Logical Architecture - Secure Token Authentication to the Platform

Problem Statement

Web and mobile clients cannot generally do two way SSL mutual authentication for service calls. The installing client certificates in browsers and mobile phone applications simply isn't a very practical option for authentication with the service platform.

Existing large scale service platforms (Twitter, Google, Facebook, Netflix) tend to do Secure Token based authentication and try to mimic many of the strong security properties of of using certificates without having to use certificates.

Requirements:

Must Have:

- The transport channel for the token needs to be secure
- The token itself needs to be secured in transport
- The token at rest needs to be secured
- The token exchange protocol itself needs to assure these things
- The client is authenticated before giving the token
- The there is non-repudiation of the client
- The clients token can be validated
- The client's token can be invalidated
- The server needs some assurances that the token has not been modified by the client.

- support for light weight clients (browser and mobile)
- remove SSL two way mutual authentication

Nice to have

- Limit complexity to client configuration to less than SSL two way mutual authentication

Not to preclude

- Using standards based authentication where possible

Scope

This Authentication pattern covers all Service Gateway clients and would eventually replace the current Authentication Model based on two way mutual SSL.

A Custom Token based solution

A REST based API for managing tokens and secrets.

This API is loosely based on OAuth 2 legged authentication to get the APOLLOAPPTOKEN and leverages the current logic and code for generating an Apollo Assertions. So this design is an additive evolution of the current token based SSO Authentication using APOLLOASSERTION.

- Applications registers with the SSO server and gets an Application Key (AppKey) applications or SSO can use this AppKey to sign AuthTokens.
- In this ecosystem the application acts as a broker for tokens and the client holds those tokens.

- Introducing a new concept of the AppToken that is essentially a new Apollo Assertion with
- ID = AppKey - the application id
- TTL
- MAXTTL
- ID_TYPE= AppKey
- This new Token is optional if an APOLLOASSERTION is currently present
- Header/Cookie/Token Name == APOLLOAPPTOKEN

Pass the Following Tokens/Headers	Application Context	Application + User Context	User Context
APOLLOASSERTION		X	X
APOLLOAPPTOKEN	X	X	
tenantid	X		

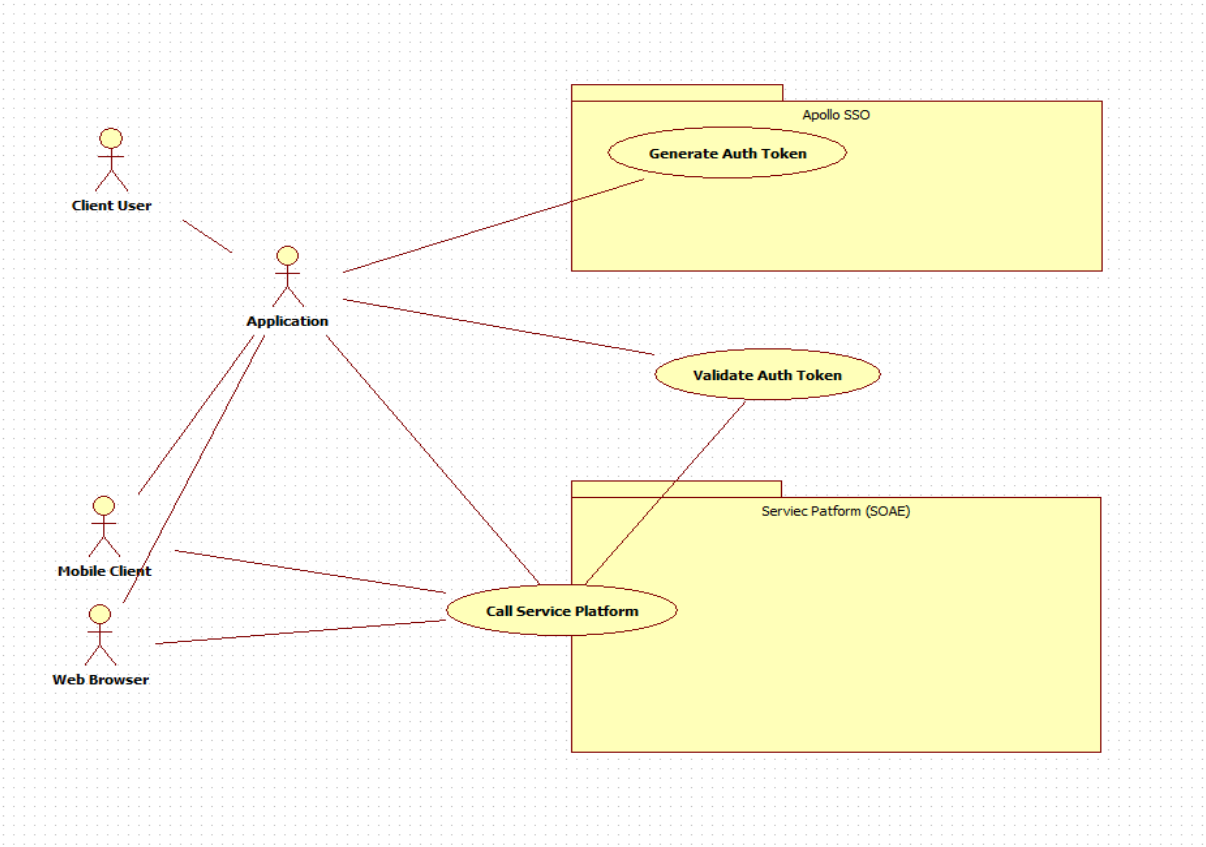
In the Application Context TenantID header is honored and this is the tenantId used for the call.

