Ping Identity®

The Essential OAuth Primer: Understanding OAuth for Securing Cloud APIs

Executive Overview

A key technical underpinning of the Cloud is the Application Programming Interface (API). APIs provide consistent methods for outside entities such as web services clients and desktop applications to interface with services in the Cloud. More and more, it will be through APIs that cloud data moves; however, the security and scalability of APIs are currently threatened by a problem called the password anti-pattern – the need for one API to collect and replay the password for a user at another API in order to access information on behalf of that user.

OAuth defeats the password anti-pattern, creating a consistent, flexible identity and policy architecture for web applications, web services, devices, and desktop clients attempting to communicate with Cloud APIs.

Table of Contents

Motivating use case – TripIt

Like many applications today, TripIt (http://tripit.com) is a cloud-based service. TripIt is a travel planning application that allows its users to track things like flights, car rentals, and hotel stays. Users email their travel itineraries to TripIt, which then builds for the users a coordinated view of their upcoming trips (as well as those of their TripIt friends – the inevitable social aspect).

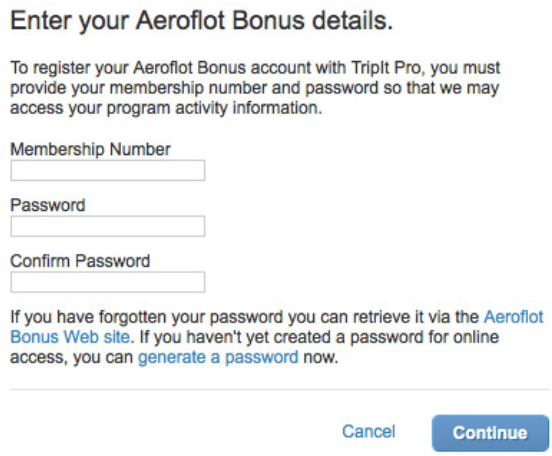
TripIt is undeniably useful as an isolated service; having a single cohesive view of travel plans (accessible from both web and phone) is a boon to business travelers. But TripIt becomes significantly more useful and valuable when, rather than being isolated, its view of travel plans becomes integrated with the user's existing identities. For instance:

a. Insertion of TripIt travel itineraries into online calendars such as Google Calendar
b. Detection of new travel itineraries directly from the user's email inbox
c. Automated publishing of travel details to social networks
d. Integration of travel dates and trip segments into corporate expense reporting applications

All of the above integrations can be accomplished through the use of APIs. In some cases TripIt must act as a web services client to leverage APIs offered up by other services. In other cases TripIt would need to make the traveler's itinerary available to other services via its own API. Given that the identity data and attributes stored behind either of these APIs are potentially sensitive, the traveler will surely want some level of control over access.

Before OAuth, the default mechanism for enabling some level of user control over API access was to leverage something that each site had for the user – his password. The so-called *password anti-pattern* allowed Site A to use an API hosted by Site B to access a user's attributes and data by asking the user for his password at Site B. By demonstrating knowledge of the user's password on subsequent API calls, Site A would prove to Site B that it had the user's (necessarily implicit) authorization to access the data.

The screenshot below illustrates the password anti-pattern – the user is being asked for his Aeroflot password by TripIt.

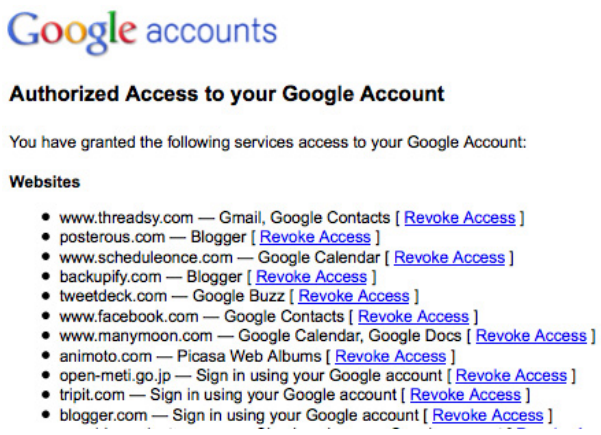


The password anti-pattern is far from optimal as a security mechanism:

