Device Token Protocol for Persistent Authentication Shared Across Applications

John Trammel, Ümit Yalçınalp, Andrei Kalfas, James Boag, Dan Brotsky

Adobe Systems Incorporated, 345 Park Avenue, San Jose, USA
{jtrammel, lyalcina, akalfas, jboag, dbrotsky}adobe.com

Abstract: This paper describes a protocol for enabling shared persistent authentication for desktop applications that comprise a common suite by extending the OAuth2.0 protocol. OAuth2.0 is the de facto standard for developing and deploying federated Identity. Our extension enables the users to authenticate and authorize on devices that host a suite of applications that are connected to backend services and systems at Adobe. It is the backbone of our subscription and licensing infrastructure for the Adobe Creative Suite® 6 and Adobe Creative Cloud™. The extended protocol works without storing users credentials on a per application basis but rather uses device identities that are managed on a centralized server to enable increased security and management capabilities for a set of applications on the device. We describe the protocol is in detail, its inherent characteristics, how it extends OAuth2.0, and how it is used in practice.

Keywords: Authentication, Identity Management, OAuth2.0, Authorization

1 Introduction

1.1 Problem

Today, software clients and applications are available on heterogeneous devices, including tablets, smart phones, and desktops. In addition, in a world where companies are beginning to develop and deploy multiple applications, user access should be accomplished in a manner that is both user-friendly and secure, and also recognizes that specific context; the user should not be required to enter separate user authentications for each application from the same vendor on a particular device. We faced this issue when we needed to deploy subscription services to multiple applications that were offered on the desktop that comprised Adobe Creative Suite and Adobe Creative Cloud membership.

In today's mobile applications, each application manages access control for the user and it is configured separately. As new applications are added, access for the user is configured anew for each application. Furthermore, a user's credentials are typically stored on the device via encryption rather than by identifying the device and the authorizations of the specific user for the device.

There are many applications in the industry that are currently use stored user credentials for authentication and authorization. Examples include email access from different vendors on the iPhone, twitter account settings on iOS or Android devices, etc. These approaches require user credentials to be stored on a per application basis.

When there are multiple applications from the same vendor on the same device, this approach becomes cumbersome and error prone. Although the user may be governed by a centralized identity service that may also support Federated Identity, security on the device can only achieved by replicating user credentials on the device on a per application basis. This is clearly not desirable.

Without a centralized mechanism on the client, a management problem occurs when the user, in a heterogeneous landscape, needs access to more devices and applications. This is a problem space that other vendors have begun to recognize: Google Application Manager on the Android platform [4] has centralized account management in a single source in the client machine. However, that implementation still stores user credentials on the client.

This leaves related areas open for further improvement: security and remote management. Mobile devices may be broken, stolen, or lost. Therefore, the ability to remotely manage and revoke authorizations on mobile devices is highly desirable to reduce data security risks. In addition, companies need to be able to terminate access for users, perhaps due to personnel changes, from a central server and have that change take affect on all associated remote devices.

Note that several web and mobile applications keep track of user authentication from the specific device. For example, Facebook and Bank of America Online Banking [5] keep track of the specific devices that the user authenticates from in their applications. Thus, device tracking is important to utilize for the added security and management, but it has not been fully utilized in conjunction with decoupling user credentials from the authentication and authorization process.

The OAuth2.0 Protocol [1] has become widely used for granting access to clients of services and applications. However, a persistent token that decouples users credentials, and that can cater to multiple client applications, rather than a single one, has not been targeted by this protocol in the state of the art, literature or standardization.

1.2 Solution:

We developed a novel mechanism, a **persistent device token**, for securely persisting authentication for a user that can be used *by a set of authorized applications on a device* (laptop, desktop, mobile phone, tablet, etc.).

