RESTful Authentication Pattern for the Hypertext Transport Protocol (HTTP)
draft-williams-http-rest-auth-03

Abstract

This document proposes a "RESTful" pattern of authentication for HTTP/1.0, 1.1, and 2.0. The goal is to make it easy to add authentication mechanisms to HTTP and to make it easy to implement them even without much help from the HTTP stack (though it is best to integrate authentication into the stack, of course).

Among other benefits of RESTauth: it is orthogonal to "HTTP routers" and proxies, it results in session Uniform Resource Identifiers (URIs) that can be DELETEd to logout, naturally supports multi-legged authentication schemes, and it can be universally implemented on the server side with the Common Gateway Interface (CGI) and FastCGI.

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 February 16, 2013.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents
# Table of Contents

1. Introduction ................................................. 4  
2. Conventions used in this document .......................... 6  
3. API-Imposed Constraints ................................. 7  
4. Protocol .................................................. 8  
4.1. Negotiable Parameters ............................... 8  
4.1.1. Strong Binding to TLS ............................... 9  
4.1.2. WWW-Authenticate Header Value Prefix Syntax .......... 9  
4.1.3. WWW-ChannelBinding-Types Header ................... 9  
4.1.4. WWW-ChannelBinding-Type Header .................... 10  
4.1.5. WWW-SessionBinding-Type Header .................... 10  
4.1.6. WWW-ReplayProtection Header ....................... 10  
4.2. Protocol Flow ......................................... 10  
4.2.1. One Round Trip Optimization: Challenges Born in WWW-Authenticate Headers ......................... 11  
4.3. Session Binding Types: Cookie, URI, and MAC ............ 11  
4.3.1. The New WWW-Session-URI Header ................... 12  
4.3.2. The New WWW-Session-MAC Header ................... 12  
4.3.3. A MAC Trailer?? ..................................... 12  
5. HTTP "Routing" and Authentication .......................... 13  
6. In-band HTTP Authentication Alternatives ................... 14  
7. Sample/Potential RESTauth Authentication Mechanisms ........ 15  
7.1. OAuth via RESTauth .................................... 15  
7.1.1. OAuth 1.0 ........................................... 15  
7.1.2. OAuth 2.0 ........................................... 15  
7.2. Adapting SSHv2 Authentication Mechanisms to RESTauth . 15  
7.2.1. RESTauth Mechanism Names for SSHv2 Userauth Methods .... 15  
7.2.2. Nonces .............................................. 16  
7.2.3. "Session ID" .......................................... 16  
7.3. Adapting IKEv2 Authentication Mechanisms to RESTauth . 16  
7.3.1. Adapting IKEv2 Password Authenticated Connection Establishment (PACE) to RESTauth ................. 16  
7.4. Using SASL Authentication Mechanisms with RESTauth .... 16  
7.4.1. Using SCRAM in RESTauth ........................... 17  
7.4.2. Using SCRAM with Round Trip Optimization in RESTauth .. 18  
7.5. Using GSS-API Authentication Mechanisms with RESTauth .... 19  
8. IANA Considerations ......................................... 20  
9. Security Considerations .................................... 21  
10. TODO ....................................................... 22  
11. References ............................................... 23  
11.1. Normative References .................................. 23  
11.2. Informative References ................................ 23  
Author’s Address .............................................. 25
1. Introduction

We propose a pattern for HTTP [RFC2616] [XXX add reference to HTTP/2.0 as well] authentication mechanisms that, by being "RESTful", obtains a number of benefits:

- authentication that works with all versions of HTTP, even when the authentication mechanism requires multiple round trips;
- compatibility with "HTTP routing" by making no assumptions that related requests are sent over the same TCP/TLS connection;
- a "session URI" results that can be used to multiplex multiple sessions onto the same TCP/TLS connections;
- a "session URI" results that can be DELETEd to effect logout;
- a "session URI" results that has better security semantics than HTTP cookies;
- the ability to refer to multiple sessions in one request wherever such a concept might be useful;
- can be implemented by any application without changes being required to any HTTP stack;
- can be implemented by the HTTP stack;
- on the server side this can be implemented entirely via CGI and FastCGI;
- by its RESTful nature, multi-legged authentication message exchanges are naturally handled without making any changes to HTTP.

There are probably other benefits not listed above.

The rough outline of the protocol is quite simple:

1. initial authentication messages are POSTed to an agreed-upon or indicated resource...

