

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 13, 2017

B. Campbell
J. Bradley
Ping Identity
October 10, 2016

Mutual X.509 Transport Layer Security (TLS) Authentication for OAuth
Clients
draft-campbell-oauth-tls-client-auth-00

Abstract

This document describes X.509 certificates as OAuth client credentials using Transport Layer Security (TLS) mutual authentication as a mechanism for client authentication to the authorization server's token endpoint.

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 April 13, 2017.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Requirements Notation and Conventions	2
2. Mutual TLS for Client Authentication	2
3. Metadata	3
4. IANA Considerations	3
4.1. Token Endpoint Authentication Method Registration	3
4.1.1. Registry Contents	3
5. Security Considerations	3
5.1. TLS Versions and Best Practices	3
5.2. Client Identity Binding	4
6. References	4
6.1. Normative References	4
6.2. Informative References	4
Appendix A. Acknowledgements	5
Appendix B. Document History	5
Authors' Addresses	5

1. Introduction

The OAuth 2.0 Authorization Framework [RFC6749] defines a shared secret method of client authentication but also allows for the definition and use of additional client authentication mechanisms when interacting with the authorization server's token endpoint. This document describes an additional mechanism of client authentication utilizing mutual TLS [RFC5246] certificate-based authentication.

1.1. Requirements Notation and 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 RFC 2119 [RFC2119].

2. Mutual TLS for Client Authentication

The following section defines, as an extension of OAuth 2.0, Section 2.3 [RFC6749], the use of mutual TLS as client credentials. OAuth 2.0 requires that access token requests by the client to the token endpoint use TLS. In order to utilize TLS for client authentication, the TLS connection MUST have been established or reestablished with mutual X.509 certificate authentication (i.e. the Client Certificate and Certificate Verify messages are sent during the TLS Handshake [RFC5246]).

For all access token requests to the token endpoint, regardless of the grant type used, the client MUST include the "client_id" parameter, described in OAuth 2.0, Section 2.2 [RFC6749]. The presence of the "client_id" parameter enables the authorization server to easily identify the client independently from the content of the certificate and allows for trust models to vary as appropriate for a given deployment. The authorization server can locate the client configuration by the client identifier and check the certificate presented in the TLS Handshake against the expected credentials for that client.

3. Metadata

The value "tls_client_auth" is used to indicate mutual TLS as an authentication method to the token endpoint for the "token_endpoint_auth_methods_supported" client metadata field defined in [RFC7591], Section 2.

The same "tls_client_auth" value can also indicate server support for mutual TLS as a client authentication method in authorization server metadata such as [OpenID.Discovery] and [I-D.ietf-oauth-discovery].

4. IANA Considerations

4.1. Token Endpoint Authentication Method Registration

This specification requests registration of the following value in the IANA "OAuth Token Endpoint Authentication Methods" registry [IANA.OAuthTEAuthnMeths] established by [RFC7591].

4.1.1. Registry Contents

- o Token Endpoint Authentication Method Name: "tls_client_auth"
- o Change Controller: IESG
- o Specification Document(s): [[this specification]]

5. Security Considerations

5.1. TLS Versions and Best Practices

TLS 1.2 [RFC5246] is cited in this document because, at the time of writing, it is latest version that is widely deployed. However, this document is applicable with other TLS versions supporting certificate-based client authentication. Implementation security considerations for TLS, including version recommendations, can be found in Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) [BCP195].

5.2. Client Identity Binding

No specific method of binding a certificate to a client identifier is prescribed by this document. However, some method should be employed so that, in addition to proving possession of the private key corresponding to the certificate, the client identity is also bound to the certificate. One such binding would be to configure for the client a value that the certificate must contain in the subject field or the subjectAltName extension and possibly a restricted set of trust anchors. An alternative method would be to configure a public key for the client directly that would have to match the subject public key info of the certificate.

6. References

6.1. Normative References

- [BCP195] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <<http://www.rfc-editor.org/info/bcp195>>.
- [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>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<http://www.rfc-editor.org/info/rfc5246>>.
- [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>>.

6.2. Informative References

- [I-D.ietf-oauth-discovery] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", draft-ietf-oauth-discovery-04 (work in progress), August 2016.
- [IANA.OAuthTEAuthnMeths] IANA, "OAuth Token Endpoint Authentication Methods", <<http://www.iana.org/assignments/oauth-parameters/oauth-parameters.xhtml#token-endpoint-auth-method>>.

[OpenID.Discovery]

Sakimura, N., Bradley, J., Jones, M., and E. Jay, "OpenID Connect Discovery 1.0", February 2014.

[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, <<http://www.rfc-editor.org/info/rfc7591>>.

Appendix A. Acknowledgements

Scott "not Tomlinson" Tomilson and Matt Peterson were involved in the original design and implementation work that informed the content of this document.

Appendix B. Document History

[[to be removed by the RFC Editor before publication as an RFC]]

-00

o Initial draft.

Authors' Addresses

Brian Campbell
Ping Identity

Email: brian.d.campbell@gmail.com

John Bradley
Ping Identity

Email: ve7jtb@ve7jtb.com
URI: <http://www.thread-safe.com/>

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2017

M. Jones
Microsoft
P. Hunt
Oracle
A. Nadalin
Microsoft
March 13, 2017

Authentication Method Reference Values
draft-ietf-oauth-amr-values-08

Abstract

The "amr" (Authentication Methods References) claim is defined and registered in the IANA "JSON Web Token Claims" registry but no standard Authentication Method Reference values are currently defined. This specification establishes a registry for Authentication Method Reference values and defines an initial set of Authentication Method Reference values.

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 September 14, 2017.

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 (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect

to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Requirements Notation and Conventions	4
1.2. Terminology	4
2. Authentication Method Reference Values	4
3. Relationship to "acr" (Authentication Context Class Reference)	6
4. Privacy Considerations	6
5. Security Considerations	7
6. IANA Considerations	7
6.1. Authentication Method Reference Values Registry	7
6.1.1. Registration Template	8
6.1.2. Initial Registry Contents	9
7. References	11
7.1. Normative References	11
7.2. Informative References	12
Appendix A. Examples	13
Appendix B. Acknowledgements	14
Appendix C. Document History	14
Authors' Addresses	15

1. Introduction

The "amr" (Authentication Methods References) claim is defined and registered in the IANA "JSON Web Token Claims" registry [IANA.JWT.Claims] but no standard Authentication Method Reference values are currently defined. This specification establishes a registry for Authentication Method Reference values and defines an initial set of Authentication Method Reference values.

For context, the "amr" (Authentication Methods References) claim is defined by Section 2 of the OpenID Connect Core 1.0 specification [OpenID.Core] as follows:

amr

OPTIONAL. Authentication Methods References. JSON array of strings that are identifiers for authentication methods used in the authentication. For instance, values might indicate that both password and OTP authentication methods were used. The definition of particular values to be used in the "amr" Claim is beyond the scope of this specification. Parties using this claim will need to agree upon the meanings of the values used, which may be

context-specific. The "amr" value is an array of case sensitive strings.

Each "amr" value typically provides an identifier for a family of closely-related authentication methods. For example, the "otp" identifier intentionally covers both time-based and HMAC-based OTPs. Many relying parties will be content to know that an OTP has been used in addition to a password; the distinction between which kind of OTP was used is not useful to them. Thus, there's a single identifier that can be satisfied in two or more nearly equivalent ways.

Similarly, there's a whole range of nuances between different fingerprint matching algorithms. They differ in false positive and false negative rates over different population samples and also differ based on the kind and model of fingerprint sensor used. Like the OTP case, many relying parties will be content to know that a fingerprint match was made, without delving into and differentiating based on every aspect of the implementation of fingerprint capture and match. The "fpt" identifier accomplishes this.

Ultimately, the relying party is depending upon the identity provider to do reasonable things. If it does not trust the identity provider to do so, it has no business using it. The "amr" value lets the identity provider signal to the relying party additional information about what it did, for the cases in which that information is useful to the relying party.

The "amr" values defined by this specification are not intended to be an exhaustive set covering all use cases. Additional values can and will be added to the registry by other specifications. Rather, the values defined herein are an intentionally small set that are already actually being used in practice.

The values defined by this specification only make distinctions that are known to be useful to relying parties. Slicing things more finely than would be used in practice would actually hurt interoper, rather than helping it, because it would force relying parties to recognize that several or many different values actually mean the same thing to them.

For context, while the claim values registered pertain to authentication, note that OAuth 2.0 [RFC6749] is designed for resource authorization and cannot be used for authentication without employing appropriate extensions, such as those defined by OpenID Connect Core 1.0 [OpenID.Core]. The existence of the "amr" claim and values for it should not be taken as encouragement to try to use

OAuth 2.0 for authentication without employing extensions enabling secure authentication to be performed.

When used with OpenID Connect, if the identity provider supplies an "amr" claim in the ID Token resulting from a successful authentication, the relying party can inspect the values returned and thereby learn details about how the authentication was performed. For instance, the relying party might learn that only a password was used or it might learn that iris recognition was used in combination with a hardware-secured key. Whether "amr" values are provided and which values are understood by what parties are both beyond the scope of this specification. The OpenID Connect MODRNA Authentication Profile 1.0 [OpenID.MODRNA] is one example of an application context that uses "amr" values defined by this specification.

1.1. Requirements Notation and 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 RFC 2119 [RFC2119].

1.2. Terminology

This specification uses the terms defined by JSON Web Token (JWT) [JWT] and OpenID Connect Core 1.0 [OpenID.Core].

2. Authentication Method Reference Values

The following is a list of Authentication Method Reference values defined by this specification:

face

Biometric authentication [RFC4949] using facial recognition

fp

Biometric authentication [RFC4949] using a fingerprint

geo

Use of geolocation information for authentication, such as that provided by [W3C.REC-geolocation-API-20161108]

hwk

Proof-of-possession (PoP) of a hardware-secured key. See Appendix C of [RFC4211] for a discussion on PoP.

iris

Biometric authentication [RFC4949] using an iris scan

- kba**
Knowledge-based authentication [NIST.800-63-2] [ISO29115]
- mca**
Multiple-channel authentication [MCA]. The authentication involves communication over more than one distinct communication channel. For instance, a multiple-channel authentication might involve both entering information into a workstation's browser and providing information on a telephone call to a pre-registered number.
- mfa**
Multiple-factor authentication [NIST.800-63-2] [ISO29115]. When this is present, specific authentication methods used may also be included.
- otp**
One-time password [RFC4949]. One-time password specifications that this authentication method applies to include [RFC4226] and [RFC6238].
- pin**
Personal Identification Number (PIN) [RFC4949] or pattern (not restricted to containing only numbers) that a user enters to unlock a key on the device. This mechanism should have a way to deter an attacker from obtaining the PIN by trying repeated guesses.
- pwd**
Password-based authentication [RFC4949]
- rba**
Risk-based authentication [JECM]
- retina**
Biometric authentication [RFC4949] using a retina scan
- sc**
Smart card [RFC4949]
- sms**
Confirmation using SMS [SMS] text message to the user at a registered number
- swk**
Proof-of-possession (PoP) of a software-secured key. See Appendix C of [RFC4211] for a discussion on PoP.

tel

Confirmation by telephone call to the user at a registered number. This authentication technique is sometimes also referred to as "call back" [RFC4949].

user

User presence test. Evidence that the end-user is present and interacting with the device. This is sometimes also referred to as "test of user presence" [W3C.WD-webauthn-20170216].

vbm

Biometric authentication [RFC4949] using a voiceprint

wia

Windows integrated authentication [MSDN]

3. Relationship to "acr" (Authentication Context Class Reference)

The "acr" (Authentication Context Class Reference) claim and "acr_values" request parameter are related to the "amr" (Authentication Methods References) claim, but with important differences. An Authentication Context Class specifies a set of business rules that authentications are being requested to satisfy. These rules can often be satisfied by using a number of different specific authentication methods, either singly or in combination. Interactions using "acr_values" request that the specified Authentication Context Classes be used and that the result should contain an "acr" claim saying which Authentication Context Class was satisfied. The "acr" claim in the reply states that the business rules for the class were satisfied -- not how they were satisfied.

In contrast, interactions using the "amr" claim make statements about the particular authentication methods that were used. This tends to be more brittle than using "acr", since the authentication methods that may be appropriate for a given authentication will vary over time, both because of the evolution of attacks on existing methods and the deployment of new authentication methods.

4. Privacy Considerations

The list of "amr" claim values returned in an ID Token reveals information about the way that the end-user authenticated to the identity provider. In some cases, this information may have privacy implications.

While this specification defines identifiers for particular kinds of credentials, it does not define how these credentials are stored or protected. For instance, ensuring the security and privacy of

biometric credentials that are referenced by some of the defined Authentication Method Reference values is beyond the scope of this specification.

5. Security Considerations

The security considerations in OpenID Connect Core 1.0 [OpenID.Core] and OAuth 2.0 [RFC6749] and the OAuth 2.0 Threat Model [RFC6819] apply to applications using this specification.

As described in Section 3, taking a dependence upon particular authentication methods may result in brittle systems since the authentication methods that may be appropriate for a given authentication will vary over time.

6. IANA Considerations

6.1. Authentication Method Reference Values Registry

This specification establishes the IANA "Authentication Method Reference Values" registry for "amr" claim array element values. The registry records the Authentication Method Reference value and a reference to the specification that defines it. This specification registers the Authentication Method Reference values defined in Section 2.

Values are registered on an Expert Review [RFC5226] basis after a three-week review period on the `jwt-reg-review@ietf.org` mailing list, on the advice of one or more Designated Experts. To increase potential interoperability, the experts are requested to encourage registrants to provide the location of a publicly-accessible specification defining the values being registered, so that their intended usage can be more easily understood.

Registration requests sent to the mailing list for review should use an appropriate subject (e.g., "Request to register Authentication Method Reference value: otp").

Within the review period, the Designated Experts will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful. Registration requests that are undetermined for a period longer than 21 days can be brought to the IESG's attention (using the `iesg@ietf.org` mailing list) for resolution.

IANA must only accept registry updates from the Designated Experts and should direct all requests for registration to the review mailing list.

It is suggested that the same Designated Experts evaluate these registration requests as those who evaluate registration requests for the IANA "JSON Web Token Claims" registry [IANA.JWT.Claims].

Criteria that should be applied by the Designated Experts includes determining whether the proposed registration duplicates existing functionality, whether it is likely to be of general applicability or whether it is useful only for a single application, whether the value is actually being used, and whether the registration description is clear.

6.1.1.1. Registration Template

Authentication Method Reference Name:

The name requested (e.g., "otp") for the authentication method or family of closely-related authentication methods. Because a core goal of this specification is for the resulting representations to be compact, it is RECOMMENDED that the name be short -- that is, not to exceed 8 characters without a compelling reason to do so. To facilitate interoperability, the name must use only printable ASCII characters excluding double quote ('"') and backslash ('\') (the Unicode characters with code points U+0021, U+0023 through U+005B, and U+005D through U+007E). This name is case sensitive. Names may not match other registered names in a case-insensitive manner unless the Designated Experts state that there is a compelling reason to allow an exception.

Authentication Method Reference Description:

Brief description of the Authentication Method Reference (e.g., "One-time password").

Change Controller:

For Standards Track RFCs, state "IESG". For others, give the name of the responsible party. Other details (e.g., postal address, email address, home page URI) may also be included.

Specification Document(s):

Reference to the document or documents that specify the parameter, preferably including URIs that can be used to retrieve copies of the documents. An indication of the relevant sections may also be included but is not required.

6.1.2. Initial Registry Contents

- o Authentication Method Reference Name: "face"
- o Authentication Method Reference Description: Facial recognition
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Authentication Method Reference Name: "fpt"
- o Authentication Method Reference Description: Fingerprint biometric
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Authentication Method Reference Name: "geo"
- o Authentication Method Reference Description: Geolocation
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Authentication Method Reference Name: "hwk"
- o Authentication Method Reference Description: Proof-of-possession of a hardware-secured key
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Authentication Method Reference Name: "iris"
- o Authentication Method Reference Description: Iris scan biometric
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Authentication Method Reference Name: "kba"
- o Authentication Method Reference Description: Knowledge-based authentication
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Authentication Method Reference Name: "mca"
- o Authentication Method Reference Description: Multiple-channel authentication
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Authentication Method Reference Name: "mfa"
- o Authentication Method Reference Description: Multiple-factor authentication
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Authentication Method Reference Name: "otp"
- o Authentication Method Reference Description: One-time password

- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Authentication Method Reference Name: "pin"
- o Authentication Method Reference Description: Personal Identification Number or pattern
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Authentication Method Reference Name: "pwd"
- o Authentication Method Reference Description: Password-based authentication
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Authentication Method Reference Name: "rba"
- o Authentication Method Reference Description: Risk-based authentication
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Authentication Method Reference Name: "retina"
- o Authentication Method Reference Description: Retina scan biometric
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Authentication Method Reference Name: "sc"
- o Authentication Method Reference Description: Smart card
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Authentication Method Reference Name: "sms"
- o Authentication Method Reference Description: Confirmation using SMS
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Authentication Method Reference Name: "swk"
- o Authentication Method Reference Description: Proof-of-possession of a software-secured key
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Authentication Method Reference Name: "tel"
- o Authentication Method Reference Description: Confirmation by telephone call
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Authentication Method Reference Name: "user"
- o Authentication Method Reference Description: User presence test
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Authentication Method Reference Name: "vbm"
- o Authentication Method Reference Description: Voice biometric
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Authentication Method Reference Name: "wia"
- o Authentication Method Reference Description: Windows integrated authentication
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

7. References

7.1. Normative References

- [IANA.JWT.Claims] IANA, "JSON Web Token Claims",
 <<http://www.iana.org/assignments/jwt>>.
- [JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, May 2015,
 <<http://www.rfc-editor.org/info/rfc7519>>.
- [OpenID.Core] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", November 2014,
 <http://openid.net/specs/openid-connect-core-1_0.html>.
- [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>>.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008,
 <<http://www.rfc-editor.org/info/rfc5226>>.
- [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>>.

7.2. Informative References

- [ISO29115] International Organization for Standardization, "ISO/IEC 29115:2013 -- Information technology - Security techniques - Entity authentication assurance framework", ISO/IEC 29115:2013, April 2013, <http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=45138>.
- [JECM] Williamson, G., "Enhanced Authentication In Online Banking", Journal of Economic Crime Management 4.2: 18-19, 2006, <<http://utica.edu/academic/institutes/ecii/publications/articles/51D6D996-90F2-F468-AC09C4E8071575AE.pdf>>.
- [MCA] ldapwiki.com, "Multiple-channel Authentication", August 2016, <<https://www.ldapwiki.com/wiki/Multiple-channel%20Authentication>>.
- [MSDN] Microsoft, "Integrated Windows Authentication with Negotiate", September 2011, <<http://blogs.msdn.com/b/benjaminperkins/archive/2011/09/14/iis-integrated-windows-authentication-with-negotiate.aspx>>.
- [NIST.800-63-2] National Institute of Standards and Technology (NIST), "Electronic Authentication Guideline", NIST Special Publication 800-63-2, August 2013, <<http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63-2.pdf>>.
- [OpenID.MODRNa] Connotte, J. and J. Bradley, "OpenID Connect MODRNa Authentication Profile 1.0", March 2017, <http://openid.net/specs/openid-connect-modrna-authentication-1_0.html>.
- [RFC4211] Schaad, J., "Internet X.509 Public Key Infrastructure Certificate Request Message Format (CRMF)", RFC 4211, DOI 10.17487/RFC4211, September 2005, <<http://www.rfc-editor.org/info/rfc4211>>.
- [RFC4226] M'Raihi, D., Bellare, M., Hoornaert, F., Naccache, D., and O. Ranen, "HOTP: An HMAC-Based One-Time Password Algorithm", RFC 4226, DOI 10.17487/RFC4226, December 2005, <<http://www.rfc-editor.org/info/rfc4226>>.

- [RFC4949] Shirey, R., "Internet Security Glossary, Version 2", FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007, <<http://www.rfc-editor.org/info/rfc4949>>.
- [RFC6238] M'Raihi, D., Machani, S., Pei, M., and J. Rydell, "TOTP: Time-Based One-Time Password Algorithm", RFC 6238, DOI 10.17487/RFC6238, May 2011, <<http://www.rfc-editor.org/info/rfc6238>>.
- [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>>.
- [SMS] 3rd Generation Partnership Project, "Technical realization of the Short Message Service (SMS)", 3GPP Technical Specification (TS) 03.40 V7.5.0 (2001-12), January 2002, <<https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=141>>.
- [W3C.REC-geolocation-API-20161108] Popescu, A., "Geolocation API Specification 2nd Edition", World Wide Web Consortium Recommendation REC-geolocation-API-20161108, November 2016, <<https://www.w3.org/TR/2016/REC-geolocation-API-20161108>>.
- [W3C.WD-webauthn-20170216] Bharadwaj, V., Le Van Gong, H., Balfanz, D., Czeskis, A., Birgisson, A., Hodges, J., Jones, M., Lindemann, R., and J. Jones, "Web Authentication: An API for accessing Scoped Credentials", World Wide Web Consortium Working Draft WD-webauthn-20170216, February 2017, <<http://www.w3.org/TR/2017/WD-webauthn-20170216/>>.

Appendix A. Examples

In some cases, the "amr" claim value returned may contain a single Authentication Method Reference value. For example, the following "amr" claim value indicates that the authentication performed used an iris scan biometric:

```
"amr": ["iris"]
```

In other cases, the "amr" claim value returned may contain multiple Authentication Method Reference values. For example, the following "amr" claim value indicates that the authentication performed used a password and knowledge-based authentication:

```
"amr": ["pwd", "kba"]
```

Appendix B. Acknowledgements

Caleb Baker participated in specifying the original set of "amr" values. Jari Arkko, John Bradley, Ben Campbell, Brian Campbell, William Denniss, Linda Dunbar, Stephen Farrell, Paul Kyzivat, Elaine Newton, James Manger, Catherine Meadows, Alexey Melnikov, Kathleen Moriarty, Nat Sakimura, and Mike Schwartz provided reviews of the specification.

Appendix C. Document History

[[to be removed by the RFC editor before publication as an RFC]]

-08

- o Added text in the IANA Registration Template saying that names can be for families of closely-related authentication methods, as suggested by Stephen Farrell.

-07

- o Clarified that the values are intended to provide identifiers for families of closely-related authentication methods.
- o Updated the MODRNA Authentication Profile reference.

-06

- o Addressed IESG comments. Identifiers are now restricted to using only printable JSON-friendly ASCII characters. Additional references to documentation relevant to specific AMR values were added.

-05

- o Specified characters allowed in "amr" values, reusing the IANA Considerations language on this topic from RFC 7638.

-04

- o Added examples with single and multiple values.
- o Clarified that the actual credentials referenced are not part of this specification to avoid additional privacy concerns for biometric data.
- o Clarified that the OAuth 2.0 Threat Model [RFC6819] applies to applications using this specification.

-03

- o Addressed shepherd comments.

-02

- o Addressed working group last call comments.

-01

- o Distinguished between retina and iris biometrics.
- o Expanded the introduction to provide additional context to readers.
- o Referenced the OpenID Connect MODRMA Authentication Profile 1.0 specification, which uses "amr" values defined by this specification.

-00

- o Created the initial working group draft from draft-jones-oauth-amr-values-05 with no normative changes.

Authors' Addresses

Michael B. Jones
Microsoft

Email: mbj@microsoft.com
URI: <http://self-issued.info/>

Phil Hunt
Oracle

Email: phil.hunt@yahoo.com

Anthony Nadalin
Microsoft

Email: tonynad@microsoft.com

OAuth
Internet-Draft
Intended status: Standards Track
Expires: September 12, 2019

W. Denniss
Google
J. Bradley
Ping Identity
M. Jones
Microsoft
H. Tschofenig
ARM Limited
March 11, 2019

OAuth 2.0 Device Authorization Grant
draft-ietf-oauth-device-flow-15

Abstract

The OAuth 2.0 Device Authorization Grant is designed for internet-connected devices that either lack a browser to perform a user-agent based authorization, or are input-constrained to the extent that requiring the user to input text in order to authenticate during the authorization flow is impractical. It enables OAuth clients on such devices (like smart TVs, media consoles, digital picture frames, and printers) to obtain user authorization to access protected resources without using an on-device user-agent.

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 <https://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 September 12, 2019.

Copyright Notice

Copyright (c) 2019 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. Terminology	5
3. Protocol	5
3.1. Device Authorization Request	5
3.2. Device Authorization Response	7
3.3. User Interaction	8
3.3.1. Non-textual Verification URI Optimization	9
3.4. Device Access Token Request	10
3.5. Device Access Token Response	11
4. Discovery Metadata	12
5. Security Considerations	12
5.1. User Code Brute Forcing	13
5.2. Device Code Brute Forcing	13
5.3. Device Trustworthiness	14
5.4. Remote Phishing	14
5.5. Session Spying	15
5.6. Non-confidential Clients	15
5.7. Non-Visual Code Transmission	15
6. Usability Considerations	15
6.1. User Code Recommendations	16
6.2. Non-Browser User Interaction	17
7. IANA Considerations	17
7.1. OAuth Parameters Registration	17
7.1.1. Registry Contents	17
7.2. OAuth URI Registration	17
7.2.1. Registry Contents	17
7.3. OAuth Extensions Error Registration	17
7.3.1. Registry Contents	18
7.4. OAuth 2.0 Authorization Server Metadata	18
7.4.1. Registry Contents	18
8. Normative References	18
Appendix A. Acknowledgements	19
Appendix B. Document History	20
Authors' Addresses	22

1. Introduction

This OAuth 2.0 [RFC6749] protocol extension, sometimes referred to as "device flow", enables OAuth clients to request user authorization from applications on devices that have limited input capabilities or lack a suitable browser. Such devices include those smart TVs, media console, picture frames and printers which lack an easy input method or suitable browser required for traditional OAuth interactions. The authorization flow defined by this specification instructs the user to review the authorization request on a secondary device, such as a smartphone which does have the requisite input and browser capabilities to complete the user interaction.

The Device Authorization Grant is not intended to replace browser-based OAuth in native apps on capable devices like smartphones. Those apps should follow the practices specified in OAuth 2.0 for Native Apps [RFC8252].

The operating requirements to be able to use this authorization grant type are:

- (1) The device is already connected to the Internet.
- (2) The device is able to make outbound HTTPS requests.
- (3) The device is able to display or otherwise communicate a URI and code sequence to the user.
- (4) The user has a secondary device (e.g., personal computer or smartphone) from which they can process the request.

As the device authorization grant does not require two-way communication between the OAuth client and the user-agent (unlike other OAuth 2 grant types such as the Authorization Code and Implicit grant types), it supports several use cases that cannot be served by those other approaches.

Instead of interacting with the end user's user agent, the client instructs the end user to use another computer or device and connect to the authorization server to approve the access request. Since the protocol supports clients that can't receive incoming requests, clients poll the authorization server repeatedly until the end user completes the approval process.

The device typically chooses the set of authorization servers to support (i.e., its own authorization server, or those by providers it has relationships with). It is not uncommon for the device application to support only a single authorization server, such as

with a TV application for a specific media provider that supports only that media provider's authorization server. The user may not have an established relationship yet with that authorization provider, though one can potentially be set up during the authorization flow.

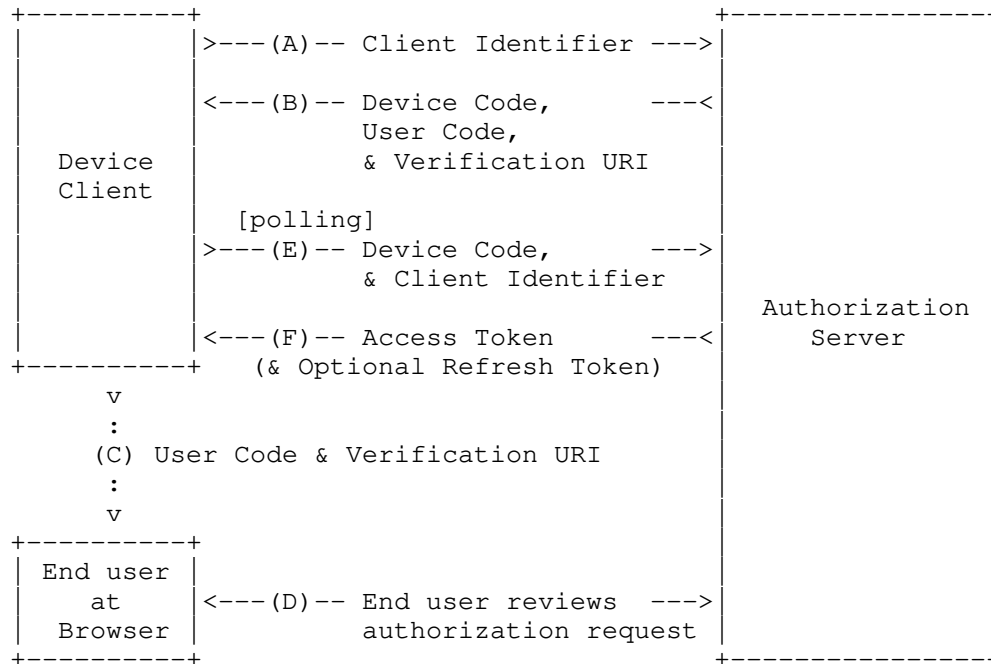


Figure 1: Device Authorization Flow

The device authorization flow illustrated in Figure 1 includes the following steps:

- (A) The client requests access from the authorization server and includes its client identifier in the request.
- (B) The authorization server issues a device code, an end-user code, and provides the end-user verification URI.
- (C) The client instructs the end user to use its user agent (on another device) and visit the provided end-user verification URI. The client provides the user with the end-user code to enter in order to review the authorization request.
- (D) The authorization server authenticates the end user (via the user agent) and prompts the user to grant the client's access

request. If the user agrees to the client's access request, the user enters the user code provided by the client. The authorization server validates the user code provided by the user.

(E) While the end user reviews the client's request (step D), the client repeatedly polls the authorization server to find out if the user completed the user authorization step. The client includes the verification code and its client identifier.

(F) The authorization server validates the verification code provided by the client and responds back with the access token if the user granted access, an error if they denied access, or indicates that the client should continue to poll.

2. Terminology

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.

Device Authorization Endpoint:

The authorization server's endpoint capable of issuing device verification codes, user codes, and verification URLs.

Device Verification Code:

A short-lived token representing an authorization session.

End-User Verification Code:

A short-lived token which the device displays to the end user, is entered by the user on the authorization server, and is thus used to bind the device to the user.

3. Protocol

3.1. Device Authorization Request

This specification defines a new OAuth endpoint, the device authorization endpoint. This is separate from the OAuth authorization endpoint defined in [RFC6749] with which the user interacts with via a user-agent (i.e., a browser). By comparison, when using the device authorization endpoint, the OAuth client on the device interacts with the authorization server directly without presenting the request in a user-agent, and the end user authorizes the request on a separate device. This interaction is defined as follows.

The client initiates the authorization flow by requesting a set of verification codes from the authorization server by making an HTTP "POST" request to the device authorization endpoint.

The client constructs the request with the following parameters, sent as the body of the request, encoded with the "application/x-www-form-urlencoded" encoding algorithm defined by Section 4.10.22.6 of [HTML5]:

`client_id`

REQUIRED, if the client is not authenticating with the authorization server as described in Section 3.2.1. of [RFC6749].
The client identifier as described in Section 2.2 of [RFC6749].

`scope`

OPTIONAL. The scope of the access request as described by Section 3.3 of [RFC6749].

For example, the client makes the following HTTPS request:

```
POST /device_authorization HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

client_id=459691054427
```

All requests from the device MUST use the Transport Layer Security (TLS) [RFC8446] protocol and implement the best practices of BCP 195 [RFC7525].

Parameters sent without a value MUST be treated as if they were omitted from the request. The authorization server MUST ignore unrecognized request parameters. Request and response parameters MUST NOT be included more than once.

The client authentication requirements of Section 3.2.1 of [RFC6749] apply to requests on this endpoint, which means that confidential clients (those that have established client credentials) authenticate in the same manner as when making requests to the token endpoint, and public clients provide the "client_id" parameter to identify themselves.

Due to the polling nature of this protocol (as specified in Section 3.4), care is needed to avoid overloading the capacity of the token endpoint. To avoid unneeded requests on the token endpoint, the client SHOULD only commence a device authorization request when prompted by the user, and not automatically, such as when the app starts or when the previous authorization session expires or fails.

3.2. Device Authorization Response

In response, the authorization server generates a unique device verification code and an end-user code that are valid for a limited time and includes them in the HTTP response body using the "application/json" format [RFC8259] with a 200 (OK) status code. The response contains the following parameters:

device_code
REQUIRED. The device verification code.

user_code
REQUIRED. The end-user verification code.

verification_uri
REQUIRED. The end-user verification URI on the authorization server. The URI should be short and easy to remember as end users will be asked to manually type it into their user-agent.

verification_uri_complete
OPTIONAL. A verification URI that includes the "user_code" (or other information with the same function as the "user_code"), designed for non-textual transmission.

expires_in
REQUIRED. The lifetime in seconds of the "device_code" and "user_code".

interval
OPTIONAL. The minimum amount of time in seconds that the client SHOULD wait between polling requests to the token endpoint. If no value is provided, clients MUST use 5 as the default.

For example:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
  "device_code": "GmRhmhcxhwAzkoEqiMEg_DnyEysNkuNhszIySk9eS",
  "user_code": "WDJB-MJHT",
  "verification_uri": "https://example.com/device",
  "verification_uri_complete":
    "https://example.com/device?user_code=WDJB-MJHT",
  "expires_in": 1800,
  "interval": 5
}
```

In the event of an error (such as an invalidly configured client), the authorization server responds in the same way as the token endpoint specified in Section 5.2 of [RFC6749].

3.3. User Interaction

After receiving a successful Authorization Response, the client displays or otherwise communicates the "user_code" and the "verification_uri" to the end user and instructs them to visit the URI in a user agent on a secondary device (for example, in a browser on their mobile phone), and enter the user code.

```
Using a browser on another device, visit:
https://example.com/device

And enter the code:
WDJB-MJHT
```

Figure 2: Example User Instruction

The authorizing user navigates to the "verification_uri" and authenticates with the authorization server in a secure TLS-protected ([RFC8446]) session. The authorization server prompts the end user to identify the device authorization session by entering the "user_code" provided by the client. The authorization server should then inform the user about the action they are undertaking and ask them to approve or deny the request. Once the user interaction is complete, the server MAY inform the user to return to their device.

During the user interaction, the device continuously polls the token endpoint with the "device_code", as detailed in Section 3.4, until the user completes the interaction, the code expires, or another error occurs. The "device_code" is not intended for the end user directly, and thus should not be displayed during the interaction to avoid confusing the end user.

Authorization servers supporting this specification MUST implement a user interaction sequence that starts with the user navigating to "verification_uri" and continues with them supplying the "user_code" at some stage during the interaction. Other than that, the exact sequence and implementation of the user interaction is up to the authorization server, for example, the authorization server may enable new users to sign up for an account during the authorization flow, or add additional security verification steps.

It is NOT RECOMMENDED for authorization servers to include the user code in the verification URI ("verification_uri"), as this increases the length and complexity of the URI that the user must type. While the user must still type the same number of characters with the "user_code" separated, once they successfully navigate to the "verification_uri", any errors in entering the code can be highlighted by the authorization server to improve the user experience. The next section documents user interaction with "verification_uri_complete", which is designed to carry both pieces of information.

3.3.1. Non-textual Verification URI Optimization

When "verification_uri_complete" is included in the Authorization Response (Section 3.2), clients MAY present this URI in a non-textual manner using any method that results in the browser being opened with the URI, such as with QR (Quick Response) codes or NFC (Near Field Communication), to save the user typing the URI.

For usability reasons, it is RECOMMENDED for clients to still display the textual verification URI ("verification_uri") for users not able to use such a shortcut. Clients MUST still display the "user_code", as the authorization server will require the user to confirm it to disambiguate devices, or as a remote phishing mitigation (See Section 5.4).

If the user starts the user interaction by browsing to "verification_uri_complete", then the user interaction described in Section 3.3 is still followed, but with the optimization that the user does not need to type the "user_code". The server SHOULD display the "user_code" to the user and ask them to verify that it matches the "user_code" being displayed on the device, to confirm they are authorizing the correct device. As before, in addition to taking steps to confirm the identity of the device, the user should also be afforded the choice to approve or deny the authorization request.

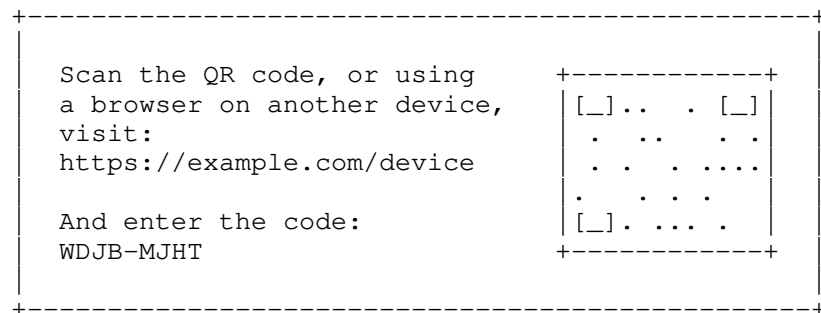


Figure 3: Example User Instruction with QR Code Representation of the Complete Verification URI

3.4. Device Access Token Request

After displaying instructions to the user, the client makes an Access Token Request to the token endpoint (as defined by Section 3.2 of [RFC6749]) with a "grant_type" of "urn:ietf:params:oauth:grant-type:device_code". This is an extension grant type (as defined by Section 4.5 of [RFC6749]) created by this specification, with the following parameters:

grant_type

REQUIRED. Value MUST be set to "urn:ietf:params:oauth:grant-type:device_code".

device_code

REQUIRED. The device verification code, "device_code" from the Device Authorization Response, defined in Section 3.2.

client_id

REQUIRED, if the client is not authenticating with the authorization server as described in Section 3.2.1. of [RFC6749]. The client identifier as described in Section 2.2 of [RFC6749].

For example, the client makes the following HTTPS request (line breaks are for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Adevice_code
&device_code=GmRhmcxhwAzkoEqiMEg_DnyEysNkuNhszIySk9eS
&client_id=459691054427
```

If the client was issued client credentials (or assigned other authentication requirements), the client MUST authenticate with the authorization server as described in Section 3.2.1 of [RFC6749]. Note that there are security implications of statically distributed client credentials, see Section 5.6.

The response to this request is defined in Section 3.5. Unlike other OAuth grant types, it is expected for the client to try the Access Token Request repeatedly in a polling fashion, based on the error code in the response.

3.5. Device Access Token Response

If the user has approved the grant, the token endpoint responds with a success response defined in Section 5.1 of [RFC6749]; otherwise it responds with an error, as defined in Section 5.2 of [RFC6749].

In addition to the error codes defined in Section 5.2 of [RFC6749], the following error codes are specified for use with the device authorization grant in token endpoint responses:

`authorization_pending`

The authorization request is still pending as the end user hasn't yet completed the user interaction steps (Section 3.3). The client SHOULD repeat the Access Token Request to the token endpoint (a process known as polling). Before each new request the client MUST wait at least the number of seconds specified by the "interval" parameter of the Device Authorization Response (see Section 3.2), or 5 seconds if none was provided, and respect any increase in the polling interval required by the "slow_down" error.

`slow_down`

A variant of "authorization_pending", the authorization request is still pending and polling should continue, but the interval MUST be increased by 5 seconds for this and all subsequent requests.

`access_denied`

The end user denied the authorization request.

expired_token

The "device_code" has expired and the device authorization session has concluded. The client MAY commence a new Device Authorization Request but SHOULD wait for user interaction before restarting to avoid unnecessary polling.

The "authorization_pending" and "slow_down" error codes define particularly unique behavior, as they indicate that the OAuth client should continue to poll the token endpoint by repeating the token request (implementing the precise behavior defined above). If the client receives an error response with any other error code, it MUST stop polling and SHOULD react accordingly, for example, by displaying an error to the user.

On encountering a connection timeout, clients MUST unilaterally reduce their polling frequency before retrying. The use of an exponential backoff algorithm to achieve this, such as by doubling the polling interval on each such connection timeout, is RECOMMENDED.

The assumption of this specification is that the separate device the user is authorizing the request on does not have a way to communicate back to device with the OAuth client. This protocol only requires a one-way channel in order to maximise the viability of the protocol in restricted environments, like an application running on a TV that is only capable of outbound requests. If a return channel were to exist for the chosen user interaction interface, then the device MAY wait until notified on that channel that the user has completed the action before initiating the token request (as an alternative to polling). Such behavior is, however, outside the scope of this specification.

4. Discovery Metadata

Support for this specification MAY be declared in the OAuth 2.0 Authorization Server Metadata [RFC8414] by including the value "urn:ietf:params:oauth:grant-type:device_code" in the "grant_types_supported" parameter, and by adding the following new parameter:

device_authorization_endpoint

OPTIONAL. URL of the authorization server's device authorization endpoint defined in Section 3.1.

5. Security Considerations

5.1. User Code Brute Forcing

Since the user code is typed by the user, shorter codes are more desirable for usability reasons. This means the entropy is typically less than would be used for the device code or other OAuth bearer token types where the code length does not impact usability. It is therefore recommended that the server rate-limit user code attempts.

The user code SHOULD have enough entropy that when combined with rate limiting and other mitigations makes a brute-force attack infeasible. For example, it's generally held that 128-bit symmetric keys for encryption are seen as good enough today because an attacker has to put in 2^{96} work to have a 2^{-32} chance of guessing correctly via brute force. The rate limiting and finite lifetime on the user code places an artificial limit on the amount of work an attacker can "do", so if, for instance, one uses a 8-character base-20 user code (with roughly 34.5 bits of entropy), the rate-limiting interval and validity period would need to only allow 5 attempts in order to get the same 2^{-32} probability of success by random guessing.

A successful brute forcing of the user code would enable the attacker to authenticate with their own credentials and make an authorization grant to the device. This is the opposite scenario to an OAuth bearer token being brute forced, whereby the attacker gains control of the victim's authorization grant. Such attacks may not always make economic sense, for example for a video app the device owner may then be able to purchase movies using the attacker's account, though a privacy risk would still remain and thus is important to protect against. Furthermore, some uses of the device flow give the granting account the ability to perform actions such as controlling the device, which needs to be protected.

The precise length of the user code and the entropy contained within is at the discretion of the authorization server, which needs to consider the sensitivity of their specific protected resources, the practicality of the code length from a usability standpoint, and any mitigations that are in place such as rate-limiting, when determining the user code format.

5.2. Device Code Brute Forcing

An attacker who guesses the device code would be able to potentially obtain the authorization code once the user completes the flow. As the device code is not displayed to the user and thus there are no usability considerations on the length, a very high entropy code SHOULD be used.

5.3. Device Trustworthiness

Unlike other native application OAuth 2.0 flows, the device requesting the authorization is not the same as the device that the user grants access from. Thus, signals from the approving user's session and device are not relevant to the trustworthiness of the client device.

Note that if an authorization server used with this flow is malicious, then it could man-in-the-middle the backchannel flow to another authorization server. In this scenario, the man-in-the-middle is not completely hidden from sight, as the end user would end up on the authorization page of the wrong service, giving them an opportunity to notice that the URL in the browser's address bar is wrong. For this to be possible, the device manufacturer must either directly be the attacker, shipping a device intended to perform the man-in-the-middle attack, or be using an authorization server that is controlled by an attacker, possibly because the attacker compromised the authorization server used by the device. In part, the person purchasing the device is counting on it and its business partners to be trustworthy.

5.4. Remote Phishing

It is possible for the device flow to be initiated on a device in an attacker's possession. For example, an attacker might send an email instructing the target user to visit the verification URL and enter the user code. To mitigate such an attack, it is RECOMMENDED to inform the user that they are authorizing a device during the user interaction step (see Section 3.3), and to confirm that the device is in their possession. The authorization server SHOULD display information about the device so that the person can notice if a software client was attempting to impersonating a hardware device.

For authorization servers that support the option specified in Section 3.3.1 for the client to append the user code to the authorization URI, it is particularly important to confirm that the device is in the user's possession, as the user no longer has to type the code manually. One possibility is to display the code during the authorization flow and asking the user to verify that the same code is being displayed on the device they are setting up.

The user code needs to have a long enough lifetime to be useable (allowing the user to retrieve their secondary device, navigate to the verification URI, login, etc.), but should be sufficiently short to limit the usability of a code obtained for phishing. This doesn't prevent a phisher presenting a fresh token, particularly in the case

they are interacting with the user in real time, but it does limit the viability of codes sent over email or SMS.

5.5. Session Spying

While the device is pending authorization, it may be possible for a malicious user to physically spy on the device user interface (by viewing the screen on which it's displayed, for example) and hijack the session by completing the authorization faster than the user that initiated it. Devices SHOULD take into account the operating environment when considering how to communicate the code to the user to reduce the chances it will be observed by a malicious user.

5.6. Non-confidential Clients

Device clients are generally incapable of maintaining the confidentiality of their credentials, as users in possession of the device can reverse engineer it and extract the credentials. Therefore, unless additional measures are taken, they should be treated as public clients (as defined by Section 2.1 of OAuth 2.0) susceptible to impersonation. The security considerations of Section 5.3.1 of [RFC6819] and Sections 8.5 and 8.6 of [RFC8252] apply to such clients.

The user may also be able to obtain the device_code and/or other OAuth bearer tokens issued to their client, which would allow them to use their own authorization grant directly by impersonating the client. Given that the user in possession of the client credentials can already impersonate the client and create a new authorization grant (with a new device_code), this doesn't represent a separate impersonation vector.

5.7. Non-Visual Code Transmission

There is no requirement that the user code be displayed by the device visually. Other methods of one-way communication can potentially be used, such as text-to-speech audio, or Bluetooth Low Energy. To mitigate an attack in which a malicious user can bootstrap their credentials on a device not in their control, it is RECOMMENDED that any chosen communication channel only be accessible by people in close proximity. E.g., users who can see, or hear the device.

6. Usability Considerations

This section is a non-normative discussion of usability considerations.

