Abstract

Mobile clients are an increasingly important channel for consumers accessing Web 2.0 and enterprise employees accessing on-premise and cloud-hosted services. This white paper explains how an identity management architecture, with the help of both SAML and OAuth, can support the two broad categories—web applications delivered through the browser and native applications installed onto the device—by providing a single consistent and cohesive identity infrastructure for both.
# Table of Contents

Abstract ................................................................................. 1  
Introduction ........................................................................ 3  
Use Cases ........................................................................... 4  
  Browser SSO for Web Applications ........................................ 4  
  Consumer Native Application ............................................... 4  
  Enterprise Native Application ............................................... 4  
Key Identity Standards .......................................................... 5  
  SAML ............................................................................... 5  
  OAuth ............................................................................. 5  
  OpenID 2.0 ....................................................................... 5  
  OpenID Connect (Emerging) ............................................... 5  
How it Works ......................................................................... 6  
  Web Applications ................................................................ 6  
  Native Applications ........................................................... 8  
  Native Application with Federation ....................................... 9  
  The Role of OAuth ............................................................ 10  
Getting Tokens ...................................................................... 10  
Delivering Tokens to Applications ......................................... 12  
Storing Tokens ...................................................................... 13  
Using Tokens ........................................................................ 14  
Refreshing Tokens ................................................................ 14  
Revocation ............................................................................ 14  
Other Considerations ............................................................ 15  
Best Practices ........................................................................ 16  
  Web Applications ................................................................ 16  
  Native Applications ........................................................... 16  
Conclusion ............................................................................ 16  
References ........................................................................... 17
Introduction

Mobile devices are an increasingly important channel for delivery of data and services. Highly capable phones, tablets, and thick desktop applications are enabling new levels of mobility and functionality. A comprehensive authentication and authorization framework should provide support for both web applications that rely on a browser on the mobile device and native applications that rely on a downloaded distinct application installed onto the native OS.

These two data access models are shown below:

Both application models have benefits.

Native applications:

- Offer performance advantages, with no need to download the UI every time (HTML5’s caching changes this)
- Enable the ability to work offline and meaningfully interact with the application—even when not connected
- Are more effective at integrating mobile device capabilities into the application, such as camera, GPS, Contact List, etc.
- Provide easier payment integration (through the markets)

Web applications:

- Are multi-platform out of the box
- Allow for easy and timely updates
- Eliminate the need for a middle-man in the distribution channel
Presented here is a standards-based cloud identity management framework, which can address both models’ needs for mobile application development. The intended audience is three-fold: Web 2.0 service providers building mobile applications for consumers, enterprises building mobile applications for their employees or customers and SaaS vendors building mobile applications for their customers.

**Use Cases**

Currently, there are three high-level, mobile application use cases relevant to the Web 2.0 service provider, enterprise and SaaS vendor market. These use cases span access to data via a mobile browser or a native mobile application.

**Browser SSO for Web Applications**

An enterprise subscribes to a SaaS vendor, which has created a version of its website that’s optimized for mobile browsers. Employees interact with the application through their mobile browsers and authenticate to the mobile SaaS site via SSO based on their existing corporate credentials.

A variation would have a mobile consumer interacting with Web 2.0 service provider, but with SSO from a social network provider such as Facebook and LinkedIn.

**Consumer Native Application**

A Web 2.0 service provider builds and distributes a native application for popular mobile operating systems, like iOS and Android. Consumers download and install the application and then link the application to an existing or new account with the enterprise. Once linked, the application can read/write consumer-specific data from/to the provider’s servers to create a more customized experience.

**Enterprise Native Application**

A SaaS provider builds a native application for popular mobile operating systems, like iOS and Android. Employees of enterprise customers download and install the application, and then link it to an existing or new account at the Saas provider using their enterprise credentials for authentication. Once linked, the application can read/write employee-appropriate data from/to the SaaS providers to create a more customized experience.

A variation would have an enterprise migrating desktop applications to mobile native applications and leveraging existing authentication mechanisms for SSO using their enterprise credentials.
Key Identity Standards

There are several existing standards that play a role in the Mobile App IDm Architecture. The combination of these protocols enables seamless and secure access to data regardless of the device.

