NETWORK WORKING GROUP
Internet-Draft
Intended status: Standards Track
Expires: December 4, 2012

N. Williams
Cryptonector

June 2, 2012

RESTful Hypertext Transfer Protocol Application-Layer Authentication Using Generic Security Services

draft-williams-rest-gss-01

Abstract

This document describes a method for application-layer authentication in Hypertext Transfer Protocol (HTTP) applications using Generic Security Services Application Programming Interface (GSS-API) mechanisms via, for simplicity, the Simple Authentication and Security Layers (SASL) mechanism bridge known as "GS2". This approach to authentication allows for pluggability, mutual authentication, and channel binding, all with no changes to HTTP nor the Transport Layer Security (TLS).

We hope that the use of mutual authentication and channel binding at the application layer will make phishing more difficult. We hope that the use of authentication at the application layer will make REST-GSS deployable.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction 4
1.1. On Application-Layer Authentication Services 5
1.2. Conventions used in this document 5
1.3. GSS-API and SASL Primer 5
1.4. Channel Binding Primer 7
1.5. Glossary 7
2. The Protocol 9
2.1. Authentication Message Format 10
2.1.1. ABNF for Initial Authentication Message Header 10
2.2. Authentication State Cookies 11
2.3. Target Service Naming 11
2.4. Authorization ID Form 12
2.5. Mechanism, Channel Binding Type, and Other Negotiation . . 12
2.6. Session Status Representation 13
2.7. Session Binding via MIC Tokens 13
2.8. Alternative Session Binding Options 14
2.9. Server Indication of Authentication Requirement 15
3. Examples 15
3.1. Server Decides When to Authenticate 15
3.2. Mechanism and Other Negotiation 15
3.3. Login, Session, and Logout, with SCRAM 16
4. Implementation and Deployment Considerations 17
4.1. Desired GSS-API Extensions 18
5. IANA Considerations 18
6. Security Considerations 18
6.1. User Interface and Scripting Interface Recommendations . . 20
6.2. Platform Integration 21
6.3. Anti-Phishing 21
7. References 21
7.1. Normative References 21
7.2. Informative References 22
Author's Address 23

1. Introduction

Hypertext transfer Protocol (HTTP) [RFC2616] applications often require authentication and related security services. These applications have a plethora of odd choices for authentication functioning at various different network layers. For example: Transport Layer Security (TLS) [RFC5246] with pre-shared secret keys (PSK), TLS with user certificates [RFC5280], HTTP Basic and Digest authentication, HTTP/Negotiate, posting of HTML forms with usernames and passwords filled in, and various methods based on passing tokens via HTTP redirection, such as OAuth and OpenID [add references].

All the authentication methods currently available to HTTP applications leave something to be desired. For example these authentication methods operate at various different network layers, making abstraction of security services particularly difficult. Another problem is the lack of a secure method of tying all of a logged-in session's HTTP requests and responses to the session, with most browser-based applications using "cookies".

We propose an alternative method of authentication that operates at the application layer, and which provides applications with access to a large number of actual security mechanisms. This method is based on an exchange of authentication messages via HTTP POST to either a well-known URI or to a URI indicated by the server or agreed a priori. These authentication messages are mostly those of mechanisms defined for the GSS-API [RFC2743]. Channel binding [RFC5056] is used to bind authentication to TLS channels. Sessions are referenced via a session URI that is indicated and authenticated in all requests for a session.

The appeal of this solution is that a) it is build on off-the-shelf technologies, b) requiring no modifications to either HTTP nor TLS, c) that puts the application in control of authentication, and d) is pluggable, all the while improving security for HTTP applications whenever GSS mechanisms are used that provide mutual authentication.

The GSS-API, and through the "GS2" mechanism bridge, Simple Authentication and Security Layers (SASL), enjoys a large and growing number of security mechanisms, such as Kerberos V5 [RFC4121], SCRAM [RFC5802], as well as a PKI-based mechanism [Add reference to PKU2U], mechanisms based on OAuth [RFC5849], OpenID [I-D.ietf-kitten-sasl-openid], SAML [I-D.ietf-kitten-sasl-saml], and EAP [I-D.ietf-abfab-gss-eap], as well as various legacy mechanisms such as NTLM [add reference] and a Diffie-Hellman mechanism [add reference].