- It teaches users to be indiscriminate with distributing their passwords (a habit that phishing ultimately relies on).
- As the hosting site is not involved in the authorization step (the user consenting to sharing his password with the requesting site), the hosting site is unable to provide to the user a record of which requesting sites he has authorized to access his data held at the hosting site. The burden is on the user to track such grants.
- The copies of the passwords at the requesting sites present a risk for breach through compromise.
- It doesn't support granular permissions, e.g. Site A can read but not write. Because it relies on the requesting site *impersonating* the user, the hosting site must grant the same privileges to the requesting site as to the users themselves.
- It doesn't support (easy) revocation – to be sure of turning off the access rights previously granted to a requesting site, users must change their password at the hosting site. If they had previously authorized other requesting sites at the same hosting site, changing the password would immediately (presumably unintentionally) revoke those permissions, as well.
- Because it relies on passwords to the hosting site being distributed across the Web, it effectively locks that site into password authentication – preventing it from adopting stronger or federated alternatives (without negatively impacting users).

The fundamental problem with the password anti-pattern is that, to assign permissions to a site like TripIt (for instance) for accessing a user's data and services held at Google (for instance) it relies on TripIt ***asking the user directly*** for the desired permissions, rather than TripIt ***asking Google to ask the user*** for the desired permissions. The latter is the OAuth model.

The OAuth model allows the user to delegate to TripIt the desired permissions – the explicit delegation occurring at Google and not implicitly at TripIt. Because the delegation (and others the user might grant) occurs at Google, Google can record it and provide an interface to the user for his management (e.g. revoking particular grants) – as shown in the diagram below.



Terminology

- **Authorization Server**—actor that issues access tokens and refresh tokens to clients on behalf of resource servers.
- **Access token**—data object by which a client authenticates to a resource server and lays claim to authorizations for accessing particular resources. Access tokens have specific authorization scope and duration.
- **Client**—actor that desires access to resource protected by a resource server, and interacts with an authorization server to obtain access tokens to do so.
- **Refresh Token**—long lived token that a client can trade-in to an authorization server in order to obtain a new access token (with the same attached authorizations as the existing access token). Refresh tokens allow clients to obtain fresh access tokens without obtaining fresh authorization from the resource owner.
- **Resource Server**—actor protecting resources and making them available to properly authenticated and authorized clients.
- **Resource Owner**—actor (typically human) that controls client access to particular resources hosted by a resource server. A resource owner specifies the authorizations at an authorization server—the authorizations then manifested in an access token issued to the client in question.

Introduction

OAuth 2.0 defines a framework for securing application access to protected resources (often but not solely identity attributes of a particular user) through Application Programming Interfaces (APIs) – typically RESTful. There are three primary participants in the OAuth flow. OAuth allows a client (an application that desires information) to send an API query to a resource server (RS), the application hosting the desired information, such that the RS can authenticate that the message was indeed sent by the client. The client authenticates to the RS through the inclusion of an *access token* in its API message – a token previously provided to the client by an authorization server (AS). In those OAuth scenarios in which the API in question protects access to a user's identity attributes, it may be the case that the access token will only be issued by the AS after the user has explicitly given consent to the client accessing those attributes.

OAuth 2.0 includes:

1. A web-redirect based mechanism by which a resource owner can delegate authorizations for access to his resources (e.g. profile) held at some site to some 3rd party client (this is the archetypical component).
2. A variety of other mechanisms by which a resource owner can delegate authorizations to his identity attributes held at some site to client applications (e.g. on a desktop, phone, set top boxes, etc.).
3. A constrained Security Token Service (STS) model similar to WS-Trust, notably designed around REST principles rather than SOAP messaging. The STS supports both token issuance and refresh.

4. A set of client authentication mechanisms for REST-based HTTP APIs – including APIs that:
   a. protect a resource owner's identity attributes for which the resource owner's explicit consent is required
   b. protect a resource owner's identity attributes but for which the resource owner's consent is implicit
   c. protect non-resource owner specific data (therefore no consent is required)

Some of the above OAuth pieces definitely do have an authorization *flavor* – others are arguably more aptly described as authentication or token mapping.

While OAuth may have been originally positioned as a focused 'delegated authorization' mechanism, even in its earliest versions it was more than that; and with OAuth 2.0 approaching standardization within the Internet Engineering Task Force (IETF), it can no longer be so succinctly described.

