Identity in the Platform - Thinking Beyond the Browser

Dirk Balfanz
Google

April 2011

Abstract

We discuss the benefits of an *Account Manager*, which is an operating system component that supplies authentication and authorization functionality to applications on the platform. As an example, we describe the Account Manager on the Android OS. The Account Manager provides benefits to both installed applications as well as the Web browser.

1 Introduction: Authentication for Installed Applications

Installed applications (also known as "native" applications or non-web applications) often need to access protected resources on servers. For example, a hypothetical MyTrips Android application might download personalized trip information for its users from the MyTrips servers.

In order to access such protected resources, the application needs to present evidence that it is allowed to do so. Today applications often use OAuth for this purpose. The OAuth protocol consists of two mechanisms: One mechanism describes how to obtain OAuth tokens (the "OAuth dance"), and another describes how to use OAuth tokens to access protected resources.

In this position paper, we argue that the former mechanism (the "OAuth dance") can beneficially be replaced by an operating-system call that allows applications to obtain OAuth tokens directly. The component in the operating system that provides the OAuth tokens is called an *Account Manager*. Account Managers hold user credentials that allow them to obtain OAuth tokens from OAuth providers. Since the particulars of how user credentials are exchanged for OAuth tokens may vary from provider to provider, Account Managers should employ a *plugin architecture*.

The Android OS has had an Account Manager from the beginning, and has used a plugin architecture since the Eclair release.

For example, the MyTrips Android application mentioned above might install an Account Manager plugin (called an *Authenticator*) into the Android operating system at installation time. The MyTrips Authenticator stores the user's MyTrips password, and can thus enable the MyTrips application (and, indeed, any other application on the device) to obtain OAuth tokens for accessing resources on MyTrips servers.

The MyTrips Authenticator plugin is responsible for obtaining user consent before returning OAuth tokens to third-party applications. The platform (in this case, the Android operating system) can assist in authenticating the third-party applications to the Authenticator.[1]

To summarize, the flow looks like this:

[1]Note that this is in contrast to using the "OAuth dance" with native applications, which doesn't allow for strong authentication of the native application during the OAuth dance.

- A third-party app requests an OAuth token for accessing protected resources at an OAuth service provider.
- That request is routed to the Account Manager, and includes both an indication of which Authenticator plugin is to be used, as well as an indication whose user's data at the OAuth service provider is to be accessed.
- The Account Manager forwards the request to the appropriate Authenticator.
- The Authenticator identifies the third-party application to the user and obtains user consent.
- The Authenticator looks up the user credentials of the account specified in the request, and obtains an OAuth token for that user (either my minting it itself, or - more likely - by retrieving it from the server),
- The Authenticator returns the OAuth token to the Account Manager, which returns the OAuth token to the third-party application.

Benefits of using an Account Manager over doing an "OAuth dance" include:

- Users never have to enter their credentials into a third-party application. (Instead, the Authenticator for the OAuth provider stores the user's credentials.)
- Applications never get access to powerful user credentials. Instead, they get access to scoped, short-lived credentials (OAuth2 access tokens).
- Application developers don't have to deal with presenting an approval screen and obtaining OAuth tokens from a browser or web view.

2 Single Sign-On with Account Managers

Let's assume that the MyTrips web application accepts federated logins from a number of Identity Providers, such as Google and Facebook. An Account Manager on, say, the Android platform can help with single sign-on to MyTrips through an IdP account. For this example, let's assume that the Account Manager has a Google Authenticator plugin installed, and that MyTrips accepts logins from Google.

The steps for single sign-on would be like this:

- A third-party app requests an OAuth token to access MyTrips servers, indicating a Google account that is provisioned on the phone (and for which the Google Authenticator plugin stores user credentials).
- The request is forwarded by the Account Manager to the Google Authenticator plugin.
- The Google Authenticator plugin obtains user consent for using the Google account to log into MyTrips.
- The Google Authenticator plugin obtains from Google servers an OAuth token that encodes OpenID Connect permissions for MyTrips.
- The OAuth token is returned to the third-party app, which uses it to access protected resources on MyTrips servers.

- Whenever a request for such protected resources is received, MyTrips servers decode the OpenID-Connect OAuth token to authenticate the user.

