OAuth Working Group
Internet-Draft
Intended status: Best Current Practice
Expires: May 17, 2017

W. Denniss
Google
J. Bradley
Ping Identity

November 13, 2016

OAuth 2.0 for Native Apps
draft-ietf-oauth-native-apps-06

Abstract

OAuth 2.0 authorization requests from native apps should only be made through external user-agents, primarily the user's browser. 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 May 17, 2017.

Copyright Notice

Table of Contents

1. Introduction

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

This best current practice recommends that only external user-agents like the browser are used for OAuth by native apps. It documents how native apps can implement authorization flows using the browser as the preferred external user-agent, and the requirements for authorization servers to support such usage.

This practice is also known as the AppAuth pattern, in reference to open source libraries that implement it.

2. 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.

3. Terminology

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

"native app" An application that is installed by the user to their device, as distinct from a web app that runs in the browser context only. Apps implemented using web-based technology but distributed as a native app, so-called hybrid apps, are considered equivalent to native apps for the purpose of this specification.

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

"external user-agent" A user-agent capable of handling the authorization request that is a separate entity to the native app making the request (such as a browser), such that the app cannot access the cookie storage or modify the page content.

"embedded user-agent" A user-agent hosted inside the native app itself (such as via a web-view), with which the app has control over to the extent it is capable of accessing the cookie storage and/or modify the page content.

"app" Shorthand for "native app".

"app store" An ecommerce store where users can download and purchase apps.

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

"browser tab" An open page of the 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 browser. Has different platform-specific product names, such as SFSafariViewController on iOS, and Chrome Custom Tab on Android.

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

"claimed HTTPS URL" Some platforms allow apps to claim a HTTPS URL after proving ownership of the domain name. URLs claimed in such a way are then opened in the app instead of the browser.

"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.

"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".

4. Overview

The best current practice for authorizing users in native apps is to perform the OAuth authorization request in an external user-agent (typically the browser), rather than an embedded user-agent (such as one implemented with web-views).

Previously it was common for native apps to use embedded user-agents (commonly implemented with web-views) for OAuth authorization requests. That approach has many drawbacks, including the host app being able to copy user credentials and cookies, and the user needing to authenticate from scratch in each app. See Section 8.1 for a deeper analysis of using embedded user-agents for OAuth.

Native app authorization requests that use the browser are more secure and can take advantage of the user's authentication state.

Being able to use the existing authentication session in the browser enables single sign-on, as users don't need to authenticate to the authorization server each time they use a new app (unless required by authorization server policy).

Supporting authorization flows between a native app and the browser is possible without changing the OAuth protocol itself, as the authorization request and response are already defined in terms of URIs, which emcompasses URIs that can be used for inter-process communication. Some OAuth server implementations that assume all clients are confidential web-clients will need to add an understanding of native app OAuth clients and the types of redirect URIs they use to support this best practice.

