THE ESSENTIAL OAUTH PRIMER:

UNDERSTANDING OAUTH FOR SECURING CLOUD APIS

WHITE PAPER

TABLE OF CONTENTS

03 EXECUTIVE OVERVIEW

03 MOTIVATING USE CASE: TRIPIT

05 TERMINOLOGY

06 INTRODUCTION

07 THE OAUTH 2.0 MODEL

07 OAUTH 2.0 OVERVIEW

- USING A TOKEN
- TOKEN TYPE

09 RELATIONSHIP TO OTHER STANDARDS

11 USE CASES

- TRIPIT REVISITED
- TOKEN EXCHANGE
- MOBILE WORKFORCE

13 RECENT DEVELOPMENT

14 SUMMARY

EXECUTIVE OVERVIEW

A key technical underpinning of the cloud and the Internet of Things are Application Programming Interfaces (APIs). 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, cloud data will move through APIs, but the security and scalability of APIs are currently threatened by a problem call the password anti-pattern.

This is the need for API clients to collect and replay the password for a user at an API in order to access information on behalf of that user via that API. OAuth 2.0 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.

MOTIVATING USE CASE: TRIPIT

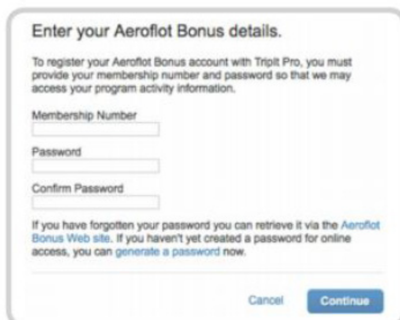
Like many applications today, TripIt **(http://tripit.com)** is a cloud-based service. It's a travel planning application that allows its users to track things like flights, car rentals, and hotel stays. Users email their travel itineraries to TripIt, which then builds a coordinated view of the users' upcoming trips (as well as those of their TripIt friends—the inevitable social aspect). TripIt is undeniably useful as an isolated service because business travelers haven't had a single cohesive view of travel plans in the past (especially one that's accessible from both web and phone). 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:

- Insertion of TripIt travel itineraries into online calendars such as Google Calendar
- Detection of new travel itineraries directly from the user's email inbox
- Automated publishing of travel details to social networks
- 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 clientto leverage APIs offered up by other services. In other cases, it 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 users—their passwords. The so-called password anti-pattern allowed Site A to use an API hosted by Site B to access user attributes and data by asking users for their passwords 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 authorization to access the data.

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

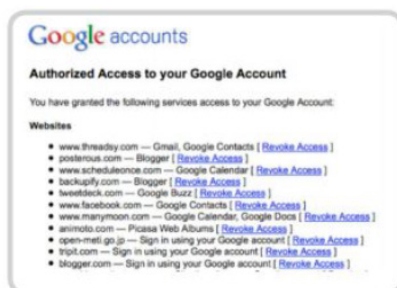


The password anti-pattern is far from optimal as a security mechanism for a variety of reasons:

1. 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.
2. 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.
3. 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 revoke those permissions as well.
4. 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 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 2.0 model.

The OAuth 2.0 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**—a long-life 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 are 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 identity attributes of a particular user) through Application Programming Interfaces (APIs, which are typically RESTful). There are three primary participants in the OAuth flow: a client, a resource server (RS) and an authorization server (AS). OAuth allows a client (an application that desires information) to send an API query to an RS (the application hosting the desired information) so 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 (previously provided to the client by an AS) in its API message. In OAuth scenarios where the API in question protects access to a user's identity attributes, the AS might only issue the access token after the user has explicitly given consent to the client accessing those attributes. OAuth 2.0 includes:

- 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 third-party client (this is the archetypical component).
- 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.
- A set of client authentication mechanisms for REST-based HTTP APIs, including APIs that:
    - protect a resource owner's identity attributes for which the resource owner's explicit consent is required.
    - protect a resource owner's identity attributes, but for which the resource owner's consent is implicit.
    - protect non-resource owner specific data (therefore no consent is required).
- Some of the above OAuth 2.0 pieces definitely do have an authorization flavor. Others are arguably more aptly described as authentication or token mapping.

