REST-GSS – Authentication for HTTP Apps, Browser and Otherwise

Nicolas Williams
nico@cryptonector.com
Cryptonector, LLC

HTTP Authentication Challenges

- Use multiple, existing authentication infrastructures, even concurrently on a site
  - Preserve existing investments, provide SSO
- Scale to Internet scale
  - Federation, authorization
- Define and protect sessions
  - Tie requests/responses to sessions, logout, alternative to cookies
- Browser and non-browser HTTP apps
  - User interfaces – Browser chrome, OS integration!
- Various threat models, including *Internet* model

Constraints on HTTP Authentication Solutions

- Must improve security
- Minimal or no modifications to existing software and *hardware* stacks
  - Clusters, load balancers, TLS concentrators, content delivery networks, proxies, etc.
  - Changing HTTP or TLS protocols is a problem
- Apps must control authentication context
  - Decide when to authenticate, and opt. with what mech
  - Find out what happened
  - UI customization is desired

REST-GSS

- Pluggable app-layer authentication
  - Passwords (not plain), Kerberos, PKI, EAP/AAA, SAML, OpenID, OAuth, etc.
- Entirely above HTTP
  - Authentication message exchanges via POST
  - Logout via DELETE
  - Works with HTTP/1.0 and 1.1, w/ and w/o pipelining, w/ and w/o proxies
- Sessions bound via “message integrity check (MIC) tokens” (think HMAC) in HTTP headers
  - Similar to draft-hammer-oauth-v2-mac-token
  - But can still use cookies where web developers insist
- Compare to Microsoft's IWA

UI & API Elements

UI

- DOM element for REST-GSS login
  - Customization options
- DOM (or browser) UI element for logout
- Browser element for inquiring status (target name, mech used, …)
  - Like lock icon, but better

API

- XMLHttpRequest extensions
  - Request REST-GSS login/logout
  - Inquire status
- XMLHttpRequest options?
  - Like DOM element options
    - Set GSS target name (though mostly with same origin restriction)
    - Set GSS mech(s)
    - Pick initiator credential [from creds that script is allowed to choose from]
- node.js

The power of abstraction: pages/scripts need not know too many specifics of how authentication is done, much less implement it.

Why REST-GSS? Alternatives?

Pro

- Off-the-shelf GSS-API implementations
  - MIT, Heimdal, Shishi, Windows, etc.
  - Lots of mechanisms now
- OS Integration!
- Pluggable
  - Need a new mech? Add it!
- Standard, simple
- Auth. at the *right* layer
- Strengthens server auth when chosen mech does target auth!
  - TLS server auth is kinda weak