In the Application + User Context the tenant id is pulled form the APOLLOASSERTION and tenantID header is ignored.

In the User Context the tenant id is pulled from the APOLLOASSERTION.

Drawbacks of Token based Authentication:

- Tokens once in the browser are not application specific. The Client then making the call is no longer the Application but the browser or mobile client itself. So Application Context no longer exists once the client is the Browser or Mobile Client. So for these light weight clients only the APOLLOASSERTION for the User Context will exists and there is no application context.



Applications Calling in Application Context using Token Based Authn

1. The application registers their application and gets an Application Key (AppKey) plus a Secret Key. This is a one time registration and is done out of band for the authentication protocol exchange.

a. The Secret Key is used to sign requests to the Apollo SSO server for AppKeys and is never revealed during the protocol exchange.

b. This registration step will be authenticated using http basic auth to a web page in Keyserver

c. Returns:
[AppKey: "applicationname", Secret:"signing secret", MAXTTL:"2013-12-01 12:00:00"]

d. Note Secrets have a Max Time to Live and Applications will need to rotate them by re-registering.

2. The application requests an APOLLOAPPTOKEN from Apollo SSO using the AppKey plus a HMAC-SHA256 hash of the AppKey + UTC hour:minute using the Secret Key passed in the original registration as their credentials.

a. Appkey is simply the name the application registered.

b. Secret key is used by Applications to encrypt the contents while requesting the APOLLOAPPTOKEN from the SSO server by passing the following information.

i. (APPKEY: "Application name"
, Signature: "HMAC-SHA256(APPKEY+Time based Salt)")

ii. Signature is HMAC-SHA256 hash of the [Appkey + UTC hour:minute].

iii. SSO Server should check the hash using the same secret key for +1/0/-1 minute of the current time.

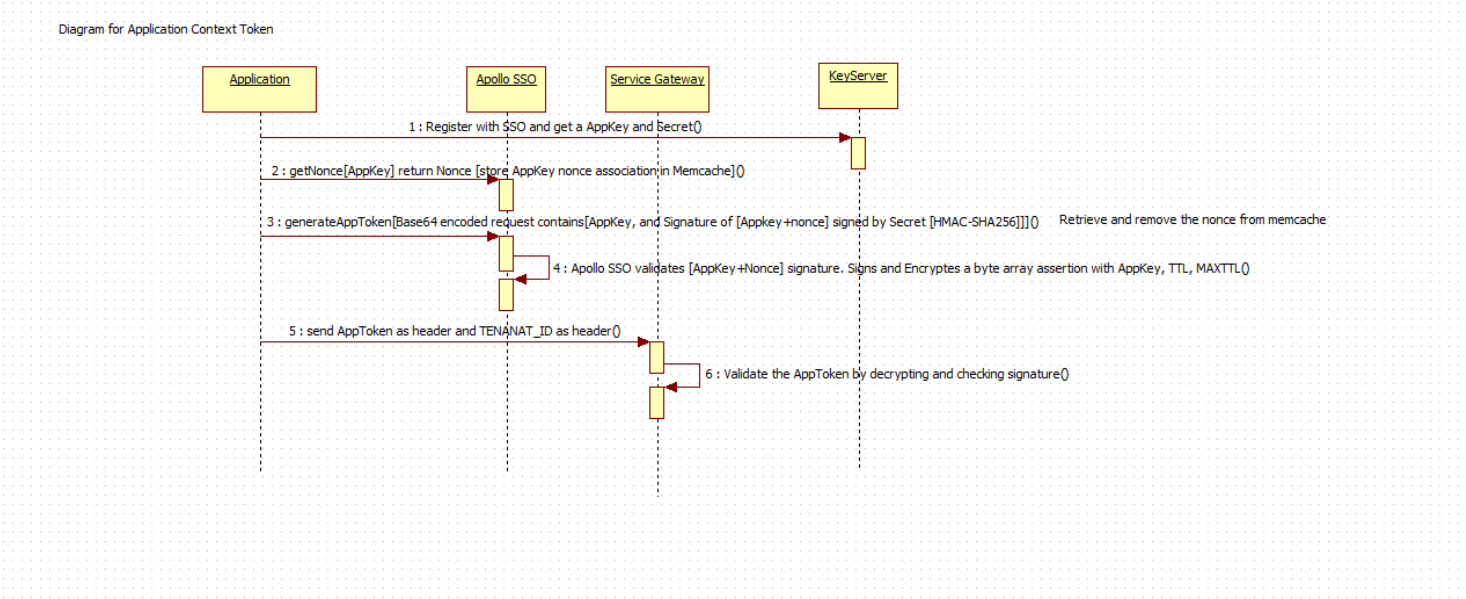
3. The AppToken then can be used by the application to make calls to the Service Platform by passing APOLLOAPPTOKEN as a header of the same name.