6.1. User Code Recommendations

For many users, their nearest Internet-connected device will be their mobile phone, and typically these devices offer input methods that are more time consuming than a computer keyboard to change the case or input numbers. To improve usability (improving entry speed, and reducing retries), these limitations should be taken into account when selecting the user-code character set.

One way to improve input speed is to restrict the character set to case-insensitive A-Z characters, with no digits. These characters can typically be entered on a mobile keyboard without using modifier keys. Further removing vowels to avoid randomly creating words results in the base-20 character set: "BCDFGHJKLMNPQRSTVWXZ". Dashes or other punctuation may be included for readability.

An example user code following this guideline containing 8 significant characters and dashes added for end-user readability, with a resulting entropy of 20^8 : "WDJB-MJHT".

Pure numeric codes are also a good choice for usability, especially for clients targeting locales where A-Z character keyboards are not used, though their length needs to be longer to maintain a high entropy.

An example numeric user code containing 9 significant digits and dashes added for end-user readability, with an entropy of 10^9 : "019-450-730".

When processing the inputted user code, the server should strip dashes and other punctuation it added for readability (making the inclusion of that punctuation by the user optional). For codes using only characters in the A-Z range as with the base-20 charset defined above, the user's input should be upper-cased before comparison to account for the fact that the user may input the equivalent lower-case characters. Further stripping of all characters outside the user_code charset is recommended to reduce instances where an errantly typed character (like a space character) invalidates otherwise valid input.

It is RECOMMENDED to avoid character sets that contain two or more characters that can easily be confused with each other like "0" and "O", or "1", "l" and "I". Furthermore, the extent practical, where a character set contains one character that may be confused with characters outside the character set the character outside the set MAY be substituted with the one in the character set that it is commonly confused with (for example, "O" for "0" when using a numerical 0-9 character set).

6.2. Non-Browser User Interaction

Devices and authorization servers MAY negotiate an alternative code transmission and user interaction method in addition to the one described in Section 3.3. Such an alternative user interaction flow could obviate the need for a browser and manual input of the code, for example, by using Bluetooth to transmit the code to the authorization server's companion app. Such interaction methods can utilize this protocol, as ultimately, the user just needs to identify the authorization session to the authorization server; however, user interaction other than via the verification URI is outside the scope of this specification.

7. IANA Considerations

7.1. OAuth Parameters Registration

This specification registers the following values in the IANA "OAuth Parameters" registry [IANA.OAuth.Parameters] established by [RFC6749].

7.1.1. Registry Contents

- o Parameter name: device_code
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification Document: Section 3.1 of [[this specification]]

7.2. OAuth URI Registration

This specification registers the following values in the IANA "OAuth URI" registry [IANA.OAuth.Parameters] established by [RFC6755].

7.2.1. Registry Contents

- o URN: urn:ietf:params:oauth:grant-type:device_code
- o Common Name: Device flow grant type for OAuth 2.0
- o Change controller: IESG
- o Specification Document: Section 3.1 of [[this specification]]

7.3. OAuth Extensions Error Registration

This specification registers the following values in the IANA "OAuth Extensions Error Registry" registry [IANA.OAuth.Parameters] established by [RFC6749].

7.3.1. Registry Contents

- o Error name: `authorization_pending`
- o Error usage location: Token endpoint response
- o Related protocol extension: `[[this specification]]`
- o Change controller: IETF
- o Specification Document: Section 3.5 of `[[this specification]]`

- o Error name: `access_denied`
- o Error usage location: Token endpoint response
- o Related protocol extension: `[[this specification]]`
- o Change controller: IETF
- o Specification Document: Section 3.5 of `[[this specification]]`

- o Error name: `slow_down`
- o Error usage location: Token endpoint response
- o Related protocol extension: `[[this specification]]`
- o Change controller: IETF
- o Specification Document: Section 3.5 of `[[this specification]]`

- o Error name: `expired_token`
- o Error usage location: Token endpoint response
- o Related protocol extension: `[[this specification]]`
- o Change controller: IETF
- o Specification Document: Section 3.5 of `[[this specification]]`

7.4. OAuth 2.0 Authorization Server Metadata

This specification registers the following values in the IANA "OAuth 2.0 Authorization Server Metadata" registry [IANA.OAuth.Parameters] established by [RFC8414].

7.4.1. Registry Contents

- o Metadata name: `device_authorization_endpoint`
- o Metadata Description: The Device Authorization Endpoint.
- o Change controller: IESG
- o Specification Document: Section 4 of `[[this specification]]`

8. Normative References

- [HTML5] IANA, "HTML5",
<<https://www.w3.org/TR/2014/REC-html5-20141028/>>.
- [IANA.OAuth.Parameters] IANA, "OAuth Parameters",
<<http://www.iana.org/assignments/oauth-parameters>>.

- [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>>.
- [RFC6755] Campbell, B. and H. Tschofenig, "An IETF URN Sub-Namespace for OAuth", RFC 6755, DOI 10.17487/RFC6755, October 2012, <<https://www.rfc-editor.org/info/rfc6755>>.
- [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>>.
- [RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <<https://www.rfc-editor.org/info/rfc7525>>.
- [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>>.
- [RFC8252] Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps", BCP 212, RFC 8252, DOI 10.17487/RFC8252, October 2017, <<https://www.rfc-editor.org/info/rfc8252>>.
- [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/info/rfc8259>>.
- [RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <<https://www.rfc-editor.org/info/rfc8414>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

Appendix A. Acknowledgements

The starting point for this document was the Internet-Draft draft-recordon-oauth-v2-device, authored by David Recordon and Brent Goldman, which itself was based on content in draft versions of the OAuth 2.0 protocol specification removed prior to publication due to

a then lack of sufficient deployment expertise. Thank you to the OAuth working group members who contributed to those earlier drafts.

This document was produced in the OAuth working group under the chairpersonship of Rifaat Shekh-Yusef and Hannes Tschofenig with Benjamin Kaduk, Kathleen Moriarty, and Eric Rescorla serving as Security Area Directors.

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

Alissa Cooper, Ben Campbell, Brian Campbell, Roshni Chandrashekhar, Eric Fazendin, Benjamin Kaduk, Jamshid Khosravian, Torsten Lodderstedt, James Manger, Dan McNulty, Breno de Medeiros, Simon Moffatt, Stein Myrseth, Emond Papegaaïj, Justin Richer, Adam Roach, Nat Sakimura, Andrew Sciberras, Marius Scurtescu, Filip Skokan, Ken Wang, and Steven E. Wright.

Appendix B. Document History

[[to be removed by the RFC Editor before publication as an RFC]]

-15

- o Renamed and dropped most usage of the term "flow"
- o Documented error responses on the authorization endpoint
- o Documented client authentication for the authorization endpoint

-14

- o Added more normative text on polling behavior.
- o Added discussion on risk of user retrieving their own device_code.
- o Editorial improvements.

-13

- o Added a longer discussion about entropy, proposed by Benjamin Kaduk.
- o Added device_code to OAuth IANA registry.
- o Expanded explanation of "case insensitive".
- o Added security section on Device Code Brute Forcing.
- o application/x-www-form-urlencoded normativly referenced.
- o Editorial improvements.

-12

- o Set a default polling interval to 5s explicitly.

- o Defined the `slow_down` behavior that it should increase the current interval by 5s.
- o `expires_in` now REQUIRED
- o Other changes in response to review feedback.

-11

- o Updated reference to OAuth 2.0 Authorization Server Metadata.

-10

- o Added a missing definition of `access_denied` for use on the token endpoint.
- o Corrected text documenting which error code should be returned for expired tokens (it's `"expired_token"`, not `"invalid_grant"`).
- o Corrected section reference to RFC 8252 (the section numbers had changed after the initial reference was made).
- o Fixed line length of one diagram (was causing xml2rfc warnings).
- o Added line breaks so the URN `grant_type` is presented on an unbroken line.
- o Typos fixed and other stylistic improvements.

-09

- o Addressed review comments by Security Area Director Eric Rescorla about the potential of a confused deputy attack.

-08

- o Expanded the User Code Brute Forcing section to include more detail on this attack.

-07

- o Replaced the `"user_code"` URI parameter optimization with `verification_uri_complete` following the IETF99 working group discussion.
- o Added security consideration about spying.
- o Required that `device_code` not be shown.
- o Added text regarding a minimum polling interval.

-06

- o Clarified usage of the `"user_code"` URI parameter optimization following the IETF98 working group discussion.

-05

- o response_type parameter removed from authorization request.
- o Added option for clients to include the user_code on the verification URI.
- o Clarified token expiry, and other nits.

-04

- o Security & Usability sections. OAuth Discovery Metadata.

-03

- o device_code is now a URN. Added IANA Considerations

-02

- o Added token request & response specification.

-01

- o Applied spelling and grammar corrections and added the Document History appendix.

-00

- o Initial working group draft based on draft-recordon-oauth-v2-device.

Authors' Addresses

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

Email: wdenniss@google.com
URI: <http://wdenniss.com/device-flow>

John Bradley
Ping Identity

Email: ve7jtb@ve7jtb.com
URI: <http://www.thread-safe.com/>

Michael B. Jones
Microsoft

Email: mbj@microsoft.com
URI: <http://self-issued.info/>

Hannes Tschofenig
ARM Limited
Austria

Email: Hannes.Tschofenig@gmx.net
URI: <http://www.tschofenig.priv.at>

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 5, 2018

M. Jones
Microsoft
N. Sakimura
NRI
J. Bradley
Ping Identity
March 4, 2018

OAuth 2.0 Authorization Server Metadata
draft-ietf-oauth-discovery-10

Abstract

This specification defines a metadata format that an OAuth 2.0 client can use to obtain the information needed to interact with an OAuth 2.0 authorization server, including its endpoint locations and authorization server capabilities.

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 <https://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 September 5, 2018.

Copyright Notice

Copyright (c) 2018 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	2
1.1. Requirements Notation and Conventions	3
1.2. Terminology	3
2. Authorization Server Metadata	4
2.1. Signed Authorization Server Metadata	7
3. Obtaining Authorization Server Metadata	8
3.1. Authorization Server Metadata Request	9
3.2. Authorization Server Metadata Response	9
3.3. Authorization Server Metadata Validation	10
4. String Operations	11
5. Compatibility Notes	11
6. Security Considerations	12
6.1. TLS Requirements	12
6.2. Impersonation Attacks	12
6.3. Publishing Metadata in a Standard Format	13
6.4. Protected Resources	13
7. IANA Considerations	13
7.1. OAuth Authorization Server Metadata Registry	14
7.1.1. Registration Template	15
7.1.2. Initial Registry Contents	15
7.2. Updated Registration Instructions	18
7.3. Well-Known URI Registry	19
7.3.1. Registry Contents	19
8. References	19
8.1. Normative References	19
8.2. Informative References	21
Appendix A. Acknowledgements	22
Appendix B. Document History	22
Authors' Addresses	25

1. Introduction

This specification generalizes the metadata format defined by "OpenID Connect Discovery 1.0" [OpenID.Discovery] in a way that is compatible with OpenID Connect Discovery, while being applicable to a wider set of OAuth 2.0 use cases. This is intentionally parallel to the way that the "OAuth 2.0 Dynamic Client Registration Protocol" [RFC7591] specification generalized the dynamic client registration mechanisms defined by "OpenID Connect Dynamic Client Registration 1.0" [OpenID.Registration] in a way that was compatible with it.

The metadata for an authorization server is retrieved from a well-known location as a JSON [RFC7159] document, which declares its

endpoint locations and authorization server capabilities. This process is described in Section 3.

This metadata can either be communicated in a self-asserted fashion by the server origin via HTTPS or as a set of signed metadata values represented as claims in a JSON Web Token (JWT) [JWT]. In the JWT case, the issuer is vouching for the validity of the data about the authorization server. This is analogous to the role that the Software Statement plays in OAuth Dynamic Client Registration [RFC7591].

The means by which the client chooses an authorization server is out of scope. In some cases, its issuer identifier may be manually configured into the client. In other cases, it may be dynamically discovered, for instance, through the use of WebFinger [RFC7033], as described in Section 2 of "OpenID Connect Discovery 1.0" [OpenID.Discovery].

1.1. Requirements Notation and 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.

All uses of JSON Web Signature (JWS) [JWS] and JSON Web Encryption (JWE) [JWE] data structures in this specification utilize the JWS Compact Serialization or the JWE Compact Serialization; the JWS JSON Serialization and the JWE JSON Serialization are not used.

1.2. Terminology

This specification uses the terms "Access Token", "Authorization Code", "Authorization Endpoint", "Authorization Grant", "Authorization Server", "Client", "Client Authentication", "Client Identifier", "Client Secret", "Grant Type", "Protected Resource", "Redirection URI", "Refresh Token", "Resource Owner", "Resource Server", "Response Type", and "Token Endpoint" defined by OAuth 2.0 [RFC6749], the terms "Claim Name", "Claim Value", and "JSON Web Token (JWT)" defined by JSON Web Token (JWT) [JWT], and the term "Response Mode" defined by OAuth 2.0 Multiple Response Type Encoding Practices [OAuth.Responses].

2. Authorization Server Metadata

Authorization servers can have metadata describing their configuration. The following authorization server metadata values are used by this specification and are registered in the IANA "OAuth Authorization Server Metadata" registry established in Section 7.1:

issuer

REQUIRED. The authorization server's issuer identifier, which is a URL that uses the "https" scheme and has no query or fragment components. Authorization server metadata is published at a ".well-known" RFC 5785 [RFC5785] location derived from this issuer identifier, as described in Section 3. The issuer identifier is used to prevent authorization server mix-up attacks, as described in "OAuth 2.0 Mix-Up Mitigation" [I-D.ietf-oauth-mix-up-mitigation].

authorization_endpoint

URL of the authorization server's authorization endpoint [RFC6749]. This is REQUIRED unless no grant types are supported that use the authorization endpoint.

token_endpoint

URL of the authorization server's token endpoint [RFC6749]. This is REQUIRED unless only the implicit grant type is supported.

jwks_uri

OPTIONAL. URL of the authorization server's JWK Set [JWK] document. The referenced document contains the signing key(s) the client uses to validate signatures from the authorization server. This URL MUST use the "https" scheme. The JWK Set MAY also contain the server's encryption key(s), which are used by clients to encrypt requests to the server. When both signing and encryption keys are made available, a "use" (public key use) parameter value is REQUIRED for all keys in the referenced JWK Set to indicate each key's intended usage.

registration_endpoint

OPTIONAL. URL of the authorization server's OAuth 2.0 Dynamic Client Registration endpoint [RFC7591].

scopes_supported

RECOMMENDED. JSON array containing a list of the OAuth 2.0 [RFC6749] "scope" values that this authorization server supports. Servers MAY choose not to advertise some supported scope values even when this parameter is used.

response_types_supported

REQUIRED. JSON array containing a list of the OAuth 2.0 "response_type" values that this authorization server supports. The array values used are the same as those used with the "response_types" parameter defined by "OAuth 2.0 Dynamic Client Registration Protocol" [RFC7591].

`response_modes_supported`

OPTIONAL. JSON array containing a list of the OAuth 2.0 "response_mode" values that this authorization server supports, as specified in OAuth 2.0 Multiple Response Type Encoding Practices [OAuth.Responses]. If omitted, the default is "["query", "fragment"]". The response mode value "form_post" is also defined in OAuth 2.0 Form Post Response Mode [OAuth.Post].

`grant_types_supported`

OPTIONAL. JSON array containing a list of the OAuth 2.0 grant type values that this authorization server supports. The array values used are the same as those used with the "grant_types" parameter defined by "OAuth 2.0 Dynamic Client Registration Protocol" [RFC7591]. If omitted, the default value is "["authorization_code", "implicit"]".

`token_endpoint_auth_methods_supported`

OPTIONAL. JSON array containing a list of client authentication methods supported by this token endpoint. Client authentication method values are used in the "token_endpoint_auth_method" parameter defined in Section 2 of [RFC7591]. If omitted, the default is "client_secret_basic" -- the HTTP Basic Authentication Scheme specified in Section 2.3.1 of OAuth 2.0 [RFC6749].

`token_endpoint_auth_signing_alg_values_supported`

OPTIONAL. JSON array containing a list of the JWS signing algorithms ("alg" values) supported by the token endpoint for the signature on the JWT [JWT] used to authenticate the client at the token endpoint for the "private_key_jwt" and "client_secret_jwt" authentication methods. This metadata entry MUST be present if either of these authentication methods are specified in the "token_endpoint_auth_methods_supported" entry. No default algorithms are implied if this entry is omitted. Servers SHOULD support "RS256". The value "none" MUST NOT be used.

`service_documentation`

OPTIONAL. URL of a page containing human-readable information that developers might want or need to know when using the authorization server. In particular, if the authorization server does not support Dynamic Client Registration, then information on how to register clients needs to be provided in this documentation.

ui_locales_supported

OPTIONAL. Languages and scripts supported for the user interface, represented as a JSON array of BCP47 [RFC5646] language tag values. If omitted, the set of supported languages and scripts is unspecified.

op_policy_uri

OPTIONAL. URL that the authorization server provides to the person registering the client to read about the authorization server's requirements on how the client can use the data provided by the authorization server. The registration process SHOULD display this URL to the person registering the client if it is given. As described in Section 5, despite the identifier "op_policy_uri", appearing to be OpenID-specific, its usage in this specification is actually referring to a general OAuth 2.0 feature that is not specific to OpenID Connect.

op_tos_uri

OPTIONAL. URL that the authorization server provides to the person registering the client to read about the authorization server's terms of service. The registration process SHOULD display this URL to the person registering the client if it is given. As described in Section 5, despite the identifier "op_tos_uri", appearing to be OpenID-specific, its usage in this specification is actually referring to a general OAuth 2.0 feature that is not specific to OpenID Connect.

revocation_endpoint

OPTIONAL. URL of the authorization server's OAuth 2.0 revocation endpoint [RFC7009].

revocation_endpoint_auth_methods_supported

OPTIONAL. JSON array containing a list of client authentication methods supported by this revocation endpoint. The valid client authentication method values are those registered in the IANA "OAuth Token Endpoint Authentication Methods" registry [IANA.OAuth.Parameters]. If omitted, the default is "client_secret_basic" -- the HTTP Basic Authentication Scheme specified in Section 2.3.1 of OAuth 2.0 [RFC6749].

revocation_endpoint_auth_signing_alg_values_supported

OPTIONAL. JSON array containing a list of the JWS signing algorithms ("alg" values) supported by the revocation endpoint for the signature on the JWT [JWT] used to authenticate the client at the revocation endpoint for the "private_key_jwt" and "client_secret_jwt" authentication methods. This metadata entry MUST be present if either of these authentication methods are specified in the "revocation_endpoint_auth_methods_supported"

entry. No default algorithms are implied if this entry is omitted. The value "none" MUST NOT be used.

`introspection_endpoint`

OPTIONAL. URL of the authorization server's OAuth 2.0 introspection endpoint [RFC7662].

`introspection_endpoint_auth_methods_supported`

OPTIONAL. JSON array containing a list of client authentication methods supported by this introspection endpoint. The valid client authentication method values are those registered in the IANA "OAuth Token Endpoint Authentication Methods" registry [IANA.OAuth.Parameters] or those registered in the IANA "OAuth Access Token Types" registry [IANA.OAuth.Parameters]. (These values are and will remain distinct, due to Section 7.2.) If omitted, the set of supported authentication methods MUST be determined by other means.

`introspection_endpoint_auth_signing_alg_values_supported`

OPTIONAL. JSON array containing a list of the JWS signing algorithms ("alg" values) supported by the introspection endpoint for the signature on the JWT [JWT] used to authenticate the client at the introspection endpoint for the "private_key_jwt" and "client_secret_jwt" authentication methods. This metadata entry MUST be present if either of these authentication methods are specified in the "introspection_endpoint_auth_methods_supported" entry. No default algorithms are implied if this entry is omitted. The value "none" MUST NOT be used.

`code_challenge_methods_supported`

OPTIONAL. JSON array containing a list of PKCE [RFC7636] code challenge methods supported by this authorization server. Code challenge method values are used in the "code_challenge_method" parameter defined in Section 4.3 of [RFC7636]. The valid code challenge method values are those registered in the IANA "PKCE Code Challenge Methods" registry [IANA.OAuth.Parameters]. If omitted, the authorization server does not support PKCE.

Additional authorization server metadata parameters MAY also be used. Some are defined by other specifications, such as OpenID Connect Discovery 1.0 [OpenID.Discovery].

2.1. Signed Authorization Server Metadata

In addition to JSON elements, metadata values MAY also be provided as a "signed_metadata" value, which is a JSON Web Token (JWT) [JWT] that asserts metadata values about the authorization server as a bundle. A set of claims that can be used in signed metadata are defined in

Section 2. The signed metadata MUST be digitally signed or MACed using JSON Web Signature (JWS) [JWS] and MUST contain an "iss" (issuer) claim denoting the party attesting to the claims in the signed metadata. Consumers of the metadata MAY ignore the signed metadata if they do not support this feature. If the consumer of the metadata supports signed metadata, metadata values conveyed in the signed metadata MUST take precedence over the corresponding values conveyed using plain JSON elements.

Signed metadata is included in the authorization server metadata JSON object using this OPTIONAL member:

`signed_metadata`

A JWT containing metadata values about the authorization server as claims. This is a string value consisting of the entire signed JWT. A "signed_metadata" metadata value SHOULD NOT appear as a claim in the JWT.

3. Obtaining Authorization Server Metadata

Authorization servers supporting metadata MUST make a JSON document containing metadata as specified in Section 2 available at a path formed by inserting a well-known URI string into the authorization server's issuer identifier between the host component and the path component, if any. By default, the well-known URI string used is `"/.well-known/oauth-authorization-server"`. This path MUST use the "https" scheme. The syntax and semantics of ".well-known" are defined in RFC 5785 [RFC5785]. The well-known URI suffix used MUST be registered in the IANA "Well-Known URIs" registry [IANA.well-known].

Different applications utilizing OAuth authorization servers in application-specific ways may define and register different well-known URI suffixes used to publish authorization server metadata as used by those applications. For instance, if the Example application uses an OAuth authorization server in an Example-specific way, and there are Example-specific metadata values that it needs to publish, then it might register and use the "example-configuration" URI suffix and publish the metadata document at the path formed by inserting `"/.well-known/example-configuration"` between the host and path components of the authorization server's issuer identifier. Alternatively, many such applications will use the default well-known URI string `"/.well-known/oauth-authorization-server"`, which is the right choice for general-purpose OAuth authorization servers, and not register an application-specific one.

An OAuth 2.0 application using this specification MUST specify what well-known URI suffix it will use for this purpose. The same

authorization server MAY choose to publish its metadata at multiple well-known locations derived from its issuer identifier, for example, publishing metadata at both `"/.well-known/example-configuration"` and `"/.well-known/oauth-authorization-server"`.

Some OAuth applications will choose to use the well-known URI suffix `"openid-configuration"`. As described in Section 5, despite the identifier `"/.well-known/openid-configuration"`, appearing to be OpenID-specific, its usage in this specification is actually referring to a general OAuth 2.0 feature that is not specific to OpenID Connect.

3.1. Authorization Server Metadata Request

An authorization server metadata document MUST be queried using an HTTP `"GET"` request at the previously specified path.

The client would make the following request when the issuer identifier is `"https://example.com"` and the well-known URI suffix is `"oauth-authorization-server"` to obtain the metadata, since the issuer identifier contains no path component:

```
GET /.well-known/oauth-authorization-server HTTP/1.1
Host: example.com
```

If the issuer identifier value contains a path component, any terminating `"/` MUST be removed before inserting `"/.well-known/"` and the well-known URI suffix between the host component and the path component. The client would make the following request when the issuer identifier is `"https://example.com/issuer1"` and the well-known URI suffix is `"oauth-authorization-server"` to obtain the metadata, since the issuer identifier contains a path component:

```
GET /.well-known/oauth-authorization-server/issuer1 HTTP/1.1
Host: example.com
```

Using path components enables supporting multiple issuers per host. This is required in some multi-tenant hosting configurations. This use of `".well-known"` is for supporting multiple issuers per host; unlike its use in RFC 5785 [RFC5785], it does not provide general information about the host.

3.2. Authorization Server Metadata Response

The response is a set of claims about the authorization server's configuration, including all necessary endpoints and public key location information. A successful response MUST use the 200 OK HTTP status code and return a JSON object using the `"application/json"`

content type that contains a set of claims as its members that are a subset of the metadata values defined in Section 2. Other claims MAY also be returned.

Claims that return multiple values are represented as JSON arrays. Claims with zero elements MUST be omitted from the response.

An error response uses the applicable HTTP status code value.

The following is a non-normative example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "issuer":
    "https://server.example.com",
  "authorization_endpoint":
    "https://server.example.com/authorize",
  "token_endpoint":
    "https://server.example.com/token",
  "token_endpoint_auth_methods_supported":
    ["client_secret_basic", "private_key_jwt"],
  "token_endpoint_auth_signing_alg_values_supported":
    ["RS256", "ES256"],
  "userinfo_endpoint":
    "https://server.example.com/userinfo",
  "jwks_uri":
    "https://server.example.com/jwks.json",
  "registration_endpoint":
    "https://server.example.com/register",
  "scopes_supported":
    ["openid", "profile", "email", "address",
     "phone", "offline_access"],
  "response_types_supported":
    ["code", "code token"],
  "service_documentation":
    "http://server.example.com/service_documentation.html",
  "ui_locales_supported":
    ["en-US", "en-GB", "en-CA", "fr-FR", "fr-CA"]
}
```

3.3. Authorization Server Metadata Validation

The "issuer" value returned MUST be identical to the authorization server's issuer identifier value into which the well-known URI string was inserted to create the URL used to retrieve the metadata. If

these values are not identical, the data contained in the response MUST NOT be used.

4. String Operations

Processing some OAuth 2.0 messages requires comparing values in the messages to known values. For example, the member names in the metadata response might be compared to specific member names such as "issuer". Comparing Unicode [UNICODE] strings, however, has significant security implications.

Therefore, comparisons between JSON strings and other Unicode strings MUST be performed as specified below:

1. Remove any JSON applied escaping to produce an array of Unicode code points.
2. Unicode Normalization [USA15] MUST NOT be applied at any point to either the JSON string or to the string it is to be compared against.
3. Comparisons between the two strings MUST be performed as a Unicode code point to code point equality comparison.

Note that this is the same equality comparison procedure described in Section 8.3 of [RFC7159].

5. Compatibility Notes

The identifiers `"/.well-known/openid-configuration"`, `"op_policy_uri"`, and `"op_tos_uri"` contain strings referring to the OpenID Connect [OpenID.Core] family of specifications that were originally defined by "OpenID Connect Discovery 1.0" [OpenID.Discovery]. Despite the reuse of these identifiers that appear to be OpenID-specific, their usage in this specification is actually referring to general OAuth 2.0 features that are not specific to OpenID Connect.

The algorithm for transforming the issuer identifier to an authorization server metadata location defined in Section 3 is equivalent to the corresponding transformation defined in Section 4 of "OpenID Connect Discovery 1.0" [OpenID.Discovery], provided that the issuer identifier contains no path component. However, they are different when there is a path component, because OpenID Connect Discovery 1.0 specifies that the well-known URI string is appended to the issuer identifier (e.g., `"https://example.com/issuer1/.well-known/openid-configuration"`), whereas this specification specifies that the well-known URI string is inserted before the path component

of the issuer identifier (e.g., "https://example.com/.well-known/openid-configuration/issuer1").

Going forward, OAuth authorization server metadata locations should use the transformation defined in this specification. However, when deployed in legacy environments in which the OpenID Connect Discovery 1.0 transformation is already used, it may be necessary during a transition period to publish metadata for issuer identifiers containing a path component at both locations. During this transition period, applications should first apply the transformation defined in this specification and attempt to retrieve the authorization server metadata from the resulting location; only if the retrieval from that location fails should they fall back to attempting to retrieve it from the alternate location obtained using the transformation defined by OpenID Connect Discovery 1.0. This backwards-compatibility behavior should only be necessary when the well-known URI suffix employed by the application is "openid-configuration".

6. Security Considerations

6.1. TLS Requirements

Implementations **MUST** support TLS. Which version(s) ought to be implemented will vary over time and depend on the widespread deployment and known security vulnerabilities at the time of implementation. The authorization server **MUST** support TLS version 1.2 [RFC5246] and **MAY** support additional transport-layer security mechanisms meeting its security requirements. When using TLS, the client **MUST** perform a TLS/SSL server certificate check, per RFC 6125 [RFC6125]. Implementation security considerations can be found in Recommendations for Secure Use of TLS and DTLS [BCP195].

To protect against information disclosure and tampering, confidentiality protection **MUST** be applied using TLS with a ciphersuite that provides confidentiality and integrity protection.

6.2. Impersonation Attacks

TLS certificate checking **MUST** be performed by the client, as described in Section 6.1, when making an authorization server metadata request. Checking that the server certificate is valid for the issuer identifier URL prevents man-in-middle and DNS-based attacks. These attacks could cause a client to be tricked into using an attacker's keys and endpoints, which would enable impersonation of the legitimate authorization server. If an attacker can accomplish this, they can access the resources that the affected client has access to using the authorization server that they are impersonating.

An attacker may also attempt to impersonate an authorization server by publishing a metadata document that contains an "issuer" claim using the issuer identifier URL of the authorization server being impersonated, but with its own endpoints and signing keys. This would enable it to impersonate that authorization server, if accepted by the client. To prevent this, the client MUST ensure that the issuer identifier URL it is using as the prefix for the metadata request exactly matches the value of the "issuer" metadata value in the authorization server metadata document received by the client.

6.3. Publishing Metadata in a Standard Format

Publishing information about the authorization server in a standard format makes it easier for both legitimate clients and attackers to use the authorization server. Whether an authorization server publishes its metadata in an ad-hoc manner or in the standard format defined by this specification, the same defenses against attacks that might be mounted that use this information should be applied.

6.4. Protected Resources

Secure determination of appropriate protected resources to use with an authorization server for all use cases is out of scope of this specification. This specification assumes that the client has a means of determining appropriate protected resources to use with an authorization server and that the client is using the correct metadata for each authorization server. Implementers need to be aware that if an inappropriate protected resource is used by the client, that an attacker may be able to act as a man-in-the-middle proxy to a valid protected resource without it being detected by the authorization server or the client.

The ways to determine the appropriate protected resources to use with an authorization server are in general, application-dependent. For instance, some authorization servers are used with a fixed protected resource or set of protected resources, the locations of which may be well known, or which could be published as metadata values by the authorization server. In other cases, the set of resources that can be used with an authorization server can be dynamically changed by administrative actions. Many other means of determining appropriate associations between authorization servers and protected resources are also possible.

7. IANA Considerations

The following registration procedure is used for the registry established by this specification.

Values are registered on a Specification Required [RFC8126] basis after a two-week review period on the `oauth-ext-review@ietf.org` mailing list, on the advice of one or more Designated Experts. However, to allow for the allocation of values prior to publication, the Designated Experts may approve registration once they are satisfied that such a specification will be published.

Registration requests sent to the mailing list for review should use an appropriate subject (e.g., "Request to register OAuth Authorization Server Metadata: example").

Within the review period, the Designated Experts will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful. Registration requests that are undetermined for a period longer than 21 days can be brought to the IESG's attention (using the `iesg@ietf.org` mailing list) for resolution.

Criteria that should be applied by the Designated Experts includes determining whether the proposed registration duplicates existing functionality, determining whether it is likely to be of general applicability or whether it is useful only for a single application, and whether the registration makes sense.

IANA must only accept registry updates from the Designated Experts and should direct all requests for registration to the review mailing list.

It is suggested that multiple Designated Experts be appointed who are able to represent the perspectives of different applications using this specification, in order to enable broadly-informed review of registration decisions. In cases where a registration decision could be perceived as creating a conflict of interest for a particular Expert, that Expert should defer to the judgment of the other Experts.

7.1. OAuth Authorization Server Metadata Registry

This specification establishes the IANA "OAuth Authorization Server Metadata" registry for OAuth 2.0 authorization server metadata names. The registry records the authorization server metadata member and a reference to the specification that defines it.

The Designated Experts must either:

(a) require that metadata names and values being registered use only printable ASCII characters excluding double quote (`'"`) and backslash

('\'') (the Unicode characters with code points U+0021, U+0023 through U+005B, and U+005D through U+007E), or

(b) if new metadata members or values are defined that use other code points, require that their definitions specify the exact Unicode code point sequences used to represent them. Furthermore, proposed registrations that use Unicode code points that can only be represented in JSON strings as escaped characters must not be accepted.

7.1.1. Registration Template

Metadata Name:

The name requested (e.g., "issuer"). This name is case-sensitive. Names may not match other registered names in a case-insensitive manner (one that would cause a match if the Unicode toLowerCase() operation were applied to both strings) unless the Designated Experts state that there is a compelling reason to allow an exception.

Metadata Description:

Brief description of the metadata (e.g., "Issuer identifier URL").

Change Controller:

For Standards Track RFCs, list the "IESG". For others, give the name of the responsible party. Other details (e.g., postal address, email address, home page URI) may also be included.

Specification Document(s):

Reference to the document or documents that specify the parameter, preferably including URIs that can be used to retrieve copies of the documents. An indication of the relevant sections may also be included but is not required.

7.1.2. Initial Registry Contents

- o Metadata Name: "issuer"
- o Metadata Description: Authorization server's issuer identifier URL
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "authorization_endpoint"
- o Metadata Description: URL of the authorization server's authorization endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "token_endpoint"

- o Metadata Description: URL of the authorization server's token endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "jwks_uri"
- o Metadata Description: URL of the authorization server's JWK Set document
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "registration_endpoint"
- o Metadata Description: URL of the authorization server's OAuth 2.0 Dynamic Client Registration Endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "scopes_supported"
- o Metadata Description: JSON array containing a list of the OAuth 2.0 "scope" values that this authorization server supports
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "response_types_supported"
- o Metadata Description: JSON array containing a list of the OAuth 2.0 "response_type" values that this authorization server supports
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "response_modes_supported"
- o Metadata Description: JSON array containing a list of the OAuth 2.0 "response_mode" values that this authorization server supports
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "grant_types_supported"
- o Metadata Description: JSON array containing a list of the OAuth 2.0 grant type values that this authorization server supports
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "token_endpoint_auth_methods_supported"
- o Metadata Description: JSON array containing a list of client authentication methods supported by this token endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "token_endpoint_auth_signing_alg_values_supported"

- o Metadata Description: JSON array containing a list of the JWS signing algorithms supported by the token endpoint for the signature on the JWT used to authenticate the client at the token endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "service_documentation"
- o Metadata Description: URL of a page containing human-readable information that developers might want or need to know when using the authorization server
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "ui_locales_supported"
- o Metadata Description: Languages and scripts supported for the user interface, represented as a JSON array of BCP47 language tag values
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "op_policy_uri"
- o Metadata Description: URL that the authorization server provides to the person registering the client to read about the authorization server's requirements on how the client can use the data provided by the authorization server
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "op_tos_uri"
- o Metadata Description: URL that the authorization server provides to the person registering the client to read about the authorization server's terms of service
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "revocation_endpoint"
- o Metadata Description: URL of the authorization server's OAuth 2.0 revocation endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "revocation_endpoint_auth_methods_supported"
- o Metadata Description: JSON array containing a list of client authentication methods supported by this revocation endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name:
"revocation_endpoint_auth_signing_alg_values_supported"
- o Metadata Description: JSON array containing a list of the JWS signing algorithms supported by the revocation endpoint for the signature on the JWT used to authenticate the client at the revocation endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "introspection_endpoint"
- o Metadata Description: URL of the authorization server's OAuth 2.0 introspection endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "introspection_endpoint_auth_methods_supported"
- o Metadata Description: JSON array containing a list of client authentication methods supported by this introspection endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name:
"introspection_endpoint_auth_signing_alg_values_supported"
- o Metadata Description: JSON array containing a list of the JWS signing algorithms supported by the introspection endpoint for the signature on the JWT used to authenticate the client at the introspection endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "code_challenge_methods_supported"
- o Metadata Description: PKCE code challenge methods supported by this authorization server
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

7.2. Updated Registration Instructions

This specification adds to the instructions for the Designated Experts of the following IANA registries, both of which are in the "OAuth Parameters" registry [IANA.OAuth.Parameters]:

- o OAuth Access Token Types
- o OAuth Token Endpoint Authentication Methods

IANA has added a link to this specification in the Reference sections of these registries. [[RFC Editor: The above sentence is written in the past tense as it would appear in the final specification, even

though these links won't actually be created until after the IESG has requested publication of the specification. Please delete this note after the links are in place.]]

For these registries, the designated experts must reject registration requests in one registry for values already occurring in the other registry. This is necessary because the "introspection_endpoint_auth_methods_supported" parameter allows for the use of values from either registry. That way, because the values in the two registries will continue to be mutually exclusive, no ambiguities will arise.

7.3. Well-Known URI Registry

This specification registers the well-known URI defined in Section 3 in the IANA "Well-Known URIs" registry [IANA.well-known] established by RFC 5785 [RFC5785].

7.3.1. Registry Contents

- o URI suffix: "oauth-authorization-server"
- o Change controller: IESG
- o Specification document: Section 3 of [[this specification]]
- o Related information: (none)

8. References

8.1. Normative References

- [BCP195] Sheffer, Y., Holz, R., and P. Saint-Andre,
"Recommendations for Secure Use of Transport Layer
Security (TLS) and Datagram Transport Layer Security
(DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May
2015, <<http://www.rfc-editor.org/info/bcp195>>.
- [IANA.OAuth.Parameters]
IANA, "OAuth Parameters",
<<http://www.iana.org/assignments/oauth-parameters>>.
- [JWE] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)",
RFC 7516, DOI 10.17487/RFC7516, May 2015,
<<http://tools.ietf.org/html/rfc7516>>.
- [JWK] Jones, M., "JSON Web Key (JWK)", RFC 7517,
DOI 10.17487/RFC7517, May 2015,
<<http://tools.ietf.org/html/rfc7517>>.

- [JWS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <<http://tools.ietf.org/html/rfc7515>>.
- [JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <<http://tools.ietf.org/html/rfc7519>>.
- [OAuth.Post] Jones, M. and B. Campbell, "OAuth 2.0 Form Post Response Mode", April 2015, <http://openid.net/specs/oauth-v2-form-post-response-mode-1_0.html>.
- [OAuth.Responses] de Medeiros, B., Ed., Scurtescu, M., Tarjan, P., and M. Jones, "OAuth 2.0 Multiple Response Type Encoding Practices", February 2014, <http://openid.net/specs/oauth-v2-multiple-response-types-1_0.html>.
- [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>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/info/rfc5246>>.
- [RFC5646] Phillips, A., Ed. and M. Davis, Ed., "Tags for Identifying Languages", BCP 47, RFC 5646, DOI 10.17487/RFC5646, September 2009, <<https://www.rfc-editor.org/info/rfc5646>>.
- [RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, DOI 10.17487/RFC5785, April 2010, <<https://www.rfc-editor.org/info/rfc5785>>.
- [RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <<https://www.rfc-editor.org/info/rfc6125>>.
- [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>>.

- [RFC7009] Lodderstedt, T., Ed., Dronia, S., and M. Scurtescu, "OAuth 2.0 Token Revocation", RFC 7009, DOI 10.17487/RFC7009, August 2013, <<https://www.rfc-editor.org/info/rfc7009>>.
- [RFC7033] Jones, P., Salgueiro, G., Jones, M., and J. Smarr, "WebFinger", RFC 7033, DOI 10.17487/RFC7033, September 2013, <<https://www.rfc-editor.org/info/rfc7033>>.
- [RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <<https://www.rfc-editor.org/info/rfc7159>>.
- [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>>.
- [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>>.
- [RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<https://www.rfc-editor.org/info/rfc7662>>.
- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.
- [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>>.
- [UNICODE] The Unicode Consortium, "The Unicode Standard", <<http://www.unicode.org/versions/latest/>>.
- [USA15] Davis, M. and K. Whistler, "Unicode Normalization Forms", Unicode Standard Annex 15, June 2015, <<http://www.unicode.org/reports/tr15/>>.

8.2. Informative References

- [I-D.ietf-oauth-mix-up-mitigation]
Jones, M., Bradley, J., and N. Sakimura, "OAuth 2.0 Mix-Up Mitigation", draft-ietf-oauth-mix-up-mitigation-01 (work in progress), July 2016.

[IANA.well-known]

IANA, "Well-Known URIs",
<<http://www.iana.org/assignments/well-known-uris>>.

[OpenID.Core]

Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and
C. Mortimore, "OpenID Connect Core 1.0", November 2014,
<http://openid.net/specs/openid-connect-core-1_0.html>.

[OpenID.Discovery]

Sakimura, N., Bradley, J., Jones, M., and E. Jay, "OpenID
Connect Discovery 1.0", November 2014,
<[http://openid.net/specs/
openid-connect-discovery-1_0.html](http://openid.net/specs/openid-connect-discovery-1_0.html)>.

[OpenID.Registration]

Sakimura, N., Bradley, J., and M. Jones, "OpenID Connect
Dynamic Client Registration 1.0", November 2014,
<[http://openid.net/specs/
openid-connect-registration-1_0.html](http://openid.net/specs/openid-connect-registration-1_0.html)>.

Appendix A. Acknowledgements

This specification is based on the OpenID Connect Discovery 1.0 specification, which was produced by the OpenID Connect working group of the OpenID Foundation. This specification standardizes the de facto usage of the metadata format defined by OpenID Connect Discovery to publish OAuth authorization server metadata.

The authors would like to thank the following people for their reviews of this specification: Shwetha Bhandari, Ben Campbell, Brian Campbell, Brian Carpenter, William Denniss, Vladimir Dzhuvinov, Donald Eastlake, Samuel Erdtman, George Fletcher, Dick Hardt, Phil Hunt, Alexey Melnikov, Tony Nadalin, Mark Nottingham, Eric Rescorla, Justin Richer, Adam Roach, Hannes Tschofenig, and Hans Zandbelt.

Appendix B. Document History

[[to be removed by the RFC Editor before publication as an RFC]]

-10

- o Clarified the meaning of "case-insensitive", as suggested by Alexey Melnikov.

-09

- o Revised the transformation between the issuer identifier and the authorization server metadata location to conform to BCP 190, as suggested by Adam Roach.
- o Defined the characters allowed in registered metadata names and values, as suggested by Alexey Melnikov.
- o Changed to using the RFC 8174 boilerplate instead of the RFC 2119 boilerplate, as suggested by Ben Campbell.
- o Acknowledged additional reviewers.

-08

- o Changed the "authorization_endpoint" to be REQUIRED only when grant types are supported that use the authorization endpoint.
- o Added the statement, to provide historical context, that this specification standardizes the de facto usage of the metadata format defined by OpenID Connect Discovery to publish OAuth authorization server metadata.
- o Applied clarifications suggested by Mark Nottingham about when application-specific well-known suffixes are and are not appropriate.
- o Acknowledged additional reviewers.

-07

- o Applied clarifications suggested by EKR.

-06

- o Incorporated resolutions to working group last call comments.

-05

- o Removed the "protected_resources" element and the reference to draft-jones-oauth-resource-metadata.

-04

- o Added the ability to list protected resources with the "protected_resources" element.
- o Added ability to provide signed metadata with the "signed_metadata" element.

- o Removed "Discovery" from the name, since this is now just about authorization server metadata.

-03

- o Changed term "issuer URL" to "issuer identifier" for terminology consistency, paralleling the same terminology consistency change in the mix-up mitigation spec.

-02

- o Changed the title to OAuth 2.0 Authorization Server Discovery Metadata.
- o Made "jwks_uri" and "registration_endpoint" OPTIONAL.
- o Defined the well-known URI string "/.well-known/oauth-authorization-server".
- o Added security considerations about publishing authorization server discovery metadata in a standard format.
- o Added security considerations about protected resources.
- o Added more information to the "grant_types_supported" and "response_types_supported" definitions.
- o Referenced the working group Mix-Up Mitigation draft.
- o Changed some example metadata values.
- o Acknowledged individuals for their contributions to the specification.

-01

- o Removed WebFinger discovery.
- o Clarified the relationship between the issuer identifier URL and the well-known URI path relative to it at which the discovery metadata document is located.

-00

- o Created the initial working group version based on draft-jones-oauth-discovery-01, with no normative changes.

Authors' Addresses

Michael B. Jones
Microsoft

Email: mbj@microsoft.com
URI: <http://self-issued.info/>

Nat Sakimura
Nomura Research Institute, Ltd.

Email: n-sakimura@nri.co.jp
URI: <http://nat.sakimura.org/>

John Bradley
Ping Identity

Email: ve7jtb@ve7jtb.com
URI: <http://www.thread-safe.com/>

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: October 10, 2021

N. Sakimura
NAT.Consulting
J. Bradley
Yubico
M. Jones
Microsoft
April 8, 2021

The OAuth 2.0 Authorization Framework: JWT Secured Authorization Request
(JAR)
draft-ietf-oauth-jwsreq-34

Abstract

The authorization request in OAuth 2.0 described in RFC 6749 utilizes query parameter serialization, which means that Authorization Request parameters are encoded in the URI of the request and sent through user agents such as web browsers. While it is easy to implement, it means that (a) the communication through the user agents is not integrity protected and thus the parameters can be tainted, (b) the source of the communication is not authenticated, and (c) the communication through the user agents can be monitored. Because of these weaknesses, several attacks to the protocol have now been put forward.

This document introduces the ability to send request parameters in a JSON Web Token (JWT) instead, which allows the request to be signed with JSON Web Signature (JWS) and encrypted with JSON Web Encryption (JWE) so that the integrity, source authentication and confidentiality property of the Authorization Request is attained. The request can be sent by value or by reference.

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 <https://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 October 10, 2021.

Copyright Notice

Copyright (c) 2021 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
1.1. Requirements Language	5
2. Terminology	6
2.1. Request Object	6
2.2. Request Object URI	6
3. Symbols and abbreviated terms	6
4. Request Object	7
5. Authorization Request	9
5.1. Passing a Request Object by Value	10
5.2. Passing a Request Object by Reference	10
5.2.1. URI Referencing the Request Object	11
5.2.2. Request using the "request_uri" Request Parameter	12
5.2.3. Authorization Server Fetches Request Object	12
6. Validating JWT-Based Requests	13
6.1. JWE Encrypted Request Object	13
6.2. JWS Signed Request Object	13
6.3. Request Parameter Assembly and Validation	14
7. Authorization Server Response	14
8. TLS Requirements	14
9. IANA Considerations	15
9.1. OAuth Parameters Registration	15
9.2. OAuth Authorization Server Metadata Registry	16
9.3. OAuth Dynamic Client Registration Metadata Registry	17
9.4. Media Type Registration	17
9.4.1. Registry Contents	17
10. Security Considerations	18
10.1. Choice of Algorithms	18
10.2. Request Source Authentication	18
10.3. Explicit Endpoints	19

10.4.	Risks Associated with request_uri	19
10.4.1.	DDoS Attack on the Authorization Server	20
10.4.2.	Request URI Rewrite	20
10.5.	Downgrade Attack	20
10.6.	TLS Security Considerations	21
10.7.	Parameter Mismatches	21
10.8.	Cross-JWT Confusion	21
11.	Privacy Considerations	22
11.1.	Collection limitation	22
11.2.	Disclosure Limitation	23
11.2.1.	Request Disclosure	23
11.2.2.	Tracking using Request Object URI	23
12.	Acknowledgements	24
13.	Revision History	24
14.	References	32
14.1.	Normative References	32
14.2.	Informative References	34
	Authors' Addresses	35

1. Introduction

The Authorization Request in OAuth 2.0 [RFC6749] utilizes query parameter serialization and is typically sent through user agents such as web browsers.

For example, the parameters "response_type", "client_id", "state", and "redirect_uri" are encoded in the URI of the request:

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=xyz
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb HTTP/1.1
Host: server.example.com
```

While it is easy to implement, the encoding in the URI does not allow application layer security to be used to provide confidentiality and integrity protection. While TLS is used to offer communication security between the Client and the user-agent as well as the user-agent and the Authorization Server, TLS sessions are terminated in the user-agent. In addition, TLS sessions may be terminated prematurely at some middlebox (such as a load balancer).