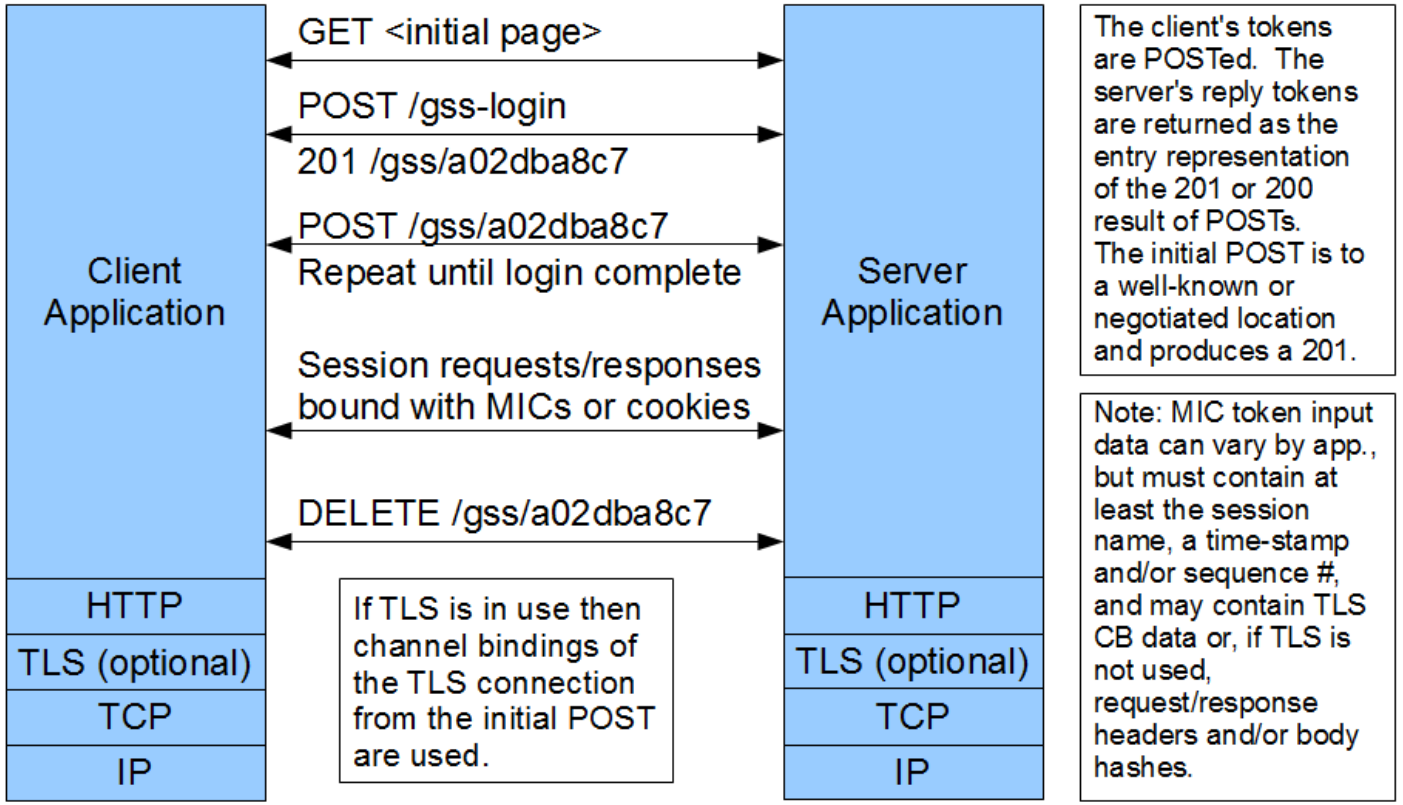
Alternatives?

- Same thing, but with SASL?
  - But GSS mechs are SASL mechs!
- Roll something new
  - Pluggable? If not, whose needs to go not served? If yes, why a new framework?
- Leverage TLS better?
  - SACRED, perhaps?
- Push new auth options to TLS or HTTP?
- draft-hammer-oauth-v2-mac-token?
  - In a way, oauth-mac ***can*** be a REST-GSS method!
- IWA?

GSS-API Primer

- GSS authenticates *principals...*
  - ...using *credentials...*
  - ...thus establishing *security contexts...*
    - “Context token exchange”
  - ...which can be used to:
    - Protect sessions, key other session protection facilities
    - Convey authorization metadata
- Channel binding: binding authentication to a lower-layer such as TLS eliminates MITMs and the need for multiple layers of session protection

REST-GSS Message Flow



REST-GSS – Authentication for HTTP Apps, Browser and Otherwise

Nicolas Williams
nico@cryptonector.com
Cryptonector, LLC

The name “GSS-REST” seemed to suggest to some people that this is a GSS mechanism that uses “REST” internally. So I changed it to REST-GSS.

A good name for this would denote that GSS is used in HTTP apps in a RESTful manner.

REST-GAuth would mean “generic authentication used RESTfully”.

HTTP Authentication Challenges

- Use multiple, existing authentication infrastructures, even concurrently on a site
  - Preserve existing investments, provide SSO
- Scale to Internet scale
  - Federation, authorization
- Define and protect sessions
  - Tie requests/responses to sessions, logout, alternative to cookies
- Browser and non-browser HTTP apps
  - User interfaces – Browser chrome, OS integration!
- Various threat models, including *Internet* model

The Internet threat model assumes the bad guys have full control of the network, but not of the hosts. This is the most we can do from a protocol perspective, as there's not much we can do to assure end-point local security (well, there's NEA w/ TPMs, but even that's not a silver bullet for local security).