a. Tenant context is still set using the **tenantid** header

b. Apollo SSO is validating the APOLLOAPPTOKEN just as it does the APOLLOASSERTION by decrypting the token using SSO's private key and validating signature.

4. Optionally if a User's APOLLOASSERTION is also present this header can also be passed along.

a. In this case the tenantid in the APOLLOASSERTION takes precedence and **tenantid** header is ignored.

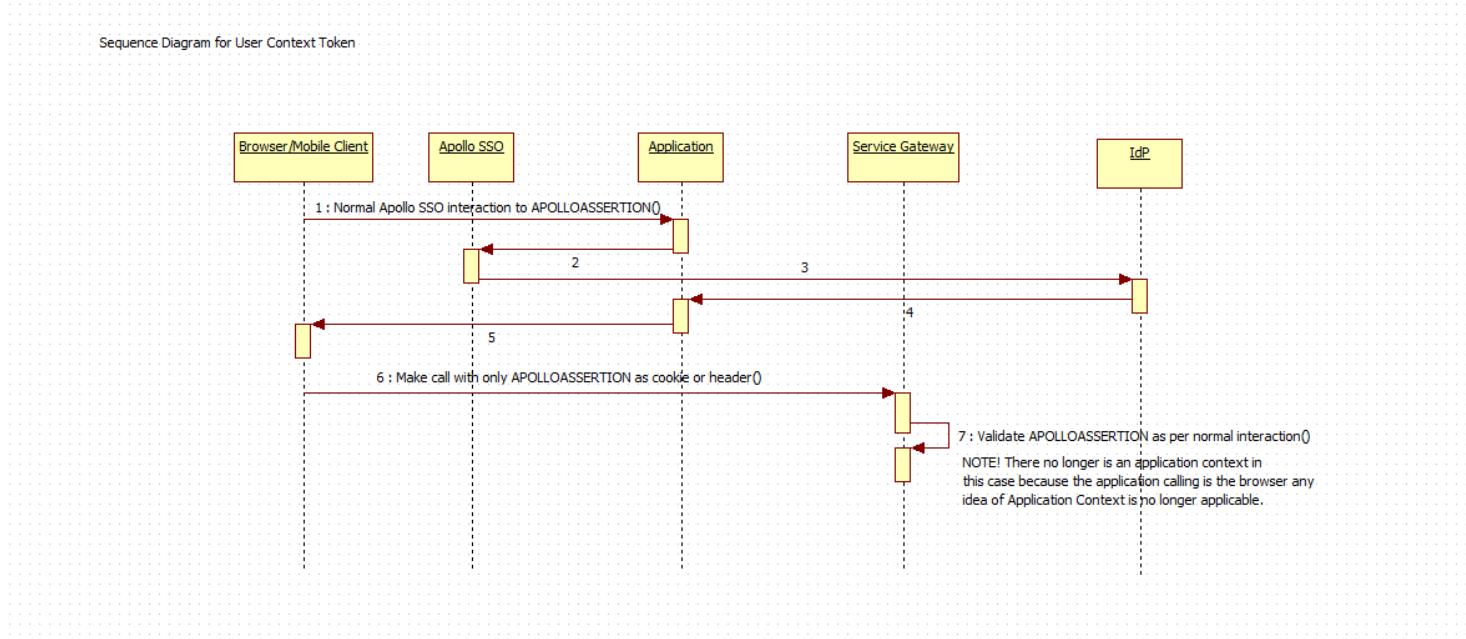


Making User Context calls from Mobile Devices and Web-browsers

1. The application should already have established a APOLLOASSERTION in the client.
2. Calls to the platform can then just be made directly using the APOLLOASSERTION set as a header or a cookie.

a. Cookie will be read first and header second if no cookie is found.

b. **Note you must call the service gateway in the proper domain matching the cookie if you choose to pass the APOLLOASSERTION as a cookie.** Otherwise the cookie will not be visible to service gateway and the call will fail.



Chosen Solution

Choosing Custom Token Solution

Reasoning:

Researching OAuth and OpenID they both don't directly address how you generate the secure session and continue to make authentication and authorized calls to the Relying Party application. Any implementation here would simply be using OAuth or OpenID as the authenticaiton mechanism but OAuth and OpenID would not address the core of the problem trying to be solved. That web-browser and mobile clients need to make authenticated calls to the Service Platform. The token and session management. would have to be a custom solution so adding OAuth or OpenID would just be adding overhead with out adding functionality.

