OAuth Working Group W. Denniss
Internet-Draft Google
Intended status: Best Current Practice J. Bradley
Expires: August 7, 2016 Ping Identity
February 04, 2016

OAuth 2.0 for Native Apps
draft-wdenniss-oauth-native-apps-02

Abstract

OAuth 2.0 authorization requests from native apps should only be made through external user-agents such as the system browser (including via an in-app browser tab). This specification details the security and usability reasons why this is the case, and how native apps and authorization servers can implement this best practice.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 7, 2016.

Copyright Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 2
1.1. Notational Conventions 3
1.2. Terminology 3
1.3. Overview 4
2. Using Inter-app URI Communication for OAuth 6
3. Initiating the Authorization Request 6
4. Receiving the Authorization Response 7
4.1. App-declared Custom URI Scheme Redirection 7
4.2. App-claimed HTTPS URI Redirection 9
4.3. Localhost-based URI Redirection 9
5. Security Considerations 10
5.1. Embedded User-Agents 10
5.2. Protecting the Authorization Code 11
5.3. Phishing 12
5.4. Limitations of Non-verifiable Clients 12
6. Other External User Agents 12
7. Client Authentication 13
8. References 13
8.1. Normative References 13
8.2. Informative References 13
Appendix A. Operating System Specific Implementation Details . . 15
A.1. iOS Implementation Details 15
A.2. Android Implementation Details 15
Appendix B. Acknowledgements 15
Authors' Addresses 16

1. Introduction

The OAuth 2.0 [RFC6749] authorization framework, documents two approaches in Section 9 for native apps to interact with the authorization endpoint: via an embedded user-agent, or an external user-agent.

This document recommends external user-agents like in-app browser tabs as the only secure and usable choice for OAuth. It documents how native apps can implement authorization flows with such agents, and the additional requirements of authorization servers needed to support such usage.

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in Key words for use in RFCs to Indicate Requirement Levels [RFC2119]. If these words are used without being spelled in uppercase then they are to be interpreted with their normal natural language meanings.

1.2. Terminology

In addition to the terms defined in referenced specifications, this document uses the following terms:

"app" A native application, such as one on a mobile device or desktop operating system.

"app store" An ecommerce store where users can download and purchase apps. Typically with quality-control measures to protect users from malicious developers.

"system browser" The operating system's default browser, typically pre-installed as part of the operating system, or installed and set as default by the user.

"browser tab" An open page of the system browser. Browser typically have multiple "tabs" representing various open pages.

"in-app browser tab" A full page browser with limited navigation capabilities that is displayed inside a host app, but retains the full security properties and authentication state of the system browser. Has different platform-specific product names, such as SFSafariViewController on iOS 9, and Chrome Custom Tab on Android.

"Claimed HTTPS URL" Some platforms allow apps to claim a domain name by hosting a file that proves the link between site and app.

Typically this means that URLs opened by the system will be opened in the app instead of the browser.

"web-view" A web browser UI component that can be embedded in apps to render web pages, used to create embedded user-agents.

"reverse domain name notation" A naming convention based on the domain name system, but where where the domain components are reversed, for example "app.example.com" becomes "com.example.app".

"custom URI scheme" A URI scheme (as defined by [RFC3986]) that the app creates and registers with the OS (and is not a standard URI

scheme like "https:" or "tel:"). Requests to such a scheme results in the app which registered it being launched by the OS. For example, "myapp:", "com.example.myapp:" are both custom URI schemes.

"inter-app communication" Communication between two apps on a device.

"OAuth" In this document, OAuth refers to OAuth 2.0 [RFC6749].

