Internet Engineering Task Force (IETF)
Request for Comments: 8252
BCP: 212
Updates: 6749
Category: Best Current Practice
ISSN: 2070-1721

W. Denniss
Google
J. Bradley
Ping Identity

October 2017

OAuth 2.0 for Native Apps

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 memo documents an Internet Best Current Practice.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on BCPs is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc8252.

Copyright Notice

Copyright (c) 2017 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 (https://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 3
2. Notational Conventions 3
3. Terminology 3
4. Overview 4
4.1. Authorization Flow for Native Apps Using the Browser . . 5
5. Using Inter-App URI Communication for OAuth 6
6. Initiating the Authorization Request from a Native App . . . 6
7. Receiving the Authorization Response in a Native App 7
7.1. Private-Use URI Scheme Redirection 8
7.2. Claimed "https" Scheme URI Redirection 9
7.3. Loopback Interface Redirection 9
8. Security Considerations 10
8.1. Protecting the Authorization Code 10
8.2. OAuth Implicit Grant Authorization Flow 11
8.3. Loopback Redirect Considerations 11
8.4. Registration of Native App Clients 12
8.5. Client Authentication 12
8.6. Client Impersonation 13
8.7. Fake External User-Agents 13
8.8. Malicious External User-Agents 14
8.9. Cross-App Request Forgery Protections 14
8.10. Authorization Server Mix-Up Mitigation 14
8.11. Non-Browser External User-Agents 15
8.12. Embedded User-Agents 15
9. IANA Considerations 16
10. References 16
10.1. Normative References 16
10.2. Informative References 17
Appendix A. Server Support Checklist 18
Appendix B. Platform-Specific Implementation Details 18
B.1. iOS Implementation Details 18
B.2. Android Implementation Details 19
B.3. Windows Implementation Details 19
B.4. macOS Implementation Details 20
B.5. Linux Implementation Details 21
Acknowledgements 21
Authors' Addresses 21

1. Introduction

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

This best current practice requires 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 as well as the requirements for authorization servers to support such usage.

This practice is also known as the "AppAuth pattern", in reference to open-source libraries [AppAuth] 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 BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Terminology

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

"native app" An app or 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.

"app" A "native app" unless further specified.

"app store" An e-commerce store where users can download and purchase apps.

"OAuth" Authorization protocol specified by the OAuth 2.0 Authorization Framework [RFC6749].

"external user-agent" A user-agent capable of handling the authorization request that is a separate entity or security domain to the native app making the request, such that the app cannot access the cookie storage, nor inspect or modify page content.

"embedded user-agent" A user-agent hosted by the native app making the authorization request that forms a part of the app or shares the same security domain such that the app can access the cookie storage and/or inspect or modify page content.

"browser" The default application launched by the operating system to handle "http" and "https" scheme URI content.

"in-app browser tab" A programmatic instantiation of the browser that is displayed inside a host app but that retains the full security properties and authentication state of the browser. It has different platform-specific product names, several of which are detailed in Appendix B.

"web-view" A web browser UI (user interface) component that is embedded in apps to render web pages under the control of the app.

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

"claimed "https" scheme URI" Some platforms allow apps to claim an "https" scheme URI after proving ownership of the domain name. URIs claimed in such a way are then opened in the app instead of the browser.

"private-use URI scheme" As used by this document, a URI scheme defined by the app (following the requirements of Section 3.8 of [RFC7595]) and registered with the operating system. URI requests to such schemes launch the app that registered it to handle the request.

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

4. Overview

For authorizing users in native apps, the best current practice 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 as well as the user needing to authenticate from scratch in each app. See Section 8.12

for a deeper analysis of the drawbacks 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 the authorization server policy).