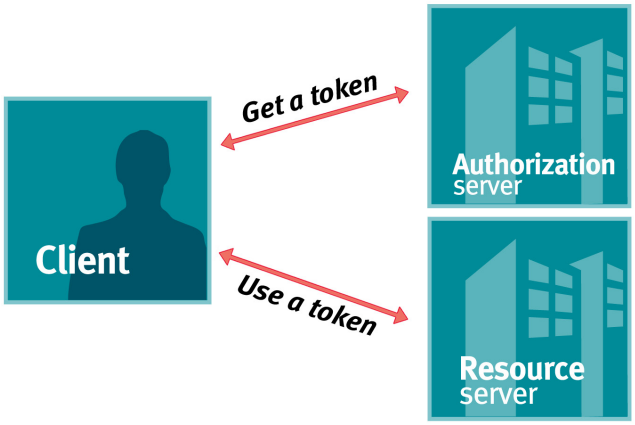
OAuth 2.0 provides a flexible authorization and authentication framework for protecting REST APIs. And with the importance of such APIs to the Cloud, OAuth will provide an integral role in securing the Cloud.

The OAuth model

Notwithstanding the above complexity, a useful organizing model for thinking of OAuth 2.0 is:

a. mechanisms by which a client can obtain a security token from an appropriate authority in order to use that token for authenticating a subsequent API call.
b. mechanisms by which a client can present a security token as part of an API call in order to authenticate itself (and thereby enable an authorization decision by the API hosting RS).

The above distinction can be simply described as 'getting a token' and 'using a token.' These two logical halves of OAuth are shown in the diagram below. The client 'gets a token' from the AS and then 'uses the token' to authenticate to the RS – behind which is the data the client desires to obtain or manipulate.



OAuth 2.0 supports a variety of mechanisms for 'getting a token' and a smaller number for 'using a token.'

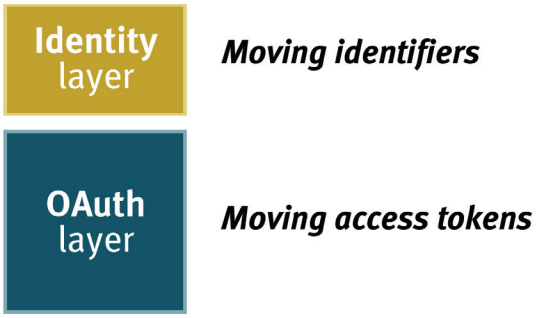
Relationship to other standards

OAuth does not exist in isolation from existing identity and security standards. We list here possible relationships between OAuth and some representative specifications.

OAuth & OpenID

In OAuth 2.0, the set of redirects and back-channel calls deliver to the client an access token for authenticating subsequent API calls. In OAuth, the identity of the resource owner in question for a given message exchange is implicit – expressed only indirectly in the permissions carried in the access token. In Web SSO, the redirects and back-channel calls allow an identity provider (IdP) to deliver to some service provider (SP) a claim/assertion that a particular resource owner has authenticated to them – with the assertion carrying an explicit identifier for the subject in question. These seem very different semantically. SSO presumes that the IdP and SP share some identifier for the subject as the means to refer to them. OAuth 2.0 on its own does not provide such an identifier; the access token is not an identifier for the user but rather a means to subsequently obtain such information.

But we can, however, imagine adding an identity layer to OAuth 2.0 – we can stipulate how to add to the existing OAuth 2.0 messages the necessary identity identifiers to make SSO possible.



This is the OpenIDConnect premise, namely, that rather than having a distinct SSO protocol like OpenID for sharing identifiers and a separate protocol for sharing access tokens, the two can be combined. Some members of the OpenID community are proposing that the next revision of OpenID should use the OpenIDConnect model.

