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 applications 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). Another goal is to make it easy to reuse existing authentication mechanisms by allowing the user (that is, the server's operators) to choose what concrete authentication mechanism(s) to use.

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, naturally supports clustering, and can be universally implemented on the server side with such server<->application interfaces as the Common Gateway Interface (CGI) and FastCGI, among others.

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 15, 2014.

Copyright Notice
Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction .................................................. 4
1.1. Motivation .................................................. 4
1.1.1. Authentication Infrastructure and Credentials Reuse .... 5
1.2. Protocol Outline ............................................ 6
1.3. API-Imposed Constraints ..................................... 7
1.4. Conventions used in this document ............................ 7
2. Alternatives .................................................. 8
2.1. In-band HTTP Authentication ................................... 8
2.2. Out-of-Band Bearer Token Mechanisms ........................... 9
2.3. Authentication in TLS ........................................ 9
3. Protocol ...................................................... 10
3.1. Negotiable Parameters ....................................... 10
3.1.1. Strong Binding to TLS .................................... 11
3.1.2. WWW-Authenticate Header Value Prefix Syntax ............... 11
3.1.3. WWW-ChannelBinding-Types Header ........................ 12
3.1.4. WWW-ChannelBinding-Type Header .......................... 12
3.1.5. WWW-SessionBinding-Type Header ........................... 12
3.1.6. WWW-ReplayProtection Header ............................. 12
3.2. Protocol Flow ............................................... 12
3.2.1. One Round Trip Optimization: Challenges Born in WWW-Authenticate Headers .................. 13
3.3. Session Binding Types: Cookie, Channel Bound Session URI, and MAC ............................. 14
3.3.1. The New WWW-Session-URI Header ............................ 14
3.3.2. The New WWW-Session-MAC Header .......................... 14
3.3.3. To MAC or not to MAC; A MAC Trailer?? .................... 15
4. Representation of Authenticated Session Resources ................. 16
5. Session URI Origin and Scope .................................. 17
6. HTTP "Routing" and Authentication ................................ 18
7. Actual Authentication Mechanisms ................................ 19
7.1. OAuth via RESTauth ......................................... 19
7.1.1. OAuth 1.0 ................................................ 19
1. Introduction

There is a great need for improved authentication options in HTTP applications, both web browser and non-browser applications. At this time there are a number of proposals being made in the HTTPauth Working Group (WG). This proposal is just one of many.

Some of the goals of RESTauth are:

- an authentication protocol that reuses existing authentication mechanisms -- in principle any and all mechanisms that we have in the Internet protocols, including: SASL [RFC4422], GSS-API [RFC2743], SSHv2 [RFC4251], IKEv2 [RFC5996], and other frameworks, as well as ad-hoc mechanisms and/or standard mechanisms used outside such frameworks;

- an authentication protocol that layers above HTTP so that applications may implement it with little or no help from (read: modifications to) the existing HTTP stacks that they use, all the while not precluding native support for RESTauth by any HTTP stack;

- an authentication protocol that supports a notion of sessions without depending on having a single HTTP/TLS/TCP connection, and which can be logged out explicitly, with explicit scoping of sessions;

- an authentication protocol that naturally supports HTTP server-side routing and clustering.

We propose a pattern for HTTP [RFC2616] [TODO: add reference to HTTP/2.0 as well?] authentication mechanisms that, by being "RESTful", obtains these goals naturally.

1.1. Motivation

Existing HTTP authentication mechanisms leave much to be desired.

Existing "in-band" mechanisms:

- Basic [RFC2617];

- Digest [RFC2617];

- Negotiate [RFC4559].

Other existing mechanisms:
o  ad-hoc username-and-password authentication using HTML forms 
   POSTed over HTTP, plus web cookies;

o  various bearer token mechanisms [TODO: add examples, such as OAuth];

o  various non-bearer token mechanisms which are not generalized or 
   generalizable to a larger universe of authentication mechanisms 
   [TODO: add examples, such as BrowserID].