1.3. Overview

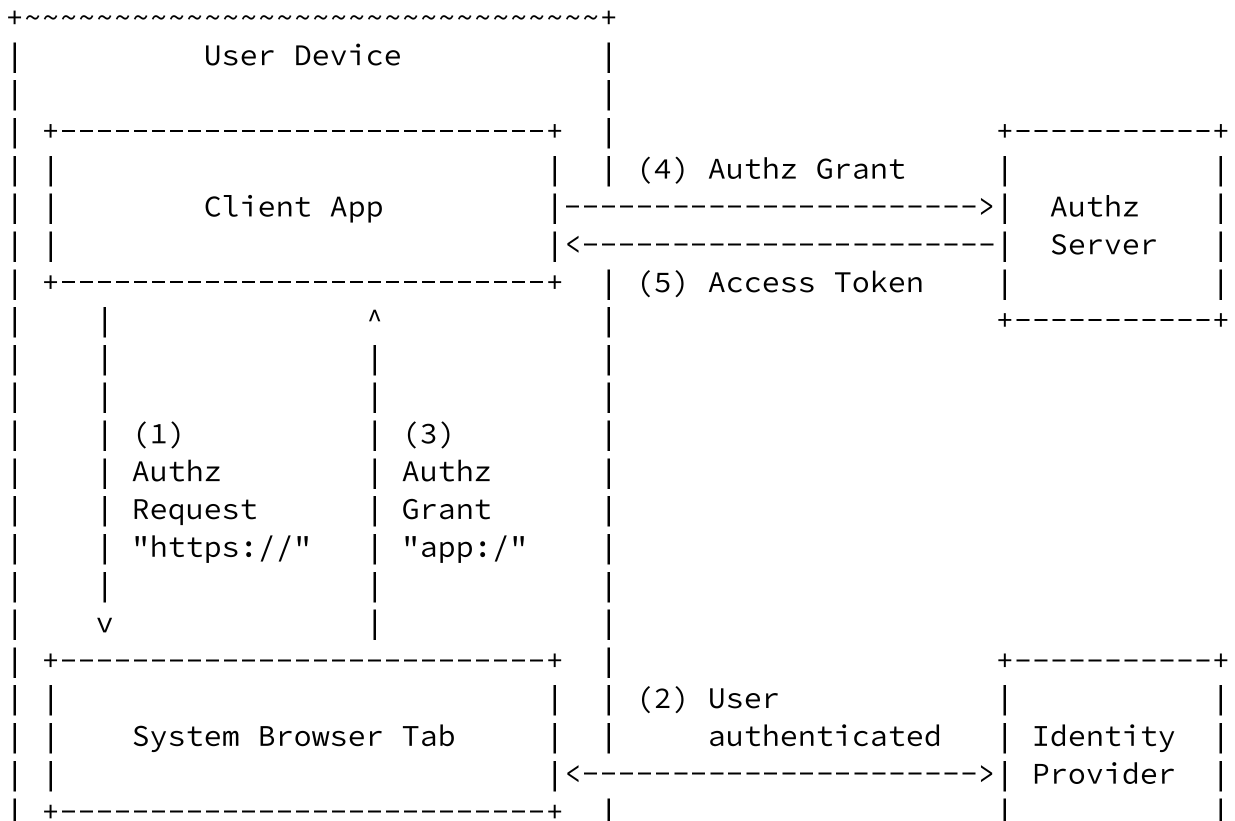
At the time of writing, many native apps are still using web-views, a type of embedded user-agent, for OAuth. That approach has multiple drawbacks, including the client app being able to eavesdrop user credentials, and is a suboptimal user experience as the authentication session can't be shared, and users need to sign-in to each app separately.

OAuth flows between a native app and the system browser (or another external user-agent) are more secure, and take advantage of the shared authentication state to enable single sign-on. The in-app browser tab pattern makes this approach even more viable, as apps can present the system browser without the user switching context something that could previously only be achieved by a web-view on most platforms.

Inter-process communication, such as OAuth flows between a native app and the system browser can be achieved through URI-based

communication. As this is exactly how OAuth works for web-based OAuth flows between RP and IDP websites, OAuth can be used for native app auth with very little modification.

1.3.1. Authorization Flow for Native Apps



```
   |                                 |                 +-----------+
   +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
```

Figure 1: Native App Authorization via External User-agent

Figure 1 illustrates the interaction of the native app with the system browser to authorize the user via an external user-agent.

1) The client app opens a system browser with the authorization request (e.g. https://idp.example.com/oauth2/auth...)

2) Server authenticates the end-user, potentially chaining to another authentication system, and issues Authorization Code Grant on success

3) Browser switches focus back to the client app using a URI with a custom scheme or claimed HTTPS URL, passing the code as a URI parameter.

4) Client presents the OAuth 2.0 authorization code and PKCE [RFC7636] proof of possession verifier.

5) Server issues the tokens requested.