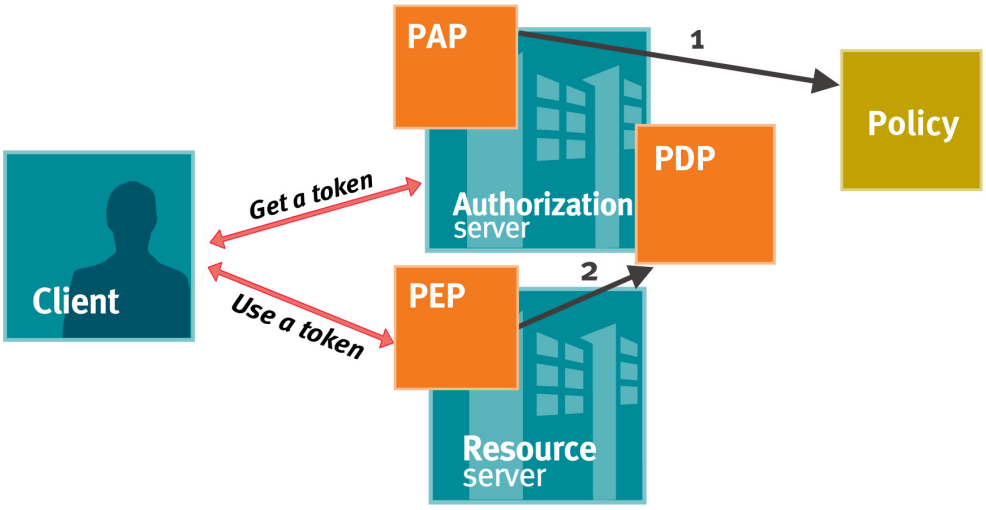
FaceBook's existing support for OAuth 2.0 anticipates the OpenIDConnect model. In their implementation, the OAuth process delivers to the OAuth client the user ID of the relevant FaceBook user, in addition to the normal access token. The user ID enables SSO, the access token subsequent access to FaceBook's Graph API (behind which the user's social network data and updates lie).

The OpenIDConnect model effectively has OAuth messages carry OpenID parameters. This is the exact opposite model of the OpenID/OAuth Hybrid proposal, in which the OpenID messages were used to carry OAuth parameters and thereby optimize the combined sequence.

OAuth & XACML

Authorization has many different facets, and to describe OAuth solely as an 'authorization standard' begs confusion with the other authorization facets. For instance, the extensible Access Control Markup Language (XACML) is manifestly focused on authorization – but there is effectively no overlap at all between XACML & OAuth (in fact they are nicely composable).

We can demonstrate this composability by overlaying XACML's authorization model onto the above OAuth diagram.



In the case of obtaining the resource owner's consent before the token is issued to the client, the OAuth AS effectively plays the role of the XACML policy administration point (PAP) – at which policy is defined and subsequently stored as an XACML policy. In this case, the XACML policy might record the fact that the resource owner consented to the client being able to read their attributes held at the resource server, but not make any changes. Once it receives the token from the AS, the client can then use that token on its API calls to the RS. At the resource server, an XACML policy enforcement point (PEP) would intercept the API call (let's assume it was an HTTP POST that attempted to add some new attribute to the resource owner's store) and call out to the XACML policy decision point (PDP) to obtain an access control decision. In this case, as the resource owner has previously specified that the client could read but not write, the POST request would be denied and the PDP would respond accordingly to the PEP.

To be clear, OAuth does not presume or require an underlying XACML infrastructure. The point here is only that OAuth and XAMCL, while both authorization-centric, are compatible.

OAuth & SAML

As you might expect for two general purpose security frameworks, there are a number of different integration points between OAuth and the Security Assertion Markup Language (SAML), including:

- SAML SSO can be used to authenticate the resource owner to the AS at the time of obtaining authorization.
- As for other SSO protocols, SAML messages can carry OAuth parameters (e.g. authorization codes, access tokens, refresh tokens etc.), thereby enabling subsequent API access following SSO.
- A SAML assertion can be traded for an OAuth access token.

We discuss some of the above integrations in the following sections.

OAuth 2.0 Overview

For an overview of OAuth 2.0, we use the previous distinction between 'getting a token' and 'using a token.'

Getting a token

To get an access token, a client interacts with an authorization server, sending a request for an access token that includes an access grant. In some scenarios, the client also sends its own credentials to the AS on this request message.