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

4.1. Authorization Flow for Native Apps Using the Browser

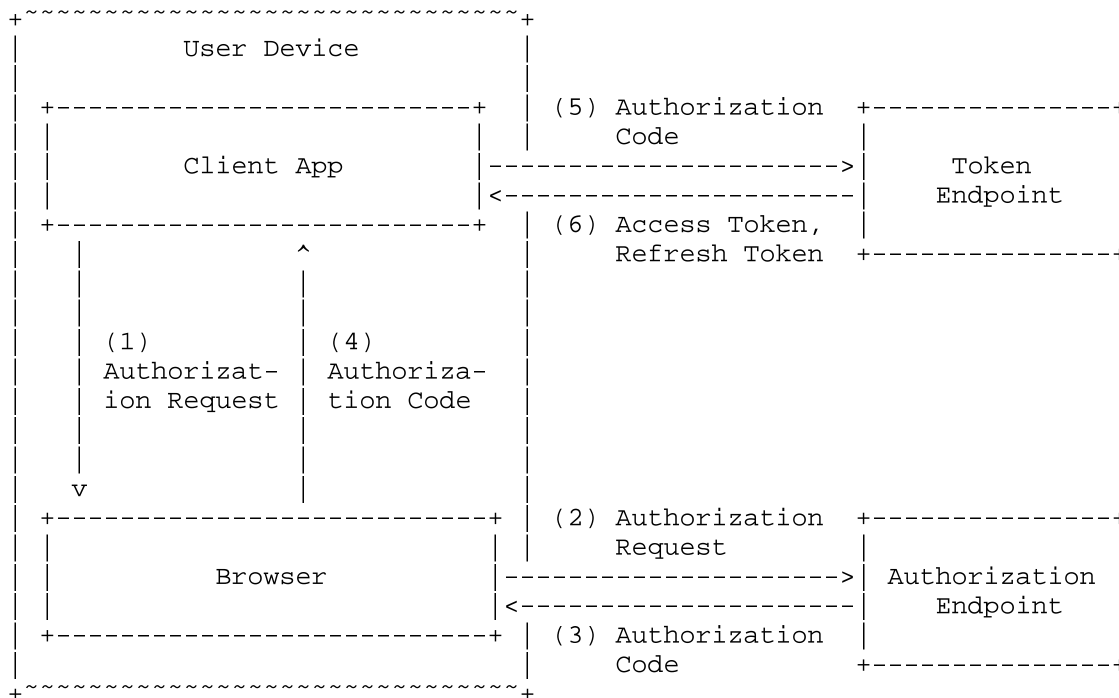

Figure 1: Native App Authorization via an External User-Agent

Figure 1 illustrates the interaction between a native app and the browser to authorize the user.

(1) 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 adopting the same methods used on the web for OAuth, benefits seen in the web context like the usability of a single sign-on session and the security of a separate authentication context are likewise gained in the native app context. Reusing the same approach also reduces the implementation complexity and increases interoperability by relying on standards-based web flows that are not specific to a particular platform.

To conform to this best practice, native apps MUST use an external user-agent to perform OAuth authorization 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 (defined in Section 7).

6. Initiating the Authorization Request from a Native App

Native apps needing user authorization create an authorization request URI with the authorization code grant type per Section 4.1 of OAuth 2.0 [RFC6749], using a redirect URI capable of being received by the native app.

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. Several options for a redirect URI that will return the authorization response to the native app in different platforms are documented in Section 7. Any redirect URI that allows the app to receive the URI and inspect its parameters is viable.

Public native app clients MUST implement the Proof Key for Code Exchange (PKCE [RFC7636]) extension to OAuth, and authorization servers MUST support PKCE for such clients, for the reasons detailed in Section 8.1.

After constructing the authorization request URI, the app uses platform-specific APIs to open the URI in an external user-agent. Typically, the external user-agent used is the default browser, that is, the application configured for handling "http" and "https" scheme URIs on the system; however, different browser selection criteria and other categories of external user-agents MAY be used.

This best practice focuses on the browser as the RECOMMENDED external user-agent for native apps. An external user-agent designed specifically for user authorization and capable of processing authorization requests and responses like a browser MAY also be used. 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.

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 offer at least the three redirect URI options described in the following subsections to native apps. Native apps MAY use whichever redirect option suits their needs best, taking into account platform-specific implementation details.

7.1. Private-Use URI Scheme Redirection

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

To perform an OAuth 2.0 authorization request with a private-use URI scheme redirect, the native app launches the browser with a standard authorization request, but one where the redirection URI utilizes a private-use URI scheme it registered with the operating system.

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 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 scheme when reversed in the same manner. A scheme such as "myapp", however, would not meet this requirement, as it is not based on a domain name.

When there are multiple apps by the same publisher, care must be taken so that each scheme is unique within that group. On platforms that use app identifiers based on reverse-order domain names, those identifiers can be reused as the private-use URI scheme for the OAuth redirect to help avoid this problem.

Following the requirements of Section 3.2 of [RFC3986], as there is no naming authority for private-use URI scheme redirects, only a single slash ("/") appears after the scheme component. A complete example of a redirect URI utilizing a private-use URI scheme is:

```
com.example.app:/oauth2redirect/example-provider
```

When the authorization server completes the request, it redirects to the client's redirection URI as it would normally. As the redirection URI uses a private-use URI scheme, it results in the operating system launching the native app, passing in the URI as a launch parameter. Then, the native app uses normal processing for the authorization response.

