GSS-REST, a Proposed Method for HTTP Application-Layer Authentication

nico@cryptonector.com

April 27, 2011

Abstract

Applications often require context-specific authentication decisions, particularly HTTP applications. TLS provides limited authentication facilities, and only at the transport-layer, which is often inconvenient. GSS-REST is a method that obtains pluggable application-layer authentication for HTTP-based applications, using off-the-shelf authentication mechanisms and without replacing TLS for transport protection. Additionally, GSS-REST provides a method by which to get away from cookies.

GSS-REST, as its name indicates, consists of POSTing GSS-API "security context tokens" to a resource in order to authenticate the client user and the service and to exchange cryptographic key material, the latter needed only optionally to strengthen cookies.

Additionally, we discus some aspects of identity selection UI issues.

Contents

1 Introduction

Today's HTTP[RFC2616] applications suffer from very limited choices for user authentication. The available user authentication methods are:

- TLS[RFC5246] renegotiation with user certificates. It is safe to say that, on the Internet scale, user certificates have failed. To begin with, user certificates and the private keys to go with them are not very portable. Also, they are difficult for most users to understand, acquire, and use.
- HTTP DIGEST-MD5, a password-based authentication protocol which, for a variety of reasons mostly related to UI limitations of various web browsers, also has not achieved wide deployment. Additionally, DIGEST-MD5 does not federate well in practice, thus leading to users having many DIGEST-MD5 credentials – a password management headache.
- HTTP Basic authentication. This method has the client send the username and password without any transformations, thus providing very little security. Additionally, this has all the same problems that HTTP DIGEST-MD5 has.
- HTTP/Negotiate[RFC4559], a GSS-API/Kerberos V5 based authentication protocol extension to HTTP. Kerberos has not had Internet-scale deployment success, therefore HTTP/Negotiate is mostly in use only in corporate networks.
- HTML form POSTing and similar, plus cookies. This method is by far the most widely-deployed user authentication method on the web.
- SASL[RFC4422], used only by non-web-browser-based HTTP applications, often with SASL mechanisms that are no stronger that HTML form posting or, at best, DIGEST-MD5.

The end result is that users have to manage enormous numbers of {username, password} credentials, often one per-site. And nothing about simple username and password authentication helps users authenticate the services to them any better than TLS does. TLS does a relatively poor job of authenticating services given that the only scalable method of doing so has turned out to be a PKI[RFC5280] with very large trust anchor sets.

Sure, recent efforts to federate authentication based on cookies, HTTP redirects, and private communications between relying parties and identity providers have improved this situation somewhat. But such methods are inherently browser-specific. Not all HTTP applications are browser-based!

We have a pent-up need for authentication methods that work for browser- and non-browser-based applications – authentication methods that federate and which are cryptographically secure.

We also require that no authentication method radically change HTTP, even HTTP/1.0, nor TLS. Moreover, HTTP applications often require that the server

decide when a user must authenticate based on context, such as the resources that the user is trying to access (some may be public, some may be private). HTTPS leaves a lot to be desired in terms of application control over when authentication happens, and how. There's also a large installed based of TLS (e.g., in the form of code and firmware, such as in concentrators) that we ought to make the most of.

Therefore we also require that solutions in this space operate at the application layer, yet also that they not abandon the use of TLS entirely, or at all.

We propose a simple protocol for application-layer authentication based on off-the-shelf components and a composition technique known as "channel binding", used in a RESTful manner.

1.1 GSS-API Primer

The GSS-API[RFC2743] is a very simple, pluggable network authentication protocol, with a simple to use (in the common cases) abstract API, and bindings of that API for multiple programming languages. Readers should not be misled by the number of API elements in the GSS-API; most applications only make use of a small subset of those.

The GSS-API protocol is quite simple: a client (known as an "initiator") sends an "initial security context token" of a chosen GSS security mechanism to a peer (known as an "acceptor"), then the two will exchange, synchronously, as many security context tokens as necessary to complete authentication or fail. The specific number of context tokens exchanged varies by security mechanism. Once authentication is complete, the initiator and the acceptor will share a "security context" that includes shared secret session key material, and they may then exchange "per-message tokens" encrypting and/or authenticating application messages.