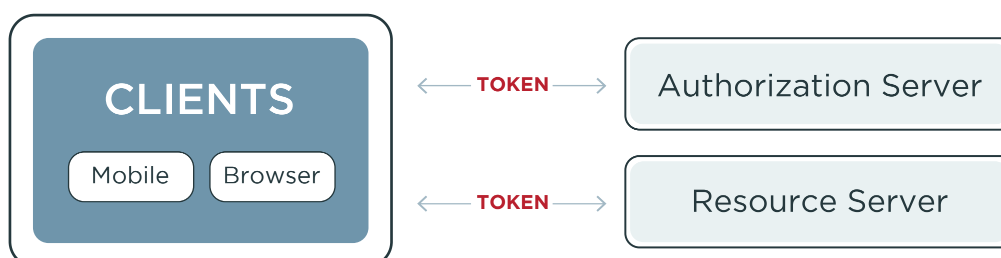
OAuth 2.0 provides a flexible authorization and authentication framework for protecting REST APIs. And with the importance of such APIs to the cloud, OAuth 2.0 will provide an integral role in securing the cloud.

THE OAUTH 2.0 MODEL

When thinking of OAuth 2.0, it helps to break it down into two categories:

- 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 to an RS.
- 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 that the client desires to obtain or manipulate.



OAUTH 2.0 OVERVIEW

For an overview of OAuth 2.0, let's consider the previous distinction between getting and using tokens. To get an access token, a client interacts with an AS by sending a request for an access token that includes what is known as an access grant. The access grant represents the resource owner's authorization for the client to be able to access the RS. The access grant is exchanged for the more fundamental access tokens. In some scenarios, the client also sends its own credentials to the AS on this request message. OAuth defines four different access grant types that generally reflect different client categories:

- **Authorization code—this access grant is returned to the client after the resource owner explicitly gave his consent to the AS for the client's desired privileges.**
- **Implicit—the AS returns the access token to the client directly, rather than an authorization code to be subsequently exchanged for the tokens.**
- **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. This is similar to the password anti-pattern, but the client must discard the password after using it to obtain the access token. This model isn't optimal and is only seen as a means of transitioning applications away from the password anti-pattern.**
- **Client credentials—this allows a client to obtain an access token in its own right, not as an authorization delegated to it by a user resource owner.**

In addition to the above grant types, OAuth 2.0 defines a Refresh token. In some of the grant types, OAuth 2.0 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.

Using a Token

To use an access token, a client sends an API call (typically RESTful) to an RS—including the access token previously obtained from the AS. By validating the access token, the RS is able to determine that the client is authorized to access the resources in question.

The OAuth 2.0 framework allows for two broad categories of access tokens: bearer tokens and Holder of Key (HoK) tokens. With bearer tokens, 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. With HoK tokens, 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.

Currently, a standard only exists for bearer tokens—namely RFC 6750. Work is underway in the IETF OAuth WG to define an HoK option, replacing the earlier MAC work.

Token Type

OAuth 2.0 does not require a particular structure for access tokens. By default, access tokens are opaque text strings that serve only to allow the RS to perform a lookup of the corresponding set of authorizations and attributes associated with that token. However, there are scenarios where it's advantageous to allow the RS to be able to directly validate an access token rather than have to call back to the issuing AS. Work is now underway in the IETF Web Authorization Protocol Working Group to define a JavaScript Object Notation (JSON) formatted structured alternative. These tokens, known as JSON Web Tokens (JWT), will provide for the RESTful API world a security token comparable to SAML assertions in SOAP Web Services.

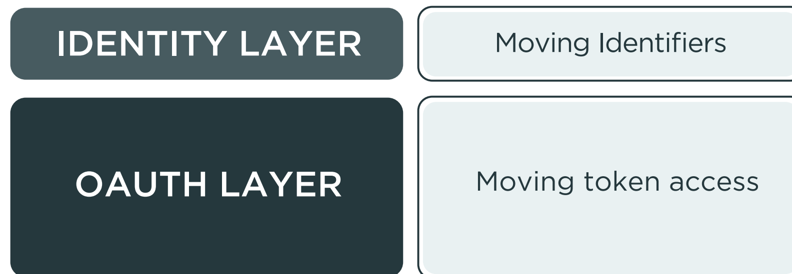
RELATIONSHIPS TO OTHER STANDARDS

OAuth does not exist in isolation from existing identity and security standards. Here's a list of possible relationships between OAuth and some representative specifications.

OAuth 2.0 & OpenID Connect

In OAuth 2.0, the set of redirects and back-channel calls deliver an access token to the client 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.

However, we can imagine adding an identity layer to OAuth 2.0. We can specify how to add the necessary identity identifiers to the existing OAuth 2.0 messages to make SSO possible.