2. .....which then results in a new resource being created with the authentication reply message as the new resource’s representation.

3. Thereafter any additional authentication message exchanges needed (for multi-legged mechanisms) are POSTed to the new resource
creating additional resources.

4. The resource created by the POSTing of the initial authentication mechanism identifies the resulting session, and its URI is known as the session URI.

5. Session URIs can be used to multiplex multiple sessions over the same TCP/TLS connections, implement logout, and share sessions across multiple related servers.

Server-initiated authentication is also possible, whereby the server sends a challenge in 401 errors in headers. If the server gives the client a choice of mechanisms (multiple WWW-Authenticate header values) and the client picks one where the server sent a challenge, then the client consumes that message and POSTs subsequent ones to the indicated URI.
2. Conventions used in this document

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

To the extent that existing Application Programming Interfaces (APIs) assume specific styles of HTTP authentication message flows, if we want those APIs to support RESTauth backwards-compactly, then those APIs may impose constraints on RESTauth.

For example, the Android Account Manager API assumes a single round trip for authentication. But the Android Account Manager could perform all but the last round trip on behalf of the application, then let the application perform the last round trip. In order for that to work we need the authentication message exchange to be orthogonal to TCP/TLS connections - that is, we need it to be possible to use multiple TCP/TLS connections for completing a single authentication exchange.

A typical constraining characteristic might be that an API assumes the use of GET with tokens encoded into the URI, or that the API makes no room for the use of headers in authentication message exchanges.

One way to work around such constraints may be to provide various options in this framework. Another might be to use OAuth 1.0 [RFC5849] or 2.0 [I-D.ietf-oauth-v2] as a bridge: the API would use this framework under the covers then obtain OAuth credentials from the server that the application can then use in any way that the API’s form allows for.

[[anchor1: TODO: Add a table/list of various APIs and their characteristics that might constrain this framework.]]
4. Protocol

The are very few normative protocol elements here besides the outline given in Section 1. The normative protocol elements are:

- the form of the WWW-Authenticate header values for RESTauth mechanisms;
- several new headers for advertising negotiable parameters that are orthogonal to WWW-Authenticate;
- the POSTing of authentication messages from the client, with the initial client authentication message going to either a pre-agreed URI or to a URI named in the WWW-Authenticate headers;
- the creation of a session URI as a result of the initial POST, and the subsequent POSTing of any additional authentication messages to the session URI;
- the new session URI resource representation resulting from POSTs being the server’s response authentication message, if any;
- the DELETEion of session URIs as signaling logout;
- a new header for referencing session URIs in normal HTTP requests;
- the use of channel binding [RFC5056] to TLS [RFC5246] for session protection;
- the use of session keys as an option for integrity protection when TLS is not used (a light-weight security mode).

4.1. Negotiable Parameters

As can be seen in the ABNF in the preceding section, the server can offer some negotiable parameters. These are:

- Mechanism names;
- Channel binding types;
- Session binding types;
- Replay protection;

Each WWW-Authenticate header value offers a single mechanism and negotiable parameters for it. The WWW-ChannelBinding-Types header allows the server to offer channel binding types.
4.1.1. Strong Binding to TLS

Strong binding to TLS is provided via channel binding [RFC5056]. When a RESTauth mechanism provides strong authentication of the service to the user, the combination of RESTauth and channel binding results in strong authentication of the server to the user even though TLS is used for session transport protection.

4.1.2. WWW-Authenticate Header Value Prefix Syntax

The ABNF for RESTauth WWW-Authenticate header values is as follows:

```
challenge           = ( "RA-" mechname SP restauth-challenge )
mechname            = TBD
restauth-challenge  = ( login-uri SP session-types SP
                      replay-prot SP *1(mech-challenge) )
login-uri           = absoluteURI
session-types       = "s=" session-type /
                      (session-type ":" session-types)
session-type        = "cookie" / "session-ID" / "MAC"
replay-prot         = "r=" ("yes" / "no")
```

Figure 1: WWW-Authenticate ABNF

For a DIGEST-like mechanism it might look like "WWW-Authenticate: RA-Digest-SHA-256 tls-server-end-point session-ID no HE4SgWGrd/3+07t16HqusA==". For example, the mechname for the Kerberos V5 GSS-API mechanism might be "gss-krb5", and a WWW-Authenticate header value for it might look like "WWW-Authenticate: RA-gss-krb5 http://foo.example/restauth-login tls-server-end-point session-ID no".