OAuth defines five different access grant types – the different grant types reflecting different authorization mechanisms:

1. Authorization code – the authorization code access grant is returned to the client after the resource owner explicitly gave his consent to the AS for the client's desired privileges.
2. Resource owner credentials – this access grant implies the client collecting the resource owner's password at the AS, and presenting the password to the AS on the access token request. While reminiscent of the password anti-pattern, the client must discard the password after using it to obtain the access token. Although Twitter uses this model for some mobile apps, it is not generally seen as preferred.
3. Refresh token – in some situations, OAuth allows the AS to return to the Client both an access token and an associated refresh token. Once the original access token expires, the corresponding *refresh token* can be sent to the AS in order to obtain a fresh access token. Refresh tokens are never sent to the RS.
4. Assertion – the assertion access grant involves the client trading some other representation of a resource owner's identity and/or consent for an access token.
5. The client credentials access grant is special; it has the client presenting his own credentials to the AS in order to obtain an access token. This access token is either associated with the client's own resources and not a particular resource owner, or is associated with a resource owner for which the client is otherwise authorized to act.

Related to the above access grant types are four client profiles – web server, user-agent, native application, and autonomous. The four profiles stipulate how a client can obtain an access token from an AS in a given deployment model (typically with particular constraints and challenges) and which of the access grant types are relevant for such deployments. For instance, the web server profile is optimized for clients capable of redirecting the resource owner's browser to the AS and that are capable of protecting a client credential used to authenticate to the AS when trading an authorization code for an access token.

Using a token

To use an access token, a client interacts with an RS – including the access token previously obtained from the AS on its API call to the RS. The access token serves to allow the RS to determine that the client is authorized to access the resources in question.

OAuth 2.0 will allow for two broad categories of access tokens: *bearer* tokens (in which the mere possession of the access token will be interpreted as providing sufficient proof to the RS that the entity presenting the token is the same as that to which the AS issued it) and *Holder of Key* tokens (in which the client will need to demonstrate knowledge of some secret bound to the token in order for the RS to grant access to the corresponding resources).

Token type

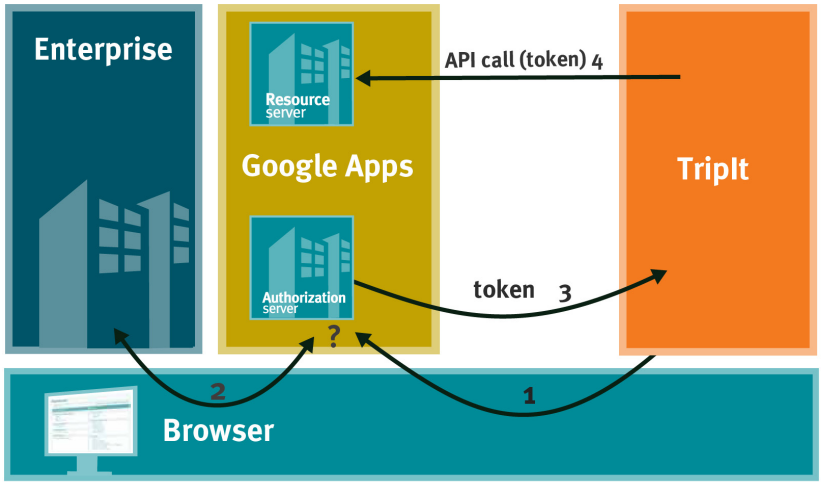
OAuth 1.0a did not stipulate a particular structure for access tokens; by default they were opaque text strings that served only to look up the corresponding set of authorizations. Work is now underway in the IETF OAuth 2.0 Working Group to define, in addition to the opaque token model, a JSON (JavaScript Object Notion) formatted structured alternative – providing similar advantages as do SAML assertions.

Use cases

In addition to the TripIt use case discussed earlier (now framed in terms of OAuth), we present in this section some representative use cases that illustrate the flexibility of the OAuth 2.0 framework.

TripIt revisited

Shown below is the conjectured TripIt and Google Apps integration – this time using OAuth's web server flow as the means by which TripIt obtains an access token from Google Apps and thereby enables access to the Google Apps APIs behind which the resource owner's data and services lie.



At the time of creating an account at TripIt, the user (an employee of an enterprise customer of Google Apps) opts to base his account off his Google Apps account and so in Step 1, TripIt sends the browser there. As the employee's enterprise has established SAML SSO to Google Apps, the employee authenticates to the enterprise SAML IdP, and then in Step 2 is returned to Google Apps with a SAML assertion. In a consumer scenario, at this point Google would ask the authenticated consumer for consent to allow TripIt to access the consumer's account. In this Google Apps enterprise scenario, it would be possible for Google Apps (implementing the defined policy of the enterprise) to automate this consent step and redirect the employee back to TripIt with the OAuth access token. (In fact, OAuth stipulates that the redirect carries not the token itself but the authorization code that can be exchanged for the access token. For the sake of clarity, we omit that detail.) Armed with the access token, TripIt can then use the APIs that Google Apps offers up to integrate with the employee's Google-hosted data services.