2. Using Inter-app URI Communication for OAuth

Just as URIs are used for OAuth 2.0 [RFC6749] on the web to initiate the authorization request and return the authorization response to the requesting website, URIs can be used by native apps to initiate the authorization request in the device's system browser and return the response to the requesting native app.

By applying the same principles from the web to native apps, we gain similar benefits like the usability of a single sign-on session, and the security by a separate authentication context. It also reduces the implementation complexity by reusing the same flows as the web, and increases interoperability by relying on standards-based web flows that are not specific to a particular platform.

It is RECOMMENDED that native apps use the URI-based communication functionality of the operating system to perform OAuth flows in an external user-agent, typically the system browser.

For usability, it is RECOMMENDED that native apps perform OAuth using the system browser by presenting an in-app browser tab where possible. This affords the benefits of the system browser, while allowing the user to remain in the app.

It is possible to create an external user-agent for OAuth that is a native app provided by the authorization server, as opposed to the system browser. This approach shares a lot of similarity with using the system browser as both use URIs for inter-app communication and is able to provide a secure, shared authentication session, and thus MAY be used for secure native OAuth, applying most of the techniques described here. However it is NOT RECOMMENDED due to the increased complexity and requirement for the user to have the AS app installed. While much of the advice and security considerations are applicable to such clients, they are out of scope for this specification.

3. Initiating the Authorization Request

The authorization request is created as per OAuth 2.0 [RFC6749], and opened in the system browser. Where the operating system supports in-app browser tabs, those should be preferred over switching to the system browser, to improve usability.

The function of the redirect URI for a native app authorization request is similar to that of a web-based authorization request. Rather than returning the authorization code to the OAuth client's server, it returns it to the native app. The various options for a redirect URI that will return the code to the native app are

documented in Section 4. Any redirect URI that allows the app to receive the URI and inspect its parameters is viable.

4. Receiving the Authorization Response

There are three main approaches to redirection URIs for native apps: custom URI schemes, app-claimed HTTP URI schemes, and http://localhost redirects.

4.1. App-declared Custom URI Scheme Redirection

Most major mobile and desktop computing platforms support inter-app communication via URIs by allowing apps to register custom URI schemes. When the system browser or another app attempts to follow a URI with a custom scheme, the app that registered it is launched to handle the request. This document is only relevant on platforms that support this pattern.

In particular, the custom URI scheme pattern is supported on the mobile platforms Android [Android.URIScheme], iOS [iOS.URIScheme], and Windows Phone [WindowsPhone.URIScheme]. Desktop operating systems Windows [Windows.URIScheme] and OS X [OSX.URIScheme] also support custom URI schemes.

4.1.1. Using Custom URI Schemes for Redirection

To perform an OAuth 2.0 Authorization Request on a supported platform, the native app launches the system browser with a normal OAuth 2.0 Authorization Request, but provides a redirection URI that utilizes a custom URI scheme that is registered by the calling app.

When the authentication server completes the request, it redirects to the client's redirection URI like it would any redirect URI, but as the redirection URI uses a custom scheme, this results in the OS launching the native app passing in the URI. The native app extracts the code from the query parameters from the URI just like a web client would, and exchanges the Authorization Code like a regular OAuth 2.0 client.

4.1.2. Custom URI Scheme Namespace Considerations

When selecting which URI scheme to associate with the app, apps SHOULD pick a scheme that is globally unique, and which they can assert ownership over.

To avoid clashing with existing schemes in use, using a scheme that follows the reverse domain name pattern applied to a domain under the app publishers control is RECOMMENDED. Such a scheme can be based on

a domain they control, or the OAuth client identifier in cases where

the authorization server issues client identifiers that are also valid DNS subdomains. The chosen scheme MUST NOT clash with any IANA registered scheme [IANA.URISchemes]. You SHOULD also ensure that no other app by the same publisher uses the same scheme.

Schemes using reverse domain name notation are hardened against collision. They are unlikely to clash with an officially registered scheme [IANA.URISchemes] or unregistered de-facto scheme, as these generally don't include a period character, and are unlikely to match your domain name in any case. They are guaranteed not to clash with any OAuth client following these naming guidelines in full.