The API is somewhat more complex than the protocol. The API elements are:

- Major types/classes: principal name, credentials, security contexts;
- Functions/methods: for creating name objects, acquiring credentials, establishing security contexts, creating and consuming "per-message tokens", and inquring on GSS objects;
- Various constants and miscaellaneous types.
- Two per-message token types: message integrity check tokens ("MIC" tokens), which are akin to MACs and which do not bear, and wrap tokens, which "wrap" application data, providing integrity protection and optional confidentiality protection (i.e., encryption).

GSS security mechanisms exist, or soon will exist, for all of these options, and more may easily be added:

- Kerberos V5[RFC4121] (the most widely known and implemented GSS mechanism)
- SCRAM[RFC5802], a DIGEST-MD5-like mechanism
- GSS-EAP – an EAP+AAA+SAML based mechanism, a work in progress
- SAML, a work in progress
- OAuth, a work in progress
- PKU2U (a PKI-based mechanism), a work in progress (but implemented by at least one vendor)
- mech_dh (a mechanism based on directories of Diffie-Hellman public keys)
- SPNEGO – a pseudo-mechanism for negotiating GSS mechanisms.

Mechanisms based on PAKEs could also be added quite easily. Most of the above have been implemented and deployed by at least one implementor, and at least two are widely implemented. Additionally, there are a number of proprietary, non-standard GSS security mechanisms available as well.

1.2 Channel Binding Primer

"Channel binding"[RFC5056] is a technique for composing security mechanisms. A security mechanism is a protocol that provides some authentication facilities and, possibly, application data protection facilities. This composition technique can be used within a single OSI network layer, or across OSI network layers equally well. A critical feature of channel binding is that it detects man-in-the-middle attacks at the lower layer, to the best of the authentication mechanism's cryptographic ability.

Channel binding is particularly useful for the purpose of "binding" application-layer authentication to lower layer transport security mechanisms (such as TLS or IPsec). As described in the introduction, application-layer authentication is highly desirable in HTTP applications, yet use of TLS for transport protection is also highly desirable – any other alternative would do too much violence to HTTP. The mechanism providing transport security is known as the "secure channel" that one binds authentication to.

In order to bind authentication to a channel, the channel must export some data that securely names that channel for the purpose of channel binding. Such data is known as "channel binding data" or "channel bindings". The authentication mechanism then ensures that this data are seen to be the same on both sides of the channel, that is, on the initiator and acceptor sides. This can be done by exchanging MACs of the channel binding data, for example, or by adding the channel bindings to any key derivation function's inputs, for another example.

Today we have two channel binding data formats defined for general use for naming TLS channels: tls-unique and tls-server-end-point[RFC5929]. The tls-unique channel binding type is based on cryptographic key material from the

TLS connection's handshake, and is therefore unique for each connection, as the channel binding type's name implies. The tls-server-end-point channel binding type is the server's TLS certificate, essentially.

The reader should read RFCs 5056 and 5929 for more information.

2 GSS-REST

The proposed solution is a simple adaption of HTTP as a transport for GSS security context tokens:

1. Initiators PUT their initial security context tokens.
2. The response to the PUT indicates the name of the resource to which subsequent context tokens should be posted, if any, and contains any response tokens from the acceptor.
3. Initiators POST subsequent security context tokens to the resource created by the PUT.
4. If authentication fails, then the resource created by the PUT is either not created or it is deleted.
5. Subsequent HTTP requests and responses may contain headers bearing MIC tokens taken over such data as: resource names, any cookies, and so on – preferably data that the application has ready access to. This MIC token binds together all the requests that make up an application session identified by the security context established in steps 1-3. Some applications may wish to use cookies instead of MIC tokens, which will be secure only if the cookies are secure-only, TLS is in use, and channel binding to TLS was performed.
6. When the client wishes to logout it DELETEs the resource created by the PUT.