SAML

The Security Assertion Markup Language (SAML) is an XML-based standard for exchanging authentication and authorization data between security domains. For example, exchanging authentication between an Identity Provider (a producer of assertions) and a Service Provider (a consumer of assertions). SAML is the industry standard choice for a Single Sign On (SSO) mechanism for enterprise or cloud web applications.

Learn more about SAML.

OAuth

OAuth is an open standard for authentication and authorization for RESTful APIs. In a common deployment model, OAuth allows users to share information like a user profile or calendar stored on one website with another website through APIs—all without divulging their credentials to the requesting site.

Learn more about OAuth.

OpenID 2.0

OpenID 2.0 is an open standard for web SSO and is optimized more for consumer deployments than for enterprise or Cloud. Consequently, SAML and OpenID complement each other well.

Learn more about OpenID.

OpenID Connect (Emerging)

OpenID Connect is an emerging suite of lightweight specifications that provide a framework for communicating identity via RESTful APIs. OpenID Connect is seen as the evolution of OpenID 2.0, and is built as a profile of OAuth 2.0 rather than a completely distinct protocol foundation. OpenID Connect profiles OAuth 2.0 for SSO functionality and API access to identity attributes.

Learn more about OpenID Connect.

The relevance of these protocols for mobile applications is shown in the diagram below – SSO (SAML for enterprise and cloud deployments, OpenID for consumer) can be used for both web applications and during the authentication step for native mobile applications (denoted below by straddling the line between web and native). OAuth is relevant for authentication and authorization of both consumer and enterprise mobile native applications.
How it Works

Federated authentication for both web and native mobile applications is based on the exchange and delivery of tokens to the application. Tokens are issued by an Identity Provider (IdP)—typically in exchange for the user authenticating to the IdP.

Tokens either carry a set of user attributes directly, for example, as a self-contained aggregation, or act only as a reference or pointer to a set of attributes at a remote service.

Rather than a Service Provider (SP) directly validating a user’s credentials such as username or password, federation presumes that the user authenticates to an IdP, which generates a token targeted for the SP. As described below, the token flows vary slightly between web applications and native applications.

Web Applications

For web applications, the token is issued by an IdP, and then delivered through the browser, typically using cookies or the query string, to the application. Once the application validates the token, it serves up appropriate application HTML to the browser. The browser plays only a passive role in the delivery of the token to the application—it neither modifies the token, nor stores it for future use, but merely passes it on.

Federated authentication for web applications is shown on the next page:
In a cloud portal scenario where a mobile enterprise employee accesses various cloud applications by first logging into their enterprise, then requesting access to a specific resource, the sequence is this:

1. Mobile employee authenticates to their enterprise.
2. Enterprise creates a token expressing relevant identity attributes of the employee and delivers it through the mobile browser to the cloud service provider.
3. The cloud service provider validates the token, extracts the identity attributes, makes an authorization decision and, if appropriate, delivers application HTML to the mobile browser.

**Note:** The above diagram shows the flow beginning at the enterprise, but the employee could also begin at the cloud service provider. After being directed to the enterprise, the employee authenticates and the above flow continues.

With SSO for enterprise and cloud-based applications, SAML is the dominant protocol and token format choice. In the consumer space, OpenID is the equivalent although OpenID does not actually define the token format.

For the above enterprise to cloud application scenario, the enterprise plays the role of the SAML Identity Provider, and the cloud application provider plays the role of SAML Service Provider.
Native Applications

Unlike the web application client (namely the browser), native applications play a more active role in obtaining the tokens they will subsequently use to invoke APIs.

For native mobile applications, just as with web applications, a user’s credentials are typically exchanged for a token. In the diagram below, the native application relies on the mobile browser for this exchange. The token is then used on API calls to acquire user information.

The two scenarios for native applications defined below represent the user directly exchanging credentials for an access token or the user entering credentials at an identity provider (such as a SAML IdP or OpenID IdP) and exchanging a token for an access token. In both scenarios, the browser is used to collect credentials for (or deliver a token) to the service provider in exchange for an access token.

After installing the native application, the user is prompted to authorize this application. The native application launches a web browser, in which:

1. User authenticates to the server and authorizes the issuance of a token
2. Token is delivered through the browser to the native application
3. Native application presents token on API calls
4. Application returns user data as JSON
Native applications can store the tokens they receive. This means the tokens issued to the application will generally exist for longer periods of time – either for the length of a session, or, for some types of tokens, much longer.