As the result, the Authorization Request of [RFC6749] has shortcomings in that:

- (a) the communication through the user agents is not integrity protected and thus the parameters can be tainted (integrity protection failure)

- (b) the source of the communication is not authenticated (source authentication failure)
- (c) the communication through the user agents can be monitored (containment / confidentiality failure).

Due to these inherent weaknesses, several attacks against the protocol, such as Redirection URI rewriting, have been identified.

The use of application layer security mitigates these issues.

The use of application layer security allows requests to be prepared by a trusted third party so that a client application cannot request more permissions than previously agreed.

Furthermore, passing the request by reference allows the reduction of over-the-wire overhead.

The JWT [RFC7519] encoding has been chosen because of

- (1) its close relationship with JSON, which is used as OAuth's response format
- (2) its developer friendliness due to its textual nature
- (3) its relative compactness compared to XML
- (4) its development status as a Proposed Standard, along with the associated signing and encryption methods [RFC7515] [RFC7516]
- (5) the relative ease of JWS and JWE compared to XML Signature and Encryption.

The parameters "request" and "request_uri" are introduced as additional authorization request parameters for the OAuth 2.0 [RFC6749] flows. The "request" parameter is a JSON Web Token (JWT) [RFC7519] whose JWT Claims Set holds the JSON encoded OAuth 2.0 authorization request parameters. Note that, in contrast to RFC 7519, the elements of the Claims Set are encoded OAuth Request Parameters [IANA.OAuth.Parameters], supplemented with only a few of the IANA-managed JSON Web Token Claims [IANA.JWT.Claims] - in particular "iss" and "aud". The JWT in the "request" parameter is integrity protected and source authenticated using JWS.

The JWT [RFC7519] can be passed to the authorization endpoint by reference, in which case the parameter "request_uri" is used instead of the "request".

Using JWT [RFC7519] as the request encoding instead of query parameters has several advantages:

- (a) (integrity protection) The request can be signed so that the integrity of the request can be checked.
- (b) (source authentication) The request can be signed so that the signer can be authenticated.
- (c) (confidentiality protection) The request can be encrypted so that end-to-end confidentiality can be provided even if the TLS connection is terminated at one point or another (including at and before user-agents).
- (d) (collection minimization) The request can be signed by a trusted third party attesting that the authorization request is compliant with a certain policy. For example, a request can be pre-examined by a trusted third party that all the personal data requested is strictly necessary to perform the process that the end-user asked for, and signed by that trusted third party. The authorization server then examines the signature and shows the conformance status to the end-user, who would have some assurance as to the legitimacy of the request when authorizing it. In some cases, it may even be desirable to skip the authorization dialogue under such circumstances.

There are a few cases that request by reference is useful such as:

1. When it is desirable to reduce the size of transmitted request. The use of application layer security increases the size of the request, particularly when public key cryptography is used.
2. When the client does not want to do the application level cryptography. The Authorization Server may provide an endpoint to accept the Authorization Request through direct communication with the Client so that the Client is authenticated and the channel is TLS protected.

This capability is in use by OpenID Connect [OpenID.Core].

1.1. Requirements Language

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.

2. Terminology

For the purposes of this specification, the following terms and definitions in addition to what is defined in OAuth 2.0 Framework [RFC6749], JSON Web Signature [RFC7515], and JSON Web Encryption [RFC7519] apply.

2.1. Request Object

JSON Web Token (JWT) [RFC7519] whose JWT Claims Set holds the JSON encoded OAuth 2.0 authorization request parameters.

2.2. Request Object URI

Absolute URI that references the set of parameters comprising an OAuth 2.0 authorization request. The contents of the resource referenced by the URI are a Request Object (Section 2.1), unless the URI was provided to the client by the same Authorization Server, in which case the content is an implementation detail at the discretion of the Authorization Server. The former is to ensure interoperability in cases where the provider of the request_uri is a separate entity from the consumer, such as when a client provides a URI referencing a Request Object stored on the client's backend service and made accessible via HTTPS. In the latter case where the Authorization Server is both provider and consumer of the URI, such as when it offers an endpoint that provides a URI in exchange for a Request Object, this interoperability concern does not apply.

3. Symbols and abbreviated terms

The following abbreviations are common to this specification.

JSON JavaScript Object Notation

JWT JSON Web Token

JWS JSON Web Signature

JWE JSON Web Encryption

URI Uniform Resource Identifier

URL Uniform Resource Locator

4. Request Object

A Request Object (Section 2.1) is used to provide authorization request parameters for an OAuth 2.0 authorization request. It MUST contain all the parameters (including extension parameters) used to process the OAuth 2.0 [RFC6749] authorization request except the "request" and "request_uri" parameters that are defined in this document. The parameters are represented as the JWT claims of the object. Parameter names and string values MUST be included as JSON strings. Since Request Objects are handled across domains and potentially outside of a closed ecosystem, per section 8.1 of [RFC8259], these JSON strings MUST be encoded using UTF-8 [RFC3629]. Numerical values MUST be included as JSON numbers. It MAY include any extension parameters. This JSON [RFC8259] object constitutes the JWT Claims Set defined in JWT [RFC7519]. The JWT Claims Set is then signed or signed and encrypted.

To sign, JSON Web Signature (JWS) [RFC7515] is used. The result is a JWS signed JWT [RFC7519]. If signed, the Authorization Request Object SHOULD contain the Claims "iss" (issuer) and "aud" (audience) as members, with their semantics being the same as defined in the JWT [RFC7519] specification. The value of "aud" should be the value of the Authorization Server (AS) "issuer" as defined in RFC8414 [RFC8414].

To encrypt, JWE [RFC7516] is used. When both signature and encryption are being applied, the JWT MUST be signed then encrypted as described in Section 11.2 of [RFC7519]. The result is a Nested JWT, as defined in [RFC7519].

The client determines the algorithms used to sign and encrypt Request Objects. The algorithms chosen need to be supported by both the client and the authorization server. The client can inform the authorization server of the algorithms that it supports in its dynamic client registration metadata [RFC7591], specifically, the metadata values "request_object_signing_alg", "request_object_encryption_alg", and "request_object_encryption_enc". Likewise, the authorization server can inform the client of the algorithms that it supports in its authorization server metadata [RFC8414], specifically, the metadata values "request_object_signing_alg_values_supported", "request_object_encryption_alg_values_supported", and "request_object_encryption_enc_values_supported".

The Request Object MAY be sent by value as described in Section 5.1 or by reference as described in Section 5.2. "request" and "request_uri" parameters MUST NOT be included in Request Objects.

A Request Object (Section 2.1) has the media type [RFC2046] "application/oauth-authz-req+jwt". Note that some existing deployments may alternatively be using the type "application/jwt".

The following is an example of the Claims in a Request Object before base64url [RFC7515] encoding and signing. Note that it includes the extension parameters "nonce" and "max_age".

```
{
  "iss": "s6BhdRkqt3",
  "aud": "https://server.example.com",
  "response_type": "code id_token",
  "client_id": "s6BhdRkqt3",
  "redirect_uri": "https://client.example.org/cb",
  "scope": "openid",
  "state": "af0ifjsldkj",
  "nonce": "n-0S6_WzA2Mj",
  "max_age": 86400
}
```

Signing it with the "RS256" algorithm [RFC7518] results in this Request Object value (with line wraps within values for display purposes only):

```
eyJhbGciOiJSUzI1NiIsImtpZCI6ImN5YmRjIn0.ewogICAgImIzcyI6ICJzNkJoZmZJrcXQzIiwKICAgICJhdWQiOiAiaHR0cHM6Ly9zZXJ2ZXIuZXhhbXBsZS5jb20iLAogICAgInJlc3BvbnNlX3R5cGUiOiAiY29kZSBpZF90b2t1biIsCiAgICAiY2xpZW50X2lkIjogInM2QmhhkUmtxdDMiLAogICAgInJlZGlyZW50X3VyaSI6ICJodHRwczovL2NsYWVudC5leGFtcGxlLm9yZy9jYiIsCiAgICAic2NvcGUiOiAib3BlbmklIiwKICAgICJzdGF0ZSI6ICJhZjBpZmpzbGRraiiIsCiAgICAibm9uY2UiOiAibi0wUzZfV3pBMklqIiwKICAgICJtYXh0YXZlIiwKICAgICJodG90MDAKfQ.Nsxa_18VUElVaPjqW_ToIlyrEJ67BgKb5xsuZRVqzGkfKrOIX7BCx0biSxYGmjK9KJPctH1OC0iQJwXu5YVY-vnW0_PLJb1C2HG-ztVzcnKZC2gE4i0vgQcpkUOCpW3SEYXnyWnKzuKzqSb1wAZALo5f89B_p6QA6j6JwBSRvdVsDPdulW8lKxGTbH82czCaQ50rLAg3EYLYaCb4ik4I1zGXE4fvim9FIMs80CMmzWIB5S-uJFfzwFjoyuPEV4hJnoVUmXR_W9typPf846lGwA8h9G9oNTIuX8Ft2jfpnZdFmLg3_wr3Wa5q3a-lfbgF3S9H_8nN3jli7tLR_5Nz-g
```

The following RSA public key, represented in JWK format, can be used to validate the Request Object signature in this and subsequent Request Object examples (with line wraps within values for display purposes only):

```
{
  "kty": "RSA",
  "kid": "k2bdc",
  "n": "x5RbkAZkmpRxia65qRQ1wwSMSxQUnS7gcpVTV_cdHmfmg21td2yabEO9XadD8
    pJNZubINPpmgHh3J1aD9WRwS05ucmFq3CfFsluLt13_7oX5yDRSKX7poXmT_5
    ko8k4NJZPMAO8fPToDTH7kHYbONSE2FYa5GZ60CUsFhSonI-dcMDJ0Ary91xI
    w5k2z4TAdARVWcs7sD07VhlMMshrwsPHBQgTatlkxyIHxbYdtak8fqvNAwr7O
    lVEvM_Ipf5OfmdB8Sd-wjzaBsyP4VhJKoi_qdgSzpC694XZeYPq45Sw-q51iF
    Ulc0lTCI7z6jltUtnR6ySn6XDGFnzH5Fe5ypw",
  "e": "AQAB"
}
```

5. Authorization Request

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

request

REQUIRED unless "request_uri" is specified. The Request Object (Section 2.1) that holds authorization request parameters stated in section 4 of OAuth 2.0 [RFC6749]. If this parameter is present in the authorization request, "request_uri" MUST NOT be present.

request_uri

REQUIRED unless "request" is specified. The absolute URI as defined by RFC3986 [RFC3986] that is the Request Object URI (Section 2.2) referencing the authorization request parameters stated in section 4 of OAuth 2.0 [RFC6749]. If this parameter is present in the authorization request, "request" MUST NOT be present.

client_id

REQUIRED. OAuth 2.0 [RFC6749] "client_id". The value MUST match the "request" or "request_uri" Request Object's (Section 2.1) "client_id".

The client directs the resource owner to the constructed URI using an HTTP redirection response, or by other means available to it via the user-agent.

For example, the client directs the end user's user-agent to make the following HTTPS request:

```
GET /authz?client_id=s6BhdRkqt3&request=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXZW40In0= HTTP/1.1
Host: server.example.com
```

The value for the request parameter is abbreviated for brevity.

The authorization request object MUST be one of the following:

- (a) JWS signed
- (b) JWS signed and JWE encrypted

The client MAY send the parameters included in the request object duplicated in the query parameters as well for the backward compatibility etc. However, the authorization server supporting this specification MUST only use the parameters included in the request object.

5.1. Passing a Request Object by Value

The Client sends the Authorization Request as a Request Object to the Authorization Endpoint as the "request" parameter value.

The following is an example of an Authorization Request using the "request" parameter (with line wraps within values for display purposes only):

```
https://server.example.com/authorize?client_id=s6BhdRkqt3&
request=eyJhbGciOiJSUzI1NiIsImtpZCI6Im9yYmRjIn0.ewogICAgImIzcyI6
ICJzNkJoZjZJrcXQzIiwKIICAgICJhdWQiOiAiAiaHR0cHM6Ly9zZXJ2ZXIuZXhhbXBs
ZS5jb20iLAogICAgInJlc3BvbmlX3R5cGUiOiAiY29kZSBpZF90b2t1biIsCiAg
ICAiY2xpZW50X2lkIjogInM2QmhhkUmtxdDMLAogICAgInJlZGlyZWNOX3VyaSI6
ICJodHRwczovL2NsaWVudC5leGFtcGxlLm9yZy9jYiIsCiAgICAIc2NvcGUiOiAi
b3BlbmklIiwKIICAgICJzdGF0ZSI6ICJhZjBpZmpzbGRraiIsCiAgICAIbm9uY2Ui
OiAibi0wUzZfV3pBMklqIiwKIICAgICJtYXhfYWdlIjogODY0MDAKfQ.Nsxa_18VU
ElVaPjqW_ToIlyrEJ67BgKb5xsuZRVqzGkfKrOIX7BCx0biSxYGmjK9KJPctH1OC
0iQJwXu5YVY-vnW0_PLJb1C2HG-ztVzcnKZC2gE4i0vgQcpkUOCpW3SEYXnyWnKz
uKzqSblwAZALo5f89B_p6QA6j6JwBSRvdVsDPdulW8lKxGTbH82czCaQ50rLAg3E
YLyaCb4ik4I1zGXE4fvim9FIMs8OCmzwIB5S-uJfFzwFjoyuPEV4hJnoVUmXR_W
9t9pPf846lGwA8h9G9oNTIuX8Ft2jfpnZdFmLg3_wr3Wa5q3a-lfbgF3S9H_8nN3
jli7tLR_5Nz-q
```

5.2. Passing a Request Object by Reference

The "request_uri" Authorization Request parameter enables OAuth authorization requests to be passed by reference, rather than by value. This parameter is used identically to the "request" parameter, other than that the Request Object value is retrieved from

the resource identified by the specified URI rather than passed by value.

The entire Request URI SHOULD NOT exceed 512 ASCII characters. There are two reasons for this restriction:

1. Many phones in the market as of this writing still do not accept large payloads. The restriction is typically either 512 or 1024 ASCII characters.
2. On a slow connection such as 2G mobile connection, a large URL would cause the slow response and therefore the use of such is not advisable from the user experience point of view.

The contents of the resource referenced by the "request_uri" MUST be a Request Object and MUST be reachable by the Authorization Server unless the URI was provided to the client by the Authorization Server. In the first case, the "request_uri" MUST be an "https" URI, as specified in Section 2.7.2 of RFC7230 [RFC7230]. In the second case, it MUST be a URN, as specified in RFC8141 [RFC8141].

The following is an example of the contents of a Request Object resource that can be referenced by a "request_uri" (with line wraps within values for display purposes only):

```
eyJhbGciOiJSUzI1NiIsImtpZCI6Im9yZy9jYiIsCiAgICAic2NvcGUiOiAib3BlbmklIiwKICAgICJzdGF0ZSI6ICJhZjBpZmpzbGRraiiIsCiAgICAibm9uY2UiOiAibi0wUzZfV3pBMklqIiwKICAgICJtYXhfYWdlIjogODY0MDAKfQ.Nsxa_18VUElVaPjqW_ToIlyrEJ67BgKb5xsuZRVqzGkfKrOIX7BCx0biSxYGmjK9KJPctH1OC0iQJwXu5YVY-vnW0_PLJb1C2HG-ztVzcnKZC2gE4i0vgQcpkUOCpW3SEYXnyWnKzuKzqSblwAZALo5f89B_p6QA6j6JwBSRvdVsDPdulW8lKxGTbH82czCaQ50rLag3EYLYaCb4ik4I1zGXE4fvim9FIMs80CMmzwIB5S-ujFfzwFjoyuPEV4hJnoVUmXR_W9typPf846lGwA8h9G9oNTIuX8Ft2jfpnZdFmLg3_wr3Wa5q3a-lfbgF3S9H_8nN3j1i7tLR_5Nz-g
```

5.2.1. URI Referencing the Request Object

The Client stores the Request Object resource either locally or remotely at a URI the Authorization Server can access. Such facility may be provided by the authorization server or a trusted third party. For example, the authorization server may provide a URL to which the client POSTs the request object and obtains the Request URI. This URI is the Request Object URI, "request_uri".

It is possible for the Request Object to include values that are to be revealed only to the Authorization Server. As such, the "request_uri" MUST have appropriate entropy for its lifetime so that the URI is not guessable if publicly retrievable. For the guidance, refer to 5.1.4.2.2 of [RFC6819] and Good Practices for Capability URLs [CapURLs]. It is RECOMMENDED that it be removed after a reasonable timeout unless access control measures are taken.

The following is an example of a Request Object URI value (with line wraps within values for display purposes only). In this example, a trusted third-party service hosts the Request Object.

```
https://tfp.example.org/request.jwt/  
GkurKxf5T0Y-mnPFCHqWOMiZi4VS138cQO_V7PZHAdM
```

5.2.2. Request using the "request_uri" Request Parameter

The Client sends the Authorization Request to the Authorization Endpoint.

The following is an example of an Authorization Request using the "request_uri" parameter (with line wraps within values for display purposes only):

```
https://server.example.com/authorize?  
client_id=s6BhdRkqt3  
&request_uri=https%3A%2F%2Ftfp.example.org%2Frequest.jwt  
%2FGkurKxf5T0Y-mnPFCHqWOMiZi4VS138cQO_V7PZHAdM
```

5.2.3. Authorization Server Fetches Request Object

Upon receipt of the Request, the Authorization Server MUST send an HTTP "GET" request to the "request_uri" to retrieve the referenced Request Object, unless it is stored in a way so that it can retrieve it through other mechanism securely, and parse it to recreate the Authorization Request parameters.

The following is an example of this fetch process. In this example, a trusted third-party service hosts the Request Object.

```
GET /request.jwt/GkurKxf5T0Y-mnPFCHqWOMiZi4VS138cQO_V7PZHAdM HTTP/1.1  
Host: tfp.example.org
```


The following is an example of the fetch response:

```
HTTP/1.1 200 OK
Date: Thu, 20 Aug 2020 23:52:39 GMT
Server: Apache/2.4.43 (tfp.example.org)
Content-type: application/oauth-authz-req+jwt
Content-Length: 797
Last-Modified: Wed, 19 Aug 2020 23:52:32 GMT

eyJhbGciOiJSUzI1NiIsImtpZCI6Im9yYmRjIn0.ewogICAgImIzcyI6ICJzNkJoZFJrcXQzIiwKICAgICJhdWQiOiAiaHR0cHM6Ly9zZXJ2ZXIuZXhhbXBsZS5jb20iLAogICAgInJlc3BvbmlX3R5cGUiOiAiY29kZSBpZF90b2t1biIsCiAgICAiY2xpZW50X2lkIjogInM2QmhkUmtxdDMiLAogICAgInJlZGlyZWNOX3VyaSI6ICJodHRwczovL2NsYWVudC5leGFtcGxlLm9yZy9jYiIsCiAgICAic2NvcGUiOiAib3Blbm1kIiwKICAgICJzdGF0ZSI6ICJhZjBpZmpzbGRraiiIsCiAgICAibm9uY2UiOiAibi0wUzZfV3pBMk1qIiwKICAgICJtYXhfYWdlIjogODY0MDAKfQ.Nsxa_18VUELvaPjqW_ToIlyrEJ67BgKb5xsuZRVqzGkfKrOIX7BCx0biSxYGmjK9KJPctH1OC0iQJwXu5YVY-vnW0_PLJb1C2HG-ztVzcnKZC2gE4i0vgQcpkUOCpW3SEYXnyWnKzuKzqSblwAZALo5f89B_p6QA6j6JwBSRvdVsDPdulW8lKxGTbH82czCaQ50rLAg3EYLYaCb4ik4I1zGXE4fvim9FIMS8OCMmzwIB5S-ujFfzwFjoyuPEV4hJnoVUmXR_W9typPf846lGwA8h9G9oNTIuX8Ft2jfpnZdFmLg3_wr3Wa5q3a-lfbgF3S9H_8nN3j1i7tLR_5Nz-g
```

6. Validating JWT-Based Requests

6.1. JWE Encrypted Request Object

If the request object is encrypted, the Authorization Server MUST decrypt the JWT in accordance with the JSON Web Encryption [RFC7516] specification.

The result is a signed request object.

If decryption fails, the Authorization Server MUST return an "invalid_request_object" error to the client in response to the authorization request.

6.2. JWS Signed Request Object

The Authorization Server MUST validate the signature of the JSON Web Signature [RFC7515] signed Request Object. If a "kid" Header Parameter is present, the key identified MUST be the key used, and MUST be a key associated with the client. The signature MUST be validated using a key associated with the client and the algorithm specified in the "alg" Header Parameter. Algorithm verification MUST be performed, as specified in Sections 3.1 and 3.2 of [RFC8725].

If the key is not associated with the client or if signature validation fails, the Authorization Server MUST return an

"invalid_request_object" error to the client in response to the authorization request.

6.3. Request Parameter Assembly and Validation

The Authorization Server MUST extract the set of Authorization Request parameters from the Request Object value. The Authorization Server MUST only use the parameters in the Request Object even if the same parameter is provided in the query parameter. The Client ID values in the "client_id" request parameter and in the Request Object "client_id" claim MUST be identical. The Authorization Server then validates the request as specified in OAuth 2.0 [RFC6749].

If the Client ID check or the request validation fails, then the Authorization Server MUST return an error to the client in response to the authorization request, as specified in Section 5.2 of OAuth 2.0 [RFC6749].

7. Authorization Server Response

Authorization Server Response is created and sent to the client as in Section 4 of OAuth 2.0 [RFC6749].

In addition, this document uses these additional error values:

invalid_request_uri

The "request_uri" in the Authorization Request returns an error or contains invalid data.

invalid_request_object

The request parameter contains an invalid Request Object.

request_not_supported

The Authorization Server does not support the use of the "request" parameter.

request_uri_not_supported

The Authorization Server does not support the use of the "request_uri" parameter.

8. TLS Requirements

Client implementations supporting the Request Object URI method MUST support TLS following Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) [BCP195].

To protect against information disclosure and tampering, confidentiality protection MUST be applied using TLS with a cipher suite that provides confidentiality and integrity protection.

HTTP clients MUST also verify the TLS server certificate, using DNS-ID [RFC6125], to avoid man-in-the-middle attacks. The rules and guidelines defined in [RFC6125] apply here, with the following considerations:

- o Support for DNS-ID identifier type (that is, the `dNSName` identity in the `subjectAltName` extension) is REQUIRED. Certification authorities which issue server certificates MUST support the DNS-ID identifier type, and the DNS-ID identifier type MUST be present in server certificates.
- o DNS names in server certificates MAY contain the wildcard character `"*"`.
- o Clients MUST NOT use CN-ID identifiers; a CN field may be present in the server certificate's subject name, but MUST NOT be used for authentication within the rules described in [BCP195].
- o SRV-ID and URI-ID as described in Section 6.5 of [RFC6125] MUST NOT be used for comparison.

9. IANA Considerations

9.1. OAuth Parameters Registration

Since the request object is a JWT, the core JWT claims cannot be used for any purpose in the request object other than for what JWT dictates. Thus, they need to be registered as OAuth Authorization Request parameters to avoid future OAuth extensions using them with different meanings.

This specification adds the following values to the "OAuth Parameters" registry [IANA.OAuth.Parameters] established by [RFC6749].

- o Name: `"iss"`
- o Parameter Usage Location: authorization request
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.1 of [RFC7519] and this document.
- o Name: `"sub"`
- o Parameter Usage Location: authorization request
- o Change Controller: IETF

- o Specification Document(s): Section 4.1.2 of [RFC7519] and this document.
- o Name: "aud"
- o Parameter Usage Location: authorization request
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.3 of [RFC7519] and this document.
- o Name: "exp"
- o Parameter Usage Location: authorization request
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.4 of [RFC7519] and this document.
- o Name: "nbf"
- o Parameter Usage Location: authorization request
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.5 of [RFC7519] and this document.
- o Name: "iat"
- o Parameter Usage Location: authorization request
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.6 of [RFC7519] and this document.
- o Name: "jti"
- o Parameter Usage Location: authorization request
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.7 of [RFC7519] and this document.

9.2. OAuth Authorization Server Metadata Registry

This specification adds the following value to the "OAuth Authorization Server Metadata" registry [IANA.OAuth.Parameters] established by [RFC8414].

- o Metadata Name: "require_signed_request_object"
- o Metadata Description: Indicates where authorization request needs to be protected as Request Object and provided through either "request" or "request_uri" parameter.
- o Change Controller: IETF
- o Specification Document(s): Section 10.5 of this document.

9.3. OAuth Dynamic Client Registration Metadata Registry

This specification adds the following value to the "OAuth Dynamic Client Registration Metadata" registry [IANA.OAuth.Parameters] established by [RFC7591].

- o Metadata Name: "require_signed_request_object"
- o Metadata Description: Indicates where authorization request needs to be protected as Request Object and provided through either "request" or "request_uri" parameter.
- o Change Controller: IETF
- o Specification Document(s): Section 10.5 of this document.

9.4. Media Type Registration

9.4.1. Registry Contents

This section registers the "application/oauth-authz-req+jwt" media type [RFC2046] in the "Media Types" registry [IANA.MediaTypes] in the manner described in [RFC6838], which can be used to indicate that the content is a JWT containing Request Object claims.

- o Type name: application
- o Subtype name: oauth-authz-req+jwt
- o Required parameters: n/a
- o Optional parameters: n/a
- o Encoding considerations: binary; A Request Object is a JWT; JWT values are encoded as a series of base64url-encoded values (some of which may be the empty string) separated by period ('.') characters.
- o Security considerations: See Section 10 of [[this specification]]
- o Interoperability considerations: n/a
- o Published specification: Section 4 of [[this specification]]
- o Applications that use this media type: Applications that use Request Objects to make an OAuth 2.0 Authorization Request
- o Fragment identifier considerations: n/a
- o Additional information:

Magic number(s): n/a

File extension(s): n/a

Macintosh file type code(s): n/a

- o Person & email address to contact for further information:
Nat Sakimura, nat@nat.consulting
- o Intended usage: COMMON
- o Restrictions on usage: none
- o Author: Nat Sakimura, nat@nat.consulting

- o Change controller: IETF
- o Provisional registration? No

10. Security Considerations

In addition to the all the security considerations discussed in OAuth 2.0 [RFC6819], the security considerations in [RFC7515], [RFC7516], [RFC7518], and [RFC8725] need to be considered. Also, there are several academic papers such as [BASIN] that provide useful insight into the security properties of protocols like OAuth.

In consideration of the above, this document advises taking the following security considerations into account.

10.1. Choice of Algorithms

When sending the authorization request object through "request" parameter, it MUST either be signed using JWS [RFC7515] or signed then encrypted using JWS [RFC7515] and JWE [RFC7516] respectively, with then considered appropriate algorithms.

10.2. Request Source Authentication

The source of the Authorization Request MUST always be verified. There are several ways to do it:

- (a) Verifying the JWS Signature of the Request Object.
- (b) Verifying that the symmetric key for the JWE encryption is the correct one if the JWE is using symmetric encryption. Note however, that if public key encryption is used, no source authentication is enabled by the encryption, as any party can encrypt content to the public key.
- (c) Verifying the TLS Server Identity of the Request Object URI. In this case, the Authorization Server MUST know out-of-band that the Client uses Request Object URI and only the Client is covered by the TLS certificate. In general, it is not a reliable method.
- (d) When an Authorization Server implements a service that returns a Request Object URI in exchange for a Request Object, the Authorization Server MUST perform Client Authentication to accept the Request Object and bind the Client Identifier to the Request Object URI it is providing. It MUST validate the signature, per (a). Since Request Object URI can be replayed, the lifetime of the Request Object URI MUST be short and preferably one-time use. The entropy of the Request Object URI

MUST be sufficiently large. The adequate shortness of the validity and the entropy of the Request Object URI depends on the risk calculation based on the value of the resource being protected. A general guidance for the validity time would be less than a minute and the Request Object URI is to include a cryptographic random value of 128bit or more at the time of the writing of this specification.

- (e) When a trusted third-party service returns a Request Object URI in exchange for a Request Object, it MUST validate the signature, per (a). In addition, the Authorization Server MUST be trusted by the third-party service and MUST know out-of-band that the client is also trusted by it.

10.3. Explicit Endpoints

Although this specification does not require them, research such as [BASIN] points out that it is a good practice to explicitly state the intended interaction endpoints and the message position in the sequence in a tamper evident manner so that the intent of the initiator is unambiguous. The following endpoints defined in [RFC6749], [RFC6750], and [RFC8414] are RECOMMENDED by this specification to use this practice :

- (a) Protected Resources ("protected_resources")
- (b) Authorization Endpoint ("authorization_endpoint")
- (c) Redirection URI ("redirect_uri")
- (d) Token Endpoint ("token_endpoint")

Further, if dynamic discovery is used, then this practice also applies to the discovery related endpoints.

In [RFC6749], while Redirection URI is included in the Authorization Request, others are not. As a result, the same applies to Authorization Request Object.

10.4. Risks Associated with request_uri

The introduction of "request_uri" introduces several attack possibilities. Consult the security considerations in Section 7 of RFC3986 [RFC3986] for more information regarding risks associated with URIs.

10.4.1. DDoS Attack on the Authorization Server

A set of malicious client can launch a DoS attack to the authorization server by pointing the "request_uri" to a URI that returns extremely large content or extremely slow to respond. Under such an attack, the server may use up its resource and start failing.

Similarly, a malicious client can specify the "request_uri" value that itself points to an authorization request URI that uses "request_uri" to cause the recursive lookup.

To prevent such attack to succeed, the server should (a) check that the value of "request_uri" parameter does not point to an unexpected location, (b) check the media type of the response is "application/oauth-authz-req+jwt", (c) implement a time-out for obtaining the content of "request_uri", and (d) not perform recursive GET on the "request_uri".

10.4.2. Request URI Rewrite

The value of "request_uri" is not signed thus it can be tampered by Man-in-the-browser attacker. Several attack possibilities rise because of this, e.g., (a) attacker may create another file that the rewritten URI points to making it possible to request extra scope (b) attacker launches a DoS attack to a victim site by setting the value of "request_uri" to be that of the victim.

To prevent such attack to succeed, the server should (a) check that the value of "request_uri" parameter does not point to an unexpected location, (b) check the media type of the response is "application/oauth-authz-req+jwt", and (c) implement a time-out for obtaining the content of "request_uri".

10.5. Downgrade Attack

Unless the protocol used by client and the server is locked down to use OAuth JAR, it is possible for an attacker to use RFC6749 requests to bypass all the protection provided by this specification.

To prevent it, this specification defines a new client metadata and server metadata "require_signed_request_object" whose value is a boolean.

When the value of it as a client metadata is "true", then the server MUST reject the authorization request from the client that does not conform to this specification. It MUST also reject the request if the request object uses "alg":"none" when this client metadata value is "true". If omitted, the default value is "false".

When the value of it as a server metadata is "true", then the server MUST reject the authorization request from any client that does not conform to this specification. It MUST also reject the request if the request object uses "alg":"none" when this server metadata value is "true". If omitted, the default value is "false".

Note that even if "require_signed_request_object" metadata values are not present, the client MAY use signed request objects, provided that there are signing algorithms mutually supported by the client and the server. Use of signing algorithm metadata is described in Section 4.

10.6. TLS Security Considerations

Current security considerations can be found in Recommendations for Secure Use of TLS and DTLS [BCP195]. This supersedes the TLS version recommendations in OAuth 2.0 [RFC6749].

10.7. Parameter Mismatches

Given that OAuth parameter values are being sent in two different places, as normal OAuth parameters and as Request Object claims, implementations must guard against attacks that could use mismatching parameter values to obtain unintended outcomes. That is the reason that the two Client ID values MUST match, the reason that only the parameter values from the Request Object are to be used, and the reason that neither "request" nor "request_uri" can appear in a Request Object.

10.8. Cross-JWT Confusion

As described in Section 2.8 of [RFC8725], attackers may attempt to use a JWT issued for one purpose in a context that it was not intended for. The mitigations described for these attacks can be applied to Request Objects.

One way that an attacker might attempt to repurpose a Request Object is to try to use it as a client authentication JWT, as described in Section 2.2 of [RFC7523]. A simple way to prevent this is to never use the Client ID as the "sub" value in a Request Object.

Another way to prevent cross-JWT confusion is to use explicit typing, as described in Section 3.11 of [RFC8725]. One would explicitly type a Request Object by including a "typ" Header Parameter with the value "oauth-authz-req+jwt" (which is registered in Section 9.4.1. Note however, that requiring explicitly typed Requests Objects at existing authorization servers will break most existing deployments, as existing clients are already commonly using untyped Request Objects, especially with OpenID Connect [OpenID.Core]. However, requiring

explicit typing would be a good idea for new OAuth deployment profiles where compatibility with existing deployments is not a consideration.

Finally, yet another way to prevent cross-JWT confusion is to use a key management regime in which keys used to sign Request Objects are identifiably distinct from those used for other purposes. Then, if an adversary attempts to repurpose the Request Object in another context, a key mismatch will occur, thwarting the attack.

11. Privacy Considerations

When the Client is being granted access to a protected resource containing personal data, both the Client and the Authorization Server need to adhere to Privacy Principles. RFC 6973 Privacy Considerations for Internet Protocols [RFC6973] gives excellent guidance on the enhancement of protocol design and implementation. The provision listed in it should be followed.

Most of the provision would apply to The OAuth 2.0 Authorization Framework [RFC6749] and The OAuth 2.0 Authorization Framework: Bearer Token Usage [RFC6750] and are not specific to this specification. In what follows, only the specific provisions to this specification are noted.

11.1. Collection limitation

When the Client is being granted access to a protected resource containing personal data, the Client SHOULD limit the collection of personal data to that which is within the bounds of applicable law and strictly necessary for the specified purpose(s).

It is often hard for the user to find out if the personal data asked for is strictly necessary. A trusted third-party service can help the user by examining the Client request and comparing to the proposed processing by the Client and certifying the request. After the certification, the Client, when making an Authorization Request, can submit Authorization Request to the trusted third-party service to obtain the Request Object URI. This process is two steps:

- (1) (Certification Process) The trusted third-party service examines the business process of the client and determines what claims they need: This is the certification process. Once the client is certified, then they are issued a client credential to authenticate against to push request objects to the trusted third-party service to get the "request_uri".

- (2) (Translation Process) The client uses the client credential that it got to push the request object to the trusted third-party service to get the "request_uri". The trusted third-party service also verifies that the Request Object is consistent with the claims that the client is eligible for, per prior step.

Upon receiving such Request Object URI in the Authorization Request, the Authorization Server first verifies that the authority portion of the Request Object URI is a legitimate one for the trusted third-party service. Then, the Authorization Server issues HTTP GET request to the Request Object URI. Upon connecting, the Authorization Server MUST verify the server identity represented in the TLS certificate is legitimate for the Request Object URI. Then, the Authorization Server can obtain the Request Object, which includes the "client_id" representing the Client.

The Consent screen MUST indicate the Client and SHOULD indicate that the request has been vetted by the trusted third-party service for adherence to the Collection Limitation principle.

11.2. Disclosure Limitation

11.2.1. Request Disclosure

This specification allows extension parameters. These may include potentially sensitive information. Since URI query parameter may leak through various means but most notably through referrer and browser history, if the authorization request contains a potentially sensitive parameter, the Client SHOULD JWE [RFC7516] encrypt the request object.

Where Request Object URI method is being used, if the request object contains personally identifiable or sensitive information, the "request_uri" SHOULD be used only once, have a short validity period, and MUST have large enough entropy deemed necessary with applicable security policy unless the Request Object itself is JWE [RFC7516] Encrypted. The adequate shortness of the validity and the entropy of the Request Object URI depends on the risk calculation based on the value of the resource being protected. A general guidance for the validity time would be less than a minute and the Request Object URI is to include a cryptographic random value of 128bit or more at the time of the writing of this specification.

11.2.2. Tracking using Request Object URI

Even if the protected resource does not include a personally identifiable information, it is sometimes possible to identify the user through the Request Object URI if persistent static per-user

Request Object URIs are used. A third party may observe it through browser history etc. and start correlating the user's activity using it. In a way, it is a data disclosure as well and should be avoided.

Therefore, per-user persistent Request Object URIs should be avoided. Single-use Request Object URIs are one alternative.

12. Acknowledgements

The following people contributed to the creation of this document in the OAuth working group and other IETF roles. (Affiliations at the time of the contribution are used.)

Annabelle Backman (Amazon), Dirk Balfanz (Google), Sergey Beryozkin, Ben Campbell (as AD), Brian Campbell (Ping Identity), Roman Danyliw (as AD), Martin Duke (as AD), Vladimir Dzhuvinov (Connect2id), Lars Eggert (as AD), Joel Halpern (as GENART), Benjamin Kaduk (as AD), Stephen Kent (as SECDIR), Murray Kucherawy (as AD), Warren Kumari (as OPSDIR), Watson Ladd (as SECDIR), Torsten Lodderstedt (yes.com), Jim Manico, Axel Nennker (Deutsche Telecom), Hannes Tschofenig (ARM), James H. Manger (Telstra), Kathleen Moriarty (as AD), John Panzer (Google), Francesca Palombini (as AD), David Recordon (Facebook), Marius Scurtescu (Google), Luke Shepard (Facebook), Filip Skokan (Auth0), Eric Vyncke (as AD), and Robert Wilton (as AD).

The following people contributed to creating this document through the OpenID Connect Core 1.0 [OpenID.Core].

Brian Campbell (Ping Identity), George Fletcher (AOL), Ryo Itou (Mixi), Edmund Jay (Illumila), Breno de Medeiros (Google), Hideki Nara (TACT), Justin Richer (MITRE).

13. Revision History

Note to the RFC Editor: Please remove this section from the final RFC.

-34

- o Addressed additional IESG comments by Murray Kucherawy and Francesca Palombini.

-33

- o Addressed IESG comments prior to 8-Apr-21 telechat. Thanks to Martin Duke, Lars Eggert, Benjamin Kaduk, Francesca Palombini, and Eric Vyncke for their reviews.

-32

- o Removed outdated JSON reference.

-31

- o Addressed SecDir review comments by Watson Ladd.

-30

- o Changed the MIME Type from "oauth.authz.req+jwt" to "oauth-authz-req+jwt", per advice from the designated experts.

-29

- o Uniformly use the Change Controller "IETF".

-28

- o Removed unused references, as suggested by Roman Danyliw.

-27

- o Edits by Mike Jones to address IESG and working group review comments, including:
- o Added Security Considerations text saying not to use the Client ID as the "sub" value to prevent Cross-JWT Confusion.
- o Added Security Considerations text about using explicit typing to prevent Cross-JWT Confusion.
- o Addressed Eric Vyncke's review comments.
- o Addressed Robert Wilton's review comments.
- o Addressed Murray Kucherawy's review comments.
- o Addressed Benjamin Kaduk's review comments.
- o Applied spelling and grammar corrections.

-20

- o BK comments
- o Section 3 Removed WAP

- o Section 4. Clarified authorization request object parameters, removed extension parameters from examples
- o Section 4. Specifies application/oauth.authz.req+jwt as mime-type for request objects
- o Section 5.2.1 Added reference to Capability URLs
- o Section 5.2.3. Added entropy fragment to example request
- o Section 8. Replaced "subjectAltName dnsName" with "DNS-ID"
- o Section 9. Registers authorization request parameters in JWT Claims Registry.
- o Section 9. Registers application/oauth.authz.req in IANA mime-types registry
- o Section 10.1. Clarified encrypted request objects are "signed then encrypted" to maintain consistency
- o Section 10.2. Clarifies trust between AS and TFP
- o Section 10.3. Clarified endpoints subject to the practice
- o Section 10.4 Replaced "redirect_uri" to "request_uri"
- o Section 10.4. Added reference to RFC 3986 for risks
- o Section 10.4.1.d Deleted "do" to maintain grammar flow
- o Section 10.4.1, 10.4.2 Replaced "application/jose" to "application/jwt"
- o Section 12.1. Extended description for submitting authorization request to TFP to obtain request object
- o Section 12.2.2. Replaced per-user Request Object URI with static per-user Request URIs
- o Section 13. Combined OAuth WG contributors together
- o Section Whole doc Replaced application/jwt with application/oauth.authz.req+jwt

-19

- o AD comments

- o Section 5.2.1. s/Requiest URI/Request URI/
- o Section 8 s/[BCP195] ./[BCP195]./
- o Section 10.3. s/sited/cited/
- o Section 11. Typo. s/Curent/Current/

-17

- o #78 Typos in content-type

-16

- o Treated remaining Ben Campbell comments.

-15

- o Removed further duplication

-14

- o #71 Reiterate dynamic params are included.
- o #70 Made clear that AS must return error.
- o #69 Inconsistency of the need to sign.
- o Fixed Mime-type.
- o #67 Inconsistence in requiring HTTPS in request URI.
- o #66 Dropped ISO 29100 reference.
- o #25 Removed Encrypt only option.
- o #59 Same with #25.

-13

- o add TLS Security Consideration section
- o replace RFC7525 reference with BCP195
- o moved front tag in FETT reference to fix XML structure
- o changes reference from SoK to FETT

-12

- o fixes #62 - Alexey Melnikov Discuss
- o fixes #48 - OPSDIR Review : General - delete semicolons after list items
- o fixes #58 - DP Comments for the Last Call
- o fixes #57 - GENART - Remove "non-normative ... " from examples.
- o fixes #45 - OPSDIR Review : Introduction - are attacks discovered or already opened
- o fixes #49 - OPSDIR Review : Introduction - Inconsistent colons after initial sentence of list items.
- o fixes #53 - OPSDIR Review : 6.2 JWS Signed Request Object - Clarify JOSE Header
- o fixes #42 - OPSDIR Review : Introduction - readability of 'and' is confusing
- o fixes #50 - OPSDIR Review : Section 4 Request Object - Clarify 'signed, encrypted, or signed and encrypted'
- o fixes #39 - OPSDIR Review : Abstract - Explain/Clarify JWS and JWE
- o fixed #50 - OPSDIR Review : Section 4 Request Object - Clarify 'signed, encrypted, or signed and encrypted'
- o fixes #43 - OPSDIR Review : Introduction - 'properties' sounds awkward and are not exactly 'properties'
- o fixes #56 - OPSDIR Review : 12 Acknowledgements - 'contribution is' => 'contribution are'
- o fixes #55 - OPSDIR Review : 11.2.2 Privacy Considerations - ' It is in a way' => 'In a way, it is'
- o fixes #54 - OPSDIR Review : 11 Privacy Considerations - 'and not specific' => 'and are not specific'
- o fixes #51 - OPSDIR Review : Section 4 Request Object - 'It is fine' => 'It is recommended'
- o fixes #47 - OPSDIR Review : Introduction - 'over- the- wire' => 'over-the-wire'

- o fixes #46 - OPSDIR Review : Introduction - 'It allows' => 'The use of application security' for
- o fixes #44 - OPSDIR Review : Introduction - 'has' => 'have'
- o fixes #41 - OPSDIR Review : Introduction - missing 'is' before 'typically sent'
- o fixes #38 - OPSDIR Review : Section 11 - Delete 'freely accessible' regarding ISO 29100

-11

- o s/bing/being/
- o Added history for -10

-10

- o #20: KM1 -- some wording that is awkward in the TLS section.
- o #21: KM2 - the additional attacks against OAuth 2.0 should also have a pointer
- o #22: KM3 -- Nit: in the first line of 10.4:
- o #23: KM4 -- Mention RFC6973 in Section 11 in addition to ISO 29100
- o #24: SECDIR review: Section 4 -- Confusing requirements for sign+encrypt
- o #25: SECDIR review: Section 6 -- authentication and integrity need not be provided if the requestor encrypts the token?
- o #26: SECDIR Review: Section 10 -- why no reference for JWS algorithms?
- o #27: SECDIR Review: Section 10.2 - how to do the agreement between client and server "a priori"?
- o #28: SECDIR Review: Section 10.3 - Indication on "large entropy" and "short lifetime" should be indicated
- o #29: SECDIR Review: Section 10.3 - Typo
- o #30: SECDIR Review: Section 10.4 - typos and missing articles

- o #31: SECDIR Review: Section 10.4 - Clearer statement on the lack of endpoint identifiers needed
- o #32: SECDIR Review: Section 11 - ISO29100 needs to be moved to normative reference
- o #33: SECDIR Review: Section 11 - Better English and Entropy language needed
- o #34: Section 4: Typo
- o #35: More Acknowledgment
- o #36: DP - More precise qualification on Encryption needed.