Some platforms use globally unique bundle or package names that follow the reverse domain name notation pattern. In these cases, the app SHOULD register that bundle id as the custom scheme. If an app has a bundle id or package name that doesn't match a domain name under the control of the app, the app SHOULD NOT register that as a scheme, and instead create a URI scheme based off one of their domain names.

For example, an app whose publisher owns the top level domain name "example.com" can register "com.example.app:/" as their custom scheme. An app whose authorization server issues client identifiers that are also valid domain names, for example "client1234.usercontent.idp.com", can use the reverse domain name notation of that domain as the scheme, i.e. "com.idp.usercontent.client1234:/". Each of these examples are URI schemes which are likely to be unique, and where the publisher can assert ownership.

As a counter-example, using a simple custom scheme like "myapp:/" is not guaranteed to be unique and is NOT RECOMMENDED.

In addition to uniqueness, basing the URI scheme off a name that is under the control of the app's publisher can help to prove ownership in the event of a dispute where two apps register the same custom scheme (such as if an app is acting maliciously). For example, if two apps registered "com.example.app:", the true owner of "example.com" could petition the app store operator to remove the counterfeit app. This petition is harder to prove if a generic URI scheme was chosen.

4.1.3. Registration of App Redirection URIs

As recommended in Section 3.1.2.2 of OAuth 2.0 [RFC6749], the authorization server SHOULD require the client to pre-register the

redirection URI. This remains true for app redirection URIs that use custom schemes.

Additionally, authorization servers MAY request the inclusion of other platform-specific information, such as the app package or bundle name, or other information used to associate the app that may be useful for verifying the calling app's identity, on operating systems that support such functions.

Authorizations servers SHOULD support the ability for native apps to register Redirection URIs that utilize custom URI schemes. Authorization servers SHOULD enforce the recommendation in Section 4.1.2 that apps follow naming guidelines for URI schemes.

4.2. App-claimed HTTPS URI Redirection

Some operating systems allow apps to claim HTTPS URLs of their domains. When the browser sees such a claimed URL, instead of the page being loaded in the browser, the native app is launched instead with the URL given as input.

Where the operating environment provided app-claimed HTTPS URIs in a usable fashion, these URIs should be used as the OAuth redirect, as they allow the identity of the destination app to be guaranteed by the operating system.

Apps on platforms that allow the user to disable this functionality, present it in a user-unfriendly way, or lack it altogether MUST fallback to using custom URI schemes.

The authorization server MUST allow the registration of HTTPS redirect URIs for non-confidential native clients to support app-claimed HTTPS redirect URIs.

4.3. Localhost-based URI Redirection

More applicable to desktop operating systems, some environments allow the app to create a local server and listen for redirect URIs that. This is an acceptable redirect URI choice for native apps on compatible platforms.

Authorization servers SHOULD support redirect URIs on the localhost host, and HTTP scheme, that is redirect URIs beginning with http://localhost (NB. in this case, HTTP is acceptable, as the request never leaves the device).

When an app is registered with such a redirect, it SHOULD be able to specify any port in the authorization request, meaning that a request with http://localhost:*/* as the redirect should be considered valid.

5. Security Considerations

5.1. Embedded User-Agents

Embedded user-agents, commonly implemented with web-views, are an alternative method for authorizing native apps. They are however unsafe for use by third-parties by definition. They involve the user signing in with their full login credentials, only to have them downscoped to less powerful OAuth credentials.

Even when used by trusted first-party apps, embedded user-agents violate the principle of least privilege by obtaining more powerful credentials than they need, potentially increasing the attack surface.

In typical web-view based implementations of embedded user-agents, the host application can: log every keystroke entered in the form to capture usernames and passwords; automatically submit forms and bypass user-consent; copy session cookies and use them to perform authenticated actions as the user.

Encouraging users to enter credentials in an embedded web-view without the usual address bar and other identity features that browsers have makes it impossible for the user to know if they are signing in to the legitimate site, and even when they are, it trains them that it's OK to enter credentials without validating the site first.

Aside from the security concerns, web-views do not share the authentication state with other apps or the system browser, requiring the user to login for every authorization request and leading to a poor user experience.

Due to the above, use of embedded user-agents is NOT RECOMMENDED, except where a trusted first-party app acts as the external user-

agent for other apps, or provides single sign-on for multiple first-party apps.