**Native Application with Federation**

The current reality is that an Authorization Server (AS) and Resource Server (RS) are typically co-located, so the above application provider would host its own AS to issue tokens to be presented to its own RS APIs. Activity around signed JSON Web Token (JWT) will likely enable a model where an RS could consume and trust tokens issued by an AS in a different domain.

The web application and native application token exchange transactions can be chained together, with a passive token exchange preceding an active exchange. This scenario is shown below.

1. User authenticates to their enterprise IdP.
2. The enterprise creates a token, which is delivered through the browser to the cloud service provider.
3. Based on the identity attributes within the token, the cloud service provider issues a different token through the browser down to the native application.
4. The native application presents a token on API calls.
5. Application returns application data as JSON.
In the above, SAML enables SSO from the enterprise to the cloud service provider. For Web 2.0 providers, OpenID can be used to authenticate the consumer using service such as Google. The SaaS provider would play the roles of SAML SP and OAuth Authorization Server, and the enterprise would play the role of SAML IdP.

The Role of OAuth

OAuth is the emerging choice for the protocol where native applications obtain tokens to subsequently use on API calls. In the above flow, it is an OAuth Authorization Server that issues access tokens to the native application client—these tokens are then presented to the Resource Server.

Getting Tokens

The OAuth Authorization Server needs to authenticate the user and then optionally obtain their consent for the application to access their data before issuing tokens back to the native application.

OAuth provides a number of grant types potentially relevant to native applications:

- **Resource owner credentials.** The native application directly collects the username and password within its own UI and then trades those credentials to the AS for tokens (and discards the username/password).

- **Authorization code.** The user authenticates to the AS in a browser, which issues and delivers an authorization code to the native application and is then subsequently exchanged for tokens.

- **Implicit in which the user authenticates to the AS in a browser.** This authentication directly results in the issuance and delivery of an access token to the native application. The implicit grant type is designed for widget type applications running in web pages and although it can be used for native applications, it neither supports client authentication nor refresh tokens.

The authorization code grant type offers significant advantages compared to the resource owner credentials grant type:

- It does not preclude stronger authentication mechanisms beyond username/password.

- It does not preclude web-based SSO, allowing the user to authenticate to the AS via some separate identity provider.

- It does not preclude the user leveraging their existing browser stores for authentication stores, whether native to the browser or through extensions.

For the native application to use a browser for user authentication, a browser window must be launched. For example, on Android:
A new browser window is launched, in which the user will authenticate. As discussed before, the user can authenticate directly to the Authorization Server or through SSO from an IdP—their enterprise for cloud scenarios or their preferred social network for consumer scenarios. After authentication, it may be appropriate to show the user a consent screen, indicating the requested scopes to be authorized, as shown below.

The above indicates a Salesforce consent screen, displayed after the user had SSOd from their enterprise using SAML.

**Note:** The authorization code grant type can also be used for embedded browsers—for example, as a WebView on Android—as opposed to a separate window as shown above. Some think that an embedded browser offers improved usability so the user is not removed from the application context for authentication. However, users are familiar with browsers launching in a separate window and will have stored passwords in the device’s main browser—not the browser specific to the native application.
Delivering Tokens to Applications

Once the AS has issued a token(s), it needs to be delivered to the application for storage and subsequent use on API calls. How the token(s) are delivered back to the client will depend on the specific grant type used to request the token.

For the Resource Owner credentials grant type, the AS simply includes the token(s) in its response to the client request, as shown here:

```
HTTP/1.1 200 OK
Cache-Control: no-store
Pragma: no-cache
Content-Type: application/json; charset=UTF-8

{
    "token_type":"Bearer",
    "access_token":"a0VuzD3NfDsjlCSTUZB5Lmxs7WPQ1x07DCHR",
    "expires_in":3600,
    "refresh_token":"mSTBpqQeSkRECNfDclfRDjREnmqeWVap0DseM6aXkixIX"
}
```

For the Authorization code grant type, the AS returns the authorization code as a parameter to the client’s registered Redirect_URI. A mechanism is needed to move the authorization code from the browser to the client – so that it can be exchanged for the more fundamental token(s).