The device token is unique to a specific device and user and can be used by one or many applications. It can be persisted, but it is non-transferable, meaning that it cannot be transferred to another device or to another user. This mechanism

- is secure (because a device token is usable only on the device and by the user for which it was issued.)
- does not require the persistence of user credentials on the device.

- enables revocation of authorizations remotely – outside of the presence of the device on a centralized service that is dedicated to identity management.
- is configurable because it allows for server side control over which client applications can make use of the mechanism.

An implementation of this concept has been developed and delivered as an extension to the OAuth2 [1] + OpenID-Connect protocol [2], specifically as a new **grant type.** We assume that the reader is familiar with the OAuth2.0 [1] protocol reading this paper. This extension to OAuth2 and OpenID-Connect is currently in use within the Adobe Creative Suite product line and Adobe Creative Cloud**.**

The extension presented does not currently exist in OAuth2 or OpenID-Connect protocols, as most applications ARE NOT configured like a suite or a group of applications on a device basis. However, this need is rapidly emerging as companies like Adobe begin developing and deploying multiple applications for its constituent user bases on devices.

Today, almost all Adobe software that connects to Adobe web services use a backend service called IMS (Identity Management Service). IMS centralizes the workflows for all clients that require authentication and authorization to various Adobe hosted services and it provides federated identity support. IMS supports OAuth2.0 [1] and OpenID- Connect Protocol [2]. IMS also provides centralized common UI workflows that are targeted to clients on specific device types, such as desktop, browser, mobile device, etc.

The protocol presented in this paper is currently being deployed worldwide as part of the Adobe Creative Suite 6.0 and Adobe Creative Cloud [6] offerings on the desktop. It is integrated with the Adobe Application Manager that manages single sign on, device tokens and user authorization with IMS on the desktop for all client applications and extensions that reside on the desktop.

2 Detailed Protocol:

2.1 Overview:

This protocol has the following characteristics and behaviors:

1. A new grant type, named Device Token is introduced. This grant type is similar to the authorization grant type, with a key difference: while the authorization code is bound to a specific client-id, the device token is bound to a specific device and a user and may be used to create access tokens for authorized clients applications on the same device. Note that there are many applications on the same device where each has a different client_id.
2. This grant type is most appropriate for client applications that reside locally on the device (native, AIR, etc.), and not appropriate for server back-ends or web-apps that are intended to run in a system-browser.

The client-application must adhere to specific requirements in the system:

a. Client applications must be specifically authorized to use the device token grant type and be configured in advance of deployment.
b. Client applications must share secure access to and generation algorithms for the device-identifier.
c. Client applications must generate the device-identifier via an algorithm, shared by the coordinating applications, each time the application launches or intends to use the device token.
d. Client applications must not persist the device-identifier in its final form.

3. The following recommendations apply to the generation of device-identifiers for use on various platforms/environments. These recommendations do not represent the actual implementation details of Adobe's applications, but rather are illustrative descriptions, appropriate for publication:

Part	Description
platform	A code specific for the platform (ios, android, win, mac, linux)
user identifier	A user id that uniquely identifies a user on the specific platform. Needed for devices that allow multiple users, where there is a need to limit device token portability between users on the same device.
profile identifier	A unique identifier for a profile. Users may have multiple profiles where there is need to distinguish different patterns and entitlements of use, such as home vs. work. Needed for devices that allow multiple users when there is a need to limit device token portability between groups/profiles on the same device.
System device identifier	The device identifier of the specific platform. OS or system appropriate value.

An excerpt from early documentation for the device token protocol follows and refers to the accompanying flow diagram. It has been edited to remove some details deemed confidential or not relevant. Terminology from OAuth2.0 is used in the flow diagram in Figure 1 and the descriptions of the steps involved. In the detailed API documentation that follows, the protocol described builds on top of the specific calls of OAuth2.0 [1].