Authorization servers SHOULD consider taking steps to detect and block logins via embedded user-agents that are not their own, where possible.

5.2. Protecting the Authorization Code

A limitation of custom URI schemes is that multiple apps can typically register the same scheme, which makes it indeterminate as to which app will receive the Authorization Code Grant. This is not an issue for HTTPS redirection URIs (i.e. standard web URLs) due to the fact the HTTPS URI scheme is enforced by the authority (as defined by [RFC3986]), the domain name system, which does not allow multiple entities to own the same domain.

If multiple apps register the same scheme, it is possible that the authorization code will be sent to the wrong app (generally the operating system makes no guarantee of which app will handle the URI when multiple register the same scheme). PKCE [RFC7636] details how this limitation can be used to execute a code interception attack (see Figure 1). This attack vector applies to public clients (clients that are unable to maintain a client secret) which is typical of most native apps.

While Section 4.1.2 details ways that this can be mitigated through policy enforcement (through being able to report and have removed any offending apps), we can also protect the authorization code grant from being used in cases where it was intercepted.

The Proof Key for Code Exchange by OAuth Public Clients (PKCE [RFC7636]) standard was created specifically to mitigate against this attack. It is a Proof of Possession extension to OAuth 2.0 that protects the code grant from being used if it is intercepted. It achieves this by having the client generate a secret verifier which it passes in the initial authorization request, and which it must present later when redeeming the authorization code grant. An app that intercepted the authorization code would not be in possession of

this secret, rendering the code useless.

Both the client and the Authorization Server MUST support PKCE [RFC7636] to use custom URI schemes, or localhost redirects. Authorization Servers SHOULD reject authorization requests using a custom scheme, or localhost as part of the redirection URI if the required PKCE parameters are not present, returning the error message as defined in Section 4.4.1 of PKCE [RFC7636]. It is RECOMMENDED to use PKCE [RFC7636] for app-claimed HTTPS redirect URIs, even though these are not generally subject to interception, to protect against attacks on inter-app communication.

5.3. Phishing

While in-app browser tabs provide a secure authentication context, as the user initiates the flow from a native app, it is possible for that native app to completely fake an in-app browser tab.

This can't be prevented directly - once the user is in the native app, that app is fully in control of what it can render, however there are several mitigating factors.

Importantly, such an attack that uses a web-view to fake an in-app browser tab will always start with no authentication state. If all native apps use the techniques described in this best practice, users will not need to sign-in frequently and thus should be suspicious of any sign-in request when they should have already been signed-in.

This is true even for authorization servers that require frequent or occasional re-authentication, as such servers can preserve some user identifiable information from the old request, like the email address or avatar. To help mitigate against phishing, it is RECOMMENDED to show the user some hint that they were previously logged in, as an attacking app would not be capable of doing this.

Users who are particularly concerned about their security may also take the additional step of opening the request in the system browser

from the in-app browser tab, and completing the authorization there, as most implementations of the in-app browser tab pattern offer such functionality. This is not expected to be common user behavior, however.

5.4. Limitations of Non-verifiable Clients

As stated in Section 10.2 of RFC 6749, the authorization server SHOULD NOT process authorization requests automatically without user consent or interaction, except when the identity of the client can be assured. Measures such as claimed HTTPS redirects can be used by native apps to prove their identity to the authorization server, and some operating systems may offer alternative platform-specific identity features which may be used, as appropriate.

6. Other External User Agents

This best practice recommends a particular type of external user-agent: the in-app browser tab. Other external user-agents patterns may also be viable for secure and usable OAuth. This document makes no comment on those patterns.

7. Client Authentication

Secrets that are statically included as part of an app distributed to multiple users should not be treated as confidential secrets, as one user may inspect their copy and learn the secret of all users. For this reason it is NOT RECOMMENDED for authorization servers to require client authentication of native apps using a secret shared by multiple installs of the app, as this serves no value beyond client identification which is already provided by the client_id request parameter. If an authorization server requires a client secret for native apps, it MUST NOT assume that it is actually secret, unless some method is being used to dynamically provision a unique secret to each installation.

8. References

8.1. Normative References

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <http://www.rfc-editor.org/info/rfc6749>.