We're glossing over a few details here, which are currently being hashed out as part of the OpenID Connect proposal[2]. The gist is that OpenID connect allows an IdP to mint OAuth tokens that can be consumed by a relying party for the purposes of identifying a user, and that an Account Manager can be used to obtain such OAuth tokens.

3 Automated Browser Login with Account Managers

We can use Account Managers to automate login to web sites in browsers. The Android Web browser has such a feature since the Honeycomb release.

For this feature, we make three assumptions: One, the browser is aware of the existence of the Account Manager. Two, the Web server returns meta-data about the login flow to user-agents (in our case, we add an HTTP response header). Three, the server supports token-based login (*i.e.*, a user can be logged into the Web application by redirecting the browser to a special endpoint and passing a special OAuth token to that endpoint during the request).

The mechanism works as follows (explained by way of an example):

- When navigating to the MyTrips login page, the server sends meta data about the login flow (mostly information about the continue-URL the browser is supposed to go to after the user logs in). This login data is opaque to the browser, except for an indication of which Authenticator can handle the login request. In this example, the Authenticator is identified as "com.mytrips".

- The browser creates a request for a special kind of login token for the specified (MyTrips) Authenticator, and passes it, together with the opaque login data, to the Account Manager.

- The Account Manager passes the login data to the MyTrips Authenticator, which obtains the login token from its server, and passes it back to the browser on the device.

- By convention, the login token is, in fact, a URL pointing to a token-based login endpoint at MyTrips, complete with a token that causes the user to be logged in.

- When the browser receives the token back from the Account Manager, it follows the convention that the token itself is a URL, and simply redirects the browser to that URL, logging the user into www.mytrips.com.

The user experience is as follows: when the browser visits the MyTrips login page, it displays (in browser chrome) all the accounts managed by the com.mytrips Authenticator as being available for automated login. Once the user selects one of the accounts, the user is automatically logged in.

4 Web Single Sign-On with Account Managers

We can extend the above mechanism to allow for password-free federated web sign-on. Let's get back to the MyTrips example and assume that a user is visiting www.mytrips.com with a browser on a platform that

[2]`http://www.openidconnect.com`

has an Account Manager. Let's further assume that MyTrips accepts federated sign-ons from Google, and that the Google Authenticator on the platform has at least one account provisioned.

When the browser visits the MyTrips login page, the server not only responds with login meta data for the com.mytrips Authenticator, but also with login meta data for the com.google Authenticator. It obtains that meta data (which is opaque both to MyTrips as well as to the browser) from Google itself, specifying the desired return-to URL the browser should end up at after login (along with other data such as its OAuth client id). The meta data (which, again, is opaque both to MyTrips and to the browser, but not to Google), encodes a login flow that takes the user first to the token login endpoint, from there to Google's OpenID endpoint, and from there back to the MyTrips-provided return-to URL.

When the browser receives login meta data both for com.mytrips as well as com.google, it displays accounts managed by both Authenticators. If the user selects a Google account to log into www.mytrips.com, the Google Authenicator will obtain a login token that is a URL. That URL will point to Google's token login endpoint, with a continue-URL of Google's OpenID endpoint, and finally, with a continue-URL of MyTrips.

What happens next is that the browser will visit Google's token login endpoint, which logs in the user. After logging in the user, the endpoint will redirect the user-agent to Google's OpenID endpoint, where the user will see an OpenID consent page ("are you sure you want to login into mytrips.com with your Google account?"). If the user consents, Google prepares an OpenID assertion and finally redirects the browser to mytrips.com, where the user is logged in through OpenID. Note that in this scenario, mytrips.com didn't even have to host a "token login" endpoint - it merely had to be an OpenID relying party.

5 Conclusions

We've explained how Account Managers can eliminate password entry for native applications, by supplying such applications with OAuth tokens. We also explained how password-less Web login (both direct and federated) is made possible by Account Managers by minting special "login tokens" that take the form of a URL.

We have glossed over many details such as what form the user credentials that are managed by Authenticators take for users that use two factor to sign in, or are federated login users to the domain of the Authenticator itself.

The Android OS has included an Account Manager for a long time, and consequently authentication has traditionally not required re-entry of user credentials. Recently (in Honeycomb) we have added the ability for automated Web login, based the same Account Manager infrastructure.