Much of this document assumes some reader familiarity with the GSS-

API and SASL. To aid readers new to the GSS-API we provide a GSS primer section, below.

1.1. On Application-Layer Authentication Services

The application layer is generally the most convenient for running authentication services that applications require. On the other hand, lower network layers have usually been more convenient for implementing transport security. As a result many existing Internet applications provide for both, but historically with no binding between authentication and transport security, and often providing two transport security options: one at the application layer, and one below. [Add a list of representative SASL and GSS-API apps and references, such as IMAP, POP3, SMTP/SUBMIT, LDAP, DNS (GSS-TSIG), FTP, SSHv2, etcetera].

The main disadvantage of application-layer authentication has been that until recently many applications had to provide options for two different "security layers": TLS (below the application layer) and SASL (at the application layer), and sometimes both might be used at the same time without any binding between them. The advent of standards for channel binding [RFC5056] [RFC5929] makes the combination of application-layer authentication with transport security at lower layers realistic. Therefore we may now consider solutions that we might once not have.

1.2. 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] .

1.3. GSS-API and SASL Primer

The GSS-API and SASL are both simple security frameworks providing pluggable authentication services and transport protection facilities to applications. By "pluggable" we mean that multiple "security mechanisms" may be used by applications without requiring different specifications for how the applications use each security mechanism. Moreover, application programming interfaces (APIs) for GSS and SASL can also be pluggable, requiring no changes to applications in order for them to use new mechanisms.

A "security mechanism" is an authentication protocol that conforms to the requirements of the framework in which it fits, and it provides the requisite authentication facilities. There are many examples of security mechanisms [add some].

The two frameworks are sufficiently similar to each other that a "bridge" has been added such that all GSS mechanisms may be used as SASL mechanisms as well. This bridge is known as "GS2" [RFC5801]. For the remainder of this section we'll describe SASL only as it works with only GS2 mechanisms.

Authentication proceeds by having a client ("initiator", in GSS terminology) send an initial authentication message ("security context token", in GSS terminology). The server ("acceptor") consumes said token and produces one of three results -success, failure, or "continue needed"-, as well as, possibly, a message to return to the client. The security mechanism may require an arbitrary number of security context tokens be exchanged, always in a synchronous fashion, until ultimate success or failure. Upon success the peers are said to have a fully-established security context, which may then be used to provide security services such as encryption.

In SASL the server may be the one to initiate the authentication message exchange, but, when using GSS mechanisms via the GS2 bridge it will always be the client that initiates the exchange. SASL also requires that the application define an "outcome of authentication message", which is distinct from any such message that the mechanism may provide.

Both frameworks allow mechanisms to provide facilities for application data transport protection -- "security layers", in SASL terminology. SASL's security layers are stream oriented (requiring ordered delivery), while GSS' are message oriented (allowing out-of-order delivery), and thus the GSS-API's security layers facilities are the more general ones. The GSS-API provides two methods of protecting application data: "wrap tokens" and "message integrity check (MIC) tokens". Wrap tokens bear application data within them, while MIC tokens do not. Thus wrap tokens may provide encryption ("confidentiality protection"), while MIC tokens only provide integrity protection. MIC tokens are very similar to HMAC -- readers should think of HMAC output with a header affixed to both, the HMAC output and the input.

The GSS-API also provides a keyed pseudo-random function (PRF) [RFC4401] for keying any application's non-standard security layers, if any.

SASL application protocols almost all have an option to use TLS, therefore SASL's security layers are now eschewed in favor of using TLS (with channel binding -- see below). Not all GSS-API application have an option to use a separate system for transport security, thus GSS applications continue to use the GSS-API's transport security facilities.

1.4. Channel Binding Primer

Channel binding is a method for composing two or more end-to-end security facilities such that one facility is used to show that the end-points of the other are logically the same as those of the first. This allows applications to perform authentication at the application layer while leaving transport protection to a lower layer (e.g., TLS) without compromising security.