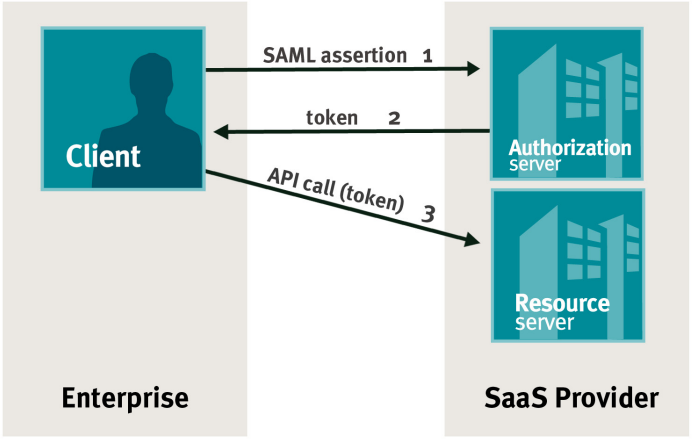
The next time the employee adds a business trip to his TripIt account, the itinerary would then show up in the corresponding Google Apps calendar and so inform the traveler's colleagues.

Note that the above adds a small twist to the perhaps typical OAuth web server flow, in that the user authenticates to Google's AS through SAML-based SSO rather than directly with Google credentials. Any of the other Web SSO protocols (OpenID, WS-Federation) could be used in the same manner – ultimately, OAuth is independent of the user's authentication to the AS.

Token exchange

This use case displays how OAuth style REST API authentication can be enabled by an existing trust relationship and SAML-based SSO infrastructure between an enterprise and a SaaS provider. An enterprise has implemented SAML SSO to the SaaS provider – allowing its employees to access browser-based resources and applications hosted by the SaaS provider. But new use cases require the enterprise to be able to call a SaaS provider hosted API to retrieve employee-specific data, e.g. for a CRM Cloud provider, sales data for a particular sales representative.

OAuth can be used to secure the REST API calls from enterprise to the Cloud, and the fact that the enterprise and the SaaS provider already have SAML SSO working between themselves can facilitate this REST API access. This scenario is shown in the diagram below:



The enterprise creates a SAML assertion for the particular sales employee as it would normally do for SAML SSO, but instead of delivering it to the SaaS provider through the browser, uses the OAuth assertion flow to trade the SAML assertion at the SaaS AS for the desired access token (Steps 1 & 2). Once armed with the access token, the enterprise client includes it on subsequent API calls to the SaaS provider RS. As it was issued based on the named employee within the SAML assertion, the access token indirectly specifies that employee, and so allows the SaaS provider to respond with employee-specific CRM data.

The named subject within the SAML assertion identifies the particular employee in question, and the enterprise signature over that assertion serves to demonstrate that the client 'belongs' to the enterprise and is implicitly authorized by the enterprise to request access tokens of the AS.

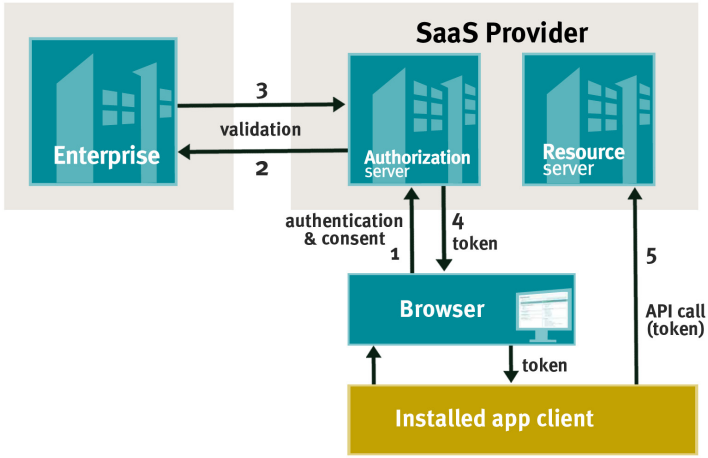
For the sake of simplicity, we don't show in the above a likely interaction between the OAuth client and a local Security Token Service to obtain the SAML assertion before trading it to the SaaS AS in Step 1. This interaction could be WS-Trust, or conceivably a future profile of OAuth's own STS.