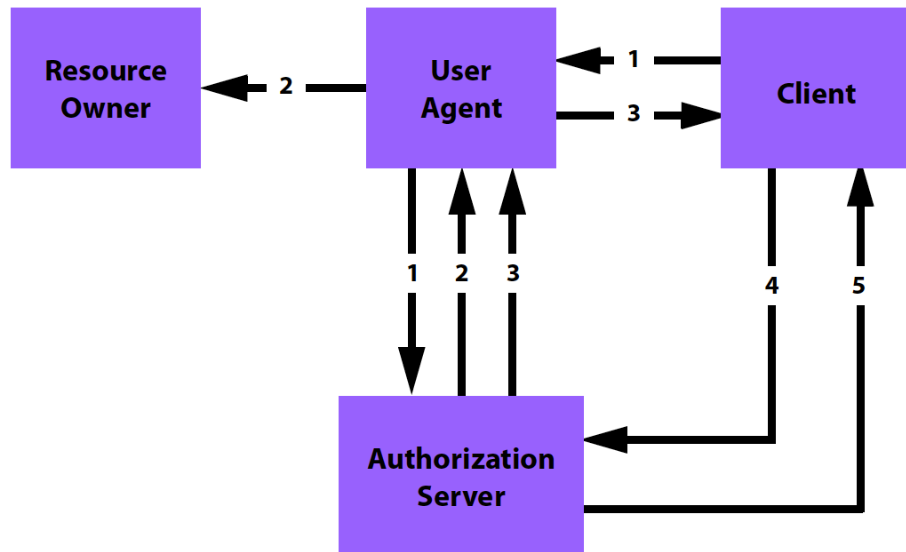

Fig. 1. Device Token Flow Diagram with OAuth2.0

1. The client initiates the flow by directing the resource owner's user-agent to the device authorization endpoint (at the Authorization Server). The client includes its device identifier, client identifier, requested scope, local state, and a redirection URI to which the authorization server will send the user-agent back once access is granted.
2. The authorization server authenticates the resource owner (via the user-agent). The user has the option to persist data on the device. If the user agrees to save data on the device, the authorization server releases a device token, otherwise it returns an authorization code.
3. Assuming the authentication is successful, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier. Depending on the user's consent, the redirection URI includes either a device token or an authorization code and any local state provided earlier by the client.
4. The client, providing either a device token or an authorization code, requests an access token from the authorization server. This transaction requires the client to authenticate.
5. The authorization server validates the client credentials and the device token or authorization code. If valid, it responds back with an access token.

In this flow, the authorization server depends on the user's consent to store a device token on the client device. This consent determines whether the device token would be used subsequently for authentication instead of an authorization code and user credentials. This is illustrated in the UI flow provided later in this document be-

low. If the user does not consent to store a device token on the device, an authorization code is used to get an access token; this authorization code is good for only one request and must not be persisted.

Each client application must be registered with a unique identifier called **client_id**. This unique identifier is given at the time of configuration of the application prior to deployment so that the authorization server at run time can uniquely identify the client with its client_id. In addition, to securely transmit each client's request, each client is configured with a client_secret at the time of registration.

A user authenticated using this protocol may have a profile and this protocol may also transmit additional parameters that are specified for this user using a scope. For example, OpenID may be specified to get these specific parameters. For more information on OpenID Connect profiles, see the reference [2].

2.2 Detailed Protocol:

Device Token Request: The client constructs the request URI by adding the following parameters to the query component of the authorization endpoint URI using the `"application/x-www-form-urlencoded"`.