In enterprise and educational settings it is common to need support 
for Kerberos [RFC4120]. Of the above only Negotiate supports 
Kerberos meaningfully or at all, and does so badly, requiring re-
authentication for every request in many implementations (among other 
problems; see Section 2.1 for more).

Enterprises and educational settings also often have RADIUS 
[RFC2865], often used via EAP [RFC3748]. These could be supported by 
the generic security mechanism based on EAP, RADIUS, and SAML being 
developed by the ABFAB WG [TODO: add references!]. These mechanisms 
in particular will be difficult to use via Negotiate because they 
involve multiple round trips, which Negotiate supports half-
heartedly; see Section 2.1.

1.1.1. Authentication Infrastructure and Credentials Reuse

All too commonly the community invents new authentication mechanisms 
that require their own authentication infrastructures or bridges to 
other mechanisms’ existing infrastructures. This is a costly habit: 
costly for those who must deploy these. A pattern of authentication 
mechanism reuse would greatly improve this situation.

Most, if not all [classical, not quantum mechanical] authentication 
mechanisms involve an exchange of one or more authentication 
messages, accept as input the names of the peers being authenticated, 
and on success output some information such as the names of the 
authenticated peers, trust transit paths, session keys, and so on.

Actual mechanisms differ in minor details, such as which party sends 
the first authentication mechanism (but there is always an initiator, 
even if initiation is implied by connecting to a network, for example 
as in EAP, which can be thought of as sending an empty initial 
authentication message). These differences can be abstracted. 
Indeed, we have at least _five_ authentication mechanism frameworks 
in Internet protocols:
o Generic Security Services (GSS-API) [RFC2743];

o Simple Authentication and Security Layers (SASL) [RFC4422];

o Secure Shell version 2 (SSHv2) [RFC4251];

o Internet Key Exchange Protocol versions 1 and 2 (IKEv2) [RFC5996];

o Extensible Authentication protocol (EAP) [RFC3748].

Five authentication frameworks is an embarrassment of riches. And yet we continually implement new ad-hoc authentication mechanisms -- this is not just embarrassing (it isn’t really embarrassing, as it is a part of the human condition that we continually re-invent things), but wasteful, both because it means we fail to reuse existing code and specifications, and because it means users are faced with costly deployment headaches.

There are also many authentication mechanisms that support (or could easily be extended to) federation.

The desire to re-invent authentication mechanisms (and frameworks) to avoid technologies of the past (e.g., ASN.1 and its encoding rules) or specific instantiations of them (e.g., GSS-API) is understandable, but at the very least it should be made easy for developers to add application support for arbitrary authentication mechanisms of any given users choice. The only way to enable use of the user’s choice of authentication mechanism is through a common protocol that embeds that authentication mechanism’s messages, and that is _exactly_ what RESTauth aims to be for HTTP.

1.2. Protocol Outline

1. initial authentication messages are POSTed to an agreed-upon or indicated "login" 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 without 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.

Authentication using mechanisms that require that the server send the first authentication message is also possible, in either of two ways: the initial authentication message is sent in headers in a 401 response, or the client POSTs an empty first message to the login resource.

1.3. 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-compatibly, then those APIs may impose constraints on RESTauth.

For example, the Android Account Manager API assumes a single round trip for authentication [TODO: add reference!]. 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. This is because the application and the account manager will likely be using different TCP/TLS connections.

A typical constraining characteristic might be that an API assumes the use of GET with tokens encoded into the URI or into a header, 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 RESTauth. Another might be to use OAuth 1.0 [RFC5849] or 2.0 [RFC6749] 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 known APIs and their characteristics that might constrain this and/or other frameworks.]]