7.2. Claimed "https" Scheme URI Redirection

Some operating systems allow apps to claim "https" scheme [RFC7230] URIs in the domains they control. When the browser encounters a claimed URI, instead of the page being loaded in the browser, the native app is launched with the URI supplied as a launch parameter.

Such URIs can be used as redirect URIs by native apps. They are indistinguishable to the authorization server from a regular web-based client redirect URI. An example is:

```
https://app.example.com/oauth2redirect/example-provider
```

As the redirect URI alone is not enough to distinguish public native app clients from confidential web clients, it is REQUIRED in Section 8.4 that the client type be recorded during client registration to enable the server to determine the client type and act accordingly.

App-claimed "https" scheme redirect URIs have some advantages compared to other native app redirect options in that the identity of the destination app is guaranteed to the authorization server by the operating system. For this reason, native apps SHOULD use them over the other options where possible.

7.3. Loopback Interface Redirection

Native apps that are able to open a port on the loopback network interface without needing special permissions (typically, those on desktop operating systems) can use the loopback interface to receive the OAuth redirect.

Loopback redirect URIs use the "http" scheme and are constructed with the loopback IP literal and whatever port the client is listening on.

That is, "http://127.0.0.1:{port}/{path}" for IPv4, and "http://[::1]:{port}/{path}" for IPv6. An example redirect using the IPv4 loopback interface with a randomly assigned port:

    http://127.0.0.1:51004/oauth2redirect/example-provider

An example redirect using the IPv6 loopback interface with a randomly assigned port:

    http://[::1]:61023/oauth2redirect/example-provider

The authorization server MUST allow any port to be specified at the time of the request for loopback IP redirect URIs, to accommodate clients that obtain an available ephemeral port from the operating system at the time of the request.

Clients SHOULD NOT assume that the device supports a particular version of the Internet Protocol. It is RECOMMENDED that clients attempt to bind to the loopback interface using both IPv4 and IPv6 and use whichever is available.

8. Security Considerations

8.1. Protecting the Authorization Code

The redirect URI options documented in Section 7 share the benefit that only a native app on the same device or the app's own website can receive the authorization code, which limits the attack surface. However, code interception by a different native app running on the same device may be possible.

A limitation of using private-use 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. Section 1 of PKCE [RFC7636] details how this limitation can be used to execute a code interception attack.

Loopback IP-based redirect URIs may be susceptible to interception by other apps accessing the same loopback interface on some operating systems.

App-claimed "https" scheme redirects are less susceptible to URI interception due to the presence of the URI authority, but the app is still a public client; further, the URI is sent using the operating system's URI dispatch handler with unknown security properties.

The PKCE [RFC7636] protocol was created specifically to mitigate this attack. It is a proof-of-possession extension to OAuth 2.0 that protects the authorization code from being used if it is intercepted. To provide protection, this extension has the client generate a secret verifier; it passes a hash of this verifier in the initial authorization request, and must present the unhashed verifier when redeeming the authorization code. An app that intercepted the authorization code would not be in possession of this secret, rendering the code useless.

Section 6 requires that both clients and servers use PKCE 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].

8.2. OAuth Implicit Grant Authorization Flow

The OAuth 2.0 implicit grant authorization flow (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 [RFC7636] (which is required in Section 8.1), the use of the Implicit Flow with native apps is NOT RECOMMENDED.

Access tokens granted via the implicit flow also cannot be refreshed without user interaction, making the authorization code grant flow -- which can issue refresh tokens -- the more practical option for native app authorizations that require refreshing of access tokens.

8.3. Loopback Redirect Considerations

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

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

Clients should listen on the loopback network interface only, in order to avoid interference by other network actors.

While redirect URIs using localhost (i.e., "http://localhost:{port}/{path}") function similarly to loopback IP redirects described in Section 7.3, the use of localhost is NOT RECOMMENDED. Specifying a redirect URI with the loopback IP literal rather than localhost avoids inadvertently listening on network

interfaces other than the loopback interface. It is also less susceptible to client-side firewalls and misconfigured host name resolution on the user's device.

8.4. Registration of Native App Clients

Except when using a mechanism like Dynamic Client Registration [RFC7591] to provision per-instance secrets, native apps are classified as public clients, as defined by Section 2.1 of OAuth 2.0 [RFC6749]; they MUST be registered with the authorization server as such. Authorization servers MUST record the client type in the client registration details in order to identify and process requests accordingly.

Authorization servers MUST require clients to register their complete redirect URI (including the path component) and reject authorization requests that specify a redirect URI that doesn't exactly match the one that was registered; the exception is loopback redirects, where an exact match is required except for the port URI component.