Note that mechanisms that may be used include: GSS mechanisms, SASL mechanisms, ad-hoc mechanisms, and so on.

4.1.3. WWW-ChannelBinding-Types Header

A new header is added by which servers MUST indicate which channel binding [RFC5056] types -if any- they support for RESTauth authentication; if the server does not support channel binding then this header MUST be absent. The header is named WWW-ChannelBinding-Types. Its values are channel binding types from the channel binding type registry [RFC5929].
4.1.4. WWW-ChannelBinding-Type Header

A new header is added by which clients MUST indicate what channel binding type they used when POSTing RESTauth authentication messages, if any; if the client did not use channel binding then this header MUST be absent. If the mechanism used has its own method for indicating the use of channel binding, then this header MAY be omitted. The header is named WWW-ChannelBinding-Type. Its value is a channel binding type from the channel binding type registry [RFC5929].

4.1.5. WWW-SessionBinding-Type Header

A new header is added by which clients MUST indicate what session binding type they choose when POSTing RESTauth authentication messages. The header is named WWW-SessionBinding-Type. Its value is a session binding type as shown in Figure 1. This header SHOULD be present in RESTauth authentication HTTP requests, but may be omitted when the selected mechanism provides its own session binding facility that is distinct from RESTauth's (this helps adapt OAuth to RESTauth with minimal or no changes).

4.1.6. WWW-ReplayProtection Header

A new header is added by which clients MUST indicate whether they desire replay protection when POSTing RESTauth authentication messages. The header is named WWW-SessionBinding-Type. Its value is "yes" or "no" (defaults to "no" if absent) as shown in Figure 1.

Replay protection is to be used only when TLS [RFC5246] is not, and only if a session binding type of "MAC" is also requested.

4.2. Protocol Flow

RESTauth can be initiated by a client that knows a priori that it needs to or wants to use RESTauth. Servers can also tell clients that access to certain resources require authentication, possibly including RESTauth mechanisms. When the server tells the client that it must authenticate, the server may also give the client an initial authentication message for one or more mechanisms.

When the client knows a priori that it must authenticate then the client MUST know the RESTauth login URI a priori as well, as well as negotiable parameters, all of which the client might know from either an application protocol specification, or from caching this information from earlier RESTauth exchanges.

The server MUST use a 401 HTTP status code and WWW-Authenticate
headers to inform the client of the need to authenticate in order to access a given resource. For RESTauth mechanisms the WWW-Authenticate header values MUST conform to the ABNF given in Section 4.1.2.

To proceed the client chooses a suitable authentication mechanism (for which, presumably, it has credentials for a desired client identity), possibly a channel binding type, possibly a session type, and whether to use replay protection.

4.2.1. One Round Trip Optimization: Challenges Born in WWW-Authenticate Headers

Some mechanisms may optimize the protocol flow by allowing the server to include challenges in the 401 response’s WWW-Authenticate header values. DIGEST-MD5 works this way, for example, sending a challenge nonce to be fed into the digest function (along with other client-side inputs).

RESTauth allows this, but this feature is OPTIONAL: it must always be possible for a client to initiate RESTauth without first obtaining a challenge in a WWW-Authenticate header value, in which case the client must incur an extra protocol leg by obtaining the challenge (if it is at all necessary) in the server’s reply to the client’s first authentication message. There are two reasons for making this optional:

1. to allow client applications that know a priori that they must authenticate, requiring no further negotiation;
2. to support authentication frameworks that require that the client initiate authentication message exchanges.

A challenge may consist of a nonce, some encrypted or MACed nonce, a time-stamp, certificates and digital signatures, etcetera. The server may include a login URI in challenge-laden WWW-Authenticate headers where the login URI encodes secure state regarding the challenge (e.g., the challenge encrypted in a symmetric key known only to the server).

4.3.  Session Binding Types: Cookie, URI, and MAC

A notion of session binding type is added for binding HTTP requests to specific RESTauth login sessions. Three types are provided:
The traditional HTTP cookie approach to session binding;

HTTP requests carry a WWW-Session-URI header identifying the session(s) (similar to cookies, but without all the associated baggage);

HTTP requests carry a WWW-Session-URI header identifying the session(s) and a WWW-Session-MAC header that carries a MAC or MACs binding the session URI(s) to the request.

