RESTauth

- RESTful auth roughly means “at the app layer”
- But it can be orthogonal to the app
  - One part of an app can be written in FCGI, another in WSGI, and so on
  - HTTP server == router

Why RESTful

- To make it easy to factor authentication code out of the application code to ease deployment
  - Your app (server-side) needs an authorization context (authenticated ID, whatever); why deal with crypto protocol details?
  - Your app (client-side) just needs to invoke the right API and let the credential manager do the rest
- This also allows authen. to be ***pluggable***

Why RESTful

- To gain more control over the UI than the HTTP or TLS stacks would give you...
- ...Without sacrificing security:
  - The raw credentials, are not available to the app
  - The UA ↔ IdP interactions are not exposed to the app

Why RESTful

- Sending login tokens POST bodies → clear demarcation of pre-/post-login session
- Sending login tokens in HTTP headers → might send sensitive info before authentication completes
  - **But**, once we have a session, it's nice to be able to use HTTP headers for tying requests to sessions

Why, again

- Refactor auth out → ease deployment
  - Pluggable → use what auth infrastructure you have

    preserve your infra investment – huge win in enterprise
- UI control
  - The reason we still have passwords in web forms
  - Bridging this in HTTP/TLS is hard

What's wrong with not RESTful

- UI issues
- Deployment issues
  - HTTP server or app have more to do to, leading to limited method selection getting baked in
  - Ditto on the client
  - Proxies. Did I mention proxies?
- This applies whether auth is in HTTP or TLS layers

Examples?

- BrowserID / Persona
  - Login token POSTed to server's login URI
- OAuth also has a profile where login tokens are POSTed

Let's talk about security for a sec

- Persona, OAuth, … – these generally are 0.5 round trip (rt) authentication protocols
  - One login token, from the UA to the RP
  - There may be more round trips UA ↔ IdP, but it's 0.5 rt UA ↔ RP
- .5 rt → absolute dependence on HTTP**S**, on TLS, with confidentiality protection
  - Else replays...
  - a.k.a., bearer tokens

Moar security

- What if we had 1.0 rt?
  - We could then do mutual authentication (depending on the authentication mechanism, or by composition)...
    - Think of Kerberos
  - ...and channel binding to...
  - ...reinforce the TLS server PKI...
  - ...at no extra cost in round trips
    - The first token was going to get ACKed by the app anyways, might as well have the ACK include a reply token
  - Replay issues remain (replay caching is hard)

Moar security!

- What if we had 1.5 rt?
  - “Unacceptable” comes to mind
  - no replay issues though!
  - and, of course, we still get mutual auth and CB

But teh **perf**!

- Replay caching as in Kerberos → total drag
- Probabilistic rcaching w/o durability →
  .5 rt optimization on 1.5 rt in the *common* case!
- **Awesome**
  - No sync writes (fsync()) needed, so it's fast
  - Probabilistic: moar fast, max size bound
    - there's a slow path to fall back on, after all
- No longer unacceptable

Moar security, again

- Pluggable → 1-1.5rt auth methods deployable
  - a way out of bearer token land
- Explicit sessions → session state can be checked
  - explicit logout
- If you really want you can exchange keys for app-layer crypto too

Clusters and proxies

- RESTful → proxy-friendly
- Session state as resources with URIs → cluster support
  - and cross-origin session sharing

So RESTauth is...

- A RESTful pattern and framework for authen. for HTTP apps supporting
  - Proxies
  - Clustering
  - Arbitrary auth methods (pluggable)
  - Strengthening of TLS server authentication
    - If your auth method can authenticate servers
  - Possibly non-TLS session cryptop option as well

RESTauth framework

- The framework part is for **session** binding
  - Where the UA shows it knows the session keys for some session it wants to use
  - Several new headers for login-time negotiations
  - Header for carrying session ID (URI)
  - Header for carrying MAC of TLS CB using session keys

Parting thoughts

- Single-sign-on means
  - Login once [in a while]...
  - ...works for ***all*** your apps
  - Either there's one auth method universally implemented in all apps
    or
    apps are pluggable
  - Guess which of those two isn't happening
    - Take your time if you must

Parting thoughts

- End-to-end session crypto at the lowest possible layer is good because:
  - Best opportunity for optimizations
  - If you multiplex traffic then you get fewer crypto contexts, less L1/L2 cache thrashing, …
- Authentication at the highest layer possible is good because:
  - Best UI and timing control, pluggability, …
- Channel binding bridges the gap

Parting thoughts

- Fast **clustered** replay caching can be had for authentication:
  - Just salt the ~~authenticator~~ token construction with a cluster member ID (e.g., IP address)
  - If you get that wrong you just fall into the slow (1.5rt) path
  - But this doesn't work for session binding

Parting thoughts

- Client-sends-first, multi-round-trip auth has a generic abstract “API”: GSS-API
  - Luke Howard has a BrowserID-as-GSS mechanism implementation, including 1 and 1.5 rt options
    - When I say GSS, read “SSPI” if it helps (or not, if it doesn't)