For private-use URI scheme-based redirects, authorization servers SHOULD enforce the requirement in Section 7.1 that clients use schemes that are reverse domain name based. At a minimum, any private-use URI scheme that doesn't contain a period character (".") SHOULD be rejected.

In addition to the collision-resistant properties, requiring a URI scheme based on 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 private-use URI scheme (where one 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. Such a petition is harder to prove if a generic URI scheme was used.

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

8.5. 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 public native apps clients 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 statically included shared secret for native app clients MUST treat the client as a public client (as defined by Section 2.1 of OAuth 2.0 [RFC6749]), and not accept the secret as proof of the client's identity. Without additional measures, such clients are subject to client impersonation (see Section 8.6).

8.6. Client Impersonation

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. This includes the case where the user has previously approved an authorization request for a given client id -- unless the identity of the client can be proven, the request SHOULD be processed as if no previous request had been approved.

Measures such as claimed "https" scheme redirects MAY be accepted by authorization servers as identity proof. Some operating systems may offer alternative platform-specific identity features that MAY be accepted, as appropriate.

8.7. Fake External User-Agents

The native app that is initiating the authorization request has a large degree of control over the user interface and can potentially present a fake external user-agent, that is, an embedded user-agent made to appear as an external user-agent.

When all good actors are using external user-agents, the advantage is that it is possible for security experts to detect bad actors, as anyone faking an external user-agent is provably bad. On the other hand, if good and bad actors alike are using embedded user-agents, bad actors don't need to fake anything, making them harder to detect. Once a malicious app is detected, it may be possible to use this knowledge to blacklist the app's signature in malware scanning software, take removal action (in the case of apps distributed by app stores) and other steps to reduce the impact and spread of the malicious app.

Authorization servers can also directly protect against fake external user-agents by requiring an authentication factor only available to true external user-agents.

Users who are particularly concerned about their security when using in-app browser tabs may also take the additional step of opening the request in the full browser from the in-app browser tab and complete the authorization there, as most implementations of the in-app browser tab pattern offer such functionality.

8.8. Malicious External User-Agents

If a malicious app is able to configure itself as the default handler for "https" scheme URIs in the operating system, it will be able to intercept authorization requests that use the default browser and abuse this position of trust for malicious ends such as phishing the user.

This attack is not confined to OAuth; a malicious app configured in this way would present a general and ongoing risk to the user beyond OAuth usage by native apps. Many operating systems mitigate this issue by requiring an explicit user action to change the default handler for "http" and "https" scheme URIs.

8.9. 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 (Cross-Site Request Forgery) attacks.

To mitigate CSRF-style attacks over inter-app URI communication channels (so called "cross-app request forgery"), it is similarly RECOMMENDED that native apps 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.

8.10. 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 REQUIRED that a unique redirect URI is used for each 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 an outgoing authorization request.

The native app MUST store the redirect URI used in the authorization request with the authorization session data (i.e., along with "state" and other related data) and MUST verify that the URI on which the authorization response was received exactly matches it.

The requirement of Section 8.4, specifically that authorization servers reject requests with URIs that don't match what was registered, is also required to prevent such attacks.

8.11. 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.

8.12. Embedded User-Agents

Section 9 of OAuth 2.0 [RFC6749] documents two approaches for native apps to interact with the authorization endpoint. This best current practice requires that native apps MUST NOT use embedded user-agents to perform authorization requests and allows that authorization endpoints MAY take steps to detect and block authorization requests in embedded user-agents. The security considerations for these requirements are detailed herein.

Embedded user-agents are an alternative method for authorizing native apps. These embedded user-agents are 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 record every keystroke entered in the login form to capture usernames and passwords, automatically submit forms to bypass user consent, and 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; 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 log in for every authorization request, which is often considered an inferior user experience.

9. IANA Considerations

This document does not require any IANA actions.

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 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.

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, <https://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, <https://www.rfc-editor.org/info/rfc3986>.

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

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[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, <https://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, <https://www.rfc-editor.org/info/rfc7636>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

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, <https://www.rfc-editor.org/info/rfc6819>.

[RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <https://www.rfc-editor.org/info/rfc7591>.

[AppAuth] OpenID Connect Working Group, "AppAuth", September 2017, <https://openid.net/code/AppAuth>.

[AppAuth.iOSmacOS] Wright, S., Denniss, W., et al., "AppAuth for iOS and macOS", February 2016, <https://openid.net/code/AppAuth-iOS>.

[AppAuth.Android] McGinniss, I., Denniss, W., et al., "AppAuth for Android", February 2016, <https://openid.net/code/AppAuth-Android>.

[SamplesForWindows] Denniss, W., "OAuth for Apps: Samples for Windows", July 2016, <https://openid.net/code/sample-oauth-apps-for-windows>.