-09

- o Minor Editorial Nits.
- o Section 10.4 added.
- o Explicit reference to Security consideration (10.2) added in section 5 and section 5.2.
- o , (add yourself) removed from the acknowledgment.

-08

- o Applied changes proposed by Hannes on 2016-06-29 on IETF OAuth list recorded as <https://bitbucket.org/Nat/oauth-jwsreq/issues/12/>.
- o TLS requirements added.
- o Security Consideration reinforced.
- o Privacy Consideration added.
- o Introduction improved.

-07

- o Changed the abbrev to OAuth JAR from oauth-jar.
- o Clarified sig and enc methods.
- o Better English.

- o Removed claims from one of the example.
- o Re-worded the URI construction.
- o Changed the example to use request instead of request_uri.
- o Clarified that Request Object parameters take precedence regardless of request or request_uri parameters were used.
- o Generalized the language in 4.2.1 to convey the intent more clearly.
- o Changed "Server" to "Authorization Server" as a clarification.
- o Stopped talking about request_object_signing_alg.
- o IANA considerations now reflect the current status.
- o Added Brian Campbell to the contributors list. Made the lists alphabetic order based on the last names. Clarified that the affiliation is at the time of the contribution.
- o Added "older versions of " to the reference to IE URI length limitations.
- o Stopped talking about signed or unsigned JWS etc.
- o 1.Introduction improved.

-06

- o Added explanation on the 512 chars URL restriction.
- o Updated Acknowledgements.

-05

- o More alignment with OpenID Connect.

-04

- o Fixed typos in examples. (request_url -> request_uri, cliend_id -> client_id)
- o Aligned the error messages with the OAuth IANA registry.
- o Added another rationale for having request object.

-03

- o Fixed the non-normative description about the advantage of static signature.
- o Changed the requirement for the parameter values in the request itself and the request object from 'MUST MATCH' to 'Req Obj takes precedence'.

-02

- o Now that they are RFCs, replaced JWS, JWE, etc. with RFC numbers.

-01

- o Copy Edits.

14. References

14.1. Normative References

- [BCP195] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, May 2015.
- [IANA.MediaTypes] IANA, "Media Types", <<http://www.iana.org/assignments/media-types>>.
- [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>>.
- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <<https://www.rfc-editor.org/info/rfc3629>>.
- [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>>.

- [RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <<https://www.rfc-editor.org/info/rfc6125>>.
- [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>>.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <<https://www.rfc-editor.org/info/rfc6750>>.
- [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>>.
- [RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <<https://www.rfc-editor.org/info/rfc7515>>.
- [RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <<https://www.rfc-editor.org/info/rfc7516>>.
- [RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <<https://www.rfc-editor.org/info/rfc7518>>.
- [RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <<https://www.rfc-editor.org/info/rfc7519>>.
- [RFC8141] Saint-Andre, P. and J. Klensin, "Uniform Resource Names (URNs)", RFC 8141, DOI 10.17487/RFC8141, April 2017, <<https://www.rfc-editor.org/info/rfc8141>>.
- [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>>.

- [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/info/rfc8259>>.
- [RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <<https://www.rfc-editor.org/info/rfc8414>>.

14.2. Informative References

- [BASIN] Basin, D., Cremers, C., and S. Meier, "Provably Repairing the ISO/IEC 9798 Standard for Entity Authentication", Journal of Computer Security - Security and Trust Principles Volume 21 Issue 6, Pages 817-846, November 2013, <<https://www.cs.ox.ac.uk/people/cas.cremers/downloads/papers/BCM2012-iso9798.pdf>>.
- [CapURLs] Tennison, J., "Good Practices for Capability URLs", W3C Working Draft, February 2014, <<https://www.w3.org/TR/capability-urls/>>.
- [IANA.JWT.Claims] IANA, "JSON Web Token Claims", <<http://www.iana.org/assignments/jwt>>.
- [IANA.OAuth.Parameters] IANA, "OAuth Parameters", <<http://www.iana.org/assignments/oauth-parameters>>.
- [OpenID.Core] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", OpenID Foundation Standards, February 2014, <http://openid.net/specs/openid-connect-core-1_0.html>.
- [RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, DOI 10.17487/RFC2046, November 1996, <<https://www.rfc-editor.org/info/rfc2046>>.
- [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>>.

- [RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <<https://www.rfc-editor.org/info/rfc6838>>.
- [RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <<https://www.rfc-editor.org/info/rfc6973>>.
- [RFC7523] Jones, M., Campbell, B., and C. Mortimore, "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7523, DOI 10.17487/RFC7523, May 2015, <<https://www.rfc-editor.org/info/rfc7523>>.
- [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>>.
- [RFC8725] Sheffer, Y., Hardt, D., and M. Jones, "JSON Web Token Best Current Practices", BCP 225, RFC 8725, DOI 10.17487/RFC8725, February 2020, <<https://www.rfc-editor.org/info/rfc8725>>.

Authors' Addresses

Nat Sakimura
NAT.Consulting
2-22-17 Naka
Kunitachi, Tokyo 186-0004
Japan

Phone: +81-42-580-7401
Email: nat@nat.consulting
URI: <http://nat.sakimura.org/>

John Bradley
Yubico
Casilla 177, Sucursal Talagante
Talagante, RM
Chile

Phone: +1.202.630.5272
Email: ve7jtb@ve7jtb.com
URI: <http://www.thread-safe.com/>

Michael B. Jones
Microsoft
One Microsoft Way
Redmond, Washington 98052
United States of America

Email: mbj@microsoft.com
URI: <https://self-issued.info/>

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 7, 2017

M. Jones
Microsoft
J. Bradley
Ping Identity
N. Sakimura
NRI
July 6, 2016

OAuth 2.0 Mix-Up Mitigation
draft-ietf-oauth-mix-up-mitigation-01

Abstract

This specification defines an extension to The OAuth 2.0 Authorization Framework that enables the authorization server to dynamically provide the client using it with additional information about the current protocol interaction that can be validated by the client and that enables the client to dynamically provide the authorization server with additional information about the current protocol interaction that can be validated by the authorization server. This additional information can be used by the client and the authorization server to prevent classes of attacks in which the client might otherwise be tricked into using inconsistent sets of metadata from multiple authorization servers, including potentially using a token endpoint that does not belong to the same authorization server as the authorization endpoint used. Recent research publications refer to these as "IdP Mix-Up" and "Malicious Endpoint" attacks.

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 January 7, 2017.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Requirements Notation and Conventions	4
1.2. Terminology	4
2. The OAuth Issuer Identifier	4
3. Mitigation Data Returned in Authorization Response	5
3.1. Mitigation Data Returned in Authorization Response Parameters	5
3.1.1. Example Authorization Response using Response Parameters	5
3.2. Mitigation Data Returned in JWT	6
3.2.1. Example Authorization Response using JWT	6
4. Validating the Authorization Response	7
5. Mitigation Data Sent to the Token Endpoint	8
5.1. Example Token Request	8
6. Validating the Token Request	9
7. Security Considerations	9
7.1. IdP Mix-Up and Malicious Endpoint Attacks	9
7.2. Duplicate Information Attacks	9
7.3. Cut-and-Paste Attacks	10
8. IANA Considerations	11
8.1. OAuth Parameters Registration	11
8.1.1. Registry Contents	11
9. References	11
9.1. Normative References	11
9.2. Informative References	12
Appendix A. Implementation Notes	13
Appendix B. Acknowledgements	13
Appendix C. Open Issues	14
Appendix D. Document History	14
Authors' Addresses	14

1. Introduction

OAuth 2.0 [RFC6749] clients use multiple authorization server endpoints when using some OAuth response types. For instance, when using the "code" response type, the client uses both the authorization endpoint and the token endpoint. It is important that endpoints belonging to the same authorization server always be used together. Otherwise, information produced by one authorization server could mistakenly be sent by the client to different authorization server, resulting in some of the attacks described in Section 7. Recent research publications refer to these specific attacks as "IdP Mix-Up" [arXiv.1601.01229v2] and "Malicious Endpoint" [arXiv.1508.04324v2] attacks.

The client obviously cannot be confused into using endpoints from multiple authorization servers in an authorization flow if the client is configured to use only a single authorization server. However, the client can potentially be tricked into mixing endpoints if it is configured to use more than one authorization server, whether the configuration is dynamic or static. The client may be confused if it has no way to determine whether the set of endpoints belongs to the same authorization server. Or, a client may be confused simply because it is receiving authorization responses from more than one authorization server at the same redirection endpoint and the client is insufficiently able to determine that the response received is associated with the correct authorization server.

This specification enables the authorization server to dynamically provide the client using it with additional information about the current protocol interaction that can be validated by the client and that enables the client to dynamically provide the authorization server with additional information about the current protocol interaction that can be validated by the authorization server. This enables them to abort interactions in which endpoints from multiple authorization servers would otherwise be used.

The mitigation data provided by the authorization server to the client is an issuer identifier, which is used to identify the authorization server, and a client ID, which is used to verify that the response is from the correct authorization server and is intended for this client. The issuer identifier is defined in Section 2 of [OAuth.Discovery]. If supported by the authorization server, the issuer identifier can also be used to obtain a consistent set of metadata describing the authorization server configuration, as also described in [OAuth.Discovery].

This mitigation data is returned to the client in the authorization response. The syntax for returning the mitigation data from the

authorization server is dependent upon the OAuth response type being used. The syntax used with the existing response types registered in the IANA "OAuth Authorization Endpoint Response Types" registry [IANA.OAuth.Parameters] as of the time of this writing is defined by this specification. Two of these response types are defined by RFC 6749 [RFC6749]; the rest are defined by [OAuth.Responses].

The mitigation data provided by the client to the authorization server is the existing "state" value defined by RFC 6749 [RFC6749], but adding also sending it from the client to the token endpoint. This is used by the authorization server to verify that the authorization code and state both belong to the same protocol interaction.

1.1. Requirements Notation and 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 RFC 2119 [RFC2119].

1.2. Terminology

This specification uses the terms "Access Token", "Authorization Code", "Authorization Endpoint", "Authorization Grant", "Authorization Server", "Client", "Client Authentication", "Client Identifier", "Client Secret", "Grant Type", "Protected Resource", "Redirection URI", "Refresh Token", "Resource Owner", "Resource Server", "Response Type", and "Token Endpoint" defined by OAuth 2.0 [RFC6749], the terms "Claim Name", "Claim Value", and "JSON Web Token (JWT)" defined by JSON Web Token (JWT) [JWT].

2. The OAuth Issuer Identifier

The OAuth issuer identifier serves as a concrete identifier for the authorization server. As defined in [OAuth.Discovery], the issuer identifier is a URL that uses the "https" scheme and has no query or fragment components. Also as specified there, this is the location where ".well-known" RFC 5785 [RFC5785] resources containing information about the authorization server are published. In particular, when discovery is supported, the authorization server's metadata is retrieved as a JSON document [RFC7159] from a path derived from this URL. This metadata document contains a consistent set of metadata describing the authorization server configuration.

Implementations supporting this specification MAY also support discovery or they MAY simply use the issuer identifier as a concrete identifier for the authorization server. This specification does not

rely upon the authorization server publishing or the client retrieving a discovery metadata document.

3. Mitigation Data Returned in Authorization Response

Mitigating the attacks relies on the authorization server returning additional data about the interaction and the client checking that data. The mitigation data returned is the client ID and the issuer identifier. The syntax for returning the mitigation data from the authorization server is dependent upon the OAuth response type being used.

3.1. Mitigation Data Returned in Authorization Response Parameters

Some OAuth response types do not already return the issuer identifier and client ID in the authorization response. When this is the case, the mitigation data is returned as additional OAuth response parameters.

These new response parameters are defined for this purpose:

`client_id`

Client that this response is intended for. It MUST contain the OAuth 2.0 client ID of the client as its value.

`iss`

Issuer identifier for the authorization server issuing the response. The "iss" value is a case-sensitive URL using the "https" scheme that contains scheme, host, and optionally, port number and path components and no query or fragment components.

As of the time of this writing, these are the existing response types that are registered in the IANA "OAuth Authorization Endpoint Response Types" registry [IANA.OAuth.Parameters] that do not already return the issuer identifier and client ID in the authorization response: "code", "code token", "none", and "token". Therefore, the client ID and issuer are returned using the new authorization response parameters when using these response types. To avoid duplication, as discussed in Section 7.2, it is NOT RECOMMENDED to also return them in this manner when the response type already returns these values in the authorization response.

3.1.1. Example Authorization Response using Response Parameters

The following example authorization response is to a request that used the "code" response type. It uses the "iss" and "client_id" response parameters to return the mitigation information to the client.

The example successful authorization response follows (with line breaks within lines for display purposes only):

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb?
  code=Qcb0Orvlzh30vLlMPRsbm-diHiMwcLyZvnlarpZv-Jxf_11jnpEX3Tgfvk
  &state=nrsz6AnHzPSVVBYPVTXV6ZTXQeg_eih7hdpewHNXmZ8
  &iss=https://server.example.com
  &client_id=5d9e8a36-569d-4c40-8d6b-6e279ac1c5f1
```

3.2. Mitigation Data Returned in JWT

As of the time of this writing, these are the existing response types that are registered in the IANA "OAuth Authorization Endpoint Response Types" registry [IANA.OAuth.Parameters] that already return the issuer identifier and client ID in the authorization response: "code id_token", "code id_token token", "id_token", and "id_token token". All of these return these values as the "iss" (issuer) claim value and as an "aud" (audience) claim value in a signed ID Token, which is a JSON Web Token [JWT], as specified in "OpenID Connect Core 1.0" [OpenID.Core]. When using these response types, the client MUST use the client ID and issuer values returned in the ID Token for validating the mitigation data.

3.2.1. Example Authorization Response using JWT

The following example authorization response is to a request that used the "id_token token" response type. It uses the "iss" and "aud" claims in the ID Token to return the mitigation information to the client.

The example successful authorization response follows (with line breaks within lines for display purposes only):

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb#
  access_token=jHkWEdUXMUlBwAsC4vtUsZwnNvTlIxEl0z9K3vx5KF0Y
  &token_type=Bearer
  &id_token=eyJraWQioiIxzTlnZGs3IiwiaWxnbG9kaXkiOiJpYXQiOiAxMzEx
ewogImlzcyI6ICJodHRwczovL3NlcnZlci5leGFtcGxlLmNvbSIsCiAic3ViIjog
IjI0ODI4OTc2MTAwMSIsCiAiYXVkJjogInM2QmhhkUmtxdDMLAogIm5vbmNlIjog
Im4tMFM2Xld6QTJNaIIsCiAiZXhwIjogMTMTxMTI4MTk3MCwKICJpYXQiOiAxMzEx
MjgwOTcwLAogImF0X2hhc2giOiAiNzdrbVVQdGpQZnpXdEYyQW5wSzlSUSIKfQ.
kdqTmftlaXg5WBYBr1wkxhkqCGZPc0k8vTiV5g2jj67jQ7XkrDamYx2bOkZLdZrp
MPIzkdyBlnZI_G8vQGQuamRhJcEIt21kblGPZ-yhEhdkAiZIZLu38rChalDS2Mh0
glE_rke5XXRhmqgoEFFdZiFdnO3p61-7y51co840EAZvARSINQaOWIzvioRfs4zw
IF0aT33Vpxfqr8HDY31z09eBW2dSQuCa071z0ENWChWoPliKlJCo_Bk9eDg2uwo
2ZwshsvH3jQTMQ0lYOTzufSlSmXIKfj10sb3nftQeR697_hA-nmZyAdL8_NRfaC37
XnAbW8WB9wCfECp7cuNuOg
  &state=af0ifjsldkj
```

Decoding the ID Token in the response will yield the following claims, which includes the mitigation information in the "iss" and "aud" claims:

```
{
  "iss": "https://server.example.com",
  "sub": "248289761001",
  "aud": "s6BhdRkqt3",
  "nonce": "n-0S6_WzA2Mj",
  "exp": 1311281970,
  "iat": 1311280970,
  "at_hash": "77QmUPtjPpzWtF2AnpK9RQ"
}
```

4. Validating the Authorization Response

Upon receiving the mitigation data in an authorization response, the client MUST validate that the response was intended for it and that the authorization server metadata that it obtained at client registration time is consistent with the authorization server metadata contained in the metadata referenced by the issuer identifier.

The client MUST validate the authorization server configuration as follows:

1. Compare the issuer identifier for the authorization server that the client received when it registered at the authorization

server that it made the request to with the issuer value returned in the "iss" response parameter or the "iss" claim in the ID Token, depending upon the response type being used. If they do not exactly match, the client MUST NOT proceed with the authorization.

2. Verify that the response is intended for this client by confirming that the client's client identifier for the authorization server the request was made to matches the value of the "client_id" response parameter or that the client's client identifier is an audience value of the ID Token, depending upon the response type being used. If not, the client MUST NOT proceed with the authorization.

5. Mitigation Data Sent to the Token Endpoint

Mitigating the attacks also relies on the client sending additional data about the interaction to the token endpoint, for response types that use it, and the authorization server checking that data. The mitigation data sent is the same state value that is sent in the authorization request and returned in the authorization response. This specification defines the new "state" token request parameter for passing this additional information.

As of the time of this writing, these are the existing response types that are registered in the IANA "OAuth Authorization Endpoint Response Types" registry [IANA.OAuth.Parameters] that use the token endpoint: "code", "code id_token", "code id_token token", and "code token". The state value is to be sent in the "state" token request parameter when using these response types, and any new response types registered that use the token endpoint.

5.1. Example Token Request

The following example token request is part of a protocol interaction that used the "code" response type. It uses the "state" request parameter to send mitigation information to the authorization server.

The example of token request follows (with line breaks within lines for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

grant_type=authorization_code
&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
&state=ZSGXNBavNc-B3kU3DeJnZoWWOzYxsbvj7jp-S0x_z8U
```

6. Validating the Token Request

When the authorization server receives a token request at the token endpoint that contains a value in the "state" parameter, it **MUST** validate that the state value received exactly matches the state value previously received in the corresponding authorization request. If the recorded state value and the state value received do not exactly match, the authorization server **MUST NOT** proceed with the authorization.

7. Security Considerations

7.1. IdP Mix-Up and Malicious Endpoint Attacks

The attacks mitigated by this extension are described in detail in "A Comprehensive Formal Security Analysis of OAuth 2.0" [arXiv.1601.01229v2] and "On the security of modern Single Sign-On Protocols: Second-Order Vulnerabilities in OpenID Connect" [arXiv.1508.04324v2]. To mitigate these attacks, clients configured to use more than one authorization server should use authorization servers that return issuer and client ID information and should validate that a consistent set of authorization server endpoints are being used when using response types that utilize multiple endpoints.

When registering, clients **SHOULD NOT** allow multiple authorization servers to return the same issuer value, and **MUST NOT** allow multiple authorization servers to return the same issuer and client ID value pair.

7.2. Duplicate Information Attacks

If a protocol is defined to return the same information in multiple locations, this can create an additional attack surface. Knowing that the information is supposed to be the same, recipients will often be lazy and use the information from only one of the locations,

not validating that all the supposedly duplicate instances are the same. This can enable attackers to create illegal protocol messages that have different values in the multiple locations and those illegal messages will not be detected or rejected by these lazy recipients.

For this reason, if an OAuth profile is being used that returns the mitigation information defined by this specification in one location, it SHOULD NOT also be returned in another. In particular, if a JWT containing the client ID and issuer values is being returned in the authorization response, they SHOULD NOT also be returned as individual authorization response parameters.

7.3. Cut-and-Paste Attacks

OAuth authorization responses are sent as redirects to redirection URIs, with the response parameters typically passed as URI query parameters or fragment values. A "cut-and-paste" attack is performed by the attacker creating what appears to be a legitimate authorization response, but that substitutes some of the response parameter values with values of the attacker's choosing. Sometimes this is done by copying or "cutting" some values out of a legitimate response and replacing or "pasting" some of these values into a different response, the original version of which may have also been legitimate, creating a combination of response values that are not legitimate and that may cause behaviors sought by the attacker. The Code Substitution threat described in Section 4.4.1.13 of [RFC6819] is one example of the use of a cut-and-paste attack.

A concern with returning the mitigation information as new individual authorization response parameters whose values are not cryptographically bound together is that cut-and-paste attacks against their values will not be detected. A security analysis has not been done of the effects of the new attacks that the use of cut-and-paste against these new values will enable.

To prevent replay of the state in another browser instance by an attacker, the state value MUST be tied to the browser instance in a way that cannot be forged by an attacker. Section 4 of [I-D.bradley-oauth-jwt-encoded-state] provides several examples of how a client can accomplish this.

In the replay attack, the attacker can set cookies in the browser. Using an unsigned cookie to bind state to the browser is not sufficient.

8. IANA Considerations

8.1. OAuth Parameters Registration

This specification registers the following parameters in the IANA "OAuth Parameters" registry [IANA.OAuth.Parameters] established by RFC 6749 [RFC6749].

8.1.1. Registry Contents

- o Parameter name: "client_id"
- o Parameter usage location: Authorization Response
- o Change controller: IESG
- o Specification document(s): Section 3.1 of [[this specification]]
- o Related information: None

- o Parameter name: "iss"
- o Parameter usage location: Authorization Response
- o Change controller: IESG
- o Specification document(s): Section 3.1 of [[this specification]]
- o Related information: None

- o Parameter name: "state"
- o Parameter usage location: Token Request
- o Change controller: IESG
- o Specification document(s): Section 5 of [[this specification]]
- o Related information: None

9. References

9.1. Normative References

- [IANA.OAuth.Parameters] IANA, "OAuth Parameters", <<http://www.iana.org/assignments/oauth-parameters>>.
- [JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <<http://tools.ietf.org/html/rfc7519>>.
- [OAuth.Discovery] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Discovery", draft-ietf-oauth-discovery-02 (work in progress), March 2016, <<http://tools.ietf.org/html/draft-ietf-oauth-discovery-02>>.

[OAuth.Responses]

de Medeiros, B., Ed., Scurtescu, M., Tarjan, P., and M. Jones, "OAuth 2.0 Multiple Response Type Encoding Practices", February 2014, <http://openid.net/specs/oauth-v2-multiple-response-types-1_0.html>.

[OpenID.Core]

Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", November 2014, <http://openid.net/specs/openid-connect-core-1_0.html>.

[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>>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <<http://www.rfc-editor.org/info/rfc3629>>.

[RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, DOI 10.17487/RFC5785, April 2010, <<http://www.rfc-editor.org/info/rfc5785>>.

[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>>.

[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>>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <<http://www.rfc-editor.org/info/rfc7159>>.

[RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<http://www.rfc-editor.org/info/rfc7662>>.

9.2. Informative References

[arXiv.1508.04324v2]

Mladenov, V., Mainka, C., and J. Schwenk, "On the security of modern Single Sign-On Protocols: Second-Order Vulnerabilities in OpenID Connect", arXiv 1508.04324v2, January 2016, <<http://arxiv.org/abs/1508.04324v2>>.

[arXiv.1601.01229v2]

Fett, D., Kuesters, R., and G. Schmitz, "A Comprehensive Formal Security Analysis of OAuth 2.0", arXiv 1601.01229v2, January 2016, <<http://arxiv.org/abs/1601.01229v2>>.

[I-D.bradley-oauth-jwt-encoded-state]

Bradley, J., Lodderstedt, T., and H. Zandbelt, "Encoding claims in the OAuth 2 state parameter using a JWT", draft-bradley-oauth-jwt-encoded-state-05 (work in progress), December 2015.

[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, <<http://www.rfc-editor.org/info/rfc7591>>.

Appendix A. Implementation Notes

The authorization server can compare the two state values either by recording the complete state value between the authorization request and the token request, possibly in the same data structure in which the authorization code issued was recorded, or by recording only a cryptographic hash of the state value, possibly resulting in substantial size savings.

Appendix B. Acknowledgements

Alfred Albrecht, John Bradley, Brian Campbell, Joerg Connotte, William Denniss, Sebastian Ebling, Florian Feldmann, Daniel Fett, Roland Hedberg, Phil Hunt, Ralf Kuesters, Torsten Lodderstedt, Christian Mainka, Vladislav Mladenov, Anthony Nadalin, Justin Richer, Nat Sakimura, Antonio Sanso, Guido Schmitz, Joerg Schwenk, Hannes Tschofenig, and Hans Zandbelt all contributed to the discussions that led to the creation of this specification.

This specification is partially based on the OpenID Connect Core 1.0 specification, which was produced by the OpenID Connect working group of the OpenID Foundation.

Appendix C. Open Issues

- o We need to do a security analysis of the cut-and-paste attacks that may be enabled when mitigation information is returned to the client using individual authorization response parameters.

Appendix D. Document History

[[to be removed by the RFC Editor before publication as an RFC]]

-01

- o Changed terms "issuer URL" and "configuration information location" to "issuer identifier" so that consistent terminology is used for this.

-00

- o Created the initial working group draft from draft-jones-oauth-mix-up-mitigation-01 with no normative changes and adding Nat Sakimura as an editor.

Authors' Addresses

Michael B. Jones
Microsoft

Email: mbj@microsoft.com
URI: <http://self-issued.info/>

John Bradley
Ping Identity

Email: ve7jtb@ve7jtb.com
URI: <http://www.thread-safe.com/>

Nat Sakimura
Nomura Research Institute, Ltd.

Email: n-sakimura@nri.co.jp
URI: <http://nat.sakimura.org/>

OAuth Working Group
Internet-Draft
Updates: 6749 (if approved)
Intended status: Best Current Practice
Expires: December 11, 2017

W. Denniss
Google
J. Bradley
Ping Identity
June 9, 2017

OAuth 2.0 for Native Apps
draft-ietf-oauth-native-apps-12

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 December 11, 2017.

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 (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

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

Table of Contents

1. Introduction	2
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 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	11
8.5. Client Authentication	12
8.6. Client Impersonation	12
8.7. Fake External User-Agent	13
8.8. Malicious External User-Agent	13
8.9. Cross-App Request Forgery Protections	14
8.10. Authorization Server Mix-Up Mitigation	14
8.11. Non-Browser External User-Agents	14
8.12. Embedded User-Agents	14
9. IANA Considerations	15
10. References	16
10.1. Normative References	16
10.2. Informative References	16
Appendix A. Server Support Checklist	17
Appendix B. Operating System Specific Implementation Details	17
B.1. iOS Implementation Details	18
B.2. Android Implementation Details	18
B.3. Windows Implementation Details	19
B.4. macOS Implementation Details	19
B.5. Linux Implementation Details	20
Appendix C. Acknowledgements	20
Authors' Addresses	20

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, 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, 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 [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 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 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" In this document, "app" means a "native app" unless further specified.

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

"OAuth" In this document, OAuth refers to 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 as a browser), such that the app cannot access the cookie storage, nor inspect or modify 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 modifying the 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 retains the full security properties and authentication state of the browser. Has different platform-specific product names, such as SFSafariViewController on iOS, and Custom Tabs on Android.

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

"claimed HTTPS URI" Some platforms allow apps to claim a 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" A private-use URI scheme defined by the app and registered with the operating system. URI requests to such schemes trigger the app which registered it to be launched to handle the request.

"web-view" A web browser UI (user interface) 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 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 8.12 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 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

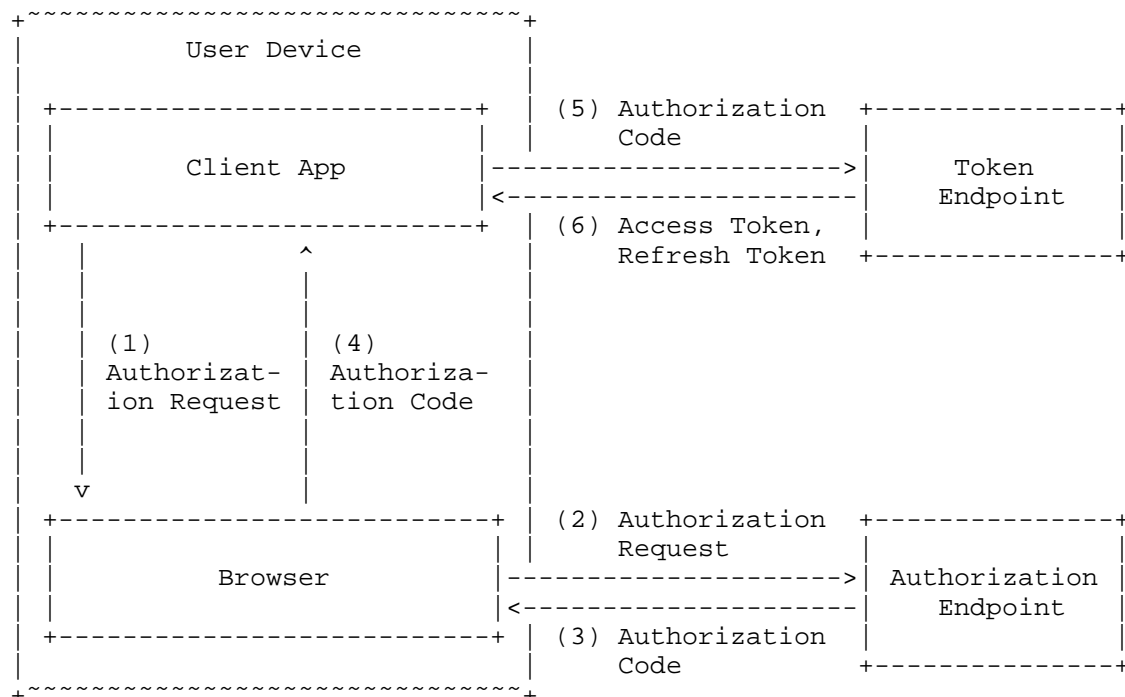


Figure 1: Native App Authorization via External User-agent

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

- (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 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. Re-using 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 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.

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, but 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 processing authorization requests capable of processing the request and redirect URIs in the same way 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 following three redirect URI options 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 custom 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 custom 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.

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

Following the requirements of [RFC3986] Section 3.2, 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:

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

When the authentication server completes the request, it redirects to the client's redirection URI as it would normally. As the redirection URI uses a custom scheme it results in the operating system launching the native app, passing in the URI as a launch parameter. The native app then processes the authorization response like normal.

7.2. Claimed HTTPS URI Redirection

Some operating systems allow apps to claim HTTPS scheme [RFC7230] URIs in 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 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://127.0.0.1:50719/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 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 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 redirects are less susceptible to URI interception due to the presence of the URI authority, but they are still public clients and the URI is sent using the operating system's URI dispatch handler 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, a hash of which it passes in the initial authorization request, and which it must present in full 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.

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 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 [RFC7636] (which is a required in Section 8.1), 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 authorization code grant flow - which can issue refresh tokens - the more practical option for native app authorizations that require refreshing.

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 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, to avoid interference by other network actors.

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. 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

Native apps, except when using a mechanism like Dynamic Client Registration [RFC7591] to provision per-instance secrets, are classified as public clients, as defined by Section 2.1 of OAuth 2.0 [RFC6749] and 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, with the exception of 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 reverse domain name based schemes. At a minimum, any 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 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. 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 redirects MAY be accepted by authorization servers as identity proof. Some operating systems may offer alternative platform-specific identity features which MAY be accepted, as appropriate.

8.7. Fake External User-Agent

The native app which 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.

The advantage when all good actors are using external user-agents is that it is possible for security experts to detect bad actors, as anyone faking an external user-agent is provably bad. 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 malicious apps are detected, it may be possible to use this knowledge to blacklist the apps signatures 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-Agent

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.

Many operating systems mitigate this issue by requiring an explicit user action to change the default handler for HTTP URIs. This attack is not confined to OAuth for Native Apps, a malicious app configured

in this way would present a general and ongoing risk to the user beyond OAuth usage.

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 using inter-app URI communication, 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 a 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 requirements of Section 8.4 that authorization servers reject requests with URIs that don't match what was registered are 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

OAuth 2.0 [RFC6749] Section 9 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: 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 which is often considered an inferior user experience.

9. IANA Considerations

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

This document has no 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, <<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>>.
- [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, <<http://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, <<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>>.
- [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, <<http://www.rfc-editor.org/info/rfc7591>>.
- [AppAuth] Denniss, W., Wright, S., McGinniss, I., Ravikumar, R., and others, "AppAuth", May 22, <<https://appauth.io>>.

[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 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 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 8.5.
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. Operating System 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 operating system specific implementation details of the best practice.

The implementation details herein are considered accurate at the time of publishing but will likely change over time. It is hoped that such change won't invalidate the generic principles in the rest of

the document, and 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 which implements the in-app browser tab pattern. Safari can be used to handle requests on old versions of iOS without `SFSafariViewController`.

To receive the authorization response, both private-use URI scheme redirects (referred to as Custom URL Schemes) 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 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 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 may include the "." character, making short reverse domain name based schemes (as recommended in Section 7.1) possible.

UWP apps can alternatively use the Web Authentication Broker API in SSO (Single Sign-on) 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 sign-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. Personalisations 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 SID, which can be found in the app's registration information. While Windows enforces the URI authority on such redirects, ensuring 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 similar 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, to prevent other apps binding to the same socket.

Appendix C. Acknowledgements

The author would like to acknowledge the work of Marius Scurtescu, and Ben Wiley Sittler whose design for using private-use URI schemes in native 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
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>

OAuth
Internet-Draft
Intended status: Informational
Expires: January 9, 2017

P. Hunt, Ed.
Oracle Corporation
J. Richer

W. Mills

P. Mishra
Oracle Corporation
H. Tschofenig
ARM Limited
July 8, 2016

OAuth 2.0 Proof-of-Possession (PoP) Security Architecture
draft-ietf-oauth-pop-architecture-08.txt

Abstract

The OAuth 2.0 bearer token specification, as defined in RFC 6750, allows any party in possession of a bearer token (a "bearer") to get access to the associated resources (without demonstrating possession of a cryptographic key). To prevent misuse, bearer tokens must be protected from disclosure in transit and at rest.

Some scenarios demand additional security protection whereby a client needs to demonstrate possession of cryptographic keying material when accessing a protected resource. This document motivates the development of the OAuth 2.0 proof-of-possession security mechanism.

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 January 9, 2017.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
2. Terminology	3
3. Use Cases	3
3.1. Preventing Access Token Re-Use by the Resource Server . .	4
3.2. TLS and DTLS Channel Binding Support	4
3.3. Access to a Non-TLS Protected Resource	4
3.4. Offering Application Layer End-to-End Security	5
4. Security and Privacy Threats	5
5. Requirements	6
6. Threat Mitigation	10
6.1. Confidentiality Protection	11
6.2. Sender Constraint	11
6.3. Key Confirmation	12
6.4. Summary	13
7. Architecture	14
7.1. Client and Authorization Server Interaction	15
7.1.1. Symmetric Keys	15
7.1.2. Asymmetric Keys	16
7.2. Client and Resource Server Interaction	17
7.3. Resource and Authorization Server Interaction (Token Introspection)	18
8. Security Considerations	19
9. IANA Considerations	19
10. Acknowledgments	19
11. References	20
11.1. Normative References	20
11.2. Informative References	21
Authors' Addresses	22

1. Introduction

The OAuth 2.0 protocol family ([RFC6749], [RFC6750], and [RFC6819]) offer a single token type known as the "bearer" token to access protected resources. RFC 6750 [RFC6750] specifies the bearer token mechanism and defines it as follows:

"A security token with the property that any party in possession of the token (a "bearer") can use the token in any way that any other party in possession of it can. Using a bearer token does not require a bearer to prove possession of cryptographic key material."

The bearer token meets the security needs of a number of use cases the OAuth 2.0 protocol had originally been designed for. There are, however, other scenarios that require stronger security properties and ask for active participation of the OAuth client in form of cryptographic computations when presenting an access token to a resource server.

This document outlines additional use cases requiring stronger security protection in Section 3, identifies threats in Section 4, proposes different ways to mitigate those threats in Section 6, outlines an architecture for a solution that builds on top of the existing OAuth 2.0 framework in Section 7, and concludes with a requirements list in Section 5.

2. Terminology

The key words 'MUST', 'MUST NOT', 'REQUIRED', 'SHALL', 'SHALL NOT', 'SHOULD', 'SHOULD NOT', 'RECOMMENDED', 'MAY', and 'OPTIONAL' in this specification are to be interpreted as described in [RFC2119], with the important qualification that, unless otherwise stated, these terms apply to the design of the protocol, not its implementation or application.

3. Use Cases

The main use case that motivates improvement upon "bearer" token security is the desire of resource servers to obtain additional assurance that the client is indeed authorized to present an access token. The expectation is that the use of additional credentials (symmetric or asymmetric keying material) will encourage developers to take additional precautions when transferring and storing access token in combination with these credentials.

Additional use cases listed below provide further requirements for the solution development. Note that a single solution does not necessarily need to offer support for all use cases.

3.1. Preventing Access Token Re-Use by the Resource Server

In a scenario where a resource server receives a valid access token, the resource server then re-uses it with other resource server. The reason for re-use may be malicious or may well be legitimate. In a legitimate case, the intent is to support chaining of computations whereby a resource server needs to consult other third party resource servers to complete a requested operation. In both cases it may be assumed that the scope and audience of the access token is sufficiently defined that to allow such a re-use. For example, imagine a case where a company operates email services as well as picture sharing services and that company had decided to issue access tokens with a scope and audience that allows access to both services.

With this use case the desire is to prevent such access token re-use. This also implies that the legitimate use cases require additional enhancements for request chaining.

3.2. TLS and DTLS Channel Binding Support

In this use case we consider the scenario where an OAuth 2.0 request to a protected resource is secured using TLS or DTLS (see [RFC4347]), but the client and the resource server demand that the underlying TLS/DTLS exchange is bound to additional application layer security to prevent cases where the TLS/DTLS connection is terminated at a TLS/DTLS intermediary, which splits the TLS/DTLS connection into two separate connections.

In this use case additional information should be conveyed to the resource server to ensure that no entity entity has tampered with the TLS/DTLS connection.

3.3. Access to a Non-TLS Protected Resource

This use case is for a web client that needs to access a resource that makes data available (such as videos) without offering integrity and confidentiality protection using TLS. Still, the initial resource request using OAuth, which includes the access token, must be protected against various threats (e.g., token replay, token modification).

While it is possible to utilize bearer tokens in this scenario with TLS protection when the request to the protected resource is made, as described in [RFC6750], there may be the desire to avoid using TLS

between the client and the resource server at all. In such a case the bearer token approach is not possible since it relies on TLS for ensuring integrity and confidentiality protection of the access token exchange since otherwise replay attacks are possible: First, an eavesdropper may steal an access token and present it at a different resource server. Second, an eavesdropper may steal an access token and replay it against the same resource server at a later point in time. In both cases, if the attack is successful, the adversary gets access to the resource owners data or may perform an operation selected by the adversary (e.g., sending a message). Note that the adversary may obtain the access token (if the recommendations in [RFC6749] and [RFC6750] are not followed) using a number of ways, including eavesdropping the communication on the wireless link.

Consequently, the important assumption in this use case is that a resource server does not have TLS support and the security solution should work in such a scenario. Furthermore, it may not be necessary to provide authentication of the resource server towards the client.

3.4. Offering Application Layer End-to-End Security

In Web deployments resource servers are often placed behind load balancers, which are deployed by the same organization that operates the resource servers. These load balancers may terminate the TLS connection setup and HTTP traffic is transmitted without TLS protection from the load balancer to the resource server. With application layer security in addition to the underlying TLS security it is possible to allow application servers to perform cryptographic verification on an end-to-end basis.

The key aspect in this use case is therefore to offer end-to-end security in the presence of load balancers via application layer security. Enterprise networks also deploy proxies that inspect traffic and thereby break TLS.

4. Security and Privacy Threats

The following list presents several common threats against protocols utilizing some form of token. This list of threats is based on NIST Special Publication 800-63 [NIST800-63]. We exclude a discussion of threats related to any form of identity proofing and authentication of the resource owner to the authorization server since these procedures are not part of the OAuth 2.0 protocol specification itself.

Token manufacture/modification:

An attacker may generate a bogus token or modify the token content (such as authentication or attribute statements) of an existing token, causing resource server to grant inappropriate access to the client. For example, an attacker may modify the token to extend the validity period. A client, which MAY be a normal client or MAY be assumed to be constrained (see [RFC7252]), may modify the token to have access to information that they should not be able to view.

Token disclosure:

Tokens may contain personal data, such as real name, age or birthday, payment information, etc.

Token redirect:

An attacker uses the token generated for consumption by the resource server to obtain access to another resource server.

Token reuse:

An attacker attempts to use a token that has already been used once with a resource server. The attacker may be an eavesdropper who observes the communication exchange or, worse, one of the communication end points. A client may, for example, leak access tokens because it cannot keep secrets confidential. A client may also reuse access tokens for some other resource servers. Finally, a resource server may use a token it had obtained from a client and use it with another resource server that the client interacts with. A resource server, offering relatively unimportant application services, may attempt to use an access token obtained from a client to access a high-value service, such as a payment service, on behalf of the client using the same access token.

Token repudiation:

Token repudiation refers to a property whereby a resource server is given an assurance that the authorization server cannot deny to have created a token for the client.

5. Requirements

RFC 4962 [RFC4962] gives useful guidelines for designers of authentication and key management protocols. While RFC 4962 was written with the AAA framework used for network access authentication in mind the offered suggestions are useful for the design of other key management systems as well. The following requirements list

applies OAuth 2.0 terminology to the requirements outlined in RFC 4962.

These requirements include

Cryptographic Algorithm Independent:

The key management protocol MUST be cryptographic algorithm independent.

Strong, fresh session keys:

Session keys MUST be strong and fresh. Each session deserves an independent session key, i.e., one that is generated specifically for the intended use. In context of OAuth this means that keying material is created in such a way that can only be used by the combination of a client instance, protected resource, and authorization scope.

Limit Key Scope:

Following the principle of least privilege, parties MUST NOT have access to keying material that is not needed to perform their role. Any protocol that is used to establish session keys MUST specify the scope for session keys, clearly identifying the parties to whom the session key is available.

Replay Detection Mechanism:

The key management protocol exchanges MUST be replay protected. Replay protection allows a protocol message recipient to discard any message that was recorded during a previous legitimate dialogue and presented as though it belonged to the current dialogue.

Authenticate All Parties:

Each party in the key management protocol MUST be authenticated to the other parties with whom they communicate. Authentication mechanisms MUST maintain the confidentiality of any secret values used in the authentication process. Secrets MUST NOT be sent to another party without confidentiality protection.

Authorization:

Client and resource server authorization MUST be performed. These entities MUST demonstrate possession of the appropriate keying material, without disclosing it. Authorization is REQUIRED

whenever a client interacts with an authorization server.
Authorization checking prevents an elevation of privilege attack.

Keying Material Confidentiality and Integrity:

While preserving algorithm independence, confidentiality and integrity of all keying material MUST be maintained.

Confirm Cryptographic Algorithm Selection:

The selection of the "best" cryptographic algorithms SHOULD be securely confirmed. The mechanism SHOULD detect attempted roll-back attacks.

Uniquely Named Keys:

Key management proposals require a robust key naming scheme, particularly where key caching is supported. The key name provides a way to refer to a key in a protocol so that it is clear to all parties which key is being referenced. Objects that cannot be named cannot be managed. All keys MUST be uniquely named, and the key name MUST NOT directly or indirectly disclose the keying material.

Prevent the Domino Effect:

Compromise of a single client MUST NOT compromise keying material held by any other client within the system, including session keys and long-term keys. Likewise, compromise of a single resource server MUST NOT compromise keying material held by any other Resource Server within the system. In the context of a key hierarchy, this means that the compromise of one node in the key hierarchy must not disclose the information necessary to compromise other branches in the key hierarchy. Obviously, the compromise of the root of the key hierarchy will compromise all of the keys; however, a compromise in one branch MUST NOT result in the compromise of other branches. There are many implications of this requirement; however, two implications deserve highlighting. First, the scope of the keying material must be defined and understood by all parties that communicate with a party that holds that keying material. Second, a party that holds keying material in a key hierarchy must not share that keying material with parties that are associated with other branches in the key hierarchy.

Bind Key to its Context:

Keying material MUST be bound to the appropriate context. The context includes the following.

- * The manner in which the keying material is expected to be used.
- * The other parties that are expected to have access to the keying material.
- * The expected lifetime of the keying material. Lifetime of a child key SHOULD NOT be greater than the lifetime of its parent in the key hierarchy.

Any party with legitimate access to keying material can determine its context. In addition, the protocol MUST ensure that all parties with legitimate access to keying material have the same context for the keying material. This requires that the parties are properly identified and authenticated, so that all of the parties that have access to the keying material can be determined. The context will include the client and the resource server identities in more than one form.

Authorization Restriction:

If client authorization is restricted, then the client SHOULD be made aware of the restriction.

Client Identity Confidentiality:

A client has identity confidentiality when any party other than the resource server and the authorization server cannot sufficiently identify the client within the anonymity set. In comparison to anonymity and pseudonymity, identity confidentiality is concerned with eavesdroppers and intermediaries. A key management protocol SHOULD provide this property.

Resource Owner Identity Confidentiality:

Resource servers SHOULD be prevented from knowing the real or pseudonymous identity of the resource owner, since the authorization server is the only entity involved in verifying the resource owner's identity.

Collusion:

Resource servers that collude can be prevented from using information related to the resource owner to track the individual. That is, two different resource servers can be prevented from determining that the same resource owner has authenticated to both

of them. Authorization servers MUST bind different keying material to access tokens used for resource servers from different origins (or similar concepts in the app world).

AS-to-RS Relationship Anonymity:

For solutions using asymmetric key cryptography the client MAY conceal information about the resource server it wants to interact with. The authorization server MAY reject such an attempt since it may not be able to enforce access control decisions.

Channel Binding:

A solution MUST enable support for channel bindings. The concept of channel binding, as defined in [RFC5056], allows applications to establish that the two end-points of a secure channel at one network layer are the same as at a higher layer by binding authentication at the higher layer to the channel at the lower layer.

There are performance concerns with the use of asymmetric cryptography. Although symmetric key cryptography offers better performance asymmetric cryptography offers additional security properties. A solution MUST therefore offer the capability to support both symmetric as well as asymmetric keys.

There are threats that relate to the experience of the software developer as well as operational practices. Verifying the servers identity in TLS is discussed at length in [RFC6125].

A number of the threats listed in Section 4 demand protection of the access token content and a standardized solution, for example, in the form of a JSON-based format, is available with the JWT [RFC7519].

6. Threat Mitigation

A large range of threats can be mitigated by protecting the content of the token, for example using a digital signature or a keyed message digest. Alternatively, the content of the token could be passed by reference rather than by value (requiring a separate message exchange to resolve the reference to the token content).

To simplify discussion in the following example we assume that the token itself cannot be modified by the client, either due to cryptographic protection (such as signature or encryption) or use of a reference value with sufficient entropy and associated secure lookup. The token remains opaque to the client. These are characteristics shared with bearer tokens and more information on

best practices can be found in [RFC6819] and in the security considerations section of [RFC6750].

To deal with token redirect it is important for the authorization server to include the identifier of the intended recipient - the resource server. A resource server must not be allowed to accept access tokens that are not meant for its consumption.

To provide protection against token disclosure two approaches are possible, namely (a) not to include sensitive information inside the token or (b) to ensure confidentiality protection. The latter approach requires at least the communication interaction between the client and the authorization server as well as the interaction between the client and the resource server to experience confidentiality protection. As an example, TLS with a ciphersuite that offers confidentiality protection has to be applied as per [RFC7525]. Encrypting the token content itself is another alternative. In our scenario the authorization server would, for example, encrypt the token content with a symmetric key shared with the resource server.

To deal with token reuse more choices are available.

6.1. Confidentiality Protection

In this approach confidentiality protection of the exchange is provided on the communication interfaces between the client and the resource server, and between the client and the authorization server. No eavesdropper on the wire is able to observe the token exchange. Consequently, a replay by a third party is not possible. An authorization server wants to ensure that it only hands out tokens to clients it has authenticated first and who are authorized. For this purpose, authentication of the client to the authorization server will be a requirement to ensure adequate protection against a range of attacks. This is, however, true for the description in Section 6.2 and Section 6.3 as well. Furthermore, the client has to make sure it does not distribute (or leak) the access token to entities other than the intended the resource server. For that purpose the client will have to authenticate the resource server before transmitting the access token.

6.2. Sender Constraint

Instead of providing confidentiality protection, the authorization server could also put the identifier of the client into the protected token with the following semantic: 'This token is only valid when presented by a client with the following identifier.' When the access token is then presented to the resource server how does it

know that it was provided by the client? It has to authenticate the client! There are many choices for authenticating the client to the resource server, for example by using client certificates in TLS [RFC5246], or pre-shared secrets within TLS [RFC4279]. The choice of the preferred authentication mechanism and credential type may depend on a number of factors, including

- o security properties
- o available infrastructure
- o library support
- o credential cost (financial)
- o performance
- o integration into the existing IT infrastructure
- o operational overhead for configuration and distribution of credentials

This long list hints to the challenge of selecting at least one mandatory-to-implement client authentication mechanism.

6.3. Key Confirmation

A variation of the mechanism of sender authentication, described in Section 6.2, is to replace authentication with the proof-of-possession of a specific (session) key, i.e., key confirmation. In this model the resource server would not authenticate the client itself but would rather verify whether the client knows the session key associated with a specific access token. Examples of this approach can be found with the OAuth 1.0 MAC token [RFC5849], and Kerberos [RFC4120] when utilizing the AP_REQ/AP_REP exchange (see also [I-D.hardjono-oauth-kerberos] for a comparison between Kerberos and OAuth).

To illustrate key confirmation, the first example is borrowed from Kerberos and use symmetric key cryptography. Assume that the authorization server shares a long-term secret with the resource server, called $K(\text{Authorization Server-Resource Server})$. This secret would be established between them out-of-band. When the client requests an access token the authorization server creates a fresh and unique session key K_s and places it into the token encrypted with the long term key $K(\text{Authorization Server-Resource Server})$. Additionally, the authorization server attaches K_s to the response message to the client (in addition to the access token itself) over a

confidentiality protected channel. When the client sends a request to the resource server it has to use K_s to compute a keyed message digest for the request (in whatever form or whatever layer). The resource server, when receiving the message, retrieves the access token, verifies it and extracts $K(\text{Authorization Server-Resource Server})$ to obtain K_s . This key K_s is then used to verify the keyed message digest of the request message.

Note that in this example one could imagine that the mechanism to protect the token itself is based on a symmetric key based mechanism to avoid any form of public key infrastructure but this aspect is not further elaborated in the scenario.

A similar mechanism can also be designed using asymmetric cryptography. When the client requests an access token the authorization server creates an ephemeral public / privacy key pair (PK/SK) and places the public key PK into the protected token. When the authorization server returns the access token to the client it also provides the PK/SK key pair over a confidentiality protected channel. When the client sends a request to the resource server it has to use the privacy key SK to sign the request. The resource server, when receiving the message, retrieves the access token, verifies it and extracts the public key PK. It uses this ephemeral public key to verify the attached signature.

6.4. Summary

As a high level message, there are various ways the threats can be mitigated. While the details of each solution are somewhat different, they all accomplish the goal of mitigating the threats.

The three approaches are:

Confidentiality Protection:

The weak point with this approach, which is briefly described in Section 6.1, is that the client has to be careful to whom it discloses the access token. What can be done with the token entirely depends on what rights the token entitles the presenter and what constraints it contains. A token could encode the identifier of the client but there are scenarios where the client is not authenticated to the resource server or where the identifier of the client rather represents an application class rather than a single application instance. As such, it is possible that certain deployments choose a rather liberal approach to security and that everyone who is in possession of the access token is granted access to the data.

Sender Constraint:

The weak point with this approach, which is briefly described in Section 6.2, is to setup the authentication infrastructure such that clients can be authenticated towards resource servers. Additionally, the authorization server must encode the identifier of the client in the token for later verification by the resource server. Depending on the chosen layer for providing client-side authentication there may be additional challenges due to Web server load balancing, lack of API access to identity information, etc.

Key Confirmation:

The weak point with this approach, see Section 6.3, is the increased complexity: a complete key distribution protocol has to be defined.

In all cases above it has to be ensured that the client is able to keep the credentials secret.

7. Architecture

The proof-of-possession security concept assumes that the authorization server acts as a trusted third party that binds keys to access tokens. These keys are then used by the client to demonstrate the possession of the secret to the resource server when accessing the resource. The resource server, when receiving an access token, needs to verify that the key used by the client matches the one included in the access token.

There are slight differences between the use of symmetric keys and asymmetric keys when they are bound to the access token and the subsequent interaction between the client and the authorization server when demonstrating possession of these keys. Figure 1 shows the symmetric key procedure and Figure 2 illustrates how asymmetric keys are used. While symmetric cryptography provides better performance properties the use of asymmetric cryptography allows the client to keep the private key locally and never expose it to any other party.

For example, with the JSON Web Token (JWT) [RFC7519] a standardized format for access tokens is available. The necessary elements to bind symmetric or asymmetric keys to a JWT are described in [I-D.ietf-oauth-proof-of-possession].

Note: The negotiation of cryptographic algorithms between the client and the authorization server is not shown in the examples below and

assumed to be present in a protocol solution to meet the requirements for crypto-agility.

7.1. Client and Authorization Server Interaction

7.1.1. Symmetric Keys

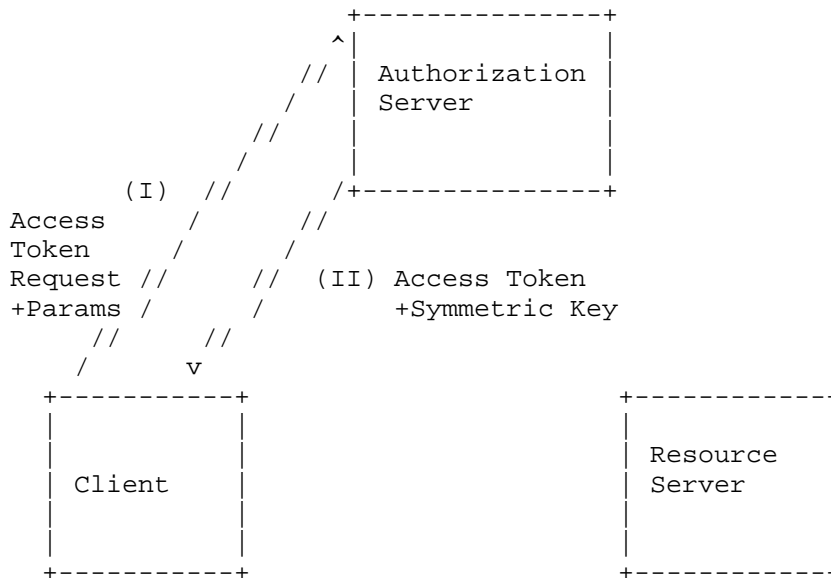


Figure 1: Interaction between the Client and the Authorization Server (Symmetric Keys).

In order to request an access token the client interacts with the authorization server as part of the a normal grant exchange, as shown in Figure 1. However, it needs to include additional information elements for use with the PoP security mechanism, as depicted in message (I). In message (II) the authorization server then returns the requested access token. In addition to the access token itself, the symmetric key is communicated to the client. This symmetric key is a unique and fresh session key with sufficient entropy for the given lifetime. Furthermore, information within the access token ties it to this specific symmetric key.

Note: For this security mechanism to work the client as well as the resource server need to have access to the session key. While the key transport mechanism from the authorization server to the client has been explained in the previous paragraph there are three ways for communicating this session key from the authorization server to the resource server, namely

Embedding the symmetric key inside the access token itself. This requires that the symmetric key is confidentiality protected.

The resource server queries the authorization server for the symmetric key. This is an approach envisioned by the token introspection endpoint [RFC7662].

The authorization server and the resource server both have access to the same back-end database. Smaller, tightly coupled systems might prefer such a deployment strategy.

7.1.2. Asymmetric Keys

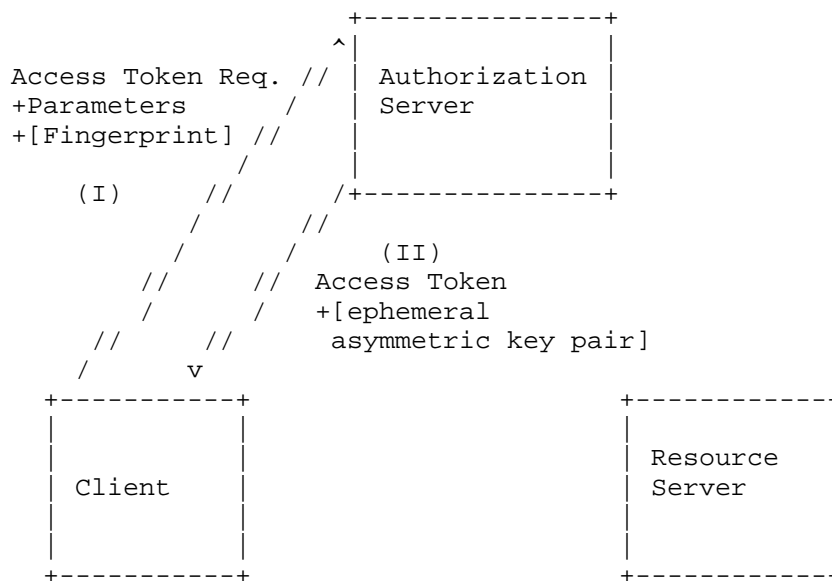


Figure 2: Interaction between the Client and the Authorization Server (Asymmetric Keys).

The use of asymmetric keys is slightly different since the client or the server could be involved in the generation of the ephemeral key pair. This exchange is shown in Figure 1. If the client generates the key pair it either includes a fingerprint of the public key or the public key in the request to the authorization server. The authorization server would include this fingerprint or public key in the confirmation claim inside the access token and thereby bind the asymmetric key pair to the token. If the client did not provide a fingerprint or a public key in the request then the authorization server is asked to create an ephemeral asymmetric key pair, binds the fingerprint of the public key to the access token, and returns the

asymmetric key pair (public and private key) to the client. Note that there is a strong preference for generating the private/public key pair locally at the client rather than at the server.

7.2. Client and Resource Server Interaction

The specification describing the interaction between the client and the authorization server, as shown in Figure 1 and in Figure 2, can be found in [I-D.ietf-oauth-pop-key-distribution].

Once the client has obtained the necessary access token and keying material it can start to interact with the resource server. To demonstrate possession of the key bound to the access token it needs to apply this key to the request by computing a keyed message digest (i.e., a symmetric key-based cryptographic primitive) or a digital signature (i.e., an asymmetric cryptographic computation). When the resource server receives the request it verifies it and decides whether access to the protected resource can be granted. This exchange is shown in Figure 3.

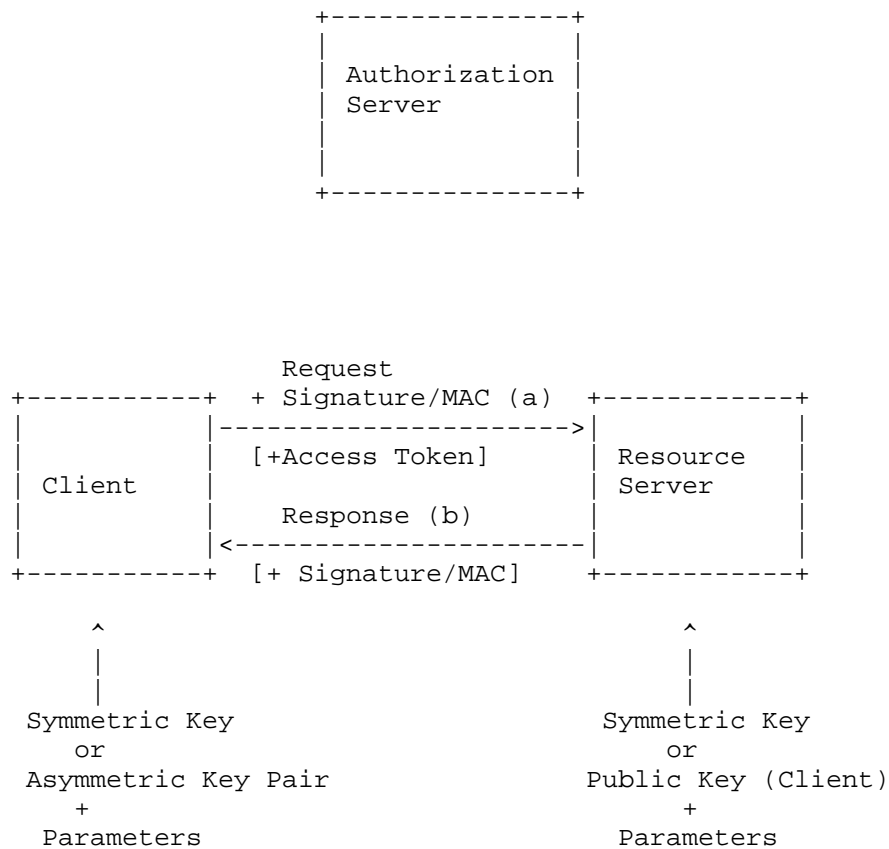


Figure 3: Client Demonstrates PoP.

The specification describing the ability to sign the HTTP request from the client to the resource server can be found in [I-D.ietf-oauth-signed-http-request].

7.3. Resource and Authorization Server Interaction (Token Introspection)

So far the examples talked about access tokens that are passed by value and allow the resource server to make authorization decisions immediately after verifying the request from the client. In some deployments a real-time interaction between the authorization server and the resource server is envisioned that lowers the need to pass self-contained access tokens around. In that case the access token merely serves as a handle or a reference to state stored at the authorization server. As a consequence, the resource server cannot autonomously make an authorization decision when receiving a request

from a client but has to consult the authorization server. This can, for example, be done using the token introspection endpoint (see [RFC7662]). Figure 4 shows the protocol interaction graphically. Despite the additional token exchange previous descriptions about associating symmetric and asymmetric keys to the access token are still applicable to this scenario.

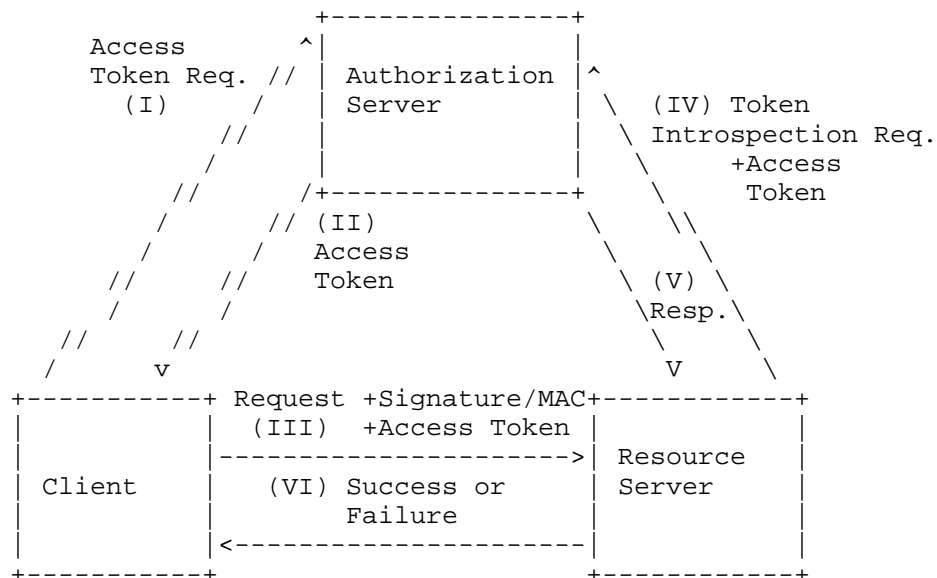


Figure 4: Token Introspection and Access Token Handles.

8. Security Considerations

The purpose of this document is to provide use cases, requirements, and motivation for developing an OAuth security solution extending Bearer Tokens. As such, this document is only about security.

9. IANA Considerations

This document does not require actions by IANA.

10. Acknowledgments

This document is the result of conference calls late 2012/early 2013 and in design team conference calls February 2013 of the IETF OAuth working group. The following persons (in addition to the OAuth WG chairs, Hannes Tschofenig, and Derek Atkins) provided their input during these calls: Bill Mills, Justin Richer, Phil Hunt, Prateek Mishra, Mike Jones, George Fletcher, Leif Johansson, Lucy Lynch, John

Bradley, Tony Nadalin, Klaas Wierenga, Thomas Hardjono, Brian Campbell

In the appendix of this document we reuse content from [RFC4962] and the authors would like thank Russ Housely and Bernard Aboba for their work on RFC 4962.

We would like to thank Reddy Tirumaleswar for his review.

11. References

11.1. Normative References

- [I-D.ietf-oauth-pop-key-distribution]
Bradley, J., Hunt, P., Jones, M., and H. Tschofenig, "OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution", draft-ietf-oauth-pop-key-distribution-02 (work in progress), October 2015.
- [I-D.ietf-oauth-proof-of-possession]
Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", draft-ietf-oauth-proof-of-possession-11 (work in progress), December 2015.
- [I-D.ietf-oauth-signed-http-request]
Richer, J., Bradley, J., and H. Tschofenig, "A Method for Signing HTTP Requests for OAuth", draft-ietf-oauth-signed-http-request-02 (work in progress), February 2016.
- [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>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<http://www.rfc-editor.org/info/rfc5246>>.
- [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>>.
- [RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <<http://www.rfc-editor.org/info/rfc7519>>.

- [RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <<http://www.rfc-editor.org/info/rfc7525>>.
- [RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<http://www.rfc-editor.org/info/rfc7662>>.

11.2. Informative References

- [I-D.hardjono-oauth-kerberos]
Hardjono, T., "OAuth 2.0 support for the Kerberos V5 Authentication Protocol", draft-hardjono-oauth-kerberos-01 (work in progress), December 2010.
- [NIST800-63]
Burr, W., Dodson, D., Perlner, R., Polk, T., Gupta, S., and E. Nabbus, "NIST Special Publication 800-63-1, INFORMATION SECURITY", December 2008.
- [RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, DOI 10.17487/RFC4120, July 2005, <<http://www.rfc-editor.org/info/rfc4120>>.
- [RFC4279] Eronen, P., Ed. and H. Tschofenig, Ed., "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", RFC 4279, DOI 10.17487/RFC4279, December 2005, <<http://www.rfc-editor.org/info/rfc4279>>.
- [RFC4347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security", RFC 4347, DOI 10.17487/RFC4347, April 2006, <<http://www.rfc-editor.org/info/rfc4347>>.
- [RFC4962] Housley, R. and B. Aboba, "Guidance for Authentication, Authorization, and Accounting (AAA) Key Management", BCP 132, RFC 4962, DOI 10.17487/RFC4962, July 2007, <<http://www.rfc-editor.org/info/rfc4962>>.
- [RFC5056] Williams, N., "On the Use of Channel Bindings to Secure Channels", RFC 5056, DOI 10.17487/RFC5056, November 2007, <<http://www.rfc-editor.org/info/rfc5056>>.
- [RFC5849] Hammer-Lahav, E., Ed., "The OAuth 1.0 Protocol", RFC 5849, DOI 10.17487/RFC5849, April 2010, <<http://www.rfc-editor.org/info/rfc5849>>.

- [RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <<http://www.rfc-editor.org/info/rfc6125>>.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <<http://www.rfc-editor.org/info/rfc6750>>.
- [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>>.
- [RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <<http://www.rfc-editor.org/info/rfc7252>>.

Authors' Addresses

Phil Hunt (editor)
Oracle Corporation

Email: phil.hunt@yahoo.com

Justin Richer

Email: ietf@justin.richer.org

William Mills

Email: wmills@yahoo-inc.com

Prateek Mishra
Oracle Corporation

Email: prateek.mishra@oracle.com

Hannes Tschofenig
ARM Limited
Hall in Tirol 6060
Austria

Email: Hannes.Tschofenig@gmx.net
URI: <http://www.tschofenig.priv.at>

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: February 9, 2017

J. Richer, Ed.

J. Bradley
Ping Identity
H. Tschofenig
ARM Limited
August 08, 2016

A Method for Signing HTTP Requests for OAuth
draft-ietf-oauth-signed-http-request-03

Abstract

This document a method for offering data origin authentication and integrity protection of HTTP requests. To convey the relevant data items in the request a JSON-based encapsulation is used and the JSON Web Signature (JWS) technique is re-used. JWS offers integrity protection using symmetric as well as asymmetric cryptography.

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 February 9, 2017.

Copyright Notice

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

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

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Terminology	3
3. Generating a JSON Object from an HTTP Request	3
3.1. Calculating the query parameter list and hash	4
3.2. Calculating the header list and hash	5
4. Sending the signed object	6
4.1. HTTP Authorization header	6
4.2. HTTP Form body	6
4.3. HTTP Query parameter	7
5. Validating the request	7
5.1. Validating the query parameter list and hash	7
5.2. Validating the header list and hash	8
6. IANA Considerations	9
6.1. The 'pop' OAuth Access Token Type	9
6.2. JSON Web Signature and Encryption Type Values Registration	9
7. Security Considerations	9
7.1. Offering Confidentiality Protection for Access to Protected Resources	9
7.2. Plaintext Storage of Credentials	10
7.3. Entropy of Keys	10
7.4. Denial of Service	10
7.5. Validating the integrity of HTTP message	11
8. Privacy Considerations	12
9. Acknowledgements	12
10. Normative References	12
Authors' Addresses	13

1. Introduction

In order to prove possession of an access token and its associated key, an OAuth 2.0 client needs to compute some cryptographic function and present the results to the protected resource as a signature. The protected resource then needs to verify the signature and compare that to the expected keys associated with the access token. This is in addition to the normal token protections provided by a bearer token [RFC6750] and transport layer security (TLS).

Furthermore, it is desirable to bind the signature to the HTTP request. Ideally, this should be done without replicating the information already present in the HTTP request more than required. However, many HTTP application frameworks insert extra headers, query

parameters, and otherwise manipulate the HTTP request on its way from the web server into the application code itself. It is the goal of this draft to have a signature protection mechanism that is sufficiently robust against such deployment constraints while still providing sufficient security benefits.

The key required for this signature calculation is distributed via mechanisms described in companion documents (see [I-D.ietf-oauth-pop-key-distribution] and [I-D.ietf-oauth-pop-architecture]). The JSON Web Signature (JWS) specification [RFC7515] is used for computing a digital signature (which uses asymmetric cryptography) or a keyed message digest (in case of symmetric cryptography).

The mechanism described in this document assumes that a client is in possession of an access token and associated key. That client then creates a JSON object including the access token, signs the JSON object using JWS, and issues an request to a resource server for access to a protected resource using the signed object as its authorization. The protected resource validates the JWS signature and parses the JSON object to obtain token information.

2. Terminology

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 RFC 2119 [RFC2119].

Other terms such as "client", "authorization server", "access token", and "protected resource" are inherited from OAuth 2.0 [RFC6749].

We use the term 'sign' (or 'signature') to denote both a keyed message digest and a digital signature operation.

3. Generating a JSON Object from an HTTP Request

This specification uses JSON Web Signatures [RFC7515] to protect the access token and, optionally, parts of the request.

This section describes how to generate a JSON [RFC7159] object from the HTTP request. Each value below is included as a member of the JSON object at the top level.

at REQUIRED. The access token value. This string is assumed to have no particular format or structure and remains opaque to the client.

- ts RECOMMENDED. The timestamp. This integer provides replay protection of the signed JSON object. Its value MUST be a number containing an integer value representing number of whole integer seconds from midnight, January 1, 1970 GMT.
- m OPTIONAL. The HTTP Method used to make this request. This MUST be the uppercase HTTP verb as a JSON string.
- u OPTIONAL. The HTTP URL host component as a JSON string. This MAY include the port separated from the host by a colon in host:port format.
- p OPTIONAL. The HTTP URL path component of the request as an HTTP string.
- q OPTIONAL. The hashed HTTP URL query parameter map of the request as a two-part JSON array. The first part of this array is a JSON array listing all query parameters that were used in the calculation of the hash in the order that they were added to the hashed value as described below. The second part of this array is a JSON string containing the Base64URL encoded hash itself, calculated as described below.
- h OPTIONAL. The hashed HTTP request headers as a two-part JSON array. The first part of this array is a JSON array listing all headers that were used in the calculation of the hash in the order that they were added to the hashed value as described below. The second part of this array is a JSON string containing the Base64URL encoded hash itself, calculated as described below.
- b OPTIONAL. The base64URL encoded hash of the HTTP Request body, calculated as the SHA256 of the byte array of the body

All hashes SHALL be calculated using the SHA256 algorithm. [[Note to WG: do we want crypto agility here? If so how do we signal this]]

The JSON object is signed using the algorithm appropriate to the associated access token key, usually communicated as part of key distribution [I-D.ietf-oauth-pop-key-distribution].

3.1. Calculating the query parameter list and hash

To generate the query parameter list and hash, the client creates two data objects: an ordered list of strings to hold the query parameter names and a string buffer to hold the data to be hashed.

The client iterates through all query parameters in whatever order it chooses and for each query parameter it does the following:

1. Adds the name of the query parameter to the end of the list.
2. Percent-encodes the name and value of the parameter as specified in [RFC3986]. Note that if the name and value have already been percent-encoded for transit, they are not re-encoded for this step.
3. Encodes the name and value of the query parameter as "name=value" and appends it to the string buffer separated by the ampersand "&" character.

Repeated parameter names are processed separately with no special handling. Parameters MAY be skipped by the client if they are not required (or desired) to be covered by the signature.

The client then calculates the hash over the resulting string buffer. The list and the hash result are added to a list as the value of the "q" member.

For example, the query parameter set of "b=bar", "a=foo", "c=duck" is concatenated into the string:

```
b=bar&a=foo&c=duck
```

When added to the JSON structure using this process, the results are:

```
"q": [["b", "a", "c"], "u4LgkGUWhP9MsKrEjA4dizI1lDXluDku6ZqCeyuR-JY"]
```

3.2. Calculating the header list and hash

To generate the header list and hash, the client creates two data objects: an ordered list of strings to hold the header names and a string buffer to hold the data to be hashed.

The client iterates through all query parameters in whatever order it chooses and for each query parameter it does the following:

1. Lowercases the header name.
2. Adds the name of the header to the end of the list.
3. Encodes the name and value of the header as "name: value" and appends it to the string buffer separated by a newline "\n" character.

Repeated header names are processed separately with no special handling. Headers MAY be skipped by the client if they are not required (or desired) to be covered by the signature.

The client then calculates the hash over the resulting string buffer. The list and the hash result are added to a list as the value of the "h" member.

For example, the headers "Content-Type: application/json" and "Etag: 742-3u8f34-3r2nv3" are concatenated into the string:

```
content-type: application/json
etag: 742-3u8f34-3r2nv3

"h": [{"content-type", "etag"},
      "bZA981YJBrPlIzOvplbu3e7ueREXXr38vSkxIBYOaxI"]
```

4. Sending the signed object

In order to send the signed object to the protected resource, the client includes it in one of the following three places.

4.1. HTTP Authorization header

The client SHOULD send the signed object to the protected resource in the Authorization header. The value of the signed object in JWS compact form is appended to the Authorization header as a PoP value. This is the preferred method. Note that if this method is used, the Authorization header MUST NOT be included in the protected elements of the signed object.

```
GET /resource/foo
Authorization: PoP eyJ....omitted for brevity...
```

4.2. HTTP Form body

If the client is sending the request as a form-encoded HTTP message with parameters in the body, the client MAY send the signed object as part of that form body. The value of the signed object in JWS compact form is sent as the form parameter `pop_access_token`. Note that if this method is used, the body hash cannot be included in the protected elements of the signed object.

```
POST /resource
Content-type: application/www-form-encoded

pop_access_token=eyJ....omitted for brevity...
```


4.3. HTTP Query parameter

If neither the Authorization header nor the form-encoded body parameter are available to the client, the client MAY send the signed object as a query parameter. The value of the signed object in JWS compact form is sent as the query parameter `pop_access_token`. Note that if this method is used, the `pop_access_token` parameter MUST NOT be included in the protected elements of the signed object.

```
GET /resource?pop_access_token=eyJ....
```

5. Validating the request

Just like with a bearer token [RFC6750], while the access token value included in the signed object is opaque to the client, it MUST be understood by the protected resource in order to fulfill the request. Also like a bearer token, the protected resource traditionally has several methods at its disposal for understanding the access token. It can look up the token locally (such as in a database), it can parse a structured token (such as JWT [RFC7519]), or it can use a service to look up token information (such as introspection [RFC7662]). Whatever method is used to look up token information, the protected resource MUST have access to the key associated with the access token, as this key is required to validate the signature of the incoming request. Validation of the signature is done using normal JWS validation for the signature and key type.

Additionally, in order to trust any of the hashed components of the HTTP request, the protected resource MUST re-create and verify a hash for each component as described below. This process is a mirror of the process used to create the hashes in the first place, with a mind toward the fact that order may have changed and that elements may have been added or deleted. The protected resource MUST similarly compare the replicated values included in various JSON fields with the corresponding actual values from the request. Failure to do so will allow an attacker to modify the underlying request while at the same time having the application layer verify the signature correctly.

5.1. Validating the query parameter list and hash

The client has at its disposal a map that indexes the query parameter names to the values given. The client creates a string buffer for calculating the hash. The client then iterates through the "list" portion of the "p" parameter. For each item in the list (in the order of the list) it does the following:

1. Fetch the value of the parameter from the HTTP request query parameter map. If a parameter is found in the list of signed parameters but not in the map, the validation fails.
2. Percent-encodes the name and value of the parameter as specified in [RFC3986]. Note that if the name and value have already been percent-encoded for transit, they are not re-encoded for this step.
3. Encode the parameter as "name=value" and concatenate it to the end of the string buffer, separated by an ampersand character.

The client calculates the hash of the string buffer and base64url encodes it. The protected resource compares that string to the string passed in as the hash. If the two match, the hash validates, and all named parameters and their values are considered covered by the signature.

There MAY be additional query parameters that are not listed in the list and are therefore not covered by the signature. The client MUST decide whether or not to accept a request with these uncovered parameters.

5.2. Validating the header list and hash

The client has at its disposal a map that indexes the header names to the values given. The client creates a string buffer for calculating the hash. The client then iterates through the "list" portion of the "h" parameter. For each item in the list (in the order of the list) it does the following:

1. Fetch the value of the header from the HTTP request header map. If a header is found in the list of signed parameters but not in the map, the validation fails.
2. Encode the parameter as "name: value" and concatenate it to the end of the string buffer, separated by a newline character.

The client calculates the hash of the string buffer and base64url encodes it. The protected resource compares that string to the string passed in as the hash. If the two match, the hash validates, and all named headers and their values are considered covered by the signature.

There MAY be additional headers that are not listed in the list and are therefore not covered by the signature. The client MUST decide whether or not to accept a request with these uncovered headers.

6. IANA Considerations

6.1. The 'pop' OAuth Access Token Type

Section 11.1 of [RFC6749] defines the OAuth Access Token Type Registry and this document adds another token type to this registry.

Type name: pop

Additional Token Endpoint Response Parameters: (none)

HTTP Authentication Scheme(s): Proof-of-possession access token for use with OAuth 2.0

Change controller: IETF

Specification document(s): [[this document]]

6.2. JSON Web Signature and Encryption Type Values Registration

This specification registers the "pop" type value in the IANA JSON Web Signature and Encryption Type Values registry [RFC7515]:

- o "typ" Header Parameter Value: "pop"
- o Abbreviation for MIME Type: None
- o Change Controller: IETF
- o Specification Document(s): [[this document]]

7. Security Considerations

7.1. Offering Confidentiality Protection for Access to Protected Resources

This specification can be used with and without Transport Layer Security (TLS).

Without TLS this protocol provides a mechanism for verifying the integrity of requests, it provides no confidentiality protection. Consequently, eavesdroppers will have full access to communication content and any further messages exchanged between the client and the resource server. This could be problematic when data is exchanged that requires care, such as personal data.

When TLS is used then confidentiality of the transmission can be ensured between endpoints, including both the request and the

response. The use of TLS in combination with the signed HTTP request mechanism is highly recommended to ensure the confidentiality of the data returned from the protected resource.

7.2. Plaintext Storage of Credentials

The mechanism described in this document works in a similar way to many three-party authentication and key exchange mechanisms. In order to compute the signature over the HTTP request, the client must have access to a key bound to the access token in plaintext form. If an attacker were to gain access to these stored secrets at the client or (in case of symmetric keys) at the resource server they would be able to perform any action on behalf of any client just as if they had stolen a bearer token.

It is therefore paramount to the security of the protocol that the private keys associated with the access tokens are protected from unauthorized access.

7.3. Entropy of Keys

Unless TLS is used between the client and the resource server, eavesdroppers will have full access to requests sent by the client. They will thus be able to mount off-line brute-force attacks to attempt recovery of the session key or private key used to compute the keyed message digest or digital signature, respectively.

This specification assumes that the key used herein has been distributed via other mechanisms, such as [I-D.ietf-oauth-pop-key-distribution]. Hence, it is the responsibility of the authorization server and or the client to be careful when generating fresh and unique keys with sufficient entropy to resist such attacks for at least the length of time that the session keys (and the access tokens) are valid.

For example, if the key bound to the access token is valid for one day, authorization servers must ensure that it is not possible to mount a brute force attack that recovers that key in less than one day. Of course, servers are urged to err on the side of caution, and use the longest key length possible within reason.

7.4. Denial of Service

This specification includes a number of features which may make resource exhaustion attacks against resource servers possible. For example, a resource server may need to process the incoming request, verify the access token, perform signature verification, and might (in certain circumstances) have to consult back-end databases or the

authorization server before granting access to the protected resource. Many of these actions are shared with bearer tokens, but the additional cryptographic overhead of validating the signed request needs to be taken into consideration with deployment of this specification.

An attacker may exploit this to perform a denial of service attack by sending a large number of invalid requests to the server. The computational overhead of verifying the keyed message digest alone is not likely sufficient to mount a denial of service attack. To help combat this, it is RECOMMENDED that the protected resource validate the access token (contained in the "at" member of the signed structure) before performing any cryptographic verification calculations.

7.5. Validating the integrity of HTTP message

This specification provides flexibility for selectively validating the integrity of the HTTP request, including header fields, query parameters, and message bodies. Since all components of the HTTP request are only optionally validated by this method, and even some components may be validated only in part (e.g., some headers but not others) it is up to protected resource developers to verify that any vital parameters in a request are actually covered by the signature. Failure to do so could allow an attacker to inject vital parameters or headers into the request, outside of the protection of the signature.

The application verifying this signature MUST NOT assume that any particular parameter is appropriately covered by the signature unless it is included in the signed structure and the hash is verified. Any applications that are sensitive of header or query parameter order MUST verify the order of the parameters on their own. The application MUST also compare the values in the JSON container with the actual parameters received with the HTTP request (using a direct comparison or a hash calculation, as appropriate). Failure to make this comparison will render the signature mechanism useless for protecting these elements.

The behavior of repeated query parameters or repeated HTTP headers is undefined by this specification. If a header or query parameter is repeated on either the outgoing request from the client or the incoming request to the protected resource, that query parameter or header name MUST NOT be covered by the hash and signature.

This specification records the order in which query parameters and headers are hashed, but it does not guarantee that order is preserved between the client and protected resource. If the order of

parameters or headers are significant to the underlying application, it MUST confirm their order on its own, apart from the signature and HTTP message validation.

8. Privacy Considerations

This specification addresses machine to machine communications and raises no privacy considerations beyond existing OAuth transactions.

9. Acknowledgements

The authors thank the OAuth Working Group for input into this work.

10. Normative References

- [I-D.ietf-oauth-pop-architecture]
Hunt, P., Richer, J., Mills, W., Mishra, P., and H. Tschofenig, "OAuth 2.0 Proof-of-Possession (PoP) Security Architecture", draft-ietf-oauth-pop-architecture-08 (work in progress), July 2016.
- [I-D.ietf-oauth-pop-key-distribution]
Bradley, J., Hunt, P., Jones, M., and H. Tschofenig, "OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution", draft-ietf-oauth-pop-key-distribution-02 (work in progress), October 2015.
- [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>>.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <<http://www.rfc-editor.org/info/rfc6750>>.

- [RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <<http://www.rfc-editor.org/info/rfc7159>>.
- [RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <<http://www.rfc-editor.org/info/rfc7515>>.
- [RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <<http://www.rfc-editor.org/info/rfc7519>>.
- [RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<http://www.rfc-editor.org/info/rfc7662>>.

Authors' Addresses

Justin Richer (editor)

Email: ietf@justin.richer.org

John Bradley
Ping Identity

Email: ve7jtb@ve7jtb.com
URI: <http://www.thread-safe.com/>

Hannes Tschofenig
ARM Limited
Austria

Email: Hannes.Tschofenig@gmx.net
URI: <http://www.tschofenig.priv.at>

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 22, 2019

M. Jones
Microsoft
B. Campbell
Ping Identity
J. Bradley
Yubico
W. Denniss
Google
October 19, 2018

OAuth 2.0 Token Binding
draft-ietf-oauth-token-binding-08

Abstract

This specification enables OAuth 2.0 implementations to apply Token Binding to Access Tokens, Authorization Codes, Refresh Tokens, JWT Authorization Grants, and JWT Client Authentication. This cryptographically binds these tokens to a client's Token Binding key pair, possession of which is proven on the TLS connections over which the tokens are intended to be used. This use of Token Binding protects these tokens from man-in-the-middle and token export and replay attacks.

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 <https://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 April 22, 2019.

Copyright Notice

Copyright (c) 2018 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
1.1. Requirements Notation and Conventions	3
1.2. Terminology	3
2. Token Binding for Refresh Tokens	4
2.1. Example Token Binding for Refresh Tokens	4
3. Token Binding for Access Tokens	6
3.1. Access Tokens Issued from the Authorization Endpoint . .	7
3.1.1. Example Access Token Issued from the Authorization Endpoint	8
3.2. Access Tokens Issued from the Token Endpoint	9
3.2.1. Example Access Token Issued from the Token Endpoint .	9
3.3. Protected Resource Token Binding Validation	11
3.3.1. Example Protected Resource Request	11
3.4. Representing Token Binding in JWT Access Tokens	11
3.5. Representing Token Binding in Introspection Responses . .	12
4. Token Binding Metadata	13
4.1. Token Binding Client Metadata	13
4.2. Token Binding Authorization Server Metadata	13
5. Token Binding for Authorization Codes	14
5.1. Native Application Clients	14
5.1.1. Code Challenge	14
5.1.1.1. Example Code Challenge	15
5.1.2. Code Verifier	15
5.1.2.1. Example Code Verifier	16
5.2. Web Server Clients	16
5.2.1. Code Challenge	17
5.2.1.1. Example Code Challenge	17
5.2.2. Code Verifier	18
5.2.2.1. Example Code Verifier	18
6. Token Binding JWT Authorization Grants and Client Authentication	19
6.1. JWT Format and Processing Requirements	19
6.2. Token Bound JWTs for Client Authentication	20
6.3. Token Bound JWTs for as Authorization Grants	20
7. Security Considerations	21
7.1. Phasing in Token Binding	21

7.2. Binding of Refresh Tokens	21
8. IANA Considerations	22
8.1. OAuth Dynamic Client Registration Metadata Registration	22
8.1.1. Registry Contents	22
8.2. OAuth Authorization Server Metadata Registration	23
8.2.1. Registry Contents	23
8.3. PKCE Code Challenge Method Registration	23
8.3.1. Registry Contents	23
9. Token Endpoint Authentication Method Registration	23
9.1. Registry Contents	24
10. Sub-Namespace Registrations	24
10.1. Registry Contents	24
11. References	24
11.1. Normative References	24
11.2. Informative References	26
Appendix A. Acknowledgements	27
Appendix B. Document History	27
Authors' Addresses	29

1. Introduction

This specification enables OAuth 2.0 [RFC6749] implementations to apply Token Binding (TLS Extension for Token Binding Protocol Negotiation [RFC8472], The Token Binding Protocol Version 1.0 [RFC8471] and Token Binding over HTTP [RFC8473]) to Access Tokens, Authorization Codes, Refresh Tokens, JWT Authorization Grants, and JWT Client Authentication. This cryptographically binds these tokens to a client's Token Binding key pair, possession of which is proven on the TLS connections over which the tokens are intended to be used. This use of Token Binding protects these tokens from man-in-the-middle and token export and replay attacks.

1.1. Requirements Notation and 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.

1.2. Terminology

This specification uses the terms "Access Token", "Authorization Code", "Authorization Endpoint", "Authorization Server", "Client", "Protected Resource", "Refresh Token", and "Token Endpoint" defined by OAuth 2.0 [RFC6749], the terms "Claim" and "JSON Web Token (JWT)" defined by JSON Web Token (JWT) [JWT], the term "User Agent" defined by RFC 7230 [RFC7230], and the terms "Provided", "Referred", "Token

Binding" and "Token Binding ID" defined by Token Binding over HTTP [RFC8473].

2. Token Binding for Refresh Tokens

Token Binding of refresh tokens is a straightforward first-party scenario, applying term "first-party" as used in Token Binding over HTTP [RFC8473]. It cryptographically binds the refresh token to the client's Token Binding key pair, possession of which is proven on the TLS connections between the client and the token endpoint. This case is straightforward because the refresh token is both retrieved by the client from the token endpoint and sent by the client to the token endpoint. Unlike the federation use cases described in Token Binding over HTTP [RFC8473], Section 4, and the access token case described in the next section, only a single TLS connection is involved in the refresh token case.

Token Binding a refresh token requires that the authorization server do two things. First, when refresh token is sent to the client, the authorization server needs to remember the Provided Token Binding ID and remember its association with the issued refresh token. Second, when a token request containing a refresh token is received at the token endpoint, the authorization server needs to verify that the Provided Token Binding ID for the request matches the remembered Token Binding ID associated with the refresh token. If the Token Binding IDs do not match, the authorization server should return an error in response to the request.

How the authorization server remembers the association between the refresh token and the Token Binding ID is an implementation detail that beyond the scope of this specification. Some authorization servers will choose to store the Token Binding ID (or a cryptographic hash of it, such a SHA-256 hash [SHS]) in the refresh token itself, provided it is integrity-protected, thus reducing the amount of state to be kept by the server. Other authorization servers will add the Token Binding ID value (or a hash of it) to an internal data structure also containing other information about the refresh token, such as grant type information. These choices make no difference to the client, since the refresh token is opaque to it.

2.1. Example Token Binding for Refresh Tokens

This section provides an example of what the interactions around a Token Bound refresh token might look like, along with some details of the involved processing. Token Binding of refresh tokens is most useful for native application clients so the example has protocol elements typical of a native client flow. Extra line breaks in all examples are for display purposes only.

A native application client makes the following access token request with an authorization code using a TLS connection where Token Binding has been negotiated. A PKCE "code_verifier" is included because use of PKCE is considered best practice for native application clients [BCP212]. The base64url-encoded representation of the exported keying material (EKM) from that TLS connection is "p6ZuSwfl6pIe8es5KyeV76T4swZmQp0_awd27jHfrbo", which is needed to validate the Token Binding Message.

```
POST /as/token.oauth2 HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Sec-Token-Binding: AIkAAgBBQGto7hHRR0Y5nkOWqc9KNfwW95dEFmSI_tCZ_Cbl
7LWlt6Xjp3DbjiDjavGfiKP2HV_2JSE42VzmKOVVV8m7eqAAQOKiDKl0i0z6v4X5B
P7uc0pFestVZ42TTOdJmoHpji06Qq3jsCiCRSJx9ck2fWJYx8tLVXRZPATB3x6c24
ay0ZEAAA

grant_type=authorization_code&code=4bwcZesc7Xacc330ltc66Wxk8EAfp9j2
&code_verifier=2x6_ylS390-8V7jaT9wj.8qP9nKmYcf.V-rD9O4r_1
&client_id=example-native-client-id
```

Figure 1: Initial Request with Code

A refresh token is issued in response to the prior request. Although it looks like a typical response to the client, the authorization server has bound the refresh token to the Provided Token Binding ID from the encoded Token Binding message in the "Sec-Token-Binding" header of the request. In this example, that binding is done by saving the Token Binding ID alongside other information about the refresh token in some server side persistent storage. The base64url-encoded representation of that Token Binding ID is "AgBBQGto7hHRR0Y5nkOWqc9KNfwW95dEFmSI_tCZ_Cbl7LWlt6Xjp3DbjiDjavGfiKP2HV_2JSE42VzmKOVVV8m7eqA".

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-cache, no-store

{
  "access_token": "EdRs7qMrLb167Z9fV2dcwoLTC",
  "refresh_token": "ACClZEIQTjW9arT9GOJGGd7QNwqOMmUYfsJTiv8his4",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

Figure 2: Successful Response

When the access token expires, the client requests a new one with a refresh request to the token endpoint. In this example, the request is made on a new TLS connection so the EKM (base64url-encoded: "va-84Ukw4Zqfd7uWotFrAJda96WwgbdaPDX2knoOiAE") and signature in the Token Binding Message are different than in the initial request.