1.4. 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].
2. Alternatives

2.1. In-band HTTP Authentication

RESTauth is "out-of-band" in the sense that the authentication messages are exchanged independently of the application’s requests for normal resources, with authentication tokens sent as message bodies rather than as header values. Of course, RESTauth exchanges may well (and often will) happen in the same TCP/TLS connection as normal application requests, so RESTauth is not out-of-band in the sense of using distinct transport connections. 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. Indeed, in common implementations of HTTP Basic, Digest, and Negotiate, cases every request requires authentication, which can be particularly costly for the Negotiate case.

Additionally, Negotiate can require multiple round trips but provides no mechanism for explicitly associating a given GSS-API security context token with a given GSS-API security context: it has to be assumed that all messages are serialized and only one multi-round trip security context establishment can be ongoing in any given HTTP connection.

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.

### 2.2. Out-of-Band Bearer Token Mechanisms

Some HTTP/web authentication mechanisms used on the web involve out-of-band (that is, outside the application’s HTTP connections to a server) communications to get bearer tokens, which the application then includes in its HTTP requests (or perhaps it POSTs them to the target server).

The main problem with bearer token systems is that they depend utterly on the ability of TLS to authenticate services. Other mechanisms, ones that depend on proof-of-secret/private-key-possession, can often provide better security in the face of weak TLS server authentication.

### 2.3. Authentication in TLS

A number of proposals use TLS [RFC5246] for authentication, some adding extensions for sending user credentials encrypted to avoid the overhead of renegotiation for privacy protection of user credentials.

There are several problems with user authentication in TLS:

- authentication mechanisms requiring multiple round trips are not supported, and though there may not be any reason to not permit them, in practice any extension of TLS handshakes to multiple round trips is likely to cause much trouble in adapting existing TLS implementations;

- new interfaces are required, affecting HTTP stacks and applications both in addition to TLS itself (compare to RESTauth, which requires only changes to applications, and which welcomes, but does not require, support by HTTP stacks), and we know from experience that it takes a long time to deploy solutions that require modifying implementations of multiple network layers.
3. Protocol

The are 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 -in 401 responses- 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;

And for applications that may not use TLS/HTTPS:

- the use of session keys as an option for integrity protection when TLS is not used (a light-weight security mode); see [I-D.williams-websec-session-continue proto].

3.1. Negotiable Parameters

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

- Authentication mechanism names;
- Channel binding types;
- Session binding types;
- Replay protection;
Each WWW-Authenticate [RFC2617] header value offers a single mechanism and negotiable parameters for it; because headers can have multiple values, WWW-Authenticate provides a method for negotiating authentication mechanisms. The WWW-ChannelBinding-Types header (added here) allows the server to list channel binding types supported by it.

3.1.1. 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.

3.1.2. WWW-Authenticate Header Value Prefix Syntax

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

```
challenge           = ( "RA-" mechname SP restauth-challenge )
mechname            = 1*( ALPHA / DIGIT / "-" )
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" /
                    "channel-bound-session-ID" /
                    "MAC"
                    ; new session-types may be added
replay-prot         = "r=" ("yes" / "no")
mech-challenge      = <base64 encoded mech-specific data>
```

Figure 1: RESTauth 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+O7t16HqusA==". 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 channel-bound-session-ID r=no".

Note that mechanisms that may be used include: GSS mechanisms, SASL mechanisms, ad-hoc mechanisms, and so on.
3.1.3. 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, such as the TLS channel binding types [RFC5929].

3.1.4. 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].

3.1.5. 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).

3.1.6. 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 used, and only if a session binding type of "MAC" is also requested.

3.2. 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 (using a 401 response, as usual), 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 3.1.2.

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.

3.2.1. 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 may 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 (and how to), requiring no further negotiation;
2. to support authentication mechanisms 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).

3.3. Session Binding Types: Cookie, Channel Bound Session URI, and MAC

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

Cookies  The traditional HTTP cookie approach to session binding;

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

Channel Bound Session URI  Like Session URI, but may only be used in HTTPS connections with the same channel bindings. (This implies use of the ‘tls-server-end-point’ channel binding type.)

MAC  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.

3.3.1. 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?]]