4.3.1. The New WWW-Session-URI Header

A new HTTP header is added called WWW-Session-URI whose values consist of session URIs. At least one session URI MUST be included. Each session URI is an absoluteURI. Session URIs MUST NOT have unescaped commas (’,’) embedded in them. Servers MAY fail to implement support for multiple session URIs being referenced by a single request, in which case they MUST answer with error code <TBD>. Servers MUST validate the session URI before processing the request; if the session URI is invalid the server MUST respond with a 401 (or TBD?) status code.

Note that referencing multiple session URIs is permitted, but this may not be meaningful for the application, thus the server MAY reject this (TODO: specify a status code for this?).

[[anchor2: I can imagine a webmail application where a client can be logged in as multiple users and get a unified view of the users’ mailboxes. This seems unlikely, but why rule out such use cases?]]

4.3.2. The New WWW-Session-MAC Header

[[anchor3: Describe the header, its values, algorithm agility, and what the MAC is to be taken over. Note too that this cannot apply to request contents as we have to consider chunking, and besides, a MAC of contents really has to go as a trailer, not a header.]]

4.3.3. A MAC Trailer??

[[anchor4: ... This is only needed for RESTauth *without* TLS, which will probably not be the common mode of use for RESTauth... unless we can produce a MAC trailer extension for HTTP/2.0, in which case this may well become a common mode of RESTauth usage.]]
5. HTTP "Routing" and Authentication

It is common to deploy HTTP services with load-balanced servers behind a load balancer and TLS concentrator. Other techniques may also result in a multiplicity of servers acting on behalf of a single service. The load balancers may even behave like routers and route HTTP requests to the same server for all requests in a single connection, or even route HTTP requests according to the verb and resource. It helps to be able to have a notion of authenticated sessions that can be referenced by all servers responding to a given service name.

The server end of a RESTauth authentication message exchange may be terminated by one server, by many servers sharing session state (via the resources named by session URIs), or by a server-side HTTP router. Once a RESTauth session is established we assume that all servers responding to the same service name will be able to access the session resource, validate session URIs, and obtain keys for computing and validating session binding MACs. Alternatively, the router may take responsibility for session binding and signal authorization information from the established session to the HTTP servers behind the router (however, we do not here specify any methods for such signaling).

By using REST for the authentication message exchange we allow this disconnection between "session" and "connection", which therefore facilitates "routing" of HTTP requests and even off-loading of authentication and/or session binding to HTTP "routers".

This approach should be flexible enough for all existing architectures for deploying HTTP services.
6. In-band HTTP Authentication Alternatives

RESTauth is "out-of-band" in the sense that the authentication messages are exchanged independently of the application’s requests for normal resources. Of course, RESTauth exchanges may well (and often will) happen in the same TCP/TLS connection as normal application requests, so RESTauth is not really out-of-band. We use "out-of-band" and "in-band" very loosely in this section.

There exist several "in-band" HTTP authentication alternatives where the authentication message exchanges happen in the context of application resources. Here the HTTP verb and resource are application-specific and have nothing to do with authentication, and the authentication messages are exchanged via HTTP request and response headers with the server responding with a 401 status code until authentication is complete.

The extant "Basic" and "DIGEST-MD5" [RFC2617] HTTP authentication methods, as well as HTTP/Negotiate [RFC4559] are "in-band" HTTP authentication methods.

In so far as an in-band authentication method results in a cookie or session URI/ID the distinction between in-band and out-of-band is almost trivial, as described above: authentication messages in headers vs. bodies, and HTTP verb and URL. However, if in-line authentication methods are strongly tied to the TCP/TLS connections over which they were utilized then that is a big disadvantage over RESTauth: each connection requires re-authenticating, and support for HTTP routing schemes is not clear.

HTTP/Negotiate is more troublesome because historically it has required re-authentication per-HTTP request(!).

Even if the only difference between in-band and out-of-band is a trivial one, using the REST pattern means that authentication can be implemented using with no help from the HTTP stack (even though it’s desirable to have it implemented within/by the HTTP stack), whereas there may not be a way to implement in-band authentication without help from the HTTP stack for some stacks.
7. Sample/Potential RESTauth Authentication Mechanisms

Here we describe (informatively, for the time being) how to use or adapt a variety of authentication mechanisms, from SSHv2, IKEv2, SASL, GSS-API, and other frameworks, so as to quickly gain a set of usable mechanisms, both, specification- and implementation-wise.

This section is also intended to show that adding RESTauth mechanisms is easy.

7.1. OAuth via RESTauth

OAuth 1.0 [RFC5849] and OAuth 2.0 [I-D.ietf-oauth-v2] are commonly deployed. Being able to use OAuth via RESTauth would be useful. We attempt to make RESTauth such that at least for OAuth 1.0 there is a standard way to use OAuth such that it conforms to RESTauth.