There are two key aspects to channel binding: a) "channels" (lower layers) must export "channel bindings data" that are cryptographically bound to the channel, and b) authentication mechanisms must be able to consume channel bindings data to ensure that those channel bindings data are seen to be the same by both end-points of the authentication mechanism.

There exists a specification for TLS channel bindings data: RFC5929 [RFC5929].

Most GSS-API and SASL/GS2 mechanisms support channel binding.

An application that supports a TLS channel for transport protection, and application-layer authentication-layer authentication using the GSS-API or SASL/GS2 can perform channel binding to ensure that the application-layer and TLS-layer end-points are the same -- that there is no unauthorized man-in-the-middle (MITM) below the application layer. (An authorized MITM might be an authorized proxy.) This is quite simple: first establish a TLS connection, then extract its channel bindings data, then initiate GSS or SASL/GS2 authentication using those channel bindings data as a channel binding input -- if authentication succeeds, then the TLS channel is bound into the GSS or SASL/GS2 authentication.

1.5. Glossary

This section is purely INFORMATIVE, being intended to inform readers who are not familiar with SASL and the GSS-API. Implementors should refer to the relevant RFCs.

Application protocol
   The protocol that is at the top of the protocol stack, such as, for example, IMAP, LDAP, WebDAV, etcetera [Add lots of references].

Authentication
   A process by which one or more parties are identify themselves and

   prove (for some value of "prove") their identities to other parties.

Authentication message
In SASL this this refers to an opaque message to be exchanged during authentication and which should carry authentication information, possibly (likely) cryptographic in nature.

Channel
A security facility providing secure, end-to-end transport of application data. For example: TLS.

Channel binding
A method of ensuring that the logical end-points of one secure channel are the same as those of another channel at a lower network layer.

GS2
An adaptation of GSS-API mechanisms to SASL. As SASL originally had such an adaptation, we now term that original adaptation "GS1" and the new adaptation is "GS2". GS2 is significantly simpler than GS1, provides channel binding (whereas GS1 did not), and requires one fewer round-trip for its authentication message exchange than GS1 does. GS2's simplicity stems from replacing a binary header required by the GSS-API with a text header, as well as not requring the use of any per-message tokens.

GSS
Generic Security Services. An abstraction around security mechanisms involving two entities (a client and a server, effectively, though a mechanism is allowed to use trusted third parties).

MIC token
Message Integrity Check. A per-message token providing integrity protection to application data. A MIC token does not carry application data within it. See also per-message tokens.

Outcome of authentication message
SASL requires that applications define, for themselves, a message known as the "outcome of authentication message", which should carry at least a bit of information indicating whether authentication succeeded or failed. This is distinct from any such outcome of authentication messages in security mechanisms (which the GSS-API effectively requires, at least for authentication success) in that it also indicates success of authorization of the authenticated client entity to the requested authorization ID (if any) on the target service.

Per-message tokens

An octet string ("token") emitted, and consumed, by the GSS-API, and bearing or authenticating application data, with cryptographic integrity protection and, optionally, confidentiality protection. There are two types of per-message tokens: MIC tokens, and wrap tokens, only the latter of which bears application data. Per-message tokens may include headers with data, with cryptographic integrity protection and, optionally, confidentiality protection.

SASL

Simple Authenication and Security Layers (SASL) is a framework for authentication and transport security for applications. SASL supports many security mechanisms, including all GSS mechanisms via the "GS2" bridge.

Security mechanism

A security mechanism is a protocol that defines an authentication message (or "security context token") exchange for authenticating one or two principals (a client and a server). A security mechanism may also provide for key exchange and transport security facilities. Examples include [list some].

Security context

A security context is the shared secret session keys and authenticated peer names that results from an authentication message exchange between two parties.

Security context token

An opaque octet string that is to be sent by the application to a peer as part of the act of authentication and security context establishment. See also authentication message.

Wrap token

A wrap token is a per-message token that bears application data, providing integrity protection to it, and possibly confidentiality protection as well. See also per-message tokens.

2. The Protocol

At some point the client application determines that REST-GSS authentication is required. How the client does this is discussed in a sub-section below, but for the purposes of this discussion, the client MUST learn or know a URI that will be used to initiate REST-GSS authentication. Once the client knows that REST-GSS authentication is required the client begins by constructing an initial message as described below, then it POSTs it to the agreed-upon URI.