Parameter	Mandatory	Description
response_type	true	Must be "device"
device_id	true	It is up to clients to generate and provide the device ID. Refer to SHA256 [3].
device_name	false	If specified, can be used to present to the user a user-friendly name of the device.
redirect_uri	false	If missing, the server will use the default redirect URI that is provisioned when the client was registered during the configuration.
client_id	true	The client identifier that is provided for the application during the registration phase.
scope	true	The scope of the access request expressed as a list of comma-delimited, case sensitive strings. *Details of scope parameter values not presented.*
locale	false	The locale to be used in the user interface, supplied in the format *language_country*. Default is en_US.
state	false	*Details around usage of state parameter not presented.*
dc	false	*Details around usage of dc parameter not presented*

Device Token Response: If the user agrees to persist data on the device, the server issues a device token and delivers it to the client by adding the following parameters to the query component of the redirection URI using the `"application/x-www-form-urlencoded"` format:

Parameter	Mandatory	Description
device_token	true	The device token bound to the device.
state	false	If present in the device token request

If the user did not agree to persisting data on the device, IMS will fallback to releasing an authorization code. The query component of the redirection URI using the `"application/x-www-form-urlencoded"` will contain:

Parameter	Mandatory	Description
code	true	The authorization code.
state	false	If present in the device token request

Error Response: If the client identifier provided is invalid, the server informs the resource owner of the error and does not redirect the user-agent anywhere. If the request fails for another reason, the server informs the client by adding the following parameters to the query component of the redirection URI using the `"application/x-www-form-urlencoded"` format:

Parameter	Mandatory	Description
error	true	A single error code with the values from bellow.
error_description	false	Additional information about the error.

error_code	Description
access_denied	If user did not authorize the client application. For instance this can happen when the user clicks on the Cancel button in the login screen.
access_denied_no_cookies	If the server detects that cookies are disabled.

Example: In the examples below, the tokens are abbreviated for clarity and designated by mnemonics, such as <DEV_TOKEN>.

Device Token Request.

```
GET
/ims/authorize/v1?client_id=AXX_YYY&response_type=device&
&device_id=MA4Y2KfwV1av8soWHoOnmubOiFWhXOg-
nwePp9dExqU&device_name=Mac&redirect_uri=http%3A%2F%2Fsto
phere.adobe.com&scope=openid HTTP/1.1
Host: ims-host.adobelogin.com
```

Device Token Response with device token.

```
HTTP/1.1 302 Moved Temporarily
Server: Apache-Coyote/1.1
Set-Cookie: relay=0526317f-3e77-46c1-8b19-957a06a9b2e8;
Path=/
Cache-Control: no-store
P3P: CP="IDC DSP COR CURa ADMa OUR IND PHY ONL COM STA"
X-RHH: B8E1750B964EE62AB7C147F0EDF12803
Location:
http://stophere.adobe.com?device_token=<DEV_TOKEN>
Content-Type: text/html;charset=UTF-8
Content-Language: en-US
Transfer-Encoding: chunked
Content-Encoding: gzip
Vary: Accept-Encoding
Date: Mon, 14 Nov 2011 12:50:01 GMT
```

Device Token Response with authorization code

```
HTTP/1.1 302 Moved Temporarily
Server: Apache-Coyote/1.1
Set-Cookie: relay=08da1a84-9179-4720-95c5-c871fdc69063;
Path=/
Cache-Control: no-store
P3P: CP="IDC DSP COR CURa ADMa OUR IND PHY ONL COM STA"
X-RHH: B8E1750B964EE62AB7C147F0EDF12803
Location: http://stophere.adobe.com?code=<AUTHR_CODE>
Content-Type: text/html;charset=UTF-8
Content-Language: en-US
Transfer-Encoding: chunked
Content-Encoding: gzip
Vary: Accept-Encoding
Date: Mon, 14 Nov 2011 12:50:14 GMT
```

Access Token Request with a Device Token: The client makes a request to the token endpoint by adding the following parameter using the "`application/x-www-form-urlencoded`" format in the HTTP request entity-body:

Parameter	Mandatory	Description
grant_type	true	Must be “device”
device_id	true	It is up to clients to generate and provide the device ID. Refer to SHA256 [3].
device_token	true	The device token received at the previous step.
client_id	true	The client_id credential received during the registration phase.

client_secret	true	The client_secret credential received during the registration phase.