In some cases we can assume a less severe network environment and use that assumption to gain some performance.

A good solution will support a range of threat models.

Constraints on HTTP Authentication Solutions

- Must improve security
- Minimal or no modifications to existing software and *hardware* stacks
  - Clusters, load balancers, TLS concentrators, content delivery networks, proxies, etc.
  - Changing HTTP or TLS protocols is a problem
- Apps must control authentication context
  - Decide when to authenticate, and opt. with what mech
  - Find out what happened
  - UI customization is desired

There's a difference between changing HTTP or TLS on the one hand, and changing HTTP stacks to provide something like REST-GSS integration. The former generally *requires* changes to the stacks, while the latter shouldn't be necessary except as a *convenience*.

REST-GSS

- Pluggable app-layer authentication
  - Passwords (not plain), Kerberos, PKI, EAP/AAA, SAML, OpenID, OAuth, etc.
- Entirely above HTTP
  - Authentication message exchanges via POST
  - Logout via DELETE
  - Works with HTTP/1.0 and 1.1, w/ and w/o pipelining, w/ and w/o proxies
- Sessions bound via “message integrity check (MIC) tokens” (think HMAC) in HTTP headers
  - Similar to draft-hammer-oauth-v2-mac-token
  - But can still use cookies where web developers insist
- Compare to Microsoft's IWA

IWA apparently requires HTTP/1.1, doesn't work with proxies, and requires that the same request be tried multiple times until the authentication exchange (carried in headers) completes. This is not RESTful.

However, IWA exemplifies what REST-GSS is aiming for: integration. In IWA scripts don't need to know how users are authenticated, much less must they know users' passwords – this is critical to improve web security! We must stop training users to type in passwords in random password-prompt-ish places.

So REST-GSS is actually quite similar to IWA, in some respects.

I'm leaving a lot of protocol details out, such as what else might be sent besides GSS context tokens – this is a lightning presentation, after all.

UI & API Elements

UI

- DOM element for REST-GSS login
  - Customization options
- DOM (or browser) UI element for logout
- Browser element for inquiring status (target name, mech used, …)
  - Like lock icon, but better

API

- XMLHttpRequest extensions
  - Request REST-GSS login/logout
  - Inquire status
- XMLHttpRequest options?
  - Like DOM element options
    - Set GSS target name (though mostly with same origin restriction)
    - Set GSS mech(s)
    - Pick initiator credential [from creds that script is allowed to choose from]
- node.js

The power of abstraction: pages/scripts need not know too many specifics of how authentication is done, much less implement it.

The API discussion shows the value of integrating REST-GSS into the HTTP stack, but this is not strictly required. It should be possible to build REST-GSS out of a vanilla HTTP stack and a vanilla GSS-API stack. But integration is so much more convenient.

Integration into the HTTP stack means that the whole GSS-API need not be exported to the application, which in turn simplifies local security (scripts should not be allowed to use GSS credentials with no constraints; think of same origin constraints).

Integration into the HTTP stack also means that the browser can provide identity management services using context-specific information to make better suggestions or choices for the user.

Privacy: script can't see all the user's IDs, and mechanisms like Project Moonshot's GSS-EAP enable federations to limit what names and attributes services see.

Why REST-GSS? Alternatives?

Pro

- Off-the-shelf GSS-API implementations
  - MIT, Heimdal, Shishi, Windows, etc.
  - Lots of mechanisms now
- OS Integration!
- Pluggable
  - Need a new mech? Add it!
- Standard, simple
- Auth. at the *right* layer
- Strengthens server auth when chosen mech does target auth!
  - TLS server auth is kinda weak

Alternatives?