The server SHOULD respond to initial authentication messages with either an error or a 201 response. If there is no error and there is a response authentication message, it will be returned to the client as the representation of the resource created and named in the 201 response, otherwise, if there is no error then the new resource will have an empty representation. The new resource name shall be the name of the REST-GSS session, known as the 'session URI'.

For security mechanisms that require multiple round-trips then additional messages from the client SHALL be POSTed to the session URI, and any response messages from the server will be returned in 200 results as the new representation of the session resource.

The server generally responds to all POSTs to the REST-GSS login and session URIs with a 201 or a 200 status, respectively. Failure is signalled by the authentication messages described below.

Any GETs of a valid session URI SHALL return a representation of the status of that session, else the response to the GET SHALL be an error.

A DELETE of the session URI logs the session out.

The requests and responses that make up a session are tied to the session via the session URI, which is sent in a header. The requests and responses that make up a session SHOULD be authenticated by a Message Integrity Check (MIC) token taken over inputs such that the request or response is bound to the session.

2.1. Authentication Message Format

The authentication messages from clients to servers SHALL be formed as per SASL's [RFC4422] GSS-API bridge (known as "GS2") [RFC5801], with the initial authentication message prefixed with a text header indicating what options were selected. The reason for this is simple: implementors who lack a GSS-API implementation will find it simpler to implement certain mechanisms if the GS2 framework is used.

The authentication messages from servers to clients SHALL be formed

SASL GS2 authentication messages pre-fixed with a header indicating authentication status. The header consists of a single byte: an ASCII character 'S' (success), 'F' (failure), or 'C' (the server expects more authentication messages from the client), followed by an ASCII newline.

2.1.1. ABNF for Initial Authentication Message Header

As described above, the initial authentication message from the

client to the server must include a small text header described by the following Augmented Backus-Naur Form (ABNF) [RFC5234]:

[Add ABNF for a header consisting of a) the selected SASL/GS2 mechanism name, b) the name of the channel binding type selected, c) the session protection options selected, d) room for extensions. -Nico]

2.2. Authentication State Cookies

REST-GSS application server implementations must build and preserve authentication state via a "GSS security context". Clients must identify such state in the case of security mechanisms that require multiple authentication message round trips. The REST-GSS session URI may suffice for this purpose.

Such state might, for example consist of a timestamp and a partially-established security context handle. Some implementations might serialize partially-established security contexts and store them somewhere, including on the client. The timestamp would be used for expiring old partially-established security contexts. The GSS-API allows for serializing security contexts into something known as a "exported security context token". Some GSS-API implementations allow for exporting partially-established security contexts.

Some servers may benefit from being able to store such authentication state temporarily on the client -- such servers MAY assign, in every authentication response message when the server expects additional authentication messages from the client. Such cookies, if present, MUST be base64-encoded and MUST be set in a REST-GSS-AuthenCookie response field, and the client MUST echo such a cookie, if present, in the next authentication message.

Note that serialization of partially-established security contexts is currently not a standard feature of the GSS-API, but it is available in some implementations. Servers that lack this feature may need to preserve authentication state in the form of an identifier for a process that holds the GSS-API security context, and an opaque security context handle, and then they must route all subsequent authentication messages through that process.

2.3. Target Service Naming

When mutual authentication facilities are available the client SHOULD set the target acceptor (service) name to be a GSS-API name of GSS_C_NT_HOSTBASED_SERVICE, with the hostname portion of the name being the name of the host to which the client is authenticating. The service name SHOULD be set as required by the application, or, if

not specified, then to "HTTP". For example, "HTTP@foo.example".

[It'd be good to explore a form of domain-based service naming without host naming. Thus one could login to a large site without having to login to each of many services hosted by different hosts in the same domain. -Nico]

2.4. Authorization ID Form

The form of the authorization ID, if any is supported, SHALL be specified by the application. Applications that make no use of the authorization ID SHOULD reject authentication attempts requesting any non-empty authorization ID.

Applications that intend to use the SASL authorization ID feature should specify a method of preparing the authorization ID, such as SASLprep [RFC4013].