Access Token Response: The server issues an access token and a refresh token, and constructs the response by adding the following parameters to the entity body of the HTTP response with a 200 (OK) status code.

Parameter	Mandatory	Description
access_token	true	The access token
refresh_token	true	The refresh token
expires_in	true	The lifetime in milliseconds of the access token. For example, the value "360000" denotes that the access token will expire in one hour from the time the response was generated.

In addition to the above parameters, the server will include in its response attributes from the user's profile based on the requested scope as specified in the Device Token Request. The parameters may be included in the entity body of the HTTP response using the appropriate media type. For example "`application/json`" media type will serialize the parameters into a JSON structure.

Error Response: The server responds with an HTTP 400 (Bad Request) status code and includes the following parameters with the response:

Parameter	Mandatory	Description
error	true	A single error code with the values from below

error code	Description
invalid request	If creating an access token fails due to internal errors.
invalid_client	If the client credentials are not correct.
unsupported_grant_type	If the client_id does not have the appropriate grant_type set.
access_denied	If the device token is invalid or if it was released for a different device id.

Example

Access Token Request with a device token

Access Token Request with Device Token

```
POST /ims/token/v1 HTTP/1.1
User-Agent: curl/7.21.4 (universal-apple-darwin11.0) lib-
curl/7.21.4 OpenSSL/0.9.8r zlib/1.2.5
Host: ims-host.adobelogin.com
Accept: */*
Content-Length: 740
Content-Type: application/x-www-form-urlencoded
grant_type=device&device_id=MA4Y2KfwV1av8soWHoOnmubOiFWhX
Og-
nwePp9dExqU&device_token=<DEV_TOKEN>&client_id=<YOUR_CLIE
NT_ID>&client_secret=<CLIENT_SECRET>
```

Access Token Response

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: relay=0c122e22-58be-4c5d-b215-d362b9142c71;
Path=/
Cache-Control: no-store
P3P: CP="IDC DSP COR CURa ADMa OUR IND PHY ONL COM STA"
X-RHH: B8E1750B964EE62AB7C147F0EDF12803
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Mon, 14 Nov 2011 13:17:56 GMT
{"to-
ken_type":"bearer","expires_in":86399952,"refresh_token":
"REFRESH_TOKEN","access_token":"ACCESS_TOKEN"}
```

Access Token Request with an authorization code: The client makes a request to the token endpoint by adding the following parameter using the "application/x-www-form-urlencoded" format in the HTTP request entity-body:

Parameter	**Man-datory**	**Description**
grant_type	true	Must be “authorization_code”.
code	true	The authorization code previously received.
client_id	true	The client_id credential received during the registration phase.
client_secret	true	The client_secret credential received during the registration phase.

The server validates the client credentials and ensures that the authorization code was issued to that client. Note that an authorization code may be used only once but the device token can be stored and reused to grant access.

Example: With an authorization code a client can request an access token as follows:

```
POST /ims/token/v1 HTTP/1.1
Host: ims-na1-dev1.adobelogin.com
Content-Type: application/x-www-form-urlencoded
grant_type=authorization_code&client_id=THE_CLIENT_ID&cli
ent_secret=THE_CLIENT_SECRET&code=AUTHR_CODE
```

Access Token Response: IMS issues an access token and a refresh token, and constructs the response by adding the following parameters to the entity body of the HTTP response with a 200 (OK) status code.

Parameter	Mandatory	Description
access_token	true	The access token.
refresh_token	true	The refresh token.
expires_in	true	The lifetime in milliseconds of the access token. For example, the value "360000" denotes that it will expire in one hour from the time the response was generated.

In addition to the above parameters, the server may include in its response attributes from the user's profile based on the requested scope specified within the Device Token Request. Again, the parameters are included in the entity body of the HTTP response using the specific media-type, such as `"application/json"` which will serialize the parameters into a JSON structure.

IMS will check the client credentials and the authorization code and if they are OK it will send back a response.

Access Token Error Response: IMS responds with an HTTP 400 (Bad Request) status code and includes the following parameters with the response:

Parameter	Mandatory	Description
error	true	A single error code with the values from below.
error_description	false	Additional information about the error.

error_code	Description

invalid_request	If creating an access token fails due to internal errors
invalid_client	If the client_id was not provisioned or the client_secret does not match.
unauthorized_client	If the client_id does not have the appropriate grant_type set.
access_denied	If the authorization code / refresh token is not valid or the authorization code was released to a different client_id.

2.3 Persistence of Device Tokens:

There are two different approaches in persisting device tokens on a device:

1) A client application may store the device token in the client application or user specific storage, akin to how many applications store user credentials today (e.g., Mac OS X Keychain). Since this approach requires each application to implement the protocol itself, it would require duplicative, coordinated implementations if used by multiple applications.
2) Another alternate is to develop a separate account management library that manages device tokens and related authentication tokens for a series of applications. This library is responsible for storing the device token on behalf of the user as well as generating authentication tokens for specific applications that need authentication and authorization. Basically, the client and the user-agent in the workflow are coupled and encapsulated in a library that allows multiple applications to use the same mechanism on the device. This approach decouples the applications from security and persistence concerns, while bringing the benefits of security and manageability.

There are use cases when is it desirable to disallow the storage of a local device token. The ability to enable local storage of a device token is configurable. If the user is allowed to store a device token locally, that option can be presented in the following manner shown in Figure 2.

Fig. 2. Signing In when Local Device Token Storage Enabled

In use cases where the device tokens will always be stored locally, the user can be informed of this via a user experience as exemplified in Fig. 3.

In the case when third party identity providers (such as Google, Yahoo or Facebook) handle authentication, the login UI is fixed and cannot be modified to prompt for device token storage. In these cases the server will show an *interstitial* page asking for consent to store the tokens locally. This is exemplified in Figure 4.

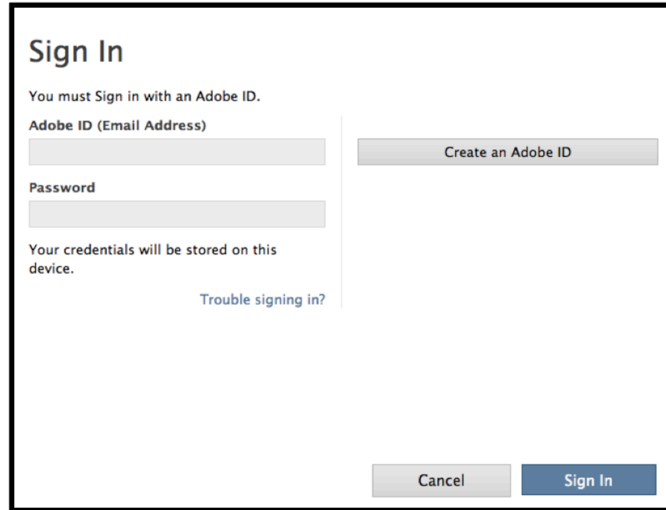