There are a number of possible mechanisms to accomplish this, including:

- **User copy and paste.** The AS returns a page to the browser with the authorization code for the user to read and paste into the native application.
- **Monitor cookies.** The AS result page will save the authorization code to a cookie, so that the native application monitoring the browser’s cookie jar can extract it.
- **Monitor browser window title.** The AS result page will include the authorization code in the HTML Title of the returned page, so that the native application monitoring the title can extract it.

What appears to be the emerging best practice is to rely on the native application registering itself as the handler for a custom URI scheme on installation, so that when the AS sends the authorization code to a Redirect_URI using this scheme, then the browser invokes the native application and hands off the authorization code for subsequent exchange by the native application for the desired token(s).

In Android, the native application can use the intent mechanism to register itself as the handler for the custom scheme (shown below as x-myApp) by adding an extra block to the AndroidManifest.xml.
After user authentication and authorization, when the AS sends the authorization code to the redirect_URI (x-myApp://redirect), the above activity kicks in and the authorization code can be extracted.

Once the native application has the authorization code, it is exchanged for the desired access token as shown in the request/response pair below:

```plaintext
POST /as/token.oauth2
Host: as.com
client_id=a&redirect_uri=mobileapp://redirecthere&grant
type=authorization_code&code=wizJmaSTPAf0wqSeB3vmDxZmNSZK6g HTTP/1.1
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
{
  "token_type":"Bearer",
  "expires_in":"600",
  "refresh_token":"oQWqwMUIL2ndeMHsWEyFO0GyalvGkM",
  "access_token":"lSBbci4Jg8MsjiSqZLBrzEXgd4mKUNhOkyF"
}
```

**Storing Tokens**

In iOS, the tokens can be stored in the key chain:

```objective-c
[accessToken storeInDefaultKeychainWithAppName:@"MyApp"
 serviceProviderName:@"Example.com"];

OAToken *accessToken = [[OAToken alloc]
       initWithKeychainUsingAppName:@"MyApp"
       serviceProviderName:@"Example.com" ];
```

Android offers comparable functionality.
Using Tokens

The native application retrieves the access token from local storage and includes the token on API calls to the RS, for example:

```
GET /double/secret/probation/resource HTTP/1.1
Host: rs.example.com
Authorization: Bearer a0Vu2D3NfDsjCsTUZB5LmXs7WPQ1x07DCHR
```

Refreshing Tokens

Access tokens are generally short-lived, so after they expire, the native application will need to obtain fresh ones. The refresh tokens the authorization server returned are exchanged for fresh access tokens, as in this request/response pair shown below:

```
POST /as/token.oauth2 HTTP/1.1
Host: as.example.com
Authorization: Basic c29tZWYoVudPbWVyaWNhJ3NlYXQ=
Content-Type: application/x-www-form-urlencoded; charset=UTF-8

client_id=clientId&grant_type=refresh_token&refresh_token=mSTBpgQcSkRECNfDc1fRDjREnmqeWVap0DseM6aXkixIX

HTTP/1.1 200 OK
Cache-Control: no-store
Pragma: no-cache
Content-Type: application/json; charset=UTF-8

{
    "token_type":"Bearer",
    "access_token":"MdqBuexY1MSogbrAwIPE47eGxGqZajuJNa",
    "expires_in":3600,"refresh_token":
    "hlyEO9PXgmvPiYI8g68KSEs2HqgrkiUQPsc9Xxskd"
}
```

Note that in the above, not only does the client receive a fresh access token, it also received a fresh refresh token. This sort of rolling refresh helps to mitigate the risk of a stolen refresh token, as the authorization server would revoke the older one upon issuing the new.

Revocation

Native applications offer an important new channel by which enterprise employees can access cloud services. But, because the access is enabled by the issuance of long-lived tokens to the native application, the channel complicates revocation of employee access to those services when their employment changes or ends. Unlike for web applications accessed through SSO, for which an employee’s access to SaaS applications can be controlled by simply no longer
issuing short-lived SAML SSO assertions, revoking the native application channel requires more explicit operation by the enterprise. More scalable than demanding an enterprise admin login to the SaaS control panel to revoke tokens is for the enterprise to push a ‘Remove User’ message to the appropriate cloud service providers using a protocol such as SCIM (Simple Cloud User Management). Upon receiving the message, each cloud service provider would revoke all access and refresh tokens previously issued to the user on various native applications.