4.1. Authorization Flow for Native Apps Using the Browser

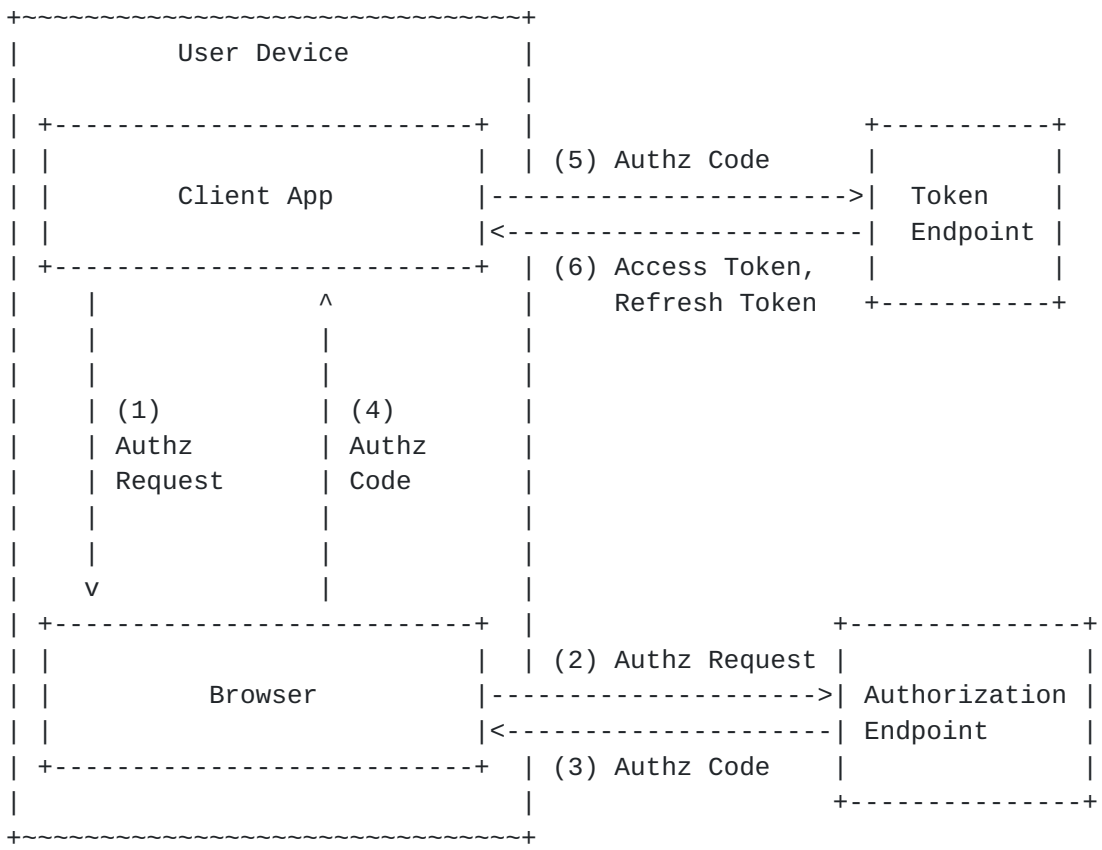

Figure 1: Native App Authorization via External User-agent

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 browser tab with the authorization request.

(2) Authorization endpoint receives the authorization request, authenticates the user and obtains authorization. Authenticating the user may involve chaining to other authentication systems.

(3) Authorization server issues an authorization code to the redirect URI.

(4) Client receives the authorization code from the redirect URI.

(5) Client app presents the authorization code at the token endpoint.

(6) Token endpoint validates the authorization code and issues the tokens requested.

5. 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 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 of 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.

Native apps MUST use an external user-agent to perform OAuth authentication requests. This is achieved by opening the authorization request in the browser (detailed in Section 6), and using a redirect URI that will return the authorization response back to the native app, as defined in Section 7.

This best practice focuses on the browser as the RECOMMENDED external user-agent for native apps. Other external user-agents, such as a native app provided by the authorization server may meet the criteria set out in this best practice, including using the same redirection URI properties, but their use is out of scope for this specification.

6. Initiating the Authorization Request from a Native App

The authorization request is created as per OAuth 2.0 [RFC6749], and opened in the user's browser using platform-specific APIs for that purpose.

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 response to the OAuth client's server, the redirect URI used by a native app returns the response to the app. The various options for a redirect URI that will return the code to the native app are documented in Section 7. Any redirect URI that allows the app to receive the URI and inspect its parameters is viable.

Some platforms support a browser feature known as in-app browser tabs, where an app can present a tab of the browser within the app context without switching apps, but still retain key benefits of the browser such as a shared authentication state and security context. On platforms where they are supported, it is RECOMMENDED for usability reasons that apps use in-app browser tabs for the Authorization Request.

7. Receiving the Authorization Response in a Native App

There are several redirect URI options available to native apps for receiving the authorization response from the browser, the availability and user experience of which varies by platform.

To fully support this best practice, authorization servers MUST support the following three redirect URI options. Native apps MAY use whichever redirect option suits their needs best, taking into account platform specific implementation details.

7.1. App-declared Custom URI Scheme Redirection

Many mobile and desktop computing platforms support inter-app communication via URIs by allowing apps to register custom URI schemes, like "com.example.app:". When the browser or another app attempts to load a URI with a custom scheme, the app that registered it is launched to handle the request.

To perform an OAuth 2.0 Authorization Request with a custom URI scheme-based redirect URI, the native app launches the browser with a normal OAuth 2.0 Authorization Request, but provides a redirection URI that utilizes a custom URI scheme registered with the operating system 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 then processes the authorization response like any OAuth client.

7.1.1. Custom URI Scheme Namespace Considerations

When choosing a URI scheme to associate with the app, apps MUST use a URI scheme based on a domain name under their control, expressed in reverse order, as recommended by Section 3.8 of [RFC7595] for private-use URI schemes.

For example, an app that controls the domain name "app.example.com" can use "com.example.app:" as their custom scheme. Some authorization servers assign client identifiers based on domain names, for example "client1234.usercontent.example.net", which can also be used as the domain name for the custom scheme, when reversed in the same manner, for example "net.example.usercontent.client1234".