7.1.1. OAuth 1.0

For OAuth 1.0 [RFC5849] the "form-encoded body" form (see section 3.5.2 of [RFC5849]) of OAuth 1.0 conforms to RESTauth without further changes.

7.1.2. OAuth 2.0

[It looks like OAuth 2.0 [I-D.ietf-oauth-v2] also uses POST to send tokens to the server, and it looks like it too effectively conforms to RESTauth.]

7.2. Adapting SSHv2 Authentication Mechanisms to RESTauth

SSHv2 "userauth" mechanisms [RFC4252] typically involve a digital signature (or similar) of an SSHv2 session ID. There is no such thing as an SSHv2 session ID in HTTP. A session URI cannot serve as a stand-in for an SSHv2 session ID because a) the session URI is an outcome of authentication in RESTauth, b) to prevent cut-n-paste and replay attacks the client and the server both must contribute to the entropy of the session ID that is signed by the client.

In order to adapt SSHv2 userauth methods properly (i.e., securely), we have replace the SSHv2 session ID in the to-be-signed data with a hash of the channel binding and nonces contributed by the client and the server. As an optimization the server nonce can be sent as a challenge (this saves a round trip).

7.2.1. RESTauth Mechanism Names for SSHv2 Userauth Methods

For hash agility reasons the hash function name is part of the SSHv2 RESTauth mechanism name. To avoid "multi-level negotiation" the
SSHv2 userauth method name is also part of the RESTauth mechanism name.

The RESTauth mechanism name form for SSHv2 userauth methods, then, is: ssh-<SSHv2-userauth-method-name>-<hash-function-name>.

The following RESTauth mechanisms are defined here:

- ssh-publickey-SHA-256
- ssh-hostbased-SHA-256

### 7.2.2. Nonces

The client and the server must each contribute 128-bit nonces.

### 7.2.3. "Session ID"

The ssh-publickey-SHA-256 and ssh-hostbased-SHA-256 mechanisms use the following instead of a traditional SSHv2 session ID:

- SHA-256(channel_binding || server_nonce || client_nonce)

Here the <channel_binding> is as per-[RFC5056]: the channel binding type name, followed by the channel binding data (e.g., ‘tls-server-end-point’ followed by the server EE certificate as sent in the server’s TLS Certificate message).

Note that use of channel binding when using SSHv2 mechanisms is REQUIRED so as to defeat cut-n-paste attacks by weakly-authenticated servers.

### 7.3. Adapting IKEv2 Authentication Mechanisms to RESTauth

[[anchor5: TBD.]]

#### 7.3.1. Adapting IKEv2 Password Authenticated Connection Establishment (PACE) to RESTauth

[[anchor6: TBD.]]

### 7.4. Using SASL Authentication Mechanisms with RESTauth

Simple Authentication and Security Layers (SASL) [RFC4422] is a simple, pluggable framework for authentication mechanisms.

To use a SASL mechanism in RESTauth just prefix "SA-" to the SASL mechanism name and use that as the RESTauth mechanism name. If the
SASL mechanism is server-initiated then the server’s challenge is sent in the server’s WWW-Authenticate header value as described above. All other SASL authentication messages are exchanged as described above (i.e., via POSTs, first to the login URI, then to the session URI, with response messages as the new representation of the session resource).

The HTTP status code functions as the application’s outcome of authentication message. If SASL succeeds but authorization fails then the server should respond with a 401 status code to the POST of the final SASL authentication message from the client.

The server’s WWW-Authenticate header values function as the mechanism listing operation. SASL security considerations [RFC4422] [RFC5801] apply (particularly regarding the negotiation of channel binding support).

7.4.1. Using SCRAM in RESTauth

The Salted Challenge Response Authentication Mechanism (SCRAM) [RFC5802] is a DIGEST-like mechanism for SASL. Nothing special is needed to use SCRAM versus any other SASL mechanism, except for a round trip optimized form of SCRAM, if we decide to pursue that (see Section 7.4.2).