In all cases, whenever TLS is used then channel binding will also be used to bind GSS authentication to the TLS channel. Use of GSS-REST without TLS is discouraged as it necessitates protecting more of the requests and responses with MIC tokens (likely an abstraction violation).

GSS-REST has many benefits:

- Look 'ma! No changes to HTTP!
- Look 'ma! All off-the-shelf components!
- Look 'ma! It's RESTful!
- Many security mechanisms from which to pick, some of which are specifically designed with federation in mind.

- Mutual authentication is generally available from most if not all GSS mechanisms (in both common senses of "mutual authentication", of authenticating two entities to each other, and providing confirmation of cryptographic key exchange success). This can strengthen the authentication of the server to the user, thus mitigating the weaknesses of today's TLS PKI.

2.1 JavaScript Bindings for GSS-REST

The most natural way to make use of GSS-REST in browser-based applications would be through an XMLHttpRequest extension for requesting GSS-REST to a service whose name is to be derived from the HTTP request's server's name. This approach has security benefits discussed below, namely that the browser can trivially know that GSS-REST authentication took place.

A more verbose (code-wise) but more generic method would be to add only native object interfaces to the GSS-API, then using XMLHttpRequest as it exists today, with the script implementing GSS-REST directly. This approach has the drawback that the browser must implement a heuristic (a reliable heuristic, fortunately) to detect that GSS-REST has taken place, so it can indicate the GSS-REST authentication state to the user. The heuristic in question would be simple: a script on a page used GSS_Init_sec_context() to successfully authenticate with the page's origin and with channel bindings input that matches those of the underlying TLS connection.

2.1.1 Accessing channel binding data from scripts

Some browsers provide XMLHttpRequest extensions for accessing information such as the server's TLS certificate. This is sufficient for implementing tls-server-end-point channel bindings. However, it would be preferable to see XMLHttpRequest extensions for accessing a TLS channel's channel bindings explicitly, and with support for tls-unique bindinds, not just tls-server-end-point. Note too that the channel binding data must be obtained for a TLS connection to be used for an as-yet not issued HTTP request.

2.1.2 JavaScript bindings for the GSS-API

The most natural way to model a JavaScript bindings of the GSS-API may be to follow the model of the Java bindings[RFC5653] for the GSS-API.

2.1.3 User interface (UI) considerations and identity selection

By having a JavaScript interface to the GSS-API and to HTTPS (XMLHttpRequest), web site developers will be able to manage some aspects of authentication UIs. Web site developers will not be able to specify the look and feel of UIs for *initial* credential acquisition, but they *may* be able to specify the look and feel of UIs for selection of specific credentials (i.e., identity selection).

Web site developers will definitely be able to decide when to authenticate.

There are some challenges here regarding identity selection UIs and presentation of security state to the user.

UI and JavaScript elements will be needed to facilitate identity selection. One possibility would be to have an HTML element by which to trigger (when the user clicks on it) GSS credential selection and GSS-REST authentication. Another possibility would be to have a JavaScript object to accomplish the same. The credential (identity) selection UI should be provided by the browser chrome, but might take some clues as input from the page or script, such as, for example, information that the browser could use to prune the set of credentials from which the user should pick. The browser should also remember the user's choice of credential as the default for a given site, thus making the user's choice simpler the next time.

The biggest challenge lies in how the browser chrome might inform the user that they and the server are mutually authenticated. The common thread in all the HTTPS requests made to load a page and by its scripts will be the TLS session between the client and the server. If GSS-REST was performed by the browser directly, rather than by a script indirectly (see section 2.1), then the browser can and must know that GSS-REST authentication took place, and therefore can add a browser chrome UI element to indicate this, and to let the user inspect the mechanism used, and the authenticated initiator acceptor names. Note that multiple GSS-REST authentications might take place for one TLS session.