URI schemes not based on a domain name (for example "myapp:") MUST NOT be used, as they are not collision resistant, and don't comply with Section 3.8 of [RFC7595].

Care must be taken when there are multiple apps by the same publisher that each URI scheme is unique within that group. On platforms that use app identifiers that are also based on reverse order domain names, those can be re-used as the custom URI scheme for the OAuth redirect.

In addition to the collision resistant properties, basing the URI scheme off a domain name that is under the control of the app can help to prove ownership in the event of a dispute where two apps claim the same custom scheme (such as if an app is acting maliciously). For example, if two apps claimed "com.example.app:", the 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 used.

7.2. App-claimed HTTPS URI Redirection

Some operating systems allow apps to claim HTTPS URLs in their domains. When the browser encounters a claimed URL, instead of the page being loaded in the browser, the native app is launched with the URL supplied as a launch parameter.

App-claimed HTTPS redirect URIs have some advantages in that the identity of the destination app is guaranteed by the operating system. Due to this reason, they SHOULD be used over the other redirect choices for native apps where possible.

App-claimed HTTPS redirect URIs function as normal HTTPS redirects from the perspective of the authorization server, though it is RECOMMENDED that the authorization server is able to distinguish

between public native app clients that use app-claimed HTTPS redirect URIs and confidential web clients. A configuration option in the client registration (as documented in Section 8.4) is one method for distinguishing client types.

7.3. Loopback URI Redirection

Desktop operating systems allow native apps to listen on a local port for HTTP redirects. This can be used by native apps to receive OAuth authorization responses on compatible platforms.

Loopback redirect URIs take the form of the loopback IP, any port (dynamically provided by the client), and a path component. Specifically: "http://127.0.0.1:{port}/{path}" for IPv4, and "http://[::1]:{port}/{path}" for IPv6.

For loopback IP redirect URIs, the authorization server MUST allow any port to be specified at the time of the request, to accommodate clients that obtain an available port from the operating system at the time of the request. Other than that, the redirect is be treated like any other.

8. Security Considerations

8.1. Embedded User-Agents

Embedded user-agents are an alternative method for authorizing native apps. They are however unsafe for use by third-parties to the authorization server by definition, as the app that hosts the embedded user-agent can access the user's full authentication credential, not just the OAuth authorization grant that was intended for the app.

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.

Even when used by trusted apps belonging to the same party as the authorization server, embedded user-agents violate the principle of least privilege by having access to more powerful credentials than they need, potentially increasing the attack surface.

Encouraging users to enter credentials in an embedded user-agent without the usual address bar and visible certificate validation 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, embedded user-agents do not share the authentication state with other apps or the browser, requiring the user to login for every authorization request and leading to a poor user experience.

Native apps MUST NOT use embedded user-agents to perform authorization requests.

Authorization endpoints MAY take steps to detect and block authorization requests in embedded user-agents.

8.2. Protecting the Authorization Code

The redirect URI options documented in Section 7 share the benefit that only a native app on the same device can receive the authorization code which limits the attack surface, however code interception by a native app other than the intended app may still be possible.

A limitation of using custom URI schemes for redirect URIs is that multiple apps can typically register the same scheme, which makes it indeterminate as to which app will receive the Authorization Code. PKCE [RFC7636] details how this limitation can be used to execute a code interception attack (see Figure 1).

Loopback IP based redirect URIs may be susceptible to interception by other apps listening on the same loopback interface.

As most forms of inter-app URI-based communication sends data over insecure local channels, eavesdropping and interception of the authorization response is a risk for native apps. App-claimed HTTPS redirects are hardened against this type of attack due to the presence of the URI authority, but they are still public clients and the URI is still transmitted over local channels with unknown security properties.

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.

Public native app clients MUST protect the authorization request with PKCE [RFC7636]. Authorization servers MUST support PKCE [RFC7636] for public native app clients. Authorization servers SHOULD reject authorization requests from native apps that don't use PKCE by returning an error message as defined in Section 4.4.1 of PKCE [RFC7636].

8.3. Loopback Redirect Considerations

Loopback interface redirect URIs use the "http" scheme (i.e. without TLS). This is acceptable for loopback interface redirect URIs as the HTTP request never leaves the device.

Clients should open the loopback port only when starting the authorization request, and close it once the response is returned.

