before being added to this registry.

Number	Meaning
0	RSA-SHA256
1	RSA-SHA1

RSA is defined in Section 8.2 of [RFC3447], and SHA-1 and SHA-256 are defined in [SHS].

For this registry the number column should contain a small positive integer. Following the ABNF above, the maximum value for this is decimal 99.

9.4. Key Identifier Types

Please create a new HOBA Key Identifier Types registry as follows, with the specification required rule for updates.

Number	Meaning
0	a hashed public key [RFC6698]
1	a URI [RFC3986]
2	an unformatted string, at the user's/UA's whim

For the number 0, hashed public keys are as done in DANE. [RFC6698]

For this registry the number column should contain a small positive integer.

9.5. Device Identifier Types

Please create a new HOBA Device Identifier Types registry as follows, with the specification required rule for updates.

The designated expert for this registry is to carefully pay attention to the notes on this field in Section 8.1, in particular the "MUST NOT" stated therein.

Number	Meaning
0	an unformatted UTF8 string, at the user's/UA's whim

For this registry the number column should contain a small positive integer.

9.6. Hobareg HTTP Header Field

Please register a new identifier in the Permanent Message Header Field Names registry as described below.

Header field name: Hobareg

Applicable protocol: HTTP (RFC 7230)

Status: Experimental

Author/Change controller: IETF

Specification document(s): Section 6.1.1 of [[this document]]

Related information: N/A

10. Implementation Status

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

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC6982]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs. Please note that the listing of any individual implementation here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations may exist.

According to [RFC6982] "this will allow reviewers and working groups to assign due consideration to documents that have the benefit of running code, by considering the running code as evidence of valuable experimentation and feedback that has made the implemented protocols more mature. It is up to the individual working groups to use this information as they see fit".

At the time of writing there are three known implementations.

One done by Stephen Farrell of HOBA-http and a HOBA-JS variant implements the current version of HOBA and is available from https://hoba.ie/ which site also includes a demonstration of HOBA.

There is another implementation by Michael Thomas of a HOBA-JS variant.

The most recent (Dec 2014) implementation is by Portugal Telecom and is available from https://github.com/razevedo/hoba-authentication

11. Acknowledgements

Thanks to the following for good comments received during the preparation of this specification: Richard Barnes, David Black, Alissa Cooper, Donald Eastlake, Amos Jeffries, Benjamin Kaduk, Watson Ladd, Barry Leiba, Matt Lepinski, Ilari Liusvaara, James Manger, Alexey Melnikov, Kathleen Moriarty, Yoav Nir, Mark Nottingham, Julian Reschke, Pete Resnick, Michael Richardson, Yaron Sheffer, and Michael Sweet. All errors and stupidities are of course the editors' fault.

12. References

12.1. Normative References

[RFC0020] Cerf, V., "ASCII format for network interchange", RFC 20, October 1969.

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

[RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.

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

[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.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, October 2012.

[RFC7231] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, June 2014.

[RFC7235] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, June 2014.

[SHS] NIST, , "Secure Hash Standard (SHS), FIPS PUB 180-4", NIST Special Publications , March 2012.

12.2. Informative References

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

[RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, April 2011.

[RFC6376] Crocker, D., Hansen, T., and M. Kucherawy, "DomainKeys Identified Mail (DKIM) Signatures", STD 76, RFC 6376, September 2011.

[RFC6982] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", RFC 6982, July 2013.

[bland] Sophos, , "Security Threat Report 2013", January 2013, <http://www.sophos.com/en-us/medialibrary/pdfs/other/sophossecuritythreatreport2013.pdf>.

[bonneau] Bonneau, , "The science of guessing: analyzing an anonymized corpus of 70 million passwords.", IEEE Symposium on Security and Privacy , 2012.

Appendix A. Problems with Passwords

By far the most common mechanism for web authentication is passwords that can be remembered by the user, called "human-memorable

passwords". There is plenty of good research on how users typically use human-memorable passwords (e.g. see [bonneau]), 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 Basic 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.