An example of such a SCIM message is:

```
DELETE /User/uid=bjensen,dc=example,dc=com
Host: example.com
Authorization: Bearer h480djs93hd8
```

Whether accomplished administratively or automatically, once the access and refresh tokens were revoked, the native application would no longer be able to access the relevant APIs, nor obtain new access tokens.

**Other Considerations**

1. Native applications are currently enjoying great popularity, primarily because of the enhanced functionality relative to simple web applications. HTML5 promises to balance the situation. If so, web browser SSO will remain important, and continue to complement OAuth-based native application enablement.

2. While web and native application models were discussed here as distinct options, you can also combine them. For example, a native application embeds web content as in an Android WebView. Platforms like PhoneGap, which allow developers to author native applications using web technologies like HTML5 and CSS, get access to native APIs and app stores, highlight this scenario. These hybrids present interesting identity management challenges. For instance, if the containing native application is authorized (and issued an access token as a result), how would you translate that into the SAML SSO necessary to ensure seamless web access in the embedded browser?

3. For some, even if the OAuth authorization dance for a given native application is performed only rarely (as enabled by refresh tokens), the need to do this for each and every native application on a device is seen as an undesirable usability burden – especially if multiple native applications fall under the same domain such as separate native applications created by a electronics retailer, or the applications a given employee uses to interact with their SaaS services. In such situations, there can be value in a “native application SSO” experience, where the user authenticates and authorizes a single native application, and subsequently, appropriate other native applications need not have the user go through an explicit OAuth authorization dance but can instead benefit from that initial explicit authorization to obtain the desired tokens.
Best Practices

In the following sections, we discuss practical considerations of applying SAML and OAuth to enable mobile web and native applications, as well as presenting best practices for deployment.

Web Applications

Fundamentally, SSO works in mobile browsers the same way it does for desktop browsers—as redirects. However, mobile browsers may be more constrained in the maximum URL length they support, and so the SSO mechanisms that rely on passing messages within the URL query string, such as OpenID 2.0 and SAML HTTP Redirect binding, may not be appropriate. For such browsers, alternative SSO mechanisms like the SAML HTTP POST or Artifact bindings may be preferred.

Similarly, screen size limitations argue for appropriately designed IdP login pages, as well perhaps as SP selector UI. OpenID Connect defines a display parameter, with values of none, popup, or mobile, allowing the Client to guide the Authorization Server’s login and consent UI. OpenID Connect also mitigates the above mentioned URL length issue by adopting a data-by-reference model (similar to SAML’s artifact) and obviate the need for sending large data objects in the URL string through the browser.

For enterprise employee access to SaaS web applications, SSO provides control by revoking access when an employee is terminated. Once an employee leaves, the enterprise will no longer issue SAML assertions for that employee, blocking unauthorized access to cloud applications.

Native Applications

We consider different stages of the lifecycle for a native application – citing examples of how these stages are addressed in both iOS and Android.

Conclusion

Mobile clients are an increasingly important channel for consumers accessing Web 2.0 and enterprise employees accessing on-premise and cloud-hosted services. An identity management architecture for such clients must support the two broad categories—web applications delivered through the browser and native applications installed onto the device. The desired architecture does not impose different identities and authorizations for the two application models, but provides a single consistent and cohesive identity infrastructure for both. Critically, the architecture will be built on open industry standards to guarantee interoperability across websites, business partners, and SaaS providers and their customers. Fortunately, there exist identity management standards, SAML and OAuth, which, both in isolation and combination, can address the identity requirements of both application models.
References

1. OAuth and Android - Dropbox is wrong; never ask for passwords –
2. OAuth and REST in Android: Part 1 -
3. Android Client-side OAuth
   http://donpark.org/blog/2009/01/24/android-client-side-oauth
4. An OAuth Consumer implementation in Objective-C -
5. Single Sign-On for Desktop and Mobile Applications using SAML and OAuth
6. OpenID Connect
   www.openidconnect.com
7. Simple Cloud Identity Management (SCIM)
   http://www.simplecloud.info

About Ping Identity | The Cloud Identity Security Leader
Ping Identity provides cloud identity security solutions to the world’s foremost companies, government organizations and cloud businesses. For more information, dial U.S. toll-free 877.898.2905 or +1.303.468.2882, email sales@pingidentity.com or visit pingidentity.com.