This is one integration scenario between SAML and OAuth. Another interesting scenario uses the SAML message flow as an alternative delivery channel for getting the OAuth access token to the OAuth client. This is the same scenario as the previously discussed OpenID/OAuth hybrid.

Mobile workforce

Devices are first-class Cloud citizens – no longer passive intermediaries of browser redirects but active consumers of APIs and data. OAuth recognizes the importance of devices with message flows optimized for their particular constraints.

Shown below is the scenario of an enterprise employee wishing to access his SaaS account through his Android Tablet – using an installed application to retrieve and/or manipulate data through API calls to the SaaS RS.

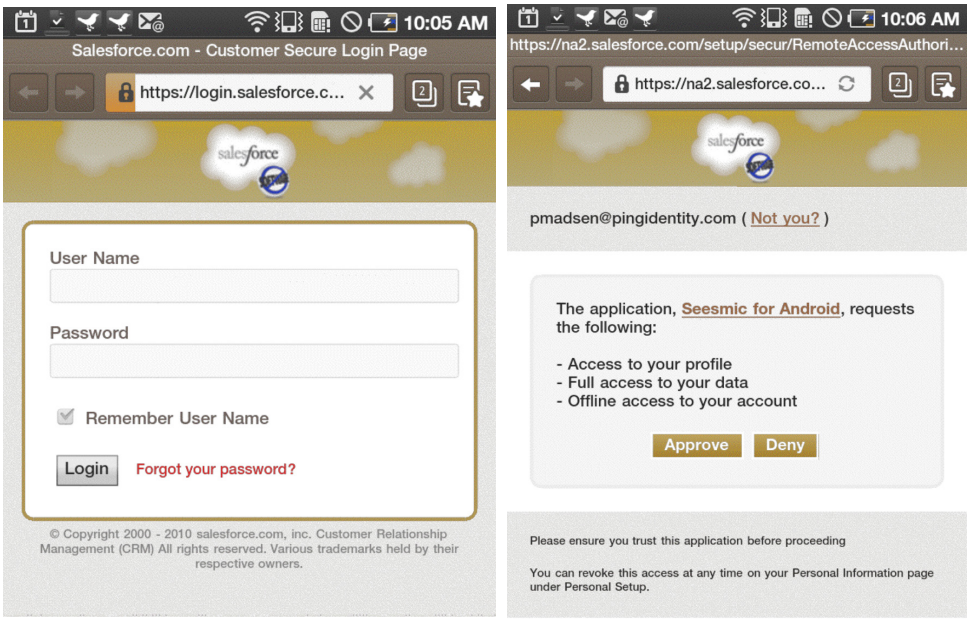


The mobile application client uses the OAuth Native Application profile in order to retrieve an access token from the AS – relying on a separately launched browser for both the authentication to the AS (Step 1) and delivery of the access token back to the client (Step 4) – this used on subsequent API calls to the SaaS RS (Step 5).

Relying on the browser for authentication to the AS and collection of consent has advantages over trying to duplicate the UI/UX within the installed application – including being able to leverage existing browser-based authentication mechanisms and aids.

As shown, the SaaS provider calls out to the enterprise for validation (Steps 2 & 3) of the employee's credentials. Alternatively, and arguably preferably, the SaaS provider and enterprise could leverage SSO.