Fig. 3. Mandatory persistence on device

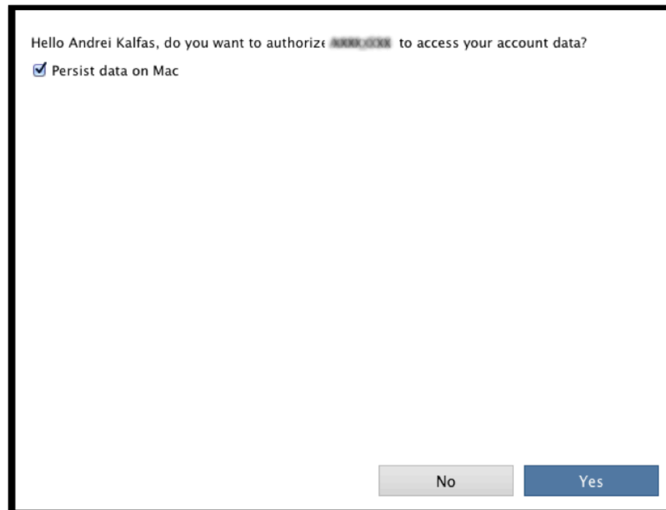

Fig. 4. Interstitial with third party providers

3 Conclusion:

We presented a novel extension to the OAuth2.0 protocol using device tokens and a new grant type to authenticate and authorize a user for multiple applications on a single device. A device token is not transferable but is persistable.

Our approach has the following advantages:

- Enhanced Security. User credentials are never stored on the device and the device token cannot be used on other devices, or for other users on the same device.
- Permanent and Revocable Authentication: The solution does not require the user to re-authenticate and re-authorize on a per client (application) basis unless changes occur that require re-approval of the user, such as new terms of use that need to be agreed to, etc.
- Unified authentication and authorization experience: The solution provides a single way of handling of authentication and authorization for multiple applications.
- Scalability of Deployment: Additional applications from a single vendor can be added to a user's device without requiring them to provide credentials for each additional application. Which client applications are associated with which device tokens can even be changed after deployment since they are managed from a central server.
- Remote Management: Tokens are remotely revocable. This makes possible the management of all devices that are permitted to run applications for a specific user on a centralized server.

The solution works for a single application as well as a collection of applications. The security (not storing credentials on the client and non-transferable device tokens) and manageability benefits apply to both configurations.

The protocol discussed in this paper is being deployed along with the Adobe Creative Suite 6.0 and Adobe Creative Cloud.

Acknowledgements:

The original version of this paper is published by Springer-Verlag in *Service-Oriented and Cloud Computing, Proceedings of First European Conference, ESOCC 2012*, Bertinoro, Italy, September 19-21, 2012, *Lecture Notes in Computer Science, Volume 7592, 2012, pp 230-243* [8]. The original publication is available at http://www.springerlink.com.

The Adobe Creative Suite, Adobe Create Cloud and IMS are corporate efforts. We thank the members of the IMSLib, OOBE and CEP teams in their endless efforts in developing, debugging and testing the client library that enables all applications that are managed by Adobe Application Manager on the desktop; the IMS team for supporting this protocol addition with IMS. We also thank the Business Architecture team in reviewing and making comments to the drafts of this paper, in particular Lois Gerber, Bob Murata, Shyama Padhi and Chris Tuller.

4 References:

1. OAuth Working Group, Hammer E. (ed), IETF, The OAuth2.0 Authorization Protocol draft 28, http://tools.ietf.org/html/draft-ietf-oauth-v2-28, (2012)
2. OpenId Foundation, Open ID Connect Protocol Suite : http://openid.net/connect/ (2012)
3. IPSec Working Group, Frankel S., Kelly S., The HMAC-SHA-256-128 Algorithm and Its Use With IPsec: http://w3.antd.nist.gov/iip_pubs/draft-ietf-ipsec-ciph-sha-256-01.txt (2002)
4. Google, Android Account Manager API: http://developer.android.com/reference/android/accounts/AccountManager.html, (2012)
5. Bank of America, Online Banking FAQ, http://www.bankofamerica.com/onlinebanking/index.cfm?template=site_key - accessolb (2012)
6. Adobe Creative Cloud™ http://creative.adobe.com (2012)
7. Adobe Creative Suite® http://www.adobe.com/products/creativesuite.html (2012)
8. Service-Oriented and Cloud Computing, Proceedings of First European Conference, ESOCC 2012, Bertinoro, Italy, September 19-21, 2012, *Lecture Notes in Computer Science, Volume 7592, 2012, pp 230-243,* http://rd.springer.com/chapter/10.1007/978-3-642-33427-6_20