```
POST /as/token.oauth2 HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Sec-Token-Binding: AikAAgBBQGto7hHRR0Y5nkOWqc9KNfwW95dEFmSI_tCZ_Cbl
  7LWlt6Xjp3DbjiDJavGFikP2HV_2JSE42VzmKOVVV8m7eqAAQCpGbaG_YRf27qOra
  L0UT4fsKKjL6PukuOT00qzamoAXxOq7m_id7O3mLpnbsM7kwSxLi7iNHzzDgCAkP
  t3lHwAAA

refresh_token=ACClZEIQTjW9arT9GOJGGd7QNwqOMmUYfsJTiv8his4
&grant_type=refresh_token&client_id=example-native-client-id
```

Figure 3: Refresh Request

However, because the Token Binding ID is long-lived and may span multiple TLS sessions and connections, it is the same as in the initial request. That Token Binding ID is what the refresh token is bound to, so the authorization server is able to verify it and issue a new access token.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-cache, no-store

{
  "access_token": "bwcESCwC4yOCQ8iPsgcn117k7",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

Figure 4: Successful Response

3. Token Binding for Access Tokens

Token Binding for access tokens cryptographically binds the access token to the client's Token Binding key pair, possession of which is proven on the TLS connections between the client and the protected resource. Token Binding is applied to access tokens in a similar manner to that described in Token Binding over HTTP [RFC8473], Section 4 (Federation Use Cases). It also builds upon the mechanisms for Token Binding of ID Tokens defined in OpenID Connect Token Bound Authentication 1.0 [OpenID.TokenBinding].

In the OpenID Connect [OpenID.Core] use case, HTTP redirects are used to pass information between the identity provider and the relying party; this HTTP redirect makes the Token Binding ID of the relying party available to the identity provider as the Referred Token Binding ID, information about which is then added to the ID Token. No such redirect occurs between the authorization server and the protected resource in the access token case; therefore, information about the Token Binding ID for the TLS connection between the client and the protected resource needs to be explicitly communicated by the client to the authorization server to achieve Token Binding of the access token.

This information is passed to the authorization server using the Referred Token Binding ID, just as in the ID Token case. The only difference is that the client needs to explicitly communicate the Token Binding ID of the TLS connection between the client and the protected resource to the Token Binding implementation so that it is sent as the Referred Token Binding ID in the request to the authorization server. This functionality provided by Token Binding implementations is described in Implementation Considerations of Token Binding over HTTP [RFC8473], Section 6.

Note that to obtain this Token Binding ID, the client may need to establish a TLS connection between itself and the protected resource prior to making the request to the authorization server so that the Provided Token Binding ID for the TLS connection to the protected resource can be obtained. How the client retrieves this Token Binding ID from the underlying Token Binding API is implementation and operating system specific. An alternative, if supported, is for the client to generate a Token Binding key to use for the protected resource, use the Token Binding ID for that key, and then later use that key when the TLS connection to the protected resource is established.

3.1. Access Tokens Issued from the Authorization Endpoint

For access tokens returned directly from the authorization endpoint, such as with the implicit grant defined in OAuth 2.0 [RFC6749], Section 4.2, the Token Binding ID of the client's TLS channel to the protected resource is sent with the authorization request as the Referred Token Binding ID in the "Sec-Token-Binding" header, and is used to Token Bind the access token.

Upon receiving the Referred Token Binding ID in an authorization request, the authorization server associates (Token Binds) the ID with the access token in a way that can be accessed by the protected resource. Such methods include embedding the Referred Token Binding ID (or a cryptographic hash of it) in the issued access token itself,

possibly using the syntax described in Section 3.4, or through token introspection as described in Section 3.5. The method for associating the referred token binding ID with the access token is determined by the authorization server and the protected resource, and is beyond the scope for this specification.

3.1.1. Example Access Token Issued from the Authorization Endpoint

This section provides an example of what the interactions around a Token Bound access token issued from the authorization endpoint might look like, along with some details of the involved processing. Extra line breaks in all examples are for display purposes only.

The client directs the user-agent to make the following HTTP request to the authorization endpoint. It is a typical authorization request that, because Token Binding was negotiated on the underlying TLS connection and the user-agent was signaled to reveal the Referred Token Binding, also includes the "Sec-Token-Binding" header with a Token Binding Message that contains both a Provided and Referred Token Binding. The base64url-encoded EKM from the TLS connection over which the request was made is "ji5UAYjs5XCPISUGQIwgcSrOiViWq4fhLVIFTQ4nLxc".

```
GET /as/authorization.oauth2?response_type=token
    &client_id=example-client-id&state=rM8pZxGlc3gKy6rEbsD8s
    &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Eorg%2Fcb HTTP/1.1
Host: server.example.com
Sec-Token-Binding: ARIAAgBBQIEE8mSMtDy2dj9EEBdXaQT9W3Rq1NS-jw8ebPoF
6FyL0jiFATVE55zIircgOTZmEglxeIrc3DsGegwjs4bhw14AQGKDLAXFFMyQkZegC
wlbTlq3F9HTt-lJxFU_pil6ezka7qVRCpSF0BQLfSqlsXmbyfSSCJX1BDtrIL7PX
jvfUAAAECAEFALBNuP3te5WrwlEwiejEz0Opesmc5PElWk7xZ5n1LSqQTj1ciIp
5vQ301LLUCyM_a2BYTUPkt5EdS-Pa1t4t6ABZADgeizR5NcTmuX4zOdC-R4cLNNVV
081Lu2Psko-UJLR XAH4O0H7-m0 nOR1zBN78nYMKPvHsz8L3zWKRvYxEGAA
```

Figure 5: Authorization Request

The authorization server issues an access token and delivers it to the client by redirecting the user-agent with the following HTTP response:

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb#state=rM8pZxGlc3gKy6rEbsD8s
    &expires_in=3600&token_type=Bearer
    &access_token=eyJhbGciOiJIUzI1IiwiaWF0IjoxNTE2MzY0MDAifQ.xxy5W5sQ
```

Figure 6: Authorization Response

The access token is bound to the Referred Token Binding ID from the authorization request, which when represented as a JWT, as described in Section 3.4, contains the SHA-256 hash of the Token Binding ID as the value of the "tbh" (token binding hash) member of the "cnf" (confirmation) claim. The confirmation claim portion of the JWT Claims Set is shown in the following figure.

```
{
  ...other claims omitted for brevity...
  "cnf":{
    "tbh": "vowQESa_MgbGJwIXaFm_BTN2QDPwh8PhuBm-EtUAqxc"
  }
}
```

Figure 7: Confirmation Claim

3.2. Access Tokens Issued from the Token Endpoint

For access tokens returned from the token endpoint, the Token Binding ID of the client's TLS channel to the protected resource is sent as the Referred Token Binding ID in the "Sec-Token-Binding" header, and is used to Token Bind the access token. This applies to all the grant types from OAuth 2.0 [RFC6749] using the token endpoint, including, but not limited to the refresh and authorization code token requests, as well as some extension grants, such as JWT assertion authorization grants [RFC7523].

Upon receiving the Referred Token Binding ID in a token request, the authorization server associates (Token Binds) the ID with the access token in a way that can be accessed by the protected resource. Such methods include embedding the Referred Token Binding ID (or a cryptographic hash of it) in the issued access token itself, possibly using the syntax described in Section 3.4, or through token introspection as described in Section 3.5. The method for associating the referred token binding ID with the access token is determined by the authorization server and the protected resource, and is beyond the scope for this specification.

Note that if the request results in a new refresh token being generated, it can be Token bound using the Provided Token Binding ID, per Section 2.

3.2.1. Example Access Token Issued from the Token Endpoint

This section provides an example of what the interactions around a Token Bound access token issued from the token endpoint might look like, along with some details of the involved processing. Extra line breaks in all examples are for display purposes only.

The client makes an access token request to the token endpoint and includes the "Sec-Token-Binding" header with a Token Binding Message that contains both Provided and Referred Token Binding IDs. The Provided Token Binding ID is used to validate the token binding of the refresh token in the request (and to Token Bind a new refresh token, if one is issued), and the Referred Token Binding ID is used to Token Bind the access token that is generated. The base64url-encoded EKM from the TLS connection over which the access token request was made is "4jTc5elQpocqPTZ5l6jsb6pRP18IFKdwwPvasYjnl-E".

```
POST /as/token.oauth2 HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Sec-Token-Binding: ARIAAGBBQJFXJir2w4gbJ7grBx9uTYWIrs9V50-PW4ZijegQ
  0LUM-_bGnGT6DizxUK-m5n3dQUIkeH7ybn6wb1C5dGyV_IAAQDDFTtoFrHt41Zppq7
  u_SEMF_E-KimAB-HewWl2MvZzAQ9QKoWiJCLFiCkjpgtr1RrA2-jaJvoB8o51DTGXQ
  ydWYkAAAECAEFaUc1G1YU83rqTGHEauloqvNwy0fDsdXzIyT_4x1FcldsMxjFkJac
  IBJFGuYcccvnCak_duFi3QKFENuwxql-H9ABAMcU7IjJOUA4IyE6YoEcfz9BMPQqw
  M5M6hw4RZNQd58fsTCCslQE_NmNc19JXy4NkdkeZBxqvZGPr0y8QZ_bmAwAA

refresh_token=gZR_ZI8EAhLgWR-gWxBimbgZRZi_8EAhLgWRgWxBimbf
&grant_type=refresh_token&client_id=example-client-id
```

Figure 8: Access Token Request

The authorization server issues an access token bound to the Referred Token Binding ID and delivers it in a response the client.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-cache, no-store

{
  "access_token": "eyJhbGciOiJIJFZlbnNlIjE9Imtp[...omitted...]1cs29j5c3",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

Figure 9: Response

The access token is bound to the Referred Token Binding ID of the access token request, which when represented as a JWT, as described in Section 3.4, contains the SHA-256 hash of the Token Binding ID as the value of the "tbh" (token binding hash) member of the "cnf" (confirmation) claim. The confirmation claim portion of the JWT Claims Set of the access token is shown in the following figure.

```
{
  ...other claims omitted for brevity...
  "cnf":{
    "tbh": "7NRBu9iDdJlYCTOqyeYuLxXv0blEA-yTpmGirAwKAws"
  }
}
```

Figure 10: Confirmation Claim

3.3. Protected Resource Token Binding Validation

Upon receiving a token bound access token, the protected resource validates the binding by comparing the Provided Token Binding ID to the Token Binding ID for the access token. Alternatively, cryptographic hashes of these Token Binding ID values can be compared. If the values do not match, the resource access attempt MUST be rejected with an error.

3.3.1. Example Protected Resource Request

For example, a protected resource request using the access token from Section 3.2.1 would look something like the following. The base64url-encoded EKM from the TLS connection over which the request was made is "7LsNP3BT1aHHdXdk6meEWjtSkiPVLb7YS6iHp-JXmuE". The protected resource validates the binding by comparing the Provided Token Binding ID from the "Sec-Token-Binding" header to the token binding hash confirmation of the access token. Extra line breaks in the example are for display purposes only.

```
GET /api/stuff HTTP/1.1
Host: resource.example.org
Authorization: Bearer eyJhbGciOiJIUzI1NiIsI[...omitted...]cs29j5c3
Sec-Token-Binding: AIkAAgBBQLGtRpWFPN66kxhxGrtaKrzcmThw7HV8yMk_-Mdr
XJXbDMYxZCWnCASRRrmHHHL5wmpP3bhYt0ChRDbsMapfh_QAQN1He3Ftj4Wa_S_fz
ZVns4saLfj6aBoMSQW6rLs19IiVhZe7LrGjKyCfPTKXjaJebxp-TLPFZCc0JTqTY5
0MBAAAA
```

Figure 11: Protected Resource Request

3.4. Representing Token Binding in JWT Access Tokens

If the access token is represented as a JWT, the token binding information SHOULD be represented in the same way that it is in token bound OpenID Connect ID Tokens [OpenID.TokenBinding]. That specification defines the new JWT Confirmation Method RFC 7800 [RFC7800] member "tbh" (token binding hash) to represent the SHA-256 hash of a Token Binding ID in an ID Token. The value of the "tbh" member is the base64url encoding of the SHA-256 hash of the Token

Binding ID. All trailing pad '=' characters are omitted from the encoded value and no line breaks, whitespace, or other additional characters are included.

The following example demonstrates the JWT Claims Set of an access token containing the base64url encoding of the SHA-256 hash of a Token Binding ID as the value of the "tbh" (token binding hash) element in the "cnf" (confirmation) claim:

```
{
  "iss": "https://server.example.com",
  "aud": "https://resource.example.org",
  "sub": "brian@example.com",
  "iat": 1467324320,
  "exp": 1467324920,
  "cnf": {
    "tbh": "7NRBu9iDdJlYCTOgyeYuLxXv0bleA-yTpmGirAwKAws"
  }
}
```

Figure 12: JWT with Token Binding Hash Confirmation Claim

3.5. Representing Token Binding in Introspection Responses

OAuth 2.0 Token Introspection [RFC7662] defines a method for a protected resource to query an authorization server about the active state of an access token as well as to determine meta-information about the token.

For a token bound access token, the hash of the Token Binding ID to which the token is bound is conveyed to the protected resource as meta-information in a token introspection response. The hash is conveyed using same structure as the token binding hash confirmation method, described in Section 3.4, as a top-level member of the introspection response JSON. The protected resource compares that token binding hash to a hash of the provided Token Binding ID and rejects the request, if they do not match.

The following is an example of an introspection response for an active token bound access token with a "tbh" token binding hash confirmation method.

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "active": true,
  "iss": "https://server.example.com",
  "aud": "https://resource.example.org",
  "sub": "brian@example.com",
  "iat": 1467324320,
  "exp": 1467324920,
  "cnf": {
    "tbh": "7NRBu9iDdJlYCTOqyeYuLxXv0blEA-yTpmGirAwKAws"
  }
}
```

Figure 13: Example Introspection Response for a Token Bound Access Token

4. Token Binding Metadata

4.1. Token Binding Client Metadata

Clients supporting Token Binding that also support the OAuth 2.0 Dynamic Client Registration Protocol [RFC7591] use these metadata values to declare their support for Token Binding of access tokens and refresh tokens:

`client_access_token_token_binding_supported`
OPTIONAL. Boolean value specifying whether the client supports Token Binding of access tokens. If omitted, the default value is "false".

`client_refresh_token_token_binding_supported`
OPTIONAL. Boolean value specifying whether the client supports Token Binding of refresh tokens. If omitted, the default value is "false". Authorization servers MUST NOT Token Bind refresh tokens issued to a client that does not support Token Binding of refresh tokens, but MAY reject requests completely from such clients if token binding is required by authorization server policy by returning an OAuth error response.

4.2. Token Binding Authorization Server Metadata

Authorization servers supporting Token Binding that also support OAuth 2.0 Authorization Server Metadata [RFC8414] use these metadata values to declare their support for Token Binding of access tokens and refresh tokens:

`as_access_token_token_binding_supported`

OPTIONAL. Boolean value specifying whether the authorization server supports Token Binding of access tokens. If omitted, the default value is "false".

`as_refresh_token_token_binding_supported`

OPTIONAL. Boolean value specifying whether the authorization server supports Token Binding of refresh tokens. If omitted, the default value is "false".

5. Token Binding for Authorization Codes

There are two variations for Token Binding of an authorization code. One is appropriate for native application clients and the other for web server clients. The nature of where the various components reside for the different client types demands different methods of Token Binding the authorization code so that it is bound to a Token Binding key on the end user's device. This ensures that a lost or stolen authorization code cannot be successfully utilized from a different device. For native application clients, the code is bound to a Token Binding key pair that the native client itself possesses. For web server clients, the code is bound to a Token Binding key pair on the end user's browser. Both variations utilize the extensible framework of Proof Key for Code Exchange (PKCE) [RFC7636], which enables the client to show possession of a certain key when exchanging the authorization code for tokens. The following subsections individually describe each of the two PKCE methods respectively.

5.1. Native Application Clients

This section describes a PKCE method suitable for native application clients that cryptographically binds the authorization code to a Token Binding key pair on the client, which the client proves possession of on the TLS connection during the access token request containing the authorization code. The authorization code is bound to the Token Binding ID that the native application client uses to resolve the authorization code at the token endpoint. This binding ensures that the client that made the authorization request is the same client that is presenting the authorization code.

5.1.1. Code Challenge

As defined in Proof Key for Code Exchange [RFC7636], the client sends the code challenge as part of the OAuth 2.0 authorization request with the two additional parameters: "code_challenge" and "code_challenge_method".

For this Token Binding method of PKCE, "TB-S256" is used as the value of the "code_challenge_method" parameter.

The value of the "code_challenge" parameter is the base64url encoding (per Section 5 of [RFC4648] with all trailing padding ('=') characters omitted and without the inclusion of any line breaks or whitespace) of the SHA-256 hash of the Provided Token Binding ID that the client will use when calling the authorization server's token endpoint. Note that, prior to making the authorization request, the client may need to establish a TLS connection between itself and the authorization server's token endpoint in order to establish the appropriate Token Binding ID.

When the authorization server issues the authorization code in the authorization response, it associates the code challenge and method values with the authorization code so they can be verified later when the authorization code is presented in the access token request.

5.1.1.1. Example Code Challenge

For example, a native application client sends an authorization request by sending the user's browser to the authorization endpoint. The resulting HTTP request looks something like the following (with extra line breaks for display purposes only).

```
GET /as/authorization.oauth2?response_type=code
    &client_id=example-native-client-id&state=oUC2jyYtzRCrMyWrVnGj
    &code_challenge=rBlgOyMY4teiuJMDgOwkrpsAjPyI07D2WseM-dnq6eE
    &code_challenge_method=TB-S256 HTTP/1.1
Host: server.example.com
```

Figure 14: Authorization Request with PKCE Challenge

5.1.1.2. Code Verifier

Upon receipt of the authorization code, the client sends the access token request to the token endpoint. The Token Binding Protocol [RFC8471] is negotiated on the TLS connection between the client and the authorization server and the "Sec-Token-Binding" header, as defined in Token Binding over HTTP [RFC8473], is included in the access token request. The authorization server extracts the Provided Token Binding ID from the header value, hashes it with SHA-256, and compares it to the "code_challenge" value previously associated with the authorization code. If the values match, the token endpoint continues processing as normal (as defined by OAuth 2.0 [RFC6749]). If the values do not match, an error response indicating "invalid_grant" MUST be returned.

The "Sec-Token-Binding" header contains sufficient information for verification of the authorization code and its association to the original authorization request. However, PKCE [RFC7636] requires that a "code_verifier" parameter be sent with the access token request, so the static value "provided_tb" is used to meet that requirement and indicate that the Provided Token Binding ID is used for the verification.

5.1.2.1. Example Code Verifier

An example access token request, correlating to the authorization request in the previous example, to the token endpoint over a TLS connection for which Token Binding has been negotiated would look like the following (with extra line breaks for display purposes only). The base64url-encoded EKM from the TLS connection over which the request was made is

```
"pNVKtPuQFvylNYn000QowWrQKoeMkeX9H32hVuU71Bs".
```

```
POST /as/token.oauth2 HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Sec-Token-Binding: AIkAAgBBQEO09GRFP-LM0hoWw6-2i318BsuuUum5AL8bt1sz
  lr1EFfp5DMXMNW3O8WjcIXr2DKJnI4xnuGse6GywQd9RbD0AQJDb3xyo9PBxj8M6Y
  jLt-6OaxgDkyoBoTkrynNbLc8tJQ0JtXomKzBbj5qPtHDduXc6xz_lzvNpxSPxi42
  8m7wkAAA
```

```
grant_type=authorization_code&code=mJAReTWKX7zI3oHUNd4o3PeNqNqxKGp6
  &code_verifier=provided_tb&client_id=example-native-client-id
```

Figure 15: Token Request with PKCE Verifier

5.2. Web Server Clients

This section describes a PKCE method suitable for web server clients, which cryptographically binds the authorization code to a Token Binding key pair on the browser. The authorization code is bound to the Token Binding ID that the browser uses to deliver the authorization code to a web server client, which is sent to the authorization server as the Referred Token Binding ID during the authorization request. The web server client conveys the Token Binding ID to the authorization server when making the access token request containing the authorization code. This binding ensures that the authorization code cannot successfully be played or replayed to the web server client from a different browser than the one that made the authorization request.

5.2.1. Code Challenge

As defined in Proof Key for Code Exchange [RFC7636], the client sends the code challenge as part of the OAuth 2.0 Authorization Request with the two additional parameters: "code_challenge" and "code_challenge_method".

The client must send the authorization request through the browser such that the Token Binding ID established between the browser and itself is revealed to the authorization server's authorization endpoint as the Referred Token Binding ID. Typically, this is done with an HTTP redirection response and the "Include-Referred-Token-Binding-ID" header, as defined in Token Binding over HTTP [RFC8473], Section 5.3.

For this Token Binding method of PKCE, "referred_tb" is used for the value of the "code_challenge_method" parameter.

The value of the "code_challenge" parameter is "referred_tb". The static value for the required PKCE parameter indicates that the authorization code is to be bound to the Referred Token Binding ID from the Token Binding Message sent in the "Sec-Token-Binding" header of the authorization request.

When the authorization server issues the authorization code in the authorization response, it associates the Token Binding ID (or hash thereof) and code challenge method with the authorization code so they can be verified later when the authorization code is presented in the access token request.

5.2.1.1. Example Code Challenge

For example, the web server client sends the authorization request by redirecting the browser to the authorization endpoint. That HTTP redirection response looks like the following (with extra line breaks for display purposes only).

```
HTTP/1.1 302 Found
Location: https://server.example.com?response_type=code
        &client_id=example-web-client-id&state=P4FUFqYzslj3ffsYCP34d3
        &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Eorg%2Fcb
        &code_challenge=referred_tb&code_challenge_method=referred_tb
Include-Referred-Token-Binding-ID: true
```

Figure 16: Redirect the Browser

The redirect includes the "Include-Referred-Token-Binding-ID" response header field that signals to the user-agent that it should

reveal, to the authorization server, the Token Binding ID used on the connection to the web server client. The resulting HTTP request to the authorization server looks something like the following (with extra line breaks for display purposes only). The base64url-encoded EKM from the TLS connection over which the request was made is "7gOdRzMhPeO-1YwZGmnVHyReN5vd2CxcSRBN69Ue4cI".

```
GET /as/authorization.oauth2?response_type=code
  &client_id=example-web-client-id&state=dryo8YFpWacBUPjhBf4Nvt51
  &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Eorg%2Fcb
  &code_challenge=referred_tb
  &code_challenge_method=referred_tb HTTP/1.1
Host: server.example.com
Sec-Token-Binding: ARIAAGBBQB-XOPf5ePlf7ikATiAFEGOS503lPmRfkyymzdWw
  HCxl0njxC3D0E_OVfBNqrIQxzIfkF7tWby2ZfyaE6XpwTsAQBYqhFX78vMOgDX_F
  d_b2dlHyHlMmkIz8iMVBY_reM98OUaJFz5IB7PG9nZ11j58LoG5QhmQoI9NXYktKZ
  RXxrYAAAECAEFAdUFTnfQADknluDbQnvJEk6oQs38L92gv-KO-qlYadLoDIKe2h53
  hSiKwIP98iRj_unedkNkAMyg9e2mY4Gp7WwBAeDUOwaSXNz1e6gKohwN4SAZ5eNyx
  45Mh8VI4woLlBipLoqrJRok6dxFkWGHRMuBRocLGUj5PiOoxybQH_Tom3gAA
```

Figure 17: Authorization Request

5.2.2. Code Verifier

The web server client receives the authorization code from the browser and extracts the Provided Token Binding ID from the "Sec-Token-Binding" header of the request. The client sends the base64url-encoded (per Section 5 of [RFC4648] with all trailing padding ('=') characters omitted and without the inclusion of any line breaks or whitespace) Provided Token Binding ID as the value of the "code_verifier" parameter in the access token request to the authorization server's token endpoint. The authorization server compares the value of the "code_verifier" parameter to the Token Binding ID value previously associated with the authorization code. If the values match, the token endpoint continues processing as normal (as defined by OAuth 2.0 [RFC6749]). If the values do not match, an error response indicating "invalid_grant" MUST be returned.

5.2.2.1. Example Code Verifier

Continuing the example from the previous section, the authorization server sends the code to the web server client by redirecting the browser to the client's "redirect_uri", which results in the browser making a request like the following (with extra line breaks for display purposes only) to the web server client over a TLS channel for which Token Binding has been established. The base64url-encoded EKM from the TLS connection over which the request was made is "EzW60vyINbsb_tajt8ij3tV6cwY2KH-i8BdEMYXcNn0".

```
GET /cb?state=dryo8YFpWacbUPjhBf4Nvt5l&code=jwD3oOa5cQvvLc81bwc4CMw
Host: client.example.org
Sec-Token-Binding: AIAAgBBQHVBu530AA5J9bg20J7yRJOqELN_C_doL_ijvqpW
  GnS6AyCnloed4UoisCD_fIkY_7p3nZDZADMoPXtpmOBqelsAQEwgC9Zpg7QFCDBib
  6GlZki3MhH32KNfLefLJclvRlxE8l7OMfPLZHP2Woxh6rEtmgBcAABubEbTz7muNl
  Ln8uoAAA
```

Figure 18: Authorization Response to Web Server Client

The web server client takes the Provided Token Binding ID from the above request from the browser and sends it, base64url encoded, to the authorization server in the "code_verifier" parameter of the authorization code grant type request. Extra line breaks in the example request are for display purposes only.

```
POST /as/token.oauth2 HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic b3JnLmV4YWlwbGUuY2xpZW50OmlldGY5OGNoaWNhZ28=

grant_type=authorization_code&code=jwD3oOa5cQvvLc81bwc4CMw
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Eorg%2Fcb
&client_id=example-web-client-id
&code_verifier=AgBBQHVBu530AA5J9bg20J7yRJOqELN_C_doL_ijv
qpWGnS6AyCnloed4UoisCD_fIkY_7p3nZDZADMoPXtpmOBqels
```

Figure 19: Exchange Authorization Code

6. Token Binding JWT Authorization Grants and Client Authentication

The JWT Profile for OAuth 2.0 Client Authentication and Authorization Grants [RFC7523] defines the use of bearer JWTs as a means for requesting an OAuth 2.0 access token as well as for client authentication. This section describes extensions to that specification enabling the application of Token Binding to JWT client authentication and JWT authorization grants.

6.1. JWT Format and Processing Requirements

In addition the requirements set forth in Section 3 of RFC 7523 [RFC7523], the following criteria must also be met for token bound JWTs used as authorization grants or for client authentication.

- o The JWT MUST contain a "cnf" (confirmation) claim with a "tbh" (token binding hash) member identifying the Token Binding ID of the Provided Token Binding used by the client on the TLS connection to the authorization server. The authorization server MUST reject any JWT that has a token binding hash confirmation

that does not match the corresponding hash of the Provided Token Binding ID from the "Sec-Token-Binding" header of the request.

6.2. Token Bound JWTs for Client Authentication

To use a token bound JWT for client authentication, the client uses the parameter values and encodings from Section 2.2 of RFC 7523 [RFC7523] with one exception: the value of the "client_assertion_type" is "urn:ietf:params:oauth:client-assertion-type:jwt-token-bound".

The "OAuth Token Endpoint Authentication Methods" registry [IANA.OAuth.Parameters] contains values, each of which specify a method of authenticating a client to the authorization server. The values are used to indicate supported and utilized client authentication methods in authorization server metadata, such as [OpenID.Discovery] and [RFC8414], and in OAuth 2.0 Dynamic Client Registration Protocol [RFC7591]. The values "private_key_jwt" and "client_secret_jwt" are designated by OpenID Connect [OpenID.Core] as authentication method values for bearer JWT client authentication using asymmetric and symmetric JWS [RFC7515] algorithms respectively. For Token Bound JWT for client authentication, this specification defines and registers the following authentication method values.

private_key_token_bound_jwt

Indicates that client authentication to the authorization server will occur with a Token Bound JWT, which is signed with a client's private key.

client_secret_token_bound_jwt

Indicates that client authentication to the authorization server will occur with a Token Bound JWT, which is integrity protected with a MAC using the octets of the UTF-8 representation of the client secret as the shared key.

Note that just as with the "private_key_jwt" and "client_secret_jwt" authentication methods, the "token_endpoint_auth_signing_alg" client registration parameter may be used to indicate the JWS algorithm used for signing the client authentication JWT for the authentication methods defined above.

6.3. Token Bound JWTs for as Authorization Grants

To use a token bound JWT for an authorization grant, the client uses the parameter values and encodings from Section 2.1 of RFC 7523 [RFC7523] with one exception: the value of the "grant_type" is "urn:ietf:params:oauth:grant-type:jwt-token-bound".

7. Security Considerations

7.1. Phasing in Token Binding

Many OAuth implementations will be deployed in situations in which not all participants support Token Binding. Any combination of the client, the authorization server, the protected resource, and the user agent may not yet support Token Binding, in which case it will not work end-to-end.

It is a context-dependent deployment choice whether to allow interactions to proceed in which Token Binding is not supported or whether to treat the omission of Token Binding at any step as a fatal error. Particularly in dynamic deployment environments in which End Users have choices of clients, authorization servers, protected resources, and/or user agents, it is recommended that, for some reasonable period of time during which Token Binding technology is being adopted, authorizations using one or more components that do not implement Token Binding be allowed to successfully proceed. This enables different components to be upgraded to supporting Token Binding at different times, providing a smooth transition path for phasing in Token Binding. However, when Token Binding has been performed, any Token Binding key mismatches **MUST** be treated as fatal errors.

In more controlled deployment environments where the participants in an authorization interaction are known or expected to support Token Binding and yet one or more of them does not use it, the authorization **SHOULD** be aborted with an error. For instance, an authorization server should reject a token request that does not include the "Sec-Token-Binding" header, if the request is from a client known to support Token Binding (via configuration or the "client_access_token_token_binding_supported" metadata parameter).

7.2. Binding of Refresh Tokens

Section 6 of RFC 6749 [RFC6749] requires that a refresh token be bound to the client to which it was issued and that, if the client type is confidential or the client was issued client credentials (or assigned other authentication requirements), the client must authenticate with the authorization server when presenting the refresh token. As a result, for non-public clients, refresh tokens are indirectly bound to the client's credentials and cannot be used without the associated client authentication. Non-public clients then are afforded protections (equivalent to the strength of their authentication credentials) against unauthorized replay of refresh tokens and it is reasonable to not Token Bind refresh tokens for such clients while still Token Binding the issued access tokens. Refresh

tokens issued to public clients, however, do not have the benefit of such protections and authorization servers MAY elect to disallow public clients from registering or establishing configuration that would allow Token Bound access tokens but unbound refresh tokens.

Some web-based confidential clients implemented as distributed nodes may be perfectly capable of implementing access token binding (if the access token remains on the node it was bound to, the token binding keys would be locally available for that node to prove possession), but may struggle with refresh token binding due to an inability to share token binding key material between nodes. As confidential clients already have credentials which are required to use the refresh token, and those credentials should only ever be sent over TLS server-to-server between the client and the Token Endpoint, there is still value in token binding access tokens without token binding refresh tokens. Authorization servers SHOULD consider supporting access token binding without refresh token binding for confidential web clients as there are still security benefits to do so.

Clients MUST declare through dynamic (Section 4.1) or static registration information what types of token bound tokens they support to enable the server to bind tokens accordingly, taking into account any phase-in policies. Authorization servers MAY reject requests from any client who does not support token binding (by returning an OAuth error response) per their own security policies.

8. IANA Considerations

8.1. OAuth Dynamic Client Registration Metadata Registration

This specification registers the following client metadata definitions in the IANA "OAuth Dynamic Client Registration Metadata" registry [IANA.OAuth.Parameters] established by [RFC7591]:

8.1.1. Registry Contents

- o Client Metadata Name:
"client_access_token_token_binding_supported"
- o Client Metadata Description: Boolean value specifying whether the client supports Token Binding of access tokens
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this specification]]

- o Client Metadata Name:
"client_refresh_token_token_binding_supported"
- o Client Metadata Description: Boolean value specifying whether the client supports Token Binding of refresh tokens
- o Change Controller: IESG

- o Specification Document(s): Section 4.1 of [[this specification]]

8.2. OAuth Authorization Server Metadata Registration

This specification registers the following metadata definitions in the IANA "OAuth Authorization Server Metadata" registry [IANA.OAuth.Parameters] established by [RFC8414]:

8.2.1. Registry Contents

- o Metadata Name: "as_access_token_token_binding_supported"
- o Metadata Description: Boolean value specifying whether the authorization server supports Token Binding of access tokens
- o Change Controller: IESG
- o Specification Document(s): Section 4.2 of [[this specification]]
- o Metadata Name: "as_refresh_token_token_binding_supported"
- o Metadata Description: Boolean value specifying whether the authorization server supports Token Binding of refresh tokens
- o Change Controller: IESG
- o Specification Document(s): Section 4.2 of [[this specification]]

8.3. PKCE Code Challenge Method Registration

This specification requests registration of the following Code Challenge Method Parameter Names in the IANA "PKCE Code Challenge Methods" registry [IANA.OAuth.Parameters] established by [RFC7636].

8.3.1. Registry Contents

- o Code Challenge Method Parameter Name: TB-S256
- o Change controller: IESG
- o Specification document(s): Section 5.1.1 of [[this specification]]
- o Code Challenge Method Parameter Name: referred_tb
- o Change controller: IESG
- o Specification document(s): Section 5.2.1 of [[this specification]]

9. Token Endpoint Authentication Method Registration

This specification requests registration of the following values in the IANA "OAuth Token Endpoint Authentication Methods" registry [IANA.OAuth.Parameters] established by [RFC7591].

9.1. Registry Contents

- o Token Endpoint Authentication Method Name:
"client_secret_token_bound_jwt"
- o Change Controller: IESG
- o Specification Document(s): Section 6 of [[this specification]]

- o Token Endpoint Authentication Method Name:
"private_key_token_bound_jwt"
- o Change Controller: IESG
- o Specification Document(s): Section 6 of [[this specification]]

10. Sub-Namespace Registrations

This specification requests registration of the following values in the IANA "OAuth URI" registry [IANA.OAuth.Parameters] established in An IETF URN Sub-Namespace for OAuth [RFC6755].

10.1. Registry Contents

- o URN: urn:ietf:params:oauth:grant-type:jwt-token-bound
- o Common Name: Token Bound JWT Grant Type for OAuth 2.0
- o Change controller: IESG
- o Specification Document: Section 6 of [[this specification]]

- o URN: urn:ietf:params:oauth:client-assertion-type:jwt-token-bound
- o Common Name: Token Bound JWT for OAuth 2.0 Client Authentication
- o Change controller: IESG
- o Specification Document: Section 6 of [[this specification]]

11. References

11.1. Normative References

- [IANA.OAuth.Parameters]
IANA, "OAuth Parameters",
<<http://www.iana.org/assignments/oauth-parameters>>.
- [JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015,
<<http://tools.ietf.org/html/rfc7519>>.
- [OpenID.TokenBinding]
Jones, M., Bradley, J., and B. Campbell, "OpenID Connect Token Bound Authentication 1.0", October 2017,
<http://openid.net/specs/openid-connect-token-bound-authentication-1_0-03.html>.

- [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>>.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.
- [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>>.
- [RFC7523] Jones, M., Campbell, B., and C. Mortimore, "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7523, DOI 10.17487/RFC7523, May 2015, <<https://www.rfc-editor.org/info/rfc7523>>.
- [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>>.
- [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>>.
- [RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<https://www.rfc-editor.org/info/rfc7662>>.
- [RFC7800] Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", RFC 7800, DOI 10.17487/RFC7800, April 2016, <<https://www.rfc-editor.org/info/rfc7800>>.
- [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>>.

- [RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <<https://www.rfc-editor.org/info/rfc8414>>.
- [RFC8471] Popov, A., Ed., Nystroem, M., Balfanz, D., and J. Hodges, "The Token Binding Protocol Version 1.0", RFC 8471, DOI 10.17487/RFC8471, October 2018, <<https://www.rfc-editor.org/info/rfc8471>>.
- [RFC8472] Popov, A., Ed., Nystroem, M., and D. Balfanz, "Transport Layer Security (TLS) Extension for Token Binding Protocol Negotiation", RFC 8472, DOI 10.17487/RFC8472, October 2018, <<https://www.rfc-editor.org/info/rfc8472>>.
- [RFC8473] Popov, A., Nystroem, M., Balfanz, D., Ed., Harper, N., and J. Hodges, "Token Binding over HTTP", RFC 8473, DOI 10.17487/RFC8473, October 2018, <<https://www.rfc-editor.org/info/rfc8473>>.
- [SHS] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS PUB 180-4, March 2012, <<http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>>.

11.2. Informative References

- [BCP212] Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps", BCP 212, RFC 8252, DOI 10.17487/RFC8252, October 2017, <<https://www.rfc-editor.org/info/rfc8252>>.
- [OpenID.Core] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", August 2015, <http://openid.net/specs/openid-connect-core-1_0.html>.
- [OpenID.Discovery] Sakimura, N., Bradley, J., Jones, M., and E. Jay, "OpenID Connect Discovery 1.0", August 2015, <http://openid.net/specs/openid-connect-discovery-1_0.html>.
- [RFC6755] Campbell, B. and H. Tschofenig, "An IETF URN Sub-Namespace for OAuth", RFC 6755, DOI 10.17487/RFC6755, October 2012, <<https://www.rfc-editor.org/info/rfc6755>>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <<https://www.rfc-editor.org/info/rfc7515>>.

Appendix A. Acknowledgements

This specification was developed within the OAuth Working Group under the chairmanship of Hannes Tschofenig and Rifaat Shekh-Yusef with Kathleen Moriarty, Eric Rescorla, and Benjamin Kaduk serving as Security Area Directors. Additionally, the following individuals contributed ideas, feedback, and wording that helped shape this specification: Dirk Balfanz, Andrei Popov, Justin Richer, and Nat Sakimura.

Appendix B. Document History

[[to be removed by the RFC Editor before publication as an RFC]]

-08

- o Update reference to -03 of openid-connect-token-bound-authentication.
- o Update the references to the core token binding specs, which are now RFCs 8471, 8472, and 8473.
- o Update reference to AS metadata, which is now RFC 8414.
- o Add chairs and ADs to the Acknowledgements.

-07

- o Explicitly state that the base64url encoding of the tbh value doesn't include any trailing pad characters, line breaks, whitespace, etc.
- o Update to latest references for tokbind drafts and draft-ietf-oauth-discovery.
- o Update reference to Implementation Considerations in draft-ietf-tokbind-https, which is section 6 rather than 5.
- o Try to tweak text that references specific sections in other documents so that the HTML generated by the ietf tools doesn't link to the current document (based on old suggestion from Barry <https://www.ietf.org/mail-archive/web/jose/current/msg04571.html>).

-06

- o Use the boilerplate from RFC 8174.
- o Update reference for draft-ietf-tokbind-https to -12 and draft-ietf-oauth-discovery to -09.
- o Minor editorial fixes.

-05

- o State that authorization servers should not token bind refresh tokens issued to a client that doesn't support bound refresh tokens, which can be indicated by the "client_refresh_token_token_binding_supported" client metadata parameter.
- o Add Token Binding for JWT Authorization Grants and JWT Client Authentication.
- o Adjust the language around aborting authorizations in Phasing in Token Binding to be somewhat more general and not only about downgrades.
- o Remove reference to, and usage of, 'OAuth 2.0 Protected Resource Metadata', which is no longer a going concern.
- o Moved "Token Binding Metadata" section before "Token Binding for Authorization Codes" to be closer to the "Token Binding for Access Tokens" and "Token Binding for Refresh Tokens", to which it is more closely related.
- o Update references for draft-ietf-tokbind- negotiation(-10), protocol(-16), and https(-10), as well as draft-ietf-oauth-discovery(-07), and BCP212/RFC8252 OAuth 2.0 for Native Apps.

-04

- o Define how to convey token binding information of an access token via RFC 7662 OAuth 2.0 Token Introspection (note that the Introspection Response Registration request for cnf/Confirmation is in <https://tools.ietf.org/html/draft-ietf-oauth-mtls-02#section-4.3> which will likely be published and registered prior to this document).
- o Minor editorial fixes.
- o Added an open issue about needing to allow for web server clients to opt-out of having refresh tokens bound while still allowing for binding of access tokens (following from mention of the problem on

slide 16 of the presentation from Chicago
<https://www.ietf.org/proceedings/98/slides/slides-98-oauth-sessb-token-binding-00.pdf>).

-03

- o Fix a few mistakes in and around the examples that were noticed preparing the slides for IETF 98 Chicago.

-02

- o Added a section on Token Binding for authorization codes with one variation for native clients and one for web server clients.
- o Updated language to reflect that the binding is to the token binding key pair and that proof-of-possession of that key is done on the TLS connection.
- o Added a bunch of examples.
- o Added a few Open Issues so they are tracked in the document.
- o Updated the Token Binding and OAuth Metadata references.
- o Added William Denniss as an author.

-01

- o Changed Token Binding for access tokens to use the Referred Token Binding ID, now that the Implementation Considerations in the Token Binding HTTPS specification make it clear that implementations will enable using the Referred Token Binding ID.
- o Defined Protected Resource Metadata value.
- o Changed to use the more specific term "protected resource" instead of "resource server".

-00

- o Created the initial working group version from draft-jones-oauth-token-binding-00.

Authors' Addresses

Michael B. Jones
Microsoft

Email: mbj@microsoft.com
URI: <http://self-issued.info/>

Brian Campbell
Ping Identity

Email: brian.d.campbell@gmail.com

John Bradley
Yubico

Email: ve7jtb@ve7jtb.com
URI: <http://www.thread-safe.com/>

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

Email: wdenniss@google.com
URI: <http://wdenniss.com/>

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 21, 2020

M. Jones
A. Nadalin
Microsoft
B. Campbell, Ed.
Ping Identity
J. Bradley
Yubico
C. Mortimore
Salesforce
July 20, 2019

OAuth 2.0 Token Exchange
draft-ietf-oauth-token-exchange-19

Abstract

This specification defines a protocol for an HTTP- and JSON- based Security Token Service (STS) by defining how to request and obtain security tokens from OAuth 2.0 authorization servers, including security tokens employing impersonation and delegation.

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 <https://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 January 21, 2020.

Copyright Notice

Copyright (c) 2019 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
1.1. Delegation vs. Impersonation Semantics	4
1.2. Requirements Notation and Conventions	6
1.3. Terminology	6
2. Token Exchange Request and Response	6
2.1. Request	6
2.1.1. Relationship Between Resource, Audience and Scope . .	9
2.2. Response	9
2.2.1. Successful Response	9
2.2.2. Error Response	11
2.3. Example Token Exchange	11
3. Token Type Identifiers	13
4. JSON Web Token Claims and Introspection Response Parameters .	15
4.1. "act" (Actor) Claim	15
4.2. "scope" (Scopes) Claim	17
4.3. "client_id" (Client Identifier) Claim	18
4.4. "may_act" (Authorized Actor) Claim	18
5. Security Considerations	19
6. Privacy Considerations	20
7. IANA Considerations	20
7.1. OAuth URI Registration	20
7.1.1. Registry Contents	20
7.2. OAuth Parameters Registration	21
7.2.1. Registry Contents	21
7.3. OAuth Access Token Type Registration	22
7.3.1. Registry Contents	22
7.4. JSON Web Token Claims Registration	22
7.4.1. Registry Contents	22
7.5. OAuth Token Introspection Response Registration	23
7.5.1. Registry Contents	23
7.6. OAuth Extensions Error Registration	23
7.6.1. Registry Contents	23
8. References	24
8.1. Normative References	24
8.2. Informative References	24
Appendix A. Additional Token Exchange Examples	26
A.1. Impersonation Token Exchange Example	26
A.1.1. Token Exchange Request	26
A.1.2. Subject Token Claims	26
A.1.3. Token Exchange Response	27

A.1.4. Issued Token Claims	27
A.2. Delegation Token Exchange Example	28
A.2.1. Token Exchange Request	28
A.2.2. Subject Token Claims	29
A.2.3. Actor Token Claims	29
A.2.4. Token Exchange Response	29
A.2.5. Issued Token Claims	30
Appendix B. Acknowledgements	31
Appendix C. Document History	31
Authors' Addresses	35

1. Introduction

A security token is a set of information that facilitates the sharing of identity and security information in heterogeneous environments or across security domains. Examples of security tokens include JSON Web Tokens (JWTs) [JWT] and SAML 2.0 Assertions [OASIS.saml-core-2.0-os]. Security tokens are typically signed to achieve integrity and sometimes also encrypted to achieve confidentiality. Security tokens are also sometimes described as Assertions, such as in [RFC7521].

A Security Token Service (STS) is a service capable of validating security tokens provided to it and issuing new security tokens in response, which enables clients to obtain appropriate access credentials for resources in heterogeneous environments or across security domains. Web Service clients have used WS-Trust [WS-Trust] as the protocol to interact with an STS for token exchange. While WS-Trust uses XML and SOAP, the trend in modern Web development has been towards RESTful patterns and JSON. The OAuth 2.0 Authorization Framework [RFC6749] and OAuth 2.0 Bearer Tokens [RFC6750] have emerged as popular standards for authorizing third-party applications' access to HTTP and RESTful resources. The conventional OAuth 2.0 interaction involves the exchange of some representation of resource owner authorization for an access token, which has proven to be an extremely useful pattern in practice. However, its input and output are somewhat too constrained as is to fully accommodate a security token exchange framework.

This specification defines a protocol extending OAuth 2.0 that enables clients to request and obtain security tokens from authorization servers acting in the role of an STS. Similar to OAuth 2.0, this specification focuses on client developer simplicity and requires only an HTTP client and JSON parser, which are nearly universally available in modern development environments. The STS protocol defined in this specification is not itself RESTful (an STS doesn't lend itself particularly well to a REST approach) but does

utilize communication patterns and data formats that should be familiar to developers accustomed to working with RESTful systems.

A new grant type for a token exchange request and the associated specific parameters for such a request to the token endpoint are defined by this specification. A token exchange response is a normal OAuth 2.0 response from the token endpoint with a few additional parameters defined herein to provide information to the client.

The entity that makes the request to exchange tokens is considered the client in the context of the token exchange interaction. However, that does not restrict usage of this profile to traditional OAuth clients. An OAuth resource server, for example, might assume the role of the client during token exchange in order to trade an access token that it received in a protected resource request for a new token that is appropriate to include in a call to a backend service. The new token might be an access token that is more narrowly scoped for the downstream service or it could be an entirely different kind of token.

The scope of this specification is limited to the definition of a basic request-and-response protocol for an STS-style token exchange utilizing OAuth 2.0. Although a few new JWT claims are defined that enable delegation semantics to be expressed, the specific syntax, semantics and security characteristics of the tokens themselves (both those presented to the authorization server and those obtained by the client) are explicitly out of scope and no requirements are placed on the trust model in which an implementation might be deployed. Additional profiles may provide more detailed requirements around the specific nature of the parties and trust involved, such as whether signing and/or encryption of tokens is needed or if proof-of-possession style tokens will be required or issued; however, such details will often be policy decisions made with respect to the specific needs of individual deployments and will be configured or implemented accordingly.

The security tokens obtained may be used in a number of contexts, the specifics of which are also beyond the scope of this specification.

1.1. Delegation vs. Impersonation Semantics

One common use case for an STS (as alluded to in the previous section) is to allow a resource server A to make calls to a backend service C on behalf of the requesting user B. Depending on the local site policy and authorization infrastructure, it may be desirable for A to use its own credentials to access C along with an annotation of some form that A is acting on behalf of B ("delegation"), or for A to be granted a limited access credential to C but that continues to

identify B as the authorized entity ("impersonation"). Delegation and impersonation can be useful concepts in other scenarios involving multiple participants as well.

When principal A impersonates principal B, A is given all the rights that B has within some defined rights context and is indistinguishable from B in that context. Thus, when principal A impersonates principal B, then insofar as any entity receiving such a token is concerned, they are actually dealing with B. It is true that some members of the identity system might have awareness that impersonation is going on, but it is not a requirement. For all intents and purposes, when A is impersonating B, A is B within the context of the rights authorized by the token. A's ability to impersonate B could be limited in scope or time, or even with a one-time-use restriction, whether via the contents of the token or an out-of-band mechanism.

Delegation semantics are different than impersonation semantics, though the two are closely related. With delegation semantics, principal A still has its own identity separate from B and it is explicitly understood that while B may have delegated some of its rights to A, any actions taken are being taken by A representing B. In a sense, A is an agent for B.

Delegation and impersonation are not inclusive of all situations. When a principal is acting directly on its own behalf, for example, neither delegation nor impersonation are in play. They are, however, the more common semantics operating for token exchange and, as such, are given more direct treatment in this specification.

Delegation semantics are typically expressed in a token by including information about both the primary subject of the token as well as the actor to whom that subject has delegated some of its rights. Such a token is sometimes referred to as a composite token because it is composed of information about multiple subjects. Typically, in the request, the "subject_token" represents the identity of the party on behalf of whom the token is being requested while the "actor_token" represents the identity of the party to whom the access rights of the issued token are being delegated. A composite token issued by the authorization server will contain information about both parties. When and if a composite token is issued is at the discretion of the authorization server and applicable policy and configuration.

The specifics of representing a composite token and even whether or not such a token will be issued depend on the details of the implementation and the kind of token. The representations of composite tokens that are not JWTs are beyond the scope of this

specification. The "actor_token" request parameter, however, does provide a means for providing information about the desired actor and the JWT "act" claim can provide a representation of a chain of delegation.

1.2. Requirements Notation and 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.

1.3. Terminology

This specification uses the terms "access token type", "authorization server", "client", "client identifier", "resource server", "token endpoint", "token request", and "token response" defined by OAuth 2.0 [RFC6749], and the terms "Base64url Encoding", "Claim", and "JWT Claims Set" defined by JSON Web Token (JWT) [JWT].

2. Token Exchange Request and Response

2.1. Request

A client requests a security token by making a token request to the authorization server's token endpoint using the extension grant type mechanism defined in Section 4.5 of [RFC6749].

Client authentication to the authorization server is done using the normal mechanisms provided by OAuth 2.0. Section 2.3.1 of [RFC6749] defines password-based authentication of the client, however, client authentication is extensible and other mechanisms are possible. For example, [RFC7523] defines client authentication using bearer JSON Web Tokens (JWTs) [JWT]. The supported methods of client authentication and whether or not to allow unauthenticated or unidentified clients are deployment decisions that are at the discretion of the authorization server. Note that omitting client authentication allows for a compromised token to be leveraged via an STS into other tokens by anyone possessing the compromised token. Thus client authentication allows for additional authorization checks by the STS as to which entities are permitted to impersonate or receive delegations from other entities.

The client makes a token exchange request to the token endpoint with an extension grant type using the HTTP "POST" method. The following parameters are included in the HTTP request entity-body using the

"application/x-www-form-urlencoded" format with a character encoding of UTF-8 as described in Appendix B of RFC6749 [RFC6749].

grant_type

REQUIRED. The value "urn:ietf:params:oauth:grant-type:token-exchange" indicates that a token exchange is being performed.

resource

OPTIONAL. A URI that indicates the target service or resource where the client intends to use the requested security token. This enables the authorization server to apply policy as appropriate for the target, such as determining the type and content of the token to be issued or if and how the token is to be encrypted. In many cases, a client will not have knowledge of the logical organization of the systems with which it interacts and will only know a URI of the service where it intends to use the token. The "resource" parameter allows the client to indicate to the authorization server where it intends to use the issued token by providing the location, typically as an https URL, in the token exchange request in the same form that will be used to access that resource. The authorization server will typically have the capability to map from a resource URI value to an appropriate policy. The value of the "resource" parameter MUST be an absolute URI, as specified by Section 4.3 of [RFC3986], which MAY include a query component and MUST NOT include a fragment component. Multiple "resource" parameters may be used to indicate that the issued token is intended to be used at the multiple resources listed. See [I-D.ietf-oauth-resource-indicators] for additional background and uses of the "resource" parameter.

audience

OPTIONAL. The logical name of the target service where the client intends to use the requested security token. This serves a purpose similar to the "resource" parameter, but with the client providing a logical name for the target service. Interpretation of the name requires that the value be something that both the client and the authorization server understand. An OAuth client identifier, a SAML entity identifier [OASIS.saml-core-2.0-os], an OpenID Connect Issuer Identifier [OpenID.Core], are examples of things that might be used as "audience" parameter values. However, "audience" values used with a given authorization server must be unique within that server, to ensure that they are properly interpreted as the intended type of value. Multiple "audience" parameters may be used to indicate that the issued token is intended to be used at the multiple audiences listed. The "audience" and "resource" parameters may be used together to indicate multiple target services with a mix of logical names and resource URIs.

scope

OPTIONAL. A list of space-delimited, case-sensitive strings, as defined in Section 3.3 of [RFC6749], that allow the client to specify the desired scope of the requested security token in the context of the service or resource where the token will be used. The values and associated semantics of scope are service specific and expected to be described in the relevant service documentation.

requested_token_type

OPTIONAL. An identifier, as described in Section 3, for the type of the requested security token. If the requested type is unspecified, the issued token type is at the discretion of the authorization server and may be dictated by knowledge of the requirements of the service or resource indicated by the "resource" or "audience" parameter.

subject_token

REQUIRED. A security token that represents the identity of the party on behalf of whom the request is being made. Typically, the subject of this token will be the subject of the security token issued in response to the request.

subject_token_type

REQUIRED. An identifier, as described in Section 3, that indicates the type of the security token in the "subject_token" parameter.

actor_token

OPTIONAL. A security token that represents the identity of the acting party. Typically, this will be the party that is authorized to use the requested security token and act on behalf of the subject.

actor_token_type

An identifier, as described in Section 3, that indicates the type of the security token in the "actor_token" parameter. This is REQUIRED when the "actor_token" parameter is present in the request but MUST NOT be included otherwise.

In processing the request, the authorization server MUST perform the appropriate validation procedures for the indicated token type and, if the actor token is present, also perform the appropriate validation procedures for its indicated token type. The validity criteria and details of any particular token are beyond the scope of this document and are specific to the respective type of token and its content.

In the absence of one-time-use or other semantics specific to the token type, the act of performing a token exchange has no impact on the validity of the subject token or actor token. Furthermore, the exchange is a one-time event and does not create a tight linkage between the input and output tokens, so that (for example) while the expiration time of the output token may be influenced by that of the input token, renewal or extension of the input token is not expected to be reflected in the output token's properties. It may still be appropriate or desirable to propagate token revocation events. However, doing so is not a general property of the STS protocol and would be specific to a particular implementation, token type or deployment.

2.1.1. Relationship Between Resource, Audience and Scope

When requesting a token, the client can indicate the desired target service(s) where it intends to use that token by way of the "audience" and "resource" parameters, as well as indicating the desired scope of the requested token using the "scope" parameter. The semantics of such a request are that the client is asking for a token with the requested scope that is usable at all the requested target services. Effectively, the requested access rights of the token are the cartesian product of all the scopes at all the target services.

An authorization server may be unwilling or unable to fulfill any token request but the likelihood of an unfulfillable request is significantly higher when very broad access rights are being solicited. As such, in the absence of specific knowledge about the relationship of systems in a deployment, clients should exercise discretion in the breadth of the access requested, particularly the number of target services. An authorization server can use the "invalid_target" error code, defined in Section 2.2.2, to inform a client that it requested access to too many target services simultaneously.

2.2. Response

The authorization server responds to a token exchange request with a normal OAuth 2.0 response from the token endpoint, as specified in Section 5 of [RFC6749]. Additional details and explanation are provided in the following subsections.

2.2.1. Successful Response

If the request is valid and meets all policy and other criteria of the authorization server, a successful token response is constructed by adding the following parameters to the entity-body of the HTTP

response using the "application/json" media type, as specified by [RFC8259], and an HTTP 200 status code. The parameters are serialized into a JavaScript Object Notation (JSON) structure by adding each parameter at the top level. Parameter names and string values are included as JSON strings. Numerical values are included as JSON numbers. The order of parameters does not matter and can vary.

access_token

REQUIRED. The security token issued by the authorization server in response to the token exchange request. The "access_token" parameter from Section 5.1 of [RFC6749] is used here to carry the requested token, which allows this token exchange protocol to use the existing OAuth 2.0 request and response constructs defined for the token endpoint. The identifier "access_token" is used for historical reasons and the issued token need not be an OAuth access token.

issued_token_type

REQUIRED. An identifier, as described in Section 3, for the representation of the issued security token.

token_type

REQUIRED. A case-insensitive value specifying the method of using the access token issued, as specified in Section 7.1 of [RFC6749]. It provides the client with information about how to utilize the access token to access protected resources. For example, a value of "Bearer", as specified in [RFC6750], indicates that the issued security token is a bearer token and the client can simply present it as is without any additional proof of eligibility beyond the contents of the token itself. Note that the meaning of this parameter is different from the meaning of the "issued_token_type" parameter, which declares the representation of the issued security token; the term "token type" is more typically used with the aforementioned meaning as the structural or syntactical representation of the security token, as it is in all "*_token_type" parameters in this specification. If the issued token is not an access token or usable as an access token, then the "token_type" value "N_A" is used to indicate that an OAuth 2.0 "token_type" identifier is not applicable in that context.

expires_in

RECOMMENDED. The validity lifetime, in seconds, of the token issued by the authorization server. Oftentimes the client will not have the inclination or capability to inspect the content of the token and this parameter provides a consistent and token-type-agnostic indication of how long the token can be expected to be

valid. For example, the value 1800 denotes that the token will expire in thirty minutes from the time the response was generated.

scope

OPTIONAL, if the scope of the issued security token is identical to the scope requested by the client; otherwise, REQUIRED.

refresh_token

OPTIONAL. A refresh token will typically not be issued when the exchange is of one temporary credential (the subject_token) for a different temporary credential (the issued token) for use in some other context. A refresh token can be issued in cases where the client of the token exchange needs the ability to access a resource even when the original credential is no longer valid (e.g., user-not-present or offline scenarios where there is no longer any user entertaining an active session with the client). Profiles or deployments of this specification should clearly document the conditions under which a client should expect a refresh token in response to "urn:ietf:params:oauth:grant-type:token-exchange" grant type requests.

2.2.2. Error Response

If the request itself is not valid or if either the "subject_token" or "actor_token" are invalid for any reason, or are unacceptable based on policy, the authorization server MUST construct an error response, as specified in Section 5.2 of [RFC6749]. The value of the "error" parameter MUST be the "invalid_request" error code.

If the authorization server is unwilling or unable to issue a token for any target service indicated by the "resource" or "audience" parameters, the "invalid_target" error code SHOULD be used in the error response.

The authorization server MAY include additional information regarding the reasons for the error using the "error_description" as discussed in Section 5.2 of [RFC6749].

Other error codes may also be used, as appropriate.

2.3. Example Token Exchange

The following example demonstrates a hypothetical token exchange in which an OAuth resource server assumes the role of the client during the exchange. It trades an access token, which it received in a protected resource request, for a new token that it will use to call to a backend service (extra line breaks and indentation in the examples are for display purposes only).

Figure 1 shows the resource server receiving a protected resource request containing an OAuth access token in the Authorization header, as specified in Section 2.1 of [RFC6750].