[RFC7636] Sakimura, N., Ed., Bradley, J., and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients", RFC 7636, DOI 10.17487/RFC7636, September 2015, <http://www.rfc-editor.org/info/rfc7636>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <http://www.rfc-editor.org/info/rfc3986>.

8.2. Informative References

[RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, <http://www.rfc-editor.org/info/rfc6819>.

[iOS.URIScheme] "Inter-App Communication", February 2015, <https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Inter-AppCommunication/Inter-AppCommunication.html>.

[OSX.URIScheme] "Launch Services Concepts", February 2015, <https://developer.apple.com/library/mac/documentation/Carbon/Conceptual/LaunchServicesConcepts/LSCConcepts/LSCConcepts.html#//apple_ref/doc/uid/TP30000999-CH202-CIHFEEAD>.

[Android.URIScheme]
"Intents and Intent Filters", February 2015, <http://developer.android.com/guide/components/intents-filters.html#ires>.

[WindowsPhone.URIScheme]
"Auto-launching apps using file and URI associations for Windows Phone 8", February 2015, <https://msdn.microsoft.com/en-us/library/windows/apps/jj206987(v=vs.105).aspx>.

[Windows.URIScheme]
"Registering an Application to a URI Scheme", February 2015, <https://msdn.microsoft.com/en-us/library/ie/aa767914%28v=vs.85%29.aspx>.

[IANA.URISchemes]
"Uniform Resource Identifier (URI) Schemes", February 2015, <http://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml>.

[ChromeCustomTab]
"Chrome Custom Tabs", July 2015, <https://developer.chrome.com/multidevice/android/customtabs>.

[SFSafariViewController]
"SafariServices Changes", July 2015, <https://developer.apple.com/library/prerelease/ios/releasenotes/General/iOS90APIDiffs/frameworks/SafariServices.html>.

[Android.AppLinks]
"App Links", July 2015, <https://developer.android.com/preview/features/app-linking.html>.

Appendix A. Operating System Specific Implementation Details

Most of this document attempts to lay out best practices in an generic manner, referencing technology available on most operating

systems. This non-normative section contains OS-specific implementation details that are accurate at the time of authorship.

It is expected that this OS-specific information will change, but that the overall principles described in this document for using external user-agents will remain valid for longer.

A.1. iOS Implementation Details

From iOS 9, apps can invoke the system browser without the user leaving the app through SFSafariViewController [SFSafariViewController], which implements the browser-view pattern. This class has all the properties of the system browser, and is considered an 'external user-agent', even though it is presented within the host app. Regardless of whether the system browser is opened, or SFSafariViewController, the return of the token goes through the same system.

A.2. Android Implementation Details

Chrome 45 introduced the concept of Chrome Custom Tab [ChromeCustomTab], which follows the browser-view pattern and allows authentication without the user leaving the app.

The return of the token can go through the custom URI scheme or claimed HTTPS URI (including those registered with the App Link [Android.AppLinks] system), or the navigation events can be observed by the host app. It is RECOMMENDED that the custom URI, or claimed HTTPS URI options be used for better portability, to allow the user to open the authorization request in the Chrome app, and to prevent accidental observation of intermediate tokens on URI parameters.

At the time of writing, Android does allow apps to claim HTTPs links (App Links), but not in a way that is usable for OAuth, the native app is only opened if the intent is fired from outside the browser.

Appendix B. Acknowledgements

The author would like to acknowledge the work of Marius Scurtescu, and Ben Wiley Sittler whose design for using custom URI schemes in native OAuth 2.0 clients formed the basis of Section 4.1.

The following individuals contributed ideas, feedback, and wording that shaped and formed the final specification:

Naveen Agarwal, John Bradley, Brian Campbell, Adam Dawes, Hannes Tschofenig, Ashish Jain, Paul Madsen, Breno de Medeiros, Eric Sachs, Nat Sakimura, Steve Wright, Erik Wahlstrom, Andy Zmolek.

Authors' Addresses

William Denniss
Google
1600 Amphitheatre Pkwy
Mountain View, CA 94043
USA

Phone: +1 650-253-0000
Email: wdenniss@google.com
URI: http://google.com/

John Bradley
Ping Identity

Phone: +1 202-630-5272
Email: ve7jtb@ve7jtb.com
URI: http://www.thread-safe.com/