Note that passwords that are not human-memorable are still subject to database attack, though are of course unlikely to be re-used across many systems. Similarly, database attacks of some form or other will work against any password based authentication scheme, regardless of the crytographic protocol used. So for example, zero-knowledge or PAKE schemes, though making use of elegant cryptographic protocols, remain as vulnerable to what is clearly the most common exploit seen when it comes to passwords. HOBA is however not vulnerable to database theft.

Appendix B. Example

The following values show an example of HOBA-http authentication to the origin https://example.com:443. Carriage-returns have been added and need to be removed to validate the example.

Public Key:

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAviE8fMrGIPZN9up94M28
6o38B99fsz5cUqYHXXJlnHIi6gGKjqLgn3P7n4snUSQswLExrkhSr0TPhRDuPH_t
fXLKLBbh17ofB7t7shnPKxmyZ69hCLbe7pB1HvaBzTxPC2KOqskDiDBOQ6-JLHQ8
egXB14W-641RQt0CsC5nXzo92kPCdV4NZ45MW0ws3twCIUDCH0nibIG9SorrBbCl
DPHQZS5Dk5pgS7P5hrAr634Zn4bzXhUnm7cON2x4rv83oqB3lRqjF4T9exEMyZBS
L26m5KbK860uSOKywI0xp4ymnHMc6Led5qfEMnJC9PEI90tIMcgdHrmdHC_vpldG
```

```
   DQIDAQAB
   -----END PUBLIC KEY-----

   Origin: https://example.com:443

   Key Identifier: vesscamS2Kze4FFOg3e2UyCJPhuQ6_3_gzN-k_L6t3w

   Challenge: pUE77w0LylHypHKhBqAiQHuGC751GiOVv4/7pSlo9jc=

   Signature algorithm: RSA-SHA256 ("0")

   Nonce: Pm3yUW-sW5Q

   Signature:

   VD-0LGVBVEVjfq4xEd35FjnOrIqzJ2OQMx5w8E52dgVvxFD6R0ryEsHcD31ykh0i
   4YIzIHXirx7bE4x9yP-9fMBCEwnHJsYwYQhfRpmScwAz-Ih1Hn4yORTb-U66miUz
   q04ZgTHm4jAj45afU20wYpGXY2r3W-FRKc6J6Glv_zI_ROghERalxgXG-QVGZrKP
   tG0V593Yf9IPnFSpLyW6fnxscCMWUA9T-4NjMdypI-Ze4HsC9J06tRTOunQdofr9
   6ZJ2i9LE6uKSUDLCD2oeEeSEvUR--4OGtrgjzYysHZkdVSxAi7OoQBK34EUWg9kI
   S13qQA43m4IMExkbApqrSg

   Authorization Header:

   Authorization: HOBA result="vesscamS2Kze4FFOg3e2UyCJPhuQ6_3_gzN-
   k_L6t3w.pUE77w0LylHypHKhBqAiQHuGC751GiOVv4/7pSlo9jc=.Pm3yUW-sW5Q
   .VD-0LGVBVEVjfq4xEd35FjnOrIqzJ2OQMx5w8E52dgVvxFD6R0ryEsHcD31ykh0
   i4YIzIHXirx7bE4x9yP-9fMBCEwnHJsYwYQhfRpmScwAz-Ih1Hn4yORTb-U66miU
   zq04ZgTHm4jAj45afU20wYpGXY2r3W-FRKc6J6Glv_zI_ROghERalxgXG-QVGZrK
   PtG0V593Yf9IPnFSpLyW6fnxscCMWUA9T-4NjMdypI-Ze4HsC9J06tRTOunQdofr
   96ZJ2i9LE6uKSUDLCD2oeEeSEvUR--4OGtrgjzYysHZkdVSxAi7OoQBK34EUWg9k
   IS13qQA43m4IMExkbApqrSg"
```

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