OAuth Working Group
Internet-Draft
Intended status: Best Current Practice
Expires: January 23, 2016

W. Denniss
Google
J. Bradley
Ping Identity

July 22, 2015

OAuth 2.0 for Native Apps
draft-wdenniss-oauth-native-apps-00

Abstract

OAuth 2.0 authorization requests from native apps should only be made through external user-agents such as the system 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

Copyright Notice

Table of Contents

1. Introduction 2
  1.1. Authorization Flow for Native Apps 3
2. Notational Conventions 4
3. Terminology 4
4. The External User-Agent 5
5. Redirection URIs for Native Apps 5
  5.1. App-claimed HTTPS URI Redirection 5
  5.2. App-declared Custom URI Scheme Redirection 6
6. Security Considerations 8
  6.1. Embedded User-Agents 8
  6.2. Protecting the Authorization Code 9
  6.3. Claimed URLs and Phishing 10
  6.4. Always Prompting for User Interaction 10
7. References 10
  7.1. Normative References 10
  7.2. Informative References 11
Appendix A. Operating System Specific Implementation Details . . 12
  A.1. iOS Implementation Details 12
  A.2. Android Implementation Details 12
Appendix B. Acknowledgements 13
Authors’ Addresses 13

1. Introduction

The OAuth 2.0 [RFC6749] authorization framework, documents two ways 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 (such as the system browser) as the only secure and usable choice for OAuth2. It documents how native apps can implement authorization flows with such agents, and the additional requirements of authorization servers needed to support such usage.

Many native apps today are using an embedded user-agent in the form of a web-view. This approach suffers from several security and usability issues including allowing the client app to eavesdrop user credentials, and forcing users 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. Operating systems are increasingly making the system browser even more viable for OAuth by allowing apps to show a browser window within the active app, removing the only usability benefit of using embedded browsers in the first place (not wanting to send the user to another app).

Inter-app communication (such as that between a native OAuth client and the system browser) can be achieved through app-specific custom URI schemes and/or claimed HTTPS URLs. For example, an app can launch the system browser with a HTTPS request (such as an OAuth request), the browser can process the request and return control to the app by simply following a URI using a scheme that the app registered (for example "com.example.app:/oauth2callback?code=..."), or a HTTPS path that the app claimed. Parameters can be passed through these URIs, allowing complete use of OAuth flows, while minimizing the added complexity for authorization servers to support native apps.

1.1. Authorization Flow for Native Apps

```
    +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
    |          User Device           |
    |                                |
    | +---------------------------+   |                     +-----------+
    | |                           |   | (4) Authz Grant     |           |
    | |        Client App         |----------------------->|  Authz    |
    | |                           |<-----------------------|  Server   |
    | +---------------------------+   | (5) Access Token    |           |
    |    |              ^            |                     +-----------+
    |    |              |            |
    |    |              |            |
    |    | (1)          | (3)        |
    |    | Authz        | Authz      |
    |    | Request      | Grant      |
    |    | "https://"   | "app:/"    |
    |    |              |            |
    |    v              |            |
    | +---------------------------+   |                     +-----------+
    | |                           |   | (2) User            |           |
    | |       System Browser      |   |     authenticated   | Identity  |
    | |                           |<---------------------->| Provider  |
    | +---------------------------+   |                     |           |
    |                                |                     +-----------+
    +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
```

Figure 1: Native App Authorization via External User-agent

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

1) The client app launches the system browser or browser-view 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 [PKCE] proof of possession verifier

5) Server issues the tokens requested

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:

"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 product users.

"system browser" The operating system’s native default browser, typically pre-installed as part of the operating system, or installed and set as default by the user. For example mobile Safari on iOS, and Chrome on Android.

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

"browser-view" 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. Goes by different names on different platforms, such as SFSafariViewController on iOS 9, and Chrome Custom Tab in Chrome for Android.

"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.

4. The External User-Agent

The external user-agent for native apps can be the system browser, or a native app provided by the authorization server.

Both the system browser and authorization server app affords several advantages for OAuth over embedded web-view based user-agents, including the security of a separate process, and usability of a shared authentication session.

The system browser is the RECOMMENDED external user-agent choice for most authorization servers, as it reduces implementation complexity by reusing the web authorization endpoint, and is often needed as a fallback even when an authorization server app is available.

5. Redirection URIs for Native Apps

5.1. App-claimed HTTPS URI Redirection

Several operating systems support a method for an app to claim a regular HTTPS URL. When such a URL is loaded in the browser, instead of the request being made and the page loaded, the native app is launched instead.

On operating systems that support app-claimed HTTPS URIs, these URIs SHOULD be used with OAuth, 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, 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.

5.2. 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.

5.2.1. 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.

5.2.2. 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.

5.2.3. Registration of App Redirection URIs

As recommended in Section 3.1.2.2 of [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 5.2.2 that apps follow naming guidelines for URI schemes.

6. Security Considerations

6.1. 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.

The only use-case where it is reasonable to use an embedded user-agent is when the app itself is a trusted and secure first-party app that acts as the external user-agent for other apps. Use of embedded user-agents by first party apps other than those that act as an external user-agent themselves is NOT RECOMMENDED, as it increases development complexity and the potential to introduce security issues, and hampers the potential for usability improvements through taking advantage of the shared authentication context.

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

6.2. 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]), being the domain name system, which does not allow multiple entities to own a single 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). Figure 1 of [PKCE] demonstrates the code interception attack. This attack vector applies to public clients (clients that are unable to maintain a client secret) which is typical of most installed apps.

While Section 5.2.2 mentions ways that this can be mitigated through policy enforcement (by being able to request that the offending app is removed), 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) [PKCE] 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.

Both the client and the Authorization Server MUST support PKCE [PKCE] to use custom URI schemes. Authorization Servers SHOULD reject requests that use a custom scheme in the redirection URI if the required PKCE parameters are not also present, returning the error message as defined in Section 4.4.1 of [PKCE]

PKCE provides proof of possession by the client generating 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.

6.3. Claimed URLs and Phishing

While using a claimed HTTPS URI for redirection in the system browser guarantees the identity of the receiving app, it is still possible for a bad app to put the user through an authentication flow in an embedded user-agent of their own, and observe the redirect URI.

We can’t directly prevent this, however it can be mitigated through user contextual awareness. Such an attack necessarily starts with no authentication state, meaning that the user will be prompted to sign-in. If all native apps are using the techniques described here, users should not be signing-in frequently, and thus should treat any password request event with more suspicion. Sophisticated users will be able to recognise the UI treatment of the browser-view or full system browser, and shouldn’t sign-in anywhere else. Users who are particularly security conscious can also use the "open in browser" functionality from the browser-view to gain even more assurances about where they are entering their credentials.

6.4. Always Prompting for User Interaction

Due to the fact that the identity of non-confidential clients cannot be assured, tokens SHOULD NOT be issued to such clients without user consent or interaction, even if the the user has consented to the scopes and approved the client previously.

7. References

7.1. 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>.

[PKCE] Sakimura, N., Ed., Bradley, J., and N. Agarwal, "The Proof Key for Code Exchange by OAuth Public Clients", February 2015, <https://tools.ietf.org/html/draft-ietf-oauth-spop>.

[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>.

7.2. 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 valid 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.

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 5.2.

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

Naveen Agarwal, John Bradley, Brian Campbell, Adam Dawes, Ashish Jain, Paul Madsen, Breno de Medeiros, Eric Sachs, Nat Sakimura, Steve Wright.

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: +44 20 8133 3718
Email: ve7jtb@ve7jtb.com
URI: http://www.thread-safe.com/