The following figure shows what SCRAM in RESTauth looks like. Note that the resource representations are taken verbatim from [RFC5802].
C->S: GET /some-resources HTTP/1.1
    Host: A.example

S->C: HTTP/1.1 401 Unauthorized
    WWW-Authenticate: RA-SA-SCRAM-SHA-1 \
        http://A.example/rest-sa-scram \n        s=session-ID,MIC r=no
    WWW-ChannelBinding-Types: tls-server-end-point

C->S: POST /rest-sa-scram HTTP/1.1
    Host: A.example
    WWW-ChannelBinding-Type: tls-server-end-point
    WWW-SessionBinding-Type: session-ID
    Content-Type: application/octet-stream
    Content-Length: nnn

    n,,n=user,r=fyko+d21bbFgONRv9qkxdawL

S->C: HTTP/1.1 201
    Location: http://A.example/restauth-9d0af5f680d4ff46
    Content-Type: application/octet-stream
    Content-Length: nnn

    r=fyko+d21bbFgONRv9qkxdawL3rfcNHYJY1ZVvWVs?j,
    s=QSXCR+Q6sek8bf92,i=4096

C->S: POST /restauth-9d0af5f680d4ff46 HTTP/1.1
    Host: A.example
    Content-Type: application/octet-stream
    Content-Length: nnn

    c=biws,r=fyko+d21bbFgONRv9qkxdawL3rfcNHYJY1ZVvWVs?j,
    p=v0X8v3Bz2T0CJGbJQyF0X+HI4Ts=

S->C: HTTP/1.1 200
    Content-Type: application/octet-stream
    Content-Length: nnn

    v=rmF9pqV8S7suAoZWja4dJRkFsQ=

Figure 2: RESTauth w/ SCRAM

7.4.2. Using SCRAM with Round Trip Optimization in RESTauth

[[anchor7: This might work by having the authentication ID function as the salt and the server offering a challenge nonce and iteration count in its optimistic challenge. However, it’s not clear that a round trip optimized form of SCRAM is desirable.]]
The following figure shows what a round trip optimized RESTauth w/ SCRAM exchange might look like.

[[anchor8: NOTE: SCRAM was not intended to be used this way. In particular this approach forces the use of an algorithmic salt, to be derived only from either the username or the username and the server’s name (or else to be remembered by the user, but that’s not likely).]]

C->S: GET /some-resources HTTP/1.1
   Host: A.example

S->C: HTTP/1.1 401 Unauthorized
   WWW-Authenticate: RA-SA-SCRAM-SHA-1 \ http://A.example/rest-sa-scram \ s=session-ID,MIC r=no \ r=fyko+d2l...JY1ZVvWVs7j,i=4096
   WWW-ChannelBinding-Types: tls-server-end-point

C->S: POST /rest-sa-scram HTTP/1.1
   Host: A.example
   WWW-ChannelBinding-Type: tls-server-end-point
   WWW-SessionBinding-Type: session-ID
   Content-Type: application/octet-stream
   Content-Length: nnn

   n,,n=user,r=fyko+d2lbFgONRv9kxdawL,
c=biws,r=fyko+d2lbFgONRv9kxdawL3rfcNHYJY1ZVvWVs7j,
p=v0X8v3B2TOCJGbJQyF0X+HI4Ts=

S->C: HTTP/1.1 200
   Content-Type: application/octet-stream
   Content-Length: nnn

   v=rmF9pqV8S7suAoZWja4dJRkFsKQ=

Figure 3: RESTauth w/ round trip optimized SCRAM

7.5. Using GSS-API Authentication Mechanisms with RESTauth

The Generic Security Services Application Programming Interface (GSS-API) [RFC2743] is another pluggable mechanism framework. Any GSS-API mechanism that supports channel binding [RFC5056] can be used as SASL mechanisms via the "SASL/GS2" bridge [RFC5801]. This includes the Kerberos V5 GSS-API mechanism [RFC4121].
8. IANA Considerations

TBD (header registrations, ...)
9. Security Considerations

This entire document deals with security considerations. [Add more, like about channel binding, same-origin-like constraints on the login and session absolute URIs', ...]
10. TODO

[[anchor9: Add references (to HTTP/2.0, CGI/fCGI, ...).]]

[[anchor10: Describe MAC session binding option and replay protection in detail. Describe how to extract keys for MAC keying from SASL/GSS/PACE.]]

[[anchor11: Figure out how to adapt IKEv2 password-based methods to RESTauth. This may not be worthwhile (since each method tends to depend heavily on the entire IKEv2 framework in ways that add messaging that we’d not need in RESTauth).]]
11. References

11.1. Normative References


11.2. Informative References


Author’s Address

Nicolas Williams
Cryptonector, LLC

Email: nico@cryptonector.com