2.5. Mechanism, Channel Binding Type, and Other Negotiation

The representation returned by a GET of the resource to which initial authentication messages are POSTed MUST be as follows:

o [Add ABNF for a field/value list with fields for:

   * SASL/GS2 mechanism list;

   * supported channel binding type list;

   * an indication of what session security facility the server prefers (cookies or MICs, and if MICs, whether TLS must always be used and, if not, whether the body of requests and responses should be protected by the MICs);

   * an indication of whether replay protection is required by the server, in which case MIC tokens MUST be used, and they MUST be taken over data that includes Request-Date and Request-Nanoseconds header fields.]

Clients that don't know a priori what mechanism, channel binding type, or session protection method to use, MUST GET this resource prior to initiatin authentication.

If a channel binding type list is not advertised by the server then the client SHOULD pick a channel binding type as agreed a priori. Applications must specify any pre-agreed channel binding type selection criteria.

In any case of ambiguity or failure to specify, the client SHOULD pick the tls-server-end-point channel binding type [RFC5929] if a server certificate was used to authenticate the server end-point of the TLS channel, else the client SHOULD pick tls-unique.

2.6. Session Status Representation

The status of a session SHALL be obtained by a GET of the session URI. The status of a session SHALL consist of:

- [Add an ABNF for a field/value list with the following elements:

   * a boolean to indicate whether the session is fully established;

   * a timestamp indicating hard expiration, if any;

   * a relative time specification indicating what the session idle timer, if any, is;

- possibly some items indicating authorization attributes of the client, such as the SASL authorization ID selected or accepted by the server, if any.]

2.7. Session Binding via MIC Tokens

MIC tokens are used to bind HTTP requests and responses to containing sessions. Requests (and their responses) can be bound to more than one session for session combination purposes.

[A word about MIC tokens: they are quite similar to HMAC [RFC2104]. For simple GSS-API mechanisms they might be nothing more than an HMAC, with, perhaps a header affixed to the application data that the MIC is applied to.]

MIC tokens for requests are generated by applying GSS_Get_MIC() to a a minimized form of the request containing only the following items:

- the request start line;

- the Host header field, if any;

- optionally a Request-Date field with the same value form as the 'Date' field (this field MUST be sent in the request as well if present in the MIC input);

- optionally a Request-Nanoseconds field bearing a nanoseconds component of the time at which the request was made, as an unsigned integer in decimal, ASCII respresentation (e.g., 1234567)

(this field MUST be sent in the request as well if present in the MIC input);

- a Channel-Binding field bearing the channel bindings data (base64-encoded) of the channel over which the message is being sent (note: the channel bindings should be prefixed with the channel binding type as described in RFC5056, and prior to base64 encoding)), if there is a channel (this field MUST NOT be included in the request);

- the request body if and only if there is no channel to bind to, else an empty request body.

The request MIC is base64-encoded, prefixed with the session URI (separated by an ASCII semi-colon) and placed in a header field named REST-GSS-Request-MIC. Multiple MICs may be placed in this field, separated by whitespace. [XXX Add ABNF for this! Also, add an indication of what CB type is used in the request MIC token.]

The optional timestamp in the request SHOULD be used for replay detection on the server side. GSS-API per-message token replay detection facilities exist, but an implementation may not make it easier to share a security context's replay state easily across multiple processes or even servers in a cluster.

MIC tokens for responses are generated by applying GSS_Get_MIC() to a a minimized form of the response containing only the following items:

- the request status line;
- the REST-GSS-Request-MIC from the request, with runs of whitespace characters replaced with a single ASCII space.
- the response body if and only if there is no channel to bind to, else an empty response body.

The response MIC is base64-encoded, prefixed with the session URI (separated by an ASCII semi-colon) and placed in a header field named REST-GSS-Response-MIC. Multiple MICs may be placed in this field, separated by whitespace.

These MIC tokens are validated by calling GSS_Verify_MIC() with the same input data as GSS_Get_MIC().

2.8. Alternative Session Binding Options

[Add text describing the use of cookies instead of MIC tokens.]