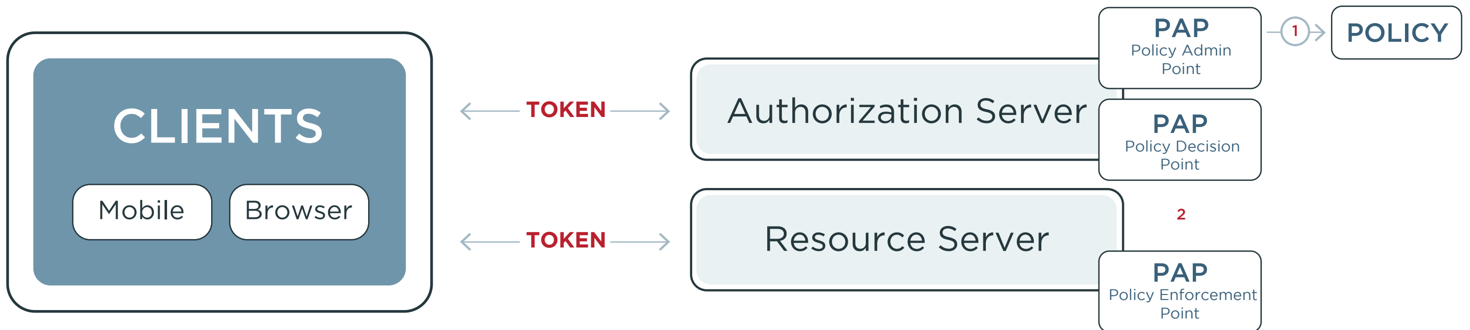
This is the OpenID Connect premise. Rather than having a distinct SSO protocol like OpenID Connect for sharing identifiers and a separate protocol for sharing access tokens, the two can be combined. OpenID Connect is being standardized in the OpenID Foundation and is currently in Implementor's Draft Review Period.

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

OAuth & XACML

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

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



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), in which the 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 RS, 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 RS, 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 2.0 and 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.

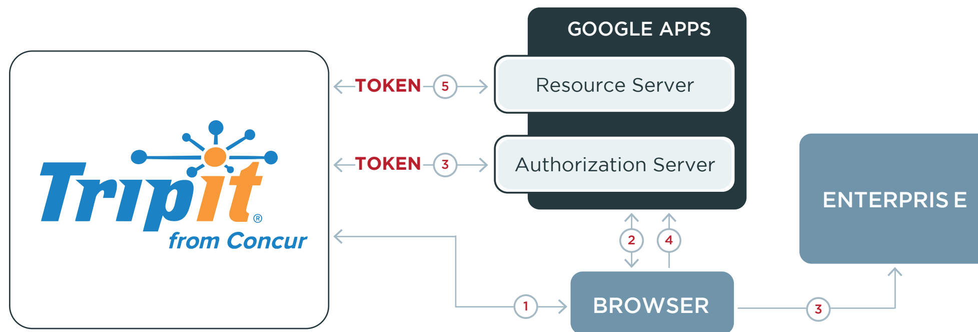
We discuss some of the above integrations in the following sections.

USE CASES

In addition to the TripIt use case discussed earlier, let's look at some representative use cases that illustrate the flexibility of the OAuth 2.0 framework.

TripIt Revisited

Now framed in terms of OAuth, the graph below shows the assumed TripIt and Google Apps integration. This time, it uses 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.



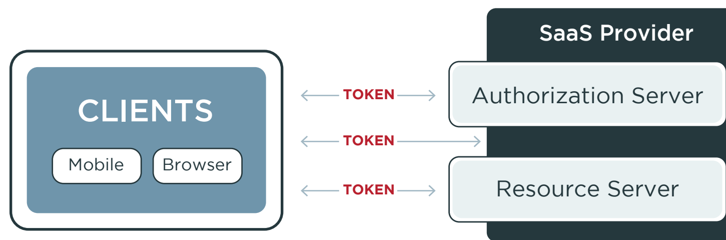
At the time of creating an account at TripIt, the user (an employee of an enterprise customer of Google Apps) opts to base his account off his Google Apps account, so TripIt sends the browser there in Step 1. As the employee's enterprise has established SAML SSO to Google Apps, the employee authenticates to the enterprise SAML IdP, and then in Step 2 is returned to Google Apps with a SAML assertion. In a consumer scenario, Google would then 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 requires that the redirect carries not the token itself, but an authorization code that can be exchanged for the access token. 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 typical OAuth 2.0 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, WSFederation) could be used in the same manner. Ultimately, OAuth 2.0 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, this might be sales data for a particular sales representative).