2.1.4 Possible local policies for identity selection

Browser chrome implementors may want to implement some or all of the following sorts of local policies for identity selection:

- Remember the user's last manual identity selection for this site.
- Offer the script access to any credentials labeled "public".
- Offer the script access to any credentials that the user has ever selected for the script's or page's origin.
- Offer the script access to any credentials for which there is a trust path to a service that the script wants to authenticate to.
- Offer the script access to any credentials where the domain/realm of the user is the same as that of the server that the script wishes to authenticate to.

2.2 Non-Broswer HTTP Applications

GSS-REST is, by its RESTful nature, equally usable by browser- and non-browser-based applications. UIs and non-JavaScript APIs for non-browser-based applications are not covered here, but the reader surely can imagine what such UIs and APIs might look like.

2.3 Server-Side Implementation of GSS-REST

Because HTTP is not modified, any application should be able to implement GSS-REST entirely above the HTTP stack, at least when using tls-server-end-point channel bindings. The only desirable modification to the HTTP stack would be to natively export TLS channel bindings to the application (so that tls-unique channel bindings may also be used). The application does, of course, need access to a GSS-API implementation.

3 Security Benefits of GSS-REST

There are three major security benefits that are provided by GSS-REST:

1. Various security mechanisms, most if not all of which are cryptographically stronger than DIGEST-MD5.
2. Federation, which should result in fewer credentials for users to manage.
3. Stronger authentication of services to users, not just of users to servers. This helps mitigate the weaknesses of the TLS PKI. This is a result of the channel binding operation's semantics, at least as long as the authentication mechanism uses credentials other than the server's TLS certificate to authenticate the server.

Other benefits of GSS-REST include:

- Application-layer authentication all the while...
- ...TLS is used for transport security.
- The GSS-API is pluggable. We do this once and we never have to do it again.
- Works with browser- and non-browser-based applications.
- HTTP is not modified. This is particularly noticeable on the server side, where applications should be able to implement GSS-REST with no modifications to the HTTP substrate (except, at most, to expose the channel bindings of the underlying TLS session).

4 Security Considerations

All the security considerations RFCs 5246, 5056, 5929, and of the GSS mechanisms that one might use, apply. Additional security considerations relate to JavaScript interfaces and UIs:

- JavaScript interfaces to the GSS-API and/or GSS-REST MUST NOT allow a script to use GSS credentials that the user does not allow the page to use explicitly or via local user policy.

- JavaScript interfaces to the GSS-API and/or GSS-REST MUST NOT allow a script to authenticate to services other than the HTTP server that the script is trying to talk to, not without explicit user approval or local user policy that allows it.
- For privacy reasons, JavaScript interfaces to the GSS-API and/or GSS-REST MUST NOT allow a script to observe all the user's credentials' names without the user's approval.
- UI elements MUST be added to browsers by which to indicate GSS-REST status to the user. Note that some elements on a page may have been obtained from other servers than the page's origin – the same security considerations apply in that case as do without GSS-REST.

The W3C may want to add an HTML element by which to trigger GSS-REST authentication without having to use scripts. Such an element should allow the page to specify some constraints on credential selection, such as a list of mechanisms supported by the server.

References

[RFC2616] R. Fielding et. al., Hypertext Transfer Protocol – HTTP/1.1

[RFC2743] J. Linn, Generic Security Service Application Program Interface Version 2, Update 1

[RFC2744] J. Wray, Generic Security Service API Version 2 : C-bindings

[RFC5056] N. Williams, On the Use of Channel Bindings to Secure Channels

[RFC5929] N. Williams, Channel Bindings for TLS

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

[RFC5802] C. Newman et. al., Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms

[RFC5653] M. Upadhyay and S. Malkani, Generic Security Service API Version 2: Java Bindings Update

[RFC4178] L. Zhu et. al., The Simple and Protected Generic Security Service Application Program Interface (GSS-API) Negotiation Mechanism

[RFC4559] K. Jaganathan et. al., SPNEGO-based Kerberos and NTLM HTTP Authentication in Microsoft Windows

[RFC4422] A. Melnikov and K. Zeilenga, Simple Authentication and Security Layer (SASL)

[RFC5280] D. Cooper et. al., Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile

[RFC4121] L. Zhu et. al., The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2