[Add text describing a method of associating REST-GSS session URIs with TLS session IDs instead of using MIC tokens on every request/response. This is only workable when the client's and server's HTTP/TLS stacks expose enough information to the application.]

2.9. Server Indication of Authentication Requirement

When the server wishes to indicate that the client must authenticate in order to access a given resource, then the server MUST respond to the client's HTTP request with either a redirection to a web page with a 303 redirect to a login page (this in the case of browser applications) or a TBD 4xx error indicating that access requires REST-GSS login and, optionally directing the client to the REST-GSS login URI by listing that URI in a response header field named 'REST-GSS-Authenticate'.

3. Examples

3.1. Server Decides When to Authenticate

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

S->C: HTTP/1.1 303 http://A.example/login.html&<encoded-URI>
```

Authentication required indication browser apps

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

S->C: HTTP/1.1 4xx
      REST-GSS-URI: http://A.example/rest-gss-login
```

Authentication required indication for non-browser apps

3.2. Mechanism and Other Negotiation

```
C->S: HTTP/1.1 GET /rest-gss-login
      Host: A.example

S->C: HTTP/1.1 200
      Content-Type: application/rest-gss-login
      Content-Length: nnn

      mechs: SCRAM-SHA-1-PLUS,GSS-KRB5-PLUS
      channel-binding-types: tls-server-end-point
```

```
          session-binding: MIC-CB, MIC-body
          replay-protection: optional
```

Negotiation

3.3. Login, Session, and Logout, with SCRAM

The following example is shamefully stolen from RFC5802, and adapted to REST-GSS.

```
      C->S: HTTP/1.1 POST /rest-gss-login
            Host: A.example
            Content-Type: application/rest-gss-login
            Content-Length: nnn

            SCRAM-SHA-1,,MIC
            n,,n=user,r=fyko+d2lbbFgONRv9qkxdawL

      S->C: HTTP/1.1 201
            Location http://A.example/rest-gss-session-9d0af5f680d4ff46
            Content-Type: application/rest-gss-login
            Content-Length: nnn

            C
            r=fyko+d2lbbFgONRv9qkxdawL3rfcNHYJY1ZVvWVs7j,
            s=QSXCR+Q6sek8bf92,i=4096

      C->S: HTTP/1.1 POST /rest-gss-session-9d0af5f680d4ff46
            Host: A.example
            Content-Type: application/rest-gss-login
            Content-Length: nnn

            c=biws,r=fyko+d2lbbFgONRv9qkxdawL3rfcNHYJY1ZVvWVs7j,
            p=v0X8v3Bz2T0CJGbJQyF0X+HI4Ts=

      S->C: HTTP/1.1 200
            Content-Type: application/rest-gss-login
            Content-Length: nnn

            A
            v=rmF9pqV8S7suAoZWja4dJRkFsKQ=
```

Authentication message exchange using SCRAM
without channel
binding

```
      C->S: HTTP/1.1 GET /some/doc.html
            Host: A.example
```

```
            REST-GSS-Request-MIC:
               http://A.example/rest-gss-session-9d0af5f680d4ff46
               <base64-encoding of output of GSS_Get_MIC() using the
                named session's security context and taken over a
                minimal version of this request:

                HTTP/1.1 GET /some/doc.html
                Host: A.example

                >

      S->C: HTTP/1.1 200
            Content-Type: text/html
            Content-Length: nnn

            <HTML source of http://A.example/some/doc.html>
```

Example request and response using MIC tokens

```
      C->S: HTTP/1.1 DELETE /rest-...-session-9d0af5f680d4ff46
            Host: A.example
            REST-GSS-Request-MIC:
               http://A.example/rest-gss-session-9d0af5f680d4ff46
               <base64-encoding of output of GSS_Get_MIC() using the
                named session's security context and taken over a
                minimal version of this request:

                HTTP/1.1 DELETE /rest-...-9d0af5f680d4ff46
                Host: A.example

                >

      S->C: HTTP/1.1 200
```

Example of session logout

4. Implementation and Deployment Considerations

It is possible to implement REST-GSS with no changes to HTTP implementations, on the client and server sides both. [Hmmm, maybe we should make sure not to add any new return codes! -Nico]. It is also possible to implement REST-GSS with no changes to TLS implementations, though it is preferable to use TLS implementations that output channel bindings data [RFC5929].