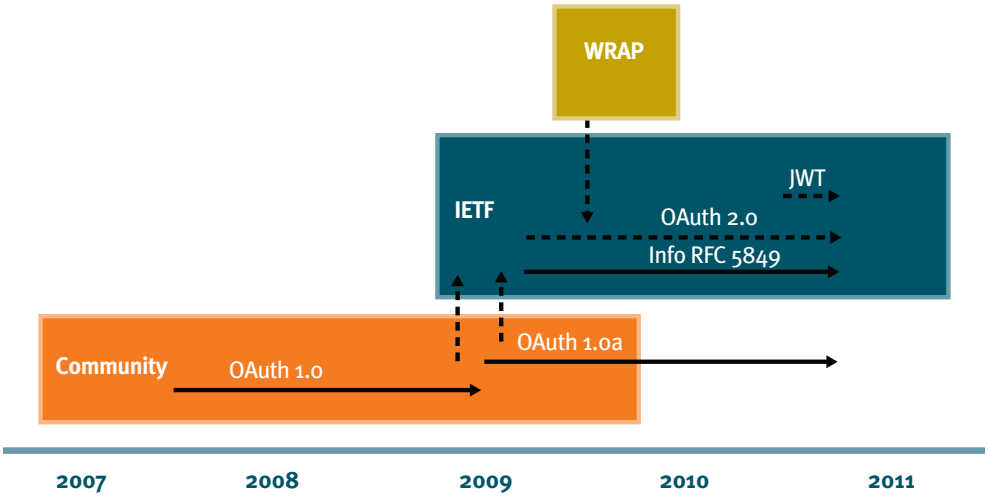
Salesforce uses the above model to authenticate and authorize mobile clients for its Chatter service. Shown below on the left is a screenshot of a Salesforce web page launched by the Seesmic for Android client for Salesforce Chatter. On the right is the Salesforce consent page.



A key enhancement of OAuth 2.0 compared to OAuth 1.0a is to allow for separation between the roles of AS & RS; they need not be performed by the same actor. Consequently, also possible for this use case would be a model in which the enterprise hosted an AS for the resources held by the SaaS provider RS. At least currently, it is fair to say that the model has not been tested. The User Managed Access (UMA) activity within the Kantara Initiative is notable for defining an architecture and protocols on top of OAuth 2.0 to explicitly enable such separation.

OAuth Timeline

The following diagram illustrates key milestones in the OAuth timeline.



OAuth 1.0 emerged from the large social providers like Facebook, Yahoo!, AOL, and Google. Each had developed its own alternative to the password anti-pattern. OAuth 1.0 reflected their agreement on a single community standard.

In 2009, an attack on OAuth 1.0 was identified which relied on an attacker initiating the OAuth authorization sequence, and then convincing a victim to finish the sequence – a result of which would be the attacker's account at an (honest) client being assigned permissions to the victim's resources at an (honest) RS. OAuth 1.0a was the revised specification version that mitigated the attack.

In 2009, recognizing the value of more formalized standardization, that community contributed OAuth 1.0 to the IETF. It was within the IETF Working Group that the original OAuth 1.0 was reworked and clarified to become the Informative RFC 5849.

In 2010, Microsoft, Yahoo!, and Google created the Web Resource Authentication Protocol (WRAP), which was soon submitted into the IETF WG as input for OAuth 2.0. WRAP proposed significant reworking of the OAuth 1.0a model. Among the changes were the deprecation of message signatures in favor of SSL, and a formal separation between the roles of 'token issuance' and 'token reliance.'

Development of OAuth 2.0 in the IETF consequently reflects the input of both OAuth 1.0, OAuth 1.0a, and the WRAP proposal. It is fair to say that the very different assumptions about what are appropriate security protections between OAuth 1.0a and WRAP have created tensions within the IETG OAuth WG.

While OAuth 2.0 initially reflected more of the WRAP input, lately (i.e. fall 2010) there has been a swing in group consensus that the signatures of OAuth 1.0a that were deprecated by WRAP are appropriate and desirable in some situations. Consequently, signatures are to be added back as an optional security mechanism.

While many deployments of OAuth 1.0a survive, more and more OAuth 2.0 deployments are appearing – necessarily against a non-final version of the spec. For instance, Facebook, Salesforce, and Microsoft Azure ACS all use draft 10 of OAuth 2.0.

Conclusions

The API is a key technical underpinning of the cloud – more and more, it will be through APIs that cloud data moves. OAuth enables a single, consistent, and flexible identity and policy architecture for web applications, web services, devices, and desktop clients for accessing cloud APIs. Critically, OAuth's token-based architecture provides important security characteristics compared to alternative API security mechanisms – including delegation support, replay prevention, and granular permissions.

*Information in this paper is based on the details in OAuth 2.0 v10.

About Ping Identity Corporation

Ping Identity is the market leader in Internet Identity Security, delivering on-premise software and on-demand services to hundreds of customers worldwide. For more information, dial U.S. toll-free **877.898.2905** or **+1.303.468.2882**, email **sales@pingidentity.com** or visit **www.pingidentity.com**.