OAuth can be used to secure the REST API calls from the 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:



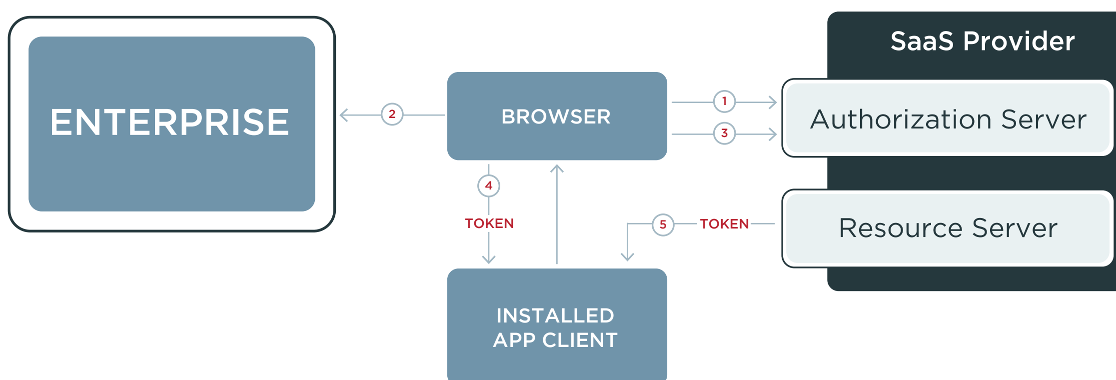
The enterprise creates a SAML assertion for the particular sales employee as it would normally do for SAML SSO, but instead of delivering it to the SaaS provider through the browser, it uses the OAuth assertion flow to trade the SAML assertion at the SaaS AS for the desired access token (Steps 1 & 2). 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 STS 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.

Mobile Workforce

Devices are first-class cloud citizens. No longer passive intermediaries of browser redirects, they've become 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.

RECENT DEVELOPMENTS

The OAuth community has continued to extend the framework beyond OAuth 2.0, including the following initiatives.

Introspection

OAuth 2.0 didn't define a particular structure for access tokens (e.g., a standard set of fields identifying the AS that issued it, the client to which it was issued, on behalf of what user, etc.). Consequently, most deployments of OAuth used a "by reference" model for access tokens in which the tokens were opaque to the RS and would therefore need to be validated by the issuing AS in order to extract this information. OAuth 2.0 also didn't standardize this request/response exchange between the RS and AS. Consequently, many deployers and implementers defined their own protocol. The introspection specification normalizes these various protocols.

Proof of Possession

The default security model for OAuth is that the access tokens be bearer tokens (e.g., any actor in possession of the access token is presumed to be the valid actor to which the token was issued). Some scenarios demand additional security protection whereby a client needs to demonstrate knowledge of some cryptographic keying material when using an access token on an API call to access a protected resource. This security model for tokens is generally known as 'proof of possession' as the client must be able to prove that they have a particular key before being able to successfully use an access token.

PKCE

There's a security issue in the above OAuth flow for a native application. Because native applications don't typically have unique secrets (reflecting how they are distributed) there's the possibility of a malicious application on the device inserting itself into the flow and obtaining the access token intended for the valid application. The Proof Key for Code Exchange (PKCE) mechanism mitigates this vulnerability—effectively allowing the native application to generate a onetime secret that can be used to authenticate to the AS , and so prevent a malicious application from obtaining tokens.

ACE

The Authentication and Authorization for Constrained Environments (ACE) working group in the IETF is exploring the potential for applying the OAuth model to IoT use cases, where many network nodes are limited in their processing power, connectivity or battery life.

SUMMARY

APIs are a key technical underpinning of the cloud. More and more, cloud data will move through RESTful APIs. 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, such as delegation support, replay prevention and granular permissions. Additionally, OAuth has emerged as a key platform on which to build in support of new use cases, as demonstrated by OpenID Connect, User Managed Access (UMA) and the ACE effort.

ABOUT PING IDENTITY: Ping Identity leads a new era of digital enterprise freedom, ensuring seamless, secure access for every user to all applications across the hyper-connected, open digital enterprise. Protecting over one billion identities worldwide, more than half of the Fortune 100, including Boeing, Cisco, Disney, GE, Kraft Foods, TIAA-CREF and Walgreens trust Ping Identity to solve modern enterprise security challenges created by their use of cloud, mobile, APIs and IoT. Visit pingidentity.com.

#3119 | 10.16 | v004