All that is required in order to implement REST-GSS is one or more GSS-API security mechanisms, whether used directly or via an actual GSS-API framework implementation. Note that an implementation of the full GSS-API framework is _not_ required. A minimal implementation of a security mechanism such as SCRAM [RFC5802] is feasible that provides nothing like the API that is the GSS-API.

Similarly, a GS2 [RFC5801] implementation is required, but given how simple GS2 is there's no need for a full-blown SASL [RFC4422] nor GS2 framework implementation.

The largest obstacle for REST-GSS implementation lies in the web browser, in the case of browser-based applications: without a native implementation of REST-GSS in the browser (or the platform, but accessed via the browser), the only way to implement REST-GSS is by implementing a security mechanism JavaScript [XXX Add reference. -Nico]. Implementing security mechanisms in scripts downloaded as needed from the same origin as the page that will use them presents a number of obvious security considerations, but as a technology demonstrator, this approach will work.

As for deployment, the availability of security mechanisms and federations is critical. Work is in progress to produce federatable security mechanisms for the GSS-API. In the meantime, there are security mechanisms such as Kerberos V5 [RFC4121] and others, that make deployment in the enterprise scale, if not the Internet scale, an immediately available option.

4.1. Desired GSS-API Extensions

At least one GSS-API extension is desired, though not required: the ability to export (serialize) partially-established security contexts. It is possible to implement REST-GSS on the server without this feature, but especially for clustered servers using multi-round-trip security mechanisms, it would be much easier to implement where this extension is available.

5. IANA Considerations

This document has IANA considerations: new HTTP fields, and, possibly, new HTTP status codes. These need to be registered. Registration information to-be-added.

6. Security Considerations

The security considerations of HTTP [RFC2616], TLS [RFC5246], the

GSS-API [RFC2743], SASL [RFC4422], and GS2 [RFC5801] apply. When channel binding is used the security considerations of [RFC5056] and [RFC5929] also apply. Some of the security considerations of HTTP and TLS are addressed by the use of mutual authentication and channel binding in REST-GSS.

REST-GSS provide a number of optional facilities, both by itself and because the GSS-API itself provides optional facilities. These facilities can provide excellent security to users and service providers, particularly mutual authentication and channel binding, which together can significantly strengthen the authentication of services otherwise provided only by TLS.

Some GSS-API security mechanisms are not secure against eavesdroppers or active attacks. Therefore REST-GSS applications MUST use TLS with confidentiality protection to protect all REST-GSS authentication message exchanges, and SHOULD require the use of a server certificate [RFC5280] unless mutual authentication and channel binding are being used.

REST-GSS applications SHOULD prefer security mechanisms that provide for mutual authentication to ones that do not, and SHOULD use channel binding to TLS whenever it's available. REST-GSS applications SHOULD NOT, by default, use security mechanisms that do not support mutual authentication or channel binding. REST-GSS applications that allow the use of security mechanisms that do not provide mutual authentication MUST require that the server be authenticated by a server certificate [RFC5280].

REST-GSS applications SHOULD use channel binding to TLS, using the channel binding data of the TLS connection that will carry the client's initial authentication message.

REST-GSS does not provide a confidentiality protection option. Therefore REST-GSS applications MUST use TLS if confidentiality protection is desired.

REST-GSS applications SHOULD use TLS if integrity protection is desired. Where they do not use TLS then they SHOULD use MIC tokens to protect the bodies of the requests and responses, not just the HTTP method and URI.

REST-GSS applications SHOULD use MIC tokens instead of cookies to tie requests to sessions. REST-GSS applications SHOULD use channel binding to TLS for session requests.

REST-GSS applications that are sensitive to replays of requests SHOULD use MIC tokens with Request-Date and Request-Nanoseconds

fields present in the data that the MIC is taken over, unless the server supports tls-unique channel bindings, in which case the application SHOULD NOT include Request-Date and Request-Nanoseconds fields in the MIC data. But servers that have suitable GSS-API per-message token replay detection implementations SHOULD NOT request that Request-Date and Request-Nanoseconds header fields be used.