While redirect URIs using localhost (i.e. "http://localhost:{port}/" function similarly to loopback IP redirects described in Section 7.3, the use of "localhost" is NOT RECOMMENDED. Opening a port on the loopback interface is more secure as only apps on the local device can connect to it. It is also less susceptible to misconfigured routing, and interference by client side firewalls.

8.4. Registration of Native App Clients

Authorization Servers SHOULD have a way to distinguish public native app clients from confidential web-clients, as the lack of client authentication means they are often handled differently. A configuration option to indicate a public native app client is one such popular method for achieving this.

As recommended in Section 3.1.2.2 of OAuth 2.0 [RFC6749], the authorization server SHOULD require the client to pre-register the complete redirection URI. This applies and is RECOMMENDED for all redirection URIs used by native apps.

For Custom URI scheme based redirects, authorization servers SHOULD enforce the requirement in Section 7.1.1 that clients use reverse domain name based schemes.

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.

8.5. OAuth Implicit Flow

The OAuth 2.0 Implicit Flow as defined in Section 4.2 of OAuth 2.0 [RFC6749] generally works with the practice of performing the authorization request in the browser, and receiving the authorization response via URI-based inter-app communication. However, as the Implicit Flow cannot be protected by PKCE (which is a recommended in Section 7.1.1), the use of the Implicit Flow with native apps is NOT RECOMMENDED.

Tokens granted via the implicit flow also cannot be refreshed without user interaction making the code flow, with refresh tokens the more practical option for native app authorizations that require refreshing.

8.6. Phishability of In-App Browser Tabs

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 the case even for authorization servers that require occasional or frequent re-authentication, as such servers can preserve some user identifiable information from the old session, like the email address or profile picture and display that on the re-authentication.

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

8.7. Limitations of Non-verifiable Clients

As stated in Section 10.2 of OAuth 2.0 [RFC6749], 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.

8.8. Non-Browser External User-Agents

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

8.9. 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 shared secret. For this reason, and those stated in Section 5.3.1 of [RFC6819], it is NOT RECOMMENDED for authorization servers to require client authentication of native apps using a shared secret, as this serves little value beyond client identification which is already provided by the "client_id" request parameter.

Authorization servers that still require a shared secret for native app clients MUST treat the client as a public client, and not treat the secret as proof of the client's identity. In those cases, it is NOT RECOMMENDED to automatically issue tokens on the basis that the user has previously granted access to the same client, as there is no guarantee that the client is not counterfeit.

8.10. Cross-App Request Forgery Protections

Section 5.3.5 of [RFC6819] recommends using the 'state' parameter to link client requests and responses to prevent CSRF attacks.

It is similarly RECOMMENDED for native apps to include a high entropy secure random number in the 'state' parameter of the authorization request, and reject any incoming authorization responses without a state value that matches a pending outgoing authorization request.

8.11. Authorization Server Mix-Up Mitigation

To protect against a compromised or malicious authorization server attacking another authorization server used by the same app, it is RECOMMENDED that a unique redirect URI is used for each different authorization server used by the app (for example, by varying the path component), and that authorization responses are rejected if the

redirect URI they were received on doesn't match the redirect URI in a pending outgoing authorization request.

Authorization servers SHOULD allow the registration of a specific redirect URI, including path components, and reject authorization requests that specify a redirect URI that doesn't exactly match the one that was registered.

9. IANA Considerations

[RFC Editor: please do not remove this section.]

Section 7.1 specifies how private-use URI schemes are used for inter-app communication in OAuth protocol flows. This document requires in Section 7.1.1 that such schemes are based on domain names owned or assigned to the app, as recommended in Section 3.8 of [RFC7595]. Per section 6 of [RFC7595], registration of domain based URI schemes with IANA is not required. Therefore, this document has no IANA actions.

10. References

10.1. Normative References

[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>.

[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>.

[RFC7595] Thaler, D., Ed., Hansen, T., and T. Hardie, "Guidelines and Registration Procedures for URI Schemes", BCP 35, RFC 7595, DOI 10.17487/RFC7595, June 2015, <http://www.rfc-editor.org/info/rfc7595>.

[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>.

10.2. 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>.

[AppAuth.iOSmacOS] Wright, S., Denniss, W., and others, "AppAuth for iOS and macOS", February 2016, <https://github.com/openid/AppAuth-iOS>.

[AppAuth.Android] McGinniss, I., Denniss, W., and others, "AppAuth for Android", February 2016, <https://github.com/openid/AppAuth-Android>.

[SamplesForWindows] Denniss, W., "OAuth for Apps: Samples for Windows", July 2016, <https://github.com/googlesamples/oauth-apps-for-windows>.

Appendix A. Server Support Checklist

OAuth servers that support native apps should:

1. Support custom URI-scheme redirect URIs. This is required to support mobile operating systems. See Section 7.1.

2. Support HTTPS redirect URIs for use with public native app clients. This is used by apps on advanced mobile operating systems that allow app-claimed HTTPS URIs. See Section 7.2.

3. Support loopback IP redirect URIs. This is required to support desktop operating systems. See Section 7.3.

4. Not assume native app clients can keep a secret. If secrets are distributed to multiple installs of the same native app, they should not be treated as confidential. See Section 8.9.

5. Support PKCE. Recommended to protect authorization code grants transmitted to public clients over inter-app communication channels. See Section 8.2

Appendix B. Operating System Specific Implementation Details

Most of this document defines best practices in an generic manner, referencing techniques commonly available in a variety of environments. This non-normative section contains OS-specific implementation details for the generic pattern, that are considered accurate at the time of publishing, but may change over time.

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.

B.1. iOS Implementation Details

Apps can initiate an authorization request in the browser without the user leaving the app, through the SFSafariViewController class which implements the browser-view pattern. Safari can be used to handle requests on old versions of iOS without SFSafariViewController.

To receive the authorization response, both custom URI scheme redirects and claimed HTTPS links (known as Universal Links) are viable choices, and function the same whether the request is loaded in SFSafariViewController or the Safari app. Apps can claim Custom URI schemes with the "CFBundleURLTypes" key in the application's property list file "Info.plist", and HTTPS links using the Universal Links feature with an entitlement file and an association file on the domain.

Universal Links are the preferred choice on iOS 9 and above due to the ownership proof that is provided by the operating system.

A complete open source sample is included in the AppAuth for iOS and macOS [AppAuth.iOSmacOS] library.

B.2. Android Implementation Details

Apps can initiate an authorization request in the browser without the user leaving the app, through the Android Custom Tab feature which implements the browser-view pattern. The user's default browser can be used to handle requests when no browser supports Custom Tabs.

Android browser vendors should support the Custom Tabs protocol (by providing an implementation of the "CustomTabsService" class), to provide the in-app browser tab user experience optimization to their users. Chrome is one such browser that implements Custom Tabs.

To receive the authorization response, custom URI schemes are broadly supported through Android Implicit Intends. Claimed HTTPS redirect

URIs through Android App Links are available on Android 6.0 and above. Both types of redirect URIs are registered in the application's manifest.

A complete open source sample is included in the AppAuth for Android [AppAuth.Android] library.

B.3. Windows Implementation Details

Apps can initiate an authorization request in the user's default browser using platform APIs for this purpose.

The custom URI scheme redirect is a good choice for Universal Windows Platform (UWP) apps as it will open the app returning the user right back where they were. Known on UWP as URI Activation, the scheme is limited to 39 characters, but may include the "." character, making short reverse domain name based schemes (as recommended in Section 7.1.1) possible.

The loopback redirect is the common choice for traditional desktop apps, listening on a loopback interface port is permitted by default Windows firewall rules.

A complete open source sample is available [SamplesForWindows].

B.4. macOS Implementation Details

Apps can initiate an authorization request in the user's default browser using platform APIs for this purpose.

To receive the authorization response, custom URI schemes are are a good redirect URI choice on macOS, as the user is returned right back to the app they launched the request from. These are registered in the application's bundle information property list using the "CFBundleURLSchemes" key. Loopback IP redirects are another viable option, and listening on the loopback interface is allowed by default firewall rules.

A complete open source sample is included in the AppAuth for iOS and macOS [AppAuth.iOSmacOS] library.

B.5. Linux Implementation Details

Opening the Authorization Request in the user's default browser requires a distro-specific command, "xdg-open" is one such tool.

The loopback redirect is the recommended redirect choice for desktop apps on Linux to receive the authorization response.

Appendix C. 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 7.1.

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

Andy Zmolek, Steven E Wright, Brian Campbell, Paul Madsen, Nat Sakimura, Iain McGinniss, Rahul Ravikumar, Eric Sachs, Breno de Medeiros, Adam Dawes, Naveen Agarwal, Hannes Tschofenig, Ashish Jain, Erik Wahlstrom, Bill Fisher, Sudhi Umarji, Michael B. Jones, Vittorio Bertocci, Dick Hardt, David Waite, and Ignacio Fiorentino.

Authors' Addresses

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

Email: wdenniss@google.com
URI: http://wdenniss.com/appauth

John Bradley
Ping Identity

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