And neither standard actually addresses using browsers or mobile devices as the Relying Parting directly. The interaction using these standards is always a Browser talking to an Application and the Application is the Relying Party. The authentication session between the Browser and the Application is left up to the Application.

So given the two prevailing standards looked at don't actually cover the core use authentication use cases that are being targeted for bearer token authorization we will pass on using them for now. This does not preclude using such standards in the future if client support for mobile and web browsers clients enters these standards.

Building Blocks

- Apollo SSO as the token generator
- Apollo KeyServer as the registration repository for the Application Name, Key and Secret
- Future plans to integrate the new Service Registry into Application Registration
- CAS to authenticate users registering Applications to the Apollo SSO ecosystem.

Risks

- Lightweight clients will no longer have an Application Context
- Bearer tokens are transferable, we are relying on the a secure token exchange and the transient nature of the Assertions as the primarily as the security mechanisms.

Assumptions

- We will completely remove two way mutual authentication even for the stand alone endpoints.

Addendum

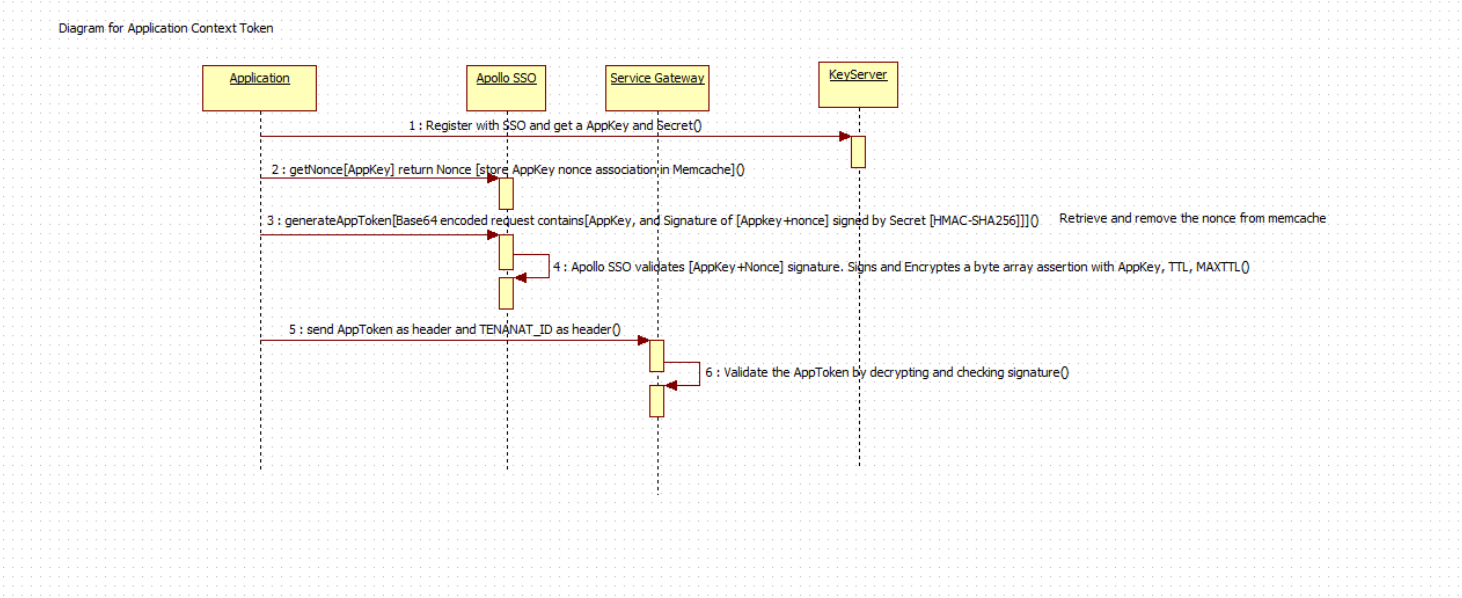
Rather that registering for a Appkey and Secret from Apollo SSO the endpoint will now be KeyServer.