REST-GSS applications SHOULD use the extended GSS mechanism inquiry API [RFC5587] to help select mechanisms that provide the features required by the application.

While it is convenient to have servers decide when authentication is required on the basis of the URIs being accessed by the client, this can leak information. It is best to require authentication, or not, for an entire site.

...

6.1. User Interface and Scripting Interface Recommendations

User interface (UI) and scripting interfaces are out of scope for this document. However, in the interest of seeding works-in-progress, we describe some such UIs and scripting APIs here, in broad strokes.

For browser-based applications we recommend the addition of an element to the HTML DOM for rendering a "login" button on a web page such that the user may activate it to initiate REST-GSS authentication. Such a DOM element should include a URI to POST initial authentication messages to. For non-browser applications we recommend a similar UI.

For all REST-GSS applications we also recommend non-DOM element by which the client may indicate REST-GSS login status to the user, as well as by which the user may initiate a logout. The status displayed to users of logged-in REST-GSS sessions should include information such as: what security mechanism was used, the authenticated client and server principal names, session protection options, etcetera.

For scripting we recommend extensions to XMLHttpRequest that allows the application to request a REST-GSS session URI as an output, implying that a session will be logged in. We also recommend inputs to XMLHttpRequest to specify what REST-GSS session to use, and/or a REST-GSS login URI. An extension should be provided for inquiring the status of a REST-GSS session.

Cross-site scripting note: browsers MUST apply same-origin-like

constraints to the REST-GSS target service names, if any, specified by scripts downloaded from a site.

6.2. Platform Integration

[Add notes about platform integration. -Nico]

6.3. Anti-Phishing

[Add notes about how mutual authentication via federated security mechanisms may reduce the scope of phishing attacks by effectively adding a service whitelist of sorts. -Nico]

7. References

7.1. Normative References

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

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, January 2000.

[RFC4422] Melnikov, A. and K. Zeilenga, "Simple Authentication and Security Layer (SASL)", RFC 4422, June 2006.

[RFC5056] Williams, N., "On the Use of Channel Bindings to Secure Channels", RFC 5056, November 2007.

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC5587] Williams, N., "Extended Generic Security Service Mechanism Inquiry APIs", RFC 5587, July 2009.

[RFC5801] Josefsson, S. and N. Williams, "Using Generic Security Service Application Program Interface (GSS-API) Mechanisms in Simple Authentication and Security Layer (SASL): The GS2 Mechanism Family", RFC 5801, July 2010.

[RFC5929] Altman, J., Williams, N., and L. Zhu, "Channel Bindings for TLS", RFC 5929, July 2010.

7.2. Informative References

[I-D.ietf-abfab-gss-eap]
Hartman, S. and J. Howlett, "A GSS-API Mechanism for the Extensible Authentication Protocol", draft-ietf-abfab-gss-eap-07 (work in progress), May 2012.

[I-D.ietf-kitten-sasl-openid]
Lear, E., Tschofenig, H., Mauldin, H., and S. Josefsson, "A SASL & GSS-API Mechanism for OpenID", draft-ietf-kitten-sasl-openid-08 (work in progress), February 2012.

[I-D.ietf-kitten-sasl-saml]
Wierenga, K., Lear, E., and S. Josefsson, "A SASL and GSS-API Mechanism for SAML", draft-ietf-kitten-sasl-saml-09 (work in progress), February 2012.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[RFC4013] Zeilenga, K., "SASLprep: Stringprep Profile for User Names and Passwords", RFC 4013, February 2005.

[RFC4121] Zhu, L., Jaganathan, K., and S. Hartman, "The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2", RFC 4121, July 2005.

[RFC4401] Williams, N., "A Pseudo-Random Function (PRF) API Extension for the Generic Security Service Application Program Interface (GSS-API)", RFC 4401, February 2006.

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

[RFC5802] Newman, C., Menon-Sen, A., Melnikov, A., and N. Williams, "Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms", RFC 5802, July 2010.

[RFC5849] Hammer-Lahav, E., "The OAuth 1.0 Protocol", RFC 5849, April 2010.

Author's Address

Nicolas Williams
Cryptonector LLC

Email: nico@cryptonector.com