```
GET /resource HTTP/1.1
Host: frontend.example.com
Authorization: Bearer accVkjcJyb4BWCxGsndESCJQbdfMogUC5PbRDqceLTC
```

Figure 1: Protected Resource Request

In Figure 2, the resource server assumes the role of client for the token exchange and the access token from the request in Figure 1 is sent to the authorization server using a request as specified in Section 2.1. The value of the "subject_token" parameter carries the access token and the value of the "subject_token_type" parameter indicates that it is an OAuth 2.0 access token. The resource server, acting in the role of the client, uses its identifier and secret to authenticate to the authorization server using the HTTP Basic authentication scheme. The "resource" parameter indicates the location of the backend service, `https://backend.example.com/api`, where the issued token will be used.

```
POST /as/token.oauth2 HTTP/1.1
Host: as.example.com
Authorization: Basic cnMwODpsb25nLXNlY3VyZS1yYW5kb20tc2VjcmV0
Content-Type: application/x-www-form-urlencoded

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Atoken-exchange
&resource=https%3A%2F%2Fbackend.example.com%2Fapi
&subject_token=accVkjcJyb4BWCxGsndESCJQbdfMogUC5PbRDqceLTC
&subject_token_type=
urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Aaccess_token
```

Figure 2: Token Exchange Request

The authorization server validates the client credentials and the "subject_token" presented in the token exchange request. From the "resource" parameter, the authorization server is able to determine the appropriate policy to apply to the request and issues a token suitable for use at `https://backend.example.com`. The "access_token" parameter of the response shown in Figure 3 contains the new token, which is itself a bearer OAuth access token that is valid for one minute. The token happens to be a JWT; however, its structure and format are opaque to the client so the "issued_token_type" indicates only that it is an access token.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-cache, no-store

{
  "access_token": "eyJhbGciOiJIJFUzI1NiIsImtpZCI6IjllciJ9.eyJhdWQiOiJo  
dHRwczovL2JhY2tlbmQuZXhhbXBsZS5jb20iLCJpc3MiOiJodHRwczovL2FzMV  
4YWlwbGUuY29tIiwiaXNjaXhwIjoibDQxOTE3NTkzLCJpYXQiOjE0NDcMTclMzMsIn  
biwiIjoiImJkY0bleGFtcGxlLnNvbSIscnNja3BlIjoieXBpIn0.40y3ZgQedw6rx  
f59WlwHDD9jryFor0_Wh3CGozQBihNBhnXEQG8AI9x3KmsPottVMLPIWvmDCM  
y5-kdXjwhw",
  "issued_token_type":  
    "urn:iETF:params:oAuth:token-type:access_token",
  "token_type": "Bearer",
  "expires_in": 60
}
```

Figure 3: Token Exchange Response

The resource server can then use the newly acquired access token in making a request to the backend server as illustrated in Figure 4.

```
GET /api HTTP/1.1
Host: backend.example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIsImtpZCI6IjllciJ9.eyJhdWQiOiJodHRwczovL2JhY2t1bmQuZXhhbXBsZS5jb20iLCJpc3MiOiJodHRwczovL2FzLmV4YW1wbGUuYy29tIiwiaXNjaWJoxNDQxOTE3NTkzLCJpYXQiOiJEONDE5MTc1MzMsInN1YiI6ImUuYy29tIiwiaWF0IjoiYXBpLn0.40y3ZgQedw6rxsf59lwhDD9jrYfOrO_Wh3CGozQBihNBhnXEQgU85AI9x3KmsPottVMLPIWvmDCMy5-kdXjwhw
```

Figure 4: Backend Protected Resource Request

Additional examples can be found in Appendix A.

3. Token Type Identifiers

Several parameters in this specification utilize an identifier as the value to describe the token in question. Specifically, they are the "requested_token_type", "subject_token_type", "actor_token_type" parameters of the request and the "issued_token_type" member of the response. Token type identifiers are URIs. Token Exchange can work with both tokens issued by other parties and tokens from the given authorization server. For the former the token type identifier indicates the syntax (e.g., JWT or SAML 2.0) so the authorization server can parse it; for the latter it indicates what the given authorization server issued it for (e.g., access_token or refresh token).

The following token type identifiers are defined by this specification. Other URIs MAY be used to indicate other token types.

`urn:ietf:params:oauth:token-type:access_token`

Indicates that the token is an OAuth 2.0 access token issued by the given authorization server.

`urn:ietf:params:oauth:token-type:refresh_token`

Indicates that the token is an OAuth 2.0 refresh token issued by the given authorization server.

`urn:ietf:params:oauth:token-type:id_token`

Indicates that the token is an ID Token, as defined in Section 2 of [OpenID.Core].

`urn:ietf:params:oauth:token-type:saml1`

Indicates that the token is a base64url-encoded SAML 1.1 [OASIS.saml-core-1.1] assertion.

`urn:ietf:params:oauth:token-type:saml2`

Indicates that the token is a base64url-encoded SAML 2.0 [OASIS.saml-core-2.0-os] assertion.

The value `"urn:ietf:params:oauth:token-type:jwt"`, which is defined in Section 9 of [JWT], indicates that the token is a JWT.

The distinction between an access token and a JWT is subtle. An access token represents a delegated authorization decision, whereas JWT is a token format. An access token can be formatted as a JWT but doesn't necessarily have to be. And a JWT might well be an access token but not all JWTs are access tokens. The intent of this specification is that `"urn:ietf:params:oauth:token-type:access_token"` be an indicator that the token is a typical OAuth access token issued by the authorization server in question, opaque to the client, and usable the same manner as any other access token obtained from that authorization server. (It could well be a JWT, but the client isn't and needn't be aware of that fact.) Whereas, `"urn:ietf:params:oauth:token-type:jwt"` is to indicate specifically that a JWT is being requested or sent (perhaps in a cross-domain use-case where the JWT is used as an authorization grant to obtain an access token from a different authorization server as is facilitated by [RFC7523]).

Note that for tokens which are binary in nature, the URI used for conveying them needs to be associated with the semantics of a base64 or other encoding suitable for usage with HTTP and OAuth.

4. JSON Web Token Claims and Introspection Response Parameters

It is useful to have defined mechanisms to express delegation within a token as well as to express authorization to delegate or impersonate. Although the token exchange protocol described herein can be used with any type of token, this section defines claims to express such semantics specifically for JWTs and in an OAuth 2.0 Token Introspection [RFC7662] response. Similar definitions for other types of tokens are possible but beyond the scope of this specification.

Note that the claims not established herein but used in examples and descriptions, such as "iss", "sub", "exp", etc., are defined by [JWT].

4.1. "act" (Actor) Claim

The "act" (actor) claim provides a means within a JWT to express that delegation has occurred and identify the acting party to whom authority has been delegated. The "act" claim value is a JSON object and members in the JSON object are claims that identify the actor. The claims that make up the "act" claim identify and possibly provide additional information about the actor. For example, the combination of the two claims "iss" and "sub" might be necessary to uniquely identify an actor.

However, claims within the "act" claim pertain only to the identity of the actor and are not relevant to the validity of the containing JWT in the same manner as the top-level claims. Consequently, non-identity claims (e.g., "exp", "nbf", and "aud") are not meaningful when used within an "act" claim, and therefore are not used.

Figure 5 illustrates the "act" (actor) claim within a JWT Claims Set. The claims of the token itself are about user@example.com while the "act" claim indicates that admin@example.com is the current actor.

```
{
  "aud": "https://consumer.example.com",
  "iss": "https://issuer.example.com",
  "exp": 1443904177,
  "nbf": 1443904077,
  "sub": "user@example.com",
  "act": {
    "sub": "admin@example.com"
  }
}
```

Figure 5: Actor Claim

A chain of delegation can be expressed by nesting one "act" claim within another. The outermost "act" claim represents the current actor while nested "act" claims represent prior actors. The least recent actor is the most deeply nested. The nested "act" claims serve as a history trail that connects the initial request and subject through the various delegation steps undertaken before reaching the current actor. In this sense, the current actor is considered to include the entire authorization/delegation history, leading naturally to the nested structure described here.

For the purpose of applying access control policy, the consumer of a token MUST only consider the token's top-level claims and the party identified as the current actor by the "act" claim. Prior actors identified by any nested "act" claims are informational only and are not to be considered in access control decisions.

The following example in Figure 6 illustrates nested "act" (actor) claims within a JWT Claims Set. The claims of the token itself are about user@example.com while the "act" claim indicates that the system https://service16.example.com is the current actor and https://service77.example.com was a prior actor. Such a token might come about as the result of service16 receiving a token in a call from service77 and exchanging it for a token suitable to call service26 while the authorization server notes the situation in the newly issued token.

```
{
  "aud": "https://service26.example.com",
  "iss": "https://issuer.example.com",
  "exp": 1443904100,
  "nbf": 1443904000,
  "sub": "user@example.com",
  "act": {
    {
      "sub": "https://service16.example.com",
      "act": {
        {
          "sub": "https://service77.example.com"
        }
      }
    }
  }
}
```

Figure 6: Nested Actor Claim

When included as a top-level member of an OAuth token introspection response, "act" has the same semantics and format as the claim of the same name.

4.2. "scope" (Scopes) Claim

The value of the "scope" claim is a JSON string containing a space-separated list of scopes associated with the token, in the format described in Section 3.3 of [RFC6749].

Figure 7 illustrates the "scope" claim within a JWT Claims Set.

```
{
  "aud": "https://consumer.example.com",
  "iss": "https://issuer.example.com",
  "exp": 1443904177,
  "nbf": 1443904077,
  "sub": "dgaf4mvfs75Fci_FL3heQA",
  "scope": "email profile phone address"
}
```

Figure 7: Scopes Claim

OAuth 2.0 Token Introspection [RFC7662] already defines the "scope" parameter to convey the scopes associated with the token.

4.3. "client_id" (Client Identifier) Claim

The "client_id" claim carries the client identifier of the OAuth 2.0 [RFC6749] client that requested the token.

The following example in Figure 8 illustrates the "client_id" claim within a JWT Claims Set indicating an OAuth 2.0 client with "s6BhdRkqt3" as its identifier.

```
{
  "aud": "https://consumer.example.com",
  "iss": "https://issuer.example.com",
  "exp": 1443904177,
  "sub": "user@example.com",
  "client_id": "s6BhdRkqt3"
}
```

Figure 8: Client Identifier Claim

OAuth 2.0 Token Introspection [RFC7662] already defines the "client_id" parameter as the client identifier for the OAuth 2.0 client that requested the token.

4.4. "may_act" (Authorized Actor) Claim

The "may_act" claim makes a statement that one party is authorized to become the actor and act on behalf of another party. The claim might be used, for example, when a "subject_token" is presented to the token endpoint in a token exchange request and "may_act" claim in the subject token can be used by the authorization server to determine whether the client (or party identified in the "actor_token") is authorized to engage in the requested delegation or impersonation.

The claim value is a JSON object and members in the JSON object are claims that identify the party that is asserted as being eligible to act for the party identified by the JWT containing the claim. The claims that make up the "may_act" claim identify and possibly provide additional information about the authorized actor. For example, the combination of the two claims "iss" and "sub" are sometimes necessary to uniquely identify an authorized actor, while the "email" claim might be used to provide additional useful information about that party.

However, claims within the "may_act" claim pertain only to the identity of that party and are not relevant to the validity of the containing JWT in the same manner as top-level claims. Consequently, claims such as "exp", "nbf", and "aud" are not meaningful when used within a "may_act" claim, and therefore are not used.

Figure 9 illustrates the "may_act" claim within a JWT Claims Set. The claims of the token itself are about user@example.com while the "may_act" claim indicates that admin@example.com is authorized to act on behalf of user@example.com.

```
{
  "aud": "https://consumer.example.com",
  "iss": "https://issuer.example.com",
  "exp": 1443904177,
  "nbf": 1443904077,
  "sub": "user@example.com",
  "may_act": {
    "sub": "admin@example.com"
  }
}
```

Figure 9: Authorized Actor Claim

When included as a top-level member of an OAuth token introspection response, "may_act" has the same semantics and format as the claim of the same name.

5. Security Considerations

Much of the guidance from Section 10 of [RFC6749], the Security Considerations in The OAuth 2.0 Authorization Framework, is also applicable here. Furthermore, [RFC6819] provides additional security considerations for OAuth and [I-D.ietf-oauth-security-topics] has updated security guidance based on deployment experience and new threats that have emerged since OAuth 2.0 was originally published.

All of the normal security issues that are discussed in [JWT], especially in relationship to comparing URIs and dealing with unrecognized values, also apply here.

In addition, both delegation and impersonation introduce unique security issues. Any time one principal is delegated the rights of another principal, the potential for abuse is a concern. The use of the "scope" claim (in addition to other typical constraints such as a limited token lifetime) is suggested to mitigate potential for such abuse, as it restricts the contexts in which the delegated rights can be exercised.

6. Privacy Considerations

Tokens employed in the context of the functionality described herein may contain privacy-sensitive information and, to prevent disclosure of such information to unintended parties, MUST only be transmitted over encrypted channels, such as Transport Layer Security (TLS). In cases where it is desirable to prevent disclosure of certain information to the client, the token MUST be encrypted to its intended recipient. Deployments SHOULD determine the minimally necessary amount of data and only include such information in issued tokens. In some cases, data minimization may include representing only an anonymous or pseudonymous user.

7. IANA Considerations

7.1. OAuth URI Registration

This specification registers the following values in the IANA "OAuth URI" registry [IANA.OAuth.Parameters] established by [RFC6755].

7.1.1. Registry Contents

- o URN: urn:ietf:params:oauth:grant-type:token-exchange
- o Common Name: Token exchange grant type for OAuth 2.0
- o Change controller: IESG
- o Specification Document: Section 2.1 of [[this specification]]

- o URN: urn:ietf:params:oauth:token-type:access_token
- o Common Name: Token type URI for an OAuth 2.0 access token
- o Change controller: IESG
- o Specification Document: Section 3 of [[this specification]]

- o URN: urn:ietf:params:oauth:token-type:refresh_token
- o Common Name: Token type URI for an OAuth 2.0 refresh token
- o Change controller: IESG
- o Specification Document: Section 3 of [[this specification]]

- o URN: urn:ietf:params:oauth:token-type:id_token
- o Common Name: Token type URI for an ID Token
- o Change controller: IESG
- o Specification Document: Section 3 of [[this specification]]

- o URN: urn:ietf:params:oauth:token-type:saml1
- o Common Name: Token type URI for a base64url-encoded SAML 1.1 assertion
- o Change Controller: IESG
- o Specification Document: Section 3 of [[this specification]]

- o URN: urn:ietf:params:oauth:token-type:saml2
- o Common Name: Token type URI for a base64url-encoded SAML 2.0 assertion
- o Change Controller: IESG
- o Specification Document: Section 3 of [[this specification]]

7.2. OAuth Parameters Registration

This specification registers the following values in the IANA "OAuth Parameters" registry [IANA.OAuth.Parameters] established by [RFC6749].

7.2.1. Registry Contents

- o Parameter name: resource
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification document(s): Section 2.1 of [[this specification]]

- o Parameter name: audience
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification document(s): Section 2.1 of [[this specification]]

- o Parameter name: requested_token_type
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification document(s): Section 2.1 of [[this specification]]

- o Parameter name: subject_token
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification document(s): Section 2.1 of [[this specification]]

- o Parameter name: subject_token_type
- o Parameter usage location: token request
- o Change controller: IESG

- o Specification document(s): Section 2.1 of [[this specification]]
- o Parameter name: actor_token
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification document(s): Section 2.1 of [[this specification]]
- o Parameter name: actor_token_type
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification document(s): Section 2.1 of [[this specification]]
- o Parameter name: issued_token_type
- o Parameter usage location: token response
- o Change controller: IESG
- o Specification document(s): Section 2.2.1 of [[this specification]]

7.3. OAuth Access Token Type Registration

This specification registers the following access token type in the IANA "OAuth Access Token Types" registry [IANA.OAuth.Parameters] established by [RFC6749].

7.3.1. Registry Contents

- o Type name: N_A
- o Additional Token Endpoint Response Parameters: (none)
- o HTTP Authentication Scheme(s): (none)
- o Change controller: IESG
- o Specification document(s): Section 2.2.1 of [[this specification]]

7.4. JSON Web Token Claims Registration

This specification registers the following Claims in the IANA "JSON Web Token Claims" registry [IANA.JWT.Claims] established by [JWT].

7.4.1. Registry Contents

- o Claim Name: "act"
- o Claim Description: Actor
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this specification]]
- o Claim Name: "scope"
- o Claim Description: Scope Values
- o Change Controller: IESG

- o Specification Document(s): Section 4.2 of [[this specification]]
- o Claim Name: "client_id"
- o Claim Description: Client Identifier
- o Change Controller: IESG
- o Specification Document(s): Section 4.3 of [[this specification]]
- o Claim Name: "may_act"
- o Claim Description: Authorized Actor - the party that is authorized to become the actor
- o Change Controller: IESG
- o Specification Document(s): Section 4.4 of [[this specification]]

7.5. OAuth Token Introspection Response Registration

This specification registers the following values in the IANA "OAuth Token Introspection Response" registry [IANA.OAuth.Parameters] established by [RFC7662].

7.5.1. Registry Contents

- o Claim Name: "act"
- o Claim Description: Actor
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this specification]]
- o Claim Name: "may_act"
- o Claim Description: Authorized Actor - the party that is authorized to become the actor
- o Change Controller: IESG
- o Specification Document(s): Section 4.4 of [[this specification]]

7.6. OAuth Extensions Error Registration

This specification registers the following values in the IANA "OAuth Extensions Error" registry [IANA.OAuth.Parameters] established by [RFC6749].

7.6.1. Registry Contents

- o Error Name: "invalid_target"
- o Error Usage Location: token error response
- o Related Protocol Extension: OAuth 2.0 Token Exchange
- o Change Controller: IETF
- o Specification Document(s): Section 2.2.2 of [[this specification]]

8. References

8.1. Normative References

- [IANA.JWT.Claims] IANA, "JSON Web Token Claims", <<http://www.iana.org/assignments/jwt>>.
- [IANA.OAuth.Parameters] IANA, "OAuth Parameters", <<http://www.iana.org/assignments/oauth-parameters>>.
- [JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <<http://tools.ietf.org/html/rfc7519>>.
- [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>>.
- [RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<https://www.rfc-editor.org/info/rfc7662>>.
- [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>>.
- [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/info/rfc8259>>.

8.2. Informative References

- [I-D.ietf-oauth-resource-indicators]
Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", draft-ietf-oauth-resource-indicators-02 (work in progress), January 2019.
- [I-D.ietf-oauth-security-topics]
Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett, "OAuth 2.0 Security Best Current Practice", draft-ietf-oauth-security-topics-13 (work in progress), July 2019.
- [OASIS.saml-core-1.1]
Maler, E., Mishra, P., and R. Philpott, "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V1.1", OASIS Standard oasis-sstc-saml-core-1.1, September 2003.
- [OASIS.saml-core-2.0-os]
Cantor, S., Kemp, J., Philpott, R., and E. Maler, "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard saml-core-2.0-os, March 2005.
- [OpenID.Core]
Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", November 2014, <http://openid.net/specs/openid-connect-core-1_0.html>.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <<https://www.rfc-editor.org/info/rfc6750>>.
- [RFC6755] Campbell, B. and H. Tschofenig, "An IETF URN Sub-Namespace for OAuth", RFC 6755, DOI 10.17487/RFC6755, October 2012, <<https://www.rfc-editor.org/info/rfc6755>>.
- [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>>.
- [RFC7521] Campbell, B., Mortimore, C., Jones, M., and Y. Goland, "Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7521, DOI 10.17487/RFC7521, May 2015, <<https://www.rfc-editor.org/info/rfc7521>>.

[RFC7523] Jones, M., Campbell, B., and C. Mortimore, "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7523, DOI 10.17487/RFC7523, May 2015, <<https://www.rfc-editor.org/info/rfc7523>>.

[WS-Trust]

Nadalin, A., Goodner, M., Gudgin, M., Barbir, A., and H. Granqvist, "WS-Trust 1.4", February 2012, <<http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.html>>.

Appendix A. Additional Token Exchange Examples

Two example token exchanges are provided in the following sections illustrating impersonation and delegation, respectively (with extra line breaks and indentation for display purposes only).

A.1. Impersonation Token Exchange Example

A.1.1. Token Exchange Request

In the following token exchange request, a client is requesting a token with impersonation semantics (with only a "subject_token" and no "actor_token", delegation is impossible). The client tells the authorization server that it needs a token for use at the target service with the logical name "urn:example:cooperation-context".

```
POST /as/token.oauth2 HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Atoken-exchange
&audience=urn%3Aexample%3Acooperation-context
&subject_token=eyJhbGciOiJIUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwczovL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbC1pc3N1ZXIuZXhhbXBsZS5uZXQlLCJleHAiOjE0NDE5MTA2MDAsIm5iZiI6MTQ0MTkwOTAwMCwic3ViIjoiYmRjQGV4YW1wbGUubmV0Iiwic2NvcGUiOiJvcmlcnMgcHJvZmlsZSBoaXN0b3J5In0.PRBg-jXn4cJujlgmYXFiGkZzRuzbXZ_sDxdE98ddW44ufsbWLKd3JJ1VZhF64pbTtfjy4VXFVBdaQpKjn5JzAw
&subject_token_type=urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Ajwt
```

Figure 10: Token Exchange Request

A.1.2. Subject Token Claims

The "subject_token" in the prior request is a JWT and the decoded JWT Claims Set is shown here. The JWT is intended for consumption by the authorization server within a specific time window. The subject of

the JWT ("bdc@example.net") is the party on behalf of whom the new token is being requested.

```
{
  "aud": "https://as.example.com",
  "iss": "https://original-issuer.example.net",
  "exp": 1441910600,
  "nbf": 1441909000,
  "sub": "bdc@example.net",
  "scope": "orders profile history"
}
```

Figure 11: Subject Token Claims

A.1.3. Token Exchange Response

The "access_token" parameter of the token exchange response shown below contains the new token that the client requested. The other parameters of the response indicate that the token is a bearer access token that expires in an hour.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-cache, no-store
```

```
{
  "access_token": "eyJhbGciOiJIJFZlIiwiaXNzIjoiYmRjQGV4YW1wbGUubWV0Iiwic2NvcGUiOiJvcmlcnMgcHJvZmlsZSBoaXN0b3J5In0.rMdWpSGNACTvnFuOL74sYZ6MVuld2Z2WkGLmQeR9ztj6w2OXraQlkJmGjyiCq24kCB7AI2VqVx13wSWnVKh85A",
  "issued_token_type":
    "urn:ietf:params:oauth:token-type:access_token",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

Figure 12: Token Exchange Response

A.1.4. Issued Token Claims

The decoded JWT Claims Set of the issued token is shown below. The new JWT is issued by the authorization server and intended for consumption by a system entity known by the logical name "urn:example:cooperation-context" any time before its expiration. The subject ("sub") of the JWT is the same as the subject the token used to make the request, which effectively enables the client to

impersonate that subject at the system entity known by the logical name of "urn:example:cooperation-context" by using the token.

```
{
  "aud": "urn:example:cooperation-context",
  "iss": "https://as.example.com",
  "exp": 1441913610,
  "sub": "bdc@example.net",
  "scope": "orders profile history"
}
```

Figure 13: Issued Token Claims

A.2. Delegation Token Exchange Example

A.2.1. Token Exchange Request

In the following token exchange request, a client is requesting a token and providing both a "subject_token" and an "actor_token". The client tells the authorization server that it needs a token for use at the target service with the logical name "urn:example:cooperation-context". Policy at the authorization server dictates that the issued token be a composite.

```
POST /as/token.oauth2 HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Atoken-exchange
&audience=urn%3Aexample%3Acooperation-context
&subject_token=eyJhbGciOiJIUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwczovL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbC1pc3NlZXIuZSxhhbXBsZS5uZXQlLCJleHAiOjE0NDE5MTAwNjAsInNjb3BlIjoic3Rh dHVzIGZlZWQiLCJzdWIiOiJlc2VyQG V4YW1wbGUubmV0IiwibWF5X2FjdCI6eyJzdWIiOiJhZG1pbkBLEGFtcGxlLm5ldCJ9fQ.4rPRSWihQbpMIgAmAoqaJoJAxj-p2X8_fAtAGTXrvMxU-eEZhnXqY0_AOZgLdxw5DyLzua8H_I10MCcekF-Q_g
&subject_token_type=urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Ajwt
&actor_token=eyJhbGciOiJIUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwczovL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbC1pc3NlZXIuZSxhhbXBsZS5uZXQlLCJleHAiOjE0NDE5MTAwNjAsInN1YiI6ImFkbWluQG V4YW1wbGUubmV0In0.7YQ-3zpFfhUvzje5oqw8COcvN5uP6NsKik9CVV6cAO f4QKgM-tKfiOwcgZoUuD L2tEs6tqPlcB lMjiSzEjm3yBg
&actor_token_type=urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Ajwt
```

Figure 14: Token Exchange Request

A.2.2. Subject Token Claims

The "subject_token" in the prior request is a JWT and the decoded JWT Claims Set is shown here. The JWT is intended for consumption by the authorization server before a specific expiration time. The subject of the JWT ("user@example.net") is the party on behalf of whom the new token is being requested.

```
{
  "aud": "https://as.example.com",
  "iss": "https://original-issuer.example.net",
  "exp": 1441910060,
  "scope": "status feed",
  "sub": "user@example.net",
  "may_act": {
    "sub": "admin@example.net"
  }
}
```

Figure 15: Subject Token Claims

A.2.3. Actor Token Claims

The "actor_token" in the prior request is a JWT and the decoded JWT Claims Set is shown here. This JWT is also intended for consumption by the authorization server before a specific expiration time. The subject of the JWT ("admin@example.net") is the actor that will wield the security token being requested.

```
{
  "aud": "https://as.example.com",
  "iss": "https://original-issuer.example.net",
  "exp": 1441910060,
  "sub": "admin@example.net"
}
```

Figure 16: Actor Token Claims

A.2.4. Token Exchange Response

The "access_token" parameter of the token exchange response shown below contains the new token that the client requested. The other parameters of the response indicate that the token is a JWT that expires in an hour and that the access token type is not applicable since the issued token is not an access token.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-cache, no-store

{
  "access_token": "eyJhbGciOiJIJFUiI1NiIsImtpZCI6IjcyIn0.eyJhdWQiOiJ1cm46ZXhhdXBsZTpjb29wZXJhdGlvbi1jb250ZXh0IiwiaXNzIjoiaHR0cHM6Ly9hcy5leGFtcGxlLmNvbSIsImV4cCI6MTQ0MTkxMzYxMCwic2NvcGUiOiJzdGF0dXMgZmVlZCIsInN1YiI6InVzZXJAZXhhdXBsZS5uZXQlLCJhY3QiOiOnsic3ViIjojYWWRtaW5AZXhhdXBsZS5uZXQifX0.3paKl9UySKYB5ng6_cUtQ2ql08Rc_y7Mea7IwEXTcYbNdwG9-G1EKCFe5fw3H0hwX-MSZ49Wpcb1SiAZaOQBtw",
  "issued_token_type": "urn:ietf:params:oauth:token-type:jwt",
  "token_type": "N_A",
  "expires_in": 3600
}
```

Figure 17: Token Exchange Response

A.2.5. Issued Token Claims

The decoded JWT Claims Set of the issued token is shown below. The new JWT is issued by the authorization server and intended for consumption by a system entity known by the logical name "urn:example:cooperation-context" any time before its expiration. The subject ("sub") of the JWT is the same as the subject of the "subject_token" used to make the request. The actor ("act") of the JWT is the same as the subject of the "actor_token" used to make the request. This indicates delegation and identifies "admin@example.net" as the current actor to whom authority has been delegated to act on behalf of "user@example.net".

```
{
  "aud": "urn:example:cooperation-context",
  "iss": "https://as.example.com",
  "exp": 1441913610,
  "scope": "status feed",
  "sub": "user@example.net",
  "act":
  {
    "sub": "admin@example.net"
  }
}
```

Figure 18: Issued Token Claims

Appendix B. Acknowledgements

This specification was developed within the OAuth Working Group, which includes dozens of active and dedicated participants. It was produced under the chairmanship of Hannes Tschofenig, Derek Atkins, and Rifaat Shekh-Yusef with Kathleen Moriarty, Stephen Farrell, Eric Rescorla, Roman Danyliw, and Benjamin Kaduk serving as Security Area Directors. The following individuals contributed ideas, feedback, and wording to this specification:

Caleb Baker, Vittorio Bertocci, Mike Brown, Thomas Broyer, Roman Danyliw, William Denniss, Vladimir Dzhuvinov, Eric Fazendin, Phil Hunt, Benjamin Kaduk, Jason Keglovitz, Torsten Lodderstedt, Barry Leiba, Adam Lewis, James Manger, Nov Mataka, Matt Miller, Hilarie Orman, Matthew Perry, Eric Rescorla, Justin Richer, Adam Roach, Rifaat Shekh-Yusef, Scott Tomilson, and Hannes Tschofenig.

Appendix C. Document History

[[to be removed by the RFC Editor before publication as an RFC]]

-19

- o Fix-up changes introduced in -18.
- o Fix invalid JSON in the Nested Actor Claim example.
- o Reference figure numbers in text when introducing the examples in Section 2 and 4.
- o Editorial updates from additional IESG evaluation comments.
- o Add an informational reference to ietf-oauth-resource-indicators
- o Update ietf-oauth-security-topics ref to 13

-18

- o Editorial updates based on a few more IESG evaluation comments.

-17

- o Editorial improvements and example fixes resulting from IESG evaluation comments.
- o Added a pointer to RFC6749's Appendix B. on the "Use of application/x-www-form-urlencoded Media Type" as a way of providing a normative citation (by reference) for the media type.
- o Strengthened some of the wording in the privacy considerations to bring it inline with RFC 7519 Sec. 12 and RFC 6749 Sec. 10.8.

-16

- o Fixed typo and added an AD to Acknowledgements.

-15

- o Updated the nested actor claim example to (hopefully) be more straightforward.
- o Reworked Privacy Considerations to say to use TLS in transit, minimize the amount of information in the token, and encrypt the token if disclosure of its information to the client is a concern per https://mailarchive.ietf.org/arch/msg/secdir/KJhx4aq_U5uk3k6zpYP-CEHbpVM
- o Moved the Security and Privacy Considerations sections to before the IANA Considerations.

-14

- o Added text in Section 4.1 about the "act" claim stating that only the top-level claims and the current actor are to be considered in applying access control decisions.

-13

- o Updated the claim name and value syntax for scope to be consistent with the treatment of scope in RFC 7662 OAuth 2.0 Token Introspection.
- o Updated the client identifier claim name to be consistent with the treatment of client id in RFC 7662 OAuth 2.0 Token Introspection.

-12

- o Updated to use the boilerplate from RFC 8174.

-11

- o Added new WG chair and AD to the Acknowledgements.
- o Applied clarifications suggested during AD review by EKR.

-10

- o Defined token type URIs for base64url-encoded SAML 1.1 and SAML 2.0 assertions.
- o Applied editorial fixes.

-09

- o Changed "security tokens obtained could be used in a number of contexts" to "security tokens obtained may be used in a number of contexts" per a WGLC suggestion.

- o Clarified that the validity of the subject or actor token have no impact on the validity of the issued token after the exchange has occurred per a WGLC comment.
- o Changed use of `invalid_target` error code to a SHOULD per a WGLC comment.
- o Clarified text about non-identity claims within the "act" claim being meaningless per a WGLC comment.
- o Added brief Privacy Considerations section per WGLC comments.

-08

- o Use the bibxml reference for OpenID.Core rather than defining it inline.
- o Added editor role for Campbell.
- o Minor clarification of the text for `actor_token`.

-07

- o Fixed typo (desecration -> discretion).
- o Added an explanation of the relationship between scope, audience and resource in the request and added an "invalid_target" error code enabling the AS to tell the client that the requested audiences/resources were too broad.

-06

- o Drop "An STS for the REST of Us" from the title.
- o Drop "heavyweight" and "lightweight" from the abstract and introduction.
- o Clarifications on the language around `xxxxxx_token_type`.
- o Remove the `want_composite` parameter.
- o Add a short mention of proof-of-possession style tokens to the introduction and remove the respective open issue.

-05

- o Defined the JWT claim "cid" to express the OAuth 2.0 client identifier of the client that requested the token.
- o Defined and requested registration for "act" and "may_act" as Token introspection response parameters (in addition to being JWT claims).
- o Loosen up the language about `refresh_token` in the response to OPTIONAL from NOT RECOMMENDED based on feedback from real world deployment experience.
- o Add clarifying text about the distinction between JWT and access token URIs.
- o Close out (remove) some of the Open Issues bullets that have been resolved.

-04

- o Clarified that the "resource" and "audience" request parameters can be used at the same time (via <http://www.ietf.org/mail-archive/web/oauth/current/msg15335.html>).
- o Clarified subject/actor token validity after token exchange and explained a bit more about the recommendation to not issue refresh tokens (via <http://www.ietf.org/mail-archive/web/oauth/current/msg15318.html>).
- o Updated the examples appendix to use an issuer value that doesn't imply that the client issued and signed the tokens and used "Bearer" and "urn:ietf:params:oauth:token-type:access_token" in one of the responses (via <http://www.ietf.org/mail-archive/web/oauth/current/msg15335.html>).
- o Defined and registered urn:ietf:params:oauth:token-type:id_token, since some use cases perform token exchanges for ID Tokens and no URI to indicate that a token is an ID Token had previously been defined.

-03

- o Updated the document editors (adding Campbell, Bradley, and Mortimore).
- o Added to the title.
- o Added to the abstract and introduction.
- o Updated the format of the request to use application/x-www-form-urlencoded request parameters and the response to use the existing token endpoint JSON parameters defined in OAuth 2.0.
- o Changed the grant type identifier to urn:ietf:params:oauth:grant-type:token-exchange.
- o Added RFC 6755 registration requests for urn:ietf:params:oauth:token-type:refresh_token, urn:ietf:params:oauth:token-type:access_token, and urn:ietf:params:oauth:grant-type:token-exchange.
- o Added RFC 6749 registration requests for request/response parameters.
- o Removed the Implementation Considerations and the requirement to support JWTs.
- o Clarified many aspects of the text.
- o Changed "on_behalf_of" to "subject_token", "on_behalf_of_token_type" to "subject_token_type", "act_as" to "actor_token", and "act_as_token_type" to "actor_token_type".
- o Added an "audience" request parameter used to indicate the logical names of the target services at which the client intends to use the requested security token.
- o Added a "want_composite" request parameter used to indicate the desire for a composite token rather than trying to infer it from the presence/absence of token(s) in the request.

- o Added a "resource" request parameter used to indicate the URLs of resources at which the client intends to use the requested security token.
- o Specified that multiple "audience" and "resource" request parameter values may be used.
- o Defined the JWT claim "act" (actor) to express the current actor or delegation principal.
- o Defined the JWT claim "may_act" to express that one party is authorized to act on behalf of another party.
- o Defined the JWT claim "scp" (scopes) to express OAuth 2.0 scope-token values.
- o Added the "N_A" (not applicable) OAuth Access Token Type definition for use in contexts in which the token exchange syntax requires a "token_type" value, but in which the token being issued is not an access token.
- o Added examples.

-02

- o Enabled use of Security Token types other than JWTs for "act_as" and "on_behalf_of" request values.
- o Referenced the JWT and OAuth Assertions RFCs.

-01

- o Updated references.

-00

- o Created initial working group draft from draft-jones-oauth-token-exchange-01.

Authors' Addresses

Michael B. Jones
Microsoft

Email: mbj@microsoft.com
URI: <http://self-issued.info/>

Anthony Nadalin
Microsoft

Email: tonynad@microsoft.com

Brian Campbell (editor)
Ping Identity

Email: brian.d.campbell@gmail.com

John Bradley
Yubico

Email: ve7jtb@ve7jtb.com

Chuck Mortimore
Salesforce

Email: cmortimore@salesforce.com

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: July 23, 2017

M. Jones
Microsoft
P. Hunt
Oracle
January 19, 2017

OAuth 2.0 Protected Resource Metadata
draft-jones-oauth-resource-metadata-01

Abstract

This specification defines a metadata format that an OAuth 2.0 client can use to obtain the information needed to interact with an OAuth 2.0 protected resource.

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 July 23, 2017.

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 (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Requirements Notation and Conventions	3
1.2. Terminology	3
2. Protected Resource Metadata	4
2.1. Signed Protected Resource Metadata	5
3. Obtaining Protected Resource Metadata	6
3.1. Protected Resource Metadata Request	6
3.2. Protected Resource Metadata Response	7
3.3. Protected Resource Metadata Validation	8
4. Authorization Server Metadata	8
5. String Operations	8
6. Security Considerations	9
6.1. TLS Requirements	9
6.2. Impersonation Attacks	9
6.3. Publishing Metadata in a Standard Format	10
6.4. Authorization Servers	10
7. IANA Considerations	11
7.1. OAuth Protected Resource Metadata Registry	11
7.1.1. Registration Template	12
7.1.2. Initial Registry Contents	12
7.2. OAuth Authorization Server Metadata Registry	14
7.2.1. Registry Contents	14
7.3. Well-Known URI Registry	14
7.3.1. Registry Contents	14
8. References	14
8.1. Normative References	14
8.2. Informative References	16
Appendix A. Acknowledgements	17
Appendix B. Document History	17
Authors' Addresses	17

1. Introduction

This specification defines a metadata format enabling OAuth 2.0 clients to obtain information needed to interact with an OAuth 2.0 protected resource. This specification is intentionally as parallel as possible to "OAuth 2.0 Dynamic Client Registration Protocol" [RFC7591], which enables a client to provide metadata about itself to an OAuth 2.0 authorization server and to OAuth 2.0 Authorization Server Metadata [OAuth.AuthorizationMetadata], which enables a client to obtain metadata about an OAuth 2.0 authorization server.

The metadata for a protected resource is retrieved from a well-known location as a JSON [RFC7159] document, which declares information about its capabilities and optionally, its relationships to other services. This process is described in Section 3.

This metadata can either be communicated in a self-asserted fashion or as a set of signed metadata values represented as claims in a JSON Web Token (JWT) [JWT]. In the JWT case, the issuer is vouching for the validity of the data about the protected resource. This is analogous to the role that the Software Statement plays in OAuth Dynamic Client Registration [RFC7591].

Each protected resource publishing metadata about itself makes its own metadata document available at a well-known location rooted at the protect resource's URL, even when the resource server implements multiple protected resources. This prevents attackers from publishing metadata supposedly describing the protected resource, but that is not actually authoritative for the protected resource, as described in Section 6.2.

The means by which the client obtains the location of the protected resource metadata document is out of scope. In some cases, the location may be manually configured into the client. In other cases, it may be dynamically discovered, for instance, through the use of WebFinger [RFC7033], in a manner related to the description in Section 2 of "OpenID Connect Discovery 1.0" [OpenID.Discovery].

1.1. Requirements Notation and 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 RFC 2119 [RFC2119].

All uses of JSON Web Signature (JWS) [JWS] and JSON Web Encryption (JWE) [JWE] data structures in this specification utilize the JWS Compact Serialization or the JWE Compact Serialization; the JWS JSON Serialization and the JWE JSON Serialization are not used.

1.2. Terminology

This specification uses the terms "Access Token", "Authorization Code", "Authorization Endpoint", "Authorization Grant", "Authorization Server", "Client", "Client Authentication", "Client Identifier", "Client Secret", "Grant Type", "Protected Resource", "Redirection URI", "Refresh Token", "Resource Owner", "Resource Server", "Response Type", and "Token Endpoint" defined by OAuth 2.0 [RFC6749], the terms "Claim Name", "Claim Value", and "JSON Web Token (JWT)" defined by JSON Web Token (JWT) [JWT].

2. Protected Resource Metadata

Protected resources can have metadata describing their configuration. The following protected resource metadata values are used by this specification and are registered in the IANA "OAuth Protected Resource Metadata" registry established in Section 7.1:

resource

REQUIRED. The protected resource's resource identifier, which is a URL that uses the "https" scheme and has no fragment components. This is the location where ".well-known" RFC 5785 [RFC5785] resources containing information about the protected resource are published. Using these well-known resources is described in Section 3.

authorization_servers

OPTIONAL. JSON array containing a list of OAuth authorization server issuer identifiers, as defined in [OAuth.AuthorizationMetadata], for authorization servers that can be used with this protected resource. Protected resources MAY choose not to advertise some supported authorization servers even when this parameter is used. In some use cases, the set of authorization servers will not be enumerable, in which case this metadata parameter would not be used.

jwks_uri

OPTIONAL. URL of the protected resource's JWK Set [JWK] document. This contains keys belonging to the protected resource. For instance, this JWK Set MAY contain encryption key(s) that are used to encrypt access tokens to the protected resource. When both signing and encryption keys are made available, a "use" (public key use) parameter value is REQUIRED for all keys in the referenced JWK Set to indicate each key's intended usage.

scopes_provided

RECOMMENDED. JSON array containing a list of the OAuth 2.0 [RFC6749] "scope" values that are used in authorization requests to request access to this protected resource. Protected resources MAY choose not to advertise some scope values provided even when this parameter is used.

bearer_methods_supported

OPTIONAL. JSON array containing a list of the OAuth 2.0 Bearer Token [RFC6750] presentation methods that this protected resource supports. Defined values are ["header", "fragment", "query"], corresponding to Sections 2.1, 2.2, and 2.3 of RFC 6750.

resource_signing_alg_values_supported

OPTIONAL. JSON array containing a list of the JWS [JWS] signing algorithms ("alg" values) [JWA] supported by the protected resource for signed content. The value "none" MAY be included.

resource_encryption_alg_values_supported

OPTIONAL. JSON array containing a list of the JWE [JWE] encryption algorithms ("alg" values) [JWA] supported by the protected resource for encrypted content.

resource_encryption_enc_values_supported

OPTIONAL. JSON array containing a list of the JWE encryption algorithms ("enc" values) [JWA] supported by the protected resource for encrypted content.

resource_documentation

OPTIONAL. URL of a page containing human-readable information that developers might want or need to know when using the protected resource

resource_policy_uri

OPTIONAL. URL that the protected resource provides to read about the protected resource's requirements on how the client can use the data provided by the protected resource

resource_tos_uri

OPTIONAL. URL that the protected resource provides to read about the protected resource's terms of service

Additional protected resource metadata parameters MAY also be used.

2.1. Signed Protected Resource Metadata

In addition to JSON elements, metadata values MAY also be provided as a "signed_metadata" value, which is a JSON Web Token (JWT) [JWT] that asserts metadata values about the protected resource as a bundle. A set of claims that can be used in signed metadata are defined in Section 2. The signed metadata MUST be digitally signed or MACed using JSON Web Signature (JWS) [JWS] and MUST contain an "iss" (issuer) claim denoting the party attesting to the claims in the signed metadata. Consumers of the metadata MAY ignore the signed metadata if they do not support this feature. If the consumer of the metadata supports signed metadata, metadata values conveyed in the signed metadata MUST take precedence over those conveyed using plain JSON elements.

Signed metadata is included in the protected resource metadata JSON object using this OPTIONAL member:

signed_metadata

A JWT containing metadata values about the protected resource as claims. This is a string value consisting of the entire signed JWT. A "signed_metadata" metadata value SHOULD NOT appear as a claim in the JWT.

3. Obtaining Protected Resource Metadata

Protected resources supporting metadata MUST make a JSON document containing metadata as specified in Section 2 available at a path formed by concatenating a well-known URI string such as `"/.well-known/oauth-protected-resource"` to the protected resource's resource identifier. The syntax and semantics of `"/.well-known"` are defined in RFC 5785 [RFC5785]. The well-known URI path suffix used MUST be registered in the IANA "Well-Known URIs" registry [IANA.well-known].

Different applications utilizing OAuth protected resources in application-specific ways may define and register different well-known URI path suffixes used to publish protected resource metadata as used by those applications. For instance, if the Example application uses an OAuth protected resource in an Example-specific way, and there are Example-specific metadata values that it needs to publish, then it might register and use the `"example-resource-configuration"` URI path suffix and publish the metadata document at the path formed by concatenating `"/.well-known/example-resource-configuration"` to the protected resource's resource identifier.

An OAuth 2.0 application using this specification MUST specify what well-known URI string it will use for this purpose. The same protected resource MAY choose to publish its metadata at multiple well-known locations relative to its resource identifier, for example, publishing metadata at both `"/.well-known/example-resource-configuration"` and `"/.well-known/oauth-protected-resource"`.

3.1. Protected Resource Metadata Request

A protected resource metadata document MUST be queried using an HTTP "GET" request at the previously specified path.

The consumer of the metadata would make the following request when the resource identifier is `"https://resource.example.com"` and the well-known URI path suffix is `"oauth-protected-resource"` to obtain the metadata, since the resource identifier contains no path component:

```
GET /.well-known/oauth-protected-resource HTTP/1.1
Host: resource.example.com
```

If the resource identifier value contains a path component, any terminating "/" MUST be removed before appending "/.well-known/" and the well-known URI path suffix. The consumer of the metadata would make the following request when the resource identifier is "https://resource.example.com/resource1" and the well-known URI path suffix is "oauth-protected-resource" to obtain the metadata, since the resource identifier contains a path component:

```
GET /resource1/.well-known/oauth-protected-resource HTTP/1.1
Host: resource.example.com
```

Using path components enables supporting multiple resources per host. This is required in some multi-tenant hosting configurations. This use of ".well-known" is for supporting multiple resources per host; unlike its use in RFC 5785 [RFC5785], it does not provide general information about the host.

3.2. Protected Resource Metadata Response

The response is a set of claims about the protected resource's configuration. A successful response MUST use the 200 OK HTTP status code and return a JSON object using the "application/json" content type that contains a set of claims as its members that are a subset of the metadata values defined in Section 2. Other claims MAY also be returned.

Claims that return multiple values are represented as JSON arrays. Claims with zero elements MUST be omitted from the response.

An error response uses the applicable HTTP status code value.

The following is a non-normative example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "resource":
    "https://resource.example.com",
  "authorization_servers":
    [ "https://as1.example.com",
      "https://as2.example.net" ],
  "bearer_methods_supported":
    [ "header", "body" ],
  "resource_documentation":
    "http://resource.example.com/resource_documentation.html"
}
```


3.3. Protected Resource Metadata Validation

The "resource" value returned MUST be identical to the protected resource's resource identifier value that was concatenated with the well-known URI path suffix to create the URL used to retrieve the metadata. If these values are not identical, the data contained in the response MUST NOT be used.

4. Authorization Server Metadata

To support use cases in which the set of legitimate protected resources to use with the authorization server is fixed and enumerable, this specification defines the "protected_resources" metadata value, which enables explicitly listing them. Note that if the set of legitimate authorization servers to use with a protected resource is also fixed and enumerable, lists in the authorization server metadata and protected resource metadata should be cross-checked against one another for consistency when these lists are used by the application profile.

The following authorization server metadata value is defined by this specification and is registered in the IANA "OAuth Authorization Server Metadata" registry established in OAuth 2.0 Authorization Server Metadata [OAuth.AuthorizationMetadata].

protected_resources

OPTIONAL. JSON array containing a list of resource identifiers for OAuth protected resources for protected resources that can be used with this authorization server. Authorization servers MAY choose not to advertise some supported protected resources even when this parameter is used. In some use cases, the set of protected resources will not be enumerable, in which case this metadata parameter would not be used.

5. String Operations

Processing some OAuth 2.0 messages requires comparing values in the messages to known values. For example, the member names in the metadata response might be compared to specific member names such as "resource". Comparing Unicode [UNICODE] strings, however, has significant security implications.

Therefore, comparisons between JSON strings and other Unicode strings MUST be performed as specified below:

1. Remove any JSON applied escaping to produce an array of Unicode code points.

2. Unicode Normalization [USA15] MUST NOT be applied at any point to either the JSON string or to the string it is to be compared against.
3. Comparisons between the two strings MUST be performed as a Unicode code point to code point equality comparison.

6. Security Considerations

6.1. TLS Requirements

Implementations MUST support TLS. Which version(s) ought to be implemented will vary over time, and depend on the widespread deployment and known security vulnerabilities at the time of implementation. The protected resource MUST support TLS version 1.2 [RFC5246] and MAY support additional transport-layer security mechanisms meeting its security requirements. When using TLS, the client MUST perform a TLS/SSL server certificate check, per RFC 6125 [RFC6125]. Implementation security considerations can be found in Recommendations for Secure Use of TLS and DTLS [BCP195].

To protect against information disclosure and tampering, confidentiality protection MUST be applied using TLS with a ciphersuite that provides confidentiality and integrity protection.

6.2. Impersonation Attacks

TLS certificate checking MUST be performed by the client, as described in Section 6.1, when making a protected resource metadata request. Checking that the server certificate is valid for the resource identifier URL prevents man-in-middle and DNS-based attacks. These attacks could cause a client to be tricked into using an attacker's resource server, which would enable impersonation of the legitimate protected resource. If an attacker can accomplish this, they can access the resources that the affected client has access to using the protected resource that they are impersonating.

An attacker may also attempt to impersonate a protected resource by publishing a metadata document that contains a "resource" claim using the resource identifier URL of the protected resource being impersonated, but containing information of the attacker's choosing. This would enable it to impersonate that protected resource, if accepted by the client. To prevent this, the client MUST ensure that the resource identifier URL it is using as the prefix for the metadata request exactly matches the value of the "resource" metadata value in the protected resource metadata document received by the client.

6.3. Publishing Metadata in a Standard Format

Publishing information about the protected resource in a standard format makes it easier for both legitimate clients and attackers to use the protected resource. Whether a protected resource publishes its metadata in an ad-hoc manner or in the standard format defined by this specification, the same defenses against attacks that might be mounted that use this information should be applied.

6.4. Authorization Servers

Secure determination of appropriate authorization servers to use with a protected resource for all use cases is out of scope of this specification. This specification assumes that the client has a means of determining appropriate authorization servers to use with a protected resource and that the client is using the correct metadata for each protected resource. Implementers need to be aware that if an inappropriate authorization server is used by the client, that an attacker may be able to act as a man-in-the-middle proxy to a valid authorization server without it being detected by the authorization server or the client.

The ways to determine the appropriate authorization servers to use with a protected resource are in general, application-dependent. For instance, some protected resources are used with a fixed authorization server or set of authorization servers, the locations of which may be well known, or which could be published as metadata values by the protected resource. In other cases, the set of authorization servers that can be used with a protected resource can be dynamically changed by administrative actions or by changes to the set of authorization servers adhering to a trust framework. Many other means of determining appropriate associations between protected resources and authorization servers are also possible.

To support use cases in which the set of legitimate authorization servers to use with the protected resource is fixed and enumerable, this specification defines the "authorization_servers" metadata value, which enables explicitly listing them. Note that if the set of legitimate protected resources to use with an authorization server is also fixed and enumerable, lists in the protected resource metadata and authorization server metadata should be cross-checked against one another for consistency when these lists are used by the application profile.

7. IANA Considerations

The following registration procedure is used for the registry established by this specification.

Values are registered on a Specification Required [RFC5226] basis after a two-week review period on the `oauth-ext-review@ietf.org` mailing list, on the advice of one or more Designated Experts. However, to allow for the allocation of values prior to publication, the Designated Experts may approve registration once they are satisfied that such a specification will be published.

Registration requests sent to the mailing list for review should use an appropriate subject (e.g., "Request to register OAuth Protected Resource Metadata: example").

Within the review period, the Designated Experts will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful. Registration requests that are undetermined for a period longer than 21 days can be brought to the IESG's attention (using the `iesg@ietf.org` mailing list) for resolution.

Criteria that should be applied by the Designated Experts includes determining whether the proposed registration duplicates existing functionality, determining whether it is likely to be of general applicability or whether it is useful only for a single application, and whether the registration makes sense.

IANA must only accept registry updates from the Designated Experts and should direct all requests for registration to the review mailing list.

It is suggested that multiple Designated Experts be appointed who are able to represent the perspectives of different applications using this specification, in order to enable broadly-informed review of registration decisions. In cases where a registration decision could be perceived as creating a conflict of interest for a particular Expert, that Expert should defer to the judgment of the other Experts.

7.1. OAuth Protected Resource Metadata Registry

This specification establishes the IANA "OAuth Protected Resource Metadata" registry for OAuth 2.0 protected resource metadata names. The registry records the protected resource metadata member and a reference to the specification that defines it.

7.1.1.1. Registration Template

Metadata Name:

The name requested (e.g., "resource"). This name is case-sensitive. Names may not match other registered names in a case-insensitive manner unless the Designated Experts state that there is a compelling reason to allow an exception.

Metadata Description:

Brief description of the metadata (e.g., "Resource identifier URL").

Change Controller:

For Standards Track RFCs, list the "IESG". For others, give the name of the responsible party. Other details (e.g., postal address, email address, home page URI) may also be included.

Specification Document(s):

Reference to the document or documents that specify the parameter, preferably including URIs that can be used to retrieve copies of the documents. An indication of the relevant sections may also be included but is not required.

7.1.1.2. Initial Registry Contents

- o Metadata Name: "resource"
- o Metadata Description: Protected resource's resource identifier URL
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "authorization_servers"
- o Metadata Description: JSON array containing a list of OAuth authorization server issuer identifiers
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "jwks_uri"
- o Metadata Description: URL of the protected resource's JWK Set document
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "scopes_provided"
- o Metadata Description: JSON array containing a list of the OAuth 2.0 "scope" values that are used in authorization requests to request access this protected resource
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "bearer_methods_supported"
- o Metadata Description: JSON array containing a list of the OAuth 2.0 Bearer Token presentation methods that this protected resource supports
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "resource_signing_alg_values_supported"
- o Metadata Description: JSON array containing a list of the JWS signing algorithms ("alg" values) supported by the protected resource for signed content
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "resource_encryption_alg_values_supported"
- o Metadata Description: JSON array containing a list of the JWE encryption algorithms ("alg" values) supported by the protected resource for encrypted content
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "resource_encryption_enc_values_supported"
- o Metadata Description: JSON array containing a list of the JWE encryption algorithms ("enc" values) supported by the protected resource for encrypted content
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "resource_documentation"
- o Metadata Description: URL of a page containing human-readable information that developers might want or need to know when using the protected resource
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "resource_policy_uri"
- o Metadata Description: URL that the protected resource provides to read about the protected resource's requirements on how the client can use the data provided by the protected resource
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "resource_tos_uri"
- o Metadata Description: URL that the protected resource provides to read about the protected resource's terms of service
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

7.2. OAuth Authorization Server Metadata Registry

The following authorization server metadata value is registered in the IANA "OAuth Authorization Server Metadata" registry established in OAuth 2.0 Authorization Server Metadata [OAuth.AuthorizationMetadata].

7.2.1. Registry Contents

- o Metadata Name: "protected_resources"
- o Metadata Description: JSON array containing a list of resource identifiers for OAuth protected resources
- o Change Controller: IESG
- o Specification Document(s): Section 4 of [[this specification]]

7.3. Well-Known URI Registry

This specification registers the well-known URI defined in Section 3 in the IANA "Well-Known URIs" registry [IANA.well-known] established by RFC 5785 [RFC5785].

7.3.1. Registry Contents

- o URI suffix: "oauth-protected-resource"
- o Change controller: IESG
- o Specification document: Section 3 of [[this specification]]
- o Related information: (none)

8. References

8.1. Normative References

- [BCP195] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <<http://www.rfc-editor.org/info/bcp195>>.
- [JWA] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <<http://tools.ietf.org/html/rfc7518>>.
- [JWE] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <<http://tools.ietf.org/html/rfc7516>>.

- [JWK] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <<http://tools.ietf.org/html/rfc7517>>.
- [JWS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <<http://tools.ietf.org/html/rfc7515>>.
- [JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <<http://tools.ietf.org/html/rfc7519>>.
- [OAuth.AuthorizationMetadata] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", draft-ietf-oauth-discovery-05 (work in progress), January 2017, <<http://tools.ietf.org/html/draft-ietf-oauth-discovery-05>>.
- [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>>.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <<http://www.rfc-editor.org/info/rfc5226>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<http://www.rfc-editor.org/info/rfc5246>>.
- [RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, DOI 10.17487/RFC5785, April 2010, <<http://www.rfc-editor.org/info/rfc5785>>.
- [RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <<http://www.rfc-editor.org/info/rfc6125>>.

- [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>>.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <<http://www.rfc-editor.org/info/rfc6750>>.
- [RFC7033] Jones, P., Salgueiro, G., Jones, M., and J. Smarr, "WebFinger", RFC 7033, DOI 10.17487/RFC7033, September 2013, <<http://www.rfc-editor.org/info/rfc7033>>.
- [RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <<http://www.rfc-editor.org/info/rfc7159>>.
- [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, <<http://www.rfc-editor.org/info/rfc7591>>.
- [UNICODE] The Unicode Consortium, "The Unicode Standard", <<http://www.unicode.org/versions/latest/>>.
- [USA15] Davis, M. and K. Whistler, "Unicode Normalization Forms", Unicode Standard Annex 15, June 2015, <<http://www.unicode.org/reports/tr15/>>.

8.2. Informative References

- [I-D.ietf-oauth-mix-up-mitigation]
Jones, M., Bradley, J., and N. Sakimura, "OAuth 2.0 Mix-Up Mitigation", draft-ietf-oauth-mix-up-mitigation-01 (work in progress), July 2016.
- [IANA.well-known]
IANA, "Well-Known URIs", <<http://www.iana.org/assignments/well-known-uris>>.
- [OpenID.Discovery]
Sakimura, N., Bradley, J., Jones, M., and E. Jay, "OpenID Connect Discovery 1.0", November 2014, <http://openid.net/specs/openid-connect-discovery-1_0.html>.

Appendix A. Acknowledgements

Thanks to George Fletcher and Tony Nadalin for their input on the specification.

Appendix B. Document History

[[to be removed by the RFC Editor before publication as an RFC]]

-01

- o Moved the "protected_resources" authorization server metadata element here, removing it from draft-ietf-oauth-discovery.

-00

- o Created the initial version. This draft reuses some text from draft-ietf-oauth-discovery-03.

Authors' Addresses

Michael B. Jones
Microsoft

Email: mbj@microsoft.com
URI: <http://self-issued.info/>

Phil Hunt
Oracle

Email: phil.hunt@yahoo.com

Open Authentication Protocol
Internet-Draft
Intended status: Informational
Expires: May 16, 2017

T. Lodderstedt, Ed.
Deutsche Telekom AG
J. Bradley
Ping Identity
A. Labunets
Facebook
November 12, 2016

OAuth Security Topics
draft-lodderstedt-oauth-security-topics-00

Abstract

This draft gives a comprehensive overview on open OAuth security topics. It is intended to serve as a tool for the OAuth working group to systematically address these open security topics, recommending mitigations, and potentially also defining OAuth extensions needed to cope with the respective security threats. This draft will potentially become a BCP over time.

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 16, 2017.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect

to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. OAuth Credentials Leakage	3
2.1. Redirect URI validation of authorization requests	3
2.1.1. Authorization Code Grant	3
2.1.2. Implicit Grant	4
2.1.3. Countermeasure: exact redirect URI matching	6
2.2. Authorization code leakage via referrer headers	7
2.2.1. Countermeasures	7
2.3. Code in browser history (TBD)	8
2.4. Access token in browser history (TBD)	8
2.5. Access token on bad resource servers (TBD)	8
2.6. Mix-Up (TBD)	9
3. OAuth Credentials Injection	9
3.1. Code Injection	9
3.1.1. Proposed Counter Measures	11
3.1.2. Access Token Injection (TBD)	13
3.1.3. XSRF (TBD)	13
4. Other Attacks	14
5. Acknowledgements	14
6. IANA Considerations	14
7. Security Considerations	14
8. Normative References	14
Authors' Addresses	15

1. Introduction

It's been a while since OAuth has been published in RFC 6749 [RFC6749] and RFC 6750 [RFC6750]. Since publication, OAuth 2.0 has gotten massive traction in the market and became the standard for API protection and, as foundation of OpenID Connect, identity providing.

- o OAuth implementations are being attacked through known implementation weaknesses and anti-patterns (XSRF, referrer header). Although most of these threats are discussed in RFC 6819 [RFC6819], continued exploitation demonstrates there may be a need for more specific recommendations or that the existing mitigations are too difficult to deploy.
- o Technology has changed, e.g. the way browsers treat fragments in some situations, which may change the implicit grant's underlying security model.

- o OAuth is used in much more dynamic setups than originally anticipated, creating new challenges with respect to security. Those challenges go beyond the original scope of both RFC 6749 [RFC6749] and RFC 6819 [RFC6819].

This remainder of the document is organized as follows: The next section describes various scenarios how OAuth credentials (namely access tokens and authorization codes) may be disclosed to attackers and proposes countermeasures. Afterwards, the document discusses attacks possible with captured credential and how they may be prevented. The last sections discuss additional threats.

2. OAuth Credentials Leakage

2.1. Redirect URI validation of authorization requests

The following implementation issue has been observed: Some authorization servers allow clients to register redirect URI patterns instead of complete redirect URIs. In those cases, the authorization servers, at runtime, match the actual redirect URI parameter value at the authorization endpoint against this pattern. This approach allows clients to encode transaction state into additional redirect URI parameters or to register just a single pattern for multiple redirect URIs. As a downside, it turned out to be more complex to implement and error prone to manage than exact redirect URI matching. Several successful attacks have been observed in the wild, which utilized flaws in the pattern matching implementation or concrete configurations. Such a flaw effectively breaks client identification or authentication (depending on grant and client type) and allows the attacker to obtain an authorization code or access token, either

- o by directly sending the user agent to a URI under the attackers control or
- o usually via the client as open redirector in conjunction with fragment handling (implicit grant) carrying the response including the respective OAuth credentials.

2.1.1. Authorization Code Grant

For a public client using the grant type code, an attack would look as follows:

Let's assume the pattern "https://*.example.com/*" had been registered for the client "s6BhdRkqt3". This pattern allows redirect URI from any host residing in the domain example.com. So if an attacker manages to establish a host or subdomain in "example.com" he

can impersonate the legitimate client. Assume the attacker sets up the host "evil.example.com".

- (1) The attacker needs to trick the user into opening a tampered URL in his browser, which launches a page under the attacker's control, say "https://www.evil.com".
- (2) This URL initiates an authorization request with the client id of a legitimate client to the authorization endpoint. This is the example authorization request (line breaks are for display purposes only):

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=xyz
&redirect_uri=https%3A%2F%2Fevil.client.example.com%2Fcb HTTP/1.1
Host: server.example.com
```

- (3) The authorization validates the redirect URI in order to identify the client. Since the pattern allows arbitrary domains host names in "example.com", the authorization request is processed under the legitimate client's identity. This includes the way the request for user consent is presented to the user. If auto-approval is allowed (which is not recommended for public clients according to RFC 6749), the attack can be performed even easier.
- (4) If the user does not recognize the attack, the code is issued and directly sent to the attacker's client.
- (5) Since the attacker impersonated a public client, it can directly exchange the code for tokens at the respective token endpoint.

Note: This attack will not work for confidential clients, since the code exchange requires authentication with the legitimate client's secret. The attacker will need to utilize the legitimate client to redeem the code. This and other kinds of injections are covered in Section OAuth Credentials Injection.

2.1.2. Implicit Grant

The attack described above for grant type authorization code works similarly for the implicit grant. If the attacker is able to send the authorization response to a URI under his control, he will directly get access to the fragment carrying the access token.

Additionally, it is possible to conduct an attack utilizing the way user agents treat fragments in case of redirects. User agents re-attach fragments to the destination URL of a redirect if the location header does not contain a fragment (see [RFC7231], section 9.5). In

this attack this behavior is combined with the client as an open redirector in order to get access to access tokens. This allows circumvention even of strict redirect URI patterns.

Assume the pattern for client "s6BhdRkqt3" is "https://client.example.com/cb?*". i.e. any parameter is allowed for redirects to "https://client.example.com/cb". Unfortunately, the client exposes an open redirector. This endpoint supports a parameter "redirect_to", which takes a target URL and will send the browser to this URL using a HTTP 302.

- (1) Same as above, the attacker needs to trick the user into opening a tampered URL in his browser, which launches a page under the attacker's control, say "https://www.evil.com".
- (2) The URL initiates an authorization request, which is very similar to the attack on the code flow. As differences, it utilizes the open redirector by encoding "redirect_to=https://client.evil.com" into the redirect URI and it uses the response type "token" (linebreaks are for display purposes only):

```
GET /authorize?response_type=token&client_id=s6BhdRkqt3&state=xyz
  &redirect_uri=https%3A%2F%2Fclient.example.com%2Fcb%26redirect_to
    %253Dhttps%253A%252F%252Fclient.evil.com%252Fcb HTTP/1.1
Host: server.example.com
```

- (3) Since the redirect URI matches the registered pattern, the authorization server allows the request and sends the resulting access token with a 302 redirect (some response parameters are omitted for better readability)

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?
redirect_to=%3Dhttps%3A%2F%2Fclient.evil.com%2Fcb
#access_token=2YotnFZFEjrlzCsicMWpAA&...
```

- (4) At the example.com, the request arrives at the open redirector. It will read the redirect parameter and will issue a HTTP 302 to the URL "https://evil.example.com/cb".

```
HTTP/1.1 302 Found
Location: https://client.evil.com/cb
```

- (5) Since the redirector at example.com does not include a fragment in the Location header, the user agent will re-attach the original fragment

"#access_token=2YotnFZFEjrlzCsicMWpAA&..." to the URL and will navigate to the following URL:

`https://client.evil.com/cb#access_token=2YotnFZFEjrlzCsicMWpAA&...`

- (6) The attacker's page at `client.evil.com` can access the fragment and obtain the access token.

2.1.3. Countermeasure: exact redirect URI matching

Since the cause of the implementation and management issues is the complexity of the pattern matching, this document proposes to recommend general use of exact redirect URI matching instead, i.e. the authorization server shall compare the two URIs using simple string comparison as defined in [RFC3986], Section 6.2.1..

This would cause the following impacts:

- o This change will require all OAuth clients to maintain the transaction state (and XSRF tokens) in the "state" parameter. This is a normative change to RFC 6749 since section 3.1.2.2 allows for dynamic URI query parameters in the redirect URI. In order to assess the practical impact, the working group needs to collect data whether this feature is used in deployed reality today.
- o The working group might also consider this change as a step towards improved interoperability for OAuth implementations since RFC 6749 is somehow vague on redirect URI validation. There is especially no rule for pattern matching. So one may assume all clients utilizing pattern matching will do so in a deployment specific way. On the other hand, RFC 6749 already recommends exact matching if the full URL had been registered.
- o Clients with multiple redirect URIs need to register all of them explicitly.
Note: clients with just a single redirect URI would not even need to send a redirect URI with the authorization request. Does it make sense to emphasize this option? Would that further simplify use of the protocol?
- o Exact redirect matching does not work for native apps utilizing a local web server due to dynamic port numbers - at least wild cards for port numbers are required.
Note: Does redirect uri validation solve any problem for native apps? Effective against impersonation when used in conjunction with claimed HTTPS redirect URIs only.

Additional recommendations:

- o It is also advisable that the domains on which callbacks are hosted should not expose open redirectors (see respective section).
- o As a further recommendation, clients may drop fragments via intermediary URL with fix fragment (e.g. <https://developers.facebook.com/blog/post/552/>) to prevent the user agent from appending any unintended fragments.

Alternatives to exact redirect URI matching: authenticate client using digital signatures (JAR? <https://tools.ietf.org/html/draft-ietf-oauth-jwsreq-09>), ...

2.2. Authorization code leakage via referrer headers

The section above already discussed use of the referrer header for one kind of attack to obtain OAuth credentials. It is also possible authorization codes are unintentionally disclosed to attackers, if a OAuth client renders a page containing links to other pages (ads, faq, ...) as result of a successful authorization request.

If the user clicks onto one of those links and the target is under the control of an attacker, it can get access to the response URL in the referrer header.

It is also possible that an attacker injects cross-domain content somehow into the page, such as `` (f.e. if this is blog web site etc.): the implication is obviously the same - loading this content by browser results in leaking referrer with a code.

2.2.1. Countermeasures

There are some means to prevent leakage as described above:

- o Use of the HTML link attribute `rel="noreferrer"` (Chrome 52.0.2743.116, FF 49.0.1, Edge 38.14393.0.0, IE/Win10)
- o Use of the "referrer" meta link attribute (possible values e.g. `noreferrer`, `origin`, ...) (cf. <https://w3c.github.io/webappsec-referrer-policy/> - work in progress (seems Google, Chrome and Edge support it))
- o Redirect to intermediate page (sanitize history) before sending user agent to other pages
Note: double check redirect/referrer header behavior

- o Use form post mode instead of redirect for authorization response

Note: There shouldn't be a referer header when loading HTTP content from a HTTPS -loaded page (e.g. help/faq pages)

Note: This kind of attack is not applicable to the implicit grant since fragments are not be included in referrer headers (cf. <https://tools.ietf.org/html/rfc7231#section-5.5.2>)

2.3. Code in browser history (TBD)

When browser navigates to "client.com/redirection_endpoint?code=abcd" as a result of a redirect from a provider's authorization endpoint.

Proposal for counter-measures: code is one time use, has limited duration, is bound to client id/secret (confidential clients only)

2.4. Access token in browser history (TBD)

When a client or just a web site which already has a token deliberately navigates to a page like provider.com/get_user_profile?access_token=abcdef.. Actually RFC6750 discourages this practice and asks to transfer tokens via a header, but in practice web sites often just pass access token in query

When browser navigates to client.com/redirection_endpoint#access_token=abcef as a result of a redirect from a provider's authorization endpoint.

Proposal: replace implicit flow with postmessage communication

2.5. Access token on bad resource servers (TBD)

In the beginning, the basic assumption of OAuth 2.0 was that the OAuth client is implemented for and tightly bound to a certain deployment. It therefore knows the URLs of the authorization and resource servers upfront, at development/deployment time. So well-known URLs to resource servers serve as trust anchor. The validation whether the client talks to a legitimate resource server is based on TLS server authentication (see [RFC6819], Section 4.5.4).

As OAuth clients nowadays more and more bind dynamically at runtime to authorization and resource servers, there need to be alternative solutions to ensure, client do not deliver access tokens to bad resource servers.

...

Potential mitigations:

- o PoP
- o Token Binding
- o AS-provided Meta Data
- o ...

2.6. Mix-Up (TBD)

Mix-up is another kind of attack on more dynamic OAuth scenarios (or at least scenarios where a OAuth client interacts with multiple authorization servers). The goal of the attack is to obtain an authorization code or an access token by tricking the client into sending those credentials to the attacker (which acts as MITM between client and authorization server)

A detailed description of the attack and potential counter-measures is given in cf. <https://tools.ietf.org/html/draft-ietf-oauth-mix-up-mitigation-01>.

Potential mitigations:

- o AS returns `client_id` and its iss in the response. Client compares this data to AS it believed it sent the user agent to.
- o ID token (so requires OpenID Connect) carries client id and issuer
- o register AS-specific redirect URIs, bind transaction to AS
- o ...

3. OAuth Credentials Injection

Credential injection means an attacker somehow obtained a valid OAuth credential (code or token) and is able to utilize this to impersonate the legitimate resource owner or to cause a victim to access resources under the attacker's control (XSRF).

3.1. Code Injection

In such an attack, the adversary attempts to inject a stolen authorization code into a legitimate client on a device under his control. In the simplest case, the attacker would want to use the code in his own client. But there are situations where this might not be possible or intended. Example are:

- o The code is bound to a particular confidential client and the attacker is unable to obtain the required client credentials to redeem the code himself and/or
- o The attacker wants to access certain functions in this particular client. As an example, the attacker potentially wants to impersonate his victim in a certain app.
- o Another example could be that access to the authorization and resource servers is somehow limited to networks, the attackers are unable to access directly.

How does an attack look like?

- (1) The attacker obtains an authorization code by executing any of the attacks described above (OAuth Credentials Leakage).
- (2) It performs an OAuth authorization process with the legitimate client on his device.
- (3) The attacker injects the stolen authorization code in the response of the authorization server to the legitimate client.
- (4) The client sends the code to the authorization server's token endpoint, along with client id, client secret and actual redirect_uri.
- (5) The authorization server checks the client secret, whether the code was issued to the particular client and whether the actual redirect URI matches the redirect_uri parameter.
- (6) If all checks succeed, the authorization server issues access and other tokens to the client.
- (7) The attacker just impersonated the victim.

Obviously, the check in step (5) will fail, if the code was issued to another client id, e.g. a client set up by the attacker.

An attempt to inject a code obtained via a malware pretending to be the legitimate client should also be detected, if the authorization server stored the complete redirect URI used in the authorization request and compares it with the redirect_uri parameter.

[RFC6749], Section 4.1.3, requires the AS to ... "ensure that the "redirect_uri" parameter is present if the "redirect_uri" parameter was included in the initial authorization request as described in Section 4.1.1, and if included ensure that their values are

identical." In the attack scenario described above, the legitimate client would use the correct redirect URI it always uses for authorization requests. But this URI would not match the tampered redirect URI used by the attacker (otherwise, the redirect would not land at the attackers page). So the authorization server would detect the attack and refuse to exchange the code.

Note: this check could also detect attempt to inject a code, which had been obtained from another instance of the same client on another device, if certain conditions are fulfilled:

- o the redirect URI itself needs to contain a nonce or another kind of one-time use, secret data and
- o the client has bound this data to this particular instance

But this approach conflicts with the idea to enforce exact redirect URI matching at the authorization endpoint. Moreover, it has been observed that providers very often ignore the `redirect_uri` check requirement at this stage, maybe, because it doesn't seem to be security-critical from reading the spec.

Other providers just pattern match the `redirect_uri` parameter against the registered redirect URI pattern. This saves the authorization server from storing the link between the actual redirect URI and the respective authorization code for every transaction. But this kind of check obviously does not fulfill the intent of the spec, since the tampered redirect URI is not considered. So any attempt to inject a code obtained using the `client_id` of a legitimate client or by utilizing the legitimate client on another device won't be detected in the respective deployments.

It is also assumed that the requirements defined in [RFC6749], Section 4.1.3, increase client implementation complexity as clients need to memorize or re-construct the correct redirect URI for the call to the tokens endpoint.

The authors therefore propose to the working group to drop this feature in favor of more effective and (hopefully) simpler approaches to code injection prevention as described in the following section.

3.1.1. Proposed Counter Measures

The general proposal is to bind every particular authorization code to a certain client on a certain device (or in a certain user agent) in the context of a certain transaction. There are multiple technical solutions to achieve this goal:

- Nonce OpenID Connect's existing "nonce" parameter is used for this purpose. The nonce value is one time use and created by the client. The client is supposed to bind it to the user agent session and sends it with the initial request to the OpenId Provider (OP). The OP associates the nonce to the authorization code and attests this binding in the ID token, which is issued as part of the code exchange at the token endpoint. If an attacker injected an authorization code in the authorization response, the nonce value in the client session and the nonce value in the ID token will not match and the attack is detected. assumption: attacker cannot get hold of the user agent state on the victims device, where he has stolen the respective authorization code.
- pro:
- existing feature, used in the wild
- con:
- OAuth does not have an ID Token - would need to push that down the stack
- State It has been discussed in the security workshop in December to use the OAuth state value much similar in the way as described above. In the case of the state value, the idea is to add a further parameter state to the code exchange request. The authorization server then compares the state value it associated with the code and the state value in the parameter. If those values do not match, it is considered an attack and the request fails. Note: a variant of this solution would be send a hash of the state (in order to prevent bulky requests and DoS).
- pro:
- use existing concept
- con:
- state needs to fulfil certain requirements (one time use, complexity)
 - new parameter means normative spec change
- PKCE Basically, the PKCE challenge/verifier could be used in the same way as Nonce or State. In contrast to its original intention, the verifier check would fail although the client uses its correct verifier but the code is associated with a challenge, which does not match.
- pro:
- existing and deployed OAuth feature
- con:
- currently used and recommended for native apps, not web apps

Token Binding Code must be bind to UA-AS and UA-Client legs -
requires further data (extension to response) to manifest
binding id for particular code.

pro:

- highly secure

con:

- highly sophisticated, requires browser support, will it
work for native apps?

per instance client id/secret ...

Note on pre-warmed secrets: An attacker can circumvent the counter-measures described above if he is able to create the respective secret on a device under his control, which is then used in the victim's authorization request.

Exact redirect URI matching of authorization requests can prevent the attacker from using the pre-warmed secret in the faked authorization transaction on the victim's device.

Unfortunately it does not work for all kinds of OAuth clients. It is effective for web and JS apps, for native apps with claimed URLs.

What about other native apps? Treat nonce or PKCE challenge as replay detection tokens (needs to ensure cluster-wide one-time use)?

3.1.2. Access Token Injection (TBD)

Note: An attacker in possession of an access token can access any resources the access token gives him the permission to. This kind of attacks simply illustrates the fact that bearer tokens utilized by OAuth are reusable similar to passwords unless they are protected by further means.

(where do we treat access token replay/use at the resource server?

<https://tools.ietf.org/html/rfc6819#section-4.6.4> has some text about it but is it sufficient?)

The attack described in this section is about injecting a stolen access token into a legitimate client on a device under the adversaries control. The attacker wants to impersonate a victim and cannot use his own client, since he wants to access certain functions in this particular client.

Proposal: token binding, hybrid flow+nonce(OIDC), other
cryptographical binding between access token and user agent instance

3.1.3. XSRF (TBD)

injection of code or access token on a victim's device (e.g. to cause client to access resources under the attacker's control)

mitigation: XSRF tokens (one time use) w/ user agent binding (cf.
[https://www.owasp.org/index.php/
CrossSite_Request_Forgery_\(CSRF\)_Prevention_Cheat_Sheet](https://www.owasp.org/index.php/CrossSite_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet))

4. Other Attacks

Using the AS as Open Redirector - error handling AS (redirects)
(draft-ietf-oauth-closing-redirectors-00)

Using the Client as Open Redirector

redirect via status code 307 - use 302

5. Acknowledgements

We would like to thank ... for their valuable feedback.

6. IANA Considerations

This draft includes no request to IANA.

7. Security Considerations

All relevant security considerations have been given in the
functional specification.

8. Normative References

- [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>>.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <<http://www.rfc-editor.org/info/rfc6750>>.
- [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>>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <<http://www.rfc-editor.org/info/rfc7231>>.

Authors' Addresses

Torsten Lodderstedt (editor)
Deutsche Telekom AG

Email: torsten@lodderstedt.net

John Bradley
Ping Identity

Email: ve7jtb@ve7jtb.com

Andrey Labunets
Facebook

Email: isciurus@fb.com