3.3.2. 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.]]
3.3.3. To MAC or not to MAC; A MAC Trailer??

[[anchor5: ... 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.]]

[[anchor6: We may want to remove this anyways and leave it for a session continuation spec. Or we may want to require the use of HTTPS.]]
4. Representation of Authenticated Session Resources

It will generally be useful to be able to GET a session resource to obtain information about the authenticated user. A GET on a session resource which is not fully established SHOULD return an empty body.

[[anchor7: TODO: Add a media type for session resource representation.]]

[[anchor8: Use JSON instead of ABNF? A schema language would be nice.]]

```abnf
session   = 1*( session-param )
session-param = session-param-name '=' session-param-value
session-param-name = 1*( ALPHA / DIGIT / '-' / '_' )
session-param-value = <quoted or base64>
```

Figure 2: Session resource representation ABNF

Session parameters include:

- established "true" or "false"
- expiration_time Datetime when the session expires.
- session_key_MAC_req Session key for MACs in requests.
- session_key_MAC_resp Session key for MACs in responses.
- authorization_data Information about the authenticated user.
- user_id The authenticated user identity.

The server MAY exclude any part of this when the entity requesting a session resource is the session’s user. The server MUST exclude (or respond with 401) all of the session resource’s representation when the entity requesting it is not authenticated or authorized to see it. The server SHOULD exclude locally-determined authorization_data and/or user_id information when the entity requesting the resource is the session’s user.
5. Session URI Origin and Scope

[[anchor9: TODO: Add a notion of session origin and scope. The origin can probably be determined naturally from the session URI. The scope should be set by the server. Perhaps we can also have the scope reflected in the session URI’s representation, which would allow the scope of a session to change over time.]]

[[anchor10: Clearly, using a session from one origin at another will require a channel binding verification operation. This will have to be added.]]
6. 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.
7. Actual 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.

Reuse of existing authentication mechanisms is a key goal of RESTauth: let us stop inventing wheels that require costly deployment of new authentication infrastructures (and credentials) and/or bridges to other authentication infrastructures.

7.1. OAuth via RESTauth

OAuth 1.0 [RFC5849] and OAuth 2.0 [RFC6749] 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 [RFC6749] 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:

  o ssh-publickey-SHA-256
  o 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:

  o 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

[[anchor11: TBD.]]

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

[[anchor12: 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 \ 
        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+d21bbFgONRv9qkxdawL3rfcNHYJY1ZVvWVs7j,
        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+d21bbFgONRv9qkxdawL3rfcNHYJY1ZVvWVs7j,
        p=v0X8v3Bz2T0CJGbJQyF0X+HI4Ts=

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

    v=rmF9pqV8S7suAoZWja4dJrkFsKQ=

Figure 3: RESTauth w/ SCRAM

7.4.2. Using SCRAM with Round Trip Optimization in RESTauth

[[anchor13: 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.

[[anchor14: 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\n        http://A.example/rest-sa-scram\n        s=session-ID,MIC r=no\n        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+d21bbFgONRv9qkxdawL,\n    c=biws,r=fyko+d21bbFgONRv9qkxdawL3rfcNHYJY1ZVvWVs7j,\n    p=v0X8v3Bz2T0CJGjQ3FyFOX+HI4Ts=

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

    v=rmF9pqV8S7suAo2Wja4dJRkFsqKQ=

Figure 4: 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].

GSS-API security mechanisms could also be used without SASL/GS2, but SASL/GS2 barely adds any overhead or complexity (a SASL
A simple header is to be prefixed to the initial security context token and to the channel binding data, with both peers always providing channel binding data.
8. Implementation Advice

RESTauth can be implemented without having to modify the HTTP nor TLS stacks.

The simplest thing to do is to implement a small API to produce and consume HTTP messages.

8.1. Server-Side Implementation Advice

On the server side the simplest thing to do is to implement POST handlers for the login and session resource URI namespace. The aspects of implementation that will be stack-specific are:

- WWW-Authenticate header value generation;
- Routing of GET and POST requests on the login and session resources to the RESTauth handlers;
- The interface between the web server and the handlers.