- Same thing, but with SASL?
  - But GSS mechs are SASL mechs!
- Roll something new
  - Pluggable? If not, whose needs to go not served? If yes, why a new framework?
- Leverage TLS better?
  - SACRED, perhaps?
- Push new auth options to TLS or HTTP?
- draft-hammer-oauth-v2-mac-token?
  - In a way, oauth-mac ***can*** be a REST-GSS method!
- IWA?

draft-hammer-oauth-v2-mac explicitly aims to protect only against passive attackers, so it doesn't meet the Internet threat model. REST-GSS could trivially provide an option to only meet the oauth-v2-mac threat model. Threat model flexibility is likely a plus. Support for the Internet threat model is a huge plus (yes, even though nowadays we can't really assume local security).

IWA has issues: requires HTTP/1.1, incompatible with proxies.

Pushing authentication down the stack has issues: wrong layer (TLS re-nego bug, anyone?) means abstraction violations, requires changing lots of implementations everywhere.

Regarding frameworks... most pluggable auth frameworks that we have (SASL, EAP, GSS, etc.) started out as retrofitting of individual mechs into apps, then later people realized that they could refactor much code into frameworks – why repeat history when we know the ending already?

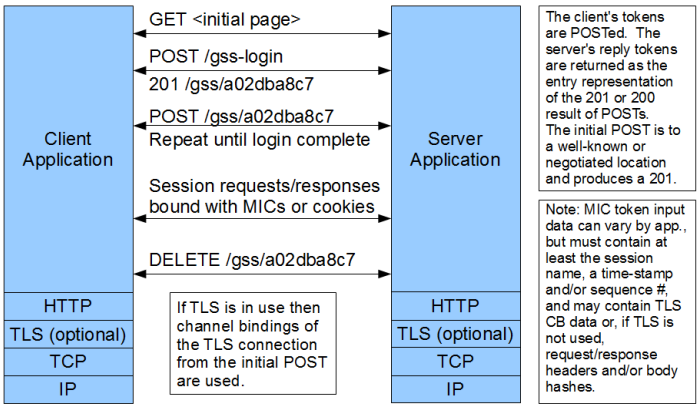
GSS-API Primer

- GSS authenticates *principals...*
  - ...using *credentials...*
  - ...thus establishing *security contexts...*
    - "Context token exchange"
  - ...which can be used to:
    - Protect sessions, key other session protection facilities
    - Convey authorization metadata
- Channel binding: binding authentication to a lower-layer such as TLS eliminates MITMs and the need for multiple layers of session protection

References:

- GSS-API, base spec: RFC2743
- GSS-API, extensions: RFCs 4401, 4768, 5178, 5554, 5587, 5588, 5896
- GSS-API C bindings: RFC2744
- GSS-API Java bindings: RFC5653
- GSS-API mechanisms: RFCs 4121 (krb5), 4178 (SPNEGO), 5802 (SCRAM), and works in progress in IETF KITTEN and ABFAB WGs.
- Channel binding: RFC 5056 and 5929
- SASL: RFCs 4422 (base spec), 5801 (SASL/GSS bridge)

REST-GSS Message Flow



The app decides when to authenticate. Either the client does (because it knows a priori that it should) or the server tells the client to (e.g., via a redirect to a page where the user gets the REST-GSS login button, or a script that starts REST-GSS).

The initial context token (+ extra bits) is POSTED to either a well-known resource or to one selected by the page/script. A 201 indicates that the context has been created and includes any reply token. Thereafter all other context tokens are POSTED to the session resource until authentication succeeds or fails. No extra round-trips are needed.

This works with tls-unique and tls-server-end-point channel binding data types both. The latter works best with existing TLS concentrators. Channel binding is not required, but is recommended, especially when TLS is desired.

Requests can be tied to sessions via cookies or, better, MIC tokens that bind some or all of the headers and/or body to the security context. When MIC tokens are used, different levels of protection can be provided depending on whether TLS is used and the application's needs, thus being flexible w.r.t. threat models.

On logout the session resource is DELETED.