The development team prefers a non-timebased approach for salting so we will instead use a NONCE.

- Registration of the Application occurs in the KeySever. Authorization to the webpage to register the Application secured byHTTP basic auth. This happens out of band and is not part of the AppToken exchange protocol.

- The client application now requests a NONCE token by passing in it's Appkey

- SSO will then store the AppKey, to NONCE token mapping in MemCache with a 5 minute cache expiry
- Client concatenates AppKey + Nonce for signing by the secret key.
- Client requests a APOLLOAPPTOKEN using (Appkey, AppKey+NONCE signed by secret key)
- SSO server pulls Secret Key based on AppKey name from KeyServer and caches it in MemCache and validates AppKey and Secret have not exceeded Max TTL of secret
- SSO then validates signing of AppKey+NONCE with the Secret Key.
- If validation passes the SSO returns back a APOLLOAPPTOKEN.



Appendix

Evaluated Solutions

OpenID

- User asserts a OpenID URL, which is that users identity, belongs to him.
- The OpenID relying party then makes the user prove they own that identity by sending the user to that site and having them log in.
- If the login is successful a OpenID Relying Party makes a final request of the OpenID provider for some identity information using a user meta-data exchange protocol.
- The Relying Party then validates this meta-data against some know facts it has about the user. If the two match the user then is fully authenticated.

Note OpenID doesn't specify how the secure session is estabilshed or what use to establish that secure session after the Authentication is done. So any tokens generated would a a custom solution.

OAuth 1.0 2 legged Auth

Classroom currently has a Oauth 1.0 two legged exchange using Secret and client signing model. Under this style a client gets a Private Key and Secret out of band from the authentication and simply uses the private key to sign the the Oauth header and appends their token.

https://wiki.apollogrp.edu/display/NGP/Service+to+Service+Authentication+using+Oauth

OAuth 1.0 3 legged Auth

The basics of 3 legged authentication are

1. The client gets a temporary token from the OAuth Provider
2. The user then is presented with a dialog asking if the user want to allow the requesting Application access
3. The user authenticates on the IDP
4. The temporary token is replaced now with an Authorization token that give the bearer the right to make further calls.

Oauth Clients are required to be able to hold a secret key used for signing and the AuthToken generated by the OAuth provider.

Here's a google tutorial walking through the process of 3 legged auth.

http://hueniverse.com/oauth/guide/authentication/

http://googlecodesamples.com/oauth_playground/index.php

Problems with OAuth and Open ID

If you consider the Web Browser the client, a **web browser (and other light weight clients) cannot be relied upon to hold secret keys securely.** The user of the browser has access to modify anything in the DOM or Cookies so any keys passed to the end user can be tampered with on the web browser client. OAuth and Open ID both require the Relying Party Client to hold a secret key and use it to sign returning headers. This means the relying party client holds a secret key which can't be allowed on a browser.

Mobile clients generally aren't as vulnerable to tampering with keys or tokens but a mobile client likewise should not be holding private keys.

Both OpenID and OAuth require signing so strict adherence to these standards really rules out using Web browsers as the relying party making the authorized call. What is happening in these cases is that OAuth or OpenId are used to authenticate to establish a secure session and then so long as the session token exists browser is allowed to access resources on the Application. OAuth and OpenID actually do not specify how the secure session with the browser is established this is left up to the implementing Application. So OAuth and OpenID as is don't address Webbrowsers and Mobile Clients the major use case for this effort.