The FCGI interface is widely supported, allowing RESTauth to be implemented in a nearly universal way on the server side.

8.1.1. Channel Binding on the Server Side

The simplest way to implement channel binding on the server side is to use ‘tls-server-end-point’ channel bindings, using a table of server certificates indexed by fully-qualified server hostname values, with the Host: header value used to index this table. The server’s end entity (EE) certificate is the channel binding data.

8.1.2. Server Cluster / Routing Support

Because each RESTauth session is a first-class resource named by a URI (natch), multiple servers behind a given origin may recognize and handle all the sessions that they should be able to: all the sessions created at that origin, and all the sessions scoped to include that origin. This is handled by the simple expedient of doing an HTTP GET of the session resource claimed by a client. Note that the representation of a session resource can be cached easily, and updates can be checked with HEAD or conditional GETs, as one would expect of any RESTful HTTP API.

It is important that server MUST NOT attempt to GET (or HEAD) session resources for origins that the server does not respond to or which the server does not expect to share sessions with the server’s origin(s).
8.2. Client-Side Implementation Advice

There are many HTTPS client stacks, too many, each with its own APIs, to make it possible to implement RESTauth universally. The application will have to bridge any generic RESTauth APIs with the HTTP stack.

A reasonable implementation strategy is to build a generic RESTauth interface that

- consumes 401 responses (or just their WWW-Authenticate header values)
- consumes channel binding data (see below)
- generates HTTP requests to send to login or session URIs
- consumes HTTP responses to requests sent to login or session URIs

The application would have to extract relevant responses and channel binding data from the HTTP stack to feed to the RESTauth interface, and it would have to extract requests from the RESTauth interface and feed them to the HTTP stack.

The interfaces, abstractly, should be:

- `restauth_new()` Create a RESTauth context handle;
- `restauth_401()` Consume a 401 and update or create a RESTauth context handle;
- `restauth_login()` Produce a POST to a login resource URI (either provided explicitly or obtained from a 401);
- `restauth_login_continue()` Consume a response to a POST to a login resource and output a) status (complete or continue), b) possibly a POST to a session URI;
- `restauth_logout()` Produce a DELETE of the given session URI or the given RESTauth context’s session URI.

Channel binding type and data would be given to `restauth_login()`.

Specific programming language bindings of this API are easy enough to produce. Whether the API outputs or consumes complete messages or decomposed messages (start-line, headers, body, trailers) will depend on the APIs of the HTTP stack being targeted. A utility API that composes/decomposes HTTP messages will help make a RESTauth
implementation widely reusable.

8.2.1. Channel Binding on the Client Side

Most HTTPS stacks provide a way for the client application to obtain the server’s EE certificate; this is sufficient to implement ‘tls-server-end-point’ channel binding.
9. IANA Considerations

TBD (header registrations, ...)

Internet-Draft RESTful Authentication August 2013

Williams Expires February 15, 2014 [Page 28]
10. 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’, ...]

Note that though servers can GET/HEAD session resources, they MUST only do it for session resources for recognized origins. See Section 8.1.2.

[[anchor15: ...]]
11. TODO

[[anchor16: Add references (to HTTP/2.0, CGI/fCGI, ...).]]

[[anchor17: Decide whether to support a MAC type of session continuation, or only channel bound sessions.]]

[[anchor18: Describe or remove the MAC session binding option and replay protection in detail -- or remove it altogether. Describe how to extract keys for MAC keying from SASL/GSS/PACE.]]

[[anchor19: 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).]]

[[anchor20: Normatively specify bindings to SASL and/or GSS-API security mechanisms. Include support for BrowserID.]]
12. References

12.1. Normative References


12.2. Informative References


Williams                Expires February 15, 2014              [Page 31]


Author’s Address

Nicolas Williams
Cryptonector, LLC

Email: nico@cryptonector.com