Appendix A. Server Support Checklist

OAuth servers that support native apps must:

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

2. Support "https" scheme redirect URIs for use with public native app clients. This is used by apps on advanced mobile operating systems that allow app-claimed "https" scheme 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 that 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 8.5.

5. Support PKCE [RFC7636]. Required to protect authorization code grants sent to public clients over inter-app communication channels. See Section 8.1

Appendix B. Platform-Specific Implementation Details

This document primarily defines best practices in a generic manner, referencing techniques commonly available in a variety of environments. This non-normative section documents implementation details of the best practice for various operating systems.

The implementation details herein are considered accurate at the time of publishing but will likely change over time. It is hoped that such a change won't invalidate the generic principles in the rest of the document and that those principles should take precedence in the event of a conflict.

B.1. iOS Implementation Details

Apps can initiate an authorization request in the browser, without the user leaving the app, through the "SFSafariViewController" class or its successor "SFAuthenticationSession", which implement the in-app browser tab pattern. Safari can be used to handle requests on old versions of iOS without in-app browser tab functionality.

To receive the authorization response, both private-use URI scheme (referred to as "custom URL scheme") redirects and claimed "https" scheme URIs (known as "Universal Links") are viable choices. Apps can claim private-use URI schemes with the "CFBundleURLTypes" key in

the application's property list file, "Info.plist", and "https" scheme URIs using the Universal Links feature with an entitlement file in the app and an association file hosted on the domain.

Claimed "https" scheme URIs are the preferred redirect 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 in-app browser tab 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, private-use URI schemes are broadly supported through Android Implicit Intents. Claimed "https" scheme 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

Both traditional and Universal Windows Platform (UWP) apps can perform authorization requests in the user's browser. Traditional apps typically use a loopback redirect to receive the authorization response, and listening on the loopback interface is allowed by default firewall rules. When creating the loopback network socket, apps SHOULD set the "SO_EXCLUSIVEADDRUSE" socket option to prevent other apps binding to the same socket.

UWP apps can use private-use URI scheme redirects to receive the authorization response from the browser, which will bring the app to the foreground. Known on the platform as "URI Activation", the URI

scheme is limited to 39 characters in length, and it may include the "." character, making short reverse domain name based schemes (as required in Section 7.1) possible.

UWP apps can alternatively use the Web Authentication Broker API in Single Sign-on (SSO) mode, which is an external user-agent designed for authorization flows. Cookies are shared between invocations of the broker but not the user's preferred browser, meaning the user will need to log in again, even if they have an active session in their browser; but the session created in the broker will be available to subsequent apps that use the broker. Personalizations the user has made to their browser, such as configuring a password manager, may not be available in the broker. To qualify as an external user-agent, the broker MUST be used in SSO mode.

To use the Web Authentication Broker in SSO mode, the redirect URI must be of the form "msapp://{appSID}" where "{appSID}" is the app's security identifier (SID), which can be found in the app's registration information or by calling the "GetCurrentApplicationCallbackUri" method. While Windows enforces the URI authority on such redirects, ensuring that only the app with the matching SID can receive the response on Windows, the URI scheme could be claimed by apps on other platforms without the same authority present; thus, this redirect type should be treated similarly to private-use URI scheme redirects for security purposes.

An open-source sample demonstrating these patterns is available [SamplesForWindows].

B.4. macOS Implementation Details

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

To receive the authorization response, private-use URI schemes 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. Apps SHOULD NOT set the "SO_REUSEPORT" or "SO_REUSEADDR" socket options in order to prevent other apps binding to the same socket.

Acknowledgements

The authors would like to acknowledge the work of Marius Scurtescu and Ben Wiley Sittler, whose design for using private-use URI schemes in native app OAuth 2.0 clients at Google 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, Nat Sakimura, Eric Sachs, Paul Madsen, Iain McGinniss, Rahul Ravikumar, Breno de Medeiros, Hannes Tschofenig, Ashish Jain, Erik Wahlstrom, Bill Fisher, Sudhi Umarji, Michael B. Jones, Vittorio Bertocci, Dick Hardt, David Waite, Ignacio Fiorentino, Kathleen Moriarty, and Elwyn Davies.

Authors' Addresses

William Denniss
Google
1600 Amphitheatre Pkwy
Mountain View, CA 94043
United States of America

Email: rfc8252@wdenniss.com
URI: http://wdenniss.com/appauth

John Bradley
Ping Identity

Phone: +1 202-630-5272
Email: rfc8252@ve7jtb.com
URI: http://www